Compresión de datos
La referencia completa

7 de marzo de 2014

Cuarta Edición
Primera en español

- La traducción está completa.
- Se incluyen todos los apéndices, de A a I.
- En esta versión se han corregido numerosas erratas y consta de un índice alfabético.

Autor:

David Salomon

Professor David Salomon (emeritus)
Computer Science Department
California State University
Northridge, CA 91330-8281
USA
Email: david.salomon@csun.edu

Giovanni Motta y David Bryant

Colaboradores en la obra original, en inglés.

David Herrera Pérez.

La persona que ha traducido el libro al castellano
(Véase el prefacio a la edición en castellano)
Alba de Tormes (Salamanca)
ESPAÑA
Email: micifut@hotmail.com

ISBN del libro original en inglés

British Library Cataloguing in Publication Data A catalogue record for this book is available from the British Library
Library of Congress Control Number: 2006931789
ISBN-10: 1-84628-602-6 e-ISBN-10: 1-84628-603-4 ISBN-13: 978-1-84628-602-5 e-ISBN-13: 978-1-84628-603-2
Printed on acid-free paper.

9 8 7 6 5 4 3 2 1

Springer Science+Business Media, LLC
springer.com

Nota: *Se incluye esta página en la traducción para todos aquellos que deseen comprar el libro original, en inglés.*

Dedicatoria:

Para Wayne Wheeler, un editor por excelencia.

Escriba su propia historia.
No deje que otros la escriban por usted.

Consejo en una galleta de la fortuna china.

Contenidos

Prefacios y agradecimientos **XXXIII**
Prefacio a la edición en castellano . . . XXXIII
Prefacio a la cuarta edición . . . XL
Prefacio a la tercera edición XLIV
Prefacio a la segunda edición . . . XLVII
Prefacio a la primera edición . . . LI
Agradecimientos . . . LII

I Compresión de datos **1**

Introducción **3**

1 Técnicas básicas **19**
1.1 Compresión intuitiva . . . 19
1.1.1 Braille . . . 19
1.1.2 Compresión irreversible de texto . . . 21
1.1.3 Compresión de texto ad hoc . . . 21
1.2 Codificación run-length . . . 25
1.3 Compresión de texto RLE . . . 25
1.3.1 Codificación relativa . . . 29
1.4 Compresión de imágenes RLE . . . 30
1.4.1 Compresión de imágenes con pérdidas . . . 34
1.4.2 Imagen condicional RLE . . . 35
1.4.3 El formato BinHex 4.0 . . . 37
1.4.4 Archivos de imagen BMP . . . 38
1.5 Codificación mover–al–frente . . . 39
1.6 Cuantificación escalar . . . 43
1.7 Reducción recursiva del rango . . . 45

2 Métodos estadísticos **51**
2.1 Conceptos de teoría de la información . . . 52
2.1.1 Contenido de la información algorítmica . . . 57
2.2 Códigos de tamaño variable . . . 58
2.3 Códigos prefijo . . . 59
2.3.1 El código unario . . . 60
2.3.2 Otros códigos prefijo . . . 61
2.4 El código Tunstall . . . 66
2.5 El código Golomb . . . 67

2.6 La desigualdad de Kraft-MacMillan . . . 76
2.7 La codificación de Shannon-Fano . . . 77
2.8 Codificación Huffman . . . 79
2.8.1 Decodificación Huffman . . . 82
2.8.2 Tamaño medio del código . . . 84
2.8.3 Número de códigos . . . 86
2.8.4 Códigos Huffman ternarios . . . 87
2.8.5 La altura de un árbol de Huffman . . . 88
2.8.6 Códigos Huffman canónicos . . . 90
2.9 Codificación de Huffman adaptativa . . . 94
2.9.1 Códigos sin comprimir . . . 95
2.9.2 Modificación del Árbol . . . 96
2.9.3 Desbordamiento del contador . . . 98
2.9.4 Desbordamiento (overflow) del código . . . 98
2.9.5 Una variante . . . 99
2.9.6 Método de Vitter . . . 100
2.10 MNP5 . . . 100
2.10.1 Actualización de la tabla . . . 102
2.11 MNP7 . . . 105
2.12 Fiabilidad . . . 107
2.13 Compresión de faxes (facsímiles) . . . 110
2.13.1 Codificación unidimensional . . . 110
2.13.2 Codificación bidimensional . . . 115
2.14 Codificación Aritmética . . . 120
2.14.1 Detalles de la implementación . . . 126
2.14.2 Underflow . . . 129
2.14.3 Comentarios Finales . . . 130
2.15 Codificación aritmética adaptativa . . . 132
2.15.1 Codificación del rango . . . 135
2.16 El codificador QM . . . 136
2.17 Compresión de Texto . . . 145
2.18 PPM . . . 146
2.18.1 Principios del PPM . . . 149
2.18.2 Ejemplos . . . 153
2.18.3 Exclusión . . . 154
2.18.4 Cuatro variantes del PPM . . . 155
2.18.5 Detalles de la implementación . . . 156
2.18.6 PPM* . . . 161
2.18.7 PPMZ . . . 163
2.18.8 PPM rápido . . . 165
2.19 Ponderación del árbol del contexto . . . 167
2.19.1 CTW para la compresión de texto . . . 174

3 Métodos de diccionario 175
3.1 Compresión de cadenas . . . 177
3.2 Una sencilla compresión de diccionario . . . 178
3.3 LZ77 (Ventana deslizante) . . . 180
3.3.1 Una cola circular . . . 182
3.4 LZSS . . . 183
3.4.1 LZARI . . . 186

3.4.2 Deficiencias . . . 186
3.5 Tiempos de repetición . . . 187
3.6 QIC-122 . . . 188
3.7 LZX . . . 191
3.8 LZ78 . . . 193
3.9 LZFG . . . 196
3.10 LZRW1 . . . 199
3.11 LZRW4 . . . 202
3.12 LZW . . . 203
3.12.1 Decodificación LZW . . . 206
3.12.2 Estructura del diccionario LZW . . . 207
3.12.3 LZW en la práctica . . . 213
3.12.4 Diferenciación . . . 213
3.12.5 Variantes LZW . . . 214
3.13 LZMW . . . 214
3.14 LZAP . . . 215
3.15 LZY . . . 217
3.16 LZP . . . 220
3.16.1 Ejemplo . . . 222
3.16.2 Consideraciones prácticas . . . 225
3.16.3 LZP1 y LZP2 . . . 225
3.16.4 LZP3 y LZP4 . . . 227
3.17 Buscador de repetición . . . 227
3.18 Compresión UNIX . . . 230
3.19 Imágenes GIF . . . 231
3.20 RAR y WinRAR . . . 231
3.21 El protocolo V.42bis . . . 234
3.22 Varias aplicaciones de LZ . . . 235
3.23 Deflate: Zip y Gzip . . . 236
3.23.1 Los detalles . . . 238
3.23.2 El formato de los bloques en el modo 3 . . . 240
3.23.3 La tabla hash . . . 245
3.23.4 Conclusiones . . . 247
3.24 LZMA y 7-ZIP . . . 248
3.25 PNG . . . 254
3.26 Compresión XML: XMill . . . 258
3.27 Compresores EXE . . . 260
3.28 CRC . . . 261
3.29 Resumen . . . 263
3.30 Las patentes de compresión de datos . . . 264
3.31 Una unificación . . . 266

4 Compresión de imágenes 269
4.1 Introducción . . . 271
4.2 Métodos para la compresión de imágenes . . . 276
4.2.1 Códigos de Gray . . . 279
4.2.2 Métricas de error . . . 287
4.3 Métodos intuitivos . . . 289
4.3.1 Submuestreo . . . 289
4.3.2 Cuantificación . . . 290

4.4 Transformadas de imágenes . . . 290
4.5 Transformadas ortogonales . . . 294
4.5.1 Transformadas bidimensionales . . . 297
4.5.2 Transformada de Walsh–Hadamard . . . 298
4.5.3 La transformada de Haar . . . 300
4.5.4 La transformada de Karhunen-Loève . . . 301
4.6 La transformada discreta del coseno . . . 303
4.6.1 Introducción . . . 304
4.6.2 La DCT como base . . . 311
4.6.3 La DCT como una rotación . . . 319
4.6.4 Los cuatro tipos de DCT . . . 322
4.6.5 La DCT en la práctica . . . 323
4.6.6 El método LLM . . . 327
4.6.7 Implementación hardware de la DCT . . . 329
4.6.8 Descomposición QR de una matriz . . . 330
4.6.9 Espacios vectoriales . . . 331
4.6.10 Rotaciones en tres dimensiones . . . 334
4.6.11 La transformada discreta del seno . . . 335
4.7 Imágenes de prueba . . . 338
4.8 JPEG . . . 343
4.8.1 Luminancia . . . 347
4.8.2 La DCT . . . 349
4.8.3 Cuantificación . . . 350
4.8.4 Codificación . . . 351
4.8.5 Modo sin pérdidas (lossless) . . . 356
4.8.6 El archivo comprimido . . . 357
4.8.7 JFIF . . . 357
4.9 JPEG-LS . . . 360
4.9.1 El codificador . . . 361
4.10 Compresión progresiva de la imagen . . . 366
4.10.1 Codificación de la geometría de crecimiento . . . 371
4.11 JBIG . . . 374
4.11.1 Compresión progresiva . . . 378
4.12 JBIG2 . . . 384
4.12.1 Decodificación de una región genérica . . . 388
4.12.2 Decodificación de una región de símbolos . . . 390
4.12.3 Decodificación de una región de semitonos . . . 391
4.12.4 El proceso de decodificación completo . . . 394
4.13 Imágenes simples: EIDAC . . . 394
4.14 Cuantificación vectorial . . . 396
4.15 Cuantificación vectorial adaptativa . . . 403
4.16 Emparejamiento de bloques . . . 408
4.16.1 Detalles de la implementación . . . 411
4.17 Codificación por truncamiento de bloques . . . 412
4.18 Métodos basados en el contexto . . . 417
4.19 FELICS . . . 420
4.20 FELICS progresivo . . . 423
4.20.1 Código subexponential . . . 426
4.21 MLP . . . 426
4.21.1 Secuenciación de píxeles . . . 428

4.21.2 Predicción . . . 428
4.21.3 Modelado de los errores . . . 429
4.21.4 Polinomios de interpolación . . . 435
4.21.5 Interpolación unidimensional . . . 435
4.21.6 Ejemplo . . . 437
4.21.7 Interpolación bidimensional . . . 438
4.22 Golomb adaptativo . . . 440
4.23 PPPM . . . 442
4.24 CALIC . . . 443
4.24.1 Tres pasadas . . . 444
4.24.2 Cuantificación del contexto . . . 446
4.25 Compresión diferencial sin pérdida . . . 447
4.26 DPCM . . . 448
4.27 Ponderación del árbol de contexto . . . 453
4.28 Descomposición de bloques . . . 454
4.29 Codificación predictiva mediante árboles binarios . . . 458
4.30 Quadtrees . . . 464
4.30.1 Bintrees (árboles binarios) . . . 468
4.30.2 Valores de composición y de diferencia . . . 468
4.30.3 Compresión bintree progresiva . . . 473
4.30.4 Compresión de estructuras N-tree . . . 474
4.30.5 Compresión prefija (mediante prefijos) . . . 480
4.31 Quadrisection (Cuatrisección) . . . 482
4.32 Curvas de relleno de espacio (*space-filling*) . . . 488
4.33 Escaneo de Hibert y VQ . . . 490
4.33.1 Ejemplos . . . 493
4.33.2 La curva de Hilbert . . . 493
4.33.3 La curva de Sierpiński . . . 494
4.33.4 Recorrido de la curva de Hilbert . . . 495
4.33.5 Recorrido de la curva de Peano . . . 499
4.34 Métodos de autómatas finitos . . . 500
4.34.1 Autómata finito ponderado . . . 500
4.34.2 Autómata finito generalizado . . . 512
4.35 Sistemas de funciones iteradas . . . 515
4.35.1 Transformaciones afines . . . 516
4.35.2 Definición de IFS . . . 519
4.35.3 Principios del IFS . . . 522
4.35.4 Decodificación del IFS . . . 525
4.35.5 Codificación del IFS . . . 525
4.35.6 IFS para imágenes en escala de grises . . . 527
4.36 Codificación de las celdas o células . . . 529

5 Métodos wavelet **533**
5.1 La transformada de Fourier . . . 534
5.2 El dominio de la frecuencia . . . 534
5.3 El principio de incertidumbre . . . 542
5.4 Compresión de imágenes de Fourier . . . 544
5.5 La CWT y su inversa . . . 546
5.6 La transformada de Haar . . . 552
5.6.1 La aplicación de la transformada de Haar . . . 556

5.6.2 Propiedades de la transformada de Haar . . . 562
5.6.3 Un enfoque con matrices . . . 566
5.7 Bancos de filtros . . . 569
5.7.1 Derivación de los coeficientes de filtro . . . 576
5.8 La DWT . . . 578
5.9 Descomposición multirresolución . . . 590
5.10 Varias descomposiciones de imágenes . . . 590
5.11 El esquema lifting . . . 598
5.11.1 La transformada wavelet lineal . . . 602
5.11.2 Subdivisión por interpolación . . . 604
5.11.3 Funciones de escalado . . . 607
5.12 La IWT . . . 608
5.13 La pirámide Laplaciana . . . 610
5.14 SPIHT . . . 614
5.14.1 Algoritmo de ordenación del conjunto de particionamiento . . . 617
5.14.2 Árboles de orientación espacial . . . 618
5.14.3 Codificación SPIHT . . . 620
5.14.4 Ejemplo . . . 622
5.14.5 QTCQ . . . 624
5.15 CREW . . . 625
5.16 EZW . . . 625
5.16.1 Ejemplo . . . 628
5.17 DjVu . . . 629
5.18 WSQ, Compresión de huellas dactilares . . . 632
5.19 JPEG 2000 . . . 638

6 Compresión de vídeo 653
6.1 Vídeo analógico . . . 653
6.1.1 El CRT . . . 653
6.2 Video compuesto y por componentes . . . 658
6.3 Vídeo Digital . . . 660
6.3.1 Televisión de alta definición . . . 661
6.4 Compresión de video . . . 664
6.4.1 Métodos de búsqueda subóptimos . . . 669
6.5 MPEG . . . 674
6.5.1 Componentes principales de MPEG-1 . . . 677
6.5.2 Sintaxis del vídeo MPEG-1 . . . 684
6.5.3 Compensación de movimiento . . . 691
6.5.4 Reconstrucción pel . . . 695
6.6 MPEG-4 . . . 695
6.7 H.261 . . . 700
6.7.1 Flujo de datos (*stream*) comprimido H.261 . . . 701
6.8 H.264 . . . 702

7 Compresión de audio 717
7.1 El sonido . . . 718
7.2 Audio digital . . . 721
7.2.1 Audio digital y distribución de Laplace . . . 724
7.3 El sistema auditivo humano . . . 726
7.3.1 Métodos convencionales . . . 729

7.3.2 Compresión de sonido con pérdidas . . . 730
7.4 Formato de audio WAVE . . . 731
7.5 Compansión (companding) μ-Law y A-Law . . . 734
7.6 Compresión de audio ADPCM . . . 739
7.7 Audio MLP . . . 744
7.8 Compresión del habla . . . 747
7.8.1 Propiedades de la voz . . . 747
7.8.2 Codificadores de forma de onda . . . 750
7.8.3 Codificadores de fuente . . . 750
7.8.4 Codificadores híbridos . . . 753
7.9 Shorten . . . 754
7.10 FLAC . . . 759
7.10.1 Polinomio de predicción de cuarto orden . . . 767
7.10.2 Codificación predictiva lineal (LPC) . . . 767
7.11 WavPack . . . 769
7.12 Audio (de) monkey . . . 778
7.13 Codificación de audio sin pérdidas MPEG-4 (ALS) . . . 779
7.14 Reproductores de audio MPEG-1/2 . . . 790
7.14.1 Codificación en el dominio de la frecuencia . . . 793
7.14.2 Formato de los datos comprimidos . . . 796
7.14.3 Codificación: Capas I y II . . . 803
7.14.4 Codificación: Capa II . . . 805
7.14.5 Modelos psicoacústicos . . . 807
7.14.6 Codificación: Capa III . . . 809
7.15 Codificación de Audio Avanzada (AAC) . . . 815
7.15.1 Detalles de la codificación avanzada de audio (AAC) . . . 821
7.15.2 Extensiones MPEG-4 para AAC . . . 835
7.15.3 AAC-LD (Low Delay —Baja latencia) . . . 836
7.15.4 Tests AAC . . . 838
7.16 Dolby AC-3 . . . 840

8 Otros métodos **845**
8.1 El método de Burrows-Wheeler . . . 846
8.1.1 La compresión de L . . . 849
8.1.2 Consejos de implementación . . . 850
8.2 Clasificación de símbolos . . . 851
8.3 ACB . . . 855
8.3.1 El codificador . . . 856
8.3.2 El decodificador . . . 858
8.3.3 Una variación . . . 859
8.3.4 Archivos de contexto . . . 861
8.4 Semejanza de contextos basados en el orden . . . 862
8.5 Cadenas dispersas . . . 867
8.5.1 Creando bits OR . . . 868
8.5.2 Códigos de tamaño variable . . . 869
8.5.3 Códigos de tamaño variable en base 2 . . . 871
8.5.4 Códigos de tamaño variable basados en Fibonacci . . . 872
8.5.5 Compresión mediante prefijos . . . 873
8.6 Compresión de texto basada en palabras . . . 878
8.6.1 Codificación de Huffman adaptativa basada en palabras . . . 879

8.6.2 LZW basado en palabras . . . 880
8.6.3 Predicción de orden 1 basado en palabras . . . 881
8.7 Compresión de imágenes de texto (textuales) . . . 882
8.8 Codificación dinámica de Markov . . . 887
8.8.1 El algoritmo DMC . . . 889
8.8.2 Inicio y parada DMC . . . 892
8.9 Curva de compresión FHM . . . 895
8.10 Sequitur . . . 897
8.11 Compresión de mallas de triángulo: Edgebreaker . . . 902
8.12 SCSU: Compresión Unicode . . . 913
8.12.1 BOCU-1: Compresión Unicode . . . 918
8.13 Formato de Documento Portátil (PDF) . . . 918
8.14 Diferenciación de archivos . . . 920
8.14.1 Utilidad diff de UNIX . . . 921
8.14.2 Diferenciación de archivos: VCDIFF . . . 922
8.14.3 Zdelta . . . 925
8.14.4 Exediff . . . 926
8.14.5 BSDiff . . . 930
8.15 Compresión de datos hiperespectrales . . . 931
8.15.1 Métodos predictivos . . . 935
8.15.2 DCT tridimensional . . . 937
8.15.3 Cuantificación vectorial . . . 939

II Complementos al libro 943

A El código ASCII 945
A.1 Características ASCII . . . 945
A.2 Los caracteres de control ASCII . . . 946

B Conceptos básicos sobre probabilidad 949
B.1 Conjunción y unión de eventos . . . 952
B.2 Probabilidad Condicional . . . 953
B.3 Distribuciones de probabilidad . . . 955
B.3.1 Distribución Gaussiana . . . 956
B.3.2 Distribución de Laplace . . . 959
B.3.3 Distribuciones discretas . . . 959

C Curvas que llenan el espacio 963
C.1 La curva de Hilbert . . . 963
C.2 La curva de Sierpiński . . . 963
C.3 Recorrido de la curva de Hilbert . . . 966
C.4 Recorrido de la curva de Peano . . . 966
C.5 Bibliografía relacionada . . . 966

D Estructuras de datos 967
D.1 Arrays (arreglos o matrices) . . . 967
D.2 Pilas y colas . . . 968
D.3 Listas . . . 969
D.4 Árboles . . . 970

D.5 Grafos . . . 974
D.6 Hashing . . . 974
D.7 Funciones hash . . . 976
D.8 Tratamiento de las colisiones . . . 977
D.9 Bibliografía relacionada . . . 978

E Códigos correctores de errores 981
E.1 Primeros principios . . . 981
E.2 Códigos de escrutinio . . . 982
E.3 Bits de verificación . . . 984
E.4 Bits de paridad . . . 985
E.5 Distancia de Hamming y detección de errores . . . 985
E.6 Códigos de Hamming . . . 987
E.7 El código SEC-DED . . . 989
E.8 Polinomios generadores . . . 990
E.9 Bibliografía relacionada . . . 991

F Autómatas de estados finitos 993
F.1 Bibliografía relacionada . . . 995

G Galería de imágenes 997

H El sistema visual humano 999
H.1 El color y los ojos . . . 999
H.2 El modelo de color HLS . . . 1001
H.3 El Modelo de color HSV . . . 1001
H.4 El modelo de color RGB . . . 1002
H.4.1 El Cubo RGB . . . 1003
H.5 Colores aditivos y sustractivos . . . 1004
H.5.1 Los modelos de color sustractivo . . . 1005
H.6 Colores complementarios . . . 1006
H.7 Visión humana . . . 1007
H.8 Luminancia y crominancia . . . 1008
H.9 Densidad espectral . . . 1012
H.10 El estándar CIE . . . 1015
H.11 Semitonos (halftoning) . . . 1017
H.11.1 Bibliografía relacionada . . . 1019
H.12 Difuminado (dithering) . . . 1019
H.12.1 Difuminado (dither) ordenado . . . 1020
H.12.2 Difuminado promedio restringido . . . 1022
H.12.3 Difuminado (dither) por difusión . . . 1023
H.12.4 Difusión de punto . . . 1026
H.12.5 Bibliografía relacionada . . . 1028

I Matemáticas introductorias 1029
I.1 Sumas útiles . . . 1029
I.2 Matrices . . . 1030
I.2.1 Operaciones con matrices . . . 1030
I.3 Identidades trigonométricas . . . 1033
I.4 Álgebra vectorial . . . 1035

I.4.1 Operaciones con vectores 1035
I.4.2 El triple producto escalar o producto mixto 1037
I.4.3 Proyección de un vector 1037
I.5 Números complejos 1038
I.6 Convolución 1040
I.7 Diagramas de Voronoi 1044
I.8 L-sistemas 1045
I.9 El alfabeto griego 1049
I.10 Polinomios de interpolación 1049

Soluciones a los ejercicios **1051**

Bibliografía **1119**

Glosario **1143**

Unirse a la comunidad sobre la compresión de datos **1167**

Notas sobre el índice alfabético **1169**

Índice alfabético **1169**

Colofón **1195**

Each memorable verse of a true poet has
two or three times the written content.[1]

—Alfred de Musset.

[1] Cada verso célebre perteneciente a un poeta verdadero, posee dos o tres veces el contenido de lo escrito.

Índice de figuras

Intr.0 Histograma de *data.compression.complete.reference.4th.pdf.* 4

1.6 RLE. Parte I: Compresión. 26
1.6 RLE. Parte II: Descompresión. 27
1.7 Superficies uniformes y muestreo de líneas. 31
1.8 Formas de muestreo en RLE. 33
1.9 (a) Código en Matlab; cálculo de *run lengths*. (b) Un *bitmap*. 34
1.10 Vecinos usados para predecir X. 36
1.20 Un árbol binario codificado con 3R. 48
1.21 Histograma de la imagen de Lena. 48

2.8 Ejemplo de código tunstall. 67
2.9 Distribuciones geométricas para $p = 0{,}2$, 0,5 y 0,8. 69
2.13 Un sencillo codificador RLE Golomb adaptativo. 75
2.16 Códigos de Huffman. 80
2.18 Un código Huffman para las 26 letras del alfabeto inglés. 83
2.19 Una máquina de estados finitos. 84
2.20 Códigos Huffman para probabilidades iguales. 84
2.23 Árboles de código de Huffman. 86
2.24 Dos árboles de código de Huffman. 87
2.26 Árboles de código de Huffman binarios y ternarios. 88
2.27 Los árboles de Huffman más cortos, y los más largos. 89
2.30 Pilas mínimas. Cribar la pila tras eliminar la raíz. 92
2.31 Pilas y hojas de Huffman en un array. 93
2.32 El código de escape. 95
2.33 Actualización de un árbol de Huffman. 97
2.34 Cuatro pasos en una variante de Huffman. 99
2.36 Intercambio de punteros en la tabla de códigos de MNP5. (I) 103
2.36 Intercambio de punteros en la tabla de códigos de MNP5. (II) 104
2.37 Intercambio de punteros en MNP5. 105
2.43 Cinco config. de run-lengths. (a) Modo paso. (b) Modo vertical. (c) Modo horizontal. 117
2.44 Diagrama de flujo para la codificación bidimensional (MMR). 118
2.45 Árbol de códigos del Grupo 4. 119
2.49 Código en *Mathematica* para la Tabla 2.52 124
2.60 Codificación aritmética adaptativa. 134
2.61 División del intervalo de probabilidad. 137
2.63 División del intervalo de probabilidad. 138
2.65 División del intervalo de probabilidad. 139
2.69 Reglas del codificador QM con inversión del intervalo. 141

2.74 Diez tries de "zxzyzxxyzx". 157
2.75 Parte I. Seis primeros tries de "assanissimassa". 158
2.75 (Continuación) Los cuatro tries siguientes de "assanissimassa". 159
2.75 (Continuación) Los dos tries finales de "assanissimassa". 160
2.76 Razón de compresión en función de la longitud máxima del contexto. 161
2.77 Contextos deterministas sin límites de longitud en PPMZ. 164
2.78 Dos ejemplos del método PPM rápido para abcbabdbaeabbabe. 166
2.81 Dos árboles de sufijos. 170
2.82 (a) Un árbol de contexto. (b) Un árbol de contexto ponderado. 171
2.83 Árboles de contexto para $b_t = 0, 1$. 173

3.2 Una cola circular. 183
3.3 Dos árboles binarios de búsqueda. 184
3.7 Definición BNF de QIC-122. 190
3.9 LZX: Procesamiento de los desplazamientos y las longitudes. 193
3.11 Un diccionario de árbol LZ78. 195
3.13 Desplazamiento de los búferes de búsqueda y preanálisis en LZRW1. 200
3.15 El codificador LZRW1. 201
3.16 Formato de la salida. 202
3.18 Actualización de una partición en LZRW4. 203
3.21 El algoritmo LZW. 204
3.22 Crecimiento de un Trie LZW para "alf␣eats␣alfalfa". (I) 211
3.22 Crecimiento de un Trie LZW para "alf␣eats␣alfalfa". (II) 212
3.27 El principio de LZP. Parte I 221
3.27 El principio de LZP. Parte II 221
3.28 Compresión LZP de xyabcabcabxy. Parte I. 223
3.28 (Continuación): Compresión LZP de xyabcabcabxy. Parte II. 223
3.35 Dos árboles de Huffman. 241
3.36 Dos árboles de Huffman para longitudes de código. 244
3.37 Árboles binarios de búsqueda en LZMA. 252
3.39 Entrelazado en PNG. 256
3.40 Intervalo (t) entre bytes. 257
3.42 Definición de las probabilidades en un árbol de diccionario. 267

4.2 Digitalización perfecta e imperfecta. 273
4.3 Compresión de diccionario de líneas paralelas. 274
4.4 Valores y diferencias de 24 píxeles adyacentes. 274
4.5 Mapas de (a) una matriz aleatoria y (b) su inversa. 275
4.7 Código en Matlab para separar los planos de bits de la imagen. 281
4.9 Planos de bits 1 y 2 de la imagen de los loros. 282
4.10 Planos de bits 3, 4 y 5 de la imagen de los loros. 283
4.11 Planos de bits 6, 7 y 8 de la imagen de los loros. 284
4.12 Primeros 32 binarios y códigos de Gray reflejados. 285
4.13 Una función en Matlab para calcular la PSNR. 289
4.14 Cuantificación vectorial intuitiva. 290
4.15 Código para la rotación de cinco puntos. 291
4.16 Rotación de una nube de puntos. 292
4.17 Distribución de píxeles de la imagen antes y después de la rotación. 293
4.18 Frecuencias de una imagen. 295
4.19 El kernel de la WHT ordenado para $N = 4$. 300

4.20 Imágenes básicas de la transformada de Haar para $n = 8$. 302
4.21 Experimentos con la DCT unidimensional. 305
4.22 (a) Entrada unidimensional. (b) Su DFT inversa. 306
4.33 Código para el patrón altamente correlacionado. 312
4.35 Frecuencias crecientes. 314
4.36 Valores de ángulo y coseno para una DCT de 8 puntos. 315
4.38 Una representación gráfica de la DCT unidimensional. 317
4.39 Las 64 imágenes base de la DCT bidimensional. 318
4.40 Un ejemplo de la DCT bidimensional. 320
4.41 Un ejemplo de la DCT bidimensional. 321
4.42 La DCT bidimensional como una rotación doble. 322
4.44 Código para cuatro tipos de DCT. 325
4.46 El producto de siete matrices. 328
4.48 Una implementación hardware de la DCT-2. 329
4.49 Una función en Matlab para la descomposición QR de una matriz. 331
4.50 El seno y el coseno como funciones impar y par, respectivamente. 336
4.51 La DCT y la DST de ocho valores de datos idénticos. 336
4.52 Las 64 imágenes base de la DST en dos dimensiones. 337
4.53 Lena y un detalle. 339
4.54 Mandril y un detalle. 340
4.55 Artefactos bloque en JPEG. 340
4.56 Pimientos y un detalle. 341
4.57 Una imagen de tonos discretos. 341
4.58 Una imagen de tonos discretos (Detalle). 342
4.59 Scans en el modo progresivo de JPEG. 346
4.60 Sensibilidad de los conos. 348
4.67 Predicción de un píxel en el modo sin pérdidas. 357
4.68 Formato del archivo JPEG. 359
4.70 Contexto para la predicción de x. 361
4.71 Detección de bordes. 362
4.72 Predicción de correcciones. 362
4.74 Actualización de los arrays A, B y N. 364
4.75 Escaneo del *run*. 365
4.76 Codificación run-length I. 365
4.77 Codificación run-length II. 366
4.78 Decodificación secuencial. 368
4.79 Decodificación progresiva. 369
4.80 Compresión progresiva de imágenes. 370
4.81 Seis reglas y patrones de crecimiento. 372
4.82 El crecimiento de un patrón. 373
4.83 Codificación recursiva de un patrón. 374
4.84 Plantillas para las capas de más baja resolución. 376
4.85 Plantillas para las otras capas. 377
4.86 Coordenadas y posiciones permitidas de los píxeles AT. 378
4.87 Cuatro *stripes* y tres capas en una imagen JBIG. 379
4.89 Píxeles de alta y baja resolución. 380
4.91 Algunos patrones de excepción de JBIG. 382
4.92 Numeración de los píxeles para la predicción determinista. 383
4.95 Regiones típicas de JBIG2. 386
4.97 Cuatro plantillas para la decodificación de una región genérica. 389

4.98 Dos plantillas. . . . 389
4.99 Coordenadas y posiciones permitidas de los píxeles AT. . . . 390
4.100 Tres mapas de bits de símbolos alineados en diferentes esquinas. . . . 391
4.101 Dos tablas de códigos de Huffman para la decodificación JBIG2. . . . 392
4.102 Rejillas de semitonos y regiones. . . . 393
4.103 Contextos de píxeles utilizados por EIDAC. . . . 396
4.104 Doce puntos y cuatro entradas $C_i^{(0)}$ del libro de códigos. . . . 399
4.106 Doce puntos y cuatro entradas $C_i^{(1)}$ del libro de códigos. . . . 401
4.108 Numeración de cuadrantes. . . . 403
4.109 (a) Cobertura en onda. (b) Puntos de crecimiento. (c) Actualización del diccionario. 405
4.110 Una imagen y un diccionario. . . . 408
4.111 Emparejamiento de bloques (LZ77 para imágenes). . . . 409
4.113 Factores de compresión de la BTC para la preservación de dos momentos. . . . 414
4.114 Factores de compresión de la BTC para tres niveles de cuantificación. . . . 417
4.115 Patrones del contexto de un píxel. . . . 418
4.117 Patrones del contexto binivel de un píxel. . . . 420
4.118 (a) Los dos vecinos. (b) Las tres regiones. . . . 421
4.122 Algunos niveles de una imagen de 16×16. . . . 424
4.123 Rotación y escalado. . . . 425
4.125 (a) Dieciséis vecinos. (b) Seis vecinos. . . . 428
4.128 (a) Distribución de errores. (b) Distribuciones de Laplace. (c) Probabilidad de k. . . . 430
4.131 Una máquina de estados finitos para la lectura de píxeles binivel. . . . 441
4.132 Contextos de predicción y estimación de varianza para PPPM. . . . 443
4.134 Sumas ponderadas para contextos de 360^o. . . . 445
4.136 Un histograma de una imagen y sus diferencias. . . . 449
4.137 (a) Un codificador diferencial. (b) DPCM. . . . 450
4.138 Predicción de píxeles y cálculo de diferencias. . . . 451
4.139 Resolución para tres pesos. . . . 452
4.140 Predicción de píxel. . . . 453
4.141 Codificación FABD de una imagen. . . . 455
4.142 Una imagen de 8×8 y sus tres primeras bandas L. . . . 460
4.143 (a) Bandas L_2 y H_2. (b) Bandas L_3 y H_3. (c) Bandas L_4 y H_4. . . . 461
4.144 Los catorce descendientes de los ceros de H_4. . . . 461
4.145 Los cuatro vecinos utilizados para predecir un valor X en H_i. . . . 462
4.146 Cuantificación en BTPC. . . . 463
4.148 Un quadtree (árbol cuaternario). . . . 465
4.149 Un quadtree para una imagen de $2^2 \times 2^2$. . . . 467
4.150 Pseudo–código para localizar un píxel en una imagen. . . . 468
4.151 Un bintree (árbol binario) para una imagen de 8×8. . . . 468
4.152 Determinación de los valores de composición y de diferencia. . . . 469
4.153 Aproximaciones sucesivas en un árbol binario. . . . 471
4.155 Un *bintree* (árbol binario) completo para una imagen binivel. . . . 474
4.156 Un *quadtree* con píxeles en el intervalo [0, 255]. . . . 476
4.157 Ejemplo de compresión prefija. . . . 481
4.158 Una matriz M_0 de 16×16. . . . 486
4.159 Un mapa de bits de 8×8. . . . 489
4.160 Escaneo de Hilbert de 16×16 bloques. . . . 491
4.161 El resultado de 256 bloques. . . . 492
4.162 Curvas de Hilbert de órdenes 1, 2, y 3. . . . 493

4.163 Curva de Hilbert de orden 4. . . . 494
4.164 Curva de Hilbert de orden 5. . . . 495
4.165 Curva de Hilbert de orden 6. . . . 496
4.166 Curvas de Sierpiński de órdenes 1 y 2. . . . 496
4.167 Curvas de Sierpiński de órdenes 1−5. . . . 497
4.168 Tres iteraciones de la curva de Peano. . . . 497
4.171 Numeración de cuadrantes. . . . 501
4.172 Grafos para imágenes sencillas. . . . 502
4.173 Un grafo de cuatro estados. . . . 503
4.174 Un grafo de dos estados. . . . 504
4.175 Imagen $f = (i + j)/2$ en tres resoluciones. . . . 506
4.176 El algoritmo de inferencia WFA recursivo. . . . 509
4.177 Imagen para el Ejercicio 4.75. . . . 513
4.178 Dieciséis transformaciones de imágenes. . . . 513
4.179 Un GFA de cinco estados. . . . 514
4.180 Transformaciones bidimensionales. . . . 516
4.181 Tijeras y shearing (extrusión o cizallamiento). . . . 517
4.182 (a) Triándulo de Sierpiński. (b) Una hoja de helecho. . . . 520
4.184 Cálculo y representación de los atractores del IFS. . . . 521
4.185 Creación del gasket de Sierpiński. . . . 523
4.186 Una imagen autosimilar. . . . 524
4.187 Codificación de un IFS. Versión I. . . . 529
4.188 Codificación de un IFS. Versión II. . . . 530
4.189 Una letra H de 8×8. . . . 530
4.190 Seis *bitmaps* de 8×8 trasladados (a−c) y reflejados (d−f). . . . 531

5.1 Funciones periódicas. . . . 535
5.2 Dominios del tiempo y de la frecuencia. . . . 536
5.3 Dominios del tiempo y de la frecuencia con una componente dc. . . . 537
5.4 Dos funciones $g(t)$ para ser transformadas. . . . 537
5.5 Los primeros tres términos y tres sumas parciales. . . . 538
5.5 (c) Tres sumas parciales en 3D. . . . 539
5.6 Cuatro sumas parciales. . . . 540
5.6 (b) Cuatro sumas parciales de $\frac{\pi}{2} - \sum_{k=1}^{\infty} \frac{\sin(k\,f)}{k}$ en 3D. . . . 541
5.7 Dos representaciones de la música. . . . 545
5.8 Las wavelets (a) de Morlet (b) sombrero Mexicano. . . . 547
5.9 La transformada wavelet continua de una onda sinusoidal pura. . . . 549
5.10 La transformada wavelet continua de un chirrido (chirp). . . . 551
5.11 Las funciones de Haar de escala base y wavelet. . . . 553
5.12 Cálculo de la transformada wavelet normalizada. . . . 555
5.13 Restauración desde una transformación wavelet normalizada. . . . 556
5.14 Una imagen de 8×8 y su descomposición en subbandas. . . . 557
5.15 Transformada wavelet estándar de imágenes y descomposición. . . . 558
5.16 Transformada wavelet de imágenes de pirámide. . . . 559
5.17 Ejemplo de transformada wavelet en pirámide de una imagen. . . . 561
5.18 La imagen de 8×8 reconstruida en la Figura 5.20 y transformada de Haar. . . . 563
5.19 (a) Una imagen de 128×128 con actividad en la derecha. (b) Su transformada. . . . 564
5.20 Tres reconstrucciones con pérdidas de una imagen de 8×8. . . . 565
5.21 Reconstrucción de una imagen simple de 128×128 a partir del 4 % de sus coeficientes. 566
5.22 Código en Matlab para la transformada de Haar de una imagen. . . . 567

5.23 Un banco de filtros de dos canales. . . . 570
5.24 Un banco de filtros ortogonal con cuatro coeficientes de filtro. . . . 571
5.25 Un banco de filtros biortogonal con reconstrucción perfecta. . . . 571
5.26 Función de escalado y wavelet como límites de un árbol logarítmico. . . . 571
5.27 Magnitudes de los filtros (a) Paso bajo y (b) Paso alto. . . . 575
5.28 Un banco de filtros general. . . . 576
5.29 La red diádica muestra la relación entre los factores de escala y el tiempo. . . . 578
5.30 Código para la transformada wavelet discreta directa unidimensional. . . . 581
5.31 Uso de ángulos entre segmentos para añadir más puntos. . . . 582
5.32 Código para la transformada wavelet discreta inversa unidimensional. . . . 582
5.33 Código para la transformada wavelet discreta directa bidimensional. . . . 583
5.34 Código para la transformada wavelet discreta inversa bidimensional. . . . 584
5.36 Ejemplos de wavelets comunes. . . . 588
5.37 Ejemplos de wavelets comunes. . . . 589
5.38 Ejemplos de descomposición multirresolución: (a) Un pico. (b) Varios picos. . . . 591
5.39 Código en Matlab: descomposición multirresolución de un vector fila unidimensional. 593
5.40 Descomposición wavelet de línea. . . . 594
5.41 Descomposición wavelet en quincuce o tresbolillo. . . . 595
5.42 Descomposición wavelet en pirámide. . . . 595
5.43 Descomposición wavelet estándar. . . . 596
5.44 Descomposición wavelet uniforme. . . . 597
5.45 Descomposición adaptativa de paquetes wavelet. . . . 598
5.46 (a) La transformada wavelet de Haar y (b) Su inversa. . . . 599
5.47 Esquema de lifting. Transformada (a) directa, (b) inversa. . . . 601
5.48 Predicción lineal. . . . 603
5.49 Resumen de la transformada wavelet lineal. . . . 604
5.50 Un ejemplo de una subdivisión lineal. . . . 605
5.51 Subdivisiones lineal y cúbica. . . . 606
5.52 Funciones de escalado $\phi_{j,k}(x)$ para $n = 2, 4, 6$, y 8. . . . 608
5.53 Ilustración de la reducción. . . . 612
5.55 Subbandas y niveles en la descomposición wavelet. . . . 614
5.57 Árboles de orientación espacial en SPIHT. . . . 619
5.58 Algoritmo de codificación SPIHT. . . . 620
5.59 Un algoritmo de codificación SPIHT simplificado. . . . 621
5.60 Dieciséis coeficientes y un árbol de orientación espacial. . . . 623
5.61 Codificación QTCQ. . . . 625
5.62 (a) Escaneado de un zerotree. (b) Clasificación de un coeficiente. . . . 627
5.63 Un ejemplo EZW: Tres niveles de una imagen de 8×8. . . . 628
5.64 Ejemplos de huellas dactilares escaneadas (cortesía de Christopher Brislawn) 633
5.66 Descomposición wavelet simétrica de imágenes. . . . 635
5.67 Cuantificación escalar WSQ. . . . 635
5.69 Enrejado de imagen en JPEG 2000. . . . 643
5.70 Extensión de una fila de píxeles. . . . 643
5.72 Organización de la subbanda en JPEG 2000. . . . 644
5.73 Subbandas, recintos, y bloques de código. . . . 645
5.74 Bandas en un bloque de código. . . . 646
5.75 Planos de bits de cuatro coeficientes en un bloque de código. . . . 648
5.77 Ocho vecinos. . . . 649

6.1 (a) Operación CRT. (b) Persistencia. (c) Lín. escan. impares (d) Lín. escan. pares. . 654

6.3 Una máscara de sombra. . . . 657
6.4 (a) Píxeles cuadrados y circulares. (b) Distancia cómoda para la visualización. . . . 657
6.5 Transmisión de televisión (a) compuesta y (b) por componentes. . . . 658
6.8 Diversas resoluciones de vídeo. . . . 663
6.9 (a) Orden de codificación. (b) Orden de visualización. . . . 665
6.10 Compensación de movimiento. . . . 666
6.11 Área de búsqueda. . . . 667
6.12 (a) Búsqueda diluida en distancia $dx = dy = 6$. (b) Una búsqueda local. . . . 670
6.13 Búsqueda monótona de cuadrantes. . . . 670
6.14 Estrategias para algoritmos de búsqueda espaciales dependientes. . . . 671
6.15 Método de búsqueda logarítmica bidimensional. . . . 673
6.16 (a) Organización del decodificador MPEG. (b) Formato fuente. . . . 676
6.17 (a) Un macrobloque. (b) Una posible estructura de sectores (*slices*). . . . 677
6.18 Redondeo de los coeficientes DCT cuantificados. . . . 678
6.20 Los tres tipos de coeficientes DC. . . . 681
6.22 Dos bloques de 8×8 de coeficientes DCT cuantificados. . . . 682
6.24 Las capas de una secuencia (*stream*) de video. . . . 685
6.33 Índices de seis bloques en un macrobloque. . . . 691
6.38 (a) CIF. (b) QCIF. (c) GOB. . . . 702
6.39 H.261 Vídeo Secuencia Layers. . . . 703
6.40 Codificador H.264. . . . 704
6.41 Decodificador H.264. . . . 704
6.42 Perfiles en H.264. . . . 706
6.43 Particiones de macrobloque en H.264. . . . 707
6.44 Nueve modos de predicción de luminancia. . . . 708
6.45 Designación de bordes en un macrobloque. . . . 710
6.46 Límites horizontales y verticales. . . . 710

7.1 Niveles de sonido normales en unidades dB SPL. . . . 720
7.3 Muestreo de una onda de sonido. . . . 722
7.4 Distribuciones de muestras de audio, de números aleatorios, y de diferencias. . . . 725
7.5 Umbral y enmascaramiento del sonido. . . . 727
7.7 Umbral y enmascaramiento del sonido. . . . 729
7.12 μ-law para valores de μ de 25, 255, y 2555. . . . 735
7.13 Palabra de código de μ-Law G.711. . . . 736
7.14 Ejemplo de codificación de entrada -656. . . . 736
7.15 (a) Cuantificación μ-Law Midtread. (b) Cuantificación A-Law Midriser. . . . 737
7.18 (a) Codificador ADPCM y (b) Decodificador. . . . 740
7.19 (a) Cuantificación ADPCM de IMA. (b) Adaptación del tamaño de paso. . . . 742
7.23 Un corte transversal de la cabeza humana. . . . 748
7.24 Ondas sonoras: (a) Con voz y (b) Sin voz. . . . 749
7.25 Calidad de voz versus tasa de bits para codificadores de voz. . . . 749
7.26 Producción del habla: (a) Real. (b) Modelo LPC. . . . 752
7.27 Predictores de órdenes 1, 2, y 3. . . . 756
7.30 Código para un polinomio de Lagrange . . . 767
7.31 El codificador de WavPack. . . . 770
7.33 Áreas medias para la codificación de Golomb recursiva. . . . 775
7.38 Organización del codificador y decodificador ALS. . . . 781
7.39 Predicción adaptativa hacia delante en ALS. . . . 782
7.40 Decodificación de las predicciones en ALS. . . . 783

7.41 Funciones de compansión (*compander*) $C(r)$ y $-C(-r)$. 784
7.42 Ejemplos de subdivisión de bloques. 785
7.43 Predicción progresiva en ALS. 786
7.44 Frecuencias fundamentales y harmónicos. 787
7.45 Codificación avanzada de residuos. 790
7.47 Audio MPEG: (a) Codificador y (b) Decodificador. 793
7.48 Banco de filtros polifase. 794
7.50 Reconstrucción de las muestras de audio. 796
7.51 Audio MPEG: (a) Codificador y (b) Decodificador. 797
7.53 Ejemplos de (a) Cuantificación y (b) Descuantificación. 801
7.56 Formato de salida comprimido. 804
7.57 Organización de las señales de subbanda de la capa II. 805
7.61 Organización de los frames de salida de las capas I y II. 808
7.62 Superposición de ventanas MDCT. 809
7.63 Reducción de aliasing en mariposa en la capa III. 811
7.65 Stream comprimido en la capa III. 813
7.66 Iteración de bucle en la capa III. 814
7.67 Enmascaramiento de la frecuencia de audio. 818
7.68 Enmascaramiento de audio en el tiempo. 818
7.69 Enmascaramiento de audio en frecuencia y tiempo.. 819
7.70 Diagrama de bloques de un codificador perceptual. 819
7.72 Codificador AAC. 823
7.73 Decodificador AAC. 824
7.75 Control de ganancia de AAC (a) Codificador y (b) Decodificador. 825
7.77 Ventanas superpuestas largas y cortas para filtrado AAC. 828
7.81 Decodificación de coeficientes de frecuencia. 832
7.84 Formas de ventana en AAC-LD. 838

8.1 Principios sobre la compresión de Burrows–Wheeler. 848
8.2 Codificación/Decodificación de L mediante mover–al–frente. 850
8.4 Mecanismos de exclusión. 853
8.5 Métodos de búsqueda de cadenas y comparación. 854
8.6 Búsqueda de contexto de orden 1. 855
8.8 Organización del diccionario. 856
8.17 El algoritmo de decodificación. 865
8.19 Representación del nodo $S[1 \ldots \text{P}.\textit{índice}]$. 866
8.20 Lista de prefijos para `yabrecabr`. 866
8.21 Reconstrucción de L_1 a partir de L_3. 869
4.157 Ejemplo de compresión mediante prefijos. 874
8.28 Procedimiento recursivo `RowCol`. 877
8.29 Algoritmo de Huffman adaptativo basado en palabras. 879
8.30 LZW basado en palabras. 880
8.31 Predictor de orden 1 basado en palabras. 881
8.32 Huellas y motas. 883
8.33 (a) Contexto clarividente. (b) Contexto secundario. 886
8.34 4- y 8-Conectividad. 887
8.35 Modelos de estados finitos para la compresión de datos. 888
8.36 Los principios de DMC. 890
8.37 El estado D es un candidato. 892
8.38 Los primeros seis estados. 893

8.39 Modelos DMC iniciales. . . . 894
8.40 Compresión FHM de una curva. . . . 896
8.41 Tres secuencias de entrada y gramáticas. . . . 899
8.42 Ejemplo detallado del codificador Sequitur (Tras [Nevill-Manning y Witten 97]). . . . 901
8.43 Los cinco códigos asignados a los triángulos. . . . 904
8.44 Manipulación de los *half-edges*. . . . 905
8.45 Manejo del caso `E`. . . . 907
8.46 Manejo del caso `C`. . . . 908
8.47 Manejo del caso `L`. . . . 908
8.48 Manejo del caso `R`. . . . 909
8.49 Manejo del caso `S`. . . . 909
8.54 Comando `diff` de UNIX. . . . 922
8.55 Parcheado con el editor `ed`. . . . 922
8.57 Cambios en archivos ejecutables tras la adición de nuevo código. . . . 927
8.58 Organización de datos hiperespectrales. . . . 933
8.59 Organización de un codificador de datos hiperespectrales. . . . 935
8.60 Dos predicciones alternativas de datos hiperespectrales. . . . 936
8.62 DCT tridimensional aplicada a datos correlacionados. . . . 938
8.63 Coeficientes de la DCT tridimensional. . . . 938
8.64 Cuantificador vectorial de particionado localmente óptimo (Codificador). . . . 941

B.1 Distribución Gaussiana (Normal). . . . 957

C.1 Un procedimiento recursivo. . . . 964
C.2 Programa en pascal para una curva de Hilbert de orden i. . . . 964
C.3 Programa en pascal para curvas de Sierpiński de orden $1-5$. . . . 965

D.1 Posición del elemento $A[i,j]$ del array. . . . 968
D.2 Pila y cola. . . . 969
D.3 Listas enlazadas. . . . 970
D.4 Varios árboles. . . . 971
D.5 (a)Un árbol AVL. (b) Un árbol B. . . . 973
D.6 Varios grafos. . . . 975

E.1 Manipulación de información. . . . 982
E.4 Cubos de varias dimensiones y numeración de las esquinas. . . . 986

F.2 Ejemplos de autómatas finitos. . . . 994

G.1 El cubo RGB. . . . 998

H.1 El espectro electromagnético. . . . 1000
H.2 (a) Colores puros. (b) Una longitud de onda dominante. . . . 1001
H.3 El doble hexcono de HLS. . . . 1002
H.4 El hexcono de HSV. . . . 1003
H.5 El cubo RGB. . . . 1004
H.6 Relaciones entre RGB y CMYK. . . . 1006
H.7 Sensibilidad de los conos. . . . 1008
H.8 Representaciones de x^γ y $x^{1/\gamma}$ para $\gamma = 2,3$. . . . 1009
H.9 Un patrón de prueba gamma. . . . 1010
H.10 Transmisión de televisión con corrección gamma. . . . 1010

H.11 Relación entre las representaciones de color RGB e YUV. . . . 1011
H.12 Algunas densidades espectrales. . . . 1013
H.13 Curva de color espectral RGB puro. . . . 1014
H.14 Combinaciones de color RGB. . . . 1015
H.15 Curva de color espectral puro. . . . 1016
H.16 El diagrama de cromaticidad CIE. . . . 1016
H.17 Fondos grises obtenidos por integración espacial. . . . 1018
H.18 Una familiar imagen en negro y blanco. . . . 1020
H.19 Difuminado (*dither*) ordenado. . . . 1021
H.20 Difuminación (*dithering*) promedio restringida. . . . 1022
H.21 Difuminado (*dither*) por difusión. . . . 1024
H.22 Algoritmo de difuminado por difusión. Para (a) color. (b) binivel. . . . 1024
H.23 Matrices de 8×8 para difusión de punto. . . . 1026
H.24 El algoritmo de difusión de punto. . . . 1027
H.25 Dos matrices de clase para la difusión de punto. . . . 1028

I.1 Proyección de un vector. . . . 1037
I.2 La convolución de $x(t)$ y $g(t)$. . . . 1042
I.3 Aplicación de la convolución a una función de eliminación de ruido. . . . 1042
I.4 Tres diagramas de Voronoi de cuatro puntos. . . . 1044
I.5 Generaciones sucesivas del copo de nieve de Koch. . . . 1046
I.6 Tres iteraciones de la curva de Peano. . . . 1047
I.9 Ejemplos de movimientos de tortuga. . . . 1048

Sol.6 Tres árboles de Huffman para ocho símbolos. . . . 1056
Sol.7 Construcción errónea de un árbol de Huffman (8 símbolos). . . . 1057
Sol.9 Códigos de Huffman para probabilidades iguales. . . . 1058
Sol.11 El cribado de la pila. . . . 1060
Sol.12 (a) Pilas. (b) Árbol de código de Huffman. . . . 1061
Sol.13 Ejercicio 2.25. Parte I. . . . 1062
Sol.13 Ejercicio 2.25. Parte II. . . . 1063
Sol.13 Ejercicio 2.25. Parte III. . . . 1064
Sol.13 Ejercicio 2.25. Parte IV. . . . 1065
Sol.14 Ejemplo de codificación bidimensional. . . . 1066
Sol.21 El trie final de "`assanissimassa`". . . . 1069
Sol.22 Árboles de contexto para 000|0, 000|00 , 000|1 y 000|11. . . . 1071
Sol.28 Matrices de covarianza de valores correlacionados y descorr.. . . . 1078
Sol.30 Códigos angulares de rueda RGC y binario. . . . 1079
Sol.31 Imágenes básicas WHT de 8×8 y el código en Matlab. . . . 1080
Sol.32 Coordenadas para la ruta en zigzag. . . . 1082
Sol.33 Doce puntos y cuatro entradas $C_i^{(1)}$ del libro de códigos. . . . 1083
Sol.37 Una interpolación bicúbica de una porción de superficie. . . . 1089
Sol.38 Resolución para tres pesos. . . . 1090
Sol.39 (a) Bandas L_2 y H_2. (b) Bandas L_3 y H_3. (c) Bandas L_4 y H_4. . . . 1090
Sol.40 Un árbol binario para una imagen de 8 bits por píxel. . . . 1091
Sol.41 Las 15 tuplas de 6 con dos píxeles blancos. . . . 1091
Sol.42 Las primeras tres iteraciones de la curva de Peano. . . . 1092
Sol.44 Un grafo de dos estados. . . . 1093
Sol.45 Código en Matlab para la matriz $m_{i,j} = (i+j)/2$. . . . 1094
Sol.46 Un GFA para el Ejercicio 4.75. . . . 1096

Sol.47 Otro *gasket* de Sierpiński 1097
Sol.48 Un producto interno para $a = 1$ y $b = 2, 4, 6$. 1098
Sol.49 Quince valores y gráfica de densidad de $W(a, b)$. 1098
Sol.50 La descomposición en subbandas de una línea diagonal. 1099
Sol.51 Imagen 128×128 de Lena: tres reconstrucciones con pérdidas. 1100
Sol.52 Cálculo de W y la transformada $W \cdot I \cdot W^T$: código y resultados. 1101
Sol.53 Código para la transformada wavelet discreta inversa 1D. 1102
Sol.54 Difuminado como resultado de una cuantificación tosca. 1103
Sol.55 Código para un polinomio de interpolación de grado 5. 1104
Sol.56 Código en Matlab para la IWT directa e inversa. 1104
Sol.57 Código para comprobar la predicción de 4^{o} orden. 1106
Sol.58 Permutaciones de "`sssssssssh`". 1108
Sol.60 Nuevo estado clonado D'. 1111
Sol.61 Estado 6 añadido. 1111
Sol.62 La mejora de la gramática de la Figura 8.41. 1112
Sol.64 Difuminado en orden: Tres áreas uniformes. 1116

Índice de tablas

Intr.1 Probabilidades de las letras en inglés. . . . 5
Intr.2 Frecuencias y probabilidades de los caracteres. . . . 6
Intr.3 El Calgary Corpus (2005). . . . 15
Intr.4 El Canterbury Corpus (2001). . . . 16
Intr.5 El Artificial Corpus (2001). . . . 16
Intr.6 El Large Corpus (2001). . . . 16

1.1 Las 26 letras del alfabeto Braille. . . . 20
1.2 (a) Algunas palabras y cadenas en Braille. . . . 20
1.2 (b) á, é, í, ó, ú, ñ y signos de puntuación en Braille. . . . 21
1.2 (c) Signos Mayúscula y Número, con algunos ejemplos. . . . 21
1.3 El Código de Visualización CDC. . . . 22
1.4 El código de Baudot. . . . 23
1.5 Compresión prefija. . . . 24
1.11 Recuento condicional para píxeles de 4 bits. . . . 36
1.12 Códigos prefijo para píxeles de 4 bits. . . . 37
1.13 La cabecera de BinHex. . . . 38
1.14 Codificación con y sin *mover–al–frente*. . . . 40
1.15 Ejemplos de códigos de tamaño variable. . . . 41
1.16 Codificación de palabras de múltiples letras. . . . 43
1.17 Una tabla de cuantificación logarítmica. . . . 45
1.18 Ejemplo de Reducción de Rango. . . . 47
1.19 El peor rendimiento de la Reducción de Rango. . . . 47

2.1 El código Morse para el idioma inglés. . . . 52
2.2 Códigos de tamaño variable. . . . 59
2.3 Algunos códigos unarios. . . . 60
2.4 El código unario general $(3, 2, 9)$. . . . 60
2.5 El código unario general $(2, 1, 10)$. . . . 61
2.6 Algunos códigos prefijo (C_1,C_2, C_3 y C_4). . . . 63
2.7 Probabilidades ideales de ocho enteros para tres códigos. . . . 64
2.10 (a) Algunos ejemplos de m, c y $2^c - m$. . . . 70
2.10 (b) Algunos códigos Golomb para $m = 2, \ldots, 13$; $n = 0, \ldots, 6$. . . . 70
2.10 (c) Algunos códigos Golomb para $m = 2, \ldots, 13$; $n = 7, \ldots, 12$. . . . 70
2.11 Algunos Códigos Golomb para $m = 14$. . . . 71
2.12 Los primeros códigos Golomb para $m = 14$ y $m = 16$. . . . 72
2.14 Ejemplo de Shannon–Fano. . . . 78
2.15 Ejemplo de Shannon–Fano balanceado. . . . 78
2.17 Ejemplo de código de Huffman. . . . 81

2.21 Composición de nodos. . . . 85
2.22 Composición de nodos. . . . 85
2.25 Códigos. . . . 87
2.28 Ejemplo de código de Huffman canónico. . . . 90
2.29 Algunos datos. . . . 90
2.35 Los tokens de MNP5. . . . 101
2.38 Las tablas de código de MNP7. . . . 106
2.39 Tiempos de transmisión de un fax. . . . 111
2.40 Los ocho documentos de adiestramiento del CCITT. . . . 111
2.41 (a) Códigos de fax Grupos 3 y 4. Códigos de terminación. . . . 112
2.41 (b.I) Códigos de fax Grupos 3 y 4. Códigos de establecimiento. . . . 113
2.41 (b.II) Códigos de Fax Grupos 3 y 4. Códigos de establecimiento. . . . 113
2.42 Códigos 2D para el método del Grupo 4. . . . 116
2.46 Códigos para el Grupo 4. (I) . . . 119
2.46 Tabla de código para el modo sin compresión. (II) . . . 120
2.47 Frecuencias y probabilidades de cinco símbolos. . . . 122
2.48 El proceso de codificación aritmética. . . . 123
2.50 El proceso de decodificación aritmética. . . . 124
2.51 Probabilidades (sesgadas) de tres símbolos. . . . 125
2.52 Codificación de la cadena $a_2\,a_2\,a_1\,a_3\,a_3$. . . . 125
2.53 Decodificación de la cadena $a_2\,a_2\,a_1\,a_3\,a_3$. . . . 125
2.54 Codificación de la cadena $a_3\,a_3\,a_3\,a_3\,eof$. . . . 126
2.55 Decodificación de la cadena $a_3\,a_3\,a_3\,a_3\,eof$. . . . 126
2.56 Codificación de `SWISS␣MISS` por desplazamiento. . . . 128
2.57 Codificación de la cadena $a_3\,a_3\,a_3\,a_3\,a_3$ por desplazamiento. . . . 130
2.58 Un alfabeto de 10 símbolos con contadores. . . . 133
2.59 Un alfabeto de 10 símbolos con contadores. . . . 133
2.62 Codificación de cuatro símbolos con $Q_e = 0{,}5$. . . . 138
2.64 Codificación de cuatro símbolos con $Q_e = 0{,}1$. . . . 138
2.66 Renormalización añadida a la Tabla 2.62. . . . 140
2.67 Renormalización añadida a la Tabla 2.64. . . . 140
2.68 Ilustrando la inversión del intervalo. . . . 141
2.70 Tabla (ilustrativa) de estimación de probabilidades. . . . 143
2.71 Tabla de estimación de probabilidades del codificador QM. . . . 144
2.72 Contextos y contadores para "`xyzzxyxyzzx`". . . . 151
2.73 Contextos, contadores (f), y probabilidades (p) de "`assanissimassa`". . . . 152
2.79 Siete valores de $P_a\,(2,3)$ . . . 168
2.80 Estimación de KT para algunos $P_e\,(a,b)$. . . . 170

3.1 Probabilidades y códigos de Huffman para un alfabeto de 2 símbolos. . . . 178
3.4 Cadenas de cinco caracteres. . . . 185
3.5 Codificación de los tiempos de repetición para $L = 3$. . . . 188
3.6 Valores del campo $< longitud >$. . . . 189
3.8 Ejemplo de codificación. . . . 191
3.10 Primeros 14 pasos de codificación en LZ78. . . . 194
3.12 El código unario general $(10, 2, 14)$. . . . 198
3.14 Primeros 7 pasos de la codificación de "`that␣thatch␣thaws`". . . . 201
3.17 Codificación de la longitud en LZRW4. . . . 202
3.19 Codificación de `sir␣sid␣eastman␣easily␣teases␣sea␣sick␣seals`. . . . 205
3.20 Un diccionario LZW. . . . 206

3.23 LZW y tres variantes. . . . 214
3.24 Ejemplo de LZMW. . . . 216
3.25 Ejemplo de LZAP. . . . 217
3.26 Ejemplo de LZY. . . . 219
3.29 Códigos usados por LZP1 y LZP2 para longitudes de *match*. . . . 226
3.30 (a) `REP` cuando $i = 8$. (b) `REP` cuando $i = 13$. . . . 228
3.31 Código prefijo propuesto. . . . 230
3.32 Edocs Literal/Longitud para el modo 2. . . . 240
3.33 Códigos Huffman para edocs en el modo 2. . . . 240
3.34 Treinta códigos prefijo para distancias, en el modo 2. . . . 241
3.38 Opciones LZMA para el *Match Finder*. . . . 253
3.41 Paridades horizontal y vertical. . . . 262

4.1 Ilustración del método IGS. . . . 273
4.6 Primeros 32 binarios y códigos de Gray reflejados. . . . 280
4.8 Primeros 32 binarios y códigos de Gray. . . . 281
4.23 DCT bidimensional de un bloque de valores correlacionados. . . . 308
4.24 DCT bidimensional de un bloque de valores aleatorios. . . . 309
4.25 Un patrón de tonos continuos. . . . 309
4.26 Sus coeficientes de la DCT. . . . 309
4.27 Cuantificación profunda de sólo 4 coeficientes distintos de cero. . . . 310
4.28 Resultados de la IDCT. . . . 310
4.29 Una imagen de tonos discretos. . . . 310
4.30 Sus coeficientes de la DCT. . . . 310
4.31 Cuantificación ligera truncando a enteros. . . . 311
4.32 La IDCT. Malos resultados. . . . 311
4.34 La matriz DCT para $n = 3$. . . . 313
4.37 La matriz DCT sin normalizar en una dimensión para $n = 8$. . . . 316
4.43 Valores de ángulos para la DCT-1 y la DCT-2. . . . 324
4.45 Número de operaciones aritméticas. . . . 327
4.47 Seis valores del coseno distintos para la DCT-2. . . . 329
4.61 Tablas de cuantificación recomendadas. . . . 351
4.62 Tabla de cuantificación $1 + (i + j) \times 2$. . . . 351
4.63 Codificación de diferencias de coeficientes DC. . . . 352
4.64 Códigos de Huffman recomendados para los coeficientes AC de luminancia. . . . 354
4.65 Códigos de Huffman recomendados para los coeficientes AC de crominancia. . . . 355
4.66 Codificación de los coeficientes AC. . . . 356
4.69 Marcadores de JPEG. . . . 358
4.73 Predicción de errores; sus mapeos, y códigos $LG\,(2,\,32)$. . . . 364
4.88 Los cuatro órdenes posibles de las capas. . . . 379
4.90 Algunos patrones de excepción de JBIG. . . . 381
4.93 Predicción del píxel 8. . . . 383
4.94 Predicción del píxel 9. . . . 383
4.96 Parámetros para la decodificación de una región genérica. . . . 388
4.105 Doce distorsiones para $k = 0$. . . . 399
4.107 Doce distorsiones para $k = 1$. . . . 402
4.112 Número de códigos unarios generales $(2, 1, k)$ para $k = 2, 3, \ldots, 11$. . . . 411
4.116 Recuentos y probabilidades para los primeros cuatro píxeles. . . . 419
4.119 Los códigos para la región central. . . . 422
4.120 Los códigos para otra región. . . . 422

4.121 Un mapa de bits de 4×4. 422
4.124 Algunos códigos subexponenciales. 427
4.126 16 pesos. (a) Enteros. (b) Normalizados. 429
4.127 Algunos valores de la distribución de Laplace con $V = 3, 4, 5$ y $1\,000$. 431
4.129 Codificación MLP. 434
4.130 Treinta y Siete valores de varianza cuantificados. 434
4.133 Los 4×4 valores $\mu[i, j]$ para una imagen de 8×8 píxeles. 444
4.135 25 valores del seno y 24 diferencias. 449
4.147 Las trece predicciones utilizadas por BTPC2. 464
4.154 Posibles códigos de Huffman para valores de diferencia de 4 bits. 472
4.169 Coordenadas de los nodos en H_i. 498
4.170 Números de nodo en H_i. 499
4.183 Seis transformaciones. 521

5.35 Coeficientes de filtro para algunas wavelets comunes (Parte I). 586
5.35 Coeficientes de filtro para algunas wavelets comunes (Parte II). 587
5.54 Valores de $\omega(m, n)$ para $a = 0{,}6$. 613
5.56 Coeficientes de transformada ordenados por magnitudes absolutas. 616
5.65 Coeficientes de filtro wavelet simétricos para la WSQ. 634
5.68 Códigos WSQ para índices de cuantificación y run lengths. 637
5.71 Extensiones izquierda y derecha mínimas. 643
5.76 Codificación de cuatro planos de bits para cuatro coeficientes. 648
5.78 Nueve contextos para la significación de la pasada de propagación. 649

6.2 Relaciones de aspecto de la televisión y el cine. 656
6.6 Parámetros de vídeo para aplicaciones típicas. 661
6.7 Resoluciones y capacidades de seis formatos de DTV. 663
6.19 Tabla de cuantificación por defecto de luminancia para codificación intra. 680
6.21 Códigos para los coeficientes DC (luminancia y crominancia). 681
6.23 Códigos run-level de longitud variable (Parte 1). 683
6.23 Códigos run-level de longitud variable (Parte 2). 684
6.25 Códigos de inicio de video de MPEG. 686
6.26 Códigos pel de relación de aspecto de MPEG. 687
6.27 Tasa de imágenes en MPEG y aplicaciones típicas. 687
6.28 Restricción de límites en los parámetros. 687
6.29 Códigos de tipos de imágenes. 688
6.30 Códigos para incremento de dirección del macrobloque. 689
6.31 Códigos de longitud variable para los tipos de macrobloque. 689
6.32 Códigos para incremento de dirección de los macrobloques. 690
6.34 Parámetros de cabecera para el cálculo del vector de movimiento. 694
6.35 Nombres genéricos para parámetros de desplazamiento de movimiento. 694
6.36 Rango de valores de `motion_r` en función de `f_code`. 694
6.37 Los estándares $p \times 64$. 700
6.47 Tamaños de paso de la cuantificación. 712
6.48 Valores de nC para seleccionar una tabla. 713
6.49 Umbrales para el incremento de `suffixLength`. 714

7.2 Niveles de ruido en unidades de potencia y presión. 721
7.6 Veintisiete bandas críticas aproximadas. 728
7.8 Muestras de 16 bits mapeadas a números de 15 bits. 731

7.9 Un bloque de formato. . . . 732
7.10 Códigos de compresión Wave. . . . 732
7.11 Ejemplo de un archivo .WAV. . . . 733
7.16 Especificaciones del cuantificador μ-Law. . . . 738
7.17 Especificaciones del cuantificador μ-Law. . . . 738
7.20 Tamaño de paso y salidas de 4 bits del cuantificador. . . . 742
7.21 Primera tabla para el ADPCM de IMA. . . . 743
7.22 Segunda tabla para el ADPCM de IMA. . . . 743
7.28 Varios códigos de Rice positivos y negativos. . . . 757
7.29 Códigos Pod, Elias Gamma , y Elias Gamma sesgado. . . . 758
7.32 Códigos binarios ajustados para seis valores de m. . . . 773
7.34 Probabilidades para la codificación de Golomb recursiva. . . . 776
7.35 Tres bits por residuo. . . . 776
7.36 Conversión óptima de magnitudes unarias. . . . 777
7.37 Ahorros netos para la conversión unaria. . . . 778
7.46 Tasas de bits en las tres capas. . . . 792
7.49 Coeficientes de la ventana de (a) análisis y (b) síntesis. . . . 795
7.52 Factores de escala de las capas I y II. . . . 800
7.54 Distribución de bits y cuantificación en la Capa I. . . . 802
7.55 Coeficientes de cuantificación in las capas I y II. . . . 803
7.58 Capa II. Patrones de transmisión del factor de escala. . . . 806
7.59 Capa II. Selección de información del factor de escala. . . . 806
7.60 Clases de cuantificación para la capa II. . . . 807
7.64 Tamaño de infor. adicional para audio MPEG de la capa III. . . . 812
7.71 Los tres perfiles de AAC. . . . 822
7.74 Los primeros 48 coeficientes $Q(n)$. . . . 826
7.76 12 conjuntos de especificaciones de frecuencia. . . . 827
7.78 Bandas de factor de escala para ventanas largas a 44,1 y 48 kHz. . . . 829
7.79 Códigos Huffman para diferencias de factores de escala. . . . 830
7.80 12 libros de códigos de Huffman. . . . 830
7.82 Libro 1 de códigos de Huffman (Parcial). . . . 833
7.83 Límite superior de frecuencia para la predicción. . . . 834
7.85 Tests AAC en 1996 (después de [Bosi y Goldberg 03]). . . . 839
7.86 Tasas de bits (kbps) en AC-3. . . . 842
7.87 Configuraciones de canal en AC-3. . . . 842

8.3 Probabilidades de conjeturas del texto en inglés. . . . 852
8.7 Seis contextos y contenidos. . . . 856
8.9 Once contextos y sus contenidos. . . . 857
8.10 Dieciocho contextos y sus contenidos. . . . 857
8.11 (a) Diccionario ordenado. (b) Lista asociativa. . . . 860
8.12 (a) Lista asociativa ordenada. (b) Tres líneas. . . . 860
8.13 (a, b) Dos posibilidades, y (c) Una imposibilidad, de tres líneas. . . . 860
8.14 Un ejemplo. . . . 861
8.15 La lista ordenada para bacacaba. . . . 863
8.16 Construcción de las listas ordenadas para bacacaba. . . . 864
8.18 (a) Lista ordenada para yabrecabr. (b) Inserción del prefijo siguiente. . . . 865
8.22 (a) n/l Run lengths. (b) $2^l - 1$ Grupos no cero. . . . 870
8.23 $\log_2(n/l)$ Clases de Run Lengths. . . . 870
8.24 Veinte códigos. . . . 871

8.25 Códigos para R_i en base 2. . . . 872
8.26 Códigos para F_i basados en Fibonacci. . . . 873
8.27 Dependencia de P de n. . . . 876
8.50 Un ejemplo de paso de preprocesamiento. . . . 911
8.51 Ventanas estáticas. . . . 915
8.52 Ventanas dinámicas. . . . 915
8.53 Desplazamientos de ventanas. . . . 916
8.56 Codificación para los flags de zdelta. . . . 926
8.61 Cinco configuraciones de predicción para datos hiperespectrales. . . . 937

A.1 El código ASCII. . . . 946

B.2 Algunos valores de la distribución de Laplace con $V = 3, 4, 5$, y 1 000. . . . 959

E.2 Bits de paridad. . . . 985
E.3 Ejemplos de códigos con $m = 2$. . . . 986

F.1 Suma de 14 y 23. . . . 993

I.7 Convenciones de un L-sistema para comandos de tortuga. . . . 1048
I.8 Parámetros de tortuga adicionales. . . . 1048

Sol.1 Codificación con el método *avanzar-k*. . . . 1053
Sol.2 Decodificación de palabras de múltiples letras. . . . 1054
Sol.3 Probabilidades y entropías de dos símbolos. . . . 1054
Sol.4 El código unario general $(10, 2, 14)$. . . . 1055
Sol.5 Ejemplo de Shannon–Fano. . . . 1056
Sol.8 Ejemplo de código de Huffman. . . . 1057
Sol.10 Códigos de Huffman para 5-8 símbolos. . . . 1057
Sol.15 Codificación de la cadena $a_2a_2a_2a_2$. . . . 1066
Sol.16 Codificación de cuatro símbolos con $Q_e = 0{,}5$. . . . 1067
Sol.17 Codificación de 4 símbolos con $Q_e = 0{,}1$. . . . 1067
Sol.18 Renormalización añadida a la Tabla Sol.16. . . . 1067
Sol.19 Renormalización añadida a la Tabla Sol.17. . . . 1067
Sol.20 Datos estables vs. datos variables. . . . 1069
Sol.23 Los 12 pasos siguientes en el ejemplo de LZ78. . . . 1072
Sol.24 Últimos pasos para codificar "`that␣thatch␣thaws`". . . . 1072
Sol.25 Codificación LZW de "`alf␣eats␣alfalfa`". . . . 1073
Sol.26 Compresión LZMW de "`swiss␣miss`". . . . 1075
Sol.27 Compresión LZMW de "`yabbadabbadabbadoo`". . . . 1075
Sol.29 Primeros 32 binarios y códigos de Gray. . . . 1077
Sol.34 Doce distorsiones para $k = 1$. . . . 1084
Sol.35 Recuentos y probabilidades de los cuatro píxeles siguientes. . . . 1085
Sol.36 Los códigos para una región central. . . . 1085
Sol.43 Las cuatro orientaciones de H_2. . . . 1093
Sol.59 La construcción de las listas ordenadas para `ubladiu`. . . . 1109
Sol.63 Código de Hamming para $m = 16$. . . . 1115

Prefacios y agradecimientos

Prefacio a la edición en castellano

La decisión de escribir este libro se debe a la no existencia, en castellano, de una obra que resuma los conceptos fundamentales de la compresión de datos, profundizando lo suficiente como para que —con un poco de inteligencia—, cualquiera sea capaz de enfrentarse al difícil problema de representar la información en el mínimo espacio posible, siendo capaz —incluso— de crear sus propios algoritmos.

Este es un libro que debería leer toda mente inquieta, que utilice a diario un compresor de datos. También deberían leerlo todos aquellos genios que han ignorado el tema de la compresión de datos; ellos, especialmente, están en condiciones de generar los más bellos algoritmos, utilizando como "semilla" este libro, y como inspiración, el mundo que nos rodea.

Todos aquellos que ya tengan conocimientos sobre la compresión de datos, podrán refrescar los conceptos fundamentales, hojeando (u ojeando, según proceda) el libro, y resolviendo la gran cantidad de ejercicios del mismo.

He procurado hacer una buena traducción del libro, aún así, podría haber algún error en el texto. Apelo a la inteligencia del lector para discernir entre lo correcto y lo incorrecto. Si algún alma inquieta detecta alguna de esas escurridizas erratas que aparecen en toda obra primeriza, puede enviar su localización y descripción a la siguiente dirección de correo electrónico creada para tal efecto: dechadobook@hotmail.es, o bien a mi dirección e-mail personal: micifut@hotmail.com.

Esta obra es algo distinta a la original —en inglés—. Tiene la misma estructura —salvo que el índice está casi al principio del libro, antes de los prefacios— y mantiene la numeración de los capítulos, las secciones, las subsecciones, e incluso de las tablas y las figuras. Pero he realizado algunas adiciones y cambios; los más importantes son los siguientes:

- Todas las tablas y figuras se han convertido al castellano hasta donde me ha sido posible.
- Ahora existe un índice de tablas y un índice de figuras. Creo que son bastante útiles.
- El histograma de bytes de la introducción (pág. 4), está realizado con datos reales, calculados por un programa realizado en pascal —y compilado con Free Pascal— que generó el código TEX necesario.
- Se ha actualizado la Tabla Intr.3 del *Calgary Corpus* con datos más recientes (2005).
- Se ha añadido una descripción de otros *Corpus*; esta información se puede obtener en la url http://corpus.canterbury.ac.nz/descriptions/ —abril de 2011—. Las tablas de los distintos *Corpus*, proceden de los conjuntos de archivos que se pueden bajar desde allí, y por tanto, corresponden a enero de 2001.
- La Tabla 1.2a, muestra algunos códigos en Braille más, para otras cadenas comunes en inglés. Se han añadido: la Tabla 1.2b —para los caracteres en español— y la curiosa Tabla 1.2c —para

representar las mayúsculas y los números (incluídos los fraccionarios) en Braille—. Un pequeño párrafo explica cómo se utilizan estas tablas.

- La Figura 1.6a, es equivalente a la original —en inglés—; pero la Figura 1.6b, ha sido modificada, porque la versión en español —a mi entender— es más correcta.
- En el antepenúltimo párrafo de la Subsección 1.3.1 se ha corregido una errata del libro original en inglés —se ha sustituido "between 0 and ±255" por "entre 0 y ±127"—.
- En la Figura 1.9a, se ha cambiado el nombre de las variables para facilitar la explicación —que también se ha incluido— del código .
- Se han corregido dos valores erróneos en las Tablas 1.18 y 1.19: en ambas, $0101 = 6$ es incorrecto —debe ser $0110 = 6$—; en la segunda, el último elemento de la segunda columna de la izquierda —111101— debe ser 111100.
- La Figura 1.21 ahora tiene integrada la imagen de Lena.
- Se ha añadido una explicación para el código en *Mathematica* de la Sección 2.1. Creo que es muy útil para comprender en poco tiempo cómo calcula la entropía de la cadena `swiss␣miss`.
- Se ha corregido una errata del segundo párrafo de la Sección 2.2: el cálculo $0,49 \cdot \log_2 0,49 \approx -0,50$ y no $-0,050$.
- En la Subsección 2.3.2, se han corregido dos erratas:
  - En el ejemplo de codificación del número 16 con el código C_3, el cálculo correcto de $C_2(5)$ es 10110.
  - Más abajo, en la expresión de $P(n)$ para el caso ideal del código C_3: $\log_2 n$ era incorrecto y se ha sustituído por $\log_2 2n$. Asimismo, se han calculado de nuevo los valores de la Tabla 2.7, corrigiendo aquellos que dependían de este error.
- El valor —para $m = 4$, $n = 12$— de la Tabla 2.10c, era erróneo: en el original —en inglés— ponía 11110 | 00, cuando el correcto es 1110 | 00. Dicha tabla se ha dividido en tres —Tablas 2.10a, 2.10b y 2.10c—.
- Se han corregido varias erratas en la Sección 2.5:
  - El segundo ejemplo para ilustrar el rendimiento del código Golomb en la compresión de los run lengths, tiene 98 bits —y no 94—, por lo que el factor de compresión es de $^{98}/_{47} = 2,1$. También el número de ceros de la cadena es 89 —y no 85—.
  - En el tercer ejemplo, el cálculo de m es:

$$m = \left\lceil -\frac{\log_2(1+p)}{\log_2(p)} \right\rceil = \left\lceil -\frac{\log_2(1+0{,}9999)}{\log_2(0{,}9999)} \right\rceil = \lceil 6930{,}625 \rceil = 6931.$$

- Los árboles de la Figura 2.16 (Sección 2.8) se han reconstruído verticalmente, debido a que ésto facilita su lectura considerablemente. Se ha modificado el texto dependiente de dicho cambio, eliminando así las inconsistencias generadas.
- En la Subsección 2.8.5, se ha corregido la errata de la línea 6, página 84 —del original, en inglés— con el texto aconsejado: "Un posible conjunto de probabilidades para los símbolos es:". La corrección está al final del tercer párrafo de la citada subsección.

- En el ejercicio 2.25 de la Subsección 2.9.4 se ha corregido la cadena de 11 símbolos; la correcta es: $sir_{\sqcup}sid_{\sqcup}is_{\sqcup}$.
- Se ha modificado la Tabla 2.35 —los tokens de MNP5— por ser incorrecta; además, se ha ampliado. Al final del quinto párrafo de la Sección 2.10 se ha corregido una errata: 111|11111111 era incorrecto, el valor correcto es 111|1111111.
- En la Sección 2.12 se ha actualizado la información referente al Instituto Nacional Americano de eStándares (ANSI).
- En la Sección 2.13 se ha actualizado la información referente a la velocidad de los faxes actuales.
- En la Subsección 2.13.1 se han añadido algunas notas explicativas de cierta importancia. La tabla 2.41 se ha completado con notas y fragmentado en: Tabla 2.41a, Tabla 2.41b-I y Tabla 2.41b-II. Asimismo, se ha modificado la URL donde se pueden localizar las recomendaciones T.4 y T.6 —en pdf—.
- En la Subsección 2.13.2 se han corregido los datos de acuedo con la información contenida en la fuente original (el archivo pdf de nombre *T-REC-T.4-200307-I!!PDF-S*, que se puede obtener en la URL indicada en la subsección anterior). La Tabla 2.42 tenía errores. Se ha cambiado el nombre de la Figura 2.43 por uno más corto. La Figura 2.44 ha sido ampliada y reconstruida en castellano. La Figura 2.45 se ha corregido, y ahora es un árbol vertical (más extenso, pero más claro). Se ha añadido una tabla de código para el modo sin compresión (Tabla 2.46-II). Y se ha corregido la Figura Sol.14, correspondiente a la respuesta del ejercicio 2.33 de esta subsección (la flecha en el modo paso).
- Los pasos de la decodificación en el ejemplo de la Subsección 2.14.1 ahora son más legibles.
- En la Tabla 2.57 de la Subsección 2.14.2 se ha corregido el valor de `H` del segundo paso, el cual estaba mal calculado.
- La Figura 3.9 se ha traducido y corregido —faltaban algunos enlaces entre algunas entidades—.
- En el párrafo de la Sección 3.12, que recoge los datos de salida mostrados en la Tabla 3.19, se ha corregido el valor 280 (`␣se`) —en la última línea del párrafo—. El valor correcto es el indicado en la citada tabla: 281 (`␣se`).
- La Figura 3.22 se ha dividido en dos (3.22-I y 3.22-II) para facilitar su lectura.
- En la columna *Salida* de la Tabla 3.24 se ha corregido el código ASCII del símbolo `t`: el correcto es 116. La columna `S'` se ha desplazado hacia arriba, pues así concuerda mejor con el pseudocódigo del algoritmo.
- Se ha corregido la Tabla 3.25: era errónea desde el paso 10.
- En el segundo párrafo de la Sección 3.15 faltaban las frases: `bad` y `do`, por lo que se han añadido en el lugar correspondiente.
- Se ha corregido la Figura 3.28a-I: La cadena que se pasa a la función hash es "`xy`", y no "`ya`".
- En la Sección 3.19 se ha modificado la url para obtener el formato GIF 89a por una válida —en el momento de la escritura del texto—:

  http://www.w3.org/Graphics/GIF/spec-gif89a.txt
  en lugar de http://delcano.mit.edu/info/gif.txt.

- En la Sección 3.20 se ha completado algún pequeño detalle sobre las características de RAR y WinRAR.
- La Sección 3.21 se ha adaptado al castellano, cambiando los nombres de los códigos especiales. Asimismo se proporciona una dirección en la cual se puede obtener el documento correspondiente al protocolo V.42bis.
- En el primer párrafo de la Sección 3.23, se ha corregido el significado del acrónimo PDF: éste procede de *Portable Document Format* y no de *Portable Document File*.
  Se ha fijado el orden del par en (*longitud*, *desplazamiento*) en todas las ocasiones (aunque, en principio, el orden podría ser cualquiera —siempre y cuando el codificador y el decodificador estén de acuerdo—.)
- En la Sección 3.24 se ha incluido una referencia a *FAR Manager*.
- En la Sección 3.28 se ha añadido una nota al pie con una breve aclaración de la Tabla 3.41; asímismo, se ha modificado lo que podría ser una errata: la expresión del polinomio generador $CRC_{12}(x)$, aunque se mantiene como nota al pie la usada en el libro original —en inglés—.
- En la Sección 3.31 se ha cambiado la Figura 3.42c por otra más clara —con las probabilidades ausentes en cada nodo, pero entre paréntesis—, asímismo, se ha modificado el texto acorde con este cambio.
- En la Subsección 4.2.1 —en el párrafo anterior al Ejercicio 4.3— se ha corregido una errata: el bit k cambian cada 2^k (no cada k) enteros. Asimismo, la que antes era la Tabla 4.12, ahora es la Figura 4.12; además se ha corregido una pequeña errata en la misma: el valor binario 21 estaba repetido (uno debe ser 20).
- En la Figura 4.35 de la Subsección 4.6.2 se ha modificado lo que parece una errata: ambos valores $-0{,}5$ de los ejes deben ser 0,5.
- En la Subsección 4.9.1 se ha corregido una errata en la expresión única en C para el cálculo del parámetro k; sobra un paréntesis, la expresión correcta es:

$$\texttt{for}\,(\texttt{k} = 0;\ (\texttt{N}\,[\texttt{Q}] << \texttt{k}) < \texttt{A}\,[\texttt{Q}]\,;\ \texttt{k}++)\,;$$

- En la Subsección 4.11.1, en el texto original (en inglés), resulta que al parámetro `SEQ` de JBIG se le llama `SET`. Todos los `SET` se han cambiado a `SEQ`, que es el nombre más habitual en la literarura sobre JBIG. Asimismo, el texto antes inherente a la Figura 4.91, ahora es el párrafo anterior al que empieza a hacer referencia a la Figura 4.91b.
- En el quinto párrafo de la Sección 4.14 se ha corregido Tera (10^{12}) por Giga (10^9), ya que $256^4 = 2^{32} = 4\,294\,967\,2964$ (unos cuatro mil millones). El primer cálculo de la expresión de la fila IV, en la Tabla 4.105 es erróneo: el resultado es 949; se ha corregido tanto este valor como el cálculo de $D^{(0)}$, que dependía de él; éste a su vez es mencionado en la solución del Ejercicio 4.24 , que ahora hace referencia al valor correcto.
- En la Figura 4.126b de la Subsección 4.21.2 se han modificado los pesos centrales, de 0,3164 a 0,3163, para que la ponderación sea correcta y los pesos sumen 1.
- En la Figura 4.160 se ha corregido el número ubicado en la primera columna (desde la izquierda), tercera fila (desde abajo): el valor correcto es 193. En la Figura 4.161 se ha corregido el mismo valor, pero esta vez está localizado en la esquina inferior izquierda: donde ponía 192, ahora pone 193.

- En la subsección 4.21.6, en la solución de **b**, se ha corregido el segundo término, ya que debe comenzar por 9 (primer elemento de la segunda fila de la matriz **N**), no por 19.
- En la Sección 5.2 se ha añadido las Figuras 5.5c y 5.6b, que muestran una representación en 3D de las tres sumas parciales de la Figura 5.5b, y de las cuatro sumas parciales de la Figura 5.6a, respectivamente.
- En el último párrafo de la Subsección 5.11.3 se ha corregido el número de coeficientes de filtro para la interpolación cúbica (son siete y no ocho).
- En la Sección 5.13, tras el párrafo donde se define la *contribución de igualdad* se ha corrregido $\omega(0) = a$ por $\hat{\omega}(0) = a$. Así mismo tras el párrafo en el que se establece $a = 0{,}6$, las expresiones de $\hat{\omega}(-2) = \hat{\omega}(2)$ y $\hat{\omega}(-1) = \hat{\omega}(1)$ estaban intercambiadas.
- En la Subsección 5.14.4, La lista LIP mostrada al final me pareció ser una errata: ponía LIP = $\{(2,1),(3,4),(3,4),(4,4)\}$; ahora es LIP = $\{(2,1),(2,4),(3,4),(4,4)\}$.
- Al final del párrafo que precede a la Ecuación (6.2), en la Subsección 6.5.1, aparentemente "quantization tule" debería ser "quantization rule", por lo que se ha traducido como "regla de cuantificación".
- En el penúltimo párrafo de la Subsección 6.7.1 se ha corregido lo que parece una errata: donde ponía "... 12 GPBs", ahora pone "...12 GOBs".
- En las Figuras 6.38a (de abajo) y 6.38b (de arriba), se ha corregido el valor 288 por el correcto: $^{288}/_{2} = 144$.
- En la Sección 6.8, predicción intra, párrafo sobre el modo 5, la expresión de $p(x,y)$ cuando z es 1, 3 ó 5 parece una errata: antes ponía $^{p\left[\left(x-\frac{y}{2}-2,-1\right)+2p\left(x-\frac{y}{2}-1,-1\right)+p\left(x-\frac{y}{2},-1\right)+2\right]}/_{4}$ y ahora pone $^{\left[p\left(x-\frac{y}{2}-2,-1\right)+2p\left(x-\frac{y}{2}-1,-1\right)+p\left(x-\frac{y}{2},-1\right)+2\right]}/_{4}$.
- En el penútimo párrafo de la Sección 7.4 se ha corregido la conversión de 10 en base 16 a decimal. El valor correcto es $\mathtt{10}_{16} = 16_{10}$.
- En el párrafo referente al filtrado de la Subsección 7.15.1, la palabra "consecurive" parece una errata de *consecutive* y ha sido traducida como "consecutivas". Así mismo, en el párrafo previo a "modelado del ruido temporal (*Temporal Noise Shaping* o TNS)" de la misma subsección la palabra "correpsonding" también parece una errata de *corresponding*, por lo que ha sido traducida como "correspondiente".
- En el último párrafo de la Subsección 7.15.3, las palabras "was performed" parecen una errata de *was performed* por lo que se han traducido como "se llevó a cabo".
- En el cuarto párrafo de la Sección 8.8, en el cálculo de las entropías de cada estado se ha corregido la identificación del cálculo de la entropía del estado 2 (antes ponía en ambas líneas "estado 1").
- En la Sección 8.12 se ha añadido un enlace a una dirección de internet estable donde se puede encontrar la documentación de la última versión del estándar Unicode.
- En el Apéndice A (Sección A.1), se ha corregido una errata: el código de "a" se obtiene a partir del de "A" cambiando el sexto bit (y no el séptimo).
- En la Tabla Sol.5, se ha corregido la columna Pasos de las filas 2, 3 y 6: se han añadido los valores 1 y 0 del final —en las filas 2 y 3— y se ha cambiado el 0 final de la fila 6 por un 1.

- En la Figura Sol.6 se han dejado nada más los árboles a, b y c. La Figura Sol.7 sólo contiene el cuarto árbol —con más información— pues los demás son una repetición de la figura anterior.
- La Figura Sol.9 es similar a la original, pero más completa y algo más artística.
- El token —en el paso 20 de la Tabla Sol.23— se ha corregido por ser erróneo: el correcto es (16,"`a`").
- En Tabla Sol.25, se ha cambiado el valor 32 (`w`) por el correcto: 32 (␣) —el carácter ␣ denota un espacio—.
- Al final del párrafo correspondiente al paso 8 —en la respuesta al Ejercicio 3.12— se ha corregido la inicialización de I: la correcta es $I = \mathtt{s}$.
- En la Tabla Sol.27, se han corregido: las columnas `S` y `S'` de la fila 6 —donde ponía `a`, ahora pone `d`—, y la columna *Añadir al diccionario* de la última fila —el valor correcto es 267-`oo`—.
- En la respuesta al Ejercicio 4.42 , se ha corregido una errata: en la expresión de **b**, se ha cambiado 19 por 9.
- En la respuesta al Ejercicio 4.43, se ha corregido una errata: el elemento de la primera fila, segunda columna, de la matriz base debe ser $\frac{1}{-\alpha^2+\alpha^3+\alpha\beta-\alpha^2\beta}$.
- En la respuesta al Ejercicio 4.75 se ha traducido correctamente una errata: donde ponía *pervious* debería poner *previous*. Asimismo, en la Figura Sol.46b se ha corregido otra errata: las flechas deben ir desde los estados 4 y 3, al estado 5.
- En la respuesta al Ejercicio 5.16 se ha corregido el orden de las dimensiones de la imagen (M_C y M_R estaban intercambiadas). Ahora la primera cifra de cada dimensión se corresponde con las filas, y la segunda con las columnas de la imagen.
- La respuesta al Ejercicio 6.7 ha sido adaptada a la realidad de la Figura 6.17, que consta de 18×17 macrobloques, y no 18×18, como se indicaba en la respuesta.
- En el párrafo siguiente al Ejercicio H.14 del Apéndice H, se ha corregido el número de clases ennegrecidas referente a la Figura H.23b, a saber, donde antes ponía 10, ahora pone 11.
- En el Apéndice I, Sección 2.1 se ha corregido la expresión de la inversa de la matriz de transformación de 3×3 (utilizada en la Sección 4.35.1). En la Sección 3 se ha corregido la expresión del $\sin(\alpha/2)$. En la Sección 4.1 se ha corregido la expresión de $(\mathbf{PR})$ en el producto triple $(\mathbf{P} \bullet \mathbf{Q})\,\mathbf{R}$. En la Sección I.6 se ha corregido la referencia a la Figura I.16, la cual es ahora I.3.
- La traducción de la bibliografía carecía de sentido, por lo que es casi igual a la original —en inglés—. Se han añadido algunas alternativas que solventan los *enlaces rotos* de algunas URLS de la bibliografía. Se han conservado las URLS originales —pues podrían proporcionar pistas sobre qué buscar—; y las nuevas, están actualizadas a fecha de diciembre de 2011. Estas son las referencias a la bibliografía con cambios:

[3R 06], [ATSC 06 y 10], [ATT 96], [Blackstock 87 y 89], [BOCU 01], [Brandenburg 99], [CREW 00], [Fenwick 96a], [H264PaperIR 06 y 10], [Haffner et al. 98], [Howard y Vitter 92a], [Howard y Vitter 92b], [Howard y Vitter 92c], [Howard y Vitter 93], [Howard y Vitter 94a], [Howard y Vitter 94b], [ITU-R/BS1116 97], [ITU-T 89], [ITU-T 90], [ITU-T 94], [ITU-T264 02], [JBIG], [JBIG2 03], [Kieffer et al. 96a,b 11], [Korn et al. 02],

[Liebchen et al. 05], [Ogg squish 06], [Okumura 98], [Percival 03b], [rarlab 06], [RFC1952 96], [RFC1962 96], [RFC2616 99], [Rice 79], [Rice 91], [Richardson 03], [Robinson 94], [Schindler 98], [sighted 06], [Softsound 03], [Sweldens y Schröder 96], [Wolff 99], [XMill 03],

- Las siguientes referencias son nuevas entradas bibliográficas:

[Stollnitz et al. 95]

Este libro se terminó de traducir (en realidad, de reescribir) el 27 de agosto de 2013.

ESPAÑA, Alba de Tormes (Salamanca) David Herrera Pérez

Prefacio a la cuarta edición

Quedé gratamente sorprendido cuando en Noviembre de 2005 me llegó un mensaje de Wayne Wheeler, el nuevo editor de computer science de Springer Verlag, notificandome que tenía la intención de calificar este libro como una de las principales obras de consulta de Springer (MRW[2]), eliminando de este modo las restricciones pasadas en el número de páginas, liberándome de la obligación de tener mi propio estilo de compresión, y haciendo posible la introducción de importantes e interesantes métodos de compresión que, o bien fueron ignorados o bien sólo mencionados en las anteriores ediciones.

> Estos fascículos representan mi mejor intento de escribir una historia secundaria; pero la ciencia de la computación ha crecido tanto, que no tengo esperanzas de ser una autoridad en todo el material que cubre este libro. Por eso necesitaré los comentarios de mis lectores para posteriormente, preparar los volúmenes oficiales.
> Intento aprender exhaustivamente ciertas áreas de la ciencia de la computación; luego trato de convertir dicho conocimiento en una forma accesible para aquellos que no tengan tanto tiempo para estudiar.
>
> —Donald E. Knuth, `http://www-cs-faculty.stanford.edu/~knuth/` (2006).

Naturalmente, todos los errores descubiertos por mi y mis lectores en la tercera edición han sido corregidos. Muchas gracias a todos aquellos que se molestaron en enviar los errores, correcciones, preguntas y comentarios. También he revisado el libro completo y he realizado numerosas adiciones, correcciones y mejoras. En suma, en esta edición, se han incluido los temas nuevos siguientes:

- Códigos Tunstall (Sección 2.4). La ventaja de los códigos de tamaño variable es bien conocida por los lectores de este libro, pero estos códigos tienen también un inconveniente: son difíciles de procesar. El codificador debe acumular y añadir varios de ellos en un pequeño buffer, esperar hasta llenar n bytes del mismo con bits de código (donde n debe ser al menos 1), escribir n bytes a la salida, desplazar n bytes en el buffer y mantener el rastro de la localización del último bit colocado en el buffer. El decodificador tiene que realizar el proceso inverso. La idea de los códigos Tunstall es construir un conjunto de códigos de tamaño fijo para la codificación de cada cadena de tamaño variable procedente de los símbolos de entrada. Como un apartado, se ha añadido también el código "pod" (Tabla 7.29 —pág. 758—).

- La reducción recursiva del rango[3] (3R) (Sección 1.7) es un sencillo algoritmo de codificación debido a Yann Guidon que ofrece una compresión decente; es fácil de programar y su rendimiento es independiente de la cantidad de datos a comprimir.

- LZARI, por Haruhiko Okumura (Sección 3.4.1), es una mejora de LZSS.

- RAR (Sección 3.20). El popular software RAR, es la creación de Eugene Roshal. RAR tiene dos modos de compresión, general y especial. El modo general emplea un algoritmo basado en LZSS similar a la deflación ZIP. El tamaño del diccionario usado en RAR puede variar desde 64Kb hasta 4Mb (por defecto 4Mb), y la longitud mínima es 2. Literales, desplazamientos y longitudes de patrones de coincidencia, se comprimen mediante un código Huffman. Una característica

[2] Major Reference Work.
[3] Aprovecha el hecho de que los números más pequeños requieren menos bits para ser representados.

importante de RAR es un código de control de errores que incrementa la fiabilidad de los archivos RAR mientras son transmitidos o almacenados.

- 7-z y LZMA (Sección 3.24). LZMA es el principal (también por defecto) algoritmo utilizado en el popular software de compresión 7z (ó 7-Zip) [7z 06]. Tanto 7z como LZMA, son creaciones de Igor Pavlov. El software funciona bajo Windows y es libre. Ambos fueron diseñados para proporcionar una alta compresión, rápida descompresión y bajos requerimientos de memoria para la descompresión.
- Stephan Wolf contribuyó en la Sección 4.30.4.
- H.264 (Sección 6.8). H.264 es un avanzado codificador de vídeo desarrollado por la ISO y la ITU para reemplazar los estándares de compresión de vídeo existentes: H.261, H.262 y H.263. H.264 tiene los principales componentes de sus predecesores, pero ha sido extendido y perfeccionado. El único componente nuevo en el H.264 es un filtro (basado ondas pequeñas[4]), desarrollado específicamente para reducir artefactos producidos cuando bloques grandes individuales son comprimidos de forma separada. La Sección 7.4 está íntimamente relacionada con el formato de audio WAVE. WAVE (ó sencillamente Wave) es el formato de archivo nativo empleado por el sistema operativo Windows para amacenar datos de audio digital.
- FLAC (Sección 7.10). FLAC (compresión de audio sin pérdidas libre) es la creación de Josh Coalson quien lo desarrolló en 1999 basado en ideas de Shorten. FLAC fue especialmente diseñado para la compresión de audio y también soporta grabación y lectura en continuo[5](sin interrupciones) y archivos de datos de audio. Coalson comenzó el proyecto FLAC en el famoso sitio Web sourceforge [sourceforge.flac 06], dejando su implementación como referencia. Desde entonces, muchos desarrolladores han contribuído al perfeccionamiento de la misma y han escrito otras alternativas. El proyecto FLAC, administrado y coordinado por Josh Coalson, mantiene el software y proporciona un código de referencia y plugins para varios reproductores de audio populares.
- *WavPack* (Sección 7.11, escrita por David Bryant). *WavPack* [WavPack 06] es un algoritmo de compresión de audio multiplataforma completamente libre y un software que soporta tres modos de compresión: sin pérdidas, de alta calidad (con pérdidas) y un modo de compresión híbrido único. Maneja ejemplos de audio de enteros de 32 bits y también datos reales en punto flotante de 32 bits, siguiendo las normas de la IEEE [IEEE754 85]. La cadena de entrada se parte en bloques por el *WavPack* que puede ser mono o estereo, y generalmente de 0,5 segundos de longitud (aunque la duración actualmente es flexible). Los bloques pueden combinarse secuencialmente por el codificador para manejar cadenas de audio multicanal. El *WavPack* soporta todos los tipos de muestras de audio en todos sus modos.
- *Audio Monkey* (Sección 7.12). *Audio Monkey* es un algoritmo de compresión sin pérdidas rápido, eficiente, libre y con una implementación que ofrece detección de errores, marcado y soporte externo.
- MPEG-4 ALS (Sección 7.13). MPEG-4, Codificación de audio sin pérdidas[6] (ALS) es la última incorporación a la familia de los codificadores de audio MPEG-4. ALS puede introducir muestras de audio de punto flotante y está basado en una combinación de predicción lineal (tanto a corto[7] como a largo plazo[8]), codificación multicanal y en un eficiente codificador de residuos de audio

[4]Wavelet.
[5]Streaming.
[6]Audio Lossless Coding.
[7]Short-term.
[8]Longterm.

procedente de códigos Rice[9] y de bloque (el segundo se conoce también como códigos de bloque de Gilbert-Moore, o BGMC [Gilbert y Moore 59] y [Reznik 04]). Debido a esta estructura, ALS no se restringe a la codificación de señales de audio y puede comprimir eficientemente y sin pérdidas otros tipos de señales correlacionadas de tamaño fijo, como las médicas (ECG and EEG) y las referentes a datos sísmicos.

- AAC (Sección 7.15). AAC (codificación de audio avanzada), es una extensión de las tres capas de MPEG-1 y MPEG-2, por lo cual a menudo se llama *mp4*. Comenzó como una parte del proyecto MPEG-2 y fue más tarde ampliado incorporándose a MPEG-4. Los ordenadores Apple adoptaron AAC en 2003 para su utilización en su bien conocido iPod, por lo que muchos creen (erroneamente) que el acrónimo AAC representa al codificador de audio de apple.

- Dolby AC-3 (Sección 7.16). AC-3, también conocido como Dolby Digital, representa la tercera generación de codificadores de audio de Dolby. AC-3 es un codificador de audio de percepción basado en los mismos principios que las tres capas de MPEG-1/2 y AAC. La nueva sección incluída en esta edición concentra las características especiales de AC-3 y que lo distingue de otros codificadores de percepción.

- Formato de Documento Portátil (PDF, Sección 8.13). PDF es un estándar popular para la creación, edición e impresión de documentos independiente de cualquier plataforma[10]. Es semejante a un documento que puede incluir texto e imágenes (gráficos y fotos) y sus componentes se comprimen con algoritmos bien conocidos.

- Sección 8.14 (escrita por Giovanni Motta). Abarca un pequeño pero importante aspecto de la compresión de datos; a saber, cómo comprimir las diferencias entre dos archivos.

- La compresión de datos hiperespectral (Sección 8.15, parcialmente escrito por Giovanni Motta), es un campo relativamente nuevo y en vías de expansión. Los datos hiperespectrales son un conjunto de muestras (llamadas píxeles) ordenadas en filas y columnas en las que cada píxel es un vector. Una cámara digital enfoca luz visible en un sensor que genera una imagen. Confrontada, una cámara montada en un satélite espía (o un satélite en busca de minerales y otros recursos), recopila y mide la radiación de muchos rayos luminosos[11]. La intensidad de cada longitud de onda se convierte en un número y los todos esos números, vistos desde un punto de tierra, forman un vector que se convierte en un píxel de datos hiperespectral.

- Otro cambio agradable es la gran ayuda que he recibido de Giovanni Motta, David Bryant y Cosmin Truţa. Cada uno propuso temas para esta edición, revisando algo del material nuevo y realizó una crítica descriptiva. En suma, David escribió la Sección 7.11 y Giovanni escribió la Sección 8.14 y parte de la Sección 8.15.

- Me gustaría dar las gracias a las siguientes personas por dar información acerca de ciertos temas y por aclararme ciertos puntos. A Igor Pavlov por ayudarme con 7z y LZMA, Stephan Wolf por su contribución, a Matt Ashland por ayudarme con el audio *Monkey*, a Yann Guidon por su ayuda con la reducción recursiva del rango (3R), a Josh Coalson por ayudarme con FLAC y a Eugene Roshal por ayudarme con RAR.

[9] Desarrollado por Robert F. Rice.
[10] Funciona en cualquier ordenador, con cualquier sistema operativo.
[11] La palabra original es "wavelegths".

> En el primer volumen de esta biografía expresé mi gratitud a aquellas personas y corporaciones sin cuya ayuda y aliento no habría sido posible realizar esta empresa; y a aquellos otros que me ayudaron en un camino u otro avanzando sus progresos. Con la finalización de este volumen, mis obligaciones se han ampliado más. Me gustaría expresar o repetir mis agradecimientos a las siguientes personas, por la ayuda que me han prestado y los permisos que me han concedido:
> Christabel Lady Aberconway; Lord Annan; Dr. Igor Anrep; ...
>
> —Quentin Bell, *Virginia Woolf: A Biography* (1972) .

- Actualmente, el sitio Web del libro es parte del sitio Web del autor, el cual se encuentra en `http://www.ecs.csun.edu/~dsalomon/`. Se ha reservado el dominio http://davidsalomon.name/ y siempre señalará a la futura localización del sitio Web. La dirección email del autor es `dsalomon@csun.edu`, pero los email enviados a `<anyname>@DavidSalomon.name` serán reenviados al autor.
- Aquellos interesados en la compresión de datos en general deberían consultar la corta sección titulada "Unirse a la Comunidad sobre la compresión de datos[12]", al final de libro, así como las siguientes fuentes:
- `http://compression.ca/`,
- `http://www-isl.stanford.edu/~gray/iii.html`,
- `http://www.hn.is.uec.ac.jp/~arimura/compression_links.html`,y
- `http://datacompression.info/`.

(las URLs tienen una vida notoriamente corta; busque en Internet).

> La gente se equivoca al pensar que mi arte surge fácilmente.
>
> —Wolfgang Amadeus Mozart.

Lakeside, California David Salomon

[12]The Data Compression Community.

Prefacio a la tercera edición

Quedé gratamente sorprendido cuando en Diciembre de 2002 me llegó un mensaje del editor pidiéndome producir la tercera edidión del libro, proponiéndome una fecha límite de Abril de 2003. Tenía la esperanza de una tercera edición, principalmente porque el campo de la compresión de datos ha hecho grandes avances desde la publicación de la segunda edición, pero también por las siguientes razones:

Razón 1: Los muchos comentarios favorables de los lectores, de los que los siguientes son ejemplos típicos:

Primero quiero darle las gracias por escribir "Data Compression: The Complete Reference." Es un libro maravilloso y lo he usado como referencia principal.

Deseo añadir algo a la lista de erratas de la segunda edición; y si se me permite, me gustaría hacer algunos comentarios y sugerencias...

—Cosmin Truţa, 2002.

Señor,

Soy Ismail de la India. Soy ingeniero en ciencias de computación. Hice un proyecto sobre compresión de datos en el que se abre un archivo de texto; se obtienen las palabras claves (símbolos, alfabetos, números) —sin repetir—; luego se ordenan por el número de ocurrencias en el archivo de texto; se guardan en un archivo de texto y siguiendo a las palabras, se almacena un indicador 000. Entonces se lee el fichero de texto original; se toma el primer carácter del archivo; se obtiene el valor posicional del carácter en las palabras clave; luego se almacena la posición en binario. Si ese binario contiene un solo dígito, se le asigna un triple bit 000; al binario con dos dígitos, se le asigna el triple bit 001; por lo que para 256 ascii se necesita un máximo de 8 dígitos binarios, más un triple bit; así, la necesidad máxima para los 256 caracteres en las palabras claves es de 11 bits. Pero la necesidad mínima para la primera palabra clave es un bit + tres bits: cuatro bits. De modo que se escriben de forma continua ceros y unos en un archivo y luego, cada 8 bits se convierten en caracteres ascii para guardarlos en el mismo. De esta manera, almacenando *palabra clave + indicador + carácter ascii convertido* puedo obtener un archivo comprimido. Luego, con el proceso inverso se puede obtener el fichero original. Estas ideas son completamente mías.

(Véase la descripción en la Sección 3.2).

Razón 2: Los errores encontrados por mi y mis lectores en la segunda edición. Están listados en el sitio Web de la segunda edición y han sido corregidos en la tercera edición.

Razón 3: El título del libro (originalmente escogido por el editor). Este título tiene que ser justificado haciendo del libro una referencia completa. Como resultado, se han añadido nuevos métodos de compresión y material que quedó en segundo plano en esta edición del libro, aunque las descripciones de algunos de los métodos más antiguos y obsoletos, se han borrado o condensado. Las adiciones y los cambios más importantes son los siguientes:

- El formato de archivo de imagen BMP es originario del sistema operativo Windows de Microsoft. La nueva Sección 1.4.4 describe la versión sencilla de RLE utilizada para comprimir estos archivos.
- La Sección 2.5 del código Golomb se ha reescrito completamente para corregir errores del texto original. Estos códigos se utilizan en un nuevo método de compresión de imágenes adaptativo tratado en la Sección 4.22.

- Se ha añadido la Sección 2.9.6 para hacer una breve mención al algoritmo de compresión adaptativo de Huffman mejorado.
- El método de compresión sin pérdidas PPM —de la Sección 2.18—, produce resultados impresionantes, pero no se utiliza mucho en la práctica, debido a que es lento. Se ha dedicado mucho esfuerzo explorando caminos, para incrementar la velocidad de PPM o hacerlo más eficiente. Esta edición, presenta tres de dichos intentos: el método PPM*, de la Sección 2.18.6, PPMZ (Seccíon 2.18.7) y el PPM rápido, de la Sección 2.18.8. Los dos primeros, tratan de explorar el efecto de los contextos de longitud libre, y añaden otras mejoras al algoritmo básico PPM. El tercero, intenta incrementar la velocidad de PPM eliminando el uso de símbolos escape e introduciendo algunas aproximaciones. En suma, la Sección 2.18.4 se ha extendido y ahora contiene información concisa sobre las dos variantes de PPM, llamadas PPMP y PPMX.
- La Sección 3.2 es nueva; describe un sencillo método de compresión basado en diccionarios.
- El tema de la Sección 3.7 trata el método LZX, una variante de LZ77 para la compresión de archivos cabinet[13].
- La Sección 8.14.2 es una corta introducción a un interesante concepto de diferencia entre ficheros, en la que se codifican las diferencias entre el archivo original y su actualización.
- El popular método *Deflate* se trata ahora con mucho detalle en la Sección 3.23.
- El popular formato de archivo de gráficos PNG se describe en la nueva Sección 3.25.
- La Sección 3.26 es una corta descripción de XMill, un compresor de propósito general para archivos XML.
- Se ha reescrito la Sección 4.6, que trata sobre DCT. Ahora, describe DCT; presenta dos formas de interpretarlo. Muestra cómo se pueden simplificar los cálculos necesarios, enumera cuatro transformadas del coseno discretas distintas e incluye mucho material de fondo. Como consecuencia, la Sección 4.8.2 es bastante corta.
- Un N-árbol es una interesante estructura de datos (una extensión de quadtrees), cuya compresión se trata en la nueva Sección 4.30.4.
- La Sección 5.19, sobre JPEG 2000, ha sido actualizada.
- MPEG-4 es un estándar internacional que surgió para aplicaciones audiovisuales. Especifica procedimientos, formatos y utilidades para el contenido multimedia de autor, entregándolo y difundiéndolo (reproduciéndolo y mostrándolo). De esta manera, MPEG-4 es mucho más que un método de compresión. La Sección 6.6 es una corta descripción de las principales características y utilidades incluídas en MPEG-4.
- El nuevo método de compresión sin pérdidas estándar aprobado para DVD-A (audio) se llama MLP. Es el tema de la Sección 7.7. Este MLP no debería confundirse con el método de compresión de imágenes MLP de la Sección 4.21.
- Se ha añadido (Sección 7.9). *Shorten*, un sencillo algoritmo de compresión para datos de ondas en general y para el habla en particular.
- SCSU es un nuevo algoritmo de compresión, diseñado especificamente para comprimir archivos de texto Unicode. Este es el tema de la Sección 8.12. La corta Section 8.12.1 está dedicada a BOCU-1, un sencillo algoritmo para la compresión Unicode.

[13] Varios archivos concatenados en una misma unidad y comprimidos.

- Varias secciones que tratan sobre viejos algoritmos han sido arreglados o completamente eliminados por falta de espacio. La mayor parte de este material está disponible en el sitio Web del libro.
- Se han borrado todos los apéndices por motivo de espacio. Están disponibles libremente, en formato PDF, en el sitio Web del libro. los apéndices son: (1) el código ASCII (incluyendo caracteres de control); (2) curvas *space-filling*[14]; (3) estructuras de datos (incluyendo hashing[15]); (4) códigos de corrección de errores; (5) autómatas de estado finito —este tema se necesita para algunos métodos de compresión, como WFA, IFS y la codificación dinámica de Markov—; (6) elementos de probabilidad; y (7) polinomios de interpolación.
- Se ha eliminado la mayor parte de los ejercicios. Las respuestas a los ejercicios también se han borrado y están disponibles en el sitio Web del libro.

Me gustaría agradecer a Cosmin Truţa su interés, ayuda y ánimo. Gracias a él, esta edición es mejor de lo que habría sido. Gracias también a Martin Cohn y Giovanni Motta por sus excelentes revisiones anticipadas del libro. Diversos lectores han ayudado también señalando errores u omisiones de la segunda edición.

Actualmente el sitio Web del libro forma parte del sitio Web del autor, el cual está situado en `http://www.ecs.csun.edu/~dsalomon/`. Se ha reservado el dominio `BooksByDavidSalomon.com`, el cual siempre apuntará a cualquier futura localización del sitio Web. La dirección email del autor es `david.salomon@csun.edu`, pero se ha ordenado que los email enviados a `<anyname>@DavidSalomon.name` sean reenviados al autor.

Los lectores dispuestos a soportar ocho segundos de anuncios pueden ser dirigirse al sitio Web del libro a `http://welcome.to/data.compression`. El email enviado a data.compression@welcome.to será también remitido.

Aquellos interesados en la compresión de datos en general deberían consultar la corta sección titulada "Unirse a la Comunidad sobre la Compresión de datos", al final de libro, así como las siguientes fuentes:

- `http://compression.ca/,`
- `http://www-isl.stanford.edu/~gray/iii.html,`
- `http://www.hn.is.uec.ac.jp/~arimura/compression_links.html,y`
- `http://datacompression.info/.`

(las URLs tienen una vida notoriamente corta; busque en Internet).

> Una consecuencia de la decisión de tomar este rumbo es que estoy, por poner por escrito estas frases, en la extraña posición de redactar mi prefacio antes que el resto de mi relato. Estamos todos familiarizados con la actitud posterior a realizar algo —cansado, autojustificado, apenado, enfurecido— compartido por capitanes de barco que se presentan ante asambleas de preguntas para explicar cómo han hecho para hacer funcionar sus naves encalladas, y por autores componiendo prefacios.
>
> —John Lanchester, *The Debt to Pleasure* (1996)
> *(La deuda del placer)*

Northridge, California David Salomon

[14] De relleno de espacio.
[15] Método de cálculo de direccionamiento (dispersión).

Prefacio a la segunda edición

Esta segunda edición ha nacido por tres razones. La primera es, por los muchos comentarios de los lectores, uno de los cuales es el siguiente:

> Acabo de terminar de leer su libro sobre compresión de datos. Me ha encantado. Y como contiene muchos algoritmos en un volumen de 20 mm de grueso, ¡el libro por sí mismo es un buen ejemplo de compresión de datos!
>
> —Fred Veldmeijer, 1998.

La segunda razón es que había errores detectados por el autor y los lectores en la primera edición. Están listados en el sitio Web del libro (véase más abajo) y han sido corregidos en la segunda edición.

La tercera razón es el título del libro (originalmente escogido por el editor). Este título tiene que ser justificado haciendo del libro una referencia completa. Como resultado, se han añadido muchos métodos de compresión y material que quedó en un segundo plano en esta edición del libro. Las adiciones y los cambios más importantes son los siguientes:

- Se han añadido tres capítulos nuevos. El primero es el Capítulo 5, que trata sobre el relativamente joven (y desconocido) tema de las ondas y sus aplicaciones a la compresión de la imagen y el sonido. El capítulo comienza con una explicación intuitiva de las ondas, utilizando la transformada de ondas (*wavelet*) continua (*Continuous Wavelet Transform* o CWT). Continúa con un ejemplo detallado que muestra cómo se usa la trasformada de Haar para comprimir imágenes. A esto, le sigue una discusión general de bancos de filtros y la transformada de ondas (*wavelet*) discreta (*Discrete Wavelet Transform* o DWT) y una lista de coeficientes de onda de muchos filtros de onda comunes. El capítulo concluye con una descripción de importantes métodos de compresión que, o bien utilizan ondas o bien están basados en ondas. Incluída entre ellos, están la pirámide Laplaciana, los árboles jerárquicos para la partición de conjuntos (SPIHT o *Set Partitioning In Hierarchical Trees*), codificación embebida usando árboles-cero[16] (EZW), el método WSQ para la compresión de huellas dactilares y JPEG 2000, un nuevo y prometedor método para la compresión de imágenes fijas (Sección 5.19).

- El segundo capítulo nuevo, el Capítulo 6, trata sobre la compresión de vídeo. El capítulo comienza con una descripción general del funcionamiento del CRT y conceptos básicos del vídeo digital y analógico. Continúa con una discusión general sobre la compresión de video y concluye con una descripción de MPEG-1 y H.261.

- El tercer capítulo nuevo, trata sobre la compresión de audio; es el Capítulo 7. El primer tema de este capítulo expone las propiedades del sistema auditivo humano y cómo puede ser explotado para conseguir una compresión de audio con pérdidas. Le sigue una discusión sobre algunos métodos de compresión de audio y el capítulo concluye con una descripción de las capas de audio de MPEG-1, incluyendo el tan popular formato `mp3`.

También es nuevo el siguiente temario:

- Imagen condicional RLE (Sección 1.4.2).

- Cuantificación escalar (Sección 1.6).

[16] Embedded coding using Zerotrees.

- El codificador QM utilizado en JPEG, JPEG 2000 y JBIG, está ahora incluido en la Sección 2.16.
- El árbol ponderado de contexto se trata en la Sección 2.19. Su extensión para la compresión sin pérdidas de imágenes es el tema de la Sección 4.24.
- La Sección 3.4 trata un método de buffer de desplazamiento llamado "tiempos de repetición".
- También se ha incluido el molesto tema de las patentes (Sección 3.25).
- Los relativamente desconocidos códigos de Gray se tatan en la Sección 4.2.1, en relación con la compresión de imágenes.
- La Sección 4.3 muestra métodos intuitivos para la compresión de imágenes, como las submuestras y la cuantificación vectorial.
- En la Sección 4.4 se trata el importante concepto de las *tranformadas de imágenes*. Se describe en detalle la transformada discreta del coseno[17] (DCT). Se incluyen las transformadas de Karhunen-Loève, de Walsh-Hadamard y de Haar. La Sección 4.4.5 es un corto inciso, en el que se trata la transformada discreta del seno, una pobre y desconocida prima de la DCT.
- JPEG-LS, un nuevo estándar internacional para la compresión de imágenes sin pérdidas y que apenas tiene pérdidas, es el tema de la nueva Sección 4.7.
- JBIG2, otro nuevo estándar internacional, esta vez para la compresión de imágenes de bajo nivel, se encuentra en la Sección 4.10.
- La Sección 4.11 trata de EIDAC, un método para la compresión de imágenes simples. Su principal novedad es el uso de contextos dobles. El contexto *intra* de un píxel P, está formado por varios de sus vecinos cercanos en su plano de bits. El contexto *inter* de P, está constituido por píxeles que tienden a estar correlacionados con P, aunque se encuentren en planos de bits diferentes.
- Hay una nueva Sección 4.12 sobre cuantificación de vectores, seguida de secciones de cuantificación vectorial adaptativa y sobre codificación por truncamiento de bloques[18](BTC).
- El emparejamiento de bloques[19] es una adaptación de LZ77 (ventana deslizante) para la compresión de imágenes. Se puede encontrar en la Sección 4.14.
- El diferencial de modulación por impulsos codificados[20] (DPCM) se incluye ahora en la nueva Section 4.23.
- Un interesante método para la compresión de imágenes de tonos discretos es la descomposición de bloque (Sección 4.25).
- La Sección 4.26 trata de la codificación predictiva de árboles binarios[21] (BTPC).
- La compresión prefija de imágenes está relacionada con árboles cuaternarios (*quadtrees*). Este es el tema de la Sección 4.27.
- Otro método de compresión de imágenes relacionado con árboles cuaternarios es la *división cuatrisección* (*quadrisection*). Éste se trata, junto con sus parientes *bisección* (*bisection*) y *octasección* (*octasection*), en la Sección 4.28.

[17] Discrete Cosine Transform.
[18] Block Truncation Coding.
[19] Block matching.
[20] Differential Pulse Code Modulation.
[21] Binary Tree Predictive Coding.

- La sección WFA (Sección 4.31) estaba equivocada en la primera edición y se ha reescrito completamente con mucha ayuda de Karel Culik y Raghavendra Udupa.
- La codificación por celdas se incluye en la Sección 4.33.
- DjVu es un método poco corriente, trata de la compresión de documentos escaneados. Fue desarrollado por los laboratorios Bell (Tecnologías Lucent) y se describe en la Sección 5.17.
- El nuevo estándar JPEG 2000 para imágenes fijas se trata en la nueva Sección 5.19.
- La Sección 8.4 es una descripción de un método de similitud de contextos basados en el orden. Este método, utiliza el contexto de un símbolo, siguiendo un camino que recuerda a ACB. También asigna categorías a los símbolos, y sus propiedades lo relacionan con el método de Burrows-Wheeler y con la clasificación de símbolos.
- Se ha añadido la compresión mediante prefijos (prefija) de cadenas dispersas (poco densas) a la Sección 8.5.
- FHM es un método poco convencional para la compresión de curvas. Utiliza números de Fibonacci, codificación Huffman y cadenas de Markov; es el tema de la Sección 8.9.
- *Sequitur*, Sección 8.10, es un método especialmente apropiado para la compresión de texto semiestructurado. Está basado en gramáticas libres de contexto.
- La Sección 8.11 es una descripción detallada de *edgebreaker*, un método sumamente original para comprimir la información de las conexiones en una malla de triángulo. Este método y sus diversas extensiones, pueden llegar a ser el estándar para comprimir superficies de polígonos; uno de los tipos de superficie más comunes utilizados en gráficos por ordenador. *Edgebreaker* es un ejemplo de método de *compresión geométric*o.
- Se han eliminado todos los apéndices por razones de espacio. Están disponibles gratuitamente, en formato PDF, en el sitio Web del libro. Los apéndices son: (1) el código ASCII (incluyendo caracteres de control); (2) curvas *space-filling*[22]; (3) estructuras de datos (incluyendo hashing[23]); (4) códigos de corrección de errores; (5) autómatas de estado finito (este tema se necesita para algunos métodos de compresión, como WFA, IFS y la codificación dinámica de Markov); (6) elementos de probabilidad; y (7) polinomios de interpolación.
- Las respuestas a los ejercicios también se han eliminado y están disponibles en el sitio Web del libro.

Actualmente el sitio Web del libro forma parte del sitio Web del autor, el cual está situado en `http://www.ecs.csun.edu/~dsalomon/`. Se ha reservado el dominio `BooksByDavidSalomon.com`, el cual siempre apuntará a cualquier futura localización del sitio Web. La dirección email del autor es `david.salomon@csun.edu`, pero se ha ordenado que los email enviados a `anyname@DavidSalomon.name` sean reenviados al autor.

Los lectores dispuestos a soportar ocho segundos de anuncios pueden ser dirigirse al sitio Web del libro a `http://welcome.to/data.compression`. El email enviado a `data.compression@welcome.` será también remitido.

Aquellos interesados en la compresión de datos en general deberían consultar la corta sección titulada "Unirse a la Comunidad sobre la Compresión de datos", al final de libro, así como las dos URLs:

[22] De relleno de espacio.
[23] Un método de cálculo de direccionamiento.

`http://www.internz.com/compression-pointers.html` y
`http://www.hn.is.uec.ac.jp/~arimura/compression_links.html`.

Northridge, California David Salomon

Prefacio a la primera edición

Historicamente, la compresión de datos no ha sido uno de los campos de la informática. Parece que los trabajadores en este campo han necesitado los primeros 20 a 25 años para recopilar y procesar datos suficientes, antes de darse cuenta de su necesidad. Hoy, cuando el campo de la computación tiene cerca de 50 años, la compresión de datos es un importante y activo campo, tanto como los grandes negocios. Quizás la mejor prueba es la popularidad de la "*Conferencia sobre Compresión de Datos*[24] (DCC, véase el final del libro).

En muy poco tiempo, muchas personas han desarrollado, principios, técnicas y algoritmos para comprimir diferentes tipos de datos, basándose en conceptos prestados de disciplinas tan variadas como estadísticas, autómatas de estado finito, curvas *space-filling*, y transformadas —de Fourier y otras—. Esta tendencia, naturalmente lleva a la publicación de muchos libros sobre el tema, que plantean la cuestión: ¿Por qué otro libro sobre la compresión de datos?.

La respuesta obvia es: porque el campo es grande y crece continuamente; así, en los últimos años se ha producido la "creación" de más lectores potenciales, y la revisión y corrección de los textos existentes, ya obsoletos.

La razón principal para escribir este libro ha sido proporcionar una presentación clara de los principios de compresión de datos y todos los métodos importantes, actualmente en uso; y una presentación adaptada a los no especialistas. La intención del autor es tener descripciones y discusiones fáciles de comprender, por cualquiera que tenga alguna formación en el uso y funcionamiento de los computadores. Como resultado, las referencias matemáticas se mantienen al mínimo y el material se presenta con ejemplos, diagramas y ejercicios. En vez de tratar de ser rigurosos y probar cada afirmación, el texto muchas veces dice "nótese que..." o "se puede demostrar que..." .

Los ejercicios son una característica del libro especialmente importante. Complementan la materia, y deben ser resueltos por cualquiera que esté interesado en una comprensión completa, de la compresión de datos y de los métodos aquí descritos. Se proporcionan casi todas las respuestas (en la página Web del libro); pero el lector, evidentemente, deberá tratar de trabajar en cada ejercicio antes de consultar la respuesta.

[24] Data Compression Conference.

Agradecimientos

Me gustaría agradecer especialmente a Nelson Beebe, que fue meticuloso durante todo el texto de la primera edición e hizo numerosas correcciones y sugerencias. Muchas gracias también a Christopher M. Brislawn, que revisó la Sección 5.18 y nos dio permiso para utilizar la Figura 5.64; a Karel Culik y Raghavendra Udupa, por su considerable ayuda con el autómata finito ponderado[25] (WFA); a Jeffrey Gilbert, quien repasó la Sección 4.28 (descomposición de bloque); a John A. Robinson, quien revisó la Sección 4.29 (codificación predictiva con árboles binarios); a Øyvind Strømme, quien revisó la Sección 5.10; a Frans Willems y Tjalling J. Tjalkins, quienes revisaron la Sección 2.19 (ponderación del árbol de contexto); y a Hidetoshi Yokoo, por su ayuda con las Secciones 3.17 y 8.4.

Al autor le gustaría también dar las gracias a Paul Amer, Guy Blelloch, Mark Doyle, Hans Hagen, Emilio Millan, Haruhiko Okumura y Vijayakumaran Saravanan, por su ayuda con los errores.

El traductor agradece a Miguel Hernández Cabronero el tiempo que dedicó a revisar la traducción. Él encontró numerosas erratas e indicó algunas mejoras en la traducción que han saneado la traduccion original del libro.

Parece ser, que tenemos una fascinación natural con la reducción y la ampliación de los objetos. Puesto que nuestra capacidad práctica a este respecto es muy limitada, nos gusta leer historias donde la gente y los objetos cambian drásticamente su tamaño natural. Como ejemplos, tenemos *Los Viajes de Gulliver* de Jonathan Swift (1726), *Alicia en el País de las Maravillas* de Lewis Carroll (1865) y *Fantastic Voyage* (*Viaje Alucinante*) de Isaac Asimov (1966).

Viaje Alucinante comenzó como un guión escrito por el famoso escritor Isaac Asimov. Mientras la película se estaba produciendo (su fecha de lanzamiento fue en 1966), Asimov la reescribió como una novela, corrigiendo en el proceso algunos de los defectos más evidentes del guión. La trama se refiere a un grupo de científicos médicos que se meten en un submarino, reducido a dimensiones microscópicas. Luego, se inyectan en el cuerpo de un paciente en un intento de eliminar un coágulo de sangre de su cerebro por medio de un haz de láser. El hecho es, que el paciente, el Dr. Benes, es el científico que mejora el proceso de miniaturización, y el primero que lo lleva a la práctica.

Debido tanto al éxito de la película como del libro, Asimov escribió más tarde Viaje Alucinante II: Destino, el Cerebro; pero esta última novela fue un fracaso.

Pero antes de continuar, hay una cuestión que puede que usted ya se haya planteado: "De acuerdo, pero ¿por qué debería estar interesado en la compresión de datos?". Muy sencillo: "¡ LA COMPRESIÓN DE DATOS LE AHORRA DINERO !".

¿Más interesado ahora?. Pensamos que debería estarlo.
Déjenos darle un ejemplo de la aplicación de la compresión de datos
que usted ve a diario: el intercambio diario de faxes...
de `http://www.rasip.etf.hr/research/compress/index.html`

Northridge, California David Salomon.

[25] Weighted Finite Automata.

Parte I

Compresión de datos

Introducción

Giambattista della Porta, un científico del Renacimiento a veces conocido como el profesor de los secretos, fue el autor, en 1558 de Magia Naturalis (Magia Natural), un libro en el que se tratan algunos temas, incluyendo la demonología, el magnetismo y la cámara oscura [della Porta 58]. El libro se volvió tremendamente popular en el siglo XVI y formó parte de más de 50 ediciones, en varios idiomas, además del Latín. El libro menciona un dispositivo imaginario, que desde entonces se conoce como "telégrafo receptivo"[26]. Este dispositivo constaba de dos cajas circulares, similares a las brújulas, cada una con una aguja magnética. Cada caja estaba etiquetada con 26 letras, en lugar de lo habitual, y lo más importante era, que las dos agujas estaban supuestamente magnetizadas por el mismo imán. Porta asumió que esto podría de alguna manera coordinar las agujas de forma que, cuando se marcaba una letra en una caja, la aguja giraría en la otra caja para apuntar a la misma letra.

Huelga decir que tal dispositivo no funciona (esto, después de todo, ocurría unos 300 años antes de Samuel Morse); pero en 1711 una mujer preocupada escribió al *Spectator*, un periódico de Londres, pidiendo consejo sobre cómo llevar las largas ausencias de su amado marido. El asesor, Joseph Addison, aportó algunas ideas prácticas; a continuación mencionó el dispositivo de Porta y añadió que un par de esas cajas podrían permitirla a ella y a su marido comunicarse entre sí, incluso cuando "estaban vigilados por espías y dispositivos de escucha, o separados por los castillos y aventuras". El señor Addison entonces añadió que, además de las 26 letras, los discos del telégrafo receptivo deberían contener, cuando se utilice por amantes, "algunas palabras enteras, que siempre tienen lugar en las epístolas apasionadas". En tal caso, el mensaje "*I love you*", por ejemplo, requeriría el envío de sólo tres símbolos, en lugar de diez.

> Una mujer rara vez pide consejo, antes de haberse comprado su ropa de boda.
>
> —Joseph Addison.

Esta sugerencia es un ejemplo anticipado del alcance de la *compresión de texto* utilizando códigos cortos para los mensajes más comunes y códigos más largos para el resto. Aún más importante, esto muestra cómo el concepto de la compresión de datos es algo natural para las personas que están interesadas en las comunicaciones. Parece que estamos preprogramados con la idea de enviar datos lo más pequeños posible con el fin de ahorrar tiempo.

La compresión de datos es el proceso de convertir una cadena de datos de entrada (la cadena fuente o los datos originales a tratar), en otra cadena de datos (la salida, la cadena de bits, o la cadena comprimida), que tenga un tamaño más pequeño. Una cadena es, o un archivo o un buffer en la memoria. La compresión de datos es popular por dos razones: (1) A la gente le gusta acumular datos y odia tirar cualquier cosa. No importa cuánta capacidad tenga un dispositivo de almacenamiento, tarde o temprano va a desbordarse. La compresión de datos, parece útil, ya que retrasa lo inevitable. (2) La gente odia esperar mucho tiempo en las transferencias de datos. Cuando se está sentado frente al ordenador, esperando la carga de una página Web o la llegada de un archivo, sentimos de forma natural, que cualquier cosa que dure más que unos pocos segundos, es un largo tiempo de espera.

El campo de la compresión de datos se llama, a menudo, *source coding* (codificación de fuente). Suponemos que la entrada de símbolos (como bits, códigos ASCII, bytes, ejemplos de audio, o valores pixel), es emitida por una cierta fuente de información y tiene que codificarse antes de ser enviada a su destino. La fuente puede no tener memoria (*memoyless*), o puede tenerla. En el primer caso, cada símbolo es independiente de sus predecesores. En el último, cada símbolo depende de algunos de sus

[26] Sympathetic telegraph.

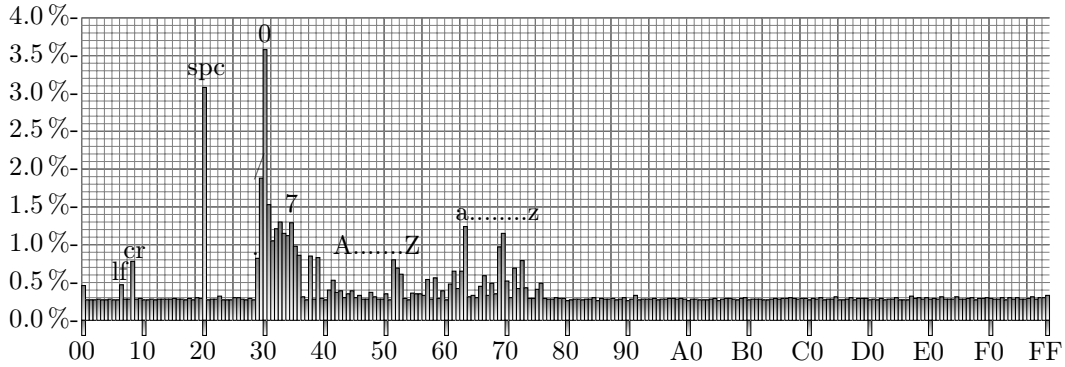


Figura Intr.0: Histograma de *data.compression.complete.reference.4th.pdf*.

predecesores, y tal vez, también de sus sucesores, por lo que están correlacionados. Una fuente sin memoria se denomina también "independiente e idénticamente distribuida" o IID.

La compresión de datos se ha consolidado en los últimos 20 años. Tanto la cantidad como la calidad de la literatura en este campo, proporciona amplias pruebas de ello. Sin embargo, la necesidad de la compresión de datos se ha sentido en el pasado, incluso antes de la llegada de las computadoras, como sugiere la siguiente cita:

> He hecho, esta carta es más larga de lo habitual, porque no tengo tiempo para hacerla más corta.
>
> —Blaise Pascal.

Hay muchos métodos conocidos para la compresión de datos. Están basados en ideas diferentes, apropiadas para distintos tipos de datos, y producen diferentes resultados; pero todos se basan en el mismo principio, a saber, comprimen eliminando la *redundancia* de los datos originales del archivo de fuente. Cualquier dato no aleatorio posee alguna estructura, la cual puede ser aprovechada para lograr una representación más pequeña de los datos; una representación, donde no hay ninguna estructura aparente. Los términos *redundancia* y *estructura* se utilizan en la literatura profesional, así como *suavidad*, *coherencia* y *correlación*; todos ellos se *refieren* a lo mismo. Así, la redundancia es un concepto clave en cualquier discusión sobre la compresión de datos.

◇ **Ejercicio Intro.1 (sol. en pág. 1051):** (Diversión) Encuéntrense palabras en inglés que contengan las 5 vocales "aeiou" en su orden original.

En un texto típico en inglés, por ejemplo, la letra E es muy frecuente, mientras que la Z, aparece raramente (Tablas Intr.1 e Intr.2). Esto se llama redundancia alfabética, y sugiere la asignación de códigos de tamaño variable a las letras; asociando con E, el código más corto, y con Z, el más largo. Otro tipo de redundancia, la redundancia de contexto, se ilustra por el hecho de que la letra Q, es casi siempre previa a la letra U (i.e., que ciertos digramas y trigramas son más comunes en inglés corriente que otros). La redundancia en las imágenes se ilustra por el hecho de que, en una imagen no aleatoria, píxeles adyacentes tienden a tener colores similares.

La Sección 2.1 trata sobre la teoría de la información, y presenta una definición rigurosa de la redundancia. Sin embargo, incluso sin una definición precisa de este término, es intuitivamente claro,

Letra	**Frecuencia**	**Probabilidad**	**Letra**	**Frecuencia**	**Probabilidad**
A	51060	0,0721	E	86744	0,1224
B	17023	0,0240	T	64364	0,0908
C	27937	0,0394	I	55187	0,0779
D	26336	0,0372	S	51576	0,0728
E	86744	0,1224	A	51060	0,0721
F	19302	0,0272	O	48277	0,0681
G	12640	0,0178	N	45212	0,0638
H	31853	0,0449	R	45204	0,0638
I	55187	0,0779	H	31853	0,0449
J	923	0,0013	L	30201	0,0426
K	3812	0,0054	C	27937	0,0394
L	30201	0,0426	D	26336	0,0372
M	20002	0,0282	P	20572	0,0290
N	45212	0,0638	M	20002	0,0282
O	48277	0,0681	F	19302	0,0272
P	20572	0,0290	B	17023	0,0240
Q	1611	0,0023	U	16687	0,0235
R	45204	0,0638	G	12640	0,0178
S	51576	0,0728	W	9244	0,0130
T	64364	0,0908	Y	8953	0,0126
U	16687	0,0235	V	6640	0,0094
V	6640	0,0094	X	5465	0,0077
W	9244	0,0130	K	3812	0,0054
X	5465	0,0077	Z	1847	0,0026
Y	8953	0,0126	Q	1611	0,0023
Z	1847	0,0026	J	923	0,0013

Frecuencias y probabilidades de las 26 letras en una edición previa de este libro

Tabla Intr.1: Probabilidades de las letras en inglés.

que un código de tamaño variable, tiene menos redundancia que un código de tamaño fijo (o sin redundancia en absoluto). Los códigos de tamaño fijo, hacen que sea más fácil trabajar con texto, por lo que son útiles, pero son redundantes.

La mayor parte, pero no todos los expertos, están de acuerdo en que las letras más corrientes en inglés —en orden— son ETAOINSHRDLU (normalmente escritas como dos palabras separadas: ETAOIN y SHRDLU). No obstante, [Fang 66] presenta un punto de vista diferente; los digramas más comunes (combinaciones de 2 letras) son: TH, HE, AN, IN, HA, OR, ND, RE, ER, ET, EA y OU. Las letras más frecuentes, al principio de las palabras, son: S, P y C, y las letras finales más frecuentes, son: E, Y y S. Las 11 letras más corrientes, en Francés, son: ESARTUNILOC.

La idea de comprimir reduciendo la redundancia, sugiere la ley general de la compresión de datos, que consiste en "asignar códigos cortos para los eventos comunes (símbolos o frases) y códigos largos para los eventos raros". Hay muchas maneras de aplicar esta ley y un análisis de cualquier método de compresión muestra que, en el fondo, funciona obedeciendo a la ley general.

La compresión de datos se realiza mediante el cambio de su representación, de ineficiente (i.e., larga) a eficiente (corta). Por lo tanto, la compresión es posible, sólo porque normalmente los datos se representan en la computadora en un formato más largo, que el absolutamente necesario. La razón de que esas ineficientes (en longitud) representaciones de datos se utilizen continuamente, es que hacen más fácil procesar los datos, y el procesamiento de los datos es más común e importante que

C.	**Frec.**	**Prob.**	**C.**	**Frec.**	**Prob.**	**C.**	**Frec.**	**Prob.**
e	85537	0,099293	x	5238	0,006080	F	1192	0,001384
t	60636	0,070387	|	4328	0,005024	H	993	0,001153
i	53012	0,061537	-	4029	0,004677	B	974	0,001131
s	49705	0,057698	)	3936	0,004569	W	971	0,001127
a	49008	0,056889	(	3894	0,004520	+	923	0,001071
o	47874	0,055573	T	3728	0,004328	!	895	0,001039
n	44527	0,051688	k	3637	0,004222	#	856	0,000994
r	44387	0,051525	3	2907	0,003374	D	836	0,000970
h	30860	0,035823	4	2582	0,002997	R	817	0,000948
l	28710	0,033327	5	2501	0,002903	M	805	0,000934
c	26041	0,030229	6	2190	0,002542	;	761	0,000883
d	25500	0,029601	I	2175	0,002525	/	698	0,000810
m	19197	0,022284	^	2143	0,002488	N	685	0,000795
\	19140	0,022218	:	2132	0,002475	G	566	0,000657
p	19055	0,022119	A	2052	0,002382	j	508	0,000590
f	18110	0,021022	9	1953	0,002267	@	460	0,000534
u	16463	0,019111	[	1921	0,002230	Z	417	0,000484
b	16049	0,018630	C	1896	0,002201	J	415	0,000482
.	12864	0,014933	]	1881	0,002183	O	403	0,000468
1	12335	0,014319	’	1876	0,002178	V	261	0,000303
g	12074	0,014016	S	1871	0,002172	X	227	0,000264
0	10866	0,012613	_	1808	0,002099	U	224	0,000260
,	9919	0,011514	7	1780	0,002066	?	177	0,000205
&	8969	0,010411	8	1717	0,001993	K	175	0,000203
y	8796	0,010211	‘	1577	0,001831	%	160	0,000186
w	8273	0,009603	=	1566	0,001818	Y	157	0,000182
$	7659	0,008891	P	1517	0,001761	Q	141	0,000164
}	6676	0,007750	L	1491	0,001731	>	137	0,000159
{	6676	0,007750	q	1470	0,001706	*	120	0,000139
v	6379	0,007405	z	1430	0,001660	<	99	0,000115
2	5671	0,006583	E	1207	0,001401	”	8	0,000009

Frecuencias y probablidades de los 93 caracteres en una prepublicación de una edición previa de este libro que contenía 861 462 caracteres.

Tabla Intr.2: Frecuencias y probabilidades de los caracteres.

la compresión de los mismos. El código ASCII para caracteres, es un buen ejemplo de representación de datos más larga de lo necesario. Utiliza códigos de 7 bits, porque es fácil trabajar con códigos de tamaño fijo. Un código de tamaño variable, sin embargo, sería más eficiente, ya que algunos caracteres se usan más que otros, y así, se podrían asignar códigos más cortos.

Por eso, en un mundo donde los datos se representan siempre con el formato más corto posible, no habría forma de comprimirlos. En lugar de escribir libros sobre la compresión de datos, los autores en ese mundo podrían escribir libros sobre cómo determinar el formato más corto para los diferentes tipos de datos.

Una Palabra para el sabio...

El objetivo principal del campo de la compresión de datos es, por supuesto, desarrollar métodos de compresión, mejores y más rápidos. Sin embargo, uno de los principales dilemas del arte de la compresión de datos, es cuándo dejar de buscar una mejor compresión. La experiencia demuestra que afinar un algoritmo, para eliminar los últimos fragmentos restantes de redundancia en los datos, reduce el rendimiento. La modificación de un algoritmo, para mejorar la compresión en un 1 %, puede aumentar el tiempo de ejecución en un 10 %, y la complejidad del programa, en más que eso. Un buen camino para salir de este dilema, fue el tomado por Fiala y Greene (Sección 3.9). Después de desarrollar sus algoritmos principales, A1 y A2, los modificaron para producir menos compresión —a mayor velocidad—, lo que se tradujo en los algoritmos B1 y B2. A continuación, modificaron A1 y A2 de nuevo, pero en sentido contrario, sacrificando la velocidad para obtener una compresión ligeramente mejor.

El principio de compresión, mediante la eliminación de la redundancia, también responde a la siguiente pregunta: "¿Por qué un archivo ya comprimido no puede comprimirse más?" La respuesta, por supuesto, es que dicho archivo tiene poca u ninguna redundancia, por lo que no hay nada que eliminar. Un ejemplo es un fichero con texto aleatorio. En tal texto, cada letra aparece con igual probabilidad, por lo que asignandolas códigos de tamaño fijo no se añade ninguna redundancia. Cuando un archivo semejante es comprimido, no hay redundancia que eliminar. (Otra respuesta, es que si fuera posible comprimir un archivo ya comprimido, tras sucesivas compresiones se podría reducir el tamaño del mismo hasta llegar a un solo byte, o incluso un solo bit. Esto, por supuesto, es ridículo ya que un solo bit no puede contener la información presente en cualquier fichero grande.) El lector debería consultar también la Sección 8.7 (Pág. 885) para darle un giro interesante al tema de la compresión de datos aleatorios.

Ya que se han mencionado los datos aleatorios, vamos a decir unas pocas palabras más acerca de ellos. Normalmente, es raro tener un archivo con datos aleatorios, pero hay un buen ejemplo —un archivo totalmente comprimido—. El dueño de un archivo comprimido, normalmente sabe que ya está comprimido, y no intenta comprimirlo aún más; pero hay una excepción, la transmisión de datos mediante módems. Los módems actuales disponen de un hardware para comprimir automáticamente los datos que envían, y si los datos ya están comprimidos, no habrá una mayor compresión. Pueden incluso ocupar más que antes. Esto es por lo que un modem debería controlar la razón de compresión "sobre la marcha", y si es baja, debería parar la compresión y enviar el resto de los datos sin comprimir. El protocolo V.42bis (Sección 3.21), es un buen ejemplo de esta técnica.

Un simple argumento, ilustra la esencia de lo expuesto: "La compresión de datos se realiza para reducir la redundancia en los datos". Esta afirmación, muestra que la mayor parte de los archivos de datos no pueden ser comprimidos, no importa qué método de compresión se utilice. Esto parece extraño

al principio, porque estamos comprimiendo nuestros archivos de datos continuamente. El hecho es, que la mayor parte de los archivos no pueden comprimirse, debido a que son aleatorios o cercanos al azar, y por ello no tienen redundancia. Los (relativamente) pocos ficheros que pueden ser comprimidos, son los únicos que queremos comprimir; son aquellos que usamos continuamente. Tienen redundancia, no son aleatorios, y por eso son útiles e interesantes.

Este es el razonamiento. Si tenemos dos archivos distintos —A y B— que han sido comprimidos, ofreciendo como salida otros dos archivos —C y D, respectivamente—, es claro, que C y D deben ser diferentes. Si fueran idénticos, no habría manera de descomprimirlos y obtener de nuevo los ficheros de origen, A y B.

Supongamos que nos dan un archivo de n bits, y que queremos comprimirlo de forma eficiente. Cualquier método de compresión que pueda reducir este archivo, digamos, 10 bits, sería bienvenido. Incluso la compresión de 11 ó 12 bits sería genial. Por eso, podemos suponer (de forma un tanto arbitraria), que la compresión de dicho archivo a la mitad de su tamaño o más, se considera una buena compresión. Hay 2^n archivos de n bits, y tendrían que ser comprimidos en 2^n archivos diferentes, de tamaños inferiores o iguales a $\frac{n}{2}$. Sin embargo, el número total de estos archivos es[27]:

$$N = 1 + 2 + 4 + \cdots + 2^{\frac{n}{2}} = 2^{1+\frac{n}{2}} - 1 \approx 2^{1+\frac{n}{2}},$$

por lo que sólo N, de los 2^n archivos de origen, tienen la suerte tener una compresión eficiente. El problema, es que N es mucho más pequeño que 2^n. Aquí hay dos ejemplos sobre la razón entre estos dos números.

Para $n = 100$ (archivos con exactamente 100 bits), el número total de archivos es 2^{100}, y el número de archivos que pueden ser comprimidos con eficiencia, es 2^{51}. La razón entre estos números, es la ridículamente pequeña cifra de $2^{-49} \approx 1{,}78 \times 10^{-15}$.

Para $n = 1000$ (archivos con exactamente 1000 bits —125 bytes—), el número total de archivos es 2^{1000}, y el número de archivos que pueden ser comprimidos con eficiencia, es 2^{501}. La razón entre estos números, es la ridículamente pequeña cifra de $2^{-499} \approx 9{,}82 \times 10^{-91}$.

La mayor parte de los archivos de interés, tienen un tamaño de al menos unos miles de bytes. Para esos ficheros, el porcentaje de los que pueden comprimirse eficientemente es tan pequeño, que no se puede calcular con números en coma flotante, incluso en una supercomputadora (el resultado aparece como cero).

La cifra del 50 % que se utiliza aquí, es arbitraria, pero incluso aumentandola al 90 %, no representa una diferencia significativa. Veamos por qué. Suponiendo que tenemos un archivo de n bits, y que $0{,}9n$ es un entero, el número de archivos de tamaño no superior a $0{,}9n$ es:

$$N = 2^0 + 2^1 + \cdots + 2^{0,9n} = 2^{1+0,9n} - 1 \approx 2^{1+0,9n}.$$

Para $n = 100$, hay 2^{100} archivos, y $2^{1+90} = 2^{91}$ pueden comprimirse bien. La razón entre estos números es: $2^{91}/2^{100} = 2^{-9} \approx 0{,}00195$. Para $n = 1000$, la fracción correspondiente es: $2^{901}/2^{1000} = 2^{-99} \approx 1{,}578 \times 10^{-30}$. Estas fracciones aún son extremadamente pequeñas.

Por lo tanto, es evidente, que ningún método de compresión puede aspirar a comprimir todos los archivos, o incluso un porcentaje significativo de ellos. Para comprimir un archivo de datos, el algoritmo de compresión tiene que examinar los datos, encontrar las redundancias existentes y tratar de eliminarlas. Las redundancias en los datos dependen del tipo de datos (texto, imágenes, sonido, etc.), que es por lo que se tiene que desarrollar un nuevo método de compresión, adaptado para que funcione lo mejor posible, para ese tipo específico de datos. No existe un algoritmo eficiente y universal para la compresión de datos[28].

La compresión de datos se ha vuelto tan importante, que algunos investigadores (véase, por ejemplo, [Wolff 99]), han propuesto la teoría SP (por "sencillez" y "poder"), lo que sugiere que: ¡todo cálculo

[27] La suma de una progresión geométrica de razón r es $S = p_1 + \cdots + p_n = \frac{p_n \cdot r - p_1}{r-1}$.

[28] Cada algoritmo se suele comportar bien sólo con determinado tipo de datos.

es compresión! En concreto, dice: La compresión de datos, puede ser interpretada como un proceso para eliminar la complejidad innecesaria (redundancia) de la información, y de ese modo, maximizar la sencillez, mientras se conserva lo más posible su poder descriptivo, no redundante. La teoría SP está basada en las siguientes conjeturas:

- Todas las clases de computación y razonamiento formal, pueden ser entendidas provechosamente, como la compresión de la información por coincidencia de patrones, unificación y búsqueda.

- El proceso para encontrar y eliminar la redundancia, siempre se puede entender a un nivel fundamental, como un proceso de búsqueda de patrones que coincidan entre sí, y la fusión o unificación de los casos repetidos de algún patrón, para construir otro.
  Este libro describe muchos métodos de compresión, algunos adecuados para textos, y otros para gráficos (imágenes fijas o películas) o para audio. La mayoría de los métodos, se clasifican en cuatro categorías: codificación run length[29] (RLE o *Run Length Encoding*), métodos estadísticos, métodos basados en diccionarios (a veces llamados LZ) y transformadas. Los Capítulos 1 y 8, describen métodos basados en otros principios.
  Antes de profundizar en los detalles, veremos algunos términos importantes sobre la compresión de datos.

- El *compresor o codificador*, es el programa que comprime los datos brutos[30] de la secuencia de entrada y crea una secuencia de salida, formada por datos comprimidos (con baja redundancia). El *descompresor o decodificador* realiza la conversión en la dirección opuesta. Tenga en cuenta que el término *encoding* —codificación— es muy general, y tiene varios significados; pero dado que tratamos sólo la compresión de datos, utilizamos el nombre *encoder* —codificador— para referirnos al compresor de datos. El término *codec*, se utiliza, a veces, para describir tanto al codificador como al decodificador. Del mismo modo, el término *companding* es la abreviatura de "compressing/expanding" —compresión/expansión—.

- En este libro, se utiliza el término "stream"[31] en lugar de "archivo"[32]. "Stream" es un término más general, porque los datos comprimidos se pueden transmitir directamente al decodificador, en vez de escribirlos en un archivo y guardarlos. Además, los datos a comprimir se pueden descargar de una red en lugar de proceder de un archivo.

- Para el *flujo*[33] *de entrada de origen*, se utilizan los términos *sin codificar*[34], *en bruto*[35], o *datos de origen*[36]. El contenido de la cadena final, comprimida, se conoce como *datos codificados* o *comprimidos*. El término *bitstream* (*flujo —secuencia o cadena— de bits*), también se utiliza en la literatura para referirse a la cadena comprimida.

[29] Una codificación basada en la longitud de rachas de elementos iguales.
[30] Los datos de origen, sin tratar.
[31] Corriente, flujo, cadena ....
[32] File.
[33] Son sinónimos: cadena o secuencia.
[34] Unencoded.
[35] Raw.
[36] Original data.

El escarabajo de oro[a]

Aquí, pues, tenemos, muy al principio, las bases para algo más que una mera suposición. El uso general que puede hacerse de la tabla es obvio; pero, en este código concreto, requeriremos su ayuda sólo muy parcialmente. Como nuestro carácter predominante es 8, comenzaremos por asumir que es la "e" del alfabeto natural. Para verificar la suposición, observamos si el 8 aparece a menudo en parejas —la "e" está duplicada con una gran frecuencia en Inglés— en esas palabras; por ejemplo, como "meet," "fleet," "speed," "seen," "been," "agree," etc.[b] En el presente caso, vemos que se duplicaba no menos de cinco veces, aunque la criptografía es breve.

—Edgar Allan Poe.

[a] The Gold Bug (literalmente el bicho de oro).
[b] "Conocer", "flota", "velocidad", "visto", "estado", "de acuerdo", etc...

- Un método de compresión *no adaptativo* es rígido, y no modifica sus operaciones, sus parámetros o sus tablas, en respuesta a los datos específicos que se están comprimiendo. Tal método, es el mejor que podemos usar para comprimir datos del mismo tipo. Ejemplos, son los métodos del Grupo 3 y Grupo 4 para compresión de facsímiles (Sección 2.13), los cuáles están especificamente diseñados para la compresión de facsímiles, y harían un mal trabajo al comprimir otros tipos de datos. En oposición, un método *adaptativo* examina los datos en bruto, y modifica sus operaciones y/o sus parámetros, como corresponda. Un ejemplo es el método adaptativo de Huffman de la Sección 2.9. Algunos métodos de compresión, utilizan un algoritmo de 2 pasadas, de manera que en la primera, lee el flujo de datos entrada para recoger estadísticas sobre los datos a comprimir, y en la segunda realiza la compresión real, utilizando los parámetros establecidos por la primera pasada. Un método como éste, puede llamarse *semiadaptativo*. Un método de compresión de datos, puede ser también localmente adaptativo, lo que significa que se adapta a las condiciones locales en el flujo de entrada, y varía esta adaptación según la zona de datos de entrada que esté tratando. Un ejemplo es el método *mover al frente*[37] (Sección 1.5).

- Compresión *con/sin pérdidas*[38]: Ciertos métodos de compresión tienen pérdidas de datos. Logran una mejor compresión, perdiendo alguna información. Cuando se descomprime la cadena comprimida, el resultado no es idéntico a la secuencia de datos original. Este método tiene sentido especialmente, en la compresión de imágenes, películas o sonidos. Si la pérdida de datos es pequeña, no podremos darnos cuenta de la diferencia. En oposición, los archivos de texto, especialmente aquellos que contienen programas informáticos, pueden quedar invalidados, incluso si sólo se modifica un bit. Tales archivos, deberían comprimirse solamente con métodos de compresión sin pérdidas. [Debemos mencionar dos ideas, en relación con los archivos de texto: (1) Si un archivo de texto contiene el código fuente de un programa, los espacios en blanco consecutivos, a menudo pueden ser sustituidos por un único espacio. (2) Cuando la salida de un procesador de texto se guarda en un archivo de texto, el archivo puede contener información acerca de los tipos de fuentes utilizadas en el texto. Dicha información puede ser descartada si el usuario sólo está interesado en salvar el texto.]

- *Compresión en cascada*[39]: La diferencia entre los codificadores sin pérdidas y los que sí tienen pérdidas, puede aclararse considerando compresiones en cascada. Imagine un archivo de datos A, que ha sido comprimido por un codificador X, produciendo como resultado un archivo comprimido B. Es posible, aunque sin sentido, pasar B a través de otro codificador Y, para producir

[37] Move-to-front.
[38] Lossy/lossless.
[39] Cascaded compression.

un tercer archivo comprimido C. El hecho es, que si los métodos X e Y no tienen pérdidas, después de la decodificación de C con Y, se producirá exactamente B, que al decodificarlo con X producirá el archivo original A. Sin embargo, si cualquiera de los algoritmos de compresión tiene pérdidas, entonces la decodificación de C con Y puede producir un archivo B' diferente de B. Pasando B' a través de X se puede producir algo muy distinto de A, y también puede generar un error, porque X puede no ser capaz de leer B'.

- *Compresión de percepción*[40]: Un codificador con pérdidas, debe aprovechar el tipo especial de los datos a comprimir. Debe eliminar sólo, los datos cuya ausencia no sería detectada por nuestros sentidos. Por eso, tal codificador debe emplear algoritmos basados en nuestra comprensión de la percepción psicoacústica y psicovisual, por lo que a menudo se le denomina codificador de percepción. Se puede conseguir, que este tipo de codificador funcione con una razón de compresión constante, donde por cada x bits de datos en bruto, produzca una salida de y bits de datos comprimidos. Esto es conveniente, en aquellos casos en los que la cadena comprimida tiene que ser transmitida a un ritmo constante. La compensación, es una cualidad subjetiva y variable. Parte de los datos originales que son difíciles de comprimir, tras la descompresión, quedan (o suenan) mal. Esos datos, pueden requerir más de y bits de salida por cada x bits de entrada.

- La *Compresión simétrica*[41] se produce cuando el compresor y descompresor utilizan básicamente el mismo algoritmo, pero trabajan en direcciones "opuestas". Este método tiene sentido para el trabajo general, donde hay tantos archivos comprimidos como descomprimidos. En un método de compresión asimétrico, uno de los dos, el compresor o el descompresor, puede tener que trabajar mucho más duro. Tales métodos tienen su utilidad, y no son necesariamente malos. Un método de compresión, en el que el compresor ejecuta un lento y complejo algoritmo, y el descompresor es sencillo, es la elección lógica cuando los archivos están comprimidos dentro de un fichero, de donde, muy a menudo, son descomprimidos y utilizados. El caso opuesto, es útil en entornos en los que los archivos se actualizan todo el tiempo y se hacen copias de seguridad. La posibilidad de que un archivo de la copia de seguridad sea usado, es pequeña, por lo que el descompresor no se utiliza muy a menudo.

> La estación de esquí se llena de chicas a la caza de maridos, y de maridos a la caza de chicas; la situación no es tan simétrica como podría parecer.
>
> —Alan Lindsay Mackay, conferencia, Birckbeck College, 1964.

◇ **Ejercicio Intro.2 (sol. en pág. 1051):** Proporcione un ejemplo de un archivo comprimido en el que es importante una buena compresión, pero tanto la velocidad del compresor como la del descompresor, no lo es.

- Muchos métodos de compresión modernos, son asimétricos. A menudo, la descripción formal (la estándar) de tales métodos, se refiere al decodificador y al formato del flujo de datos a comprimir, pero no determina el funcionamiento del codificador. Cualquier codificador que genera una cadena comprimida correcta, se considera *compatible*, así como también cualquier decodificador que pueda leer y decodificar dicha cadena. La ventaja de tal descripción, es que cualquier persona es libre de desarrollar y aplicar nuevos y sofisticados algoritmos para el codificador. El implementador, no necesita ni siquiera publicar los detalles del codificador, que puede considerarse de su propiedad. Si se demuestra que un codificador compatible, es mejor que los codificadores

[40] Perceptive compression.
[41] Symmetrical compression.

de la competencia, puede convertirse en un éxito comercial. Con este sistema, el codificador se considera *algorítmico*, mientras que el decodificador, que normalmente es mucho más sencillo, se denomina *determinista*. Un buen ejemplo de este enfoque es el método de compresión de audio MPEG-1 (Sección 7.14).

- Un método de compresión de datos se llama *universal*, cuando ni el compresor ni el descompresor conocen las estadísticas del flujo de datos de entrada. Un método universal es óptimo, si el compresor puede producir factores de compresión que se acercan asintóticamente a la entropía de largas cadenas de entrada.

- El término *diferenciación de archivo*s, se refiere a cualquier método que localiza y comprime las diferencias entre dos archivos. Imagine un archivo A, con dos copias mantenidas por dos usuarios. Cuando una copia se actualiza por un usuario, debe ser enviada al otro, para mantener las dos copias idénticas. En lugar de enviar una copia de A, que puede ser grande, se puede enviar un archivo mucho más pequeño, que contiene sólo las diferencias —en formato comprimido—, y utilizarlo en el receptor, para actualizar la copia de A. En la Sección 8.14.2 se describen algunos de los detalles, y se muestra por qué la compresión puede ser considerada un caso especial de la diferenciación de archivos. Tenga en cuenta que el término "diferenciación" se utiliza en la Sección 1.3.1 para describir un método de compresión completamente diferente.

- La mayor parte de los métodos de compresión funcionan en *modo continuo*[42] —sin interrupciones—, donde el codificador introduce uno o varios bytes, los procesa y continúa el proceso hasta detectar el final del archivo. Algunos métodos, tales como Burrows-Wheeler (Sección 8.1), trabajan en *modo bloque*, donde el flujo de entrada se lee bloque por bloque y cada bloque se codifica por separado. El tamaño del bloque, en este caso, debería ser un parámetro controlado por el usuario, ya que su tamaño podría afectar significativamente al rendimiento del método.

- Casi todos los métodos de compresión son *físicos*. Miran sólo los bits de la secuencia de entrada e ignoran el significado de los elementos de datos entrantes (e.g., los elementos de datos pueden ser palabras, píxeles o muestras de audio). Este método convierte una secuencia de bits en otra más corta y única. Sólo hay una manera de dar sentido a la secuencia de salida (para decodificarlo), y es saber cómo se ha codificado. Algunos métodos de compresión son *lógicos*. Éstos, buscan elementos de datos individuales en la secuencia de entrada y reemplazan los elementos comunes con códigos cortos. Un método así, tiene normalmente un uso especial, y solamente puede tener éxito con determinados tipos de datos. El método de sustitución de patrones, descrito en la Sección 1.4 (RLE), es un ejemplo de un método de compresión lógica.

- *Rendimiento de la compresión*: Se utilizan varias medidas estándar para expresar el rendimiento de un método de compresión.

1. La *ratio o razón*[43] *de compresión* se define como:

$$\text{Razón de compresión} = \frac{\text{tamaño de la cadena de salida}}{\text{tamaño de la cadena de entrada}}.$$

Un valor de 0,6, significa que los datos ocupan, tras la compresión, un 60 % que su tamaño original. Valores mayores que 1, implican un flujo de salida mayor que el de entrada (compresión negativa). La razón de compresión, puede llamarse también *bpb* (bit por bit), ya que es igual, al número de bits necesarios de la cadena comprimida, en promedio, para comprimir un bit de la cadena de entrada. En la compresión de imágenes, el mismo término, bpb, significa "bits por

[42] Streaming mode.
[43] En la palabra razón está implícita un cómputo por lo que es mejor opción que "relación".

píxel". En la actualidad, eficientes métodos de compresión de texto, hacen que tenga sentido hablar de bpc (bits por carácter) —el número de bits que se necesitan, en promedio, para comprimir un carácter de la secuencia de entrada—.
Debemos mencionar otros dos términos relacionados con la razón de compresión. La *tasa de bits*[44], es un término general para bpb y bpc. En consecuencia, el principal objetivo de la compresión de datos, es representar cualquier dato en tasas de bits bajas. El término *bit de coste*[45], se refiere a la función que desempeñan los bits individuales en la cadena comprimida. Imagine una secuencia de datos comprimida, donde el 90 % de los bits son códigos de tamaño variable de algunos símbolos, y el restante 10 % se utiliza para codificar ciertas tablas; el bit de coste para las tablas es del 10 %.

2. La inversa de la razón de compresión, se llama *factor de compresión*:

$$\text{Factor de compresión} = \frac{\text{tamaño de la cadena de entrada}}{\text{tamaño de la cadena de salida}}.$$

   En este caso, los valores superiores a 1 indican compresión, y los valores menores que 1 implican expansión. Esta medida parece natural a mucha gente, ya que cuanto mayor sea el factor, mayor será la compresión. Está lejanamente emparentada con la proporción de raleza o dispersión, una medida de rendimiento que se explica en Sección 5.6.2.

3. La expresión $100\times$ (1 − Razón de compresión), es también una medida razonable del rendimiento de la compresión. Un valor de 60 significa que la cadena de salida ocupa el 40 % de su tamaño original (o que la compresión ha producido un ahorro del 60 %).

4. En la compresión de imágenes, la cantidad bpp (bits por píxel), es de uso común. Es igual al número de bits necesarios, en promedio, para comprimir un píxel de la imagen. Esta cantidad debe compararse siempre con el bpp de antes de la compresión.

5. La *ganancia de compresión* se define como

$$100 \log_e \frac{\text{tamaño de referencia}}{\text{tamaño comprimido}},$$

   donde *tamaño de referencia* es el tamaño del flujo de datos entrantes, o el tamaño de una cadena comprimida —producida por algún método de compresión de datos estándar y sin pérdidas—. Para números pequeños x, es cierto que $\log_e(1+x) \approx x$, por lo que un pequeño cambio en la ganancia de compresión es muy similar al mismo cambio en la razón de compresión. Debido al uso del logaritmo, dos ganancias de compresión pueden ser comparadas, sencillamente restándolas. La unidad de la ganancia de compresión se llama *tanto por ciento de la razón logaritmica*[46] y se denota por $\%\!\!\!\!\bigcirc$.

6. La velocidad de compresión puede medirse en *ciclos por byte* (CPB). Este es el número promedio de ciclos de máquina, que se necesita para comprimir un byte. Esta medida es importante cuando la compresión se hace por un hardware especial.

7. Otras cantidades, tales como el *error cuadrado medio* (MSE o *Mean Square Error*) y la *máxima señal de la proporción de ruido* (PSNR o *Peak Signal to Noise Ratio*), se utilizan para medir la distorsión causada por la compresión con pérdida, de imágenes y películas. La Sección 4.2.2 proporciona información sobre ellos.

[44] Bitrate (o "bit rate").
[45] Bit budget.
[46] Percent log ratio.

8. La *compresión relativa*, se utiliza para medir la ganancia de compresión en los métodos de compresión de audio sin pérdidas, tales como MLP (Sección 7.7). Refleja la calidad de la compresión, indicando en cuántos bits se ha reducido cada muestra de audio.

- El *Calgary Corpus*, es un conjunto de 18 archivos utilizados tradicionalmente para probar los algoritmos de compresión de datos y sus implementaciones. Incluyen texto, imágenes y archivos de objetos, con un total de más de 3,2 millones de bytes (Tabla Intr.3). El corpus se puede descargar por FTP anónimo desde [Calgary 06].

- El *Canterbury Corpus* (Tabla Intr.4), es otra collección de archivos introducida en 1997, para proporcionar una alternativa al Calgary corpus en la evaluación de métodos de compresión. Las inquietudes que condujeron al nuevo corpus fueron las siguientes:

  1. El Calgary corpus ha sido utilizado por muchos investigadores para desarrollar, probar y comparar muchos métodos de compresión y existe la posibilidad de que los métodos nuevos se optimicen, involuntariamente, para ese corpus. Pueden rendir bien con los documentos de Calgary, pero mal con otros archivos.
  2. El Calgary corpus se creó en 1987 y se está volviendo viejo. Los documentos "Típicos" cambian cada unas décadas (por ejemplo, los documentos HTML no existían hasta hace poco) y cualquier conjunto de documentos utilizados para evaluación debe examinarse de vez en cuando.
  3. El Calgary corpus es más o menos una colección arbitraria de documentos; sin embargo, un buen corpus para la evaluación de algoritmos, debe seleccionarse cuidadosamente.

  El Canterbury corpus comenzó con cerca de 800 documentos candidatos, todos de dominio público. Fueron divididos en 11 clases, en representación de los diferentes tipos de documentos. Se seleccionó un documento "promedio" representativo de cada clase —mediante la compresión de cada archivo de la clase, utilizando diferentes métodos, y la elección de aquel cuya compresión era más cercana a la media (según lo determina la regresión estadística)—. El corpus se resume en la Tabla Intro.4 y se puede obtener desde [Canterbury 06].

- El *Artificial Corpus* (Tabla Intr.5), es una collección de archivos para probar el comportamiento de los métodos de compresión en 3 casos: con pocas repeticiones o ninguna (ej. random.txt), con abundancia de repeticiones (ej. alphabet.txt), o con archivos muy pequeños (ej. a.txt).
  El promedio de los resultados tiene poca o ninguna relevancia, pues los archivos han sido diseñados para detectar valores atípicos. Además los tiempos de ejecución no son significativos con archivos "triviales". El corpus se resume en la Tabla Intro.5 y se puede obtener también a partir de [Canterbury 06].

- El *Miscellaneous Corpus*, es una colección de archivos variados diseñados por investigadores y otras personas que desean publicar los resultados de la compresión utilizando sus propios archivos. En el 2001, sólo constaba del archivo *pi.txt* en cuyo interior se encuentran los primeros dígitos del número π hasta completar 1 000 000 de bytes. Se puede obtener también a partir de [Canterbury 06].

- El *Large Corpus* (Tabla Intr.6), es una colección de archivos relativamente grandes. Si bien la mayoría de los métodos de compresión pueden ser evaluados satisfactoriamente en archivos más pequeños, algunos requieren grandes cantidades de datos para obtener una buena compresión, y otros son tan rápidos, que necesitan archivos de mayor tamaño para medir su velocidad de forma más fiable. Se puede obtener también a partir de [Canterbury 06].

Nombre	**Tipo**	**Líneas**	**Palabras**	**Bytes**	**Descripción**
bib	Texto	6 280	19 274	111 261	Bibliografía en formato *de referencia* UNIX.
book1	Texto	16 622	141 274	768 771	*Far from the madding crowd* (Lejos del mundanal ruido) de Hardy.
book2	Texto	15 634	101 221	610 856	*Principles of computer speech* (Principios del lenguaje informático) de Ian Witten.
geo	Datos	18	617	102 400	Datos Geofísicos.
news	Texto	10 059	53 939	377 109	Archivos de Noticias de usuarios de la red.
obj1	Objeto	87	495	21 504	Compilación de progp (Código compilado para VAX).
obj2	Objeto	1 213	4 600	246 814	Sistema de apoyo al conocimiento (Código compilado para Apple Macintosh).
paper1	Texto	1 250	8 512	53 161	*Arithmetic coding for data compression* de Witten, Neal y Cleary.[a]
paper2	Texto	1 731	13 829	82 199	*Computer (in)security* ((In)seguridad informática) de Witten.
paper3	Texto	1 100	7 219	46 526	*In search of "autonomy"* (En busca de "autonomía") de Witten.
paper4	Texto	294	2 166	13 286	*Programming by example revisited* de Cleary.[b]
paper5	Texto	320	2 099	11 954	*A logical implementation of arithmetic* de Cleary.[c]
paper6	Texto	1 019	6 753	38 105	*Compact hash tables using bidirectional linear probing* de Cleary.[d]
pic	Imagen	0	49	513 216	Imágenes de Fax de archivos de test de CCITT (texto + imágenes).
progc	Fuente	1 487	6 313	39 611	Versión 4.0 de compress (Código fuente en C).
progl	Fuente	2 244	9 235	71 646	Sofware de sistema (Código fuente en Lisp).
progp	Fuente	1 966	4 847	49 379	Programa de predicción por evaluación de concordancia parcial (en Pascal).
trans	Texto	2 737	9 255	93 695	Transcripción de una sesión de terminal.

[a] Códigos aritméticos para la compresión de datos.
[b] Programando mediante reutilización de ejemplos.
[c] Una implementación lógica de la aritmética.
[d] Compactación de tablas hash usando pruebas lineales bidireccionales.

Tabla Intr.3: El Calgary Corpus (2005).

Nombre	**Tipo**	**Categoría**	**Tamaño**
alice29.txt	text	Texto en Inglés	152 089
asyoulik.txt	play	Shakespeare	125 179
cp.html	html	Fuente HTML	24 603
fields.c	Csrc	Fuente C	11 150
grammar.lsp	list	Fuente LISP	3 721
kennedy.xls	Excl	Hoja de cálculo de Excel	1 029 744
Icet10.txt	tech	Escrito técnico	426 754
plrabn12.txt	poem	Poesía	481 861
ptt5	fax	Conjunto de tests CCITT	513 216
sum	SPRC	Ejecutable SPARC	38 240
xargs.1	man	Página de manual GNU	4 227

Tabla Intr.4: El Canterbury Corpus (2001).

Nombre	**Tipo**	**Categoría**	**Tamaño**
a.txt	a	La letra 'a'	1
aaa.txt	aaa	Shakespeare	100 000
alphabet.txt	alfabeto	Suficientes repeticiones del alfabeto como para llenar 100 000 caracteres.	100 000
random.txt	random	100 000 caracteres aleatorios del conjunto [a-z | A-Z | 0-9 |! |] (alfabeto de tam. 64).	100 000

Tabla Intr.5: El Artificial Corpus (2001).

Nombre	**Tipo**	**Categoría**	**Tamaño**
E.coli	E.coli	Genoma completo de la bacteria E.Coli	4 638 690
bible.txt	biblia	Versión de la biblia de King James	4 047 392
world192.txt	mundo	El libro de hechos del mundo de la CIA	2 473 400

Tabla Intr.6: El Large Corpus (2001).

- El *modelo de probabilidad*. Este concepto es importante en los métodos de compresión de datos estadísticos. En estos métodos, tiene que construirse un modelo de los datos, antes de poder comenzar la compresión. Un modelo típico, puede construirse leyendo por completo los datos de entrada, contando el número de veces que aparece cada símbolo (su frecuencia de aparición), y calculando la probabilidad de ocurrencia de cada símbolo. Se vuelven a leer los datos de entrada, símbolo a símbolo, y se comprimen con la información del modelo de probabilidad. Un ejemplo típico se muestra en la Tabla 2.47.
  La lectura completa de los datos de entrada dos veces es lenta, razón por la cual los métodos de compresión más prácticos utilizan estimaciones, o se adaptan a los datos a medida que son leídos de la secuencia de entrada y comprimidos. Es fácil escanear grandes cantidades de, digamos, un texto en inglés y calcular las frecuencias y las probabilidades de cada carácter. Esta información, puede servir después como un modelo aproximado y puede ser utilizado por los métodos de compresión de texto para comprimir cualquier texto en inglés. También es posible comenzar asignando probabilidades iguales a todos los símbolos de un alfabeto; luego, al mismo tiempo que se leen los símbolos y son comprimidos, se calculan también las frecuencias y se va construyendo el modelo de probabilidad. Este es el principio oculto en los *métodos de compresión de adaptativos.*

[Fin de los términos sobre compresión de datos.]

El concepto de *integridad y fiabilidad de datos* (pág. 107) es, en cierto sentido, contrario a la compresión de datos. Sin embargo, los dos conceptos están relacionados muy a menudo, ya que cualquier buen programa de compresión de datos debe generar un código fiable y por lo tanto, tendría que ser capaz de utilizar códigos de detección y corrección de errores.

Este libro es para aquellos lectores que tienen un conocimiento básico de informática; que saben algo acerca de programación y estructuras de datos; que se sienten a gusto con términos como *bit, mega, ASCII, archivo , E/S* y *búsqueda binaria* y que quieren conocer cómo se comprimen los datos. La base matemática necesaria es mínima y se limita a: logaritmos, matrices, polinomios, diferenciación o integración y el concepto de probabilidad. Este libro no pretende ser una guía para los implementadores de software y tiene pocos programas.

Las siguientes direcciones URLs tienen enlaces útiles y referencias a algunos recursos sobre compresión de datos disponibles en Internet y otros lugares:

http://www.hn.is.uec.ac.jp/~arimura/compression_links.html
http://cise.edu.mie-u.ac.jp/~okumura/compression.html
http://compression.ca/(sobre todo las comparaciones) y
http://datacompression.info/

La última URL tiene mucha información sobre la compresión de datos, incluyendo tutoriales, enlaces a trabajadores en ese campo y listas de libros. El sitio es mantenido por Mark Nelson.

La referencia [Okumura 98], analiza la historia de la compresión de datos en Japón.

Recursos para la compresión de datos

Hay disponible un gran número de recursos para la compresión de datos. Cualquier búsqueda en Internet de “data compression”, “lossless data compression”, “image compression”, “audio compression”, “compresión de datos”, “compresión de datos sin pérdidas”, “compresión de imágenes”, “compresión de audio” y temas similares devuelve al menos unos miles de resultados. Los recursos tradicionales (impresos) abarcan, desde textos generales y textos sobre aspectos específicos o métodos particulares, hasta artículos en revistas, informes técnicos y documentos de investigación de publicaciones científicas. A continuación se ofrece una breve lista de libros (la mayoría genéricos), ordenados por fecha de publicación:

Khalid Sayood, *Introduction to Data Compression*, Morgan Kaufmann, 3rd edition (2005).

Ida Mengyi Pu, *Fundamental Data Compression*, Butterworth-Heinemann (2005).

Darrel Hankerson, *Introduction to Information Theory and Data Compression*, Chapman & Hall (CRC), 2nd edition (2003).

Peter Symes, *Digital Video Compression*, McGraw-Hill/TAB Electronics (2003).

Charles Poynton, *Digital Video and HDTV Algorithms and Interfaces*, Morgan Kaufmann (2003).

IainE.G.Richardson, *H.264 and MPEG-4 Video Compression: Video Coding for Next Generation Multimedia*, John Wiley and Sons (2003).

Khalid Sayood, *Lossless Compression Handbook*, Academic Press (2002).

Touradj Ebrahimi and Fernando Pereira, *The MPEG-4 Book*, Prentice Hall (2002).

Adam Drozdek, *Elements of Data Compression*, Course Technology (2001).

David Taubman and Michael Marcellin (Editors), *JPEG2000: Image Compression Fundamentals, Standards and Practice*, Springer Verlag (2001).

Kamisetty Ramam Rao, *The Transform and Data Compression Handbook*,CRC (2000).

Ian H. Witten, Alistair Moffat, and Timothy C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images*, Morgan Kaufmann, 2nd edition (1999).

Peter Wayner, *Compression Algorithms for Real Programmers*, Morgan Kaufmann (1999).

John Miano, *Compressed Image File Formats: JPEG, PNG, GIF, XBM, BMP, ACM* Press and Addison-Wesley Professional (1999).

Mark Nelson and Jean-Loup Gailly, *The Data Compression Book*, M&T Books, 2nd edition (1995).

William B. Pennebaker and Joan L. Mitchell, *JPEG: Still Image Data Compression Standard*, Springer Verlag (1992).

Timothy C. Bell, John G. Cleary, and Ian H. Witten, *Text Compression*, Prentice Hall (1990).

James A. Storer, *Data Compression: Methods and Theory*, Computer Science Press (1988).

John Woods, ed., *Subband Coding*, Kluwer Academic Press (1990).

El símbolo "␣", se utiliza para indicar un espacio en blanco en aquellos lugares donde pueda haber ambigüedad.

Algunos lectores han puesto en entredicho el título de este libro. ¿Qué significa para un trabajo de este tipo ser completo y cuán completo es este libro? Esta es mi opinión sobre el asunto: me gusta pensar que si todo el campo de la compresión de datos fuera (Dios no lo quiera) destruida, una parte sustancial de éste podría ser reconstruido a partir de este trabajo. Naturalmente, yo no me comparo con James Joyce, ya que sus obras nos ofrecen un ejemplo similar. A él le gustaba decir que si el Dublín de su época fuera destruído, podría ser reconstruido a partir de sus trabajos.

Los lectores que deseen tener una idea del esfuerzo que hubo que realizar para escribir este libro, deben consultar el colofón.

El autor agradece cualesquier comentarios, sugerencias y correcciones. Envíense a dsalomon@csun.edu. En caso de no recibir respuesta, los lectores pueden probar la dirección <anything>@DavidSalomon.name.

Sin duda existen semejanzas entre la publicación y el comercio de esclavos, pero no son sólo los autores quienes consiguen vender.
—Anthony Powell, Books Do Furnish A Room[47] *(1971)*

[47] Los libros amueblan una habitación.

Capítulo 1

Técnicas básicas

1.1. Compresión intuitiva

Los datos se comprimen reduciendo su redundancia, pero esto también hace que sean, menos fiables y más propensos a errores. El aumento de la integridad de los datos, por otra parte, se realiza mediante la adición de bits de verificación y bits de paridad, un proceso que aumenta el tamaño de los datos, lo que incrementa la redundancia. La compresión y la fiabilidad de los datos son, por lo tanto, opuestos y es interesante señalar que el último, pertenece a un campo relativamente reciente, mientras que el primero existía incluso antes de la llegada de las computadoras. El telégrafo receptivo, considerado en el Prefacio, el código Braille de 1820 (Sección 1.1.1) y el código Morse de 1838 (Tabla 2.1) utilizan formas de compresión sencillas e intuitivas. Por tanto, parece que la reducción de la redundancia es algo natural para cualquiera que trabaja con códigos, pero su incremento es algo que "va en contra de los principios" de los seres humanos. Esta sección trata métodos de compresión sencillos e intuitivos, que se han utilizado en el pasado. Hoy en día, estos métodos son, en su mayoría, de interés histórico, ya que generalmente son ineficientes y no pueden competir con los modernos métodos de compresión desarrollados en las últimas décadas.

1.1.1. Braille

Este conocido código, que permite a los ciegos a leer, fue desarrollado por Louis Braille en 1820, y después de haber sido modificado varias veces, todavía se sigue utilizando hoy en día. Hay disponibles muchos libros en Braille en el National Braille Press. El código Braille se compone de grupos (o celdas) de 3×2 puntos cada una, con relieve en papel grueso. Cada uno de los 6 puntos de un grupo pueden ser planos o elevados, lo que implica que el contenido de la información de un grupo es equivalente a 6 bits, siendo posibles por ello un total de $2^6 = 64$ grupos. Las letras (Tabla 1.1), dígitos y signos de puntuacion más corrientes no requieren los 64 códigos, por lo que los grupos restantes pueden ser utilizados para codificar las palabras comunes como "and, for, of, the y with" y las cadenas de letras más utilizadas, como "ch, gh, sh, th, wh, ed, er, ou y ow" (Tabla 1.2a).

En español, parte de los grupos sobrantes representan las vocales acentuadas, la ñ y algunos signos de puntuación (Tabla 1.2b).

Para optimizar códigos: Una letra es mayúscula, si va precedida del *signo Mayúscula*; los números tienen la misma representación que las 10 primeras letras (1–2–3–4–5–6–7–8–9–0 se corresponden, respectivamente, con a–b–c–d–e–f–g–h–i–j), y deben ir precedidos del *signo Número*. Puesto que en las primeras letras, la última fila del codigo es siempre "○○", se aprovecha esta circunstancia para representar las fracciones (las dos primeras filas del código del numerador de la fracción son desplazadas

a	b	c	d	e	f	g	h	i	j	k	l	m
												

n	o	p	q	r	s	t	u	v	w	x	y	z
												

Tabla 1.1: Las 26 letras del alfabeto Braille.

and	for	of	the	with	ch	gh
						

sh	th	wh	ed	er	ou	ow
						

Tabla 1.2: (a) Algunas palabras y cadenas en Braille.

hacia abajo). Se pueden ver los signos especiales comentados, así como algunos ejemplos en la Tabla 1.2c.

La redundancia en situaciones cotidianas

A pesar de que no aumentar innecesariamente la redundancia en nuestros datos, usamos datos redundantes todo el tiempo, la mayor parte de las veces, sin darnos cuenta. estos son algunos ejemplos: Todas las lenguas naturales son redundantes. Un portugués que no habla italiano, puede leer un periódico italiano y aún así entender la mayoría de las noticias, porque reconoce la forma básica de muchos verbos y nombres italianos, y porque gran parte del texto que no entiende, es superfluo (i.e., redundante). PIN es un acrónimo de "Personal Identification Number"[a], pero los bancos siempre le preguntan por su "número PIN". SALT era un acrónimo de "Strategic Arms Limitations Talks"[b]; pero los locutores de televisión, en la década de 1970, seguían hablando de "SALT Talks"[c]. Estos son sólo dos ejemplos que ilustran cómo lo natural es encontrar redundancia en las situaciones de la vida cotidiana. Se pueden encontrar más ejemplo en la URL `http://www.corsinet.com/braincandy/twice.html`.

[a] Número de Identificación Personal.
[b] Limitaciones en las negociaciones sobre armas estratégicas.
[c] Negociaciones SALT.

◇ **Ejercicio 1.1 (sol. en pág. 1051):** Encuéntrense frases redundantes que formen parte de la vida cotidiana.

La magnitud de la compresión conseguida por el Braille, es pequeña pero importante, porque los libros en Braille tienden a ser muy grandes (un solo grupo abarca el área de alrededor de diez letras de imprenta). Incluso esta modesta compresión tiene un precio. Si un libro en Braille es manipulado o se hace viejo, y algunos puntos se aplanan, se pueden manifestar graves errores en su lectura ya que se utiliza cada grupo posible.

(*Windots2*, de [windots 06], *iBraile* de [sighted 06] y *Duxbury Braille Translator*, de [afb 06], son los programas actuales para aquellos que quieran experimentar con el sistema Braille).

á	é	í	ó	ú	ñ

,	;	:	.	-	¡!	¿?	“	(	)
									

Tabla 1.2: (b) á, é, í, ó, ú, ñ y signos de puntuación en Braille.

Mayúscula	Número

B mayúscula	2	20	458	5,6	$^{6}/_{93}$
					

Tabla 1.2: (c) Signos Mayúscula y Número, con algunos ejemplos.

1.1.2. Compresión irreversible de texto

Algunas veces, es aceptable “comprimir” texto, simplemente desechando alguna información. Esto se conoce como *compresión irreversible de texto* o *compactación*. El texto descomprimido no es idéntico al original, por lo que estos métodos no son de propósito general; sólo pueden utilizarse en casos especiales.

Una secuencia consecutiva de espacios en blanco, puede reemplazarse por un solo espacio. Esto es aceptable en textos literarios y en la mayor parte del código fuente de programas informáticos; pero no debería ser usado cuando el formato de los datos es tabular.

En casos extremos, todos los caracteres, excepto letras y espacios, pueden ser despreciados, y se puede convertir todo el texto, a mayúsculas o minúsculas[1], reduciendo así el número de signos a codificar. De esta manera, un texto en inglés estará compuesto por combinaciones de exactamente 27 símbolos, cada uno de los cuales puede ser codificado con 5 bis, en lugar de los 8 usuales. La razón de compresión es de $^{5}/_{8} = 0{,}625$, que no es mala, pero se puede mejorar. (Un ejemplo interesante de un texto similar, es el último capítulo de *Ulysses*, de James Joyce. Además de las letras, los dígitos y los espacios, este largo capítulo contiene sólamente unos pocos signos de puntuación).

◇ **Ejercicio 1.2 (sol. en pág. 1051):** Un conjunto de caracteres que incluya las 26 letras en mayúsculas y el espacio, puede ser codificado con códigos de 5 bits, pero dejaría cinco códigos sin usar. Sugiérase una forma de utilizarlos.

1.1.3. Compresión de texto ad hoc

A continuación, mostramos algunas sencillas e intuitivas ideas para aquellos casos en los que la compresión debe ser reversible (sin pérdidas):

- Si el texto contiene muchos espacios, pero no están agrupados, se pueden eliminar; sus posiciones, se indican entonces mediante una cadena de bits, que contiene un 0 por cada carácter del texto que no es un espacio y un 1 por cada espacio. Por lo tanto, el texto

  `Aquí hay algunas ideas`,

[1] En inglés, a esta conversión se le llama *case flattening*.

Bits	Bits (posiciones 2 1 0)							
5 4 3	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0		A	B	C	D	E	F	G
1	H	I	J	K	L	M	N	O
2	P	Q	R	S	T	U	V	W
3	X	Y	Z	0	1	2	3	4
4	5	6	7	8	9	+	-	*
5	/	(	)	$	=	sp	,	.
6	≡	[	]	:	≠	_	∨	∧
7	↑	↓	<	>	≤	≥	¬	;

Tabla 1.3: El Código de Visualización CDC.

se codifica en la cadena de bits "000010001000000100000" seguida de

`Aquí`hayalgunasideas.

Si el número de espacios en blanco es pequeño, la cadena de bits será dispersa y se pueden emplear los métodos de la Sección 8.5 para comprimirla considerablemente.

- Dado que los códigos ASCII se codifican esencialmente con 7 bits, el texto puede ser comprimido utilizando 7 bits por carácter —en lugar de 8—, en la secuencia de salida. Esto se puede llamar *compactación*[2]. La razón de compresión es, por supuesto, $7/8 = 0{,}875$.

- Los números $40^3 = 64\,000$ y $2^{16} = 65\,536$, no son muy diferentes, y satisfacen la relación $40^3 < 2^{16}$. Esto puede servir como la base de un método de compresión intuitivo, para un pequeño conjunto de símbolos. Si los datos a comprimir, se componen de un texto con un máximo de 40 caracteres diferentes (como 26 letras, 10 dígitos, un espacio y tres signos de puntuación), entonces este método produce un factor de compresión de $(8 \times 3)/16 = 1{,}5$. He aquí cómo funciona:
  Dado un conjunto de 40 elementos, cogemos una cadena de caracteres del conjunto y la agrupamos en tripletes. Cada carácter puede tomar uno de los 40 valores, por lo que un trío de caracteres puede tener uno de los $40^3 = 64\,000$ valores. Cada uno de esos valores, puede representarse con 16 bits, porque 40^3 es menor que 2^{16}. Sin compresión, cada uno de los 40 caracteres requiere un byte, por lo que nuestro método intuitivo produce un factor de compresión de $24/16 = 3/2 = 1{,}5$. (Este es uno de esos raros casos, en los que el factor de compresión es constante y conocido de antemano).

- Si el texto incluye solamente letras mayúsculas, dígitos y algunos signos de puntuación, se puede utilizar el viejo código de visualización CDC de 6 bits (Tabla 1.3). Este código se utilizaba normalmente en computadoras de segunda generación (e incluso en unas pocas de tercera generación). Estos equipos, no necesitaban más de 64 caracteres, ya que no tenían ningún monitor y enviaban su salida a impresoras, las cuáles solamente podían imprimir un conjunto limitado de caracteres.

- Otro viejo código que merece la pena mencionar es el código de Baudot (Tabla 1.4). Era un código de 5 bits desarrollado por J.M.E. Baudot en torno al año 1880 para la comunicación telegráfica. Se hizo popular, y en 1950 fue designado el Código Internacional de Telégrafos Nº 1. Se utilizaba en muchos equipos de primera y de segunda generación. El código utiliza 5 bits

[2] Packing.

Letras	Código	Figuras	Letras	Código	Figuras
A	10000	1	Q	10111	/
B	00110	8	R	00111	-
C	10110	9	S	00101	SP
D	11110	0	T	10101	na
E	01000	2	U	10100	4
F	01110	na	V	11101	’
G	01010	7	W	01101	?
H	11010	+	X	01001	,
I	01100	na	Y	00100	3
J	10010	6	Z	11001	:
K	10011	(	LS	00001	LS
L	11011	=	FS	00010	FS
M	01011	)	CR	11000	CR
N	01111	na	LF	10001	LF
O	11100	5	ER	00011	ER
P	11111	%	na	00000	na

LS: Letter Shift; FS: Figure Shift.
CR: Carriage Return; LF: Line Feed.
ER: Error; na: Not Assigned; SP: Space.

Tabla 1.4: El código de Baudot.

por carácter, pero codifica más de 32 caracteres. Cada combinación de 5 bits puede ser el código de dos caracteres, una letra y una figura. Los códigos[3] LS (Letter Shift) y FS (Figure Shift), se utilizan para alternar entre letras y figuras[4].
Usando esta técnica, el código Baudot puede representar $32 \times 2 - 2 = 62$ caracteres (cada código puede tener dos significados, excepto LS y FS). La cantidad real de números y de caracteres, no obstante, es más pequeño que el mencionado, porque cinco de los códigos tienen tienen un significado propio[5] (CR, LF, ER, na y SP), y otros pocos no están asignados.
El código de Baudot no es fiable porque no se utiliza un bit de paridad. Un bit erróneo puede transformar un carácter en otro distinto. En particular, un bit erróneo en un carácter shift[6] provoca una interpretación errónea de todos los elementos siguientes, hasta el siguiente shift.

- Si los datos se componen sólo de números enteros, cada dígito decimal se puede codificar con 4 bits (dos dígitos empaquetados en un byte). Los datos que consisten en fechas se pueden representar como el número de días desde el 1 de Enero de 1900 (u otra fecha de inicio conveniente). Cada día puede ser almacenado en 16 ó 24 bits (2 o 3 bytes). Si los datos se componen de pares fecha/hora, una posible compresión es indicar el número de segundos desde una fecha de inicio conveniente. Si se almacena como un número de 32 bits (4 bytes), la representación puede ser suficiente para abarcar unos 136 años.

- El diccionario de datos (o cualquier lista ordenada lexicográficamente), puede comprimirse utilizando el concepto de la compresión prefija[7]. Se basa en la observación de que las palabras

[3] $LS = Cambiar\ a\ letra$ y $FS = Cambiar\ a\ figura$.
[4] Una figura es cualquier carácter que no sea una letra.
[5] $CR = Retorno\ de\ carro$; $LF = Avance\ de\ línea$; $ER = Error$; $na = No\ asignado$ y $SP = Espacio$.
[6] LS o FS.
[7] Front Compression.

Texto	Compresión
a	a
aardvark	1ardvark
aback	1back
abaft	3ft
abandon	3ndon
abandoning	7ing
abasement	3sement
abandonment	3ndonment
abash	3sh
abated	3ted
abate	5
abbot	2bot
abbey	3ey
abbreviating	3reviating
abbreviate	9e
abbreviation	9ion

Tabla 1.5: Compresión prefija.

adyacentes en dicha lista, tienden a compartir algunas de sus características iniciales. Una palabra, por lo tanto, puede comprirse eliminando los n caracteres que comparte con su predecesora en la lista y reemplazándolos por n.
La Tabla 1.5, muestra un pequeño ejemplo tomado de una lista de palabras para crear anagramas. Está claro que es fácil de conseguir una compresión significativa con este sencillo método (véase también [Robinson y Singer 81] y [Nix 81]).

- El procesador de textos MacWrite [Young 85], utiliza un código especial de 4 bits para representar los 15 caracteres más comunes, "␣etnroaisdlhcfp", además del código escape. Cada uno de estos 15 caracteres se codifica con 4 bits. Cualquier otro carácter, se codifica como el código escape seguido de los 8 bits del código ASCII del carácter; un total de 12 bits. Cada párrafo se trata por separado, y si el resultado de la codificación resulta una expansión del original, el párrafo se almacena como texto en ASCII. A cada párrafo, se le añade un bit adicional para indicar si se ha utilizado o no compresión.

El Síndrome del 19/9/89

¿Cómo puede representarse una fecha, como el 12/11/71, en una computadora?. Una manera de hacer esto, es almacenar el número de días desde el 1 de enero de 1900 en una variable entera. Si la variable ocupa 16 bits (incluyendo 15 bits de magnitud y un bit de signo), se producirá un desbordamiento después de $2^{15} = 32\,K = 32\,768$ días, que corresponde al 19 de septiembre de 1989. Esto es precisamente lo que ocurrió ese día en varios equipos (véase el número del mes de enero de 1991 de la revista *Communications of the ACM*). Observe que la duplicación del tamaño de dicha variable a 32 bits habría retrasado el problema hasta después de $2^{31} = 2$ giga días, que se produciría en el otoño del año 5 885 416.

> El principio de parsimonia valora la capacidad de una teoría para comprimir un máximo de información en un mínimo de formalismo. La ecuación $E = m \cdot c^2$, del célebre Einstein, obtiene parte de su bien merecida fama, de la sorprendente riqueza de conocimiento inmersa en su pequeña estructura. Las ecuaciones de Maxwell, las reglas de la mecánica cuántica, e incluso las ecuaciones básicas de la teoría de la relatividad general, igualmente satisfacen el requisito de parsimonia de una teoría fundamental: son lo suficientemente compactas para caber en una camiseta. Como contraste, el proyecto del genoma humano, que requiere la cuantificación de cientos de miles de secuencias genéticas, representa la misma antítesis de la parsimonia.
>
> —Hans C. von Baeyer, Maxwell's Demon[a], 1998
>
> [a]El demonio de Maxwell.

1.2. Codificación run-length[8]

La idea básica de este método es la siguiente: Si un dato d aparece n veces consecutivas en el flujo de entrada, se cambian las n ocurrencias con el par único nd. Las n apariciones consecutivas de un elemento de datos se llama *run length*[9] de n, y este enfoque para la compresión de datos se llama *codificación run-length o RLE*. Aplicamos esta primera idea a la compresión de texto y luego a la compresión de imágenes.

1.3. Compresión de texto RLE

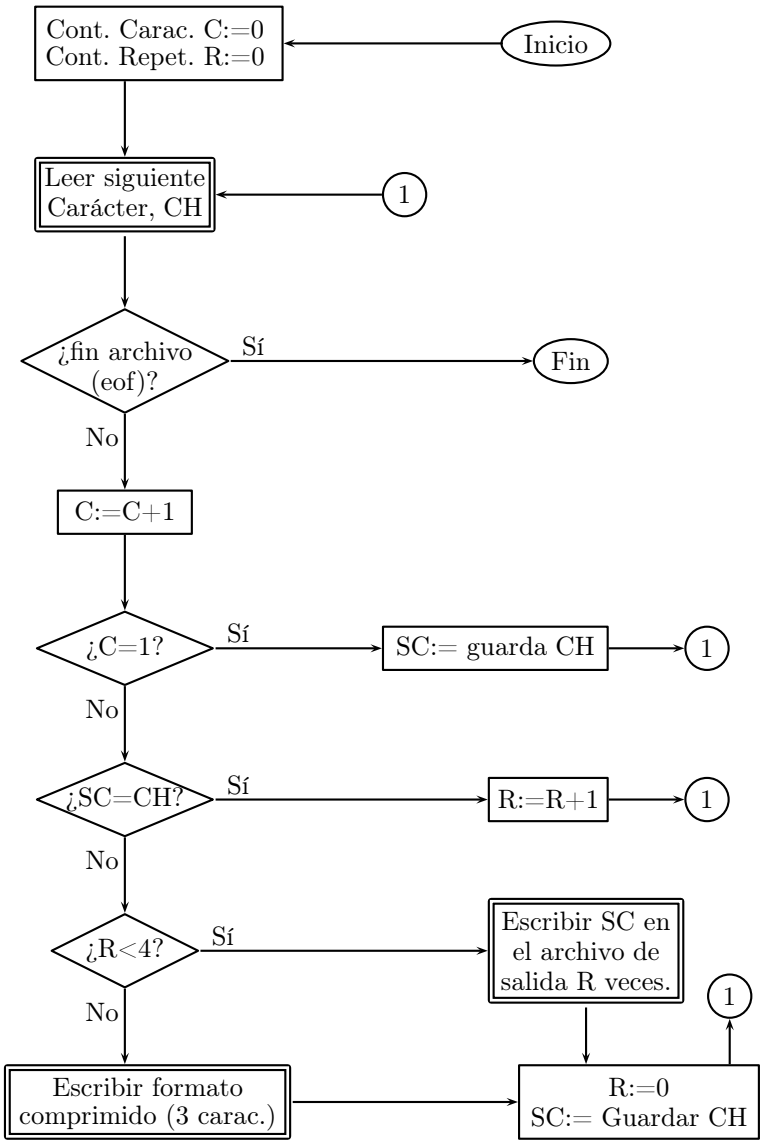
El reemplazo exacto de `2.␣all␣is␣too␣well` con `2.␣a2␣is␣t2␣we2`, es ambiguo y no funciona . Claramente, el descompresor debería tener una manera de expresar que el primer 2 es parte del texto, mientras que los demás indican el número de repeticiones de las letras `o` y `l`. Incluso la cadena `2.␣a2l␣is␣t2o␣we2l`, sigue sin resolver este problema (y además no proporciona compresión alguna). Un camino para resolver este problema es preceder cada repetición con un carácter especial de cambio de código (o código de escape). Si usamos `@` como carácter de cambio de código, entonces la cadena `2.␣a@2l␣is␣t@2o␣we@2l`, puede ser descomprimida sin ambigüedad. Sin embargo, esta cadena es más larga que la original, ya que sustituye dos letras consecutivas, con tres caracteres. Tenemos que adoptar la convención de que sólo se reemplacen por un factor de repetición, aquellos grupos compuestos por tres o más repeticiones de un mismo carácter. La Figura 1.6a es un diagrama de flujo, que explica el funcionamiento de un sencillo compresor de texto *run-length*.

Después de leer el primer elemento —CH—, el contador de caracteres es $C = 1$, por lo que CH se almacena ($SC :=$ guarda CH). Los siguientes elementos se comparan con éste (¿$SC = CH$?), y si son idénticos, se incrementa el contador de repeticiones ($R := R + 1$). Cuando se lee un carácter diferente, la operación depende del valor de R. Si es pequeño, se escribe el carácter grabado —SC— en el archivo comprimido (R veces); se actualiza SC con el último carácter leído, y se reinicia el contador ($R := 0$), para seguir con el siguiente carácter de la cadena. De lo contrario, se escribe el indicador de cambio de código —@—, seguido por el valor de R y el carácter SC; a continuación se pone a cero R, y se guarda el siguiente elemento del flujo de datos a comprimir —en SC—. Todo este ciclo se repite hasta procesar todos los datos.

La descompresión también es sencilla. Se muestra en la Figura 1.6b. Se explora la secuencia de entrada hasta encontrar el carácter de cambio de código —@— (todos los caracteres, hasta la aparición

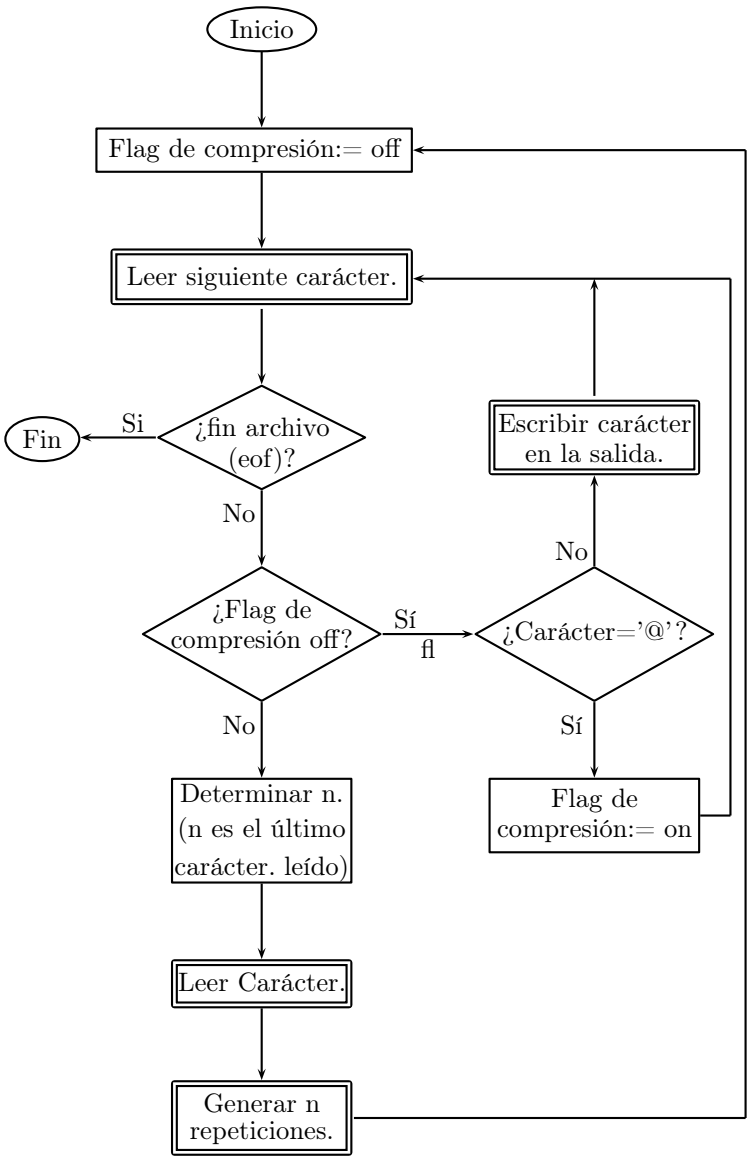
[8]Una codificación basada en la longitud de rachas de elementos iguales.

[9]De ahora en adelante, usaré las palabras en inglés —con o sin guión—, aunque las sutituiré por "Racha, tirada, tramo, recorrido, serie...", cuando no cree confusión.



(a)

Figura 1.6: RLE. Parte I: Compresión.



(b)

Figura 1.6: RLE. Parte II: Descompresión.

de dicho símbolo, se escriben en la salida); inmediatamente después, se lee el contador de repeticiones —n— y el carácter a repetir, escribiendolo en la cadena de salida —n veces—; este proceso se repite hasta que no haya más datos en la entrada. A continuación, exploramos los principales problemas de este método:

1. En un texto de inglés cotidiano, no hay muchas repeticiones. Existen muchos "dobletes", pero los "tripletes", son raros. El carácter más frecuente, es el espacio. Algunas veces, también aparecen repeticiones de guiones o asteriscos. En textos matemáticos, pueden encontrarse dígitos repetidos. El siguiente "párrafo", es un ejemplo artificial.

   > The abbott from Abruzzi accedes to the demands of all abbesses from Narragansett and Abbevilles from Abyssinia. He will accommodate them, abbreviate his sabbatical, and be an accomplished accessory.[a]
   >
   > [a]El Abad de Abruzzi, accede a las demandas de todas las abadesas de Narragansett y Abbevilles de Abisinia. Las aloja, abrevia su año sabático y es un experto cómplice.

2. El carácter "@", puede formar parte del texto de la secuencia de entrada, en cuyo caso se debe elegir un carácter de cambio de código distinto. A veces, los datos de entrada pueden contener todos los caracteres posibles del alfabeto. Un ejemplo, es un archivo objeto —el resultado de la compilación de un programa—. Este archivo contiene las instrucciones máquina, y se puede considerar una cadena de bytes que pueden tener cualquier valor. El método MNP5, que se describe a continuación y en la Sección 2.10, proporciona una solución.

3. Puesto que el número de repeticiones aparece en la secuencia de salida como un byte, la cuenta máxima es de 255. Esta limitación se puede suavizar algo, cuando nos damos cuenta de que la existencia de un contador de repeticiones, significa que hay una repetición (al menos tres caracteres idénticos consecutivos). Podemos adoptar la convención de que un contador de repeticiones a 0, significa la repetición de tres elementos, lo que implica que un contador con un valor de 255, equivale a una secuencia de 258 caracteres idénticos.

   El método MNP —clase 5—, se utilizó para la compresión de datos en los módems antiguos. Ha sido desarrollado por Microcom, Inc., un fabricante de módems (MNP son siglas de Microcom Network Protocol[10]), y utiliza una combinación de los métodos RLE (run-length encoding) y la codificación de frecuencia adaptativa.

> Hay tres clases de mentiras: las mentiras, las grandes mentiras y las estadísticas.
>
> (Atribuido por Mark Twain a Benjamin Disraeli)

La última técnica se describe en la Sección 2.10; sin embargo, a continuación se indica cómo resuelve MNP5 el segundo problema —planteado más arriba—.

Cuando se encuentran tres o más bytes idénticos consecutivos en la secuencia de entrada, el compresor escribe tres copias del byte en la salida, seguidas del número de repeticiones. Cuando el descompresor lee tres bytes idénticos consecutivos, sabe que el siguiente byte es un contador de repeticiones (que puede ser 0, indicando que tan sólo hay tres repeticiones). Una desventaja de este método, es que una secuencia de tres caracteres en la cadena de entrada genera cuatro caracteres de salida: ¡expansión!. Cuatro caracteres seguidos no producen compresión. Sólo se comprime si hay más de cuatro caracteres. Otro pequeño problema, es que el número máximo del contador está limitado artificialmente en MNP5 a 250, en lugar de a 255.

Para tener una idea de las razones compresión producidas por RLE, consideramos una cadena de N caracteres, que necesita ser comprimida. Suponemos que la cadena contiene M repeticiones, con

[10]Protocolo de Red de Microcom.

una longitud media de L elementos cada una. Cada una de las M repeticiones, se sustituye por 3 caracteres (el cambio de código, el contador y el dato), por lo que el tamaño de la cadena comprimida es $N - M \cdot L + M \cdot 3 = N - M \cdot (L - 3)$, y el factor de compresión es:

$$\frac{N}{N - M \cdot (L - 3)}.$$

(Para MNP5, ponemos 4 en vez de 3). Ejemplos: $N = 1000$, $M = 10$, $L = 3$, producen un factor de compresión de ${}^{1000}/_{[1000-10(4-3)]} = 1{,}01$. Un mejor resultado se obtiene en el caso $N = 1000$, $M = 50$, $L = 10$, donde el factor es de ${}^{1000}/_{[1000-50(10-3)]} = 1{,}538$.

Una variante de la codificación *run-length* para textos es la *codificación de digramas*. Este método es adecuado cuando el número de elementos distintos del bloque a comprimir se reduce sólo, a un número limitado de caracteres; e.g., solamente las letras, los dígitos y los signos de puntuación. La idea es identificar pares de ocurrencias comunes de caracteres, y sustituir cada par (un digrama), por un carácter que nunca pueda aparecer en los datos de entrada (e.g., uno de los caracteres de control ASCII). Se pueden obtener buenos resultados, si es posible analizar los datos de antemano. Sabemos que en inglés común ciertos pares de caracteres, tales como `E␣`, `␣T`, `TH` y `␣A`, aparecen con frecuencia. Otros tipos de datos, pueden formar diferentes digramas. El método *sequitur* de la Sección 8.10, es un ejemplo de una forma de comprimir datos, localizando repeticiones de digramas (así como las frases más frecuentes) y sustituyéndolos por símbolos especiales.

Una variante similar, es la *sustitución de patrones*. Es adecuada para la compresión de programas informáticos, donde ciertas palabras, como `for`, `repeat` y `print`, son muy comunes. Cada una de ellas, se sustituye por un carácter de control, o si hay muchas, por un carácter de cambio de código —@, en el ejemplo—, seguido de un carácter de código —`a`, en el ejemplo—. Suponiendo que se le asigna el código `a` a la palabra `print`, el texto: `m :␣ print, b, a;` será comprimido a `m :␣ @a, b, a;`.

1.3.1. Codificación relativa

Esta es otra variante, a veces llamada *diferenciación* ([Gottlieb et al. 75]). Se utiliza cuando los datos a comprimir, están formados por una serie de números que no difieren en mucho entre sí (e.g., en la telemetría); o bien cuando se componen de cadenas similares. El último caso, se utiliza en la compresión de datos para envío por fax descrita en la Sección 2.13 y también en la compresión LZW (Sección 3.12.4).

En la telemetría, se utiliza un detector para recopilar datos a determinados intervalos y transmitirlos a una central para su posterior procesamiento. Un ejemplo, es el estudio de la temperatura de un lugar, en el que se realizan mediciones cada hora. Dos temperaturas sucesivas, no difieren mucho normalmente, por lo que el sensor necesita enviar sólo la primera de ellas, seguida por las diferencias. De este modo, la secuencia de temperaturas: 70, 71, 72,5, 73,1,..., puede representarse con esta otra: 70, 1, 1,5, 0,6,.... Debido a que las diferencias son pequeñas, se pueden utilizar menos bits para guardar cada número, lo que permite la compresión de los datos.

Tenga en cuenta que las diferencias pueden ser negativas, y en ocasiones, pueden ser grandes. En este último caso, el compresor envía el valor real de la medida siguiente, en lugar de la diferencia. Así pues, la secuencia 110, 115, 121, 119, 200, 202,... se puede comprimir en esta otra: 110, 5, 6, −2, 200, 2,.... Pero desgraciadamente, ahora tenemos que distinguir entre una diferencia y un valor real. Esto puede hacerlo el compresor mediante la adición de un bit extra —un flag[11]— para cada número enviado, acumulando esos bits y enviándolos al descompresor cada cierto tiempo, como parte de la transmisión. Suponiendo que cada diferencia se manda como un byte, el compresor debería posponer —o preceder— cada grupo de 8 bytes, con un byte formado por sus 8 flags.

[11] Indicador, bandera, código señalizador.

Otra forma práctica de enviar las diferencias mezcladas con los valores reales, es enviar pares de bytes. Cada par es, o bien una medida real de 16 bits (positiva o negativa) o bien dos diferencias con signo de 8 bits. Por lo tanto, las medidas reales pueden estar entre 0 y $\pm 32K$, y las diferencias entre 0 y ± 127. Para cada par, el compresor crea un código: 0 si el par es un valor real; 1 si es un par de diferencias. El compresor envía 16 pares, y tras ellos, los 16 bits de flag correspondientes.

Ejemplo: La secuencia de mediciones 110, 115, 121, 119, 200, 202,..., se envía como: (110), $(5, 6)$, $(-2, -1)$, (200), $(2, \ldots)$, donde cada par de paréntesis, muestra un par de bytes. Un -1 tiene el valor: 11111111_2, que es ignorado por el descompresor (indica que, en el par en el que se encuentra, sólo hay una diferencia). Mientras se envía esta información, el compresor prepara los *flags* —01101...—, que manda tras los primeros 16 pares.

La codificación relativa puede generalizarse, y así convertirse en un algoritmo de compresión con pérdida; entonces recibe el nombre de *codificación diferencial*. Un ejemplo de uso del método de codificación diferencial, es la modulación por codificación de pulsos diferencial (o DPCM[12] —Sección 4.26—).

1.4. Compresión de imágenes RLE

RLE es un candidato natural para la compresión de datos gráficos. Una imagen digital, se compone de pequeños puntos llamados *píxeles*. Cada píxel puede ocupar, uno o varios bits; con un bit, sólo se pueden representar dos colores —normalmente blanco o negro—; con n bits por píxel, el número de tonalidades disponibles aumenta a 2^n, colores o tonos de gris. Suponemos que los píxeles se almacenan en la memoria en una matriz llamada *mapa de bits* (*bitmap*); en la cual, por lo tanto, se tienen los datos de entrada de la imagen. Los píxeles se disponen normalmente en el mapa de bits en *líneas de exploración* (*scan lines*), de manera que el primer píxel del mapa de bits, es el punto situado en la esquina superior izquierda de la imagen, y el último, es el situado en la esquina inferior derecha.

La compresión de una imagen mediante RLE, se basa en la observación de que, al seleccionar un píxel de la imagen al azar, existe una probabilidad muy alta de encontrar otros adyacentes a él —sus vecinos— con el mismo color (véanse también, las Secciones 4.30 y 4.32). El compresor, por lo tanto, explora el mapa de bits fila por fila, en busca de secuencias de píxeles con el mismo color. Si el bitmap comienza, e.g., con 17 píxeles blancos, seguidos de 1 píxel negro, 55 blancos, etc, entonces sólo es necesario escribir, como datos de salida, los números: 17, 1, 55,....

El compresor asume que el bitmap comienza con píxeles blancos. Si esto no es cierto, se considera entonces que el mapa de bits empieza con cero píxeles blancos, y se indica en la secuencia de salida, escribiendo un 0 como primer *run length*. La resolución del bitmap también debería guardarse al principio de dicha secuencia.

El tamaño de la cadena comprimida, depende de la complejidad de la imagen. A más detalle, peor compresión. De todos modos, en la Figura 1.7, se muestra cómo escanear cuando la región es uniforme. Una línea de exploración entra a través de un punto del perímetro de la región y sale a través de otro punto; estos dos puntos, no forman parte de ninguna otra línea. Ahora es claro, que el número de líneas de exploración que atraviesan una región uniforme, es prácticamente igual a la mitad de la longitud —medida en píxeles— de su perímetro. Puesto que la región es uniforme, cada línea de exploración contribuye con dos *runs* en la cadena de salida, para cada zona que atraviesa. La razón de compresión de una región uniforme, por lo tanto, es más o menos igual a:

$$\frac{2 \times \text{la mitad de la longitud del perímetro}}{\text{número total de píxeles en la región}} = \frac{\text{perímetro}}{\text{área}}.$$

[12] Differential Pulse Code Modulation.

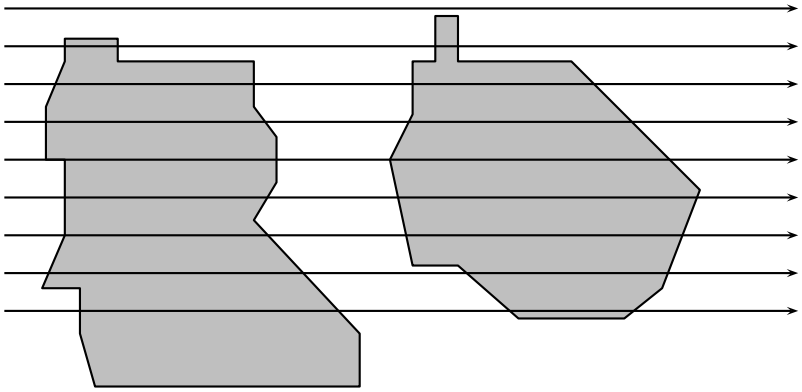
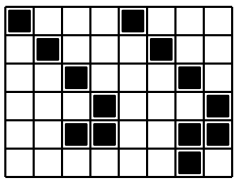


Figura 1.7: Superficies uniformes y muestreo de líneas.

◇ **Ejercicio 1.3 (sol. en pág. 1051):** ¿Cuál sería el archivo comprimido para el siguiente bitmap de 6×8?



RLE también puede utilizarse para comprimir imágenes en escala de grises. Cada bloque (*run*) de píxeles con la misma intensidad (nivel de gris) se codifica como un par (*run length*, valor del píxel). El *run length*, ocupa normalmente un byte, lo que permite secuencias de hasta 255 píxeles. El valor del píxel ocupa varios bits, dependiendo del número de niveles de gris (típicamente, entre 4 y 8 bits).

Ejemplo: Un mapa de bits, en escala de grises de 8 bit de profundidad, que comienza con

$$12, 12, 12, 12, 12, 12, 12, 12, 12, 35, 76, 112, 67, 87, 87, 87, 5, 5, 5, 5, 5, 5, 1, \ldots$$

se comprime como $\boxed{9}$,12, 35, 76, 112, 67,$\boxed{3}$,87, $\boxed{6}$,5, 1,..., donde los números en las cajas son contadores que indican la cantidad de veces que se repite el siguiente elemento. El problema, es distinguir entre un byte que contiene un valor de escala de grises —como 12—, y uno que contiene un recuento —por ejemplo, $\boxed{9}$—. Éstas son algunas de las soluciones (aunque no las únicas posibles):

1. Si la imagen está limitada a sólo 128 escalas de grises, podemos dedicar un bit de cada byte para indicar si el byte contiene un valor de escala de grises o el número de apariciones del siguiente elemento.

2. Si el número de escalas de grises es de 256, se puede reducir a 255 y reservar un valor de flag que preceda a cada uno de los bytes de recuento. Si el flag es, digamos, 255, entonces la secuencia de arriba se convierte en

$$255, 9, 12, 35, 76, 112, 67, 255, 3, 87, 255, 6, 5, 1, \ldots$$

3. Una vez más, se dedica un bit de cada byte para indicar si el byte contiene un valor de escala de grises o un contador. Esta vez, sin embargo, estos bits extra se acumulan en grupos de 8 y

cada grupo se escribe en la salida precediendo (o posponiendo) a los 8 bytes correspondientes. Ejemplo: la secuencia $\boxed{9}$,12, 35, 76, 112, 67, $\boxed{3}$,87, $\boxed{6}$,5, 1,... se convierte en

$$\boxed{10000010}, 9, 12, 35, 76, 112, 67, 3, 87, \boxed{100\ldots}, 6, 5, 1, \ldots.$$

El coste de los bytes extra —formados por un bit por cada byte de la cadena de salida— es, por supuesto, $1/8$ del total, por lo que el incremento de tamaño es de un $(1/8 \cdot 100)\ \% = 12{,}5\,\%$.

4. Un grupo de m píxeles diferentes entre sí, se precede de un byte con el valor negativo $-m$. La secuencia de arriba se codifica como

$$9, 12, -4, 35, 76, 112, 67, 3, 87, 6, 5, ?, 1, \ldots$$

(el valor del byte con ? es positivo o negativo en función de lo que haya tras el 1). El peor caso, es una secuencia de píxeles (p_1, p_2, p_2), repetida n veces por todo el mapa de bits. Cada una, se codifica como $(-1, p_1, 2, p_2)$, ¡cuatro números, en lugar de los tres originales!. Si cada píxel requiere un byte, entonces el original —de tres bytes—, se expande a cuatro bytes. Si cada píxel se representa con tres bytes, los tres píxeles originales —un total de 9 bytes—, se comprimen en $1 + 3 + 1 + 3 = 8$ bytes.

Hay que considerar tres cuestiones más:

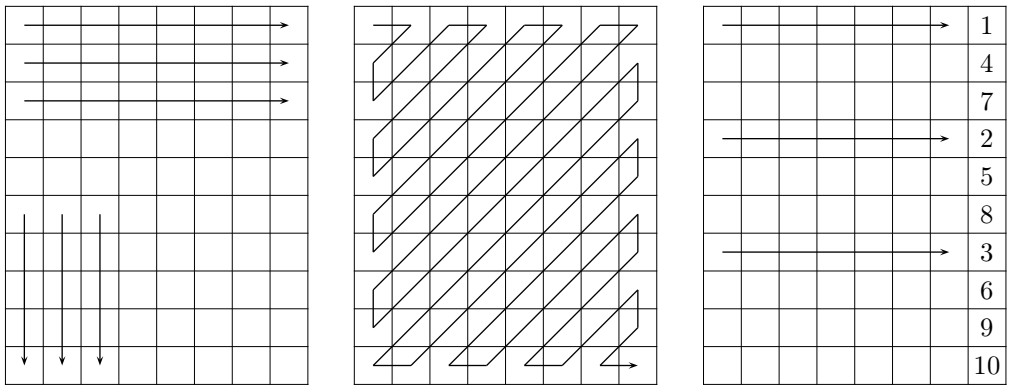
1. Dado que el tamaño del *run length* no puede ser 0, no tiene sentido escribir [*run length* menos uno] en la secuencia de salida. Así, el par $(3, 87)$ representa a un grupo de *cuatro* píxeles con una intensidad de 87. De esta manera, un bloque puede tener hasta 256 píxeles de longitud.

2. En las imágenes en color, es común que cada pixel se almacene como tres bytes, que representan la intensidad de los componentes —rojo, verde y azul— del píxel. En tal caso, los grupos de cada color deberían ser codificados por separado. Así, los píxeles $(171, 85, 34)$, $(172, 85, 35)$, $(172, 85, 30)$ y $(173, 85, 33)$, han de dividirse en tres secuencias: $(171, 172, 172, 173, \ldots)$, $(85, 85, 85, 85, \ldots)$ y $(34, 35, 30, 33, \ldots)$. Cada secuencia debe ser una codificación *run-length*, distinta de las demás . Esto significa, que cualquier método para comprimir imágenes en escala de grises, también se puede aplicar a imágenes en color.

3. Es preferible codificar cada fila del mapa de bits de forma individual. Así, si una fila termina con cuatro píxeles de intensidad 87, y la línea siguiente comienza con 9 píxeles similares a los anteriores, es mejor escribir $\ldots, 4, 87, 9, 87, \ldots$, en la secuencia de salida, en lugar de $\ldots, 13, 87, \ldots$. Es incluso mejor, escribir la secuencia $\ldots, 4, 87, eol, 9, 87, \ldots$, donde "*eol*" es un código especial que indica el final de una línea —End Of Line—. La razón es, que a veces el usuario puede decidir aceptar o rechazar una imagen, examinando su forma general —sin considerar los detalles—. Si cada línea está codificada de forma individual, el algoritmo de decodificación puede empezar por decodificar y mostrar las líneas $1, 6, 11, \ldots$, continuar con las líneas $2, 7, 12, \ldots$, etc. Las filas individuales de la imagen, se entrelazan, y la imagen se construye en la pantalla poco a poco, en varios pasos. De esta manera, es posible hacerse una idea de lo que representa la imagen en una etapa temprana. La Figura 1.8c, muestra un ejemplo de este tipo de exploración.

Otra de las ventajas de la codificación individual de filas es la posibilidad de extraer sólo una parte de una imagen codificada —como las filas de la k a la l–. Otra aplicación es la de combinar dos imágenes comprimidas sin tener que descomprimirlas antes.

Si se adopta esta idea (la codificación de cada fila del bitmap de forma individual), entonces la cadena comprimida debe contener información sobre el lugar de comienzo de cada fila del mapa de bits en esa cadena. Esto se puede hacer escribiendo un encabezado precediendo a la misma, formado por un grupo de 4 bytes —32 bits— por cada fila del bitmap. El grupo $k - ésimo$ contiene el desplazamiento



(a) (b) (c)

Figura 1.8: Formas de muestreo en RLE.

—en bytes— desde el principio de la cadena hasta el comienzo de la información para la fila k. Esto incrementa el tamaño de la secuencia comprimida, pero todavía puede ofrecer una buena compensación entre espacio (tamaño de la cadena comprimida) y tiempo (para decidir si aceptar o rechazar la imagen).

◇ **Ejercicio 1.4 (sol. en pág. 1052):** Hay otra razón obvia, por la que cada fila de un bitmap debería ser codificada individualmente. ¿Cuál es?

La Figura 1.9a (pág. 34) muestra un programa en Matlab[13] para calcular los *run lengths* de una imagen binivel. El código es muy sencillo. Comienza por la conversión de la matriz —`M`— en un vector unidimensional —`x`—; para ello, guarda en `f` y `c` el número de filas y de columnas, respectivamente —$[\texttt{f},\texttt{c}] = \texttt{size}\,(\texttt{M})$—, y con el bucle `for` van introduciendo en x las filas de `la` matriz —$\texttt{M}\,(\texttt{k},:)$—, una tras otra. Cada *run length*, continúa de una línea a la siguiente y se calculan todos a partir de `x`, mediante las siguientes operaciones:

- $\texttt{N} = \texttt{f} * \texttt{c}$, calcula el número de elementos de la matriz, `N`.
- La suma de los elementos —de 2 a `N`— de `x` ($\texttt{x}\,(\texttt{2} : \texttt{N})$), con los elementos —de 1 a $\texttt{N} - \texttt{1}$— de `x` ($\texttt{x}\,(\texttt{1} : \texttt{N} - \texttt{1})$), proporciona el vector `z`, que contendrá unos en los lugares clave[14].
- Con la búsqueda de los unos en z, obtenemos una matriz unidimensional `z1` ($\texttt{z1} = \texttt{find}(\texttt{z} == \texttt{1})$), con la que se construyen dos vectores: el primero —`i1`—, formado por todos los elementos de `z1` con `N` al final ($\texttt{i1} = [\texttt{z1}\ \texttt{N}]$); el segundo —`i2`—, formado por todos los elementos de $z1$ precedidos por 0 ($\texttt{i2} = [\texttt{0}\ \texttt{z1}]$).
- La resta $\texttt{i1} - \texttt{i2}$ genera todos los *run lengths* de la matriz `M`.

La desventaja del algoritmo RLE para imágenes consiste en que cuando se modifica la imagen, normalmente los *run lengths* tienen que ser reconstruidos por completo. La salida proporcionada por el método RLE en imágenes complejas, a veces puede ser más grande que el almacenamiento de la imagen sin comprimir (i.e., una imagen sin comprimir —un volcado píxel a píxel del bitmap original—).

[13] Si no se dispone del Matlab existen clones gratuítos —como "Octave"—, disponible en la url http://www.gnu.org/software/octave/; (añádase *endfunction* para determinar la finalización de la función tras la asignación de R.)

[14] El primer elemento de x es $x\,(1)$.

```
% Devuelve los run lengths de
% una matriz de 0s y 1s
function R=runlengths(M)
[f,c]=size(M);
for k=1:f;
  x(c*(k-1)+1:c*k)=M(k,:);
end
N=f*c;
y=x(2:N);
u=x(1:N-1);
z=y+u;
z1=find(z==1);
i1=[z1 N];
i2=[0 z1];
R=i1-i2;
```

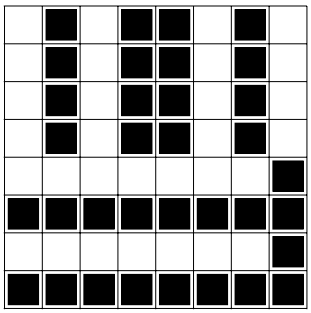
El test:

```
M=[0 0 0 1; 1 1 1 0; 1 1 1 0]
runlengths(M)
```

produce:

```
3     4     1     3     1
```

(a)



(b)

Figura 1.9: (a) Código en Matlab; cálculo de *run lengths*. (b) Un *bitmap*.

Supongamos que tenemos un gráfico con muchas líneas verticales. Cuando se escanea en horizontal, produce *run lengths* muy cortos, dando lugar a una compresión muy mala, o incluso a una expansión. Lo más práctico es, que el compresor RLE para imágenes sea capaz de muestrear el mapa de bits —por filas, por columnas o en zigzag— (Figura 1.8a,b —pág. 33—), para elegir automáticamente la manera idónea de recogida de datos del bitmap, y lograr así la mejor compresión.

◇ **Ejercicio 1.5 (sol. en pág. 1052):** Dado el bitmap de 8×8 de la Figura 1.9b (pág. 34), utilice el algoritmo RLE para comprimirlo, cogiendo primero muestras, fila por fila, y luego columna por columna. Descríbanse los resultados en detalle.

1.4.1. Compresión de imágenes con pérdidas[15]

Es posible obtener tasas de compresión mucho mejores si se ignoran los grupos de repeticiones (*runs*) cortos. Con este método se pierde información al comprimir, pero a veces esto es aceptable para el usuario. (Las imágenes que no permiten ninguna pérdida de datos son las radiografías médicas y las fotografías tomadas por los grandes telescopios, donde el precio de una imagen es astronómico).

[15] En general, sólo es útil para imágenes monocromáticas.

Un algoritmo de codificación *run-length* con pérdidas, debería comenzar por preguntar al usuario por la longitud del *run* más largo que puede ser ignorado. Si el usuario especifica, por ejemplo, 3, el programa fusiona todos los *runs* de 1, 2, ó 3 píxeles idénticos, con sus vecinos. Los datos comprimidos "$6, 8, 1, 2, 4, 3, 11, 2$" podrían guardarse, en este caso, como "$6, 8, 7, 16$", donde 7 es la suma $1 + 2 + 4$ (tres *runs* fusionados) y 16 es la suma $3 + 11 + 2$. Esto tiene sentido para imágenes de alta resolución grandes, donde la pérdida de algunos detalles pueden ser invisibles para el ojo; sin embargo, puede reducir significativamente el tamaño de la secuencia de salida (véase también el Capítulo 4).

1.4.2. Imagen condicional RLE

La compresión de faxes (Sección 2.13), utiliza un código Huffman modificado, pero puede considerarse también una modificación de RLE. Esta sección analiza otra variación de RLE, propuesta en [Gharavi 87]. Suponiendo que tenemos una imagen con n escalas de grises, el método se inicia con la asignación de un código de n bits para cada pixel, que depende de sus vecinos cercanos. A continuación, concatena los códigos de n bits en una cadena larga y calcula los *run lengths*. A cada *run length* se le asigna un código prefijo (Huffman u otros —Sección 2.3—) que pasa a formar parte de los datos comprimidos.

El método considera, que cada línea de exploración en la imagen es un modelo de Markov de segundo orden. En este modelo, el valor del elemento de datos actual depende sólo de dos de sus vecinos del pasado —no necesariamente los dos más inmediatos—. La Figura 1.10 (pág. 36) muestra la elección de dos elementos adyacentes —A y B—, utilizada por nuestro método para predecir el píxel actual —X— (compárese con el modo *sin pérdidas* de JPEG, Sección 4.8.5). Se usa un conjunto de imágenes de entrenamiento, para contar —para cada posible par de valores de los vecinos A,B— cuántas veces aparece cada valor de X. Si A y B tienen valores parecidos, es natural esperar que X tenga un valor similar. Si A y B tienen valores muy diferentes, esperamos que X tenga muchos valores distintos, cada uno con una baja probabilidad. Los recuentos, por lo tanto, producen las probabilidades condicionales $P(X \mid A, B)$ (la probabilidad de que el pixel actual tenga el valor X sabiendo que sus vecinos tienen valores A y B). La Tabla 1.11 (pág. 36), muestra una pequeña parte de los resultados obtenidos al contar de esta manera varias imágenes de entrenamiento de 4 bits por píxel.

A cada píxel de la imagen a comprimir, se le asigna un nuevo código de 4 bits que depende de su probabilidad condicional —como se indica en la tabla—. Supongamos que el píxel actual X tiene valor 1, y sus vecinos tienen valores $A = 3$, $B = 1$. La tabla indica que la probabilidad condicional $P(1 \mid 3, 1)$ es elevada, por lo que X debería asignarse a un nuevo código de 4 bits con pocos *runs* (i.e., códigos que contienen 1s ó 0s consecutivos). Por otra parte, el mismo $X = 1$ —con los vecinos $A = 3$ y $B = 3$— se puede asignar a un nuevo código de 4 bits con muchos *runs*, ya que la tabla indica que la probabilidad condicional $P(1 \mid 3, 3)$ es baja. El método usa, por lo tanto, las probabilidades condicionales para detectar píxeles coincidentes (de ahí el nombre *RLE condicional*), y les asigna códigos con pocos *runs*.

Examinando los 16 códigos de cuatro bits —desde W_1 hasta W_{16}—, encontramos que los dos códigos —0000 y 1111— tienen un *run* cada uno, mientras que 0101 y 1010 tienen cuatro cada uno. Los códigos deberían colocarse en orden creciente de *runs*. Para los códigos de 4 bits obtenemos estos cuatro grupos:

1. De W_1 a W_2: 0000, 1111,
2. De W_3 a W_8: 0001, 0011, 0111, 1110, 1100, 1000,
3. De W_9 a W_{14}: 0100, 0010, 0110, 1011, 1101, 1001,
4. De W_{15} a W_{16}: 0101, 1010.

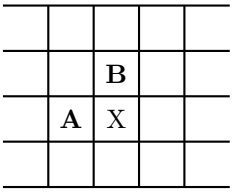


Figura 1.10: Vecinos usados para predecir X.

A	B		W_1	W_2	W_3	W_4	W_5	W_6	$W_7 \ldots$
		Valor X:	4	3	10	0	6	8	1 ...
2	15	Cuenta:	21	6	5	4	2	2	1 ...
		Valor X:	0	1	3	2	11	4	15 ...
3	0	Cuenta:	443	114	75	64	56	19	12 ...
		Valor X:	1	2	3	4	0	5	6 ...
3	1	Cuenta:	1139	817	522	75	55	20	8 ...
		Valor X:	2	3	1	4	5	6	0 ...
3	2	Cuenta:	7902	4636	426	264	64	18	6 ...
		Valor X:	3	2	4	5	1	6	7 ...
3	3	Cuenta:	33927	2869	2511	138	93	51	18 ...
		Valor X:	4	3	5	2	6	7	1 ...
3	4	Cuenta:	2859	2442	240	231	53	31	13 ...

Tabla 1.11: Recuento condicional para píxeles de 4 bits.

Los códigos del grupo i, tienen i *runs* cada uno. Los códigos de cada grupo se seleccionan de tal manera que los elementos en la segunda mitad de un grupo son los complementos de aquellos de la primera mitad.

◇ **Ejercicio 1.6 (sol. en pág. 1052):** Aplíquese este principio a la construcción de los 32 códigos de cinco bits.

Ahora se puede describir el método en detalle. La imagen es muestreada en zigzag, de arriba abajo (en orden *raster*). Para cada pixel X, se localizan sus vecinos A y B, y se busca el triplete correspondiente en la tabla. Si el trío se encuentra en la columna i, se selecciona el código W_i. El primer píxel no tiene vecinos, así que su valor es i; se le asocia el código W_i. Si X está situado en la fila de arriba de la imagen, tiene un vecino A —pero no uno B—, por lo que este caso se trata de manera diferente. Imagine un pixel $X = 2$ en la fila superior, con un vecino $A = 3$. En este caso, se examinan todas las filas de la tabla con $A = 3$, y se selecciona el que tenga el mayor valor de cuenta para $X = 2$. En nuestro ejemplo, es la fila con la cuenta 7902, por lo que se elige el código W_1. Los píxeles X, que no tienen vecinos A (los de la columna de la izquierda), se tratan de manera similar.

Regla de complementación: Después de haber seleccionado el código para X, se compara con el código anterior. Si el bit menos significativo del código anterior es 1, se complementa el código actual. Es de suponer, que esto reducirá el número de *runs*. Como ejemplo, considere la típica secuencia de códigos de 4 bits: W_2, W_4,W_1,W_6,W_3,W_2,W_1

1111, 0011, 0000, 1110, 0001, 1111, 0000.

Cuando estos códigos se concatenan, la cadena resultante —de 28 bits—, consta de ocho *runs*. Después de la aplicación de la regla anterior, los códigos se convierten en una cadena con tan solo seis *runs*:

A	B		W_1	W_2	W_3	W_4	W_5	W_6	$W_7 \ldots$
		Valor X:	4	3	10	0	6	8	1 . . .
2	15	Código:	01	00	111	110	1011	1010	10010 . . .
		Valor X:	0	1	3	2	11	4	15 . . .
3	0	Código:	11	10	00	010	0110	011111	011101 . . .
		Valor X:	1	2	3	4	0	5	6 . . .
3	1	Código:	0	11	100	1011	10100	101010	10101111 . . .
		Valor X:	2	3	1	4	5	6	0 . . .
3	2	Código:	0	11	100	1011	10100	101010	10101111 . . .
		Valor X:	3	2	4	5	1	6	7 . . .
3	3	Código:	0	11	100	1011	101001	1010000	10101000 . . .
		Valor X:	4	3	5	2	6	7	1 . . .
3	4	Código:	11	10	00	0111	0110	0100	010110 . . .

Tabla 1.12: Códigos prefijo para píxeles de 4 bits.

1111, 1100, 0000, 1110, 0001, 0000, 0000.

◇ **Ejercicio 1.7 (sol. en pág. 1052):** Hágase lo mismo para la secuencia de códigos: W_1, W_2,W_3,W_6,W_1,W_4,W_2.

Una variación de este método utiliza los recuentos de la Tabla 1.11 (pág. 36), pero no sus códigos y sus *run lengths*. En su lugar, se asigna un código prefijo al píxel actual —X— que depende de sus vecinos A y B. La Tabla 1.12 (pág. 37) es un ejemplo. Cada fila tiene un conjunto diferente de códigos prefijo, construídos acorde con las cuentas de la fila.

1.4.3. El formato BinHex 4.0

BinHex 4.0 es un formato de archivo para la transferencia fiable de archivos, diseñado por Yves Lempereur para utilizarlo en ordenadores Macintosh. Antes de profundizar en los detalles del formato, el lector debe entender por qué tal formato es útil. ASCII es un código de 7 bits. Cada carácter está codificado como un número de 7 bits, lo que permite 128 caracteres en la tabla ASCII. El estándar ASCII recomienda añadir un octavo bit de paridad por carácter para obtener una mayor fiabilidad. Sin embargo, no especifica la paridad impar o par, y muchos equipos simplemente ignoran el bit extra o incluso lo ponen a 0. Como resultado, cuando los archivos se transfieren en una red de ordenadores, algunos programas de transferencia pueden ignorar el octavo bit y transferir sólo siete bits por carácter. Esto no es tan malo cuando el archivo es de texto, pero si es binario, los bits no deben ser ignorados. Por ello, es más seguro transferir archivos de texto que archivos binarios en las redes informáticas.

La idea de BinHex es convertir cualquier fichero a un archivo de texto. El programa BinHex lee un archivo de entrada (de texto o binario) y produce un archivo de salida con el siguiente formato:

1. El comentario[16]:

   (`This␣file␣must␣be␣converted␣with␣BinHex␣4.0`)

2. Una cabecera, incluyendo los ítems listados en la Tabla 1.13 (pág. 38).

[16]Este archivo debe ser convertido con BinHex 4.0.

Campo	Tamaño
Length of FileName (Longitud de ...) (1–63)	byte
FileName (Nombre del archivo)	("Length" bytes)
Version	byte
Type	long
Creator	long
Flags (And $F800)	word
Length of Data Fork (Longitud de ...)	long
Length of Resource Fork (Longitud de ...)	long
CRC[a]	word
Data Fork (Datos propios del archivo)	("Data Length" bytes)
CRC	word
Resource Fork (Recursos usados por el archivo)	("Rsrc Length" bytes)
CRC	word

[a]Código de redundancia cíclica para corrección de errores.

Tabla 1.13: La cabecera de BinHex.

3. Entonces —como primer paso— lee el archivo de entrada y aplica el algoritmo RLE. El carácter 90_{16} se utiliza como marcador RLE; los siguientes ejemplos hablan por sí solos:

Cadena fuente	Cadena empaqueada
00 11 22 33 44 55 66 77	00 11 22 33 44 55 66 77
11 22 22 22 22 22 22 33	11 22 90 06 33
11 22 90 33 44	11 22 90 00 33 44

(El carácter 00 indica ninguna secuencia de elementos adyacentes idénticos.) Esta forma de codificación se utiliza para los *runs* de 3 a 255 caracteres.

◇ **Ejercicio 1.8 (sol. en pág. 1052):** ¿Cómo se codifica la cadena "11 22 90 00 33 44"?

4. A continuación, se produce la codificación en caracteres ASCII de 7 bits. El archivo de entrada se considera una secuencia de bits. A medida que se leen los datos, se dividen en bloques de 6 bits, y cada bloque se utiliza como un puntero a la tabla BinHex de más abajo. El carácter al que apuntan en esta tabla, se escribe en el archivo de salida. La tabla es:
   `!"#$%&'()*+,-012345689@ABCDEFGHIJKLMNPQRSTUVXYZ[‘abcdefhijklmpqr`

El archivo de salida se organiza en "líneas" de 64 caracteres cada una —excepto, quizás, la última línea—. Cada línea es precedida y seguida por un par de dos puntos "**:**". La elección de los elementos de la tabla no ha sido aleatoria; el diseñador lo aclara en la siguiente cita:

> *"Los caracteres de esta tabla han sido seleccionados para obtener la máxima protección contra el ruido."*

◇ **Ejercicio 1.9 (sol. en pág. 1052):** Conviértase manualmente la cadena "123ABC" a BinHex. Ignorar el comentario y el encabezado de archivo.

1.4.4. Archivos de imagen BMP

BMP es el formato nativo para los archivos de imagen del sistema operativo Windows de Microsoft. Se ha modificado varias veces desde su creación, pero se ha mantenido estable desde la versión 3 de

Windows. BMP es un formato de archivo de gráficos basados en paleta, para imágenes con 1, 2, 4, 8, 16, 24 ó 32 planos de bits. Para comprimir imágenes con 4 u 8 planos de bits, utiliza un algoritmo RLE básico . El formato de un archivo BMP es sencillo. Comienza con un encabezado de archivo —que contiene los dos bytes **BM** y el tamaño del archivo—; a continuación, un encabezado de imagen —con la anchura, la altura y el número de planos de bits— (hay dos formatos diferentes para esta cabecera). Tras las dos cabeceras, la paleta de colores (que puede estar en uno de los tres formatos posibles), seguida por los píxeles de la imagen —en formato `RAW`[17], o comprimidos mediante RLE—. Se puede encontrar información detallada sobre el formato de archivo BMP en, por ejemplo, [Miano 99] y [Swan 93]. Esta sección describe la versión particular de RLE utilizada por BMP para comprimir píxeles.

Para imágenes con ocho planos de bits, los píxeles comprimidos se organizan en pares de bytes. El primer byte de un par es un contador C, y el segundo byte, es un píxel de valor P que se repite C veces. Así, el par $04_{16}\,02_{16}$ se expande en los cuatro píxeles $02_{16}\,02_{16}\,02_{16}\,02_{16}$. Un contador a 0, se traduce como un carácter de cambio de código, y su significado depende del byte siguiente. Un byte cero seguido por otro cero indica *fin de línea*. El resto de la fila de la imagen actual, se llena con píxeles 00 según sea necesario. Un byte cero seguido de 01_{16} señala el final de la imagen. El resto de la imagen se llena con píxeles 00. Un byte cero seguido de 02_{16} indica un salto a otra posición en la imagen. Un par $00_{16}\,02_{16}$, debe ir seguido de 2 bytes que determinan cuántas columnas y filas hay que saltar para alcanzar el siguiente píxel distinto de cero. Todos los píxeles saltados se llenan con ceros. Un byte cero seguido de un byte C mayor que 2, indica C píxeles en formato raw. Es indispensable que los C píxeles se encuentren a continuación de dicho par. Considerando una imagen de 4×8 píxeles —de 8 bits cada uno—, la secuencia:

$$04_{16}\,02_{16},\ 00_{16}\,04_{16}\,a35b1247_{16},\ 01_{16}\,f5_{16},\ 02_{16}\,e7_{16},\ 00_{16}\,02_{16}\,0001_{16},$$
$$01_{16}\,99_{16},\ 03_{16}\,c1_{16},\ 00_{16}\,00_{16},\ 00_{16}\,04_{16}\,08926bd7_{16},\ 00_{16}\,01_{16}$$

es la representación comprimida de estos 32 píxeles:

02	02	02	02	$a3$	$5b$	12	47
$f5$	$e7$	$e7$	00	00	00	00	00
00	00	99	$c1$	$c1$	$c1$	00	00
08	92	$6b$	$d7$	00	00	00	00

Las imágenes con cuatro planos de bits, se comprimen de manera similar, pero con dos excepciones. La primera es, que un par (C, P) representa un contador de un byte —C—, y un byte de dos valores de píxel que se alternan —P—. El par $05_{16}\,a2_{16}$, por ejemplo, es la representación comprimida de los cinco píxeles de 4 bits: a, 2, a, 2 y a; mientras que el par $07_{16}\,ff_{16}$ representa siete píxeles f_{16} consecutivos de 4 bits. La segunda excepción, tiene que ver con los pares $(0, C)$, donde C es mayor que 2. Tales pares deben ir seguidos por C píxeles de 4 bits empaquetados —dos por byte—. El valor de C, es normalmente un múltiplo de 4, en cuyo caso, un par $(0, C)$ especifica los píxeles necesarios para llenar un número entero de palabras (donde una palabra es de 2 bytes). Si C no es un múltiplo de 4, el resto de la última palabra se rellena con ceros. Así, $00_{16}\,08_{16}\,a35b1247_{16}$, especifica 8 píxeles y ocupa 4 bytes (o dos palabras) —$a35b1247_{16}$—; mientras que $00_{16}\,06_{16}\,a35b12_{16}$, especifica seis píxeles y también ocupa 4 bytes —$a35b1200_{16}$—.

1.5. Codificación mover–al–frente

La idea básica de este método [Bentley 86] es tratar el alfabeto de símbolos —A— como una lista, donde los elementos que aparecen con más frecuencia están ubicados cerca del principio. Un símbolo

[17] Datos en bruto, sin comprimir.

a	abcdmnop	0	a	abcdmnop	0	a	abcdmnop	0	a	abcdmnop	0
b	abcdmnop	1	b	abcdmnop	1	b	abcdmnop	1	b	abcdmnop	1
c	bacdmnop	2	c	abcdmnop	2	c	bacdmnop	2	c	abcdmnop	2
d	cbadmnop	3	d	abcdmnop	3	d	cbadmnop	3	d	abcdmnop	3
d	dcbamnop	0	d	abcdmnop	3	m	dcbamnop	4	m	abcdmnop	4
c	dcbamnop	1	c	abcdmnop	2	n	mdcbanop	5	n	abcdmnop	5
b	cdbamnop	2	b	abcdmnop	1	o	nmdcbaop	6	o	abcdmnop	6
a	bcdamnop	3	a	abcdmnop	0	p	onmdcbap	7	p	abcdmnop	7
m	abcdmnop	4	m	abcdmnop	4	a	ponmdcba	7	a	abcdmnop	0
n	mabcdnop	5	n	abcdmnop	5	b	aponmdcb	7	b	abcdmnop	1
o	nmabcdop	6	o	abcdmnop	6	c	baponmdc	7	c	abcdmnop	2
p	onmabcdp	7	p	abcdmnop	7	d	cbaponmd	7	d	abcdmnop	3
p	ponmabcd	0	p	abcdmnop	7	m	dcbaponm	7	m	abcdmnop	4
o	ponmabcd	1	o	abcdmnop	6	n	mdcbapon	7	n	abcdmnop	5
n	opnmabcd	2	n	abcdmnop	5	o	nmdcbapo	7	o	abcdmnop	6
m	nopmabcd	3	m	abcdmnop	4	p	onmdcbap	7	p	abcdmnop	7
	mnopabcd						ponmdcba				
	(a)			(b)			(c)			(d)	

Tabla 1.14: Codificación con y sin *mover–al–frente*.

`s` se codifica como el número de símbolos que lo preceden en dicha lista. Así, si $A = (\texttt{t}, \texttt{h}, \texttt{e}, \texttt{s}, \ldots)$, y el siguiente símbolo a tratar en la secuencia de entrada es `e`, éste se codifica como 2, ya que está precedido por dos símbolos. Hay varias variantes posibles de este método; la más básica, añade un paso más: Después de haber codificado el símbolo `s`, éste se mueve al principio de la lista A. De este modo, después de la codificación de `e`, el alfabeto se convierte en $A = (\texttt{e}, \texttt{t}, \texttt{h}, \texttt{s}, \ldots)$. Este desplazamiento, antepone el último elemento encontrado al resto de la lista, con la esperanza de que aparezca muchas más veces en la secuencia de entrada, convirtiéndose —al menos por un tiempo— en un símbolo habitual. El método *mover–al–frente* es *localmente adaptativo*, ya que se adapta a las frecuencias de los símbolos en áreas locales de los datos de entrada.

El método da buenos resultados si la secuencia de entrada cumple esta expectativa, i.e., si contiene concentraciones de símbolos idénticos (si la frecuencia local de los símbolos cambia significativamente de región a región en los datos de entrada). Esto recibe el nombre de *propiedad de concentración*. A continuación, exponemos dos ejemplos que ilustran la idea de *mover–al–frente*. Ambos suponen que el alfabeto es $A = (\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \texttt{m}, \texttt{n}, \texttt{o}, \texttt{p})$.

1. La cadena de entrada `abcddcbamnopponm` se codifica como:

   $$C = (0, 1, 2, 3, 0, 1, 2, 3, 4, 5, 6, 7, 0, 1, 2, 3)$$

   (Tabla 1.14a —pág. 40—). Sin el paso *mover–al–frente* se codifica como:

   $$C' = (0, 1, 2, 3, 3, 2, 1, 0, 4, 5, 6, 7, 7, 6, 5, 4)$$

   (Tabla 1.14b —pág. 40—). Ambas —C y C'— contienen códigos del mismo intervalo —$[0, 7]$—, pero los elementos de C son —en promedio— más pequeños, ya que la entrada comienza con una concentración de `abcd` y continúa con una concentración de `mnop`. (El valor medio de C es 2,5, mientras que el de C' es 3,5.)

2. La cadena de entrada `abcdmnopabcdmnop` se codifica como:

   $$C = (0, 1, 2, 3, 4, 5, 6, 7, 7, 7, 7, 7, 7, 7, 7, 7)$$

i	Código	Tamaño
1	1	1
2	010	3
3	011	3
4	00100	5
5	00101	5
6	00110	5
7	00111	5
8	0001000	7
9	0001001	7
⋮	⋮	⋮
15	0001111	7
16	000010000	9

Tabla 1.15: Ejemplos de códigos de tamaño variable.

(Tabla 1.14c —pág. 40—). Sin el paso *mover–al–frente* se codifica como:

$$C' = (0, 1, 2, 3, 4, 5, 6, 7, 0, 1, 2, 3, 4, 5, 6, 7)$$

(Tabla 1.14d —pág. 40—). El promedio de C es ahora de 5,25, mayor que el de C' —que es 3,5—. La regla *mover–al–frente* crea un resultado peor en este caso, ya que la entrada no contiene concentraciones de símbolos idénticos (no satisfacen la propiedad de concentración).

Antes de entrar en más detalles, es importante comprender la ventaja de tener números pequeños en C. Esta característica hace posible codificar eficientemente C, ya sea con el algoritmo de Huffman, ya sea con la codificación aritmética (Capítulo 2). Veamos cuatro maneras para llevar a cabo esta mejora:

1. Asignar códigos de Huffman a los enteros en el rango $[0, n]$ de forma que los más pequeños obtengan los códigos más cortos. Un ejemplo para los enteros de 0 a 7 es:

Número	0	1	2	3	4	5	6	7
Código	0	10	110	1110	11110	111110	1111110	1111111

2. Asignar códigos a los enteros de modo que el código del entero $i \geqslant 1$ es su código binario precedido por $[log_2 i]$ (El corchete denota "parte entera de"). La Tabla 1.15 (pág. 41) muestra algunos ejemplos.

   ◇ **Ejercicio 1.10 (sol. en pág. 1053):** ¿Cuál es el tamaño total del código i en este caso?

3. Utilizar el código de Huffman adaptativo (Sección 2.9).

4. Realizar dos pasadas sobre C —para obtener la compresión máxima—; en la primera se cuentan las frecuencias de los códigos, y en la segunda se realiza la codificación real. Las frecuencias obtenidas en la pasada 1, se usan para calcular las probabilidades y asignar los códigos de Huffman que serán utilizados posteriormente en la pasada 2.

Se puede demostrar que el método *mover–al–frente* —en el peor caso— se comporta ligeramente peor que la codificación Huffman. En el mejor caso, su rendimiento es significativamente mejor.

Como se ha mencionado anteriormente, es fácil elaborar variaciones sobre la idea base de la codificación *mover–al–frente*. Estas son algunas de ellas:

1. *Avanzar-k*. El elemento de A que coincide con el símbolo actual se desplaza k posiciones hacia el frente de A, sin llegar necesariamente a convertirse en el primer elemento de A. El parámetro k puede ser especificado por el usuario, fijando un valor por defecto de n ó 1. Esto tiende a reducir el rendimiento —i.e., a incrementar el tamaño medio de los elementos de C— con las entradas que satisfagan la propiedad de concentración, pero funciona mejor con otras secuencias de datos. Téngase en cuenta que la asignación $k = n$ es idéntica a *mover–al–frente*. El caso $k = 1$ es especialmente simple, ya que sólo requiere el intercambio de un elemento de A con el primero que le precede.

   ◇ **Ejercicio 1.11 (sol. en pág. 1053):** Utilice el método *avanzar-k* para codificar cada una de las cadenas —`abcddcbamnopponm` y `abcdmnopabcdmnop`— dos veces, con $k = 1$ y $k = 2$.

2. *Esperar-c-y-mover*. Un elemento de A se mueve al principio sólo después de haberse encontrado c veces en los símbolos de la secuencia de entrada (no necesariamente c veces consecutivas). Cada elemento de A debe tener un contador asociado con él, para determinar el número de veces que ha sido localizado. Este método tiene sentido en implementaciones donde el movimiento y la reorganización de los elementos de A son lentos.

3. Normalmente, el símbolo que se lee de la entrada es un byte. Si los datos de entrada están compuestos por texto, puede tener sentido tratar cada palabra —en lugar de cada carácter— como un símbolo. Consideremos el simple caso, en el que la entrada consiste sólo en letras minúsculas, espacios y un marcador fin-de-texto al final. Podemos definir una palabra como una cadena de letras, seguidas por un espacio o por el marcador fin-de-texto. El número de palabras —en este caso— puede ser enorme, por lo que el alfabeto A debe comenzar vacío y las palabras deben añadirse según vayan apareciendo en la entrada y sean codificadas. Utilizamos el siguiente texto como ejemplo:

   `the␣boy␣on␣my␣right␣is␣the␣right␣boy`

   La primera palabra que se va a procesar es `the`. No se encuentra en A —ya que A está vacío—, por lo que entra a formar parte de A. El codificador emite un 0 —el número de palabras que preceden a `the` en A—, seguido por `the`. El decodificador también comienza con A vacío. El 0 le indica que debe seleccionar la primera palabra de A; pero puesto que A está vacío, supone que tras el 0 va a encontrar una palabra que añade a la lista A.

   La siguiente palabra es `boy`. Se añade al final de A —ahora, $A = (\texttt{the}, \texttt{boy})$— y el codificador emite: 1`boy`. La palabra `boy` se traslada al frente de A, por consiguiente, $A = (\texttt{boy}, \texttt{the})$. El decodificador lee el 1 —que se refiere a la segunda palabra de A—, pero el alfabeto A del decodificador sólo tiene una palabra. Por eso, el decodificador deduce que tras el 1 debe existir una palabra nueva; la lee y la coloca al principio de A. La Tabla 1.16 (pág. 43) resume los pasos de la codificación para este ejemplo.

   Una lista puede llegar a ser muy extensa en esta variante, pero cualquier implementación práctica debe limitar su tamaño. Es por esto por lo que el último elemento de A —el menos utilizado recientemente— tiene que ser eliminado cuando A exceda su tamaño límite. Esta es otra diferencia entre esta variante y el método básico de *mover–al–frente*.

   ◇ **Ejercicio 1.12 (sol. en pág. 1053):** Decodifíquese el texto:

   `the␣boy␣on␣my␣right␣is␣the␣right␣boy`

   y resúmanse los pasos en una tabla.

Palabra	(Antes de añadir) A	(Después de añadir) A	Código emitido
the	()	(the)	0the
boy	(the)	(the, boy)	1boy
on	(boy, the)	(boy, the, on)	2on
my	(on, boy, the)	(on, boy, the, my)	3my
right	(my, on, boy, the)	(my, on, boy, the, right)	4right
is	(right, my, on, boy, the)	(right, my, on, boy, the, is)	5is
the	(is, right, my, on, boy, the)	(is, right, my, on, boy, the)	5
right	(the, is, right, my, on, boy)	(the, is, right, my, on, boy)	2
boy	(right, the, is, my, on, boy)	(right, the, is, my, on, boy)	5
	(boy, right, the, is, my, on)		

Tabla 1.16: Codificación de palabras de múltiples letras.

"Voy a sacar mi dinero ahora, Sadi," dijo Issus.
"Tan pronto como estemos seguros de que este es el niño correcto", respondió Sadi.
"Pregúntale cuál es su nombre," dijo un susurro sibilante desde la oscuridad, detrás de Garion.
"Lo haré, Maas." Sadi parecía ligeramente molesto por la sugerencia. "He hecho esto antes".
"Se está tomando demasiado tiempo", dijo la voz sibilante.
"Diga su nombre, muchacho," dijo Sadi Garion.
"Doroon", Garion mintió rápidamente. "Estoy realmente muy sediento."

—David Eddings, *The Belgariad, Queen of Sorcery*[a].

[a] Crónicas de Belgarah, Reina de la Hechicería.

1.6. Cuantificación escalar

La definición del diccionario para el término "cuantificación" es "restringir una cantidad variable a valores discretos, en lugar de a una serie continua de valores." En el campo de la compresión de datos, la cuantificación se utiliza de dos maneras:

1. Si los datos a comprimir son números grandes, se usa la cuantificación para convertirlos en números pequeños. Los números pequeños ocupan menos espacio que los grandes, por lo que cuantificación genera compresión. Por otra parte, generalmente contienen menos información que los grandes; esto hace que la cuantificación produzca como resultado una compresión con pérdida.

2. Si los datos a comprimir son analógicos (i.e., un voltaje que cambia con el tiempo), la cuantificación se utiliza para digitalizarlos en números pequeños. Cuanto menor sea el número mejor será la compresión, pero también será mayor la pérdida de información. Este aspecto de la de cuantificación es utilizado por varios métodos de compresión del lenguaje.

> Yo no tendría la valentía de plantear esta posibilidad, si el académico Arkhangelsky no hubiera llegado tentativamente a la misma conclusión. Él y yo hemos estado en desacuerdo acerca de la cuantificación de los desplazamientos hacia el rojo de los quasar, la explicación de fuentes de luz superluminosas, la masa en reposo del neutrino, la física de quarks en estrellas de neutrones.... Hemos tenido muchas discrepancias.
>
> —Carl Sagan, *Contact*[a](1986)
>
> [a]Contacto.

En esta discusión suponemos que los datos que se van a comprimir están en forma de números y que llegan —número a número— de una secuencia de datos de entrada (o fuente). La Sección 4.14 describe una generalización de la cuantificación discreta para aquellos casos en que los datos consten de conjuntos —llamados vectores— de números en lugar de números individuales.

El primer ejemplo es una sencilla cuantificación discreta de una cadena de entrada de números de 8 bits. Simplemente, podemos eliminar los cuatro bits menos significativos de cada elemento de datos. Este es uno de los raros casos en el cual el factor de compresión (= 2) se conoce de antemano y no depende de los datos. Los datos de entrada constan de 256 símbolos diferentes, mientras que los de salida se componen de sólo 16 símbolos diferentes. Este método es simple pero no muy práctico ya que se pierde demasiada información con el fin de obtener el impresionante factor de compresión de 2.

Con el fin de conseguir una mejor aproximación, suponemos de nuevo que los datos se componen de números sin signo de 8 bits. De este modo, los símbolos de entrada están en el intervalo $[0, 255]$ (si los datos de entrada son con signo, los símbolos de entrada tienen valores en el rango $[-128, +127]$). Seleccionamos un parámetro de separación —s— y calculamos la secuencia uniforme de valores cuantificados 0, s, $2s$, ..., ks, de modo que $(k+1)\,s > 255$ y $ks \leq 255$. Cada símbolo de entrada S está cuantificado —convirtiéndolo en el valor más cercano contenido en esta secuencia—. Seleccionando $s = 3$, e.g., se produce la secuencia uniforme 0, 3, 6, 9, 12, ..., 252, 255. Seleccionando $s = 4$, se obtiene 0, 4, 8, 12, ..., 252, 255 (ya que el siguente múltiplo de 4 —después de 252— es 256).

Un enfoque similar consiste en seleccionar valores cuantificados de manera que cualquier número en el intervalo $[0, 255]$ no esté a más de d unidades de distancia de uno de los valores de los datos que se están cuantificando. Esto se consigue dividiendo el intervalo en segmentos de tamaño $2d + 1$ y centrándolos en el intervalo $[0, 255]$. Si, e.g., $d = 16$, entonces el intervalo $[0, 255]$ se divide en siete segmentos de tamaño 33 cada uno, con 25 números sobrantes. Así pues, podemos comenzar el primer segmento con los 12 números del principio del intervalo, lo cual produce la secuencia[18] de 9 números: 12, 45, 78, 111, 144, 177, 210, 243 y 255. Cualquier elemento en el intervalo $[0, 255]$ se encuentra a no más de 16 unidades de distancia de cualquiera de estos números. Si queremos limitar la secuencia cuantificada a sólo ocho números —para que cada uno pueda expresarse con 3 bits—, se puede aplicar el mismo método para calcular la nueva secuencia: 8, 41, 74, 107, 140, 173, 206 y 239.

Las secuencias de cuantificación anteriores tienen sentido cuando cada símbolo de los datos de entrada aparece con igual probabilidad (ocurre cuando la fuente *no tiene memoria*). Si los datos de entrada no están uniformemente distribuidos, la secuencia de valores cuantificados deberá ser distribuida de la misma manera que los datos.

Imagine, e.g., un flujo de entrada de datos de elementos sin signo de 8 bits, donde la mayoría son cero o cercanos a cero y pocos son grandes. Una buena secuencia de valores cuantificados de tales datos debería tener la misma distribución, i.e., muchos valores pequeños y pocos grandes. Una forma de calcular tal secuencia es seleccionar un valor para el parámetro longitud —l—, y construir una

[18]El primer segmento hasta 12 —la parte entera de la mitad de 25—, para centrar; para el resto, se va sumando 33, excepto el último, que también es de 12 elementos.

bbbbbbbb		bbbbbbbb	
1	1	.	
10	2	.	
11	3	.	
100	4	100|000	32
101	5	101|000	40
110	6	110|000	48
111	7	111|000	56
100|0	8	100|0000	64
101|0	10	101|0000	80
110|0	12	110|0000	96
111|0	14	111|0000	112
100|00	16	100|00000	128
101|00	20	101|00000	160
110|00	24	110|00000	192
111|00	28	111|00000	224

Tabla 1.17: Una tabla de cuantificación logarítmica.

"ventana" con el formato:

$$1\underbrace{b\ldots bb}_{l},$$

donde cada b es un bit que se coloca debajo de cada una de las posiciones de los 8 bits de un elemento de datos. Si la ventana sobresale por la derecha, se trunca alguno de los l bits. A medida que la ventana se desplaza hacia la izquierda, se van añadiendo ceros por la derecha de la misma. La Tabla 1.17 (pág. 45) ilustra esta construcción con $l = 2$. Es fácil ver cómo el resultado de los valores cuantificados comienza con una separación inicial de una unidad, aumentando sucesivamente a dos, cuatro unidades, ... hasta que el espaciado de los últimos cuatro valores es de 32 unidades. Los números 0 y 255 deberían añadirse manualmente a esta cuasi-secuencia logarítmica para hacerla más general.

La cuantificación escalar es un ejemplo de método de compresión con pérdida, donde es fácil controlar el equilibrio entre la razón de compresión y el volumen de datos perdidos. Sin embargo, debido a que es tan simple, su uso está limitado a los casos donde se puede tolerar una gran pérdida. Muchos métodos de compresión de imagen son con pérdidas, pero la cuantificación escalar no es adecuada para la compresión de imágenes, ya que crea molestas imperfecciones en la imagen descomprimida. Imagine una imagen con un área casi uniforme en la que todos los píxeles tienen valores 127 ó 128. Si 127 está cuantificado a 111 y 128 está cuantificado a 144, entonces el resultado —después de la descompresión— puede parecerse a un tablero de ajedrez donde los píxeles adyacentes se alternan entre 111 y 144. Ésta es la razón por la que los algoritmos prácticos usan la *cuantificación vectorial*, en lugar de la cuantificación escalar, para la compresión con pérdida —y a veces sin pérdidas— de imágenes y sonido. Véase también la Sección 4.1.

1.7. Reducción recursiva del rango[19]

En su documento original de 1977 [Ziv y Lempel 77], Lempel y Ziv demostraron que su método de diccionario puede comprimir los datos hasta llegar al valor de la entropía, pero señalaron que se necesitarían grandes cantidades de datos para aproximarse a la compresión ideal. Otros algoritmos, sobre todo PPM (Sección 2.18), sufren del mismo problema. A menudo, un usuario está dispuesto a

[19]Recursive Range Reduction.

sacrificar el rendimiento de la compresión en favor de una fácil implementación y el conocimiento de que el rendimiento del algoritmo es independiente del número de datos. El método *reducción recursiva del rango* (3R) aquí descrito, obtiene una compresión bastante buena, es fácil de programar y su rendimiento es independiente de la cantidad de datos a comprimir. Estas características lo convierten en un atractivo candidato para la compresión en sistemas embebidos, microcontroladores de bajo costo y otras aplicaciones donde el espacio es limitado o los recursos están restringidos. El método es la creación de Yann Guidon, quien lo describió en [3R 06]. Está relacionado remotamente con los códigos de tamaño variable.

El método se describe en primer lugar para una lista ordenada de números enteros, donde su aplicación es sencilla y no se necesita recursión. Denominamos a esta versión *reducción del rango* (RR). Luego mostramos cómo RR puede extenderse a una versión *recursiva* (RRR o 3R) que se puede aplicar a cualquier conjunto de números enteros.

Dada una lista de números enteros, eliminamos primero los efectos del bit de signo. O bien rotamos cada entero para mover el bit de signo a la posición menos significativa (un plegado) o añadimos un desplazamiento a todos los enteros, de manera que no sean negativos. La lista se ordena en orden decreciente. El primer elemento de la lista es el mayor y suponemos que su bit más significativo —*Most Significant Bit* ó MSB— es un 1 (i.e., que no tiene ceros adicionales a la izquierda).

Es evidente que los enteros que siguen al primer elemento pueden tener algunos de los bits más significativos a cero, y el corazón del método RR es eliminar la mayor parte de esos bits, mientras deja información suficiente para que el decodificador pueda restaurarlos. El primer elemento de la cadena a comprimir es un encabezado con la longitud L del mayor entero. Esta longitud se almacena en un campo de tamaño fijo, suficiente para cualquier dato que pueda ser encontrado. En los ejemplos considerados aquí, este tamaño es de cuatro bits, permitiendo enteros de hasta $2^4 = 16$ bits de longitud. Una vez que el decodificador recibe L, conoce la longitud del primer entero. El MSB de este entero es un 1, por lo que este bit puede ser eliminado y el primer entero puede ser emitido en $L - 1$ bits. El siguiente entero se escribe en la salida como un número de L bits, permitiendo así al decodificador leerlo sin ambigüedades. El decodificador controla entonces los bits más significativos del segundo entero. Si los k bits situados más a la izquierda son ceros, entonces el decodificador sabe que el próximo entero (i.e., el tercero) fue escrito sin sus k ceros de más a la izquierda, y por lo tanto, se encuentra en los $L - k$ bits siguientes de la cadena comprimida. Así es como RR comprime una lista ordenada de enteros no negativos. La eficiencia de la compresión depende de los datos, pero no de su cantidad. El método no parece mejorar cuando se aplica a grandes cantidades de datos.

La Tabla 1.18 es un ejemplo. Los datos se componen de diez enteros de 7 bits. El campo longitud de 4 bits se ha establecido en 6, lo que indica enteros de 7 bits (Tenga en cuenta que L no puede ser cero, por lo que 6 significa una longitud de siete bits). El primer entero se emite sin su bit de más a la izquierda y el siguiente sale completo. El tercer entero también se emite tal cual es, pero su cero de más a la izquierda —en negrita— indica al decodificador que el siguiente entero —el cuarto— tendrá sólo seis bits.

El tamaño total de los diez enteros es de 70 bits y esto debe compararse con los 54 bits creados por RR y los 64 bits resultantes del código Rice[20] (con $n = 4$) de los mismos enteros (véase la Sección 7.9 para estos códigos). La compresión no es impresionante, pero es evidente que no se ve afectada por la cantidad de datos.

La limitada experiencia disponible con RR parece indicar —aunque aún se necesita una demostración rigurosa— que este método funciona mejor con los datos que disminuyen de manera exponencial, donde cada entero es aproximadamente la mitad del tamaño de su predecesor. En una lista así, cada

[20] En un código Rice, los n bits menos significativos del dato son los n bits menos significativos del código, y los primeros bits del código son: el bit de signo —ignorado en el ejemplo—, seguido del valor de los restantes bits del dato —convertido a código unario—. Por ejemplo, considere el segundo dato de la Tabla 1.18 —100**1011** (n es 4)—; entonces, los últimos n bits del código son 1011, y los primeros son 00001 ($100_2 = 4_{10}$ —por eso son cuatro ceros seguidos por un 1—), produciendo el código <*bit de signo 0 ó 1*>00001 1011.

Datos	Código RR	Código Rice
	6=0110	
1011101	011101	000001 1101
1001011	1001011	00001 1011
0110001	**0**110001	0001 0001
0101100	101100	001 1100
0001110	**00**1110	1 1110
0001101	1101	1 1101
0001100	1100	1 1100
0001001	1001	1 1001
0000010	**00**10	1 0010
0000001	01	1 0001
70 bits	54 bits	64 bits

Tabla 1.18: Ejemplo de Reducción de Rango.

Datos	Código RR	Datos	Código RR
	6=0110		13=1101
1000010	000010	10011100010000	0011100010000
1000001	1000001	00001111101000	**0000**1111101000
1000000	1000000	00000001100100	**000**1100100
0111111	**0**111111	00000000001010	**000**1010
0111110	111110	00000000000001	0001
0111101	111101		
0111100	111100		
49 bits	49 bits	70 bits	52 bits

Tabla 1.19: El peor rendimiento de la Reducción de Rango.

número es un poco más corto que su predecesor. Peores resultados se obtienen con los datos que, o bien disminuyen lentamente —como (900, 899, 898, 897, 896)—, o que decrecen rápidamente —como (100000, 1000, 10, 1)—. Estos casos se ilustran en la Tabla 1.19. Tenga en cuenta que el encabezado es la única información adicional que se envía al decodificador y que RR nunca expande los datos.

Ahora consideramos datos desordenados. Dada una lista desordenada de números enteros, podemos ordenarla, comprimirla con RR, y luego incluir información adicional sobre el orden; de modo que el decodificador pueda restablecer el desorden de los datos correctamente, después de descomprimirlos. Se necesita un método eficaz para especificar la relación entre una lista de números y su versión ordenada; el método aquí descrito se conoce como reducción recursiva del rango ó 3R.

El algoritmo 3R implica la creación de un árbol binario donde cada camino desde la raíz hasta una hoja es una lista decreciente de enteros. Cada ruta está codificada con 3R por separado, lo cual requiere un algoritmo recursivo (de ahí la palabra "recursivo" en el nombre del algoritmo). La Figura 1.20 muestra un ejemplo. Dados cinco enteros desordenados, de A a E, se construye un árbol binario de manera que cada par de enteros consecutivos se convierte en un subárbol en el nivel inferior inmediato. Cada uno de los cinco caminos desde la raíz a una hoja es una lista ordenada, y está comprimida usando 3R. Es evidente que la escritura de los resultados comprimidos de las cinco listas en la salida normalmente genera expansión, por lo que realmente sólo se escriben ciertos nodos. La regla es seguir cada camino desde la raíz y seleccionar los nodos a lo largo de los bordes que van hacia la izquierda (o sólo los que van a la derecha). Así, en nuestro ejemplo, escribimos los siguientes valores en la salida: (1) un encabezado con la longitud de la raíz, (2) la ruta codificada en 3R (raíz, $A + B$,

A,), (3) el valor del nodo C —después se codifica la ruta (raíz, $C + D + E$, C)— y (4) el valor del nodo D —después se codifica la ruta (raíz, $C + D + E$, $D + E$, D)—.

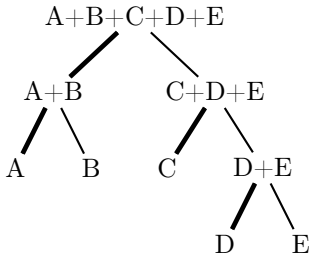


Figura 1.20: Un árbol binario codificado con 3R.

El decodificador lee el encabezado y el primer camino (raíz, $A + B$, A). Decodifica dicha ruta para obtener los tres valores, y resta $(A + B) - A$ para deducir B. A continuación, realiza la operación $Raíz - (A + B)$ para obtener $C + D + E$. Lee C; ahora puede decodificar el camino (raíz, $C + D + E$, C). A continuación, el decodificador resta C de $C + D + E$ para obtener $D + E$, lee D y decodifica la ruta (raíz, $C + D + E$, $D + E$, D). Con una resta más, deduce E.

Esto suena como a mucho trabajo para una compresión tan pobre, ¡y lo es! El método 3R es no el algoritmo de compresión más eficiente, pero puede encontrar sus aplicaciones. La característica más atractiva de la estructura de árbol aquí descrita, es que el número de nodos escritos en la salida es igual al tamaño de los datos originales. En nuestro ejemplo, hay cinco elementos de datos que producen cinco códigos 3R en la salida. Es fácil ver por qué esto es cierto en general. Dada un lista de n elementos de datos, observamos que van a terminar siendo las hojas del árbol binario. Lo que eventualmente se escribe en la salida, es la mitad de los nodos de cada nivel del árbol. Un árbol binario completo con n hojas tiene $2n$ nodos, de modo que la mitad de este número —n nodos— acaba formando parte de la salida.

Figura 1.21: Histograma de la imagen de Lena.

Parece que 3R podría ser adecuado para aplicaciones en las que los datos no se ajustan a ninguna

de las distribuciones estadísticas estándar y sin embargo no son aleatorios. Un ejemplo de tales datos es el histograma de una imagen —La Figura 1.21, muestra los histogramas CMYK[21] de la imagen de Lena, Figura 4.53 (pág. 339)—. Por otra parte, 3R se limita a números enteros no negativos y el codificador requiere dos pasos: uno para la construcción del árbol, y otro para recorrerlo y recoger los nodos.

Ya se ha mencionado que RR nunca expande los datos. Sin embargo, cuando se le administran datos aleatorios, 3R genera una salida que aumenta aproximadamente un bit por cada elemento de datos. Debido a esta característica, su desarrollador se refiere a 3R como un codificador cercano a la entropía, no como a un codificador de entropía real.

Explicación. Debido a las sumas, los datos aleatorios se "reparten proporcionalmente", por lo que después de unos pocos niveles de árbol, son como la mayor parte de los datos monótonos. Después, estos datos se suman de nuevo juntos, añadiendo aún un bit por nivel, pero reduciendo a la mitad el número de sumas cada vez, de manera que el final es equivalente a un bit por cada elemento de la entrada.

—Yann Guidon (comunicación privada)

Los algoritmos de compresión son a menudo descritos como "squeezing, squashing, crunching or imploding data"[22], pero
éstas no son muy buenas descripciones de lo que está
sucediendo realmente.

—James D. Murray y William Vanryper (1994)

[21] Cyan, Magenta, Yellow, Black (Cian, Magenta, Amarillo, Negro, en castellano).
[22] Algoritmo para apretar, aplastar, hacer crujir o implosionar datos.

Capítulo 2

Métodos estadísticos

Los métodos expuestos hasta ahora tienen una característica común: asignan códigos de tamaño fijo a los símbolos —caracteres o píxeles— con los que operan. En contraste, los métodos estadísticos usan códigos de tamaño variable, asignando los más cortos a los símbolos o grupos de símbolos que aparecen con más frecuencia en los datos —los que tienen una mayor probabilidad de ocurrencia—. Los diseñadores y los implementadores de códigos de tamaño variable tienen que tratar los dos problemas: (1) de asignación de códigos que puedan ser decodificados sin ambiguedad y (2) de asignación de códigos con el tamaño medio mínimo.

Samuel Morse utilizó códigos de tamaño variable cuando diseñó su famoso código para el telégrafo (Tabla 2.1). Es interesante observar que la primera versión de su código, desarrollado por Morse durante un viaje transatlántico en 1832, era más compleja que la versión que él estableció en en 1843. En la primera versión se enviaban guiones cortos y largos que eran recibidos y dibujados en una tira de papel, donde las secuencias de los guiones representaban los números. A cada palabra —no a cada letra— se le asignó un código numérico y Morse construyó un libro —o diccionario— de dichos códigos en 1837. Esta primera versión fue, por lo tanto, una forma primitiva de compresión. Posteriormente, Morse abandonó esta versión en favor de la más famosa —de puntos y rayas—, desarrollada junto con Alfred Vail.

> Morse estableció la primera línea de larga distancia —entre Washington y Baltimore— que se inauguró el 24 de mayo de 1844, con un mensaje seleccionado por la Srta. Annie Ellsworth, la hija del comisionado de patentes —la última frase del vigésimo tercer verso del vigésimo tercer capítulo del cuarto libro de la Biblia Hebrea, the Book of Numbers[a]: "What hath God wrought[b]!"
>
> —George B. Dyson, *Darwin Among the Machines*[c] (1997)
>
> [a] El Libro de los Números.
> [b] ¡Qué ha forjado Dios! ó ¡Lo que Dios ha creado!
> [c] Darwin entre las máquinas.

La mayor parte de este capítulo está dedicado a los diferentes algoritmos estadísticos (Shannon-Fano, Huffman, codificación aritmética y otros). Sin embargo, empezamos con una breve presentación de conceptos importantes de la *teoría de la información*. Esto lleva a una definición de redundancia, para que después podamos ver claramente y calcular cómo la redundancia puede reducirse o eliminarse, usando los distintos métodos.

A	.-	N	-.	1	.----	Period [.]	.-.-.-
B	-...	O	---	2	..---	Coma [,]	--..--
C	-.-.	P	.--.	3	...--	Colon [:]	---...
Ch	----	Q	--.-	4	....-	Question mark [?]	..--..
D	-..	R	.-.	5	.....	Apostrophe [']	.----.
E	.	S	...	6	-....	Hyphen [-]	-....-
F	..-.	T	-	7	--...	Dash [–,–,—]	-..-.
G	--.	U	..-	8	---..	Parentheses [( )]	-.--.-
H	....	V	...-	9	----.	Quotation marks ["]	.-..-.
I	..	W	.--	0	-----		
J	.---	X	-..-				
K	-.-	Y	-.--				
L	.-..	Z	--..				
M	--						

Si se toma como unidad la duración de un punto , entonces un guión consta de tres unidades. El espacio entre los puntos y las rayas de un carácter, una unidad; entre caracteres, es de tres unidades; y entre las palabras, seis unidades (cinco para la transmisión automática). Para indicar que se ha producido un error y para que el receptor elimine la última palabra, envíense "........" (ocho puntos).

Tabla 2.1: El código Morse para el idioma inglés.

2.1. Conceptos de teoría de la información

Intuitivamente sabemos qué es la información. Estamos constantemente recibiendo y enviando información en forma de texto, sonido e imágenes. También consideramos que la información tiene una esquiva cantidad —no matemática— que no puede ser definida con precisión, capturada o medida. Las definiciones más comunes que el diccionario da de la información son: (1) el conocimiento derivado del estudio, experiencia, o la instrucción; (2) el conocimiento de un evento o situación específica; inteligencia; (3) una colección de hechos o datos; (4) el acto de informar o la condición de ser informado; comunicación de conocimiento.

Imagínese que una persona que no sabe qué es la información. ¿Podrían esas definiciones aclarárselo? Es poco probable.

La importancia de la teoría de la información consiste en que ésta cuantifica la información. Muestra cómo medir la información, de manera que podamos responder a la pregunta: "¿cuánta información forma parte de esta muestra de datos?" ¡Con un número preciso! La cuantificación de la información se basa en la observación de que el contenido de información de un mensaje es equivalente a la cantidad de *elementos sorprendentes* en el mensaje. Si le digo algo que ya sabe —por ejemplo, "usted y yo trabajamos aquí"—, no le he dado ninguna información. Si le digo algo nuevo —por ejemplo, "que ambos recibimos un aumento"—, le he dado alguna información. Si le digo algo que realmente le sorprenda —por ejemplo, "sólo yo he recibido un aumento"—, le he dado más información, independientemente del número de palabras que he utilizado y de cómo se siente acerca de mi información.

Empecemos con una prueba sencilla y familiar que es fácil de analizar, a saber, el lanzamiento de una moneda. Hay dos valores posibles, por lo que el resultado de cualquier lanzamiento es inicialmente incierto. En realidad, tenemos que lanzar la moneda con el fin de resolver la incertidumbre. El resultado es cara o cruz, que también se puede expresar como un sí o un no, o como un 0 ó un 1; un bit.

Un solo bit resuelve la incertidumbre en el lanzamiento de una moneda. Lo que hace este ejemplo importante es el hecho de que es fácilmente generalizable. Muchos problemas de la vida real pueden ser resueltos y sus soluciones expresadas, mediante varios bits. Por lo tanto, el principio a realizar

es encontrar el número mínimo de preguntas *sí/no* que deben ser consideradas con el fin de llegar al resultado. Dado que la respuesta a una pregunta de este tipo se puede expresar con un bit, el número de preguntas será igual al número de bits que se necesitan para expresar la información contenida en el resultado.

Un ejemplo algo más complejo es una baraja de 64 naipes. Por simplicidad vamos a ignorar sus nombres y números tradicionales y simplemente los numeramos de 1 a 64. Considerese el caso de una persona **A** que roba una carta, y otra **B** que tenga que adivinar cuál es. Debe adivinar un número entre 1 y 64. ¿Cuál es el número mínimo de preguntas *sí/no* necesario para adivinar la carta? Aquellos que están familiarizados con la *búsqueda binaria* conocen la respuesta. Usando esta técnica, **B** tiene que dividir el intervalo 1–64 en dos y debe comenzar por preguntar: "¿está el resultado entre 1 y 32?" Si la respuesta es no, entonces el resultado se encuentra entre 33 y 64. Este intervalo se divide por dos y la siguiente pregunta de **B** debería ser: "¿está el resultado entre 33 y 48?" Este proceso continúa hasta que el intervalo seleccionado por **B** se reduce a solamente un número.

No hace falta indagar mucho para ver que las preguntas necesarias para llegar al resultado son exactamente seis. Esto se debe a que 6 es el número de veces que 64 puede ser dividido por la mitad. Matemáticamente, esto es equivalente a la ecuación $6 = \log_2 64$. Por ello, el **logaritmo** es la función matemática que cuantifica la información.

Otro enfoque del mismo problema es hacer la pregunta: Dado un entero no negativo N, ¿cuántos dígitos se necesitan para expresarlo? La respuesta —por supuesto— depende de N. A mayor N, más dígitos se necesitan. Los primeros 100 números enteros no negativos (0 a 99) pueden expresarse con dos dígitos decimales. Los primeros 1000 números enteros, pueden expresarse con tres dígitos. Una vez más, no nos lleva mucho tiempo ver la conexión. El número de dígitos necesarios para representar N, equivale aproximadamente a $\log N$. La base del logaritmo es la igual a la base de los dígitos. Para los números decimales, se utiliza la base 10, para los binarios (bits), se usa la base 2. Si estamos de acuerdo en que el número de dígitos que se necesitan para expresar N es proporcional al contenido de la información de N, entonces —de nuevo— el logaritmo es la función que nos da una medida de la información.

◇ **Ejercicio 2.1 (sol. en pág. 1053):** ¿Cuál es el tamaño preciso (en bits) del entero binario i?

Este es otro enfoque para la cuantificación de la información: Estamos familiarizados con los diez dígitos decimales. Sabemos que el valor de un dígito en un número depende de su posición. Así, el valor del dígito 4 en el número 14 708 es de 4×10^3 ó 4 000, ya que está en la posición 3 (las posiciones se numeran de derecha a izquierda, comenzando por 0). También estamos familiarizados con los dos dígitos binarios —bits— 0 y 1. Del mismo modo, el valor de un bit en un número binario depende de su posición —excepto que se utilizan potencias de 2—. Matemáticamente, no hay ninguna razón especial para escoger 2 ó 10. Podemos usar el número 3 como base de nuestra aritmética, en cuyo caso requerimos tres dígitos: 0, 1 y 2 —que llamaremos *trits*—. Un *trit* t en la posición i tiene un valor de $t \times 3^i$.

◇ **Ejercicio 2.2 (sol. en pág. 1053):** En realidad, hay algo especial en el 10. Nosotros usamos los números de base 10 porque tenemos diez dedos. También hay algo especial sobre el uso del 2 como base para un sistema de numeración. ¿Qué es?

Dado un número decimal —base 10— o uno ternario —base 3— de k dígitos, una pregunta natural es: ¿cuánta información se incluye en este número de k dígitos? Respondemos a esto determinando el número de bits que se necesitan para expresar el número dado. Suponiendo que la respuesta es x, entonces $10^k - 1 = 2^x - 1$. Esto se debe a que $10^k - 1$ es el número decimal más grande de k dígitos y $2^x - 1$ es el número binario más grande de x bits. La resolución de la ecuación con x como incógnita,

es fácil usando logaritmos y sus propiedades:

$$10^k = 2^x;\ \log 10^k = \log 2^x;\ k \cdot \log 10 = x \cdot \log 2;\ x = k \cdot \frac{\log 10}{\log 2}.$$

Podemos utilizar cualquier base para los logaritmos, siempre y cuando utilicemos la misma para $\log 10$ y $\log 2$. La selección de la base 2 simplifica el resultado, que se convierte en $x = k \cdot \log_2 10 \approx 3{,}32 \cdot k$. Esto muestra que la información incluida en un dígito decimal es igual a la contenida en 3,32 bits. En general, dados números en base n, podemos escribir $x = k \cdot \log_2 n$, que expresa el hecho de que la información incluida en un número en base n es igual a la contenida en $\log_2 n$ bits.

◇ **Ejercicio 2.3 (sol. en pág. 1053):** ¿Cuántos bits se necesitan para expresar la información incluida en un trit?

Ahora es el turno del transmisor: una pieza de hardware que puede transmitir datos a través de una línea de comunicación —un canal—. En la práctica, tal transmisor envía datos binarios (un módem es un buen ejemplo). Sin embargo, con el fin de obtener resultados generales, suponemos que los datos son una cadena formada por ocurrencias de los n símbolos $a_1, \ldots, a_n$. Este conjunto constituye un alfabeto de n símbolos. Dado que hay n símbolos, podemos considerar que cada uno es un dígito en base n, lo que significa que equivale a $\log_2 n$ bits. En cuanto al hardware se refiere, esto quiere decir que debe ser capaz de transmitir n niveles discretos.

Si el transmisor gasta $1/s$ unidades de tiempo para transmitir un solo símbolo, entonces la velocidad de la transmisión es de s símbolos por unidad de tiempo. Un ejemplo común es $s = 28\,800$ baud (*baud*[1] es el término para "bits por segundo"), que se traduce en $1/s \approx 34{,}7$ μsec (donde la letra griega μ representa "micro" y $1\mu\text{seg} = 10^{-6}\text{seg}$). En una unidad de tiempo, el transmisor puede enviar s símbolos, que equivale a un contenido de información de $s \cdot \log_2 n$ bits. Denotamos $H = s \cdot \log_2 n$ a la cantidad de información —medida en bits— transmitida en cada unidad de tiempo.

El siguiente paso consiste en expresar H en términos de las probabilidades de aparición de los n símbolos. Damos por sentado que el símbolo a_i aparece en los datos con una probabilidad P_i. La suma de las probabilidades es igual, por supuesto, a la unidad: $P_1 + P_2 + \cdots + P_n = 1$. Cuando las n probabilidades son iguales —$P_i = P$— obtenemos $1 = \sum P_i = nP$, lo que implica que $P = 1/n$, y por lo tanto, $H = s \cdot \log_2 n = s \cdot \log_2 (1/P) = -s \cdot \log_2 P$. En general, las probabilidades son diferentes, y queremos expresar H en términos de todos ellos. Ya que el símbolo a_i acapara una fracción P_i de tiempo en los datos, aparece en promedio $s \cdot P_i$ veces cada unidad de tiempo, por lo que su contribución a H es: $-s \cdot P_i \cdot \log_2 P_i$. La *suma* de las contribuciones de los n símbolos a H es, por lo tanto, $H = -s \cdot \sum_1^n P_i \cdot \log_2 P_i$.

Como recordatorio, H es la cantidad de información —en bits— enviada por el transmisor en una unidad de tiempo. La cantidad de información contenida en un símbolo en base n es, por lo tanto, H/s (porque para transmitir un símbolo se necesita un tiempo $1/s$), ó $-\sum_1^n P_i \cdot \log_2 P_i$. Esta cantidad se llama *entropía* de los datos que se están transmitiendo. Por analogía, podemos definir la entropía de un solo símbolo a_i como $-P_i \cdot \log_2 P_i$. Éste es el menor número de bits necesarios —en promedio— para representar el símbolo.

(La teoría de la información se desarrolló a finales de 1940, por Claude Shannon, de Bell Labs y eligió el término *entropía* porque este término se utiliza en termodinámica para indicar la cantidad de desorden en un sistema físico.)

[1] *Baudios.*

> Ya que pienso que es mejor tomar los nombres de este tipo de cantidades —que son importantes para la ciencia— de las lenguas antiguas, de modo que puedan introducirse sin cambios en todas las lenguas modernas, propongo nombrar a la magnitud S, *entropía* del cuerpo, derivada de la palabra griega "trope", que indica "transformación". He formado con toda la intención la palabra "entropía" con el fin de que sea lo más similar posible a la palabra "energía", ya que estas dos cantidades, que han de ser conocidas por esos nombres, están tan relacionadas entre sí en su significado físico, que una cierta similitud en sus nombres me parecía ventajosa.
>
> —Rudolph Clausius, 1865 (traducido al inglés por Hans C. von Baeyer)

La entropía de los datos depende de las probabilidades individuales P_i y la mayor se produce —véase el Ejercicio 2.4— cuando las n probabilidades son iguales. Este hecho se utiliza para determinar la *redundancia* —R— en los datos. Ésta se define como la diferencia entre la mayor entropía posible de un conjunto de símbolos y su entropía actual. En consecuencia[2]:

$$R = \left[-\sum_{1}^{n} P \cdot \log_2 P \right] - \left[-\sum_{1}^{n} P_i \cdot \log_2 P_i \right] = \log_2 n + \sum_{1}^{n} P_i \cdot \log_2 P_i .$$

La prueba para conseguir una compresión total de los datos es reducir la redundancia a cero, por lo tanto, debe ser $\log_2 n + \sum_1^n P_i \cdot \log_2 P_i = 0$.

◇ **Ejercicio 2.4 (sol. en pág. 1053):** Analícese la entropía de un conjunto de dos símbolos.

Dada una cadena de caracteres, la probabilidad de aparición de un carácter puede determinarse calculando su frecuencia —contando cuántos hay— y dividiéndola por la longitud de la cadena. Una vez que las probabilidades de todos los caracteres sean conocidos, se puede calcular la entropía de la cadena completa. Con el poderoso software matemático disponible en la actualidad, es fácil calcular la entropía de una cadena dada. El código en *Mathematica*[3]:

```
Frequencias[list_]:=Map[{Count[list,#],#}&, Union[list]];
Entropia[list_]:=-Plus @@ N[# Log[2,#]]& @
 (First[Transpose[Frequencias[list]]]/Length[list]);
Characters["swiss miss"]
Entropia[%]
```

hace esto y demuestra que la entropía de la cadena `swiss␣miss` es 1,96096. Expliquemos el ejemplo en detalle (␣ denota un espacio):

1. El código `Characters["swiss miss"]` de la penúltima línea produce un vector cuyos elementos son todos los caracteres de la cadena sin ordenar: `{s,w,i,s,s,␣,m,i,s,s}`.

2. El carácter `%` de la última línea se refiere al valor previamente calculado (el obtenido en el punto 1, en este caso). `Entropia[%]` llama a la función definida en las líneas 2 y 3.

3. En la primera línea se define una función que calcula la frecuencia de cada carácter de la cadena que se le ofrece. La función `Union` realiza una unión ordenada de los caracteres: `{␣,i,m,s,w}`; `Count[`a`,`b`]` cuenta el número de veces que aparece b en a; El carácter `#` representa el primer

[2] La máxima entropía se da cuando $P = \frac{1}{n}$ por lo que el factor $-\sum_1^n P \cdot \log_2 P = -n \cdot \left(\frac{1}{n}\right) \cdot \log_2 \left(\frac{1}{n}\right) = -\left(\log_2 1 - \log_2 n\right) = \log_2 n$.

[3] Lenguaje de programación de gran potencia para cálculos matemáticos, utilizado por el programa *Mathematica* y originalmente concebido por Stephen Wolfram.

argumento proporcionado a una función pura. En este caso, `{Count[list,#],#}&` , define una función que podrá ser usada las veces que queramos para generar pares $\{n, c\}$, donde n es el número de veces que aparece c en `list`; `Map` es la función que se encarga de pasar como parámetro cada elemento calculado por `Union`, a `Count`; proporciona como salida un vector cuyos elementos son pares, en los que el primer elemento del par, es el número de veces que aparece el segundo elemento del par en la cadena que recibió la función `Frequencias` al ser llamada; En el ejemplo, la salida es: `{{1,Null},{2,i},{1,m},{5,s},{1,w}}`.

4. `Transpose` transpone los primeros dos niveles del vector calculado en el punto 3, ofreciendo como salida otro vector cuyo primer elemento está compuesto por los primeros elementos de cada par, y el segundo, por los segundos de cada par: `{{1,2,1,5,1},{Null,i,m,s,w}}`; `First` obtiene el primer elemento de este par (que contiene las frecuencias de cada elemento de la cadena): `{1,2,1,5,1}`.

5. `Length` calcula la longitud de la cadena (10 en el ejemplo); en la tercera línea se divide cada frecuencia por este valor. En ese momento se tiene un vector de probabilidades, cuyos elementos son la frecuencia de aparición de cada elemento dividida por la longitud de la cadena tratada: $\left\{\frac{1}{10}, \frac{1}{5}, \frac{1}{10}, \frac{1}{2}, \frac{1}{10}\right\} = \{\mathtt{P}_1, \mathtt{P}_2, \mathtt{P}_3, \mathtt{P}_4, \mathtt{P}_5\}$.

6. `N[# Log[2,#]]&` define una función pura para obtener el producto $P \cdot \log_2 P$ donde P es cada uno de los valores de probabilidad de aparición de los distintos caracteres de la cadena; `@` indica que se va a aplicar la función a la expresión que le sigue, por lo que `#` será sustituído por cada elemento del vector de probabilidades del punto 5, obteniendo un vector en el que el elemento i es $P_i \cdot \log_2 P_i$.

7. `-Plus` suma los elementos del vector calculado en el punto 6 y cambia de signo al resultado (`f@@expr` reemplaza la cabeza de *expr* por f). El resultado de la operación es: $-\sum_1^5 P_i \cdot \log_2 P_i = 1{,}96096$.

You have two chances—
One of getting the germ
And one of not.
And if you get the germ
You have two chances—
One of getting the disease
And one of not.
And if you get the disease
You have two chances—
One of dying
And one of not.
And if you die—
Well, you still have two chances.

—Unknown[a]

[a] Usted tiene dos posibilidades— ; Una, contraer el germen; Y otra, no contraerlo. Y si se ha infectado; Tiene dos posibilidades— ; Una, contraer la enfermedad; Y otra, no contraerla. Y si enferma; Tiene dos posibilidades—; Una morir; Y otra, no hacerlo. Y si usted se muere—; Bueno, todavía tiene dos posibilidades.; —Desconocido.

El principal teorema demostrado por Shannon dice —en esencia—, que un mensaje de n símbolos puede —en promedio— comprirse hasta nH bits, pero no más. También dice que existen compresores

casi óptimos (llamados *codificadores de entropía*), pero no muestra cómo construirlos. La codificación aritmética (Sección 2.14) es un ejemplo de un codificador de entropía, así como también los algoritmos basados en diccionario del Capítulo 3 (pero estos últimos necesitan grandes cantidades de datos para llegar al nivel de entropía).

2.1.1. Contenido de la información algorítmica

Considere las siguientes tres secuencias:

$$S_1 = 1001001001001001001001001001001001001001001\ldots,$$

$$S_2 = 0101101101101011011010110110110110101101\ldots,$$

$$S_3 = 0111001001111001000010110000001110111 1\ldots.$$

La primera secuencia, S_1, es sólo una repetición del sencillo patrón 100. S_2 es menos regular. Puede ser descrito como un 01, seguido por r_1 repeticiones de 011, seguido por otro 01, seguido por r_2 repeticiones de 011, etc., donde $r_1 = 3$, $r_2 = 2$, $r_3 = 4$ —y los otros r_i no se muestran—. S_3 es más difícil de describir, ya que no parece tener ninguna regularidad aparente; parece aleatorio. Nótese que el significado de los puntos suspensivos es claro en el caso de S_1 (sólo tiene que repetir el patrón "100"), menos clara en S_2 (¿cuáles son los demás r_i?) y completamente desconocido en S_3 (¿es aleatorio?).

Ahora supongamos que estas secuencias son muy largas (digamos, 999 999 bits cada una) y cada una continúa "del mismo modo". ¿Cómo podemos definir la complejidad de tal secuencia binaria —al menos cualitativamente—? Una forma de hacerlo, llamada prueba de complejidad de Kolmogorov-Chaitin[4] (KCC), es definir la complejidad de una cadena binaria S como la longitud —en bits— del programa informático más corto, que cuando se ejecuta, genera S (visualizarlo, imprimirlo, o escribirlo en un fichero). Esta definición también se llama *contenido de la información algorítmica*[5] de la cadena S.

Un programa informático P_1 —para generar la cadena S_1— podría necesitar exactamente 333 333 bucles y obtener como resultado "100" en cada iteración. Alternativamente, el programa podría utilizar 111 111 bucles y generar "100100100" en cada iteración. Un programa de estas características es muy corto —sobre todo al compararlo con la longitud de la secuencia que genera—, coincidiendo con nuestra intuición de que S_1 tiene una complejidad baja.

Un programa P_2 —para generar la cadena S_2— tiene que conocer los valores de todos los r_i. Éstos podrían estar ya implementados o bien es el usuario quien los introduce en tiempo de ejecución. El programa inicializa a 1 una variable i. Luego imprime "01", realiza r_i bucles para mostrar "011" en cada iteración, incrementa i en 1, y repite este comportamiento hasta se hayan generado los 999 999 bits. Un programa de este tipo es más largo que P_1, lo que refleja nuestra intuición de que S_2 es más compleja que S_1.

Un programa P_3 —para generar la cadena S_3— debería (suponiendo que no podemos expresar esta cadena en una forma regular) simplemente imprimir los 999 999 bits de la secuencia. Este programa es tan largo como la secuencia en sí misma, lo que implica que la KCC de S_3 es tan grande como S_3.

Usando esta definición de complejidad, Gregory Chaitin mostró —véase [Chaitin 77] o [Chaitin 97]— que la mayoría de las cadenas binarias de longitud n son aleatorias; sus complejidades son cercanas a n. Sin embargo, las "interesantes" —o "prácticas"— cadenas binarias, las que se utilizan en la práctica para representar texto, imágenes y sonido, y se comprimen a todas horas, son similares a S_2. Estas no son aleatorias. Muestran cierta regularidad, lo que hace posible comprimirlas. Las cadenas muy regulares —como S_1— aparecen raramente en la práctica.

[4] Kolmogorov-Chaitin complexity.
[5] The algorithmic information content.

El contenido de la información algorítmica es una medida de la cantidad de información incluída en un mensaje. Se relaciona con la KCC y es diferente de la forma en que se mide la información en la teoría de la información. La teoría de la información de Shannon define la cantidad de información en una cadena, considerando la cantidad de *elementos sorprendentes* que esta información contiene cuando se revela. El contenido algorítmico de la información, por otra parte, mide la información que ya ha sido revelada. Un ejemplo puede servir para ilustrar esta diferencia. Imaginemos que dos personas —A (lectora, sofisticada y bien informada) y B (sin experiencia e ingenuo)—, leen la misma historia. Hay pocos elementos sorprendentes en la historia de A. Ya ha leído muchas historias similares y puede predecir la línea de desarrollo de la historia, el comportamiento de los personajes, y hasta el final. Lo contrario es cierto para B. A medida que lee, se sorprende por los giros y vueltas inesperadas que —para él— toma la historia, y por el comportamiento impredecible —para él— de los personajes. La pregunta es: ¿Cuánta información contiene realmente la historia?

La teoría de la información Shannon nos dice que la historia contiene menos información para A que para B, ya que contiene menos sorpresas para A que para B. Recordemos que la mente de A ya tiene recuerdos de intrigas y personajes similares. A medida que leen más y más, sin embargo, la historia se vuelve más y más familiar para ambos, y por lo tanto, cada vez menos sorprendente (aunque a ritmos diferentes). Por consiguiente, reciben cada vez menos información (del tipo de Shannon). Al mismo tiempo, ya que se revela más de la historia ante ellos, la complejidad de sus mentes aumenta (de nuevo a un ritmo diferente). Por lo tanto, obtienen de ella más información algorítmica. La suma de la información de Shannon y la KCC es por lo tanto constante (o casi constante).

Este ejemplo sugiere una manera de medir el contenido de la información de la historia de un modo absoluto, independientemente del lector en particular. Es la *suma* de la información de Shannon y la KCC. Esta medida ha sido propuesta por el físico Wojciech Zurek [Zurek 89], quien lo denominó "entropía física."

2.2. Códigos de tamaño variable

Consideremos los cuatro símbolos: a_1, a_2, a_3 y a_4 . Si aparecen en nuestras cadenas de datos con iguales probabilidades (= 0,25), entonces la entropía de los datos es: $-4 \cdot (0{,}25 \cdot \log_2 0{,}25) = 2$. En este caso, dos es el menor número de bits necesarios —en promedio— para representar cada símbolo. Podemos simplemente asignar a nuestros símbolos, los cuatro códigos de 2 bits: 00, 01, 10 y 11. Dado que las probabilidades son iguales, la redundancia es cero y los datos no se pueden comprimir a menos de 2 bits por símbolo.

A continuación, consideremos el caso en el que los cuatro símbolos aparecen con diferentes probabilidades (como se muestra en la Tabla 2.2), donde a_1 se encuentra en los datos —en promedio— cerca de la mitad de las veces, a_2 y a_3 tienen probabilidades iguales, y a_4 es raro. En este caso, la entropía de los datos es:

$$-(0{,}49 \cdot \log_2 0{,}49 + 0{,}25 \cdot \log_2 0{,}25 + 0{,}25 \cdot \log_2 0{,}25 + 0{,}01 \cdot \log_2 0{,}01) \approx$$
$$-(-0{,}50 - 0{,}5 - 0{,}5 - 0{,}066) = 1{,}57.$$

El menor número de bits necesarios —en promedio— para representar cada símbolo se ha reducido a 1,57.

Si volvemos a asignar a nuestros símbolos los cuatro códigos de 2 bits —00, 01, 10 y 11—, la redundancia sería $R = -1{,}57 + \log_2 4 = 0{,}43$. Esto sugiere la asignación de *códigos de tamaño variable* a los símbolos. El *Código1* de la Tabla 2.2 se ha diseñado de tal manera que al símbolo más común —a_1— se le asigna el código más corto. Cuando se transmiten largas cadenas de datos usando el *Código1*, el tamaño medio —el número de bits por símbolo— es de $1 \times 0{,}49 + 2 \times 0{,}25 + 3 \times 0{,}25 + 3 \times 0{,}01 = 1{,}77$, que es muy próximo al mínimo. La redundancia —en este caso— es: $R = 1{,}77 - 1{,}57 = 0{,}20$ bits por

Símbolo	Probabilidad	Código1	Código2
a_1	0,49	1	1
a_2	0,25	01	01
a_3	0,25	010	000
a_4	0,01	001	001

Tabla 2.2: Códigos de tamaño variable.

símbolo. Un ejemplo interesante es la siguiente cadena de 20 elementos:

$$a_1\, a_3\, a_2\, a_1\, a_3\, a_3\, a_4\, a_2\, a_1\, a_1\, a_2\, a_2\, a_1\, a_1\, a_3\, a_1\, a_1\, a_2\, a_3\, a_1\,,$$

donde los cuatro símbolos aparecen con —aproximadamente— las frecuencias correctas. La codificación de esta cadena con el *Código1* produce los siguientes 37 bits:

1|010|01|1|010|010|001|01|1|1|01|01|1|1|010|1|1|01|010|1

(sin las barras verticales). Usando 37 bits para codificar 20 símbolos, se obtiene un tamaño medio de 1,85 bits por símbolo, no muy lejos del tamaño medio calculado. (El lector debe tener en cuenta que nuestros ejemplos son cortos. Para obtener resultados cercanos al mejor teóricamente posible, se necesita una cadena de entrada formada por unos miles de elementos.)

Sin embargo, cuando tratamos de *decodificar* la cadena binaria de arriba, es obvio que *Código1* es una mala elección. El primer bit es 1 y ya que sólo se asigna a_1 a este código, éste —a_1— debe ser el primer símbolo. El siguiente bit es 0, pero los códigos a_2, a_3 y a_4 empiezan con un 0, por lo el decodificador tiene que leer el siguiente bit. Éste es 1, pero ambos códigos —a_2 y a_3— empiezan con 01. El decodificador no sabe si decodificar la cadena como 1|010|01 . . . —que es $a_1\, a_3\, a_2$ . . .—, o como 1|01|001 . . . —que es $a_1\, a_2\, a_4$ . . .—. *Código1* es, por lo tanto, *ambiguo*. Por el contrario, *Código2* —que tiene el mismo tamaño medio que *Código1*— puede ser decodificado sin ambigüedad.

La propiedad de *Código2* que hace que sea mucho mejor que *Código1* se llama *propiedad prefija*. Ésta requiere que una vez se ha asignado un cierto patrón de bits como código de un símbolo, ningún otro código debe comenzar con el mismo patrón (el patrón no puede ser el *prefijo* de cualquier otro código). Una vez asociada la cadena "1" como código de a_1, los demás códigos deben empezar con 0. A partir del momento en el que se asigna el código "01" a a_2, ningún otro código podrá comenzar por 01. Ello obliga a que los dos primeros dígitos de los códigos de a_3 y a_4 sean 00. Naturalmente, para diferenciarlos se necesita un dígito adicional, por lo que se convirtieron en 000 y 001 respectivamente.

El diseño de códigos de longitud variable, se realiza siguiendo estos dos principios: (1) Asignar códigos cortos a los símbolos más frecuentes y (2) obedecer la propiedad prefijo. El uso de estos principios produce códigos cortos y sin ambigüedades, pero no necesariamente los mejores (i.e., los más cortos). Además de estos principios, se necesita un algoritmo que siempre produce un conjunto de códigos cortos (los que tienen el tamaño medio mínimo). La única entrada a este tipo de algoritmo son las frecuencias —o las probabilidades— de los símbolos del alfabeto. Dos de estos algoritmos —el método de Shannon-Fano y el método de Huffman— se tratan en las Secciones 2.7 y 2.8.

(Cabe señalar que no todos los métodos de compresión estadística asignan códigos de tamaño variable a los símbolos individuales del alfabeto. Una excepción notable es la codificación aritmética —Sección 2.14—.)

2.3. Códigos prefijo

Un código prefijo es un código de tamaño variable que satisface la propiedad prefijo. La expresión binaria de los enteros no cumple la propiedad prefijo. Otra desventaja de esta representación es que

n	Código	Código alt.
1	0	1
2	10	01
3	110	001
4	1110	0001
5	11110	00001

Tabla 2.3: Algunos códigos unarios.

n	$a = 3 + n \cdot 2$	n-ésima palabra clave	Número de palabras clave	Rango de enteros
0	3	$0xxx$	$2^3 = 8$	0–7
1	5	$10xxxx$	$2^5 = 32$	8–39
2	7	$110xxxxxx$	$2^7 = 128$	40–167
3	9	$111xxxxxxxx$	$2^9 = 512$	168–679
		Total	680	

Tabla 2.4: El código unario general $(3, 2, 9)$.

el tamaño —n— de conjunto de enteros debe conocerse de antemano, ya que determina el tamaño del código, que es $1 + [\log_2 n]$ (los corchetes denotan "la parte entera de"). En algunas aplicaciones, se requiere un código prefijo para codificar un conjunto de enteros cuyo tamaño no se conoce de antemano. Aqui se presentan varios de estos códigos, la mayoría de los cuales se deben a Peter Elias [Elias 75]. Se puede encontrar más información sobre los códigos prefijo en el excelente informe técnico de 15 páginas [Fenwick 96a].

2.3.1. El código unario

El *código unario* de un entero positivo n se define como $n - 1$ unos seguidos de un solo cero (Tabla 2.3) o bien —como alternativa—, $n - 1$ ceros seguidos por un solo uno. La longitud del código unario para el entero n es, por lo tanto, n bits. En la edad de piedra se indicaba el número entero n marcando n barras verticales adyacentes en una piedra, por lo que el código unario a veces se llama binario de la edad de piedra[6] y cada uno de sus $n - 1$ unos recibe el nombre de bit de la edad de piedra[7].

◇ **Ejercicio 2.5 (sol. en pág. 1054):** Estúdiese el uso del código unario como un código de tamaño variable.

Además es posible definir códigos unarios generales, también conocidos como códigos inicio-paso-parada[8]. Tales códigos dependen de un triplete (inicio, paso, parada) de parámetros enteros y se definen de la siguiente manera: Se crean palabras clave para codificar los símbolos utilizados en los datos, de manera que el código n-ésimo se compone de n unos, seguidos de un 0, seguido de todas las combinaciones de a bits, donde $a = \mathit{inicio} + n \times \mathit{paso}$. Si $a = \mathit{parada}$, entonces el único 0 que precede a los a bits se desprecia. El número de códigos para un triplete dado es finito y depende de la elección de los parámetros. Las Tablas 2.4 y 2.5 muestran los 680 códigos de $(3, 2, 9)$ y los 2044 los códigos de $(2, 1, 10)$, respectivamente (véase también la Tabla 4.112 —pág. 411—). Estos códigos se discuten en la Sección 3.9 en relación con el método de compresión LZFG y en la Sección 4.16 para la compresión de imágenes sin pérdidas mediante búsqueda de bloque.

[6] Stone-age binary.
[7] Stone-age bit.
[8] Start-step-stop codes.

n	$a = 2 + n \cdot 1$	n-ésima palabra clave	Número de palabras clave	Rango de enteros
0	2	$0xx$	4	0-3
1	3	$10xxx$	8	4-11
2	4	$110xxxx$	16	12-27
3	5	$1110xxxxx$	32	28-59
	...	...		...
8	10	$\underbrace{11\ldots1}_{8}\underbrace{xx\ldots x}_{10}$	1024	1020-2043
		Total	2044	

Tabla 2.5: El código unario general $(2, 1, 10)$.

El número de códigos unarios generales diferentes es:

$$\frac{2^{\text{stop+step}} - 2^{\text{start}}}{2^{\text{step}} - 1} = \frac{2^{\text{parada+paso}} - 2^{\text{inicio}}}{2^{\text{paso}} - 1}.$$

Obsérvese que esta expresión aumenta de forma exponencial con el parámetro "parada," por lo que se pueden generar grandes conjuntos de estos códigos con pequeños valores de los tres parámetros.

◇ **Ejercicio 2.6 (sol. en pág. 1055):** ¿Qué códigos se definen con los parámetros $(n, 1, n)$ y cuáles con $(0, 0, \infty)$?

◇ **Ejercicio 2.7 (sol. en pág. 1055):** ¿Cuántos códigos produce el triplete (1, 1, 30)?

◇ **Ejercicio 2.8 (sol. en pág. 1055):** Dedúzcase el código unario general para (10,2,14).

2.3.2. Otros códigos prefijo

En esta sección, describimos cuatro códigos prefijo más. Denotamos como $B(n)$, al binario correspondiente a un entero n. Así, $|B(n)|$ es la longitud —en bits— de esta representación y $\bar{B}(n)$ denota a $B(n)$ sin su bit más significativo (que es siempre 1).

El código C_1 consta de dos partes. Para codificar el entero positivo n, en primer lugar se genera el código unario de $|B(n)|$ (el número de bits necesarios para la representación binaria de n), y luego se le agrega $\bar{B}(n)$. Un ejemplo es $n = 16 = 10000_2$. El tamaño de $B(16)$ es 5, por lo que comenzamos con el código unario 11110 (ó 00001) y anexamos al mismo $\bar{B}(16) = 0000$. Así, el código completo es 11110|0000 (ó 00001|0000). Otro ejemplo es $n = 5 = 101_2$, cuyo código es 110|01. La longitud de $C_1(n)$ es de $2 \cdot [\log_2 n] + 1$ bits[9]. Nótese que este código es idéntico al código unario general $(0, 1, \infty)$.

El código C_2 es una reordenación de C_1, donde se alternan cada uno de los $1 + [\log_2 n]$ bits de la primera parte (el código unario) de C_1 con uno de los bits de la segunda parte —de forma ordenada, y cada vez uno distinto, hasta agotarlos—. Por lo tanto, los códigos C_2 de 16 y 5 son: $C_2(16) = 101010100$ y $C_2(5) = 10110$.

El código C_3 comienza con $|B(n)|$ codificado con C_2, seguido por $\bar{B}(n)$. Así, 16 se codifica como $C_2(5) = 10110$ seguido por $\bar{B}(16) = 0000$ y 5 se codifica como $C_2(3) = 110$ seguido por $\bar{B}(5) = 01$. El tamaño de $C_3(n)$ es $1 + [\log_2 n] + 2 \cdot [\log_2(1 + [\log_2 n])]$.

Código C_4 se compone de varias partes. Comenzamos con $B(n)$. A la izquierda de éste, escribimos la representación binaria de $|B(n)| - 1$ (la longitud de n, menos 1). Se continúa de forma recursiva, hasta llegar a un número de 2 bits. Luego se añade un cero a la derecha del número completo generado,

[9] $[x]$ denota la parte entera de x.

para poder decodificarlo. Así, para codificar 16, comenzamos con 10000; añadimos a su izquierda: primero $|B(16)| - 1 = 4 = 100_2$ y luego $|B(4)| - 1 = 2 = 10_2$; puesto que éste es de dos bits, sólo queda añadir un cero a la derecha del número completo; el resultado es 10|100|10000|0. Para codificar 5, comenzamos con 101, agregamos $|B(5)| - 1 = 2 = 10_2$ a su izquierda, y finalmente un cero a la derecha. El resultado es 10|101|0.

◇ **Ejercicio 2.9 (sol. en pág. 1055):** ¿Cómo consigue el cero de la derecha hacer el código decodificable?

La Tabla 2.6 muestra ejemplos de los cuatro códigos anteriores, así como de $B(n)$ y $\bar{B}(n)$. Las longitudes de los cuatro códigos expresados en la tabla aumentan como $\log_2 n$, en contraste con la longitud del código unario —que crece como n—. Estos códigos, por lo tanto, son una buena elección en aquellos casos en que los datos se componen de n números enteros, con probabilidades que satisfacen ciertas condiciones. En concreto, la longitud L del código unario de n es $L = n = \log_2 2^n$, por lo que es ideal cuando $P(n) = 2^{-L} = 2^{-n}$. La longitud del código $C_1(n)$ es $L = 1 + 2 \cdot [\log_2 n] = \log_2 2 + \log_2 n^2 = \log_2 \left(2n^2\right)$, por lo que es ideal para el caso en que se cumpla:

$$P(n) = 2^{-L} = \frac{1}{2n^2}.$$

La longitud del código $C_3(n)$ es[10]:

$$L = 1 + [\log_2 n] + 2 \cdot [\log_2 (1 + [\log_2 n])] = \log_2 2 + [\log_2 n] + 2 \cdot [\log_2 (\log_2 2n)],$$

por lo tanto, es ideal cuando[11]:

$$P(n) = 2^{-L} = \frac{1}{2n \cdot (\log_2 2n)^2}.$$

La Tabla 2.7 muestra las probabilidades ideales que los ocho primeros números enteros positivos deberían tener para utilizar los tres códigos anteriores.

Pueden diseñarse más códigos prefijo para los números enteros positivos —adecuados para aplicaciones especiales—, siguiendo un enfoque general. Para ello, seleccione v_i enteros positivos y combínelos en una lista V (que puede ser finita o infinita, de acuerdo a las necesidades). El código del entero positivo n se prepara siguiendo estos pasos:

1. Busque un k tal que:
$$\sum_{i=1}^{k-1} v_i < n \leq \sum_{i=1}^{k} v_i.$$
2. Calcule la diferencia:
$$d = n - \sum_{i=1}^{k-1} v_i - 1.$$
El mayor valor de n es $\sum_1^k v_i$, por lo que el mayor valor de d es:
$$\sum_{i=1}^{k} v_i - \sum_{i=1}^{k-1} v_i - 1 = v_k - 1;$$
un número que se puede escribir en $[\log_2 (v_k)]$ bits. El número d se codifica, utilizando el código binario estándar, ya sea en este número de bits o —si $d < 2^{[\log_2 (v_k)]} - v_k$— en $[\log_2 (v_k)]$ bits.

[10] Se usan las propiedades de los logaritmos: $\log_a a = 1$ y $\log_a (x \cdot y) = \log_a x + \log_a y$.
[11] Recuérdese que $2^{\log_2 (x)} = x$, y $\log (x^n) = n \cdot \log (x)$.

n	$Unario$	$B(n)$	$\overline{B}(n)$	C_1	C_2	C_3	C_4
1	0	1		0|	0	0|	0
2	10	10	0	10|0	100	100|0	10|0
3	110	11	1	10|1	110	100|1	11|0
4	1110	100	00	110|00	10100	110|00	10|100|0
5	11110	101	01	110|01	10110	110|01	10|101|0
6	111110	110	10	110|10	11100	110|10	10|110|0
7	$\cdots$	111	11	110|11	11110	110|11	10|111|0
8		1000	000	1110|000	1010100	10100|000	11|1000|0
9		1001	001	1110|001	1010110	10100|001	11|1001|0
10		1010	010	1110|010	1011100	10100|010	11|1010|0
11		1011	011	1110|011	1011110	10100|011	11|1011|0
12		1100	100	1110|100	1110100	10100|100	11|1100|0
13		1101	101	1110|101	1110110	10100|101	11|1101|0
14		1110	110	1110|110	1111100	10100|110	11|1110|0
15		1111	111	1110|111	1111110	10100|111	11|1111|0
16		10000	0000	11110|0000	101010100	10110|0000	10|100|10000|0
31		11111	1111	11110|1111	111111110	10110|1111	10|100|11111|0
32		100000	00000	111110|00000	10101010100	11100|00000	10|101|100000|0
63		111111	11111	111110|11111	11111111110	11100|11111	10|101|111111|0
64		1000000	000000	1111110|000000	1010101010100	11110|000000	10|110|1000000|0
127		1111111	111111	1111110|111111	1111111111110	11110|111111	10|110|1111111|0
128		10000000	0000000	11111110|0000000	101010101010100	1010100|0000000	10|111|10000000|0
255		11111111	1111111	11111110|1111111	111111111111110	1010100|1111111	10|111|11111111|0

Tabla 2.6: Algunos códigos prefijo (C_1,C_2, C_3 y C_4).

n	Unario	C_1	C_3
1	0,5	0,5000000	
2	0,25	0,1250000	0,06250000
3	0,125	0,0555556	0,02494252
4	0,0625	0,0312500	0,01388889
5	0,03125	0,0200000	0,00906191
6	0,015625	0,0138889	0,00648410
7	0,0078125	0,0102041	0,00492748
8	0,00390625	0,0078125	0,00390625

Tabla 2.7: Probabilidades ideales de ocho enteros para tres códigos.

3. Codifique n en dos partes. Comience con k codificado en algún código prefijo y concatene el código binario de d. Ya que k se codifica en forma de código prefijo, cualquier decodificador debería saber cuántos bits tiene que leer para obtener k. Después de leer y decodificar k, el decodificador puede calcular el valor $2^{[\log_2(v_k)]} - v_k$, que le indica cuántos bits tiene que leer para determinar d.

Un sencillo ejemplo es la secuencia infinita $V = \left(1, 2, 4, 8, \ldots, 2^{i-1}, \ldots\right)$, con k codificado en unario. El número entero $n = 10$ cumple

$$\sum_{i=1}^{3} v_i < 10 \leq \sum_{i=1}^{4} v_i,$$

para $k = 4$ (con el código unario 1110) y $d = 10 - \sum_{i=1}^{3} v_i - 1 = 2$. Por lo tanto, el código de 10 es 1110|010.

Véase también el código Golomb —Sección 2.5—, los códigos binarios por etapas de la Sección 2.9.1, los códigos Rice —Sección 7.9—, y el código subexponential de la Sección 4.20.1.

Bases numéricas

Los números decimales utilizan la base 10. El número 2037_{10}, e.g., tiene un valor de $2\times10^3+0\times10^2+3\times10^1+7\times10^0$. Podemos decir que 2037 es la suma de los dígitos 2, 0, 3 y 7, cada uno ponderado por una potencia de 10. Los números fraccionarios se representan de la misma manera, utilizando potencias negativas de 10. De este modo, $0{,}82 = 8\times10^{-1}+2\times10^{-2}$ y $300{,}7 = 3\times10^2+7\times10^{-1}$.

Los números binarios utilizan la base 2. Tales números se representan como una suma de sus dígitos, cada uno ponderado por una potencia de 2. Así, $101{,}11_2 = 1\times2^2+0\times2^1+1\times2^0+1\times2^{-1}+1\times2^{-2}$.

Como no hay nada especial[12] en usar uno u otro, 10 ó 2, debe ser fácil convencerse de que cualquier entero positivo $n > 1$ puede servir como base para representar los números. En este caso se requieren n "dígitos" (si $n > 10$, utilizamos los diez dígitos y las letras $A, B, C, \ldots$); el número $d_3d_2d_1d_0.d_{-1}$ se representa como la suma de los $d_i$dígitos, cada uno multiplicado por una potencia de n, de este modo: $d_3n^3 + d_2n^2 + d_1n^1 + d_0n^0 + d_{-1}n^{-1}$. La base para un sistema numérico no tiene que consistir en potencias de un entero, sino que puede ser cualquier secuencia *superaditiva* que comience con 1.

Definición: Una *secuencia superaditiva* —$a_0, a_1, a_2, \ldots$— es aquella donde cualquier elemento —a_i— es mayor que la suma de todos sus predecesores. Un ejemplo es $1, 2, 4, 8, 16, 32, 64, \ldots$ donde cada elemento es igual a uno, más la suma de todos sus predecesores. Esta secuencia se compone de las

[12] En realidad, lo hay. Dos es el número entero más pequeño que puede ser una base para un sistema de numeración. Diez es el número de nuestros dedos.

familiares potencias de 2, así que sabemos que cualquier número entero se puede expresar por ella usando sólo los dígitos 0 y 1 (los dos bits). Otro ejemplo es $1, 3, 6, 12, 24, 50, \ldots$, donde cada elemento es igual a 2, más la suma de todos sus predecesores. Es fácil ver que cualquier número entero se puede expresar por ella usando sólo los dígitos 0, 1 y 2 (los tres trits).

Dado un entero positivo k, la secuencia $1, 1+k, 2+2k, 4+4k, \ldots, 2^i\,(1+k)$ es superaditiva, ya que cada elemento es igual a la suma de todos sus predecesores, más k. Cualquier entero no negativo puede ser representado *de forma única* en un sistema así como el número $x \ldots xxy$, donde los x son bits e y es un solo dígito en el rango $[0, k]$.

Por el contrario, una secuencia superaditiva general —como $1, 8, 50, 3102$— puede ser utilizada para representar enteros, pero no de forma única. El número 50, por ejemplo, es igual a $8 \times 6 + 1 + 1$, por lo que se puede representar como $0062 = 0 \times 3102 + 0 \times 50 + 6 \times 8 + 2 \times 1$, pero también se puede expresar como $0100 = 0 \times 3102 + 1 \times 50 + 0 \times 8 + 0 \times 1$.

Se puede demostrar que $1 + r + r^2 + \cdots + r^k$ es menor que r^{k+1} para cualquier número real $r > 1$. Esto implica que las potencias de cualquier número real $r > 1$ pueden servir como base de un sistema numérico que usa los dígitos $0, 1, 2, \ldots, d$ para algún d.

El número $\phi = \frac{1}{2}\left(1 + \sqrt{5}\right) \approx 1{,}618$ es la bien conocida *razón áurea*. Puede servir como base de un sistema numérico que usa los dos dígitos binarios. Así, e.g., $100{,}1_\phi = \phi^2 + \phi^{-1} \approx 3{,}23_{10}$.

Algunas bases reales tienen propiedades especiales. Por ejemplo, cualquier entero positivo R puede expresarse como $R = b_1F_1 + b_2F_2 + b_3F_3 + b_4F_5 + \cdots$ (es b_4F_5, no b_4F_4), donde los b_i son 0 ó 1 y los F_i son los números de Fibonacci $1, 2, 3, 5, 8, 13, \ldots$. Esta representación tiene la interesante propiedad de que la cadena $b_1b_2\ldots$ no contiene ningún 1 adyacente (esta propiedad la utilizan algunos métodos de compresión de datos, consulte la Sección 8.5.4). Como ejemplo, el entero 33 es igual a la suma $1 + 3 + 8 + 21$, por lo que se expresa en la base Fibonacci como el número de 7 bits 1010101.

◇ **Ejercicio 2.10 (sol. en pág. 1055):** Muéstrese cómo los números de Fibonacci se pueden utilizar para la construcción de un código prefijo.

Un entero no negativo se puede representar como una suma finita de coeficientes binomiales

$$n = \binom{a}{1} + \binom{b}{2} + \binom{c}{3} + \binom{d}{4} + \cdots, \quad \text{donde } 0 \le a < b < c < d \cdots$$

son números enteros y $\binom{i}{n}$ es el coeficiente binomial $\frac{i!}{n!(i-n)!}$. Este es el *sistema numérico binomial*.

¡Era una palabra tan majestuosa, la que Tarzán creó a partir de GOD[a]! El prefijo masculino de los simios es BU, el femenino MU; Tarzán había nombrado "g" como LA, "o" la pronunció TU y "d" era MO. Así que la palabra God se convirtió en BULAMUTUMUMO, o en inglés, he-g-she-o-she-d[b].

—Edgar Rice Burroughs, *The Jungle Tales of Tarzan*[c]

[a] No traduzco DIOS para no modificar la palabra obtenida por Tarzán.
[b] En español, él-g-ella-o-ella-d.
[c] *Los Cuentos de la Jungla de Tarzán.*

2.4. El código Tunstall

La principal ventaja de los códigos de longitud variable es su tamaño variable. Algunos códigos son cortos —y esta característica es la que produce compresión—. Como desventaja, es más difícil trabajar con códigos de tamaño variable. El codificador tiene que acumular y agregar varios de esos códigos en una memoria intermedia —un pequeño buffer—, esperar hasta llenar n bytes del buffer (donde n debe ser de al menos 1 bit), escribir los n bytes en la salida, desplazar n bytes del búfer y mantener el seguimiento de la ubicación del último bit colocado en el mismo. El decodificador tiene que realizar el proceso inverso. Es definitivamente más fácil tratar con códigos de tamaño fijo; y los códigos Tunstall aquí descritos son un ejemplo de cómo se pueden diseñar tales códigos. La idea es construir un conjunto de códigos de tamaño fijo, cada uno de los cuales codifica una cadena de tamaño variable de los símbolos de la entrada.

Imagine un alfabeto compuesto por dos símbolos A y B, donde A es más frecuente. Dada una cadena típica de este alfabeto, esperamos subcadenas de la forma AA, AAA, AB, AAB y B; pero raras veces cadenas de la forma BB. Por lo tanto, podemos asignar códigos de tamaño fijo para las siguientes cinco subcadenas como sigue: $AA = 000$, $AAA = 001$, $AB = 010$, $ABA = 011$ y $B = 100$. Una rara aparición de dos B consecutivas será codificada como 100100, pero la mayoría de las ocurrencias de B estarán precedidas por una A y se codificarán como 010, 011 ó 100.

Este ejemplo es malo e ineficiente. Malo, porque $AAABAAB$ puede codificarse como los cuatro códigos AAA, B, AA y B; pero también como los tres códigos AA, ABA, AB; la codificación no es única y puede requerir varias pasadas para determinar el código más corto. Esto ocurre porque nuestras cinco subcadenas no satisfacen la propiedad prefijo. Este ejemplo es ineficiente porque sólo se utilizan cinco de los ocho posibles códigos de 3 bits. Un código Tunstall de n bits debería utilizar los 2^n códigos posibles. Otro punto es que nuestros códigos se seleccionaron sin tener en cuenta las frecuencias relativas de los dos símbolos; y como resultado, no podemos estar seguros de que éste es el mejor código para nuestro alfabeto.

Por lo tanto, se necesita un algoritmo para desarrollar el mejor código Tunstall de n bits para un alfabeto dado de N símbolos; se proporciona un algoritmo válido en [Tunstall 67]. Dado un alfabeto de N símbolos, comenzamos con una tabla de códigos formada por esos símbolos. A continuación, iteramos mientras el tamaño de la tabla de códigos sea menor o igual que el número de códigos —2^n—. Cada iteración realiza los siguientes pasos:

- Selecciona el símbolo con la mayor probabilidad en la tabla. Llamémosle S.
- Retira S y añade N subcadenas Sx donde x recorre los N símbolos. Este paso incrementa el tamaño de la tabla en $N - 1$ símbolos (algunos de ellos pueden ser subcadenas). Por lo tanto, después de k iteraciones, el tamaño de la tabla será de $N + k\,(N - 1)$ elementos.
- Si $N + k\,(N - 1) \leq 2^n$, comienza otra iteración.

Es fácil ver que los elementos (símbolos y subcadenas) de la tabla satisfacen la propiedad prefijo y de este modo se garantiza una codificación única. Si la primera iteración agrega el elemento AB a la tabla, debe eliminar el elemento A. Por lo tanto, A no es un prefijo de AB. Si en la siguiente iteración crea el elemento ABR, entonces elimina el elemento AB, por lo que AB no es un prefijo de ABR. Esta construcción también minimiza el número promedio de bits por símbolo del alfabeto, debido a que se requiere la selección del elemento —o un elemento— de máxima probabilidad en cada iteración. Este requisito es similar a la forma en que se construye un código Huffman —Sección 2.8—; lo ilustramos con un ejemplo:

Dado un alfabeto con los tres símbolos —A, B y C— ($N = 3$), con probabilidades 0,7, 0,2 y 0,1, respectivamente, decidimos construir un conjunto de códigos Tunstall de 3 bits (por lo tanto, $n = 3$). Comenzamos la tabla de códigos con un árbol que consta de una raíz y tres hijos (Figura 2.8a). En la

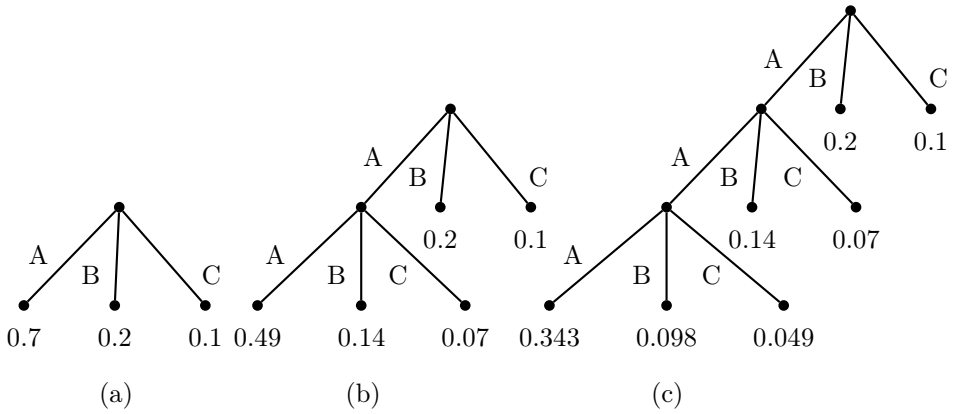


Figura 2.8: Ejemplo de código tunstall.

primera iteración, seleccionamos A y la convertimos en la raíz de un subárbol con hijos AA, AB y AC con probabilidades $0{,}7 \times 0{,}7 = 0{,}49$, $0{,}7 \times 0{,}2 = 0{,}14$ y $0{,}7 \times 0{,}1 = 0{,}07$, respectivamente (Figura 2.8b). La mayor probabilidad en el árbol es la del nodo AA, por lo que en la segunda iteración se convierte en la raíz de un subárbol con nodos AAA, AAB y AAC, con probabilidades $0{,}49 \times 0{,}7 = 0{,}343$, $0{,}49 \times 0{,}2 = 0{,}098$ y $0{,}49 \times 0{,}1 = 0{,}049$, respectivamente (Figura 2.8c). Después de cada iteración contamos el número de hojas del árbol y lo comparamos con $2^3 = 8$. Tras la segunda iteración hay siete hojas en el árbol, por lo que se finaliza el bucle. Se asignan siete códigos arbitrarios de 3 bits a los elementos AAA, AAB, AAC, AB, AC, B y C. El octavo código disponible debe asignarse a la subcadena que tiene la mayor probabilidad y que además satisface la propiedad prefijo.

La longitud media —en bits— de este código se calcula fácilmente así:

$$\frac{3}{3\,(0{,}343 + 0{,}098 + 0{,}049) + 2\,(0{,}14 + 0{,}07) + 0{,}2 + 0{,}1} = 1{,}37.$$

En general, damos a p_i y l_i la probabilidad y la longitud del nodo i del árbol, respectivamente. Si hay m nodos en el árbol, la longitud media en bits del código Tunstall es $n/\sum_{i=1}^{m} p_i l_i$. La entropía de nuestro alfabeto es:

$$-\,(0{,}7 \times \log_2 0{,}7 + 0{,}2 \times \log_2 0{,}2 + 0{,}1 \times \log_2 0{,}1) = 1{,}156,$$

por lo que los códigos Tunstall no proporcionan la mejor compresión.

Una propiedad importante de los códigos Tunstall es su fiabilidad. Si un bit es erróneo, sólo fallará un código. Normalmente, los códigos de longitud variable no se caracterizan por su seguridad. Un bit erróneo puede corromper la decodificación del resto de una larga secuencia de este tipo de códigos. Es posible la incorporación de códigos de control de errores en una cadena de códigos de tamaño variable, pero aumenta su tamaño y se reduce la compresión.

2.5. El código Golomb

El matemático francés Blaise Pascal del siglo XVII es conocido hoy sobre todo por sus contribuciones al campo de la probabilidad, pero hizo importantes contribuciones durante su corta vida a muchas áreas. En general hoy se acepta que él inventó —una primera versión de— el juego de la ruleta[13] (aun-

[13]Se atribuye a Blaise Pascal la primera ruleta con 36 números (sin el cero), un número vinculado a la magia, ya que la suma de los 36 primeros números es $\frac{36\times37}{2} = 666$.

que algunos creen que este juego se originó en China y fue traída a Europa por los monjes Dominicos que estaban comerciando con los Chinos). La versión moderna de la ruleta apareció en 1842.

La rueda de la ruleta tiene 37 depresiones poco profundas —conocidas como slots— numeradas del 0 al 36 (la versión americana cuenta con 38 ranuras numeradas 00, 0, y del 1 al 36). El crupier[14] —croupier— hace girar la rueda, mientras lanza una pequeña bola dentro de la misma, de manera que se desplace en la dirección opuesta al giro. Los jugadores pueden hacer sus apuestas mientras gira hasta que el crupier manifiesta: "no más apuestas". Cuando la rueda se detiene, el slot que contiene la bola determina el resultado del juego. A los jugadores que eligieron el número ganador se les paga según el tipo de apuesta que hicieron, mientras que los jugadores que se decidieron por los otros números pierden su apuesta a favor de la casa. [Bass 92] es un relato entretenido sobre un intento para predecir científicamente —y beneficiarse de— el resultado de un giro de la ruleta.

El tipo más simple de apuesta es a un solo número. Un jugador que gana esta apuesta obtiene 35 veces la cantidad apostada. Así, un jugador que juega varias veces y apuesta 1 dólar cada vez espera —en promedio— perder 36 veces y ganar una de cada conjunto de 37 juegos. El jugador, por lo tanto, pierde una media de 37 dólares por cada 35 de beneficio.

La probabilidad de ganar una apuesta es $p = 1/37 \approx 0{,}027027$, y la de perder una es —la mucho más alta— $q = 1 - p = 36/37 = 0{,}972973$. La probabilidad $P(n)$ de ganar una vez y perder $n - 1$ veces en una secuencia de n apuestas es el producto $q^{n-1}p$. Esta probabilidad está normalizada porque

$$\sum_{n=1}^{\infty} P(n) = \sum_{n=1}^{\infty} q^{n-1}p = p\sum_{n=0}^{\infty} q^n = \frac{p}{1-q} = \frac{p}{p} = 1.$$

A medida que n crece, $P(n)$ se reduce lentamente debido al valor mucho mayor de q. Los valores de $P(n)$ para $n = 1, 2, \ldots, 10$ son: 0,027027, 0,026297, 0,025586, 0,024895, 0,024222, 0,023567, 0,022930, 0,022310, 0,021707 y 0,021120.

Se dice que la función de probabilidad $P(n)$ obedece a una *distribución geométrica*. La razón para el nombre "geométrica" es el parecido de esta distribución a la secuencia geométrica. Una secuencia donde la razón entre dos elementos consecutivos es una constante q se llama geométrica. Esta secuencia tiene como elementos $a, aq, aq^2, aq^3, \ldots$. La suma (infinita) de estos elementos es una serie geométrica, $\sum_{i=0}^{\infty} aq^i$. El caso interesante es cuando q cumple $-1 < q < 1$, en el que la serie converge a $a/(1-q)$. La Figura 2.9 muestra la distribución geométrica para $p = 0{,}2$, 0,5 y 0,8.

Muchos métodos de compresión se basan en la codificación run-length (RLE). Imagínese una cadena binaria donde aparece un cero con una probabilidad p y un uno con una probabilidad $1 - p$. Si p es grande, habrá secuencias de ceros adyacentes, lo que sugiere el uso de RLE para comprimir la cadena. La probabilidad de encontrar n ceros seguidos es p^n y la probabilidad de que aparezcan n ceros seguidos de un 1 es $p^n(1-p)$, lo que indica que los *run lengths* (grupos de elementos o rachas —de unos, de ceros, etc—) se distribuyen geométricamente. Un enfoque sencillo para comprimir una cadena de este tipo es calcular la probabilidad de cada *run length* y aplicar el método de Huffman (Sección 2.8) para obtener los mejores códigos prefijo para cada uno de ellos. En la práctica, sin embargo, puede haber un gran número de run lengths y esta cantidad no se conoce de antemano. Un enfoque mejor es construir una familia infinita de códigos prefijo óptimos, de modo que —sea cual sea la longitud de un *run*— exista un código de la familia para codificarla. Los códigos de la familia deben depender de la probabilidad p, por lo que estamos buscando un conjunto infinito de *códigos prefijo parametrizados*. Los códigos Golomb aquí descritos [Golomb 66], son de este tipo y son los mejores para la compresión de elementos de datos que se distribuyen geométricamente.

Examinemos primero algunos números para ver por qué estos códigos deben depender de p. Para $p = 0{,}99$, las probabilidades de aparición de secuencias adyacentes de dos y 10 ceros, son respectivamente de $0{,}99^2 = 0{,}9801$ y $0{,}99^{10} = 0{,}90$ (ambas grandes). En cambio, para $p = 0{,}6$, los mismos

[14] Repartidor, negociador; en este caso a esta persona se le llama crupier.

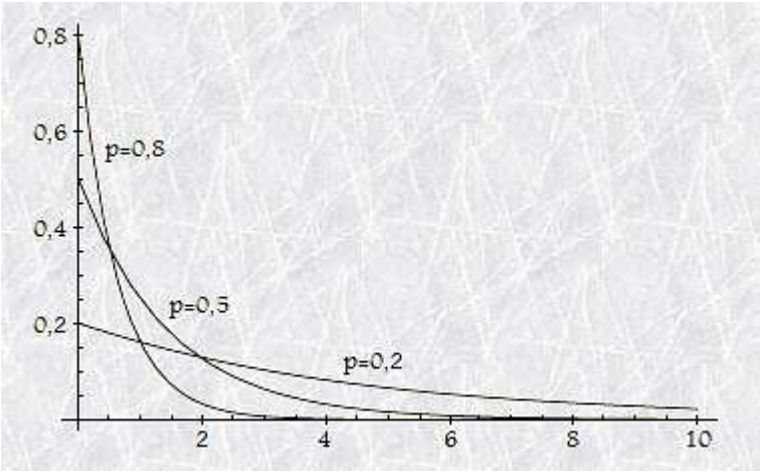


Figura 2.9: Distribuciones geométricas para $p = 0{,}2$, 0,5 y 0,8.

run lengths tienen probabilidades mucho menores: 0,36 y 0,006. La razón $^{0{,}9801}/_{0{,}36}$ es 2,7225, pero la razón $^{0{,}9}/_{0{,}006} = 150$ es mucho mayor. Por lo tanto, una p grande implica mayores probabilidades para encontrar *runs* largos, mientras que una p pequeña indica que esas largas secuencias de elementos apenas aparecerán.

Dos conceptos estadísticos relevantes son la media y la mediana de una secuencia de *run lengths*. Éstos están representados por la cadena binaria:

$$0000010011000101000000111010001000001000100100011010000 1001 \tag{2.1}$$

que contiene los 18 *run lengths* 5, 2, 0, 3, 1, 6, 0, 0, 1, 3, 5, 3, 2, 3, 0, 1, 4 y 2. Su media es:

$$\frac{5+2+0+3+1+6+0+0+1+3+5+3+2+3+0+1+4+2}{18} \approx 2{,}28.$$

Su mediana m es el valor que cumple que aproximadamente la mitad de los *run lengths* sean más cortos que m y la otra mitad sean iguales o más grandes que m. Para encontrar m, ordenamos los 18 *run lengths* para obtener 0, 0, 0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3, 3, 4, 5, 5 y 6 y buscamos la mediana (el número central), que en este caso es 2.

Ahora estamos preparados para ver una descripción del código Golomb.

Codificación. El código Golomb para n enteros no negativos depende de la elección de un parámetro m (veremos más adelante que para RLE, m debe depender de la probabilidad p y de la mediana de los *run lengths*). Por lo tanto, es un código prefijo parametrizado, lo que lo convierte en especialmente útil en aquellos casos donde puedan calcularse o estimarse valores ventajosos para el parámetro (véase, por ejemplo, la Sección 4.22). El primer paso para codificar el código Golomb del entero no negativo n, es calcular las tres cantidades —q(cociente), r(resto) y c— como sigue:

$$q = \left\lfloor \frac{n}{m} \right\rfloor, \qquad r = n - qm \qquad y \qquad c = \lceil \log_2 m \rceil,$$

tras lo cual se construye el código en dos partes; la primera, es el valor de q codificado en unario (Ejercicio 2.3.1 —pág 60—) y la segunda, es el valor binario de r codificado de una manera especial: los primeros $2^c - m$ valores de r se codifican como enteros sin signo —en $c - 1$ bits cada uno— y el resto, se codifica en c bits cada uno (terminando con el mayor número de c bits, que consiste en c 1's).

m	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
c	1	2	2	3	3	3	3	4	4	4	4	4	4	4	4
$2^c - m$	0	1	0	3	2	1	0	7	6	5	4	3	2	1	0

Tabla 2.10: (a) Algunos ejemplos de m, c y $2^c - m$.

m/n	0	1	2	3	4	5	6
2	0 | 0	0 | 1	10 | 0	10 | 1	110 | 0	110 | 1	1110 | 0
3	0 | 0	0 | 10	0 | 11	10 | 0	10 | 10	10 | 11	110 | 0
4	0 | 00	0 | 01	0 | 10	0 | 11	10 | 00	10 | 01	10 | 10
5	0 | 00	0 | 01	0 | 10	0 | 110	0 | 111	10 | 00	10 | 01
6	0 | 00	0 | 01	0 | 100	0 | 101	0 | 110	0 | 111	10 | 00
7	0 | 00	0 | 010	0 | 011	0 | 100	0 | 101	0 | 110	0 | 111
8	0 | 000	0 | 001	0 | 010	0 | 011	0 | 100	0 | 101	0 | 110
9	0 | 000	0 | 001	0 | 010	0 | 011	0 | 100	0 | 101	0 | 110
10	0 | 000	0 | 001	0 | 010	0 | 011	0 | 100	0 | 101	0 | 1100
11	0 | 000	0 | 001	0 | 010	0 | 011	0 | 100	0 | 1010	0 | 1011
12	0 | 000	0 | 001	0 | 010	0 | 011	0 | 1000	0 | 1001	0 | 1010
13	0 | 000	0 | 001	0 | 010	0 | 0110	0 | 0111	0 | 1000	0 | 1001

Tabla 2.10: (b) Algunos códigos Golomb para $m = 2, \ldots, 13$; $n = 0, \ldots, 6$.

Hay un caso especial, cuando m es una potencia de 2 ($m = 2^c$), ya que no requiere ningún código de $c - 1$ bits. Sabemos que $n = r + qm$, por lo que una vez decodificado un código Golomb, los valores de q y r se pueden utilizar para reconstruir fácilmente n.

Ejemplos. La elección de $m = 3$, produce $c = 2$ y los tres restos 0, 1 y 2. Calculamos $2^2 - 3 = 1$, por lo que el primer resto se codifica en $c - 1 = 1$ bits como 0 y los dos restantes se codifican en dos bits cada uno —terminando con 11_2—, como 10 y 11. Si seleccionamos $m = 5$, quedan determinados $c = 3$ y los cinco restos —del 0 al 4—. Los tres primeros ($2^3 - 5 = 3$), se codifican en $c - 1 = 2$ bits cada uno, y los dos restantes, se codifican en tres bits finalizando con el 111_2 —esto es, 00, 01, 10, 110 y 111. A continuación se indica una regla sencilla que muestra cómo codificar los números de c bits, de tal manera que el último de ellos sea c 1's: denotamos el mayor de los números de $c - 1$ bits por b;

m/n	7	8	9	10	11	12
2	1110 | 1	11110 | 0	11110 | 1	111110 | 0	111110 | 11	1111110 | 0
3	110 | 10	110 | 11	1110 | 0	1110 | 10	1110 | 11	11110 | 0
4	10 | 11	110 | 00	110 | 01	110 | 01	110 | 11	1110 | 00
5	10 | 10	10 | 110	10 | 111	110 | 00	110 | 01	110 | 10
6	10 | 01	10 | 100	10 | 101	10 | 110	10 | 111	110 | 00
7	10 | 00	10 | 010	10 | 011	10 | 100	10 | 101	10 | 110
8	0 | 111	10 | 000	10 | 001	10 | 010	10 | 011	10 | 100
9	0 | 1110	0 | 1111	10 | 000	10 | 001	10 | 010	10 | 011
10	0 | 1101	0 | 1110	0 | 1111	10 | 000	10 | 001	10 | 010
11	0 | 1100	0 | 1101	0 | 1110	0 | 1111	10 | 000	10 | 001
12	0 | 1011	0 | 1100	0 | 1101	0 | 1110	0 | 1111	10 | 000
13	0 | 1010	0 | 1011	0 | 1100	0 | 1101	0 | 1110	0 | 1111

Tabla 2.10: (c) Algunos códigos Golomb para $m = 2, \ldots, 13$; $n = 7, \ldots, 12$.

n	0	1	2	3	...	13	14	15	16	17
$q = \left\lfloor \frac{n}{14} \right\rfloor$	0	0	0	0	...	0	1	1	1	1
$unario\,(q)$	0	0	0	0	...	0	10	10	10	10
r	000	001	0100	0101	...	1111	000	001	0100	0101
n						...	27	28	29	30
$q = \left\lfloor \frac{n}{14} \right\rfloor$						...	1	2	2	2
$unario\,(q)$						...	10	110	110	110
r						...	1111	000	001	0100

Tabla 2.11: Algunos Códigos Golomb para $m = 14$.

luego, construimos el entero $b + 1$ con $c - 1$ bits, y añadimos un cero a la derecha; el resultado es el primero de los números de c bits, y los restantes se obtienen incrementando.

La Tabla 2.10a muestra algunos ejemplos de m, c y $2^c - m$; las Tablas2.10b y 2.10c muestran algunos códigos Golomb para $m = 2, \ldots, 13$.

Ilustremos con un ejemplo un poco más largo, seleccionando $m = 14$. Este valor produce $c = 4$ y los 14 restos —del 0 al 13—. Los dos primeros ($2^4 - 14 = 2$) se codifican en $c - 1 = 3$ bits cada uno y los 12 restantes se codifican en cuatro bits cada uno, terminando en el 1111_2 (y como consecuencia, comenzando por el 0100_2). Por lo tanto, tenemos 000, 001, seguidos por los 12 valores: 0100, 0101, 0110, 0111, ..., 1111. En la Tabla 2.11 se pueden ver varios ejemplos detallados. La Tabla 2.12 muestra 48 códigos para $m = 14$ y para $m = 16$; los primeros comienzan con dos códigos de 4 bits, seguidos por conjuntos de 14 códigos, de manera que todos los códigos de un conjunto se representan con un bit más que los del precedente; los últimos son más sencillos, porque 16 es una potencia de 2. Los códigos Golomb para $m = 16$ consisten en conjuntos de 16 códigos cada uno y un bit más que el anterior. Los códigos Golomb para el caso en el que m es una potencia de 2, han sido desarrollados por Robert F. Rice y se conocen como códigos Rice. Muchos algoritmos de compresión de audio sin pérdidas emplean este código. Un ejemplo típico es Shorten (Sección 7.9).

Las Tablas 2.11 y 2.12 ilustran el efecto de m en la longitud del código. Para valores pequeños de m, los códigos Golomb empiezan siendo cortos y aumentan rápidamente su longitud. Son apropiados para RLE en aquellos casos en que la probabilidad p de aparición del cero es pequeña, lo que implica muy pocos *runs* largos. Para valores grandes de m, los códigos iniciales (para $n = 1, 2, \ldots$) son largos, pero sus longitudes aumentan lentamente. Estos códigos tienen sentido para RLE cuando p es grande, lo que implica que se esperan muchos *runs* largos.

Decodificación. Los códigos Golomb se diseñaron de esa manera tan especial para facilitar su decodificación. En primer lugar, demostremos la decodificación para el sencillo caso $m = 16$ (m es una potencia de 2). Para decodificar, se comienza por el extremo izquierdo del código y se cuenta el número —A— de 1's que preceden el primer 0. La longitud del código es $A + c + 1$ bits (para $m = 16$, ésta es $A + 5$ bits). Si denotamos los cinco bits de más a la derecha del código por R, entonces el valor del código es $16A + R$. Esta simple decodificación refleja la forma como se construyó el código. Para codificar n con $m = 16$, se comienza dividiéndolo por 16 para obtener $n = 16A + R$; a continuación, se escriben A 1's seguidos por un solo cero, seguido por la representación de R en 4 bits.

Para valores de m que no son potencias de 2, la decodificación es ligeramente más complicada. Suponiendo de nuevo que un código comienza con A 1's, se empieza por eliminarlos junto con el cero que se encuentra inmediatamente tras ellos. Designamos los $c - 1$ bits siguientes por R. Si $R < 2^c - m$, entonces la longitud total del código es $A + 1 + (c - 1)$ (los A 1's, el cero tras ellos y los $c - 1$ bits siguientes) y su valor es $m \times A + R$. Si $R \geq 2^c - m$, entonces la longitud total del código es $A + 1 + c$ y su valor es $m \times A + R' - (2^c - m)$, donde R' es el entero de c bits formado por R y el bit siguiente

$m = 14$				$m = 16$			
n	*Código*	n	*Código*	n	*Código*	n	*Código*
0	0000	24	101100	0	00000	24	101000
1	0001	25	101101	1	00001	25	101001
		26	101110	2	00010	26	101010
2	00100	27	101111	3	00011	27	101011
3	00101	28	110000	4	00100	28	101100
4	00110	29	110001	5	00101	29	101101
5	00111			6	00110	30	101110
6	01000	30	1100100	7	00111	31	101111
7	01001	31	1100101	8	01000		
8	01010	32	1100110	9	01001	32	1100000
9	01011	33	1100111	10	01010	33	1100001
10	01100	34	1101000	11	01011	34	1100010
11	01101	35	1101001	12	01100	35	1100011
12	01110	36	1101010	13	01101	36	1100100
13	01111	37	1101011	14	01110	37	1100101
14	10000	38	1101100	15	01111	38	1100110
15	10001	39	1101101			39	1100111
		40	1101110	16	100000	40	1101000
16	100100	41	1101111	17	100001	41	1101001
17	100101	42	1110000	18	100010	42	1101010
18	100110	43	1110001	19	100011	43	1101011
19	100111			20	100100	44	1101100
20	101000	44	11100100	21	100101	45	1101101
21	101001	45	11100101	22	100110	46	1101110
22	101010	46	11100110	23	100111	47	1101111
23	101011	47	11100111				

Tabla 2.12: Los primeros códigos Golomb para $m = 14$ y $m = 16$.

a R.

Un ejemplo es el código $0001xxx$, para $m = 14$. No hay 1's al principio, por lo que A es 0. Después de quitar el cero de más a la izquierda, los $c - 1 = 3$ bits siguientes son $R = 001$. Puesto que $R < 2^c - m = 2$, concluímos que la longitud del código es $0 + 1 + (4 - 1) = 4$ y su valor es 001. Del mismo modo, el código $00100xxx$ para $m = 14$ determina $A = 0$ y $R = 010_2 = 2$. En este caso, $R \geq 2^c - m = 2$, por lo que la longitud del código es $0 + 1 + c = 5$, el valor de R' es $0100_2 = 4$, y el valor del código es: $14 \times 0 + 4 - 2 = 2$.

Las Secciones 4.9.1 y 4.22 ilustran el uso del código Golomb para la compresión de imágenes sin pérdidas.

Ahora está claro que el mejor valor para m depende de p, y se puede demostrar que este valor es el entero más cercano a $-1/\log_2 p$ o, equivalentemente, el valor que satisface:

$$p^m \approx \frac{1}{2}. \tag{2.2}$$

También puede demostrarse que en el caso de una sucesión de *run lengths*, este entero es la mediana de los *run lengths*. Así, para $p = 0{,}5$, m debe ser $^{-1}/_{\log_2 0{,}5} = 1$; para $p = 0{,}7$, m debe ser 2, porque $^{-1}/_{\log_2 0{,}7} \approx 1{,}94$; y para $p = {}^{36}/_{37}$, m debe ser 25, porque $^{-1}/_{\log_2({}^{36}/_{37})} \approx 25{,}29$.

Además debe mencionarse que Gallager y van Voorhis [Gallager y van Voorhis 75] han refinado y

extendido la Ecuación (2.2) en la relación —más precisa—:

$$p^m + p^{m+1} \leq 1 < p^m + p^{m-1}. \tag{2.3}$$

Demostraron que el código Golomb es el mejor código prefijo cuando m se selecciona cumpliendo su desigualdad. En primer lugar, observamos que para un p dado, la desigualdad (2.3) sólo tiene una solución para m. A continuación manipulamos esta desigualdad con los cuatro pasos siguientes:

$$p^m (1+p) \leq 1 < p^{m-1} (1+p),$$

$$p^m \leq \frac{1}{(1+p)} < p^{m-1},$$

$$m \geq \frac{1}{\log_2 p} \log_2 \frac{1}{1+p} > m-1,$$

$$m \geq -\frac{\log_2 (1+p)}{\log_2 p} > m-1,$$

desde la cual es claro que el único valor para m es:

$$m = \left[-\frac{\log_2 (1+p)}{\log_2 p} \right]. \tag{2.4}$$

Presentamos aquí tres ejemplos, para ilustrar el rendimiento del código Golomb en la compresión de los *run lengths*. El primer ejemplo, es la cadena binaria (2.1), que consta de 41 ceros y 18 unos. La probabilidad de aparición de un cero es, por lo tanto, $^{41}/_{(41+18)} \approx 0{,}7$, que genera $m = [-^{\log_2 1{,}7}/_{\log_2 0{,}7}] = [1{,}487] = 2$. La secuencia de *run lengths* es: 5, 2, 0, 3, 1, 6, 0, 0, 1, 3, 5, 3, 2, 3, 0, 1, 4 y 2; por lo tanto, pueden codificarse con los códigos Golomb para $m = 2$ en la siguiente cadena de 18 códigos:

1101|100|00|101|01|11100|00|00|01|101|1101|101|100|101|00|01|1100|100.

El resultado es una cadena de 52 bits que comprime los 59 bits de origen. Casi no hay compresión porque p no es grande. Téngase en cuenta que esa cadena (2.1) tiene cuatro *runs* de longitud 0 (creando tres *runs* de 1's) y tres de longitud 1. El siguiente ejemplo es la cadena de 98 bits

00000000001000000000100000001000000000001000000001
00000000000010000000010000000100000000000100000000,

—que es menos densa—, y por lo tanto, se comprime mejor. Se compone de 89 ceros y 9 unos, por lo que $p = {}^{89}/_{(89+9)} = 0{,}91$. El mejor valor para m es, por lo tanto, $m = [-^{\log_2(1{,}91)}/_{\log_2(0{,}91)}] = [6{,}86] = 7$. Los 10 *runs* de ceros tienen longitudes 10, 9, 7, 11, 8, 12, 8, 7, 10 y 7. Cuando se codifica con los códigos Golomb para $m = 7$, los *run lengths* se convienten en la cadena de 47 bits:

10100 | 10011 | 1000 | 10101 | 10010 | 10110 | 10010 | 1000 | 10100 | 1000,

lo que produce un factor de compresión de $^{98}/_{47} = 2{,}1$.

El tercero es un caso extremo, compuesto por una cadena binaria —de digamos, 10^6 bits— muy poco densa —sólo 100 son unos, y están distribuídos uniformemente— (consúltese la Sección 8.5 para ver otros métodos de compresión de cadenas poco densas). La probabilidad de aparición del cero es $p = {}^{10^6}/_{(10^6+10^2)} = 0{,}9999$, lo que implica que $m = 6931$. Hay 101 *runs* con aproximadamente 10^4 ceros cada uno . El código Golomb de 10^4 para $m = 6931$ es de 14 bits de longitud, por lo cual, los

101 *runs* se pueden comprimir en no más de $14 \times 101 = 1414$ bits, dando el impresionante factor de compresión de ¡$^{10^6}/_{1414} \approx 707$!

En resumen, dada una cadena binaria, podemos comprimir con RLE siguiendo estos pasos: (1) cuéntense el número de ceros y de unos; (2) calcúlese la probabilidad p de aparición de un cero; (3) úsese la ecuación (2.4) para calcular m; (4) constrúyase el conjunto de códigos Golomb para m; y (5) para cada *run-length* de n ceros, escríbase el código Golomb de n en la secuencia de datos comprimida.

Para que los *run lengths* sean significativos, p debe ser grande. Con valores pequeños de p, como *0,1*, se produce una cadena con más unos que ceros, y por lo tanto, con muchos *runs* cortos de ceros, y largos de unos. En este caso, es posible utilizar RLE para comprimir los *runs* de unos. En general, podemos hablar de una cadena binaria cuyos elementos son r y s (por *run* y *stop*). Para r, debemos seleccionar el elemento más común, pero tiene que ser muy frecuente (la distribución de r y de s debe ser sesgada) para que RLE produzca una buena compresión. Los valores de p cercanos a 0,5 se traducen en *runs* compuestos por ceros y unos, por lo que independientemente de qué bit se seleccione para el elemento r, habrá muchos *runs* de longitud cero. Por ejemplo, la cadena 000111001100001111101000111 tiene los siguientes *run lengths* de ceros: 3, 0, 0, 2, 0, 4, 0, 0, 0, 1, 3, 0, 0 y si consideramos los *run lengths* de unos obtenemos resultados similares: 0, 0, 0, 3, 0, 2, 0, 0, 0, 4, 1, 0, 0, 3. En cuyo caso, RLE no es una buena elección para la compresión y deben considerarse otros métodos.

También es común el caso donde p es desconocido y no se puede calcular (o incluso ser estimado) por adelantado; o bien porque la cadena es muy larga, o bien porque tiene que comprimirse en tiempo real mientras llegan los datos de origen. En tal caso, un algoritmo adaptativo —el cual varía m de acuerdo con los datos recibidos hasta el momento—, es la mejor opción. Dicho algoritmo, llamado *Goladap* [Langdon 83a], es el que aquí describimos.

Goladap se basa en la observación de que el mejor valor de m se da cuando $p^m = 1/2$. También emplea el hecho de que el código Golomb de un *run length* n comienza con el cociente $n \div m$ (expresado en unario) y termina con el resto correspondiente. Cuando el codificador comienza a leer los ceros de un *run*, no se conoce la longitud n del mismo. Por consiguiente, puede incrementar un contador cada vez que lee un cero y cuando éste alcance el valor m; añadir un 1 al código unario, poner a cero el contador y seguir leyendo ceros del *run*. Cuando termina el *run* (se ha leído un 1 de la entrada), el codificador anexa un cero para completar el código unario y a continuación agrega el valor del contador (ya que éste es el resto). El caso $m = 2^k$ es especialmente sencillo, porque un contador de k bits puede contar desde 0 hasta $k - 1$; al añadir 1 a su cuenta máxima, se desborda y se pone a cero a sí mismo. Si se selecciona m usando la mediana, la mitad de los *runs* deben ser inferiores a m y no causar desbordamiento, mientras que la otra mitad sí que deben producirlo.

La parte adaptativa de Goladap cambia el valor de m (mientras sea una potencia de 2) conforme al desbordamiento del contador. Si no hay desbordamiento del contador durante la codificación de un *run*, entonces k se decrementa en 1 (lo que reduce m a la mitad); de lo contrario se incrementa en 1 (lo que duplica m) cada vez que se desborde el contador. El nuevo valor de m se utiliza para codificar el siguiente *run* de ceros, por lo que el decodificador puede actualizar m en el mismo momento que el codificador. La justificación para este método es que un *run* largo (se producen varios desbordamientos del contador durante la codificación de un *run*) indica que hay muchos ceros (una p grande) y por lo tanto, una distribución asimétrica de ceros y unos, por lo que es conveniente que m sea grande. Un *run* corto (no hay desbordamiento en el contador) indica lo contrario, por lo que un m más pequeño debería obtener mejores resultados.

Así funciona el codificador Goladap: Emplea dos variables —K (codificación) y C (cuenta)—. Ambas se inicializan a cero. Para cada 0 leído de la entrada, se incrementa C (pero sólo si K es positivo). Cuando `C mod 2K` es cero: se anexa un uno al código unario, se borra C y se incrementa K en 1 (excepto si el contador ya está al máximo). Cuando se lee un 1 de la entrada (señalando el final del *run* actual), se anexa un 0 al código disponible hasta el momento, concluyendo la parte unaria. A su derecha se añaden los K bits menos significativos de C (el resto). La cuenta de C se pone a cero y K se decrementa en 1 (a menos que ya sea 0). A continuación mostramos un ejemplo de codificación.

```
L = 0; % inicializar
N = 0;
m = 1; % o preguntar al usuario por m
% bucle principal
para cada run de r ceros hacer
    construir el código Golomb para r usando el actual m.
    escribirlo en la cadena comprimida.
    L = L + r; % actualizar L, N y m
    N = N + 1;
    p = L/(L + N);
    m = [−1/log2p + 0.5];
finpara;
```

Figura 2.13: Un sencillo codificador RLE Golomb adaptativo.

Imagine un *run* de nueve ceros seguidos de un 1. Tanto C como K se inicializan a 0.

1. El primer cero de la entrada añade un 1 al —hasta ahora vacío— código unario, e incrementa K en 1.

2. El segundo cero incrementa C a 1.

3. El tercer cero incrementa C a 2. Dado que $k = 1$, se anexa otro 1 al código unario, se incrementa K a 2 y se pone C a cero.

4. Los siguientes cuatro ceros (del cuarto a séptimo) incrementan C, desde 0 hasta 4; en ese momento se adjunta otro 1 al código unario, K se incrementa a 3 y C se pone a cero.

5. Los siguientes dos ceros (octavo y noveno) incrementan C a 2.

6. (El código unario es ahora 111.) Se lee el 1 siguiente al *run*. Se anexa un 0 al código unario, seguido por el valor de C en K bits, i.e., 010. El código Golomb completo es 1110010.

El decodificador Goladap lee los primeros 1's del código unario. Por cada 1, emite 2^k ceros e incrementa K. Cuando se lee el 0 que finaliza el código unario, el decodificador lee los siguientes K bits, los convierte en un entero y genera ese número de ceros, seguido por un solo 1. Si K no es ya cero, se decrementa.

Otro enfoque para el RLE adaptativo es utilizar las cadenas binarias de entrada disponibles hasta el momento para la estimación de p y a partir de ella, calcular m; y luego utilizar el nuevo valor de m para codificar el siguiente *run length* (no el actual, debido a que el decodificador no puede imitar ésto). Imagínese que se han introducido hasta ahora tres *runs* de 10, 15 y 21 ceros, y los dos primeros han ya han sido comprimidos. El *run* actual de 21 ceros se comprime primero con el valor actual de m; después se calcula un nuevo valor para p —el resultado de $^{(10+15+21)}/_{[(10+15+21)+3]}$— y se utiliza para actualizar m, haciendo la operación $-^1/_{\log_2 p}$ o la indicada en la ecuación (2.4). (3 es el número de entradas a 1 recibidas hasta el momento —una por cada *run*—.) El nuevo m se utiliza para comprimir el siguiente *run*. El algoritmo acumula las longitudes de los *runs* en la variable L, y su número en N. La Figura 2.13 es un listado de un sencillo pseudocódigo de este método. (Una implementación práctica debería reducir a la mitad los valores de L y N de vez en cuando, para evitar que se desborden.)

El autor está en deuda con Cosmin Truţa por señalar los errores en las anteriores versiones de esta sección y por la revisión de la versión actual.

Además de los códigos, Solomon W. Golomb tiene su "propia" constante Golomb: 0,624329988543550870992936383100837244179642620180529286.

2.6. La desigualdad de Kraft-MacMillan

Esta inecuación se relaciona con códigos no ambiguos de longitud variable. Su primera parte dice que dado un código no ambiguo de tamaño variable, con n códigos de tamaños L_i, entonces

$$\sum_{i=1}^{n} 2^{-L_i} \leq 1. \tag{2.5}$$

La segunda parte declara el opuesto: dado un conjunto de n enteros positivos $(L_1, L_2, \ldots, L_n)$ que satisfacen la ecuación (2.5), existe un código no ambiguo de tamaño variable tal que los L_i son los tamaños de sus códigos individuales. Juntas, ambas partes dicen que un código no es ambiguo si y sólo si satisface la relación (2.5).

Esta desigualdad se puede relacionar con la entropía observando que las longitudes —L_i— siempre pueden escribirse como $L_i = -\log_2 P_i + E_i$, donde E_i es simplemente la cantidad en la que L_i es mayor que la entropía (la longitud adicional del código i).

Esto implica que:

$$2^{-L_i} = 2^{(\log_2 P_i - E_i)} = \frac{2^{\log_2 P_i}}{2^{E_i}} = \frac{P_i}{2^{E_i}}.$$

En el caso especial en el que todas las longitudes extras son iguales ($E_i = E$), la desigualdad de Kraft dice que

$$1 \geq \sum_{i=1}^{n} \frac{P_i}{2^E} = \frac{\sum_{i=1}^{n} P_i}{2^E} = \frac{1}{2^E} \Rightarrow 2^E \geq 1 \Rightarrow E \geq 0.$$

Un código no ambiguo tiene una longitud extra no negativa, lo que significa que su longitud es mayor que o igual a la longitud determinada por su entropía.

A continuación mostramos un sencillo ejemplo del uso de esta desigualdad. Consideremos el sencillo caso de n códigos binarios de igual longitud. El tamaño de cada código es $L_i = log_2 n$ y la suma de Kraft-MacMillan es:

$$\sum_{1}^{n} 2^{-L_i} = \sum_{1}^{n} 2^{-\log_2 n} = \sum_{1}^{n} \frac{1}{n} = 1.$$

La desigualdad se cumple, por lo que dicho código no es ambiguo (la decodificación es única).

Las estadísticas muestran que hay más mujeres en el mundo que cualquier otra cosa, excepto insectos.

—Glenn Ford como Johnny Farrell en *Gilda* (1946)

Un ejemplo más interesante es cuando tenemos n códigos, donde el primero está comprimido y el segundo expandido. Asignamos $L_1 = \log_2 n - a$, $L_2 = \log_2 n + e$ y $L_3 = L_4 = \cdots = L_n = \log_2 n$, donde a y e son positivos. Demostremos que $e > a$, lo que significa que la compresión de un símbolo por un

factor a requiere la expansión de otro símbolo por un factor mayor. Podemos beneficiarnos de esto sólo si la probabilidad de aparición del símbolo comprimido es mayor que la del símbolo expandido.

$$\begin{aligned}\sum_1^n 2^{-L_i} &= 2^{-L_1} + 2^{-L_2} + \sum_3^n 2^{-\log_2 n} \\ &= 2^{-\log_2 n + a} + 2^{-\log_2 n - e} + \sum_1^n 2^{-\log_2 n} - 2 \times 2^{-\log_2 n} \\ &= \frac{2^a}{n} + \frac{2^{-e}}{n} + 1 - \frac{2}{n}.\end{aligned}$$

La desigualdad de Kraft-MacMillan requiere que

$$\frac{2^a}{n} + \frac{2^{-e}}{n} + 1 - \frac{2}{n} \leq 1, \qquad \text{ó} \qquad \frac{2^a}{n} + \frac{2^{-e}}{n} - \frac{2}{n} \leq 0,$$

ó $2^{-e} \leq 2 - 2^a$, implicando $-e \leq \log_2(2 - 2^a)$ ó $e \geq -\log_2(2 - 2^a)$.

La desigualdad anterior implica $a \leq 1$ (de lo contrario, $2 - 2^a$ es negativo); pero a también es positivo (ya que hemos supuesto la compresión del primer símbolo). El rango de valores posibles para a es —por lo tanto— (0,1]; y en este intervalo, e es mayor que a, lo que demuestra la afirmación anterior. (Es fácil ver que $a = 1 \to e \geq -\log_2 0 = \infty$ y $a = 0{,}1 \to e \geq -\log_2\left(2 - 2^{0{,}1}\right) \approx 0{,}10745$.)

Se puede demostrar que esto es sólo un caso especial de un resultado general que dice: Si se tiene un alfabeto de n símbolos y se comprimen algunos de ellos por un factor determinado, entonces los demás deberán ser expandidos por un factor mayor.

2.7. La codificación de Shannon-Fano

La codificación de Shannon-Fano, denominada así por Claude Shannon y Robert Fano, fue el primer algoritmo para construir un conjunto de los mejores códigos de longitud variable. Comienza con un conjunto de n símbolos con probabilidades conocidas (o frecuencias) de ocurrencia. En primer lugar se disponen los símbolos en orden decreciente, según sus probabilidades. El conjunto de símbolos se divide en dos subconjuntos que tienen las mismas (o casi las mismas) probabilidades. A todos los símbolos de un subconjunto se les asignan códigos que comienzan con un 0, mientras que los códigos de los símbolos del otro subconjunto comienzan con un 1. Después, cada subconjunto se divide en dos partes de probabilidades casi iguales, y el segundo bit de todos los códigos se determina de forma similar. Esto se repite de forma recursiva con cada división. Cuando el subconjunto contiene sólo dos símbolos, se añade un bit diferenciador a cada uno. El proceso continúa hasta que no queden más subconjuntos. La Tabla 2.14 ilustra el algoritmo de Shannon–Fano para un alfabeto de siete símbolos. Nótese que no se muestran los símbolos mismos, sólo sus probabilidades.

El primer paso se divide el conjunto de siete símbolos en dos subgrupos: uno con dos símbolos y una probabilidad total de 0,45, y el otro con los otros cinco símbolos y una probabilidad total de 0,55. A los dos símbolos del primer subconjunto se les asignan códigos que empieza con 1, por lo que sus códigos finales son 11 y 10. El segundo subconjunto está dividido —en el segundo paso— en dos símbolos (con una probabilidad total de 0,30 y códigos que comienzan con 01) y tres símbolos (con una probabilidad total de 0,25 y códigos que comienzan con 00). El paso tres divide los últimos tres símbolos en dos grupos: de 1 elemento (con probabilidad 0,10 y código 001) y de 2 elementos (con probabilidad total 0,15 y códigos que comienzan con 000).

El tamaño medio de este código es:

$$0{,}25 \times 2 + 0{,}20 \times 2 + 0{,}15 \times 3 + 0{,}15 \times 3 + 0{,}10 \times 3 + 0{,}10 \times 4 + 0{,}05 \times 4 = 2{,}7\, {}^{bits}/_{\text{símbolo}}.$$

	Probabilidad	Pasos					Final
1.	0,25	1	1				:11
2.	0,20	1	0				:10
3.	0,15	0		1	1		:011
4.	0,15	0		1	0		:010
5.	0,10	0		0		1	:001
6.	0,10	0		0		0	1 :0001
7.	0,05	0		0		0	0 :0000

Tabla 2.14: Ejemplo de Shannon–Fano.

	Probabilidad	Pasos			Final
1.	0,25	1	1		:11
2.	0,25	1	0		:10
3.	0,125	0	1	1	:011
4.	0,125	0	1	0	:010
5.	0,125	0	0	1	:001
6.	0,125	0	0	0	:000

Tabla 2.15: Ejemplo de Shannon–Fano balanceado.

Éste es un buen resultado, porque la entropía (el número mínimo de bits necesarios, en promedio, para representar cada símbolo) es:

$$-(0{,}25 \cdot log_2 0{,}25 + 0{,}20 \cdot \log_2 0{,}20 + 0{,}15 \cdot \log_2 0{,}15 + 0{,}15 \cdot \log_2 0{,}15 \\ +0{,}10 \cdot \log_2 0{,}10 + 0{,}10 \cdot \log_2 0{,}10 + 0{,}05 \cdot \log_2 0{,}05) \approx 2{,}67.$$

◇ **Ejercicio 2.11 (sol. en pág. 1055):** Repítase el cálculo, pero haciendo la primera división entre el tercer y cuarto símbolos. Calcúlese el tamaño medio del código y demuéstrese que es mayor que $2{,}67\,{}^{\text{bits}}/_{símbolo}$.

El código de la tabla que figura en la respuesta del Ejercicio 2.11 tiene un tamaño mayor que el medio, porque las divisiones —en este caso— no eran tan buenas como las de la Tabla 2.14. Esto sugiere que el método de Shannon–Fano produce un mejor código cuando las divisiones son mejores, i.e., cuando los dos subconjuntos de cada división tienen probabilidades totales muy cercanas. Llevar este argumento a su límite sugiere que con divisiones perfectas se obtiene el mejor código. La Tabla 2.15 ilustra este caso. Los dos subconjuntos de cada división tienen idénticas probabilidades totales, produciendo un código con el tamaño medio mínimo (redundancia cero). Su tamaño medio es:

$$0{,}25 \times 2 + 0{,}25 \times 2 + 0{,}125 \times 3 + 0{,}125 \times 3 + 0{,}125 \times 3 + 0{,}125 \times 3 = 2{,}5\,{}^{bits}/_{símbolo},$$

que es idéntico a su entropía. Esto significa que es el tamaño medio mínimo teórico.

La conclusión es que este método produce los mejores resultados cuando los símbolos tienen probabilidades de ocurrencia que son potencias negativas de 2.

◇ **Ejercicio 2.12 (sol. en pág. 1056):** Calcúlese la entropía de los códigos de la Tabla 2.15.

El método de Shannon–Fano es fácil de implementar, pero el código que produce no es —en general— tan bueno como el producido por el método de Huffman (Sección 2.8).

2.8. Codificación Huffman

La codificación Huffman es un método muy valorado para la compresión de datos. Sirve como base para varios programas populares que se ejecutan en diversas plataformas. Algunos de ellos, utilizan sólo el método de Huffman, mientras que en otros, forma parte de un proceso de compresión de varios pasos. El método de Huffman [Huffman 52] es —en cierto modo— similar al método de Shannon–Fano. En general, produce mejores códigos; al igual que el método de Shannon–Fano, obtiene el mejor código cuando las probabilidades de los símbolos son potencias negativas de 2. La principal diferencia entre los dos métodos es, que Shannon–Fano construye sus códigos de arriba abajo (desde los bits de más a la izquierda a los situados más a la derecha), mientras que Huffman lo hace mediante un árbol de código —desde lo más profundo del mismo, hasta arriba (genera los códigos de derecha a izquierda)—. Desde su desarrollo —en 1952, por D. Huffman—, este método ha sido objeto de una intensa investigación en el ámbito de la compresión de datos.

El algoritmo se inicia mediante la creación de una lista de todos los símbolos alfabéticos en orden descendente de sus probabilidades. A continuación, se construye un árbol, con un símbolo en cada hoja, de abajo arriba. Esto se hace por etapas, donde a cada paso se seleccionan los dos símbolos con las probabilidades más pequeñas; se añaden a la parte superior del árbol parcial, se eliminan de la lista y se reemplazan por un símbolo auxiliar que representa a los dos símbolos originales. Cuando la lista se reduce a sólo un símbolo auxiliar (representando todo el alfabeto), el árbol está completo. Posteriormente, se recorre el árbol para determinar los códigos de los símbolos.

Este proceso se ilustra mejor con un ejemplo. Dados cinco símbolos $a_1, \ldots, a_5$, con probabilidades de aparición 0,4, 0,2, 0,2, 0,1 y 0,1, respectivamente, podemos agruparlos como se muestra en la Figura 2.16a, siguiendo este orden:

1. a_4 se combina con a_5 y ambas se sustituyen por el símbolo a_{45}, cuya probabilidad es de 0,2.

2. Ahora quedan cuatro símbolos disponibles: a_1—con una probabilidad de 0,4— y a_2, a_3, a_{45} —con probabilidades de 0,2 cada uno—. Se seleccionan de forma arbitraria a_3 y a_{45}, combinándolos y reemplazándolos por el símbolo auxiliar a_{345} —cuya probabilidad es de 0,4—.

3. En este momento tenemos tres símbolos, a_1, a_2 y a_{345} —con probabilidades 0,4, 0,2 y 0,4, respectivamente—. Se seleccionan de forma arbitraria a_2 y a_{345}, combinándolos y reemplazándolos por el símbolo auxiliar a_{2345} —cuya probabilidad es de 0,6—.

4. Finalmente, combinamos los dos símbolos restantes, a_1 y a_{2345}, sustituyéndolos por a_{12345} —con una probabilidad de 1,0—.

El árbol —ahora— está completo. Se muestra en la Figura 2.16a: la raíz principal, en la parte superior; y según se desciende, sus ramas —izquierda y derecha (que pueden ser la raíz de otro subárbol o una hoja)—; cada elemento o combinación de elementos está acompañado por su probabilidad de aparición; en cada rama del árbol se indica el código que se le ha asignado (cada par determina dos posibles caminos que se diferencian de forma única marcando uno de ellos con un 0 y el otro con un 1). Esto se traduce en los códigos 0, 10, 111, 1101 y 1100, para a_1, a_2, a_3, a_4 y a_5, respectivamente. La asignación de los bits a las ramas es arbitraria.

El tamaño medio de este código es:

$$0{,}4 \times 1 + 0{,}2 \times 2 + 0{,}2 \times 3 + 0{,}1 \times 4 + 0{,}1 \times 4 = 2{,}2\, {}^{\text{bits}}/_{símbolo},$$

pero más importante aún: el código Huffman no es único. Algunos de los pasos anteriores han sido elegidos arbitrariamente, ya que había más de dos símbolos con probabilidades pequeñas. La Figura 2.16b: muestra cómo los mismos cinco símbolos pueden combinarse para obtener otro código Huffman

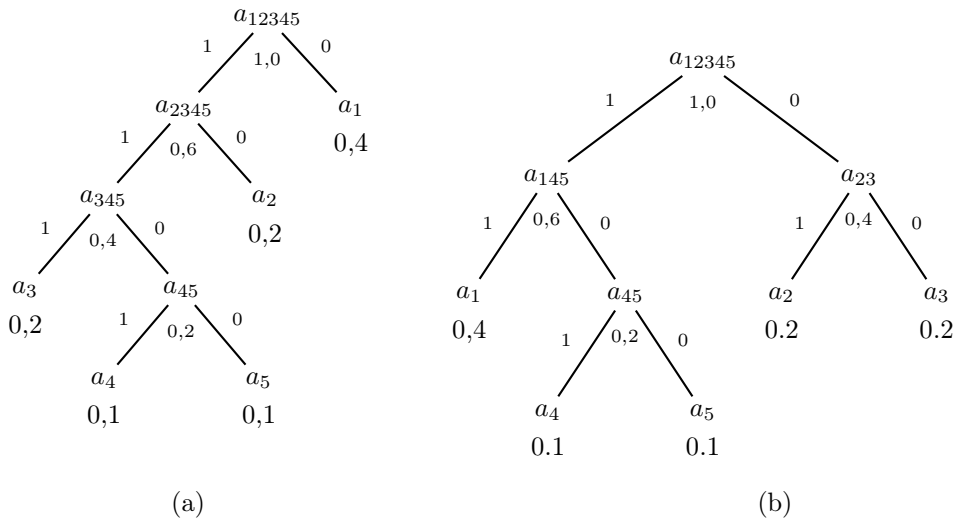


Figura 2.16: Códigos de Huffman.

diferente —11, 01, 00, 101 y 100, para a_1, a_2, a_3, a_4 y a_5, respectivamente—. El tamaño medio de este código es:

$$0{,}4 \times 2 + 0{,}2 \times 2 + 0{,}2 \times 2 + 0{,}1 \times 3 + 0{,}1 \times 3 = 2{,}2\,{}^{\text{bits}}/_{símbolo},$$

la misma que el código anterior.

◇ **Ejercicio 2.13 (sol. en pág. 1056):** Dados los ocho símbolos A, B, C, D, E, F, G y H con probabilidades respectivas de $^1/_{30}$, $^1/_{30}$, $^1/_{30}$, $^2/_{30}$, $^3/_{30}$, $^5/_{30}$, $^5/_{30}$ y $^{12}/_{30}$, dibújense tres árboles Huffman diferentes con alturas de 5 y 6 símbolos y calcúlese el tamaño medio del código de cada árbol.

◇ **Ejercicio 2.14 (sol. en pág. 1056):** La Figura Sol.7 muestra otro árbol de Huffman —con una altura de 4— para los ocho símbolos introducidos en el Ejercicio 2.13. Explíquese por qué este árbol es erróneo.

De todo esto se infiere que las decisiones arbitrarias en la construcción del árbol de Huffman afectan a los códigos individuales pero no al tamaño medio del código. Aún así, tenemos que responder la pregunta obvia: ¿cuál de todos los diferentes códigos Huffman —para un determinado conjunto de símbolos— es el mejor? La respuesta, aunque no obvia, es simple: el mejor código, es el que tiene la varianza más pequeña. La varianza de un código mide cuánto se desvían los tamaños de los códigos individuales del tamaño medio (véase la página 4.21.3 para la definición de la varianza). La varianza del código 2.16a (Figura 2.16a) es:

$$\begin{aligned} 0{,}4 \cdot (1 - 2{,}2)^2 + 0{,}2 \cdot (2 - 2{,}2)^2 + 0{,}2 \cdot (3 - 2{,}2)^2 \\ +0{,}1 \cdot (4 - 2{,}2)^2 + 0{,}1 \cdot (4 - 2{,}2)^2 = 1{,}36, \end{aligned}$$

mientras que la varianza del código 2.16b (Figura 2.16b) es:

$$\begin{aligned} 0{,}4 \cdot (2 - 2{,}2)^2 + 0{,}2 \cdot (2 - 2{,}2)^2 + 0{,}2 \cdot (2 - 2{,}2)^2 \\ +0{,}1 \cdot (3 - 2{,}2)^2 + 0{,}1 \cdot (3 - 2{,}2)^2 = 0{,}16. \end{aligned}$$

El código 2.16b es, por lo tanto, preferible al 2.16a (véase más abajo). Una mirada cuidadosa a los dos árboles muestra cómo seleccionar el que queremos. En el árbol de la Figura 2.16a, el símbolo a_{45} se combina con a_3, mientras que en el árbol de la 2.16b se combina con a_1. La regla es: cuando hay más

P_i	Código	$-\log_2 P_i$	$[-\log_2 P_i]$
,01	000	6,644	7
∗,30	001	1,737	2
,34	01	1,556	2
,35	1	1,515	2

Tabla 2.17: Ejemplo de código de Huffman.

de dos nodos con la probabilidad más pequeña, se seleccionan —para fusionarlos— el nodo más bajo y el más alto del árbol. Esto combina los símbolos de menor probabilidad con los de mayor probabilidad, lo que reduce la varianza total del código.

Si el codificador simplemente escribe la secuencia comprimida en un archivo, la varianza del código no produce ninguna diferencia. Un código Huffman de varianza pequeña es preferible sólo en aquellos casos donde el codificador *transmite* la secuencia comprimida, ya que se está generando sobre una línea de comunicaciones. En tal caso, un código con una varianza grande provoca que el codificador genere bits a una velocidad que varía continuamente. Dado que los bits tienen que ser transmitidos a una velocidad constante, el codificador tiene que usar una memoria intermedia (búfer ó buffer). Los bits de la secuencia comprimida se introducen en el búfer a medida que se van generando y salen de él a una velocidad constante, para ser transmitidos. Es fácil ver intuitivamente que un código Huffman con varianza cero introducirá bits en el búfer a una velocidad constante, por lo que sólo necesitará un buffer pequeño. Cuanto mayor es la varianza del código, más variable es la velocidad con la que entran los bits en el búfer, obligando al codificador a utilizar un buffer mayor.

A veces encontramos la siguiente declaración en la literatura:

> Se puede demostrar que el tamaño del código Huffman de un símbolo a_i,con una probabilidad P_i es siempre menor o igual que[15] $[-\log_2 P_i]$.

A pesar de que es correcta en muchos casos, en general, esta afirmación no es cierta. Parece ser un corolario sacado por algunos autores de una interpretación erronea de la desigualdad de Kraft-MacMillan, Ecuación (2.5). El autor está en deuda con Guy Blelloch por señalar este fallo y también por el ejemplo de la Tabla 2.17.

◇ **Ejercicio 2.15 (sol. en pág. 1056):** Encuéntrese un ejemplo donde el tamaño del código Huffman de un símbolo a_i es mayor que $[-\log_2 P_i]$.

◇ **Ejercicio 2.16 (sol. en pág. 1057):** Aparentemente, el tamaño de un código también debe depender del número —n— de símbolos (el tamaño del alfabeto). Un alfabeto pequeño requiere sólo unos pocos códigos, por lo que todos pueden ser cortos; un alfabeto grande requiere muchos códigos, por lo que algunos deben ser largos. Siendo esto así, ¿cómo podemos decir que el tamaño del código de un símbolo a_i sólo depende de su probabilidad P_i?

La Figura 2.18 muestra un código Huffman para 26 letras.

Como ejercicio personal, el lector puede calcular el tamaño medio, la entropía y la varianza de este código.

◇ **Ejercicio 2.17 (sol. en pág. 1057):** Estúdiense los códigos Huffman de elementos que tienen probabilidades iguales.

El Ejercicio 2.17 muestra que los símbolos con probabilidades iguales no se pueden comprimir usando el método de Huffman. Esto es comprensible, ya que las cadenas de tales símbolos normalmente producen texto aleatorio y el texto aleatorio no puede comprimirse. Existen casos especiales en los que

[15] Los corchetes deben leerse: "la parte entera de".

cadenas de símbolos con probabilidades iguales no son aleatorias y pueden ser comprimidas. Un buen ejemplo es la cadena $a_1a_1 \ldots a_1a_2a_2 \ldots a_2a_3a_3 \ldots$ en la que —al considerar una secuencia larga— aparecen todos y cada uno de los símbolos. Esta cadena puede comprimirse con RLE, pero no con los códigos Huffman.

Observe que el método de Huffman no se puede aplicar a un alfabeto de dos símbolos. En dicho alfabeto, a un símbolo se le puede asignar el código 0 y al otro el 1. El método de Huffman no puede asignar a cualquier símbolo un código más corto que un bit, así que no se puede mejorar este código tan sencillo. Si los datos originales (la fuente) consisten en bits individuales —como en el caso de una imagen con dos niveles (monocromática)— es posible combinar varios bits (tal vez cuatro u ocho) en un nuevo símbolo, y suponer que el alfabeto consta de esos (16 ó 256) símbolos. El problema con este enfoque es, que los datos binarios originales pueden tener ciertas correlaciones estadísticas entre los bits, y algunas de estas correlaciones se pierden cuando los bits se combinan en símbolos. Cuando se digitaliza una imagen típica de dos niveles (una tabla o un diagrama) explorando línea por línea, es más probable que el píxel siguiente a uno dado sea idéntico, a que sea el opuesto. Por lo tanto, se obtiene un archivo que puede comenzar con un 0 o un 1 (cada uno tiene una probabilidad de 0,5 de ser el primer bit). Es más probable que a un cero le siga otro 0, y a un 1 otro 1.

La Figura 2.19 es una máquina de estados finitos que ilustra esta situación. Si estos bits se combinan en, digamos, grupos de ocho, los bits dentro de un grupo todavía estarán correlacionados, pero los grupos en sí no están correlacionados con las probabilidades originales del píxel. Si la secuencia de entrada contiene, e.g., los dos grupos adyacentes 00011100 y 00001110, serán codificados de manera independiente, ignorando la correlación entre el último 0 del primer grupo y el primer 0 del grupo siguiente. La selección de grandes grupos, mejora esta situación, pero incrementa el número de grupos, lo que implica la necesidad de más memoria para la tabla de códigos y más tiempo para calcular la tabla.

◇ **Ejercicio 2.18 (sol. en pág. 1057):** ¿Cómo aumenta el número de grupos cuando cada uno de ellos incrementa su tamaño en n bits (desde s hasta $s + n$)?

Una propuesta más compleja para la compresión de imágenes mediante la codificación Huffman es crear varios juegos completos de códigos Huffman. Si el tamaño del grupo es, e.g., ocho bits, entonces se generan varios conjuntos de 256 códigos. Cuando se codifica un símbolo S, se selecciona uno de los conjuntos y se codifica S usando su código en ese conjunto. La elección del conjunto depende del símbolo que precede a S.

◇ **Ejercicio 2.19 (sol. en pág. 1058):** Imagínese una imagen con píxeles de 8 bits donde la mitad de ellos tienen valor 127 y la otra mitad 128. Analícese el rendimiento de RLE en planos de bits individuales de una imagen de ese tipo y compárese con el que puede lograrse con la codificación Huffman.

2.8.1. Decodificación Huffman

Antes de comenzar la compresión de una secuencia de datos, el compresor (codificador) tiene que determinar los códigos. Lo hace basándose en las probabilidades (o frecuencias de ocurrencia) de los símbolos. Las probabilidades o frecuencias tienen que guardarse —como información adicional— con la cadena comprimida, para que cualquier descompresor Huffman (decodificador) sea capaz de descomprimirla. Esto es fácil, ya que las frecuencias son números enteros y las probabilidades se pueden escribir como números enteros a escala. Normalmente se añaden sólo unos pocos cientos de bytes a la secuencia comprimida. También se pueden escribir los propios códigos de longitud variable en la cadena, lo cual puede ser difícil, porque los códigos tienen diferentes tamaños. Así mismo, es posible escribir el árbol de Huffman junto a los datos comprimidos, pero esto puede requerir más espacio del que ocupan sólo las frecuencias.

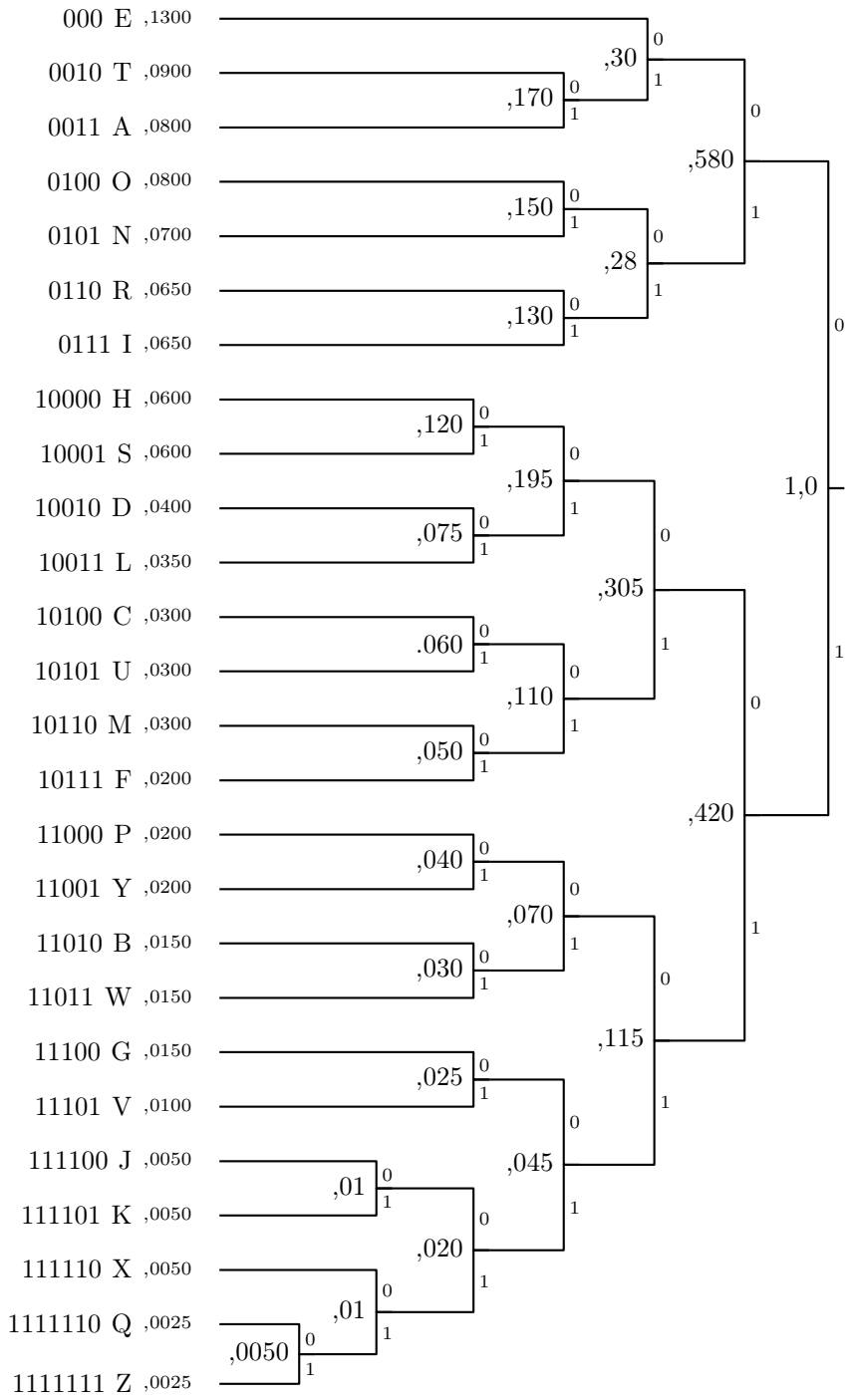


Figura 2.18: Un código Huffman para las 26 letras del alfabeto inglés.

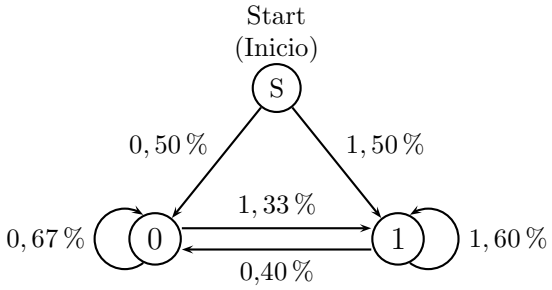


Figura 2.19: Una máquina de estados finitos.

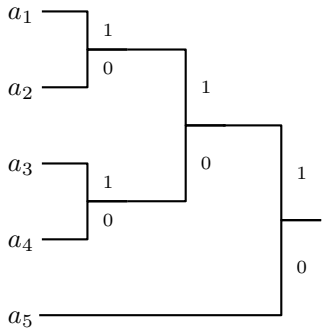


Figura 2.20: Códigos Huffman para probabilidades iguales.

En cualquier caso, el decodificador tiene que saber qué hay al comienzo de la cadena y con esa información construir el árbol de Huffman para el alfabeto. Sólo así puede leer y decodificar el resto de la secuencia. El algoritmo para decodificar es simple: Comienza en la raíz y lee el primer bit de los datos comprimidos. Si es cero, se dirige hacia la rama asociada al cero del árbol y si es uno, continúa por la otra rama. Lee el bit siguiente y se desplaza por otra rama hacia las hojas del árbol. Cuando el decodificador llega a una hoja, se encuentra el código original sin comprimir del símbolo (normalmente su código ASCII) y es ése código el generado por el decodificador. El proceso comienza de nuevo en la raíz con el bit siguiente.

A continuación, ilustramos este proceso para el alfabeto de cinco símbolos de la Figura 2.20. Los cuatro símbolos de la cadena de entrada $a_4a_2a_5a_1$ se codifican como 1001100111. El decodificador comienza en la raíz, lee el primer bit —1— y se dirige por la rama izquierda. El segundo bit —0— , le indica que debe bajar por la rama derecha; lo mismo ocurre con el tercer bit. Esto hace que el decodificador encuentre la hoja a_4, la cual emite. Una vez más, vuelve a la raíz; lee 110; sigue el camino: izquierda, izquierda y derecha, llegando a la hoja a_2 —que ofrece como salida— y así sucesivamente.

> La verdad es más extraña que la ficción, pero esto es porque la ficción está obligada a atenerse a la probabilidad; la verdad no lo está.
>
> —Anónimo.

2.8.2. Tamaño medio del código

La Figura 2.23a muestra un conjunto de cinco símbolos con sus probabilidades y un árbol de Huffman típico. El símbolo A aparece el *55 %* de las veces y se le asigna un código de 1 bit, por lo que

contribuye con $0{,}55 \cdot 1$ bits al tamaño medio del código. El símbolo E aparece sólo el *2 %* de las veces y se le asigna un código Huffman de 4 bits, por lo que contribuye con $0{,}02 \cdot 4 = 0{,}08$ bits al tamaño del código. El cálculo del tamaño medio del código es pues:

$$0{,}55 \cdot 1 + 0{,}25 \cdot 2 + 0{,}15 \cdot 3 + 0{,}03 \cdot 4 + 0{,}02 \cdot 4 = 1{,}7 \,\text{bits por símbolo.}$$

Sorprendentemente, se obtiene el mismo resultado sumando los valores de los cuatro nodos internos del árbol de códigos Huffman: $0{,}05 + 0{,}2 + 0{,}45 + 1 = 1{,}7$. Esto proporciona una forma de calcular el tamaño medio del código de un conjunto de códigos Huffman sin realizar ninguna multiplicación. Sólo hay que sumar los valores de todos los nodos internos del árbol. La Tabla 2.21 ilustra por qué ésto funciona.

$$\begin{aligned}
&\mathbf{0{,}05} = 0{,}02 + 0{,}03\\
&\mathbf{0{,}20} = \mathbf{0{,}05} + 0{,}15 = 0{,}02 + 0{,}03 + 0{,}15\\
&\mathbf{0{,}45} = \mathbf{0{,}20} + 0{,}25 = 0{,}02 + 0{,}03 + 0{,}15 + 0{,}25\\
&\mathbf{1{,}00} = \mathbf{0{,}45} + 0{,}55 = 0{,}02 + 0{,}03 + 0{,}15 + 0{,}25 + 0{,}55
\end{aligned}$$

Tabla 2.21: Composición de nodos.

(En dicha tabla se muestran en negrita los nodos internos.) La columna de la izquierda está formada por los valores de todos los nodos internos. Las columnas de la derecha muestran cómo cada nodo interno es la suma de algunos de los nodos hoja. Los valores de la columna de la izquierda suman 1,7 y en las otras columnas se puede ver que este 1,7 es la suma: de los cuatro valores 0,02, los cuatro valores 0,03, los tres valores 0,15, los dos valores 0,25 y el valor único 0,55.

$$\begin{array}{rcccccl}
\mathbf{0{,}05} & = & & & & = & 0{,}02 + 0{,}03 + \cdots\\
\mathbf{a_1} & = & \mathbf{0{,}05} & + & \ldots & = & 0{,}02 + 0{,}03 + \cdots\\
\mathbf{a_2} & = & \mathbf{a_1} & + & \ldots & = & 0{,}02 + 0{,}03 + \cdots\\
\vdots & = & & & & & \\
\mathbf{a_{d-2}} & = & \mathbf{a_{d-3}} & + & \ldots & = & 0{,}02 + 0{,}03 + \cdots\\
\mathbf{1{,}0} & = & \mathbf{a_{d-2}} & + & \ldots & = & 0{,}02 + 0{,}03 + \cdots
\end{array}$$

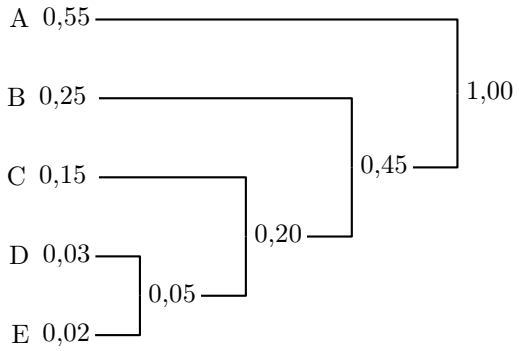
Tabla 2.22: Composición de nodos.

Este argumento se puede extender al caso general. Es fácil demostrar que en un árbol Huffman o similar (aquel donde cada nodo es la suma de sus hijos), la suma ponderada de las hojas, cuyos pesos son las distancias de las hojas a la raíz, es igual a la suma de los nodos internos. (Esta propiedad ha sido comunicada al autor por John M. Motil.)

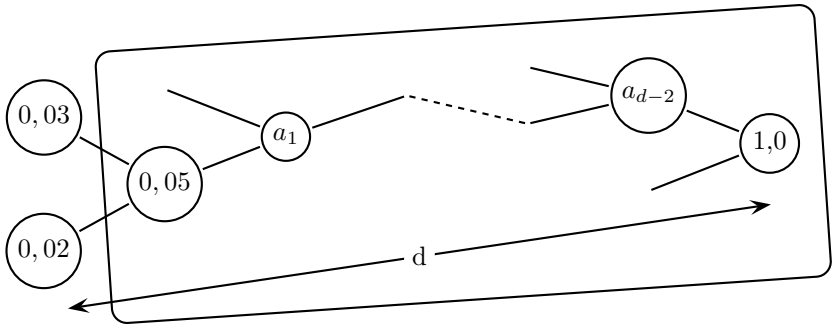
La Figura 2.23b muestra un árbol, donde se supone que las dos hojas 0,02 y 0,03 tienen códigos Huffman de d bits. En el interior del árbol, estas hojas se convierten en hijos del nodo interno 0,05, que a su vez está conectado a la raíz a través de los $d-2$ nodos internos: $a_1, \ldots, a_{d-2}$.

La Tabla 2.22 tiene d filas y muestra que los dos valores 0,02 y 0,03 se incluyen en los distintos nodos internos exactamente d veces. Sumando los valores de todos los nodos internos se produce un total que incluye las contribuciones $0{,}02 \cdot d$ y $0{,}03 \cdot d$ de las dos hojas. Dado que estas hojas son arbitrarias, es claro que esta suma incluye contribuciones similares de todas las otras hojas, por lo que dicha suma es el tamaño del código medio. Dado que esta suma es también igual a la de la columna izquierda, que es la suma de los nodos internos, es claro que la suma de los nodos internos es igual al tamaño medio del código.

Nótese que esta prueba no supone que el árbol es binario. La propiedad que se ilustra aquí es válida para cualquier árbol cuyos nodos sean la suma de sus hijos.



(a)



(b)

Figura 2.23: Árboles de código de Huffman.

2.8.3. Número de códigos

Dado que el código Huffman no es único, la pregunta natural es: ¿Cuántos códigos diferentes hay? La Figura 2.24a muestra un árbol de código Huffman para seis símbolos, con los cuales podemos responder a esta pregunta de dos maneras diferentes:

Respuesta 1. El árbol de la Figura 2.24a tiene cinco nodos interiores; en general, un árbol de código Huffman para n símbolos tiene $n - 1$ nodos interiores. De cada nodo interior salen dos aristas etiquetadas con 0 y 1. El intercambio de las dos etiquetas produce un árbol de código Huffman distinto, por lo que el número total de árboles de código Huffman diferentes es 2^{n-1} (en nuestro ejemplo, 2^5 ó 32). El árbol de la Figura 2.24b, por ejemplo, muestra el resultado de intercambiar las etiquetas de las ramas de la raíz. La Tabla 2.25a,b muestra los códigos generados por los dos árboles.

Respuesta 2. Los seis códigos de la Tabla 2.25a se pueden dividir en cuatro categorías, $00x$, $10y$, 01 y 11, donde x e y son de 1 bit cada uno. Es posible crear códigos de Huffman diferentes cambiando los dos primeros bits de cada grupo. Puesto que hay cuatro clases, esto es lo mismo que crear todas las permutaciones de cuatro objetos, lo que equivale a $4! = 24$ posibilidades. En cada una de las 24 permutaciones también es posible cambiar los valores de x e y de cuatro maneras diferentes (ya que son bits) por lo que el número total de códigos de Huffman diferentes en el ejemplo de seis símbolos

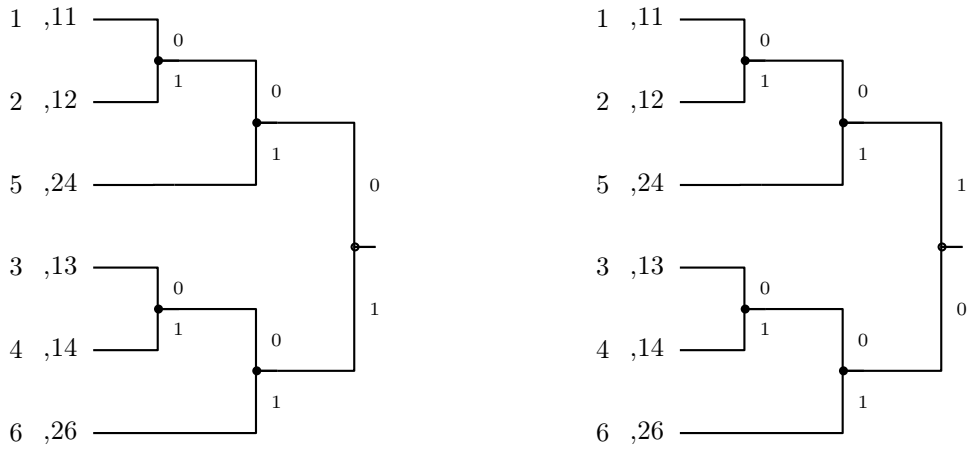


Figura 2.24: Dos árboles de código de Huffman.

es de $24 \times 4 = 96$.

Las dos respuestas son diferentes porque dicen cosas distintas. La respuesta 1 cuenta el número de árboles de código Huffman diferentes y la respuesta 2 cuenta los distintos códigos de Huffman. Resulta que nuestro ejemplo puede generar 32 árboles de código diferentes, pero sólo 94 códigos distintos —en lugar de 96—. Esto demuestra que ¡existen códigos Huffman que no pueden ser generados por el método de Huffman! La Tabla 2.25c muestra un ejemplo. Una mirada a los árboles de la Figura 2.24 debería convencer al lector de que los códigos de los símbolos 5 y 6 tienen que comenzar con bits diferentes, pero en el código de la Tabla 2.25c ambos comienzan por 1. Por ello, este código es imposible de generar reetiquetando los nodos de los árboles de la Figura 2.24.

(a)	(b)	(c)
000	100	000
001	101	001
100	000	010
101	001	011
01	11	10
11	01	11

Tabla 2.25: Códigos.

2.8.4. Códigos Huffman ternarios

El código Huffman no es único. Por otra parte, ¡no tiene por qué ser binario! El método Huffman se puede aplicar fácilmente a códigos basados en otros sistemas de numeración. La Figura 2.26a muestra un árbol de código Huffman para cinco símbolos con probabilidades 0,15, 0,15, 0,20, 0,25 y 0,25. El tamaño medio del código es:

$$2 \times 0{,}25 + 3 \times 0{,}15 + 3 \times 0{,}15 + 2 \times 0{,}20 + 2 \times 0{,}25 = 2{,}3 \text{ bits por símbolo.}$$

La Figura 2.26b muestra un árbol de código Huffman ternario para los mismos cinco símbolos. El árbol se construye seleccionando en cada paso, los tres símbolos con menor probabilidad y fusionándolos en un símbolo padre, con la probabilidad combinada. El tamaño medio del código de este árbol es:

$$2 \times 0{,}15 + 2 \times 0{,}15 + 2 \times 0{,}20 + 1 \times 0{,}25 + 1 \times 0{,}25 = 1{,}5 \text{ trits por símbolo.}$$

Nótese que los códigos ternarios utilizan los dígitos 0, 1 y 2.

◇ **Ejercicio 2.20 (sol. en pág. 1058):** Dados siete símbolos con probabilidades 0,02, 0,03, 0,04, 0,04, 0,12, 0,26 y 0,49, constrúyanse árboles de código Huffman binarios y ternarios para ellos y calcúlese el tamaño medio del código en cada caso.

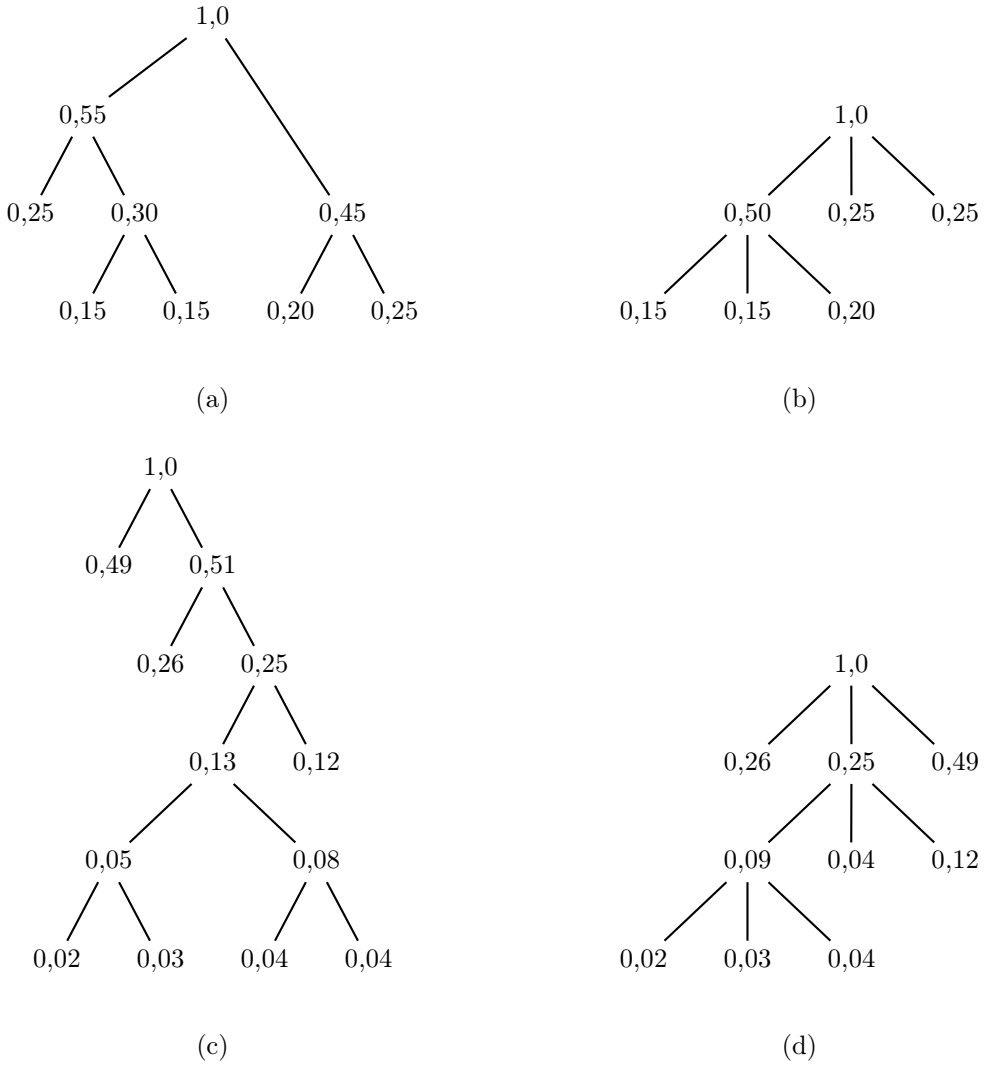


Figura 2.26: Árboles de código de Huffman binarios y ternarios.

2.8.5. La altura de un árbol de Huffman

La altura del árbol de código generado por el algoritmo de Huffman puede ser a veces importante porque la altura es también la longitud de los códigos más largos del árbol. El método *Deflate* (Sección 3.23), por ejemplo, limita la longitud de determinados códigos Huffman a sólo tres bits.

Es fácil ver que el árbol de Huffman más corto se crea cuando los símbolos tienen probabilidades iguales. Si los símbolos se representan por `A`, `B`, `C`, y así sucesivamente, entonces el algoritmo combina pares de símbolos, como `A` y `B`, `C` y `D` en el nivel más bajo y el resto del árbol se compone de nodos interiores, tal y como se muestra en la Figura 2.27a. El árbol está balanceado o casi equilibrado y su altura es $[\log_2 n]$. En el caso especial en que el número de símbolos —n— es una potencia de 2, la altura es exactamente $\log_2 n$. Con el fin de generar el árbol más alto, se necesita asignar probabilidades a los símbolos, de tal manera que en cada paso del método de Huffman se aumente la altura del árbol en 1. Recuérdese que en cada paso del algoritmo de Huffman se combinan dos símbolos. Por ello, el árbol más alto se obtiene cuando el primer paso combina dos de los n símbolos y en cada paso

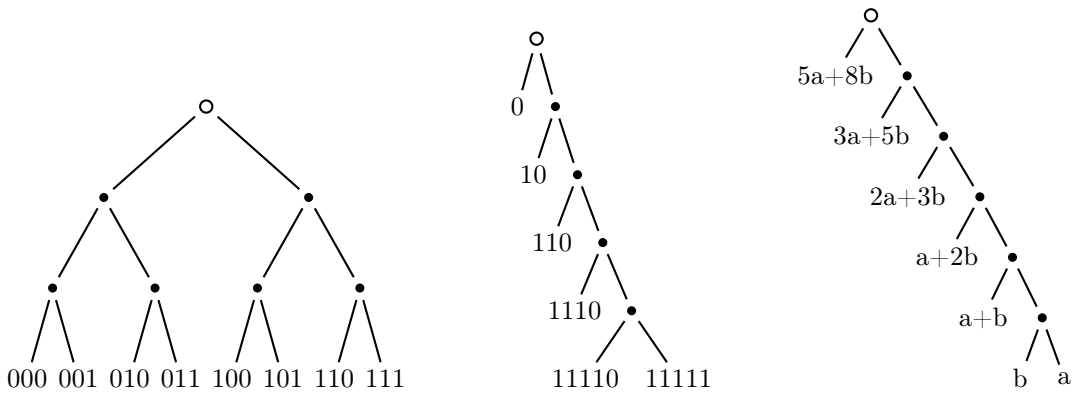


Figura 2.27: Los árboles de Huffman más cortos, y los más largos.

posterior combina el resultado de su predecesor con uno de los símbolos restantes, como se ve en la Figura 2.27b. La altura del árbol completo es, por lo tanto, $n-1$ y se conoce como árbol asimétrico, no balanceado ó desequilibrado.

Es fácil ver qué probabilidades tienen los símbolos en un árbol de este tipo. Denotamos las dos probabilidades más pequeñas por a y b. Éstas se combinan en el primer paso para formar un nodo cuya probabilidad es $a+b$. El segundo paso será fusionar este nodo con un símbolo original si uno de los símbolos tiene una probabilidad $a+b$ (o menor) y todos los símbolos restantes tienen una probabilidad mayor. Así, después del segundo paso, la raíz del árbol tiene una probabilidad $a+b+(a+b)$ y el tercer paso combina esta raíz con uno de los símbolos restantes si su probabilidad es $a+b+(a+b)$ y las probabilidades de los $n-4$ símbolos restantes son mayores. Un posible conjunto de probabilidades para los símbolos es:

$$\begin{aligned} p_1 = a, \quad p_2 = b, \\ p_3 = a+b = p_1+p_2, \quad p_4 = b+(a+b) = p_2+p_3, \\ p_5 = (a+b)+(a+2b) = p_3+p_4, \quad p_6 = (a+2b)+(2a+3b) = p_4+p_5 \end{aligned}$$

y así sucesivamente, como se indica en la Figura 2.27c. Estas probabilidades forman una secuencia de Fibonacci cuyos dos primeros elementos son a y b. Como ejemplo, seleccionamos $a=5$ y $b=2$ y generamos la secuencia de 5 números de Fibonacci 5, 2, 7, 9 y 16. Estos cinco números suman 39, por lo que dividiendolos por 39 se producen las cinco probabilidades $^5/_{39}$, $^2/_{39}$, $^7/_{39}$, $^9/_{39}$, y $^{16}/_{39}$. El árbol de Huffman generado por ellos tiene una altura máxima (que es 4).

En principio, los símbolos de un conjunto pueden tener cualquier probabilidad, pero en la práctica, las probabilidades de los símbolos del archivo de entrada se calculan contando el número de ocurrencias de cada símbolo. Imagine un archivo de texto en el que sólo aparecen los nueve símbolos: de la `A` a la `I`. Para que dicho archivo produzca el árbol de Huffman más alto —en el que los códigos tengan longitudes de 1 á 8 bits— las frecuencias de ocurrencia de los nueve símbolos tienen que formar una secuencia de Fibonacci con las probabilidades. Esto sucede cuando las frecuencias de los símbolos son: 1, 1, 2, 3, 5, 8, 13, 21 y 34 (o múltiplos enteros de los mismos). La suma de estas frecuencias es 88, por lo que nuestro archivo tiene que tener al menos esa longitud para que un símbolo se pueda expresar con códigos de Huffman de 8 bits. Del mismo modo, si queremos limitar el tamaño de los códigos de Huffman de un conjunto de n símbolos a 16 bits, es necesario contar con frecuencias de por lo

1:	000	011	9:	10100	01000
2:	001	100	10:	101010	000000
3:	010	101	11:	101011	000001
4:	011	110	12:	101100	000010
5:	10000	00100	13:	101101	000011
6:	10001	00101	14:	101110	000100
7:	10010	00110	15:	101111	000101
8:	10011	00111	16:	110000	000110
	(a)	(b)		(a)	(b)

Tabla 2.28: Ejemplo de código de Huffman canónico.

menos 4180 símbolos. Para limitar el tamaño del código a 32 bits, el tamaño mínimo de los datos es de 9 227 464 símbolos.

Si un conjunto de símbolos pasa a tener probabilidades que siguen la secuencia de Fibonacci y, por lo tanto, producen un árbol de Huffman de altura máxima, con códigos que son demasiado largos, el árbol puede ser reconfigurado (y la longitud máxima del código acortada) modificando ligeramente las probabilidades de los símbolos; así no son muy diferentes de las originales, pero no forman una secuencia de Fibonacci.

2.8.6. Códigos Huffman canónicos

El código de la Tabla 2.25c tiene una interpretación sencilla. Asigna a los cuatro primeros símbolos los códigos 0, 1, 2 y 3 (de 3 bits), y a los dos últimos símbolos, los códigos 2 y 3 (de 2 bits). Éste es un ejemplo de *código Huffman canónico*[16]. La palabra "canónico" significa que este código en particular ha sido seleccionado de entre varios (o muchos) de los posibles códigos de Huffman debido a que sus propiedades lo hacen más fácil y rápido de usar.

length	numl	first
1	0	2
2	0	4
3	4	3
4	0	5
5	5	4
6	7	0

Tabla 2.29: Algunos datos.

La Tabla 2.28 muestra un ejemplo un poco más grande de un código de Huffman canónico. Imagine un conjunto de 16 símbolos (cuyas probabilidades son irrelevantes y no se muestran) tal que a cuatro símbolos se les asignan códigos de 3 bits, a cinco símbolos códigos de 5 bits, y a los siete símbolos restantes se les asignan códigos de 6 bits. La Tabla 2.28a muestra un conjunto de códigos Huffman posibles, mientras que la Tabla 2.28b muestra un conjunto de códigos de Huffman canónicos. Es fácil ver que los siete códigos canónicos de 6 bits son simplemente los números enteros del 0 al 6 representados con 6 bits; los cinco códigos son los enteros del 4 al 8 representados con 5 bits y los cuatro códigos son los enteros del 3 al 6 representados con 3 bits. En primer lugar, mostramos cómo se generan estos códigos y luego cómo se utilizan.

La columna *length* de la Tabla 2.29 indica las longitudes de código posibles: de 1 a 6 bits. La columna *numl* muestra el número de códigos para cada longitud y la columna *first* lista el primer código de cada grupo. Ésta es la razón por la que los tres grupos de códigos comienzan con los valores: 3, 4 y 0. Para obtener las dos primeras filas necesitamos calcular las longitudes de todos los códigos de Huffman para el alfabeto considerado (véase más adelante). La tercera columna se calcula ejecutando "`first[6] := 0;`" y la siguiente iteración[17]:

[16] Elegido entre un canon o catálogo, convencional.

[17] Los corchetes que englogan la expresión `(first[l+1]+numl[l+1])/2` indican el valor entero de la expresión redon-

```
for l:=5 downto 1 do
    first[l]:= [(first[l+1]+numl[l+1])/2];
```

Ésto garantiza que todos los prefijos de 3 bits de los códigos de más de tres bits serán menores que `first[3]` (que es 3); todos los prefijos de 5 bits de los códigos de más de cinco bits serán menores que `first[5]` (que es 4) y así sucesivamente.

Ahora indicamos el uso de estos inusuales códigos . Los códigos de Huffman canónicos son útiles cuando el alfabeto es grande y en aquellos casos en que se requiere una decodificación rápida. Debido a la construcción de los códigos, al decodificador le resulta fácil identificar la longitud de un código simplemente leyendo y analizando los bits de entrada uno por uno. Una vez que se conoce la longitud, se puede encontrar el símbolo en un solo paso. El siguiente pseudocódigo muestra las reglas para la decodificación:

```
l:=1; input v;
while v<first[l]
    añadir el siguiente bit de la entrada a v; l:=l+1;
endwhile
```

A modo de ejemplo, supongamos que el siguiente código es 00110. Como los bits son de entrada y se añaden a `v`, éste va adquiriendo los valores: 0, 00 = 0, 001 = 1, 0011 = 3, 00110 = 6, mientras `l` se incrementa desde 1 hasta 5. Todos los pasos excepto el último satisfacen `v` < `first`[`l`], por lo que el último paso determina el valor de `l` (la longitud del código) en 5. El símbolo en sí mismo se obtiene restando `v` − `first`[5] = 6 − 4 = 2, por lo que es el tercer símbolo (la numeración comienza por 0) en el grupo `l` = 5 (símbolo 7 de los 16 símbolos).

Se ha mencionado que los códigos Huffman canónicos son útiles cuando el alfabeto es grande y la decodificación rápida es importante. Un ejemplo práctico es una colección de documentos archivados y comprimidos por un codificador de Huffman adaptativo *basado en palabras* (Sección 8.6.1). Para un archivo, un codificador lento es aceptable, pero el decodificador debe ser rápido. Cuando los símbolos individuales son palabras, el alfabeto puede ser enorme, por lo que es poco práctico o incluso imposible construir el árbol de código de Huffman. Sin embargo, incluso con un enorme alfabeto, el número de longitudes de código diferentes es pequeño —raramente sobrepasan los 20 bits (sólo el número de códigos de 20 bits es de aproximadamente un millón)—. Si se utilizan códigos de Huffman canónicos y la longitud máxima del código es `L`, entonces la longitud del código `l` de un símbolo se obtiene con el decodificador en como mucho `L` pasos y el propio símbolo se identifica en un solo paso más.

> Él utiliza las estadísticas como un borracho utiliza las farolas: como soporte y no como iluminación.
>
> —Andrew Lang, *Treasury of Humorous Quotations*[a]

[a]El tesoro de las citas humorísticas.

El último punto a discutir es el codificador. Para construir el código Huffman canónico, el codificador necesita saber la longitud del código de Huffman de cada símbolo. El principal problema es el gran tamaño del alfabeto, lo que puede hacer que sea poco práctico o incluso imposible construir el árbol de código de Huffman completo en la memoria. El algoritmo aquí descrito (véanse [Hirschberg y Lelewer 90] y [Sieminski 88]) resuelve este problema. Calcula los tamaños de los códigos para un alfabeto de n símbolos usando una sola matriz de tamaño $2n$. La mitad de esta matriz se utiliza como una *pila*, por lo que empezamos con una breve descripción de esta útil estructura de datos.

deada; así $first\,[4] = \left[\frac{(5+4)}{2}\right] = [4{,}5] = 5$.

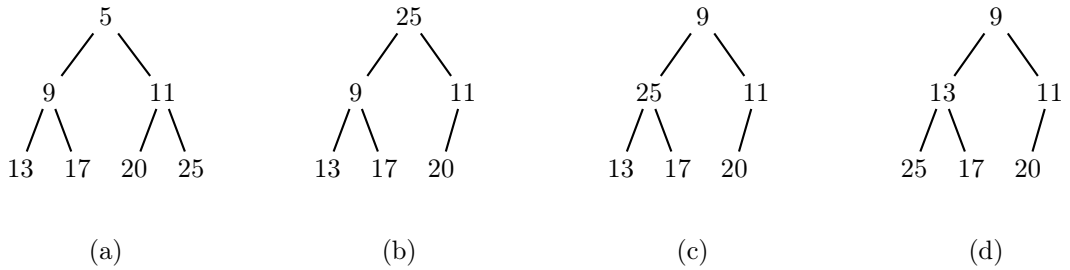


Figura 2.30: Pilas mínimas. Cribar la pila tras eliminar la raíz.

Un *árbol binario* es un árbol donde cada nodo tiene a lo sumo dos hijos (i.e., puede tener 0, 1 ó 2 hijos). Un *árbol binario completo* es un árbol binario donde cada nodo, excepto las hojas tiene exactamente dos hijos. Un *árbol binario balanceado* es un árbol binario completo en el que algunos de los nodos de la parte inferior derecha pueden faltar (en la página 132 se puede ver otra aplicación de este tipo de árboles). Una ***pila*** es un árbol binario balanceado en el que cada hoja contiene un ítem de datos y los ítems están dispuestos de tal manera que cada camino desde una hoja hasta la raíz pasa por nodos que están en orden, ya sea creciente (una pila máxima) o decreciente (una pila mínima). La Figura 2.30a muestra un ejemplo de pila mínima.

Una operación común en una pila es eliminar la raíz y reorganizar el resto de los nodos para obtener de nuevo una pila. A ésto se le llama *cribar*[18] la pila. Las cuatro partes de la Figura 2.30 muestran cómo se criba una pila después de eliminar la raíz (con el ítem de datos 5). La criba comienza moviendo el nodo inferior derecha para convertirlo en la nueva raíz. Esto garantiza que la pila seguirá siendo un árbol binario balanceado. La raíz se compara con sus hijos, y si es necesario, se intercambia con uno de ellos a fin de preservar el orden de la pila. Pueden requerirse varios intercambios más para restaurar completamente dicho orden. Es fácil ver que el número máximo de intercambios es igual a la altura del árbol, que es $[\log_2 n]$.

La razón por la que una pila debe permanecer equilibrada[19] es que así es posible almacenarla en la memoria sin utilizar punteros. Se dice entonces que la pila está "alojada[20]" en un array[21]. Para alojar una pila en un array, la raíz se guarda en la primera posición del array (con índice 1); los dos hijos del nodo de índice i en el array, en las posiciones $2i$ y $2i + 1$; y el padre del nodo de índice j en la matriz se coloca en la posición $[j/2]$. En consecuencia, la pila de la Figura 2.30a se aloja en un array cuyos primeros siete elementos son los nodos 5, 9, 11, 13, 17, 20 y 25.

El algoritmo utiliza una matriz unidimensional `A` de tamaño $2n$. Las frecuencias de ocurrencia de los n símbolos se situan en la mitad superior de `A` (posiciones desde $n + 1$ hasta $2n$) y en la mitad inferior de `A` (posiciones desde 1 hasta n) se construye una pila mínima cuyos elementos son punteros a las frecuencias de los datos de la mitad superior (Figura 2.31a). A continuación, el algoritmo entra en un bucle donde en cada iteración se utiliza la pila para identificar las dos frecuencias más pequeñas y reemplazarlas por su suma. La suma se almacena en las última posición de la pila —`A[h]`— y la pila mengua en una posición (Figura 2.31b). El bucle se repite hasta que la pila se reduce a un sólo puntero (Figura 2.31c).

Vamos a ilustrar esta parte del algoritmo usando siete frecuencias. En la siguiente tabla se muestra cómo se alojan inicialmente las frecuencias y la pila en un array de tamaño 14; los punteros se muestran

[18] La palabra *sifting* se asocia a una ordenación por intercambio de parejas clave.
[19] Balanceada.
[20] Housed.
[21] Arreglo ó matriz, en este caso unidimensional.

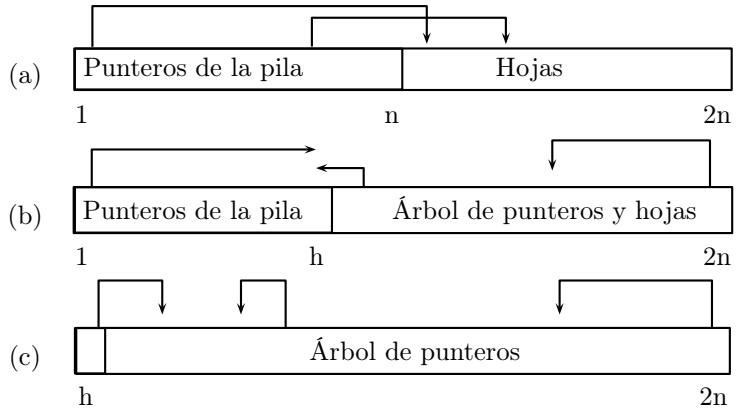


Figura 2.31: Pilas y hojas de Huffman en un array.

en cursiva y la pila está delimitada por corchetes:

$$\begin{array}{cccccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\ \hline [14 & 12 & 13 & 10 & 11 & 9 & 8] & 25 & 20 & 13 & 17 & 9 & 11 & 5 \end{array}$$

La primera iteración selecciona la frecuencia más pequeña (5), elimina la raíz de la pila (puntero 14) y deja $A\,[7]$ vacío:

$$\begin{array}{cccccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\ \hline [12 & 10 & 13 & 8 & 11 & 9] & & 25 & 20 & 13 & 17 & 9 & 11 & 5 \end{array}$$

La pila está cribada; su nueva raíz (12) apunta a la segunda frecuencia más pequeña (9) en $A\,[12]$. La suma $5 + 9$ se almacena en la posición vacía (7), y se modifican los tres elementos de la matriz —`A[1]`, `A[12]` y `A[14]`— para que apunten a esa posición:

$$\begin{array}{cccccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\ \hline [7 & 10 & 13 & 8 & 11 & 9] & 5+9 & 25 & 20 & 13 & 17 & 7 & 11 & 7 \end{array}$$

Ahora se criba la pila:

$$\begin{array}{cccccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\ \hline [13 & 10 & 7 & 8 & 11 & 9] & 14 & 25 & 20 & 13 & 17 & 7 & 11 & 7 \end{array}$$

La nueva raíz es 13, lo que implica que la frecuencia más pequeña (11) está almacenada en `A[13]`. Se elimina la raíz y la pila se reduce a sólo cinco elementos, dejando vacía la posición 6:

$$\begin{array}{cccccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\ \hline [10 & 11 & 7 & 8 & 9] & & 14 & 25 & 20 & 13 & 17 & 7 & 11 & 7 \end{array}$$

Ahora la pila está cribada. La nueva raíz es 10, mostrando que la segunda frecuencia más pequeña (13) se encuentra en $A\,[10]$. La suma $11 + 13$ se almacena en la posición vacía y se modifican `A[1]`, `A[13]`, y `A[10]` para que apunten a 6.

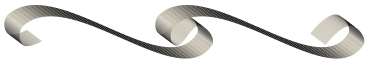
$$\begin{array}{cccccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\ \hline [6 & 11 & 7 & 8 & 9] & 11+13 & 14 & 25 & 20 & 6 & 17 & 7 & 6 & 7 \end{array}$$

◇ **Ejercicio 2.21 (sol. en pág. 1058):** Complétese este bucle.

◇ **Ejercicio 2.22 (sol. en pág. 1059):** ¿Cómo se puede obtener la longitud de los códigos de los símbolos a partir de la respuesta del ejercicio anterior?

◇ **Ejercicio 2.23 (sol. en pág. 1059):** Encuéntrense las longitudes del resto de los otros códigos.

> Ley de Considine: Cada vez que una palabra o letra pueda cambiar el significado completo de una oración, la probabilidad de que ocurra un error será directamente proporcional a la vergüenza que produce.
>
> —Bob Considine.



2.9. Codificación de Huffman adaptativa

El método de Huffman asume que el compresor conoce las frecuencias de ocurrencia de todos los símbolos del alfabeto. En la práctica, las frecuencias raramente o casi nunca se conocen de antemano. Una aproximación a este problema es hacer que el compresor lea los datos originales dos veces. La primera vez, sólo calcula las frecuencias; la segunda, comprime los datos. Entre los dos pasos, el compresor construye el árbol de Huffman. Este método se llama semiadaptativo (pág. 10) y normalmente es demasiado lento para ser práctico. El método que se utiliza en la práctica se llama codificación de Huffman adaptativa (o dinámica). Se utiliza como base del programa `compact` de UNIX. (Véase también la Sección 8.6.1 para obtener una versión basada en palabras de la codificación de Huffman adaptativa.) El método fue originalmente desarrollado por [Faller 73] y [Gallager 78] con mejoras sustanciales de [Knuth 85].

La idea principal es que el compresor y el descompresor se inicien con un árbol de Huffman vacío, y modificarlo a medida que los símbolos sean leídos y procesados (en el caso del compresor, la palabra "procesados" significa comprimidos; en el caso del descompresor, significa descomprimidos). El compresor y el descompresor deberían modificar el árbol de la misma manera, para que en cualquier momento del proceso utilicen los mismos códigos, aunque éstos pueden cambiar de un paso a otro. Decimos que el compresor y descompresor se sincronizan o que trabajan en *lockstep* (al unísono, aunque no necesariamente funcionan juntos; la compresión y la descompresión normalmente se realizan en instantes diferentes). El término *reflejo*[22] es quizás una mejor opción. El decodificador refleja las operaciones del codificador.

Inicialmente, el compresor comienza con un árbol de Huffman vacío. Todavía no se han asignado códigos a los símbolos. El primer símbolo recibido se anota en la secuencia de salida en formato comprimido. El símbolo se añade al árbol y se le asigna un código. La próxima vez que este símbolo aparezca, su código actual se escribirá en la salida y su frecuencia se incrementará en uno. Ya que se modifica el árbol, éste se examina para ver si todavía es un árbol de Huffman (con los mejores códigos). Si no, se reordena, lo que cambia los códigos (Sección 2.9.2).

El descompresor refleja los mismos pasos. Cuando lee el formato sin comprimir de un símbolo, lo añade al árbol y le asigna un código. Cuando lee un código comprimido (de tamaño variable), recorre

[22] La palabra "mirroring" se usa para indicar una duplicación simultanea de información.

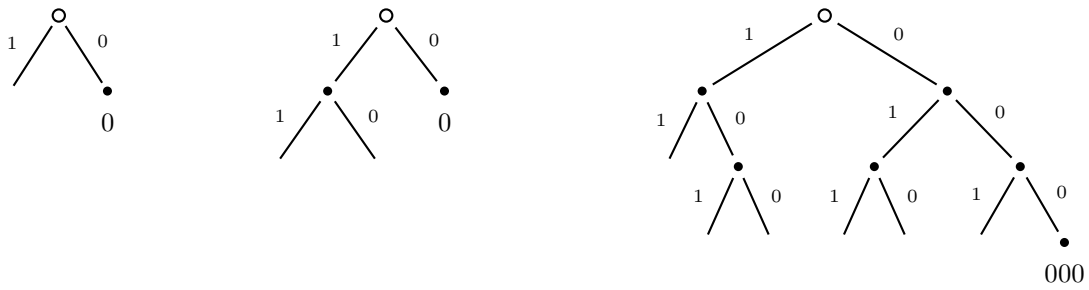


Figura 2.32: El código de escape.

el árbol actual para determinar qué símbolo le corresponde al código; incrementa la frecuencia del símbolo y vuelve a reorganizar el árbol de la misma manera que el compresor.

El único punto delicado es que el descompresor necesita saber si el elemento que acaba de entrar es un símbolo sin comprimir (normalmente un código ASCII de 8 bits, sin embargo, consúltese la Sección 2.9.1) o un código de tamaño variable. Para eliminar toda ambigüedad, cada símbolo sin comprimir va precedido por un código especial de tamaño variable, el *código de escape*[23]. Cuando el descompresor lee este código, sabe que los próximos 8 bits son el código ASCII de un símbolo que aparece en la cadena comprimida por primera vez.

El problema es, que el código de escape no puede ser ninguno de los códigos de tamaño variable utilizados para los símbolos. De todos modos, estos códigos se están modificando cada vez que se reordena el árbol, por lo que el código de escape también se debe modificar. Una manera natural de hacer esto es añadir una hoja vacía al árbol —una hoja con una frecuencia de ocurrencia cero que siempre se asigna a la rama 0 del árbol—. Dado que la hoja está en el árbol, se le asigna un código de tamaño variable. Éste es el código de escape que precede a todos los símbolos sin comprimir. A medida que se cambia el árbol, la posición de la hoja vacía —y por lo tanto, su código— cambia; pero este código de escape se utiliza siempre para identificar los símbolos sin comprimir de la secuencia de datos comprimida. La Figura 2.32 muestra cómo se mueve y cambia el código escape a medida que el árbol crece.

Este método se utilizó para comprimir/descomprimir datos en el protocolo V.32 para módems de 14 400 baudios.

> Escapar no es su plan. Tengo que enfrentarme a él. Solo.
>
> —David Prowse como Darth Vader en *Star Wars*[a] (1977)
>
> [a]La guerra de las galaxias.

2.9.1. Códigos sin comprimir

Si los símbolos a comprimir son caracteres ASCII, simplemente se les pueden asignar sus códigos ASCII sin comprimir. En el caso general, cuando podemos encontrar cualquier símbolo, se pueden asignar códigos sin comprimir de dos tamaños diferentes con un sencillo método. A continuación se ofrece un ejemplo para el caso $n = 24$: A los primeros 16 símbolos se les puede asignar, como códigos, los números del 0 al 15. Éstos sólo requieren 4 bits, pero los codificamos con 5 bits. A los símbolos del 17 al 24 se les pueden asignar los números: $17 - 16 - 1 = 0$, $18 - 16 - 1 = 1$, ..., $25 - 16 - 1 = 7$, representados con 4 bits. Esto es, los dieciseis códigos de 5 bits:00000, 00001, ..., 01111, seguidos por los ocho códigos de 4 bits: 0000, 0001, ..., 0111.

[23]Puede ser cualquier código que se diferencie de forma inequívoca de los datos.

En general, se adopta un alfabeto que consta de los n símbolos: $a_1, a_2, \ldots, a_n$. Se seleccionan enteros m y r tales que $2^m \leq n < 2^{m+1}$ y $r = n - 2^m$. Los primeros 2^m símbolos se codifican como números desde 0 hasta $2^m - 1$ representados con $(m + 1)$ bits. Los símbolos restantes se codifican como números de m bits, de tal manera que el código de a_k es $k - 2^m - 1$. Este código también se llama *código binario por etapas*[24] (en la Tabla 3.31 se puede ver una aplicación de dichos códigos).

2.9.2. Modificación del Árbol

La idea principal es comprobar el árbol cada vez que llega un símbolo. Si el árbol ya no es un árbol de Huffman, debe actualizarse. Un vistazo a la Figura 2.33a muestra lo que significa que un árbol binario sea de Huffman. El árbol contiene cinco símbolos: `A`, `B`, `C`, `D` y `E`; se muestra con los símbolos y sus frecuencias (entre paréntesis), después de haber introducido y procesado 16 elementos. La propiedad que lo convierte en un árbol de Huffman consiste en que si lo exploramos nivel por nivel de izquierda a derecha y desde abajo (las hojas) hasta arriba (la raíz), las frecuencias están ordenadas en orden creciente. Por lo tanto, el nodo inferior izquierdo (`A`) tiene la frecuencia más baja, y el nodo superior derecho (la raíz) tiene la mayor frecuencia. Esto se denomina *propiedad de los hermanos*[25].

◇ **Ejercicio 2.24 (sol. en pág. 1059):** ¿Por qué este criterio hace que un árbol sea de Huffman?

Lo siguiente es un resumen de las operaciones necesarias para actualizar el árbol. El ciclo se inicia en el nodo actual (el que corresponde sólo al símbolo de entrada). Éste es una hoja que denotamos por X, con una frecuencia de ocurrencia de F. Cada iteración del bucle consta de tres pasos:

1. Comparar X con sus sucesores en el árbol (de izquierda a derecha y de abajo arriba). Si el sucesor inmediato tiene una frecuencia de $F + 1$ o superior, los nodos están aún ordenados y no hay necesidad de cambiar nada. De lo contrario, algunos sucesores de X tienen frecuencias menores o iguales que F. En este caso, X debe intercambiarse con el último nodo de dicho grupo (salvo que X no debe ser intercambiado con su padre).

2. Incrementar la frecuencia de X de F a $F + 1$ e incrementar las frecuencias de todos sus padres.

3. Si X es la raíz, el bucle se detiene; de lo contrario, el bucle se repite con el padre del nodo X.

La Figura 2.33b muestra el árbol después de incrementar la frecuencia del nodo `A` desde 1 hasta 2. Es fácil seguir las tres reglas anteriores para ver cómo el incremento de la frecuencia de `A` produce un incremento de las frecuencias de todos sus padres. En este sencillo caso, no se han necesitado cambios entre nodos, porque la frecuencia de `A` no es superior a la frecuencia de su sucesor inmediato: `B`. La Figura 2.33c muestra lo que ocurre cuando la frecuencia de `A` se incrementa de nuevo desde 2 hasta 3. Los tres nodos que le siguen —`B`, `C` y `D`— tienen una frecuencia de 2, por lo que `A` se intercambia con el último de ellos: `D`. Las frecuencias de los nuevos padres de `A`, se incrementan y cada uno se compara con su sucesor; pero no son necesarios más cambios.

La Figura 2.33d muestra el árbol después de incrementar la frecuencia de `A` hasta 4. Una vez que determinamos que `A` es el nodo actual, su frecuencia (que aún es 3) se compara con la de su sucesor (4) y la decisión no produce ningún cambio. Se incrementa la frecuencia de `A` y a continuación las de sus padres.

En la Figura 2.33e, `A` es de nuevo el nodo actual. Su frecuencia (4) es igual que la de su sucesor, por lo tanto deben intercambiarse. Esto se muestra en la Figura 2.33f, donde la frecuencia de `A` es 5. La siguiente iteración del bucle examina el padre de `A`, con una frecuencia de 10. Debe ser intercambiado con su sucesor `E` (con una frecuencia de 9), lo que lleva al árbol final de la Figura 2.33g .

[24] Phased-in binary code.
[25] The sibling property.

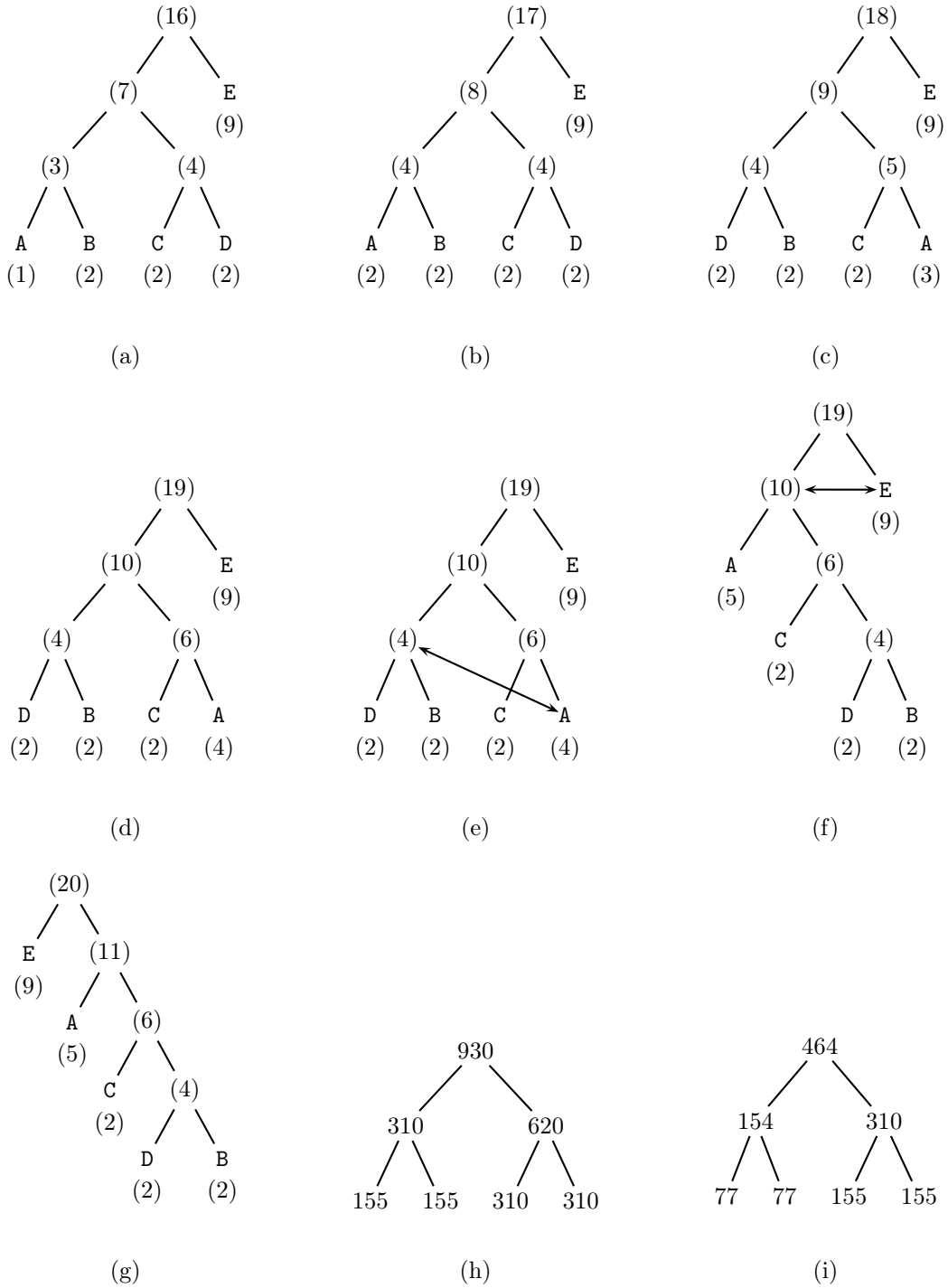


Figura 2.33: Actualización de un árbol de Huffman.

2.9.3. Desbordamiento del contador

El recuento de la frecuencia se acumula en el árbol de Huffman en los campos de tamaño fijo y éstos pueden sufrir desbordamiento. Un campo de 16 bits sin signo puede alojar un número cuyo valor máximo es $2^{16} - 1 = 65\,535$. Una solución sencilla para el problema de desbordamiento del contador es mirar el campo contador de la raíz cada vez que se incrementa y cuando alcanza su valor máximo, *reescalar* todos los contadores de frecuencias dividiéndolos por 2 (división entera). En la práctica, esto se hace dividiendo los campos contador de las hojas, y a continuación, actualizando la cuenta de los nodos interiores. Cada nodo interior recibe la suma de los contadores de sus hijos. El problema es que son números enteros y la división entera reduce la precisión. Esto puede cambiar un árbol de Huffman en uno que no satisface la propiedad de los hermanos.

En la Figura 2.33h se muestra un sencillo ejemplo. Después de reducir a la mitad los contadores de las hojas, los tres nodos interiores se actualizan como en la Figura 2.33i. Sin embargo, el último árbol ya no es de Huffman debido a que los valores de los contadores ya no están en el orden establecido. La solución es reconstruir el árbol cada vez que se hace un reescalado de los contadores, lo que no sucede muy a menudo. Un programa de compresión Huffman de datos destinado al uso general, por tanto, debe tener contadores de gran capacidad que no se desborden muy a menudo. Un contador de 4 bytes se desborda al llegar a $2^{32} - 1 \approx 4{,}3 \times 10^9$.

Nótese que después de reescalar los contadores, los nuevos símbolos leídos y comprimidos tienen más efecto en la cuenta que los símbolos antiguos (aquellos contados antes del reescalado). Ésto resulta ser fortuíto, ya que se sabe por experiencia que la probabilidad de aparición de un símbolo depende más de los elementos inmediatamente anteriores a él que de los que han aparecido en el pasado distante.

2.9.4. Desbordamiento (overflow) del código

Un problema aún más grave es el desbordamiento del código. Ésto puede ocurrir cuando se agregan muchos símbolos al árbol y éste crece mucho en altura. Los propios códigos no se almacenan en el árbol, ya que cambian todo el tiempo, y el compresor tiene que descifrar el código de cada símbolo `X` en el instante que llega. He aquí los detalles del proceso:

1. El codificador tiene que localizar el símbolo `X` en el árbol. El árbol se implementa como un array de estructuras —en cada nodo uno—, y el array se busca de forma lineal.

2. Si no encuentra `X`, se emite el código de escape, seguido por el código sin comprimir de `X`; se añade `X` al árbol.

3. Si encuentra `X`, el compresor mueve el nodo `X` hasta la raíz, construyendo el código bit a bit, a medida que avanza. Cada vez que pasa de un hijo izquierdo a un padre, se añade un bit "1" al código; al pasar de un hijo derecho a un padre se agrega un bit "0" al código (o viceversa, pero de forma coherente, ya que se refleja en el decodificador). Dichos bits tienen que ser acumulados en algún lugar, ya que tienen que emitirse en el *orden inverso* al que se crearon. A medida que el árbol crece en altura, los códigos se hacen más largos. Si se acumulan en un entero de 16 bits, entonces los códigos de más de 16 bits podrían causar un funcionamiento erróneo.

Una solución al problema del desbordamiento del código es acumular los bits de un código en una lista enlazada, en la cual se pueden crear los nuevos nodos, limitados en número sólo por la cantidad de memoria disponible; éste es un método general, pero lento. Otra solución es acumular los códigos en una variable entera de gran capacidad (tal vez de 50 bits) e indicar en la documentación, que una de las limitaciones del programa es que el tamaño máximo del código es de 50 bits.

Afortunadamente, este problema no afecta al proceso de decodificación. El decodificador lee el código comprimido bit a bit y utiliza cada uno de ellos para dar un paso por el árbol hacia abajo —a la izquierda o a la derecha— hasta que alcanza un nodo hoja. Si la hoja es el código de escape, el

N	Cont	Cod	N	Cont	Cod	N	Cont	Cod	N	Cont	Cod
a_1	0	0	a_2	1	0	a_2	1	0	a_4	2	0
a_2	0	10	a_1	0	10	a_4	1	10	a_2	1	10
a_3	0	110	a_3	0	110	a_3	0	110	a_3	0	110
a_4	0	111	a_4	0	111	a_1	0	111	a_1	0	111
	(a)			(b)			(c)			(d)	

Figura 2.34: Cuatro pasos en una variante de Huffman.

decodificador lee el código sin comprimir del símbolo situado fuera de la cadena comprimida (y añade el símbolo al árbol); de lo contrario, el código sin comprimir se encuentra en el nodo hoja.

◇ **Ejercicio 2.25 (sol. en pág. 1059):** Dada la cadena de 11 símbolos: $sir_{\sqcup}sid_{\sqcup}is_{\sqcup}$, aplicar el método de Huffman adaptativo a la misma. Para cada símbolo de entrada, muéstrese la salida, el árbol después de añadir el símbolo, el árbol después de haber sido reordenado (si es necesario) y la lista de nodos recorridos de izquierda a derecha y de abajo arriba.

2.9.5. Una variante

Esta variante del método de Huffman adaptativo es más sencilla, pero menos eficiente. La idea es calcular un conjunto de n códigos de tamaño variable basados en la igualdad de probabilidades, para asignarlos a n símbolos aleatorios, y cambiar las asignaciones "sobre la marcha", a medida que los símbolos se van leyendo y son comprimidos. El método no es eficiente porque los códigos no se basan en las probabilidades reales de los símbolos de la secuencia de datos de entrada. Sin embargo, es más sencillo de implementar y también más rápido que el método adaptativo descrito anteriormente, porque para actualizar las frecuencias de los símbolos, tiene que intercambiar las filas de una tabla, en lugar de actualizar un árbol.

La estructura de datos principal es una tabla de $n \times 3$ en la que las tres columnas almacenan los nombres de los n símbolos —N—, sus frecuencias de aparición en ese instante —Cont— y sus códigos —Cod—. La tabla se mantiene siempre ordenada por la segunda columna. Cuando la frecuencia de los contadores de la segunda columna cambian, se intercambian las filas; pero sólo se mueven las columnas 1 y 2. Los códigos de la columna 3 nunca cambian. La Figura 2.34 muestra un ejemplo de cuatro símbolos y el comportamiento del método cuando se comprime la cadena a_2, a_4, a_4.

La Figura 2.34a muestra el estado inicial. Después de leer el primer símbolo —a_2— se incrementa su contador, y como ahora es el mayor número, se intercambian las filas 1 y 2 (Figura 2.34b). Después de leer el segundo símbolo —a_4— se incrementa su contador y se intercambian las filas 2 y 4 (Figura 2.34c). Finalmente, después de leer el último símbolo —a_4— su contador es el mayor, por lo que se intercambian las filas 1 y 2 (Figura 2.34d).

El único punto que puede causar un problema con este método es el desbordamiento de los campos contador. Si un campo es de k bits, su valor máximo es: $2^k - 1$, por lo que se desbordará cuando se incremente 2^k veces. Ésto puede ocurrir si el tamaño de la secuencia de entrada no se conoce de antemano, que es lo habitual. Afortunadamente, en realidad no es necesario conocer el valor de los contadores, sólo se necesita tenerlos ordenados, lo que hace que este problema sea fácil de resolver.

Una solución es contar los símbolos de la entrada, y después de introducir y comprimir $2^k - 1$ símbolos, dividir (división entera) todos los campos contador por 2 (o desplazarlos una posición a la derecha, si resulta más fácil).

Otra solución similar es comprobar cada campo contador cada vez que se incremente, y si se ha alcanzado su valor máximo (si se compone de todo unos), realizar la división entera de todos los

campos contador por 2, como se mencionó anteriormente. Este enfoque requiere menos divisiones, pero pruebas más complejas.

Naturalmente, cualquier solución que se adopte deberá ser utilizada tanto por el compresor como por el descompresor.

2.9.6. Método de Vitter

Una mejora del algoritmo original, debida a [Vitter 87], que también incluye amplios análisis, se basa en las siguientes ideas clave:

1. Se debe utilizar un esquema diferente para numerar los nodos en el árbol de Huffman dinámico. Se llama *numeración implícita* y numera los nodos de abajo arriba, y en cada nivel, de izquierda a derecha.

2. El árbol de Huffman debe actualizarse de tal manera que siempre se cumpla lo siguiente: Para cada peso w, todas las hojas de peso w preceden (en el sentido de la numeración implícita) a todos los nodos internos del mismo peso. Éste es un *invariante*.

Estas ideas producen los siguientes beneficios:

1. En el algoritmo original, es posible que un reordenamiento del árbol haga necesario mover un nodo hacia abajo un nivel. En la versión mejorada, ésto no sucede.

2. Cada vez que el árbol de Huffman se actualiza con el algoritmo original, algunos nodos deben moverse hacia arriba. En la versión mejorada, no es necesario desplazar hacia arriba más de un nodo.

3. El árbol de Huffman en la versión mejorada minimiza la suma de las distancias desde la raíz hasta las hojas y también tiene la altura mínima.

Una estructura de datos especial, llamada *árbol flotante*, se propone facilitar el mantenimiento del invariante necesario. Se puede demostrar que esta versión funciona mucho mejor que el algoritmo original. En concreto, si un método de Huffman de dos pasos comprime un archivo de entrada de n símbolos a S bits, entonces el algoritmo Huffman adaptativo original puede comprimirlo a un máximo de $2S + n$ bits, mientras que la versión mejorada puede comprimirlo por debajo de $S + n$ bits; ¡una diferencia significativa! Téngase en cuenta que estos resultados no dependen del tamaño del alfabeto, sólo del tamaño —n— de los datos a comprimir y de su tipo (lo que determina S).

> "Creo que está al principio de la cuestión," dijo Haydock, "y puedo ver que se avecina uno de esos terribles ejercicios de probabilidad, donde seis hombres tienen sombreros blancos y otros seis, sombreros negros; y hay que calcular la probabilidad de que los sombreros se mezclen y en qué proporción." Si empieza a pensar en cosas como ésa, va a volverse loco. ¡Permítame convencerle de eso!
>
> —Agatha Christie, *The Mirror Crack'd*[a]
>
> [a] El espejo roto.

2.10. MNP5

Microcom, Inc., un fabricante de módems, ha desarrollado un protocolo (llamado MNP, por *Microcom Networking Protocol*[26]) para utilizar en sus módems. Entre otras cosas, el protocolo MNP

[26] Protocolo de creación de redes de Microcom.

Byte	Frec.	Token	Byte	Frec.	Token	Byte	Frec.	Token
0	0	000|0	18	0	100|0010	36	0	101|00100
1	0	000|1	19	0	100|0011	⋮	0	⋮
2	0	001|0	20	0	100|0100	50	0	101|10010
3	0	001|1	21	0	100|0101	⋮	0	⋮
4	0	010|00	22	0	100|0110	64	0	110|000000
5	0	010|01	23	0	100|0111	⋮	0	⋮
6	0	010|10	24	0	100|1000	127	0	110|111111
7	0	010|11	25	0	100|1001	128	0	111|0000000
8	0	011|000	26	0	100|1010	⋮	0	⋮
9	0	011|001	27	0	100|1011	247	0	111|1110111
10	0	011|010	28	0	100|1100	248	0	111|1111000
11	0	011|011	29	0	100|1101	249	0	111|1111001
12	0	011|100	30	0	100|1110	250	0	111|1111010
13	0	011|101	31	0	100|1111	251	0	111|1111011
14	0	011|110	32	0	101|00000	252	0	111|1111100
15	0	011|111	33	0	101|00001	253	0	111|1111101
16	0	100|0000	34	0	101|00010	254	0	111|1111110
17	0	100|0001	35	0	101|00011	255	0	111|11111110

Los valores: 37 a 49, 51 a 63, 65 a 126 y 129 a 246, siguen el mismo patrón.

Tabla 2.35: Los tokens de MNP5.

especifica cómo desempaquetar bytes en bits individuales antes de ser enviados por el módem, cómo transmitir los bits en serie en los modos síncrono y asincróno, y qué técnicas de modulación utilizar. Cada especificación se denomina *clase* y las clases 5 y 7 especifican los métodos para la compresión de datos. Estos métodos (especialmente MNP5) se han vuelto muy populares y han sido utilizados en muchos módems de las décadas de 1980 y 1990.

El método MNP5 es un proceso de dos etapas que comienza con la codificación *run-length*, seguida por la codificación de frecuencia adaptativa.

La primera etapa se describe en la página 28 y se repite aquí. Cuando se encuentran tres o más bytes idénticos consecutivos en la cadena fuente, el compresor emite tres copias de los bytes en su secuencia de salida, seguidas del número de repeticiones. Cuando el descompresor lee tres bytes consecutivos idénticos, sabe que el siguiente byte es un contador de repeticiones (que puede ser cero, indicando sólo tres repeticiones). Un inconveniente del método es que un *run* de tres caracteres en la cadena de entrada se convierte en cuatro caracteres en la secuencia de salida (expansión). Un *run* de cuatro caracteres no produce compresión. Sólo pueden comprimirse *runs* de más de cuatro caracteres. Otro ligero problema es que la cuenta máxima está limitada artificialmente a 250 en lugar de a 255.

La segunda etapa opera con los bytes de la secuencia parcialmente comprimida generada en la primera fase. La etapa 2 es similar al método de la Sección 2.9.5. Comienza con una tabla de 256×2 entradas, donde cada entrada corresponde a uno de los 256 valores posibles de 8 bits, bytes desde el 00000000 hasta el 11111111. La primera columna —el contador de frecuencias— se inicializa a todo ceros. La columna 2 se inicializa con códigos de longitud variable —llamados ***tokens***— que varían desde el corto "000|0" al extenso "111|11111110". La columna 2 con los tokens se muestra en la Tabla 2.35 (con frecuencias cero en la columna 1). Cada token comienza con una cabecera de 3 bits, seguida por un código de tantos bits como indica dicha cabecera.

Los bits de código (con tres excepciones) son: los dos códigos de 1 bit: 0 y 1; los cuatro códigos de

2 bits: de 0 a 3; los ocho códigos de 3 bits: de 0 a 7; los dieciséis códigos de 4 bits; los treinta y dos códigos de 5 bits; los sesenta y cuatro códigos de 6 bits y los ciento veintiocho códigos de 7 bits. Esto proporciona un total de $2+4+8+16+32+64+128 = 254$ códigos. Las tres excepciones son: los dos primeros códigos "000|0" y "000|1" y el último código, que es "111|11111110" en lugar del esperado "111|1111111".

Cuando comienza la fase 2, a todas las 256 entradas de la columna 1 se les asigna un contador de frecuencia cero. Cuando se lee el byte siguiente —B— de la secuencia de entrada (en realidad, se lee de la salida proporcionada por la primera etapa), se escribe el token correspondiente en la cadena de salida y se incrementa en 1 la frecuencia de la entrada B. Después de ésto, los tokens pueden intercambiarse para garantizar que las entradas de la tabla con frecuencias grandes siempre tengan los tokens más cortos (para más detalles, véase la sección siguiente). Observe que sólo se intercambian los tokens, no los contadores de frecuencia. Así, la primera entrada corresponde siempre al byte "00000000" y contiene su contador de frecuencia. El token de este byte, no obstante, puede ser algo más largo que el original "000|0" si otros bytes tienen almacenadas cuentas de frecuencia más altas.

El recuento de la frecuencia se almacena en campos de 8 bits (contadores). Cada vez que se incrementa un contador, el algoritmo comprueba si ha alcanzado su valor máximo. Si es así, todos los contadores se reducen a escala dividiéndolos por 2 (división entera).

Otro punto sutil, tiene que ver con la interacción entre las dos etapas de la compresión. Recordemos que cada repetición de tres o más caracteres se reemplaza —en la etapa 1— por ésos tres elementos idénticos seguidos de un byte con el número de repeticiones. Cuando estos cuatro bytes llegan a la fase 2, se sustituyen por tokens; pero el cuarto no causa un incremento del contador de frecuencia.

Ejemplo: Supongamos que el carácter con código ASCII 52 se repite seis veces. La etapa 1 generará los cuatro bytes: `52,␣52,␣52,␣3`; la etapa 2 sustituirá cada uno de ellos por un token, incrementará la entrada correspondiente a `52` (que es la `53` en la tabla) en 3, pero no aumentará la entrada para `3` (que es la 4 en la tabla). (Los tres tokens para los tres bytes `52` pueden ser diferentes, ya que los tokens puede intercambiarse cada vez que se lee y procesa un `52`.)

La salida de la etapa 2 se compone de tokens de diferentes tamaños, desde 4 hasta 11 bits. Ésta se empaqueta en grupos de 8 bits, que se escriben en la secuencia de salida. A continuación, se escribe un código especial que consta de 11 bits a 1 (el token de vaciado), seguido de tantos bits 1 como sea necesario, para completar el último grupo de 8 bits.

La eficiencia de MNP5 la proporciona el resultado de ambas etapas. La eficiencia de la etapa 1 depende en gran medida de los datos originales. La etapa 2 también depende de los datos originales, pero en menor medida. En ésta se intentan identificar los elementos más frecuentes en los datos y asignarles los códigos más cortos. Un vistazo a la Tabla 2.35 muestra que 32 de los 256 elementos tienen tokens de 7 bits de longitud o menos, lo que produce compresión. Los otros 224 tienen tokens de 8 bits o más; el resultado de sustituir estos bytes por los tokens correspondientes es: ninguna compresión (tokens de 8 bits) o incluso expansión (los demás).

La eficiencia de MNP5 depende, por consiguiente, del número de caracteres que predominan en los datos originales. Si todos los caracteres tienen la misma frecuencia de aparición, se producirá una expansión de los datos. En el otro caso extremo, si sólo aparecen cuatro caracteres en los datos, a cada uno se le asignará un token de 4 bits y el factor de compresión será 2.

◇ **Ejercicio 2.26 (sol. en pág. 1059):** Suponiendo que todos los 256 caracteres aparecen en los datos de origen con la misma probabilidad ($1/256$ cada uno), cuál será el factor de expansión en la etapa 2?

2.10.1. Actualización de la tabla

El proceso de actualización de la tabla de códigos de MNP5 mediante el intercambio de las filas se puede realizar de dos maneras:

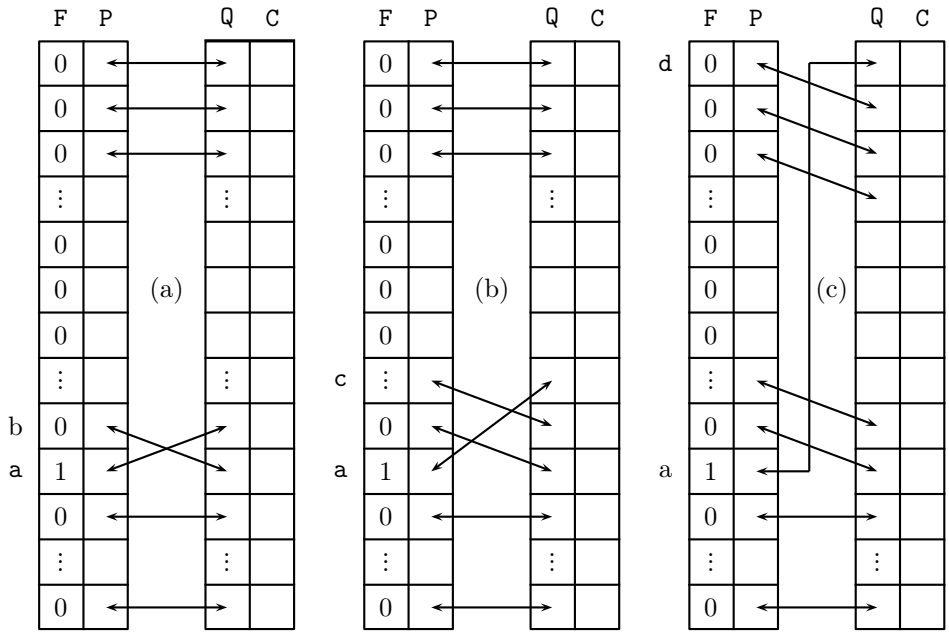


Figura 2.36: Intercambio de punteros en la tabla de códigos de MNP5. (I)

1. Ordenando toda la tabla cada vez que se incrementa una frecuencia. Ésto es sencillo en concepto, pero demasiado lento en la práctica, porque la tabla tiene 256 entradas.

2. Usando punteros en la tabla e intercambiándolos de tal manera que los ítems con frecuencias más altas apunten a los códigos más cortos. Este enfoque se ilustra en las Figuras 2.36-I y 2.36-II. La figura muestra la tabla de códigos organizada en cuatro columnas etiquetadas `F`, `P`, `Q` y `C`. Las columnas `F` y `C` contienen las frecuencias y los códigos; las columnas `P` y `Q` contienen punteros que siempre señalan de una a la otra, por consiguiente, si `P[i]` contiene el índice `j` (i.e., apunta a `Q[j]`), entonces `Q[j]` apunta a `P[i]`. Los siguientes párrafos se corresponden con las nueve partes diferentes de la Figura 2.36:

- (*a*) Se lee el primer ítem de datos —`a`— y se incrementa `F[a]`, de 0 a 1. El algoritmo comienza con el puntero `P[a]` que contiene, digamos, `j`. Examina el puntero `Q[j − 1]`, que inicialmente apunta a la entrada `F[b]` —el que se encuentra a la derecha y por encima de `F[a]`—. Como `F[a] > F[b]`, la entrada `a` tiene que asignarse a un código corto y ésto se hace mediante el intercambio de los punteros `P[a]` y `P[b]` (y también los punteros correspondientes de `Q`) .

- (*b*) Se repite el mismo proceso. El algoritmo comienza de nuevo con el puntero `P[a]`, que ahora señala más arriba: a la entrada `b`. Suponiendo que `P[a]` contiene el índice `k`, el algoritmo examina el puntero `Q[k − 1]` —que apunta a la entrada `c`—. Como `F[a] > F[c]`, la entrada `a` debe ser asignada a un código más corto que el de `c`. De nuevo, esto se realiza intercambiando punteros —esta vez `P[a]` y `P[c]`—.

- (*c*) Este proceso se repite, y puesto que `F[a]` es mayor que todas las frecuencias por encima de ella, se intercambian los punteros hasta que `P[a]` apunta a la entrada superior —`d`—. En este punto, la entrada `a` tiene asignada el código más corto.

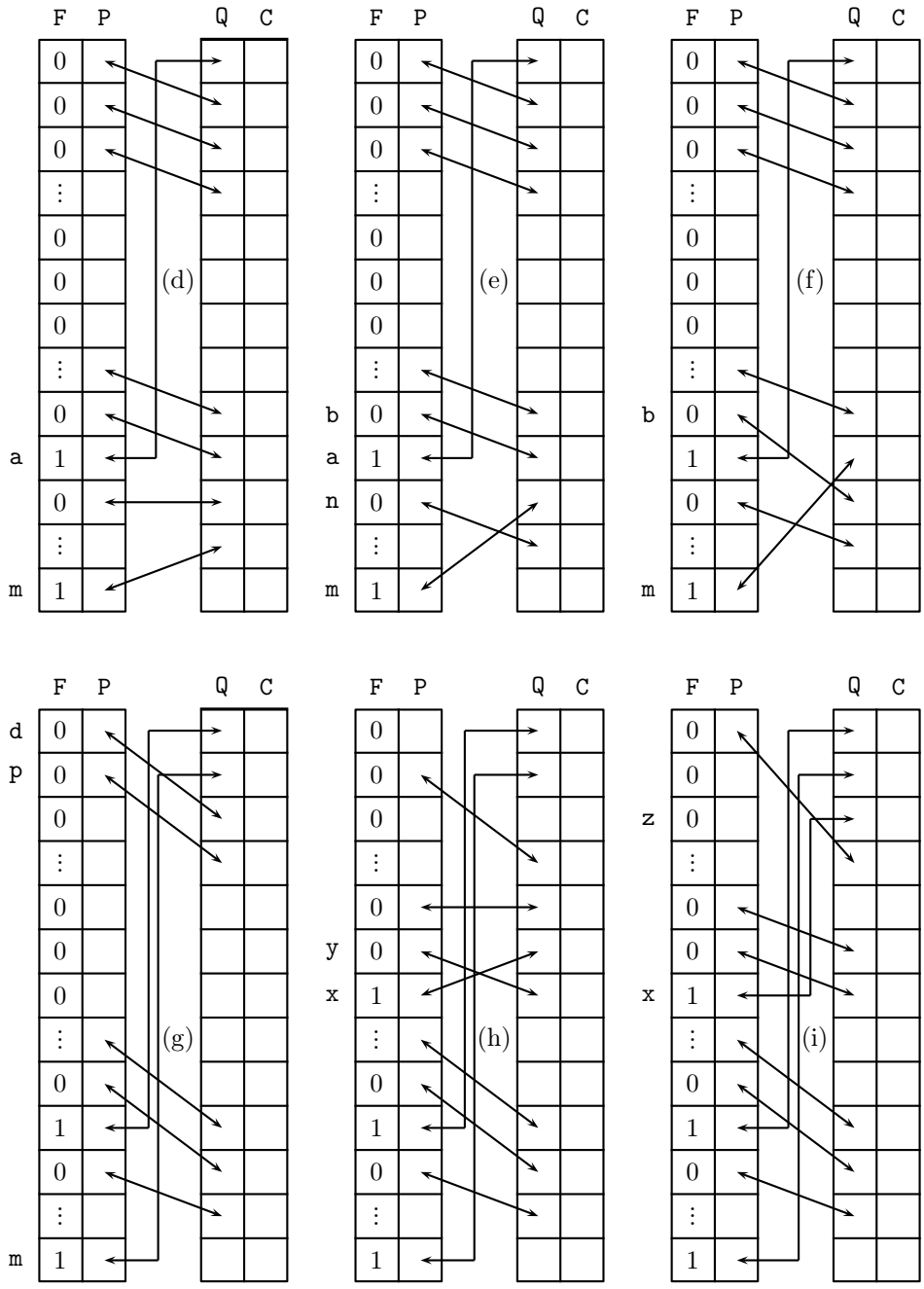


Figura 2.36: Intercambio de punteros en la tabla de códigos de MNP5. (II)

```
F[i]:=F[i]+1;
repeat siempre
  j:=P[i];
  if j=1 then exit;
  j:=Q[j-1];
  if F[i]<=F[j] then exit
    else
    tmp:=P[i]; P[i]:=P[j]; P[j]:=tmp;
    tmp:=Q[P[i]]; Q[P[i]]:=Q[P[j]]; Q[P[j]]:=tmp
  endif;
end repeat
```

Figura 2.37: Intercambio de punteros en MNP5.

- (d) Ahora supongamos que se ha introducido el siguiente ítem de datos y se ha incrementado `F[m]` a 1. Los punteros `P[m]` y el que está por encima de él se intercambian como en (a).
- (e) Después de unos pocos intercambios más, `P[m]` ahora está apuntando a la entrada `n` (la que está justo debajo de `a`). El siguiente paso realiza las asignaciones `j` := `P[m]`; y `b` := `Q[j − 1]`; y el algoritmo compara `F[m]` con `F[b]`. Como `F[m]` > `F[b]`, se intercambian los punteros —como muestra la Figura 2.36f—.
- (f) Después del intercambio, `P[m]` apunta a la entrada `a` y `P[b]` a la entrada `n`.
- (g) Tras unos pocos intercambios más, el puntero `P[m]` apunta a la segunda entrada `p`. De esta manera, a la entrada `m` se le asigna el segundo código más corto. El puntero `P[m]` no se intercambia con `P[a]`, ya que tienen las mismas frecuencias.
- (h) Ahora supongamos que se ha introducido el tercer ítem de datos y se incrementa `F[x]`. Los punteros `P[x]` y `P[y]` se intercambian.
- (i) Después de algunos intercambios más, el puntero `P[x]` apunta a la tercera entrada de la tabla —`z`—. Así es cómo, a la entrada `x` se le asigna el tercer código más corto.

Suponiendo que a continuación se incrementa `F[x]`, se invita al lector a tratar de averiguar cómo se intercambia `P[x]` —primero con `P[m]`, y luego con `P[a]`—, de modo que a la entrada `x` se le asigne el código más corto.

El pseudocódigo de la Figura 2.37 resume el proceso del intercambio de punteros.

> No hay probabilidades de ser aceptado, ¿simplemente porque no son certezas?
>
> —Jane Austen, *Sense and Sensibility*
> *(Sentido y sensibilidad)*

2.11. MNP7

Más complejo y sofisticado que MNP5, MNP7 combina la codificación *run-length* con una variante bidimensional de la codificación de Huffman adaptativa. La etapa 1 identifica las secuencias de caracteres repetidos y emite tres copias del carácter implicado, seguido por un contador de 4 bits que guarda

Carácter anterior \ Carácter actual	**0**	**1**	**2**	...	**254**	**255**
	0 0	0 0	0 0	...	0 0	0 0
	1 0	1 0	1 0	...	1 0	1 0
	2 0	2 0	2 0	...	2 0	2 0
	3 0	3 0	3 0	...	3 0	3 0
	⋮	⋮	⋮		⋮	⋮
	254 0	254 0	254 0	...	254 0	254 0
	255 0	255 0	255 0	...	255 0	255 0

(a)

...**a**	**b**	**c**	**d**	**e**...
t	l	h	o	d
h	e	o	a	r
c	u	r	e	s
⋮	⋮	⋮	⋮	⋮

(b)

Tabla 2.38: Las tablas de código de MNP7.

el número de caracteres restantes en la secuencia. Un valor cero implica una racha de longitud 3 y uno de 15 (el mayor posible en un nibble[27] de 4 bits), una racha de longitud 18. La etapa 2 comienza asignando a cada carácter una tabla completa con muchos códigos de tamaño variable. Cuando se lee un carácter `C`, se selecciona y emite uno de los códigos de su tabla, en función del *carácter anterior* a `C` en la cadena de entrada. Si este carácter es —digamos— `P`, entonces el contador de frecuencia del par (digrama) `PC` se incrementa en 1; las filas de la tabla pueden intercambiarse, usando el mismo algoritmo que para MNP5, para mover el par a una posición en la tabla que tenga un código más corto.

MNP7 se basa por tanto en un modelo de Markov de primer orden, donde se procesa cada elemento dependiendo del mismo y uno de sus predecesores. En un modelo de Markov de orden k, un ítem se procesa en función de sí mismo y k de sus predecesores (no necesariamente los k inmediatos).

A continuación se indican los detalles: Cada uno de los 256 bytes de 8 bits tiene una tabla de códigos asignada —de tamaño 256×2— donde cada fila corresponde a uno de los 256 bytes posibles. La columna 1 de la tabla se inicializa con los números enteros, de 0 a 255; y la columna 2 (los contadores de frecuencia) se inicializa a todo ceros. El resultado es: 256 tablas, cada una con una doble columna de 256 filas (Tabla 2.38a). Se asignan códigos de tamaño variable a las filas, de manera que el primer código es de 1 bit, y los otros crecen en tamaño a medida que se avanza hacia la parte inferior de la tabla. Los códigos se almacenan en una tabla de códigos adicionales que nunca cambia.

Cuando se introduce un carácter `C` (el ítem actual a comprimir), su valor se utiliza como un puntero para seleccionar una de las 256 tablas. Se busca la primera columna de la tabla para encontrar la fila con el valor de 8 bits del carácter anterior a `P`. Una vez que localizada la fila, se emite el código de la misma fila de la tabla de códigos, y el contador de la segunda columna se incrementa en 1. Las filas de la tabla pueden intercambiarse cuando el nuevo valor del contador del digrama `PC` sea lo suficientemente grande.

Después de introducir suficientes caracteres e intercambiar las filas, las tablas empiezan a reflejar las verdaderas frecuencias de los digramas de los datos. La Tabla 2.38b muestra un posible estado, suponiendo que los digramas más corrientes son: `ta`, `ha`, `ca`, `lb`, `eb`, `ub`, `hc`, etc.. Puesto que el digrama de la parte superior está codificado en un bit, MNP7 puede ser muy eficiente. Si los datos originales se componen de texto en un lenguaje natural, donde algunos digramas son muy comunes, MNP7 produce normalmente una razón de compresión alta.

[27] En ensamblador se usa nibble para indicar media palabra. La palabra en este caso es de 8 bits, por lo que el nibble es de 4 bits.

2.12. Fiabilidad

La desventaja principal de los códigos de tamaño variable es su vulnerabilidad a los errores. Se usa la propiedad prefija para decodificar ésos códigos, por lo que un error en un solo bit puede producir una pérdida de sincronización en el descompresor y la consiguiente incapacidad de decodificar el resto de la secuencia comprimida. En el peor caso, el descompresor puede incluso leer, decodificar e interpretar el resto de la datos comprimidos de forma incorrecta, sin darse cuenta de que ha ocurrido un problema.

Ejemplo: Usando el código de la Figura 2.18, la cadena `CARE` se codifica como 10100 0011 0110 000 (sin los espacios). Suponiendo un error en el tercer bit: 10 0 00 0011 0110 000, el descompresor no detectará ningún problema, pero decodificará la cadena como `HARE`.

◇ **Ejercicio 2.27 (sol. en pág. 1059):** ¿Qué sucederá en el caso 11 1 11 0011 0110 000 . . . (la cadena `WARE`. . . con un bit erróneo)?

Los usuarios de los códigos Huffman se han dado cuenta hace mucho tiempo que estos códigos se recuperan rápidamente de un error. Sin embargo, si los códigos de Huffman se utilizan para codificar *run lengths*, entonces esta propiedad no ayuda, ya que todos los *run lengths* estarían desplazados después de un error.

Una forma sencilla de añadir fiabilidad a los códigos de longitud variable es la de trocear las secuencias largas de datos comprimidos —a medida que se están transmitiendo— en grupos de 7 bits y añadir un bit de paridad a cada grupo. De esta manera, el descompresor por lo menos es capaz de detectar un problema, y puede mostrar un mensaje de error o solicitar una retransmisión. Por supuesto, es posible añadir más de un bit de paridad a un grupo de bits de datos, aumentando la fiabilidad. Sin embargo, la fiabilidad es, en cierto sentido, lo opuesto a la compresión. La compresión se realiza disminuyendo la redundancia, mientras que la fiabilidad se logra incrementándola. El trozo de datos más fiable es el de menor compresión, por lo que se debe tener cuidado cuando las dos operaciones se utilizan juntas.

Algunos grupos y organizaciones de estándares importantes

ANSI (American National Standards Institute[28]) es el sistema del sector privado de estandarización voluntaria de los Estados Unidos. Sus miembros son asociaciones profesionales, grupos de consumidores, asociaciones comerciales, y algunas agencias reguladoras del gobierno (esto es, una federación). Recoge y distribuye los estándares elaborados por sus miembros. Su misión es realzar la competitividad global de los negocios de U.S. y la calidad de vida de América, promoviendo y facilitando voluntariamente los estándares consensuales y dando la conformidad de sistemas de evaluación y promocionando su integridad.

ANSI fue fundada en 1918 por cinco sociedades de ingeniería y tres agencias del gobierno. Sigue siendo una organización privada sin fines de lucro. Actualmente representa los intereses de 125 000 compañias y 3,5 millones de profesionales y sus miembros están divididos en las seis categorías: Compañías, Gobierno, Organizaciones, Educación, Internacional e Individual.

ANSI está situado en la calle 25 West 43rd, (entre las avenidas 5ª y 6ª), piso 4, Nueva York, NY 10036, USA. Véase también http://web.ansi.org/.

ISO (International Organization for Standardization[29], Organisation Internationale de Normalisation) es una agencia de las Naciones Unidas, cuyos miembros son las organizaciones de estándares de cerca de 100 países miembros (una organización de cada país). Desarrolla una amplia gama de estándares para las industrias en todos los ámbitos.

Establecido en 1947, su misión es "promover el desarrollo de la estandarización en el mundo con el fin de facilitar el intercambio internacional de bienes y servicios y desarrollar la cooperación en las

[28] Instituto Nacional Americano de eStándares.
[29] Organización Internacional de Normalización (ó eStandarización).

esferas de actividad intelectual, científica, tecnológica y la actividad económica." Este es el centro en el que se acuerda, por ejemplo, qué grosor debe tener una tarjeta de crédito, para que todos los países del mundo sigan un estándar compatibles. La dirección de contacto de la ISO es: ISO Central Secretariat; Head, Human Resources Services; 1, ch. de la Voie-Creuse; CP 56; 1211 Genève 20; Switzerland. Véase también http://www.iso.ch/.

ITU (International Telecommunication Union[30]) es otra agencia de las Naciones Unidas que desarrolla estándares para las telecomunicaciones. Sus miembros son en su mayoría las empresas que fabricación de equipos de telecomunicaciones, los grupos interesados en estándares y las agencias del gobierno involucradas con las telecomunicaciones. La ITU es el sucesor de una organización fundada por unos 20 países de Europa en París en 1865.

La Comisión Electrotécnica Internacional (IEC o *International Electrotechnical Commission*) es una organización internacional no gubernamental que prepara y publica estándares internacionales para las tecnologías eléctricas, electrónicas y similares. La IEC fue fundada en 1906, como resultado de una resolución aprobada en el Congreso Eléctrico Internacional, celebrada en St. Louis (USA) en 1904. Está compuesto por más de 50 países participantes, incluidas todas las principales naciones comerciales del mundo y un número creciente de países industrializados.

La misión de la IEC es promover, a través de sus miembros, la cooperación internacional sobre todas las cuestiones de estandarización electrotécnica y relacionadas, tales como la evaluación de la conformidad con los estándares, en los campos de la electricidad, la electrónica y las tecnologías afines.

La oficina central de la IEC se localiza en: 3, rue de Varembé; P.O. Box 131; CH-1211 GENEVA 20; Switzerland. Véase también http://www.iec.ch/.

QIC (Quarter-Inch Cartridge[31]) es una asociación comercial internacional, constituida en 1987, para estimular y fomentar el uso generalizado de la tecnología del cartucho de un cuarto de pulgada. Su misión incluye la promoción de la tecnología QIC entre los usuarios de ordenadores, revendedores, distribuidores, OEMs, integradores de sistemas, analistas de la industria, el comercio y prensa técnica, y la formulación de estándares de desarrollo para la compatibilidad entre los diferentes fabricantes de discos, cartuchos y subsistemas.

La actividad promocional de QIC enfatiza la rentabilidad, fiabilidad y facilidad de uso, ya que establece estándares sobre tipos de productos que refuerzan continuamente la confianza del usuario y la migración de la tecnología hacia el futuro.

El QIC es una extensión del Working Group for Quarter-Inch Cartridge Drive Compatibility[32] (también conocido como QIC), una organización informal que comenzó en 1982 por varios fabricantes de unidades.

La pertenencia ejecutiva en QIC está abierta a todos los fabricantes de unidades de cartuchos de un cuarto de pulgada y medios de comunicación; los fabricantes de cabezas, componentes críticos y software pueden convertirse en Miembros Tecnológicos y cualquiera de las partes interesadas que se comprometa con la tecnología QIC es bienvenido a convertirse en un asociado de la organización. Está situado en: 100 North Hope Avenue; Suite 20C; Santa Barbara; CA 93110-1600; USA (http://www.qic.org/).

---

[30] Unión Internacional de Telecomunicaciones.
[31] Cartucho de un cuarto de pulgada.
[32] Grupo de Trabajo para la compatibilidad de unidades de cartucho de un cuarto de pulgada.

Su único otro visitante era una mujer: la mujer que había asistido a su lectura. En ese momento ella le parecía la única persona presente que no había prestado la menor atención a sus palabras. Con una coqueta timidez, ella se acercó a su mesa. Richard le dio la bienvenida cortésmente y resultó significativo que ella extrajera de su bolso una copia de una novela escrita, no por Richard Tull, sino por Fyodor Dostoievski: *The Idiot*[a]. De pie junto a él, inclinada sobre él, con el rostro terriblemente caliente y cerca, empezó a hojear sus páginas, explicándolo. Además, el libro estaba coloreado, no por gotas de sangre, sino por dos escrituras destacadas rivalizando entre sí, una azul y una rosa. Y no sólo dos páginas, sino las seiscientas. Cada vez que las letras *h* y *e* aparecían juntas, como en *the, then, there* y en *forehead, Pashlishtchev, sheepskin*, estaban sombreadas en azul. Cada vez que las letras *s, h* y *e* aparecían juntas, como en *she, sheer, ashen, sheepskin,* etc, estaban sombreadas en color rosa. Y como cada *she* contenía un *he*, era indiscutible el predominio masculino, como era de esperar. Lo que era exactamente su característica principal. "¿Lo ve?", dijo con su aliento caliente, con olor a sustancias metálicas, de baterías y planchas de impresión. "¿Lo ve?"... Los organizadores sabían todo acerca de esta mujer —esta lamentable recurrencia, este infatigable obstáculo— y seguía acercándose para intentar engatusarle incesantemente.

—Martin Amis, *La Información.*

[a]El Idiota.

2.13. Compresión de faxes (facsímiles)

La compresión de datos es especialmente importante cuando las imágenes se transmiten a través de una línea de comunicación, porque el usuario está a menudo esperando en el receptor, deseoso de ver algo rápidamente. Los documentos transferidos entre las máquinas de fax se envían como mapas de bits, por lo que, cuando esas máquinas llegaron a ser populares, se hizo necesario un método de compresión estándar de datos. El ITU-T (UIT-T, en español) propuso y desarrolló varios métodos.

La ITU-T es una de las cuatro partes permanentes de la Unión Internacional de Telecomunicaciones (ITU o UIT), con sede en Geneva, Switzerland (http://www.itu.ch/). Emite recomendaciones de los estándares aplicables a los módems, los interfaces de conmutación de paquetes, conectores V.24 y dispositivos similares. Aunque no tiene poder de aplicación, los estándares que recomienda son generalmente aceptados y adoptados por la industria. Hasta marzo de 1993, la ITU-T era conocido como el Comité de Consulta para la Telefonía y Telegrafía International (Comité Consultatif International Télégraphique et Téléphonique, o CCITT).

CCITT: Can't Conceive Intelligent Thoughts Today.[a]

[a]Hoy no se pueden concebir pensamientos inteligentes.

Los primeros estándares de compresión de datos desarrollados por la ITU-T fueron el T2 (también conocido como Grupo 1) y el T3 (Grupo 2). Éstos se han quedado obsoletos y han sido sustituidos por el T4 (Grupo 3) y el T6 (Grupo 4). El Grupo 3 se utiliza actualmente por todas las máquinas de fax diseñadas para operar con la red telefónica pública conmutada (PSTN[33]). Éstas son las máquinas que tenemos en casa, y actualmente operan a velocidades que varían entre 2,4 y 33,6 Kbps. El Grupo 4 se utiliza en las máquinas de fax, diseñadas para funcionar en una red digital, como ISDN. Tienen velocidades típicas de 64K baudios. Ambos métodos pueden producir factores de compresión de 10 o superior, reduciendo el tiempo de transmisión de una página típica en alrededor de un minuto con el primero y unos pocos segundos con el último.

Algunas referencias para la compresión de faxes son: [Adelson et al. 87], [Hunter y Robinson 80], [Marking 90] y [McConnell 92].

2.13.1. Codificación unidimensional

Una máquina de fax escanea un documento línea a línea, convirtiendo cada línea en pequeños puntos blancos y negros, llamados *pels* (de Picture ELement). La resolución horizontal es siempre de unos 8,04 pels por milímetro[34] (alrededor de 205 pels por pulgada[35]). Una línea de escaneo de 8,5 pulgadas de ancho[36] se convierte, por lo tanto, en 1728 pels. El estándar T4, sin embargo, recomienda escanear sólo alrededor de 8,2 pulgadas, lo que equivale a 1664 pels por línea de exploración (estos números, así como los del siguiente párrafo, tienen todos una precisión de $\pm 1\,\%$).

La resolución vertical es de 3,85 líneas por milímetro en modo estándar y $7{,}7\,^{líneas}/_{\mathrm{mm}}$ en modo fino. Muchas máquinas de fax tienen también un modo muy fino, en el que se exploran $15{,}4\,^{líneas}/_{\mathrm{mm}}$. La Tabla 2.39 supone que la página es de 10 pulgadas de alto (254 mm) y muestra el número de líneas escaneadas por página, el total de pels por línea, el total de pels por página y los tiempos de transmisión típica (en segundos y minutos), para los tres modos sin compresión. Los tiempos son largos, lo que ilustra la importancia de la compresión de datos en las transmisiones de fax.

[33]Public Switched Telephone Network.

[34]Un total de 1728 pels representa la exploración de una línea horizontal de 215 mm; ésto corresponde a $^{1728\,pels}/_{215\,mm} \approx 8{,}04\,^{pels}/_{mm}$.

[35]$1\,pulgada = 25{,}4\,mm$.

[36]El formato de carta norteamericano es de 215,9 × 279,4 mm.

Líneas escaneadas	Pels por línea	Pels por página	Tiempo (seg.)	Tiempo (min.)
978	1664	1,670M	170	2,83
1956	1664	3,255M	339	5,65
3912	1664	6,510M	678	11,3
Diez pulgadas son 254 mm. El número de pels está en millones y los tiempos de transmisión, a 9600 baudios y sin compresión, están entre 3 y 11 minutos, dependiendo del modo. Sin embargo, si la página es más corta que 10 pulgadas o si la mayor parte está en blanco, el factor de compresión puede ser de 10 ó mejor, obteniendo tiempos de transmisión de entre 17 y 68 segundos.				

Tabla 2.39: Tiempos de transmisión de un fax.

Imagen	Descripción
1	Cartas comerciales con tipo (en inglés)
2	Diagrama del circuito (dibujado a mano)
3	Factura impresa con tipo (en francés)
4	Informe densamente escrito (en francés)
5	Artículo técnico impreso, con figuras y ecuaciones (en francés)
6	Gráfico con títulos impresos (en francés)
7	Documento denso (en kanji)
8	Nota manuscrita con letras blancas sobre negro muy grandes (en inglés)

Tabla 2.40: Los ocho documentos de adiestramiento del CCITT.

Para obtener el código del Grupo 3, la ITU-T ha contado todas las rachas de pels blancos y negros que aparecen en un conjunto de ocho documentos de "adiestramiento" que, a su juicio, representan el texto y las imágenes más frecuentes que se envían por fax y utiliza el algoritmo de Huffman para asignar un código de tamaño variable a cada racha. (Los ocho documentos se describen en la Tabla 2.40. No se muestran porque son propiedad de la ITU-T). Las rachas más comunes encontradas son de 2, 3 y 4 píxeles negros, por lo que a éstos se les han asignado los códigos más cortos (Tabla 2.41a 2.41a). Les siguen las rachas de 2 a 7 píxeles blancos, asociados a códigos un poco más largos. Las rachas que aparecen raramente tienen asignadas los códigos más largos, de 12 bits. De este modo, el Grupo 3 utiliza una combinación de códigos RLE y de Huffman.

◇ **Ejercicio 2.28 (sol. en pág. 1059):** Una racha de 1664 pels blancos tiene asignado el corto código 011000. ¿Por qué es esta cantidad tan común?

Dado que las rachas pueden ser largas, el algoritmo de Huffman fue modificado. Los códigos se asignaron para *run lengths* de 1 a 63 pels (que son los códigos de terminación de la Tabla 2.41a) y para los *run lengths* múltiplos de 64 pels (los códigos de establecimiento de la Tabla 2.41b-I) y de la Tabla 2.41b-II). El Grupo 3 es, por tanto, un *código de Huffman modificado* (también llamado *MH*). El código de un *run length* o racha es: o bien un código de terminación único (si el *run length* es corto) o uno o más códigos de establecimiento, seguidos por un código de terminación (si es largo). Estos son algunos ejemplos:

1. Una racha de 12 pels blancos se codifica: *001000*.
2. Una racha de 76 pels blancos ($= 64 + 12$) se codifica: *11011/001000*.

Racha	Palabra de de código (blancos)	Palabra de de código (negros)	Racha	Palabra de de código (blancos)	Palabra de de código (negros)
0	00110101	0000110111	32	00011011	000001101010
1	000111	010	33	00010010	000001101011
2	0111	11	34	00010011	000011010010
3	1000	10	35	00010100	000011010011
4	1011	011	36	00010101	000011010100
5	1100	0011	37	00010110	000011010101
6	1110	0010	38	00010111	000011010110
7	1111	00011	39	00101000	000011010111
8	10011	000101	40	00101001	000001101100
9	10100	000100	41	00101010	000001101101
10	00111	0000100	42	00101011	000011011010
11	01000	0000101	43	00101100	000011011011
12	001000	0000111	44	00101101	000001010100
13	000011	00000100	45	00000100	000001010101
14	110100	00000111	46	00000101	000001010110
15	110101	000011000	47	00001010	000001010111
16	101010	0000010111	48	00001011	000001100100
17	101011	0000011000	49	01010010	000001100101
18	0100111	0000001000	50	01010011	000001010010
19	0001100	00001100111	51	01010100	000001010011
20	0001000	00001101000	52	01010101	000000100100
21	0010111	00001101100	53	00100100	000000110111
22	0000011	00000110111	54	00100101	000000111000
23	0000100	00000101000	55	01011000	000000100111
24	0101000	00000010111	56	01011001	000000101000
25	0101011	00000011000	57	01011010	000001011000
26	0010011	000011001010	58	01011011	000001011001
27	0100100	000011001011	59	01001010	000000101011
28	0011000	000011001100	60	01001011	000000101100
29	00000010	000011001101	61	00110010	000001011010
30	00000011	000001101000	62	00110011	000001100110
31	00011010	000001101001	63	00110100	000001100111

Tabla 2.41: (a) Códigos de fax Grupos 3 y 4. Códigos de terminación.

Run length (Racha)	**Palabra de de código (blancos)**	**Palabra de de código (negros)**	**Run length (Racha)**	**Palabra de de código (blancos)**	**Palabra de de código (negros)**
64	11011	0000001111	960	011010100	0000001110011
128	10010	000011001000	1024	011010101	0000001110100
192	010111	000011001001	1088	011010110	0000001110101
256	0110111	000001011011	1152	011010111	0000001110110
320	00110110	000000110011	1216	011011000	0000001110111
384	00110111	000000110100	1280	011011001	0000001010010
448	01100100	000000110101	1344	011011010	0000001010011
512	01100101	0000001101100	1408	011011011	0000001010100
576	01101000	0000001101101	1472	010011000	0000001010101
640	01100111	0000001001010	1536	010011001	0000001011010
704	011001100	0000001001011	1600	010011010	0000001011011
768	011001101	0000001001100	1664	011000	0000001100100
832	011010010	0000001001101	1728	010011011	0000001100101
896	011010011	0000001110010	EOL	000000000001	000000000001

NOTA: Esta tabla sirve para terminales que aceptan papel de mayor anchura conservando la resolución horizontal normal.

(b) — Parte I

Tabla 2.41: (b.I) Códigos de fax Grupos 3 y 4. Códigos de establecimiento.

Run length (Racha)	**Palabra de código (blancos y negros)**	**Run length (Racha)**	**Palabra de código (blancos y negros)**
1792	00000001000	2240	000000010110
1856	00000001100	2304	000000010111
1920	00000001101	2368	000000011100
1984	000000010010	2432	000000011101
2048	000000010011	2496	000000011110
2112	000000010100	2560	000000011111
2176	000000010101		

NOTA: Las rachas superiores a 2624 pels se codifican poniendo en primer lugar el código de establecimiento 2560. Si el número de repeticiones después de dicho código es superior o igual a 2560 elementos, se emiten tantos códigos 2560 adicionales como sean necesarios, hasta que el número de pels desde el último código hasta el final de la racha sea inferior a 2560. Esta parte restante se codifica con un código de terminación o un código de establecimiento seguido de un código de terminación determinado por las tablas.

(b) — Parte II

Tabla 2.41: (b.II) Códigos de Fax Grupos 3 y 4. Códigos de establecimiento.

3. Una racha de 140 pels blancos (= 128 + 12) se codifica: *10010/001000*.
4. Una racha de 64 pels negros (= 64 + 0) se codifica: *0000001111/0000110111*.
5. Una racha de 2561 pels negros (= 2560 + 1) se codifica: *000000011111/010*.

◇ **Ejercicio 2.29 (sol. en pág. 1059):** No hay rachas de longitud cero. ¿Por qué entonces se han asignado códigos a *runs* de cero pels negros y cero blancos?

◇ **Ejercicio 2.30 (sol. en pág. 1059):** Una exploración de una línea de 8,5 pulgadas de ancho da como resultado 1728 pels, por lo tanto, ¿cómo puede haber una racha de 2561 pels consecutivos?

Cada línea de exploración se codifica por separado y finaliza con el código especial EOL de 12 bits 000000000001. Cada línea también obtiene un pel blanco añadido a la izquierda cuando es escaneada. Esto se hace para eliminar cualquier ambigüedad a la hora de decodificar la línea en el receptor. Después de leer el código EOL de la línea anterior, el receptor asume que la nueva línea se inicia con una racha de pels blancos e ignora la primera de ellas. Ejemplos:

1. La línea de 14 pels ■■■□□■■□□□□□□□ se codifica como los *run lengths 1b 3n 2b 2n 7b EOL*, que se convierten en la secuencia binaria *000111 | 10 | 0111 | 11 | 1111 | 000000000001*. El decodificador ignora el pel blanco del principio.
2. La línea □□■■■■■□□□□□■■ se codifica como los *run lengths 3b 5n 5b 2n EOL*, que se convierten en la cadena binaria *1000 | 0011 | 1100 | 11 | 000000000001*.

◇ **Ejercicio 2.31 (sol. en pág. 1059):** El código del grupo 3 para un *run length* de cinco pels negros (*0011*) es también el prefijo de los códigos de los *run lengths* de 61, 62 y 63 pels blancos. Explíquese ésto.

El código del Grupo 3 no tiene corrección de errores, pero muchos errores pueden ser detectados. Debido a la naturaleza del código de Huffman, incluso un error en un solo bit en la transmisión puede hacer que el receptor pierda la sincronización y produzca una cadena erronea de pels. Ésta es la razón por la que cada línea de exploración se codifica por separado. Si el receptor detecta un error, se salta bits, buscando un código EOL. De esta forma, un error puede provocar la recepción incorrecta de, como mucho, una línea de exploración. Si el receptor no encuentra un EOL después de un cierto número de líneas, supone que la tasa de error es alta y anula el proceso, notificándoselo al transmisor. Puesto que los códigos son entre 2 y 12 bits de largo, el receptor detecta un error si no puede decodificar un código válido después de leer 12 bits.

Cada página del documento codificado es precedida por un código EOL y seguido por seis códigos EOL. Como cada línea está codificada por separado, este método es un sistema de *codificación unidimensional*. La razón de compresión depende de la imagen. Las imágenes con gran número de zonas contiguas en negro o blanco (texto o imágenes en blanco y negro) alcanzan un alto grado de compresión. Las imágenes con muchos *run lengths* cortos a veces pueden producir compresión negativa. Esto es especialmente cierto en el caso de imágenes en escala de grises (como fotografías escaneadas). Las tonalidades se producen por semitonos, que abarcan zonas con mucha alternancia de pels negros y blancos (*runs* o rachas de longitud uno).

◇ **Ejercicio 2.32 (sol. en pág. 1061):** ¿Cuál es la razón de compresión para *runs* de longitud uno (i.e., estrictamente pels alternos)?

El estándar T4 también permite incluir bits de relleno entre los bits de datos y el código EOL. Esto se hace en los casos en que se necesita una pausa o cuando el número total de bits transmitidos por una línea de exploración debe ser un múltiplo de 8. Los bits de relleno son ceros.

Ejemplo: La cadena binaria 000111 | 10 | 0111 | 11 | 1111 | 0000000001 se convierte en

$$000111\,|\,10\,|\,0111\,|\,11\,|\,1111\,|\,00\,|\,0000000001$$

después de añadir dos ceros de relleno, con lo que la longitud total de la cadena es de 32 bits ($= 8 \times 4$). El decodificador ve los dos ceros de relleno, seguidos por los 11 ceros del código EOL y el 1, por lo que sabe que ha encontrado un relleno seguido por un EOL.

Véase http://docstore.mik.ua/rfc/rfc0804.html para obtener una descripción del grupo 3.

En el momento de escribir esto, las recomendaciones T.4 y T.6 también se pueden encontrar en formato pdf en la URL http://www.itu.int/rec/T-REC-T/en.

2.13.2. Codificación bidimensional

La codificación bidimensional fue desarrollada porque la unidimensiónal no produce buenos resultados para imágenes en escala de grises. La codificación bidimensional es opcional en los faxes que utilizan el Grupo 3, pero es el único método utilizado por las máquinas destinadas a trabajar en una red digital. Cuando una máquina de fax usa el grupo 3, admite opcionalmente la codificación bidimensional; tras cada EOL se agrega un bit extra para indicar el método de compresión utilizado en el escaneo de la siguiente línea. Ése bit es 1 si la siguiente línea se codifica con la codificación unidimensional y 0 si se codifica con la codificación bidimensional.

El método de codificación bidimensional también se llama MMR, por *modified modified READ*[37], donde READ es un acrónimo de *relative element address designate*[38]. El término "modified modified" se utiliza porque es una modificación de la codificación unidimensional, la cual es en sí misma una modificación del método de Huffman original. La codificación bidimensional funciona comparando la línea de escaneo actual (llamada *línea de codificación*[39]) con la precedente (llamada *línea de referencia*[40]) y registrando las diferencias entre ellas; se parte del supuesto de que dos líneas consecutivas de un documento normalmente se diferencian en sólo unos pocos pels. El método supone que hay una línea en blanco en la parte superior de la página, que se utiliza como línea de referencia para la exploración de primera línea de la página. Después de codificar la primera línea, ésta se convierte en la línea de referencia y se codifica la segunda línea escaneada. Al igual que en la codificación unidimensional, se supone que cada línea comienza con un pel blanco, que es ignorado por el receptor.

El método de codificación bidimensional es menos fiable que el unidimensional, ya que un error en la decodificación de una línea provocará errores en la decodificación de todas las siguientes y se propagará por todo el documento. Ésta es la razón por la que el estándar T.4 (Grupo 3) incluye el requisito de que después de codificar una línea con el método unidimensional, no deben ser codificadas con el método bidimensional más de $K - 1$ líneas. $K = 2$ para la resolución estándar y $K = 4$ para la resolución fina. El estándar T.6 (Grupo 4) no tiene este requisito y utiliza exclusivamente la codificación bidimensional.

El escaneo de la línea de codificación y la comparación con la línea de referencia produce tres casos o modos. El modo se identifica mediante la comparación del *run length* de la línea de referencia [(b_1b_2) en la Figura 2.43] con el *run length* actual (a_0a_1) y el siguiente (a_1a_2), en la línea de codificación. Cada una de estas tres rachas puede ser de elementos negros o blancos. Los tres modos son los siguientes (véase también el diagrama de flujo de la Figura 2.44):

1. **Modo paso**. Ocurre cuando (b_1b_2) está a la izquierda de (a_1a_2) y b_2 está a la izquierda de a_1 (Figura 2.43a). Este modo no incluye el caso en el que b_2 está justo encima de a_1. Cuando este modo se identifica, se codifica la longitud del *run* (b_1b_2) utilizando los códigos de la Tabla 2.42

[37] READ modificado modificado.
[38] Dirección designada relativa de un elemento.
[39] Coding line.
[40] Reference line.

Modo	Run lengths a codificar	Notación	Palabra de código
Paso	b_1b_2	P	0001 + longitud codificada de b_1b_2
Horizontal	a_0a_1, a_1a_2	H	001 + long. codificada de a_0a_1 y a_1a_2
	$a_1b_1 = 0$	$V\,(0)$	1
	$a_1b_1 = +1$	$V_R\,(1)$	011
	$a_1b_1 = +2$	$V_R\,(2)$	000011
	$a_1b_1 = +3$	$V_R\,(3)$	0000011
Vertical	$a_1b_1 = -1$	$V_L\,(1)$	010
	$a_1b_1 = -2$	$V_L\,(2)$	000010
	$a_1b_1 = -3$	$V_L\,(3)$	0000010
	Un valor positivo de a_1b_1 indica que a_1 está a la derecha de b_1. Un valor negativo de a_1b_1 indica que a_1 está a la izquierda de b_1.		
Ampliación			0000001000

Tabla 2.42: Códigos 2D para el método del Grupo 4.

y se transmite. El puntero a_0 se situa debajo de b_2 y se actualizan los cuatro valores b_1, b_2, a_1 y a_2.

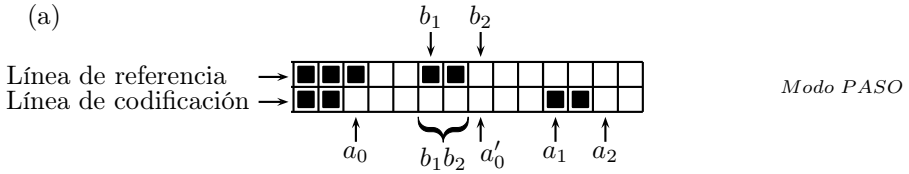
2. **Modo vertical**. (b_1b_2) se solapa con (a_1a_2) por no más de tres píxeles (Figuras 2.43b1 y b2). Suponiendo que las líneas consecutivas no difieran en mucho, éste es el caso más común. Cuando este modo se identifica, se produce uno de los siete códigos (Tabla 2.42) y se transmite. Los punteros se actualizan como en el modo paso. El rendimiento del método de codificación bidimensional depende de lo común que sea este caso.

3. **Modo horizontal**. (b_1b_2) se solapa con (a_1a_2) en más de tres píxeles (Figuras 2.43c1 y c2). Cuando este modo se identifica, la longitud de los *runs* (a_0a_1) y (a_1a_2) se codifica utilizando los códigos de la Tabla 2.42 y se transmite. Los punteros se actualizan como en los casos 1 y 2 anteriores.

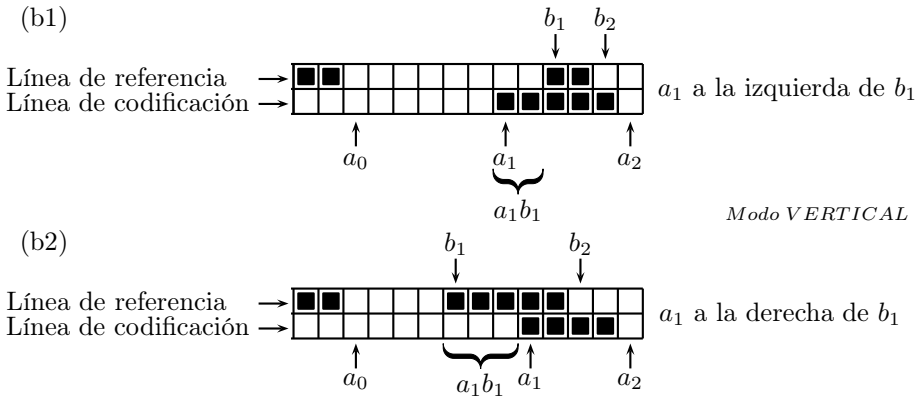
> ... y pensaste "impresionante"; las estadísticas eran 36-24-36.
>
> —Anuncio, *The American Statistician*, noviembre de 1979

Cuando el escaneo comienza, al puntero a_0 se le asigna un pel blanco imaginario a la izquierda de la línea de codificación y se establece a_1 de manera que apunte al primer pel negro de la línea de codificación. (Recuérdese que a_0 corresponde a un pel imaginario, por lo que la primera longitud de *run* es $|a_0a_1| - 1$.) El puntero a_2 pasa a apuntar al primer pel blanco que se encuentre tras ése. Los punteros b_1 y b_2 pasan a señalar al inicio del primer y segundo *run* de la línea de referencia, respectivamente.

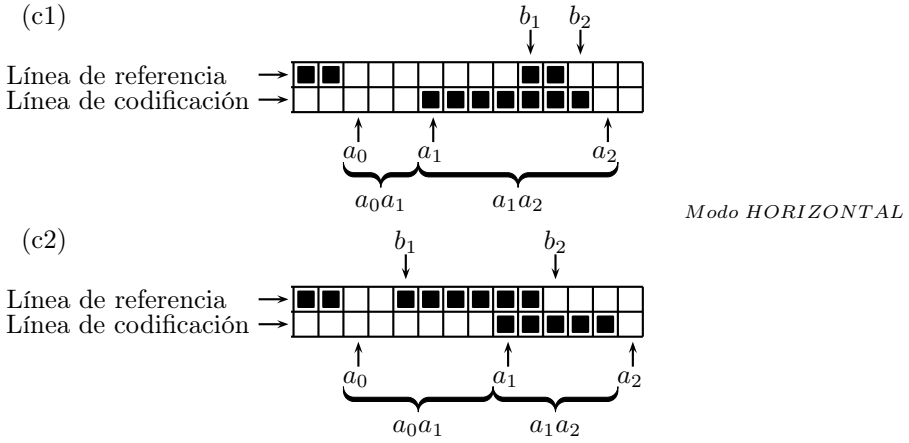
Después de identificar el modo actual y transmitir los códigos de acuerdo con la Tabla 2.42, se actualiza a_0 como muestra el diagrama de flujo[41] (Figura 2.44) y los otros cuatro punteros se modifican en relación al nuevo a_0. El proceso continúa hasta alcanzar el final de la línea de codificación. El codificador asume la existencia de un pel extra a la derecha de la línea, con un color opuesto al del último pel.



Racha b_1b_2 codificada. El nuevo a_0 se sitúa bajo el viejo b_2 (en a'_0).



Run length a_1b_1 codificado. El nuevo a_0 se sitúa en el viejo a_1.



a_0a_1 (blanco) y a_1a_2 (negro) codificados. El nuevo a_0 se sitúa en el viejo a_2.

Notas

(1) a_0 es el primer pel de una palabra de código nueva y puede ser blanco o negro.
(2) a_1 es el primer pel a la derecha de a_0 con un color diferente.
(3) a_2 es el primer pel a la derecha de a_1 con un color diferente.
(4) b_1 es el primer pel de la línea de referencia a la derecha de a_0 con distinto color.
(5) $b2$ es el primer pel de la línea de referencia a la derecha de b_1 con distinto color.

Figura 2.43: Cinco configuraciones de run-lengths. (a) Modo paso. (b) Modo vertical. (c) Modo horizontal.

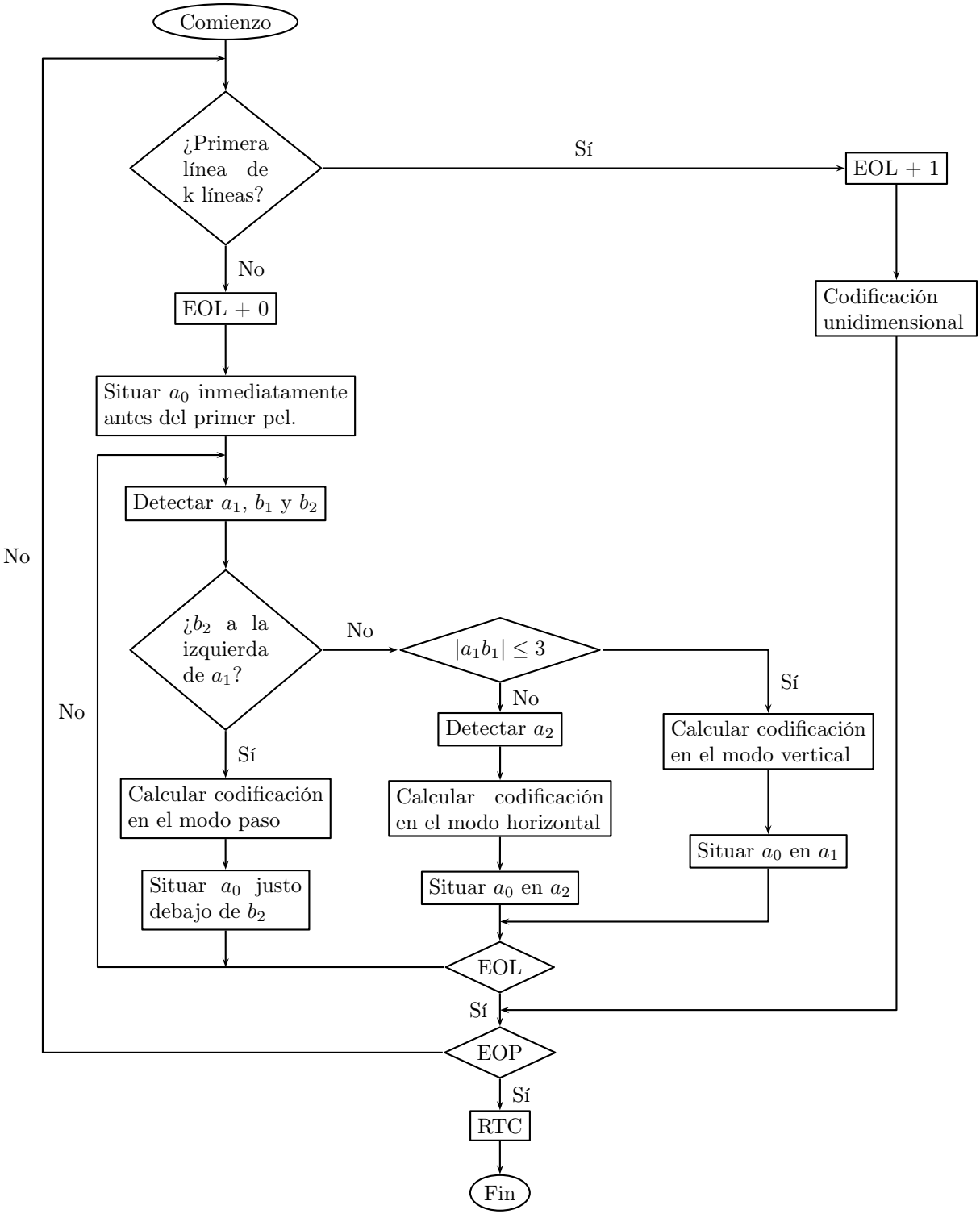


Figura 2.44: Diagrama de flujo para la codificación bidimensional (MMR).

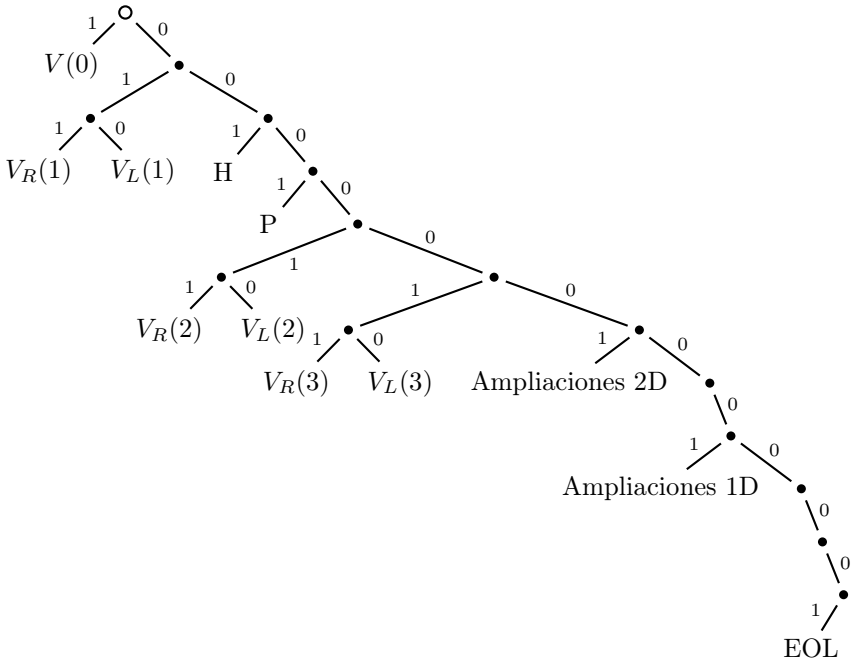


Figura 2.45: Árbol de códigos del Grupo 4.

Modo	**Elementos que se codifican**		**Notación**	**Palabra de código**
Paso	b_1, b_2		P	0001
Horizontal	a_0a_1, a_1a_2		H	001 + $M(a_0a_1)$ y $M(a_1a_2)$
Vertical	a_1 justo bajo b_1	$a_1b_1 = 0$	$V(0)$	1
	a_1 a la derecha de b_1	$a_1b_1 = 1$	$V_R(1)$	011
		$a_1b_1 = 2$	$V_R(2)$	000011
		$a_1b_1 = 3$	$V_R(3)$	0000011
	a_1 a la izquierda de b_1	$a_1b_1 = 1$	$V_L(1)$	010
		$a_1b_1 = 2$	$V_L(2)$	000010
		$a_1b_1 = 3$	$V_L(3)$	0000010
Ampliación	Bidimensional (2D)			0000001xxx
	Unidimensional (1D)			000000001xxx
EOL (fin de línea)				000000000001
Codificación 1D de línea siguiente				EOL+‘1’
Codificación 2D de línea siguiente				EOL+‘0’

Tabla 2.46: Códigos para el Grupo 4. (I)

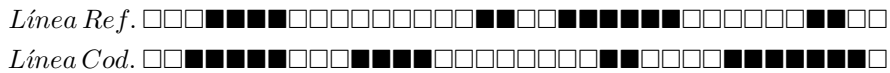
Código de entrada en el modo sin compresión	Línea codificada unidimensionalmente: 000000001111 Línea codificada bidimensionalmente: 0000001111	
	Configuración de la imagen	**Palabra de código**
	1	1
	01	01
Código de modo sin compresión	001	001
	0001	0001
	00001	00001
	00000	000001
		0000001T
	0	00000001T
Código de salida del modo sin compresión	00	000000001T
	000	0000000001T
	0000	00000000001T

T es un bit que define el color de la gama de repeticiones siguiente (negro=1, blanco=0).

Tabla 2.46: Tabla de código para el modo sin compresión. (II)

El código de ampliación de la Tabla 2.42 se usa para abortar prematuramente el proceso de codificación, antes de llegar al final de la página. Ésto es necesario si el resto de la página se transmite en un código diferente o incluso sin comprimir.

◇ **Ejercicio 2.33 (sol. en pág. 1061):** Averígüese manualmente el código generado a partir de las dos líneas siguientes:



Línea Ref. □□□■■■■□□□□□□□□□■■□□■■■■■■□□□□□□■■□□
Línea Cod. □□■■■■■□□□■■■■□□□□□□□□□■■□□□□■■■■■■■□

Las Tablas 2.46-I y 2.46-II resumen los códigos emitidos por el codificador del grupo 4. La Figura 2.45 es un árbol con los mismos códigos. Cada rama derecha corresponde a otro cero y cada rama izquierda, a otro 1.

> El trabajo en equipo es la capacidad de trabajar en grupo con una visión en común, incluso si esa visión se torna borrosa en extremo.
>
> —Anónimo

El código M() en el modo horizontal representa las palabras de código de las Tablas 2.41a, 2.41b-I y 2.41b-II. En la Tabla 2.46-II se indican los códigos para el modo sin compresión —cuando el valor de xxx es 111 (véase también la Tabla 2.46-I)—.

2.14. Codificación Aritmética

El método de Huffman es simple, eficiente y produce los mejores códigos para símbolos de datos individuales. Sin embargo, la Sección 2.8 muestra que el único caso donde produce códigos ideales de

[41] EOL(End Of Line) = Fin de línea; EOP(End Of Page)= Fin de página; RTC(Return to Control)= Retorno a control.

longitud variable (cuyo tamaño medio es igual a la entropía) es aquel cuyos símbolos tienen probabilidades de ocurrencia que son potencias negativas de 2 (i.e., números como 1/2, 1/4 ó 1/8). Esto se debe a que el método Huffman asigna un código con un número entero de bits a cada símbolo del alfabeto. La teoría de la información muestra que a un símbolo con una probabilidad de 0,4 podría idealmente asignársele un código de 1,32 bits, ya que $-\log_2 0{,}4 \approx 1{,}32$. El método de Huffman, sin embargo, normalmente le asigna un código de 1 ó 2 bits.

La codificación aritmética supera el problema de la asignación de códigos enteros a símbolos individuales mediante la asignación de un código (normalmente largo) al archivo de entrada completo. El método comienza con un cierto intervalo, lee el archivo origen símbolo a símbolo y utiliza la probabilidad de cada símbolo para reducir el intervalo. Especificando un intervalo más estrecho se requieren más bits, por lo que el número construido por el algoritmo crece continuamente. Para lograr compresión, el algoritmo está diseñado de tal manera que un símbolo de alta probabilidad reduce el intervalo menos que un símbolo de baja probabilidad, de modo que los símbolos con mayor probabilidad contribuyen con menos bits a la salida.

Un intervalo puede especificarse indicando sus límites inferior y superior o bien un límite y el ancho. Vamos a utilizar este último método para ilustrar cómo la especificación de un intervalo se hace más larga a medida que el intervalo se estrecha. El intervalo $[0, 1]$ se puede definir con los dos números de 1 bit, 0 y 1. El intervalo [0,1, 0,512], con los números 0,1 y 0,412. El muy estrecho intervalo [0,12575, 0,1257586] se especifica mediante los vastos números 0,12575 y 0,0000086.

La salida de la codificación aritmética se interpreta como un número perteneciente al intervalo [0,1). [La notación [a,b) significa: todos los números reales desde a hasta b, incluyendo el extremo a, pero no el b. El intervalo es "cerrado" en a y "abierto" en b.] Por lo tanto, el código 9746509 se interpreta como 0,9746509, aunque la parte "0." no se incluye en el archivo de salida.

Antes de sumergirnos en los detalles, veamos un poco de historia. El principio de la codificación aritmética fue propuesto por primera vez por Peter Elias hacia 1960 y fue descrito por primera vez en [Abramson 63]. Las primeras implementaciones prácticas de este método fueron desarrolladas por [Rissanen 76], [Pasco 76] y [Rubin 79]. Merecen una especial mención [Moffat et al. 98] y [Witten et al. 87]. Éstos tratan tanto los principios como los detalles prácticos de la codificación aritmética y muestran algunos ejemplos.

El primer paso es calcular, o al menos estimar, las frecuencias de ocurrencia de cada símbolo. Para obtener los mejores resultados, las frecuencias exactas se calculan leyendo el fichero completo en la primera fase de un trabajo de compresión de dos etapas. Sin embargo, si el programa puede obtener una buena estimación de las frecuencias de una fuente diferente, el primer paso puede omitirse.

El primer ejemplo involucra a los tres símbolos a_1, a_2 y a_3, con probabilidades $P_1 = 0{,}4$, $P_2 = 0{,}5$ y $P_3 = 0{,}1$, respectivamente. El intervalo [0,1) se divide entre los tres símbolos asignando a cada uno un subintervalo proporcional en tamaño a su probabilidad. El orden de los subintervalos es irrelevante. En nuestro ejemplo, a los tres símbolos se les han asignado los subintervalos [0,0,4), [0,4,0,9) y [0,9,1,0). Para codificar la cadena "$a_2\, a_2\, a_2\, a_3$", comenzamos con el intervalo [0,1). El primer símbolo a_2 reduce este intervalo al subintervalo que abarca desde el punto que representa su 40 % al correspondiente a su 90 %; el resultado es [0,4,0,9). El segundo a_2 reduce [0,4,0,9) de la misma manera (véase la nota de más abajo) a [0,6,0,85), el tercer a_2 reduce éste a [0,7,0,825) y a_3 reduce éste al tramo que abarca desde el punto que representa el 90 % de [0,7,0,825) al que indica su 100 %, produciendo como resultado [0,8125,0,8250). El código final generado por nuestro método puede ser cualquier número en este último intervalo.

(Nota: el subintervalo [0,6,0,85) se obtiene a partir el intervalo [0,4,0,9) realizando los siguientes cálculos: $0{,}4 + (0{,}9 - 0{,}4) \times 0{,}4 = 0{,}6$ y $0{,}4 + (0{,}9 - 0{,}4) \times 0{,}9 = 0{,}85$.)

Con este ejemplo en mente, debería ser fácil comprender las siguientes reglas, que resumen las principales etapas de la codificación aritmética:

1. Comenzar definiendo el "intervalo actual" como [0,1).

Carácter	Frecuencia	Probabilidad	Rango	Frecuencia acumulada
		Frec. acumulada total=		**10**
S	5	$5/10 = 0{,}5$	[0,5, 1,0)	5
W	1	$1/10 = 0{,}1$	[0,4, 0,5)	4
I	2	$2/10 = 0{,}2$	[0,2, 0,4)	2
M	1	$1/10 = 0{,}1$	[0,1, 0,2)	1
␣	1	$1/10 = 0{,}1$	[0,0, 0,1)	0

Tabla 2.47: Frecuencias y probabilidades de cinco símbolos.

2. Repetir los dos pasos siguientes para cada símbolo s de la secuencia de entrada:
   2.1. Dividir el intervalo actual en subintervalos cuyos tamaños sean proporcionales a probabilidades de los símbolos.
   2.2. Seleccionar el subintervalo para s y definirlo como el nuevo intervalo actual.

3. Cuando el flujo de datos de entrada se ha procesado completamente de este modo, debe generarse en la salida cualquier número que identifique el intervalo actual (i.e., cualquier número dentro del intervalo actual).

Por cada símbolo procesado, el intervalo actual se hace más pequeño, por lo que se necesitan más bits para expresarlo, pero el caso es que el resultado final es un solo número y no está compuesto por los códigos de los símbolos individuales. El tamaño medio del código puede obtenerse dividiendo el tamaño de la salida (en bits) por el tamaño de la entrada (en símbolos). Observe también que las probabilidades utilizadas en el paso 2.1 pueden cambiar continuamente, ya que pueden ser suministradas por un modelo de probabilidad adaptativo (Sección 2.15).

> Una teoría sólo tiene la alternativa de ser correcta o incorrecta. Un modelo tiene una tercera posibilidad: puede estar en lo cierto, pero ser irrelevante.
>
> —Eigen Manfred, The Physicist's Conception of Nature[a]
>
> [a]Concepto de física de la Naturaleza.

El siguiente ejemplo es un poco más complicado. Mostramos los pasos de la compresión para la corta cadena `SWISS␣MISS`. La Tabla 2.47 muestra la información preparada en la primera etapa (El *modelo estadístico* de los datos). Los cinco símbolos que aparecen en la entrada pueden encontrarse en cualquier orden. Para cada símbolo, el primer valor es su frecuencia, seguida de su probabilidad de ocurrencia (la frecuencia dividida por el tamaño de la cadena, 10). El rango [0,1) se divide entre los símbolos en cualquier orden, de forma que con cada símbolo se obtiene una parte del intervalo (o un subrango), igual en tamaño a su probabilidad. Así `S` produce el subintervalo [0,5,1,0) (de tamaño de 0,5), mientras que el subrango de `I` es [0,2,0,4) (de tamaño de 0,2). El algoritmo de decodificación de la página 126 utiliza la columna de frecuencias acumuladas.

Los símbolos y las frecuencias de la Tabla 2.47 se escriben en la secuencia de salida antes que cualquiera de los bits del código comprimido. Esta tabla será lo primero que obtendrá el decodificador.

El proceso de codificación comienza mediante la definición de dos variables —`Low` y `High`— y su inicialización a 0 y 1, respectivamente, para determinar el intervalo [`Low`,`High`). A medida que los símbolos se van introduciendo y procesado, los valores `Low` y `High` cambian juntos, para reducir el intervalo.

Después de procesar el primer símbolo —`S`—, `Low` y `High`, se actualizan a 0,5 y 1, respectivamente. El código resultante para la cadena de entrada completa será un número en este rango

Carácter		El cálculo de low y de high
S	L	$0{,}0 + (1{,}0 - 0{,}0) \times 0{,}5 = 0{,}5$
	H	$0{,}0 + (1{,}0 - 0{,}0) \times 1{,}0 = 1{,}0$
W	L	$0{,}5 + (1{,}0 - 0{,}5) \times 0{,}4 = 0{,}70$
	H	$0{,}5 + (1{,}0 - 0{,}5) \times 0{,}5 = 0{,}75$
I	L	$0{,}7 + (0{,}75 - 0{,}70) \times 0{,}2 = 0{,}71$
	H	$0{,}7 + (0{,}75 - 0{,}70) \times 0{,}4 = 0{,}72$
S	L	$0{,}71 + (0{,}72 - 0{,}71) \times 0{,}5 = 0{,}715$
	H	$0{,}71 + (0{,}72 - 0{,}71) \times 1{,}0 = 0{,}720$
S	L	$0{,}715 + (0{,}72 - 0{,}715) \times 0{,}5 = 0{,}7175$
	H	$0{,}715 + (0{,}72 - 0{,}715) \times 1{,}0 = 0{,}7200$
␣	L	$0{,}7175 + (0{,}72 - 0{,}7175) \times 0{,}0 = 0{,}71750$
	H	$0{,}7175 + (0{,}72 - 0{,}7175) \times 0{,}1 = 0{,}71775$
M	L	$0{,}7175 + (0{,}71775 - 0{,}7175) \times 0{,}1 = 0{,}717525$
	H	$0{,}7175 + (0{,}71775 - 0{,}7175) \times 0{,}2 = 0{,}717550$
I	L	$0{,}717525 + (0{,}71755 - 0{,}717525) \times 0{,}2 = 0{,}717530$
	H	$0{,}717525 + (0{,}71755 - 0{,}717525) \times 0{,}4 = 0{,}717535$
S	L	$0{,}717530 + (0{,}717535 - 0{,}717530) \times 0{,}5 = 0{,}7175325$
	H	$0{,}717530 + (0{,}717535 - 0{,}717530) \times 1{,}0 = 0{,}7175350$
S	L	$0{,}7175325 + (0{,}717535 - 0{,}7175325) \times 0{,}5 = 0{,}71753375$
	H	$0{,}7175325 + (0{,}717535 - 0{,}7175325) \times 1{,}0 = 0{,}71753500$

Tabla 2.48: El proceso de codificación aritmética.

($0{,}5 \leq$ *Código* $< 1{,}0$). El resto de la secuencia de entrada se determinará con precisión cuando, en el intervalo [0,5,1), el código final quede fijado. Una buena manera de comprender el proceso es imaginar que el nuevo intervalo [0,5,1) se divide entre los cinco símbolos de nuestro alfabeto utilizando las mismas proporciones que para el intervalo inicial [0,1). El resultado son los cinco subintervalos: [0,5,0,55), [0,55,0,60), [0,60,0,70), [0,70,0,75) y [0,75,1,0). Cuando se introduce el siguiente símbolo —`W`—, se selecciona el cuarto subintervalo y éste a su vez se divide en otros cinco.

A medida que se van introduciendo y procesando más símbolos de la entrada, `Low` y `High`, se van actualizando de acuerdo con:

```
NewHigh:= OldLow + Range*HighRange(X);
NewLow:= OldLow + Range*LowRange(X);
```

donde `Range=OldHigh-OldLow` y `LowRange(X)`, `HighRange(X)` indican los límites inferior y superior del rango correspondiente al símbolo `X`, respectivamente. En el ejemplo anterior, el segundo símbolo de la entrada es `W`, por lo que `Low` y `High` pasan a tener los valores `Low` $:= 0{,}5 + (1{,}0 - 0{,}5) \times 0{,}4 = 0{,}70$ y `High` $:= 0{,}5 + (1{,}0 - 0{,}5) \times 0{,}5 = 0{,}75$. El nuevo intervalo [0,70,0,75) abarca el tramo [40 %,50 %) del subrango de `S`. La Tabla 2.48 muestra todos los pasos involucrados en la codificación de la cadena $SWISS_{\sqcup}MISS$ (Los tres primeros pasos se ilustran gráficamente en la Figura 2.61a). El código final es el valor final de `Low` —0,71753375—, del que sólo necesitan ser escritos en la secuencia de salida los ocho dígitos 71753375 (véase más adelante una modificación de esta declaración).

El decodificador funciona a la inversa. Comienza introduciendo los símbolos y sus rangos y reconstruye la Tabla 2.47. A continuación, introduce el resto del código. El primer dígito es el 7, por lo que el decodificador sabe de inmediato que todo el código está integrado en un número de la forma 0,7.... Este número se encuentra dentro del subrango [0,5,1) de `S`, por lo tanto, el primer símbolo es

```
lowRange={0.998162,0.023162,0.};
highRange={1.,0.998162,0.023162};
low=0.; high=1.;
enc[i_]:=Module[{nlow,nhigh,range},
range=high-low;
nhigh=low+range highRange[[i]];
nlow=low+range lowRange[[i]];
low=nlow; high=nhigh;
Print["r=",N[range,25]," l=",N[low,17]," h=",N[high,17]]]
enc[2]
enc[2]
enc[1]
enc[3]
enc[3]
```

Figura 2.49: Código en *Mathematica* para la Tabla 2.52

Carácter	**Código - low**		**Rango**
S	0,71753375–0,5	= 0,21753375	/0,5 = 0,4350675
W	0,4350675–0,4	= 0,0350675	/0,1 = 0,350675
I	0,350675–0,2	= 0,150675	/0,2 = 0,753375
S	0,753375–0,5	= 0,253375	/0,5 = 0,50675
S	0,50675–0,5	= 0,00675	/0,5 = 0,0135
␣	0,0135–0	= 0,0135	/0,1 = 0,135
M	0,135–0,1	= 0,035	/0,1 = 0,35
I	0,35–0,2	= 0,15	/0,2 = 0,75
S	0,75–0,5	= 0,25	/0,5 = 0,5
S	0,5–0,5	= 0	/0,5 = 0

Tabla 2.50: El proceso de decodificación aritmética.

S. Después, el decodificador elimina el efecto del símbolo S en el código restando el límite inferior 0,5 de S y dividiendo por la amplitud del subrango de S (0,5). El resultado es 0,4350675, que le dice al decodificador que el siguiente símbolo es W (ya que el subrango de W es [0,4,0,5)).

Para eliminar el efecto del símbolo X del código, el decodificador realiza la operación `Code:= (Code-LowRange(X))/Range`, donde `Range` es el ancho del subrango de X. La Tabla 2.50 resume los pasos para la decodificación de nuestra cadena de ejemplo (nótese que tiene dos filas por símbolo).

El siguiente ejemplo es de tres símbolos con las probabilidades que se muestran en la Tabla 2.51a. Observe que las probabilidades son muy diferentes. Una es grande (97.5 %) y las otras mucho más pequeñas. Este es un caso de *probabilidades sesgadas*[42].

La codificación de la cadena $a_2\,a_2\,a_1\,a_3\,a_3$ produce los extraños números (precisión de 16 dígitos) de la Tabla 2.52, donde las dos filas para cada símbolo corresponden a los valores de `Low` y `High`, respectivamente. La Figura 2.49 muestra el código en *Mathematica* que calcula la tabla.

A primera vista, parece que el código resultante es más largo que la cadena original, sin embargo, en la Sección 2.14.3 se muestra cómo obtener la compresión real alcanzada por la codificación aritmética.

La decodificación de esta cadena se muestra en la Tabla 2.53 y produce un problema especial. Después de eliminar el efecto de a_1 en la línea 3, el resultado es 0. Anteriormente, asumimos implícitamente que esto significaba el fin del proceso de decodificación, pero ahora sabemos que hay dos

[42]Skewed probabilities.

Car.	Prob.	Rango
a_1	0,001838	[0,998162, 1,0)
a_2	0,975	[0,023162, 0,998162)
a_3	0,023162	[0,0, 0,023162)

(a)

Car.	Prob.	Rango
eof	0,000001	[0,999999,1,0)
a_1	0,001837	[0,998162, 0,999999)
a_2	0,975	[0,023162, 0,998162)
a_3	0,023162	[0,0, 0,023162)

(b)

Tabla 2.51: Probabilidades (sesgadas) de tres símbolos.

a_2	$0{,}0 + (1{,}0 - 0{,}0) \times 0{,}023162$	$= 0{,}023162$
	$0{,}0 + (1{,}0 - 0{,}0) \times 0{,}998162$	$= 0{,}998162$
a_2	$0{,}023162 + {,}975 \times 0{,}023162$	$= 0{,}04574495$
	$0{,}023162 + {,}975 \times 0{,}998162$	$= 0{,}99636995$
a_1	$0{,}04574495 + 0{,}950625 \times 0{,}998162$	$= 0{,}99462270125$
	$0{,}04574495 + 0{,}950625 \times 1{,}0$	$= 0{,}99636995$
a_3	$0{,}99462270125 + 0{,}00174724875 \times 0{,}0$	$= 0{,}99462270125$
	$0{,}99462270125 + 0{,}00174724875 \times 0{,}023162$	$= 0{,}994663171025547$
a_3	$0{,}99462270125 + 0{,}00004046977554749998 \times 0{,}0$	$= 0{,}99462270125$
	$0{,}99462270125 + 0{,}00004046977554749998 \times 0{,}023162$	$= 0{,}994623638610941$

Tabla 2.52: Codificación de la cadena $a_2\, a_2\, a_1\, a_3\, a_3$.

ocurrencias más de a_3 que deben ser decodificadas. Éstas se muestran en las líneas 4 y 5 de la tabla. Este problema siempre se produce cuando el último símbolo de la secuencia de entrada es el subrango que comienza por cero. Con el fin de poder distinguir entre un símbolo y el final de la secuencia de entrada, tenemos que definir un símbolo adicional: *end-of-input*[43] (o *end-of-file*[44], *eof*). Este símbolo debe añadirse, con una probabilidad pequeña, a la tabla de frecuencias, como se puede ver en la Tabla 2.51b y debe ser codificado al final de la secuencia de entrada.

Las Tablas 2.54 y 2.55 muestran cómo el número 0,0000002878086184764172 es la codificación de la cadena $a_3\, a_3\, a_3\, a_3\, eof$ y luego se decodifica correctamente. Sin el símbolo eof, una cadena compuesta sólo por símbolos a_3 se habría codificado como un 0.

Observe cómo el valor de `low` es 0 hasta que *eof* se introduce y procesa, y cómo el valor `high` se aproxima rápidamente a 0. Ahora es el momento de mencionar que el código final no tiene que ser el valor final de `low`, sino que puede ser cualquier número entre los valores finales de `low` y `high`. En el ejemplo de $a_3\, a_3\, a_3\, a_3\, eof$, el código final puede ser el número (mucho más corto) 0,0000002878086 (ó

[43] Fin de entrada.
[44] Fin de archivo.

Car.	Código - low		Rango	
a_2	$0{,}99462270125 - 0{,}023162$	$= 0{,}97146070125$	$/0{,}975$	$= 0{,}99636995$
a_2	$0{,}99636995 - 0{,}023162$	$= 0{,}97320795$	$/0{,}975$	$= 0{,}998162$
a_1	$0{,}998162 - 0{,}998162$	$= 0{,}0$	$/0{,}00138$	$= 0{,}0$
a_3	$0{,}0 - 0{,}0$	$= 0{,}0$	$/0{,}023162$	$= 0{,}0$
a_3	$0{,}0 - 0{,}0$	$= 0{,}0$	$/0{,}023162$	$= 0{,}0$

Tabla 2.53: Decodificación de la cadena $a_2\, a_2\, a_1\, a_3\, a_3$.

a_3	$0{,}0 + (1{,}0 - 0{,}0) \times 0{,}0$	$= 0{,}0$
	$0{,}0 + (1{,}0 - 0{,}0) \times 0{,}023162$	$= 0{,}023162$
a_3	$0{,}0 + {,}023162 \times 0{,}0$	$= 0{,}0$
	$0{,}0 + {,}023162 \times 0{,}023162$	$= 0{,}000536478244$
a_3	$0{,}0 + 0{,}000536478244 \times 0{,}0$	$= 0{,}0$
	$0{,}0 + 0{,}000536478244 \times 0{,}023162$	$= 0{,}000012425909087528$
a_3	$0{,}0 + 0{,}000012425909087528 \times 0{,}0$	$= 0{,}0$
	$0{,}0 + 0{,}000012425909087528 \times 0{,}023162$	$= 0{,}0000002878089062853235$
eof	$0{,}0 + 0{,}0000002878089062853235 \times 0{,}999999$	$= 0{,}0000002878086184764172$
	$0{,}0 + 0{,}0000002878089062853235 \times 1{,}0$	$= 0{,}0000002878089062853235$

Tabla 2.54: Codificación de la cadena $a_3\, a_3\, a_3\, a_3\, eof$.

Car.	**Código - low**	**Rango**
a_3	$0{,}0000002878086184764172 - 0$	
	$= 0{,}0000002878086184764172$	$/0{,}023162 = 0{,}0000124258966616189102 8$
a_3	$0{,}0000124258966616189102 8 - 0$	
	$= 0{,}0000124258966616189102 8$	$/0{,}023162 = 0{,}000536477707521756$
a_3	$0{,}000536477707521756 - 0$	
	$= 0{,}000536477707521756$	$/0{,}023162 = 0{,}023161976838$
a_3	$0{,}023161976838 - 0$	
	$= 0{,}023161976838$	$/0{,}023162 = 0{,}999999$
eof	$0{,}999999 - 0{,}999999$	
	$= 0{,}0$	$/0{,}000001 = 0{,}0$

Tabla 2.55: Decodificación de la cadena $a_3\, a_3\, a_3\, a_3\, eof$.

0,0000002878087 ó incluso 0,0000002878088).

◇ **Ejercicio 2.34 (sol. en pág. 1061):** Codifíquese la cadena $a_2\, a_2\, a_2\, a_2$ y resúmanse los resultados en una tabla similar la Tabla 2.54. ¿Cómo difieren los resultados de los obtenidos con la cadena $a_3\, a_3\, a_3\, a_3$?

Si se conoce el tamaño de la secuencia de entrada, es posible prescindir de un símbolo *eof*. El codificador puede comenzar escribiendo este tamaño (sin codificar) en la salida. El decodificador lee el tamaño, empieza a decodificar y se detiene cuando la cadena decodificada llega a este tamaño. Si el decodificador lee la cadena comprimida byte a byte, el codificador puede tener que añadir algunos ceros al final, para asegurarse de que la cadena comprimida se pueda leer en grupos de 8 bits.

2.14.1. Detalles de la implementación

El proceso de codificación descrito anteriormente no es práctico, ya que asume que los números de precisión ilimitada pueden ser almacenados en `Low` y `High`. El proceso de decodificación descrito en la página 124 ("Después, el decodificador elimina el efecto del símbolo `S` en el código restando... y dividiendo...") es simple —en principio—, pero también impracticable. El código, que es un único

número, es normalmente largo y puede llegar a ser extremadamente largo. Un archivo de 1 MByte puede ser codificado en, digamos, un archivo de 500 Kbytes, que consiste en un solo número. La división en la que algún operando ocupa 500 Kbytes es compleja y lenta.

Cualquier aplicación práctica de la codificación aritmética debe utilizar sólo números enteros (porque la aritmética de punto flotante es lenta y la precisión se pierde) y no deben ser muy largos (preferiblemente números de simple precisión). Describimos una implementación práctica aquí, usando dos variables enteras `Low` y `High`. En nuestro ejemplo se usan sólo cuatro dígitos decimales, pero en la práctica podrían ser de 16 ó 32 bits de longitud. Estas variables contienen los límites bajo (`Low`) y alto (`High`) del subintervalo actual, pero no hay que dejarlas crecer demasiado. Una mirada a la Tabla 2.48 muestra que una vez que los dígitos de más a la izquierda de `Low` y `High` son idénticos, nunca cambian. Por lo tanto, podemos sacar esos dígitos de las dos variables y escribirlos como un número en la secuencia de salida. De esta manera, las dos variables no tienen que almacenar el código completo, sino sólo la parte más reciente del mismo. A medida que los dígitos se desplazan fuera de las dos variables, se añade un cero en el extremo derecho de `Low` y un 9 en el extremo derecho de `High`. Una buena manera de entender esto es pensar en cada una de las dos variables como el extremo izquierdo de un número infinitamente largo. `Low` contiene $xxxx00\ldots$ y `High` es igual a $yyyy99\ldots$.

Un problema es que `High` debe ser inicializado a 1, pero tanto el contenido de `Low` como el de `High` debe ser interpretado como fracciones menones que 1. La solución es inicializar `High` a 9999..., ya que la fracción infinita 0,999... es igual a 1.

(Esto es fácil de probar: Si $0{,}999\ldots < 1$, entonces su promedio $a = {}^{(1+0{,}999\ldots)}/_{2}$ debe ser un número entre $0{,}999\ldots$ y 1, pero no hay manera de escribir a. Es imposible darle más dígitos que a $0{,}999\ldots$, ya que éste ya tiene un número infinito de dígitos; y es imposible que los dígitos sean más grandes, pues todos son nueves. Es por ello que la fracción infinita $0{,}999\ldots$ debe ser igual a 1.)

◇ **Ejercicio 2.35 (sol. en pág. 1061):** Escríbase el número 0,5 en binario.

La Tabla 2.56 describe el proceso de codificación de la cadena `SWISS␣MISS`. La columna 1 muestra el siguiente símbolo de la entrada. La columna 2 muestra los nuevos valores de `Low` y `High`. La columna 3 muestra estos valores como números enteros a escala, después de haber decrementado `High` en 1. La columna 4 muestra el siguiente dígito enviado a la secuencia de salida. La columna 5 muestra los nuevos valores de `Low` y `High` después de hacer el desplazamiento a la izquierda. Observe cómo el último paso envía los cuatro dígitos 3750 a la cadena de salida. El resultado final es 717533750.

Para la decodificación se sigue el proceso inverso a la codificación. Comenzamos con `Low`=0000, `High`=9999 y `Code`=7175 (el código empieza siendo los primeros cuatro dígitos de la cadena comprimida). Estas variables se actualizan en cada paso del bucle de decodificación. `Low` y `High` se aproximan entre sí (y cada una de ellas aproxima `Code`) hasta que sus dígitos más significativos son los mismos. Luego se desplazan hacia la izquierda, lo cual las separa de nuevo y en ese instante `Code` también sufre ese desplazamiento. Se calcula un índice en cada paso que se utiliza para buscar en columna de frecuencias acumuladas de la Tabla 2.47 el elemento adecuado y así averiguar el símbolo actual.

Cada iteración del bucle se compone de los siguientes pasos:

1. Calcular `index:=((Code-Low+1)x10-1)/(High-Low+1)` y truncarlo con una precisión de entero. (En nuestro ejemplo, el número 10 es la frecuencia total acumulada.)

2. Utilizar el índice `index` para encontrar el siguiente símbolo comparándo con la columna de las frecuencias acumuladas de la tabla 2.47. En el ejemplo de más abajo, el primer valor de `index` es 7,1759, truncado a 7. Siete se encuentra entre 5 y 10 en la tabla, por lo que se selecciona la `S`.

3. Actualizar `Low` y `High` de acuerdo con

```
Low:=Low+(High-Low+1)LowCumFreq[X]/10;
High:=Low+(High-Low+1)HighCumFreq[X]/10-1;
```

1	2	3	4	5
S	$L = 0 + (1-0) \times 0{,}5 = 0{,}5$	5000		5000
	$H = 0 + (1-0) \times 1{,}0 = 1{,}0$	9999		9999
W	$L = 0{,}5 + (1-0{,}5) \times 0{,}4 = 0{,}7$	7000	7	0000
	$H = 0{,}5 + (1-0{,}5) \times 0{,}5 = 0{,}75$	7499	7	4999
I	$L = 0 + (0{,}5-0) \times 0{,}2 = 0{,}1$	1000	1	0000
	$H = 0 + (0{,}5-0) \times 0{,}4 = 0{,}2$	1999	1	9999
S	$L = 0 + (1-0) \times 0{,}5 = 0{,}5$	5000		5000
	$H = 0 + (1-0) \times 1{,}0 = 1{,}0$	9999		9999
S	$L = 0{,}5 + (1-0{,}5) \times 0{,}5 = 0{,}75$	7500		7500
	$H = 0{,}5 + (1-0{,}5) \times 1{,}0 = 1{,}0$	9999		9999
␣	$L = 0{,}75 + (1-0{,}75) \times 0{,}0 = 0{,}75$	7500	7	5000
	$H = 0{,}75 + (1-0{,}75) \times 0{,}1 = 0{,}775$	7749	7	7499
M	$L = 0{,}5 + (0{,}75-0{,}5) \times 0{,}1 = 0{,}525$	5250	5	2500
	$H = 0{,}5 + (0{,}75-0{,}5) \times 0{,}2 = 0{,}55$	5499	5	4999
I	$L = 0{,}25 + (0{,}5-0{,}25) \times 0{,}2 = 0{,}3$	3000	3	0000
	$H = 0{,}25 + (0{,}5-0{,}25) \times 0{,}4 = 0{,}35$	3499	3	4999
S	$L = 0 + (0{,}5-0) \times 0{,}5 = 0{,}25$	2500		2500
	$H = 0 + (0{,}5-0) \times 1{,}0 = 0{,}5$	4999		4999
S	$L = 0{,}25 + (0{,}5-0{,}25) \times 0{,}5 = 0{,}375$	3750	3750	
	$H = 0{,}25 + (0{,}5-0{,}25) \times 1{,}0 = 0{,}5$	4999		4999

Tabla 2.56: Codificación de SWISS␣MISS por desplazamiento.

donde `LowCumFreq[X]` y `HighCumFreq[X]` son las frecuencias acumuladas del símbolo `X` y del símbolo situado encima de él en la Tabla 2.47.

4. Si los dígitos del extremo izquierdo de `Low` y `High` son idénticos, se desplazan `Low`, `High` y `Code` una posición a la izquierda. `Low` añade un 0 a su derecha, `High` un 9 y `Code` el siguiente dígito a procesar de la cadena comprimida.

Aquí están todos los pasos de la decodificación de nuestro ejemplo:

1. Se proporciona a las variables los valores iniciales: `Low`=0000, `High`=9999 y `Code`=7175.

2. `index`= $[(7175-0+1) \times 10-1]/(9999-0+1) = 7{,}1759 \longrightarrow 7$.
   Se selecciona el símbolo `S`.
   `Low` $= 0 + (9999 - 0 + 1) \times 5/10 = 5000$.
   `High` $= 0 + (9999 - 0 + 1) \times 10/10 - 1 = 9999$.

3. `index`= $[(7175-5000+1) \times 10-1]/(9999-5000+1) = 4{,}3518 \longrightarrow 4$.
   Se selecciona el símbolo `W`.
   `Low` $= 5000 + (9999-5000+1) \times 4/10 = 7000$.
   `High` $= 5000 + (9999-5000+1) \times 5/10-1 = 7499$.
   Después de sacar el 7, tenemos: `Low`=000**0**, `High`=499**9** y `Code`=175**3**.

4. `index`= $[(1753-0+1) \times 10-1]/(4999-0+1) = 3{,}5078 \longrightarrow 3$.
   Se selecciona el símbolo `I`.
   `Low` $= 0 + (4999-0+1) \times 2/10 = 1000$.

`High` = 0 + (4999−0 + 1) × 4/10−1 = 1999.
Después de sacar el 1, tenemos: `Low`=000**0**, `High`=999**9** y `Code`=753**3**.

5. `index`= [(7533−0 + 1) × 10−1]/(9999−0 + 1) = 7,5339 ⟶ 7.
   Se selecciona el símbolo `S`.
   `Low` = 0 + (9999−0 + 1) × 5/10 = 5000.
   `High` = 0 + (9999−0 + 1) × 10/10−1 = 9999.

6. `index`= [(7533−5000 + 1) × 10−1]/(9999−5000 + 1) = 5,0678 ⟶ 5.
   Se selecciona el símbolo `S`.
   `Low` = 5000 + (9999−5000 + 1) × 5/10 = 7500.
   `High` = 5000 + (9999−5000 + 1) × 10/10−1 = 9999.

7. `index`= [(7533−7500 + 1) × 10−1]/(9999−7500 + 1) = 0,1356 ⟶ 0.
   Se selecciona el símbolo ␣.
   `Low` = 7500 + (9999−7500 + 1) × 0/10 = 7500.
   `High` = 7500 + (9999−7500 + 1) × 1/10−1 = 7749.
   Después de sacar el 7, tenemos: `Low`=500**0**, `High`=749**9** y `Code`=533**7**.

8. `index`= [(5337−5000 + 1) × 10−1]/(7499−5000 + 1) = 1,3516 ⟶ 1.
   Se selecciona el símbolo `M`.
   `Low` = 5000 + (7499−5000 + 1) × 1/10 = 5250.
   `High` = 5000 + (7499−5000 + 1) × 2/10−1 = 5499.
   Después de sacar el 5, tenemos: `Low`=250**0**, `High`=499**9** y `Code`=337**5**.

9. `index`= [(3375−2500 + 1) × 10−1]/(4999−2500 + 1) = 3,5036 ⟶ 3.
   Se selecciona el símbolo `I`.
   `Low` = 2500 + (4999−2500 + 1) × 2/10 = 3000.
   `High` = 2500 + (4999−2500 + 1) × 4/10−1 = 3499.
   Después de sacar el 3, tenemos: `Low`=000**0**, `High`=499**9** y `Code`=375**0**.

10. `index`= [(3750−0 + 1) × 10−1]/(4999−0 + 1) = 7,5018 ⟶ 7.
    Se selecciona el símbolo `S`.
    `Low` = 0 + (4999−0 + 1) × 5/10 = 2500.
    `High` = 0 + (4999−0 + 1) × 10/10−1 = 4999.

11. `index`= [(3750−2500 + 1) × 10−1]/(4999−2500 + 1) = 5,0036 ⟶ 5.
    Se selecciona el símbolo `S`.
    `Low` = 2500 + (4999−2500 + 1) × 5/10 = 3750.
    `High` = 2500 + (4999−2500 + 1) × 10/10−1 = 4999.

◇ **Ejercicio 2.36 (sol. en pág. 1061):** ¿Cómo sabe el decodificador cuándo detener el bucle en este punto?

2.14.2. Underflow[45]

La Tabla 2.57 muestra los pasos de la codificación de la cadena $a_3\,a_3\,a_3\,a_3\,a_3$ por desplazamiento. Esta tabla es similar a la Tabla 2.56 e ilustra el problema del *underflow*. `Low` y `High` se aproximan entre sí y como `Low` es siempre 0 en este ejemplo, `High` pierde sus dígitos significativos a medida que se aproxima a `Low`.

[45] Desbordamiento inferior.

1	2	3	4	5
1	$L = 0 + (1{-}0) \times 0{,}0 = 0{,}0$	000000	0	000000
	$H = 0 + (1{-}0) \times 0{,}023162 = 0{,}023162$	023162	0	231629
2	$L = 0 + (0{,}231629{-}0) \times 0{,}0 = 0{,}0$	000000	0	000000
	$H = 0 + (0{,}231629{-}0) \times 0{,}023162 = 0{,}00536499089$	005364	0	053649
3	$L = 0 + (0{,}053649{-}0) \times 0{,}0 = 0{,}0$	000000	0	000000
	$H = 0 + (0{,}053649{-}0) \times 0{,}023162 = 0{,}00124261813$	001242	0	012429
4	$L = 0 + (0{,}012429{-}0) \times 0{,}0 = 0{,}0$	000000	0	000000
	$H = 0 + (0{,}012429{-}0) \times 0{,}023162 = 0{,}00028788049$	000287	0	002879
5	$L = 0 + (0{,}002879{-}0) \times 0{,}0 = 0{,}0$	000000	0	000000
	$H = 0 + (0{,}002879{-}0) \times 0{,}023162 = 0{,}00006668339$	000066	0	000669

Tabla 2.57: Codificación de la cadena $a_3\,a_3\,a_3\,a_3\,a_3$ por desplazamiento.

El *underflow* puede ocurrir no sólo en este caso, sino en cualquier otro en el que `Low` y `High` necesiten converger estrechamente. Debido al tamaño finito de las variables `Low` y `High`, éstas pueden obtener valores de digamos, 499996 y 500003; desde ahí, en lugar de llegar a cifras en las que los dígitos más significativos sean idénticos, alcanzan valores de 499999 y 500000. Debido a que los dígitos más significativos son diferentes, el algoritmo no saca nada, no va a hacer desplazamientos y la siguiente iteración sólo añadirá dígitos más allá de los seis primeros; dígitos que se perderán, manteniendo invariables los seis primeros. El algoritmo hará una iteración sin generar ningún resultado hasta que llegue a *eof*.

La solución a este problema consiste en detectar tales casos lo antes posible y *reescalar* ambas variables. En el ejemplo anterior, este ajuste debe realizarse cuando las dos variables alcanzan valores de $49xxxx$ y $50yyyy$. La operación debe excluir los segundos dígitos más significativos, finalizar con $4xxxx0$ y $5yyyy9$ e incrementar un contador `cntr`. El algoritmo puede necesitar reescalar varias veces antes de que los dígitos más significativos sean iguales. En ese punto, el dígito más significativo (que puede ser 4 ó 5) debe sacarse, seguido por `cntr` ceros (si las dos variables convergen a 4) o nueves (si convergen a 5).

2.14.3. Comentarios Finales

Todos los ejemplos vistos hasta ahora han utilizado números en decimal, ya que los cálculos son más fáciles de comprender en esta base numérica. Resulta que todos los algoritmos y las reglas descritas anteriormente se pueden aplicar al caso binario con sólo hacer un cambio: Cada aparición de un 9 (el dígito decimal más grande) debe ser sustituida por un 1 (el dígito binario más grande).

Los ejemplos anteriores aparentemente no ofrecen ninguna compresión. Aparentemente, las tres cadenas de ejemplo (`SWISS␣MISS`, $a_2\,a_2\,a_1\,a_3\,a_3$ y $a_3\,a_3\,a_3\,a_3\,eof$) se codifican en un número excesivamente largo. De hecho, parece que la longitud del código final depende de la probabilidades involucradas. Las probabilidades mayores de la Tabla 2.51a generan números largos en el proceso de codificación, mientras que las probabilidades menores de la Tabla 2.47 se traducen en unos valores más razonables para `Low` y `High`, indicados en la Tabla 2.48. Este comportamiento exige una explicación.

> Me da vergüenza decirle cuántas cifras me llevaron estos cálculos, al no tener otro asunto en ese momento.
>
> —Isaac Newton

Para averiguar el grado de compresión que alcanza la codificación aritmética, tenemos que considerar dos hechos:

1. En la práctica, todas las operaciones se realizan con números binarios, por lo que tenemos que traducir los resultados finales a binario antes de poder estimar la eficiencia de la compresión.

2. Puesto que el último símbolo codificado es *eof*, el código final no tiene que ser el último cálculo de `Low`; puede ser cualquier valor entre `Low` y `High`. Esto hace posible seleccionar un número más corto que el del código final que se está sacando.

La Tabla 2.48 codifica la cadena `SWISS␣MISS` en los valores finales para `Low` y `High` de 0,71753375 y 0,717535, respectivamente. Los valores aproximados[46] de estos números binarios son:

$$0.\mathbf{10110111101100000010}10101011 \; y \; 0.\mathbf{10110111101100000010}11111011,$$

por lo que se puede seleccionar el número **10110111101100000010**0 como elección final de secuencia de salida comprimida. La cadena de diez símbolos se codifica así en un número de 20 bits. ¿Representa esto una buena compresión?

La respuesta es sí. Utilizando las probabilidades de la Tabla 2.47, es fácil calcular la probabilidad de la cadena `SWISS␣MISS`. Ésta es $P = 0{,}5^5 \times 0{,}1 \times 0{,}2^2 \times 0{,}1 \times 0{,}1 = 1{,}25 \times 10^{-6}$. La entropía de esta cadena es, por lo tanto: $-\log_2 P = 19{,}6096$. Como consecuencia, el mínimo número de bits necesarios —en la práctica— para codificar la cadena son veinte.

Los símbolos de la Tabla 2.51a tienen probabilidades de 0,975, 0,001838 y 0,023162. Estos números requieren bastantes dígitos decimales y como resultado, los valores finales de `Low` y `High` de la Tabla 2.52 son los números 0,99462270125 y 0,994623638610941. Una vez más parece que no hay compresión, pero un análisis similar al realizado anteriormente muestra que la compresión está muy cerca de la entropía.

La probabilidad de la cadena $a_2\,a_2\,a_1\,a_3\,a_3$ es $0{,}975^2 \times 0{,}001838 \times 0{,}023162^2 \approx 9{,}37361 \times 10^{-7}$, y $-\log_2 9{,}37361 \times 10^{-7} \approx 20{,}0249$.

Las representaciones binarias[47] de los valores finales de `Low` y `High` de la Tabla 2.52 son:

$$0.\mathbf{11111110100111111}0010111111001 \; y \; 0.\mathbf{11111110100111111}0100111101,$$

respectivamente. Seleccionamos el número de 19 bits **1111111010011111110**, ya que podemos elegir cualquier número entre estos dos. (Debería haber sido un número de 21 bits, pero los números de la Tabla 2.52 tienen una precisión limitada y no son exactos.)

◇ **Ejercicio 2.37 (sol. en pág. 1066):** Dados los tres símbolos a_1, a_2 y *eof*, con probabilidades $P_1 = 0{,}4$, $P_2 = 0{,}5$ y $P_{eof} = 0{,}1$, codifíquese la cadena $a_2\,a_2\,a_2\,eof$ y compruébese que el tamaño del código final es igual al mínimo (práctico).

El siguiente argumento muestra por qué la codificación aritmética puede, en principio, ser un método de compresión muy eficiente. Denotamos s a una secuencia de símbolos a codificar y b el número de bits necesarios para codificarlos. A medida que s se alarga, su probabilidad $P(s)$ se hace más pequeña y b se hace más grande. Dado que el logaritmo es la función que nos proporciona la cantidad de información, es fácil ver que b debe crecer al mismo ritmo, que $\log_2 P(s)$ se reduce. Su producto debe, por lo tanto, ser constante o casi constante. La teoría de la información nos dice que b y $P(s)$ satisfacen la doble desigualdad:

$$2 \le 2^b P(s) < 4,$$

que implica:

$$1 - \log_2 P(s) \le b < 2 - \log_2 P(s). \tag{2.6}$$

[46] La última cifra decimal está redondeada.
[47] La última cifra decimal está redondeada.

A medida que s es más larga, su probabilidad $P(s)$ disminuye, la cantidad $-\log_2 P(s)$ se convierte en un número positivo más grande; y la doble desigualdad de la ecuación (2.6) muestra que en el límite, b se aproxima a $-\log_2 P(s)$. Ésta es la razón por la cual la codificación aritmética puede, en principio, comprimir una cadena de símbolos a su límite teórico.

Para obtener más información sobre este tema, véanse [Moffat et al. 98] y [Witten et al. 87].

2.15. Codificación aritmética adaptativa

Hay dos características de la codificación aritmética que son fáciles de extender:

1. Uno de los principales pasos de la codificación (pág. 123) actualiza `NewLow` y `NewHigh`. Del mismo modo, uno de los principales pasos de la decodificación (paso 3, en la página 127) actualiza `Low` y `High` de acuerdo con:

   ```
   Low:=Low+(High-Low+1)LowCumFreq[X]/10;
   High:=Low+(High-Low+1)HighCumFreq[X]/10-1;
   ```

   Esto significa que con el fin de codificar símbolo `X`, el codificador debe tener las frecuencias acumuladas del símbolo y del que tenga encima (véase la Tabla 2.47 como ejemplo de frecuencias acumuladas). Esto también implica que la frecuencia de `X` (o, equivalentemente, su probabilidad) puede cambiarse cada vez que se codifica, siempre que el codificador y el decodificador estén de acuerdo en cómo hacerlo.

2. El orden de los símbolos en la Tabla 2.47 no es importante. Incluso se pueden intercambiar en la tabla durante el proceso de codificación, siempre y cuando el codificador y decodificador lo hagan de la misma manera.

Con esto en mente, es fácil entender cómo funciona la codificación aritmética adaptativa. El algoritmo de codificación tiene dos partes: el modelo de probabilidad y el codificador aritmético. El modelo lee el símbolo de la cadena de entrada e invoca al codificador, pasándole el símbolo y las dos frecuencias acumuladas necesarias. A continuación, el modelo incrementa el contador de símbolos y actualiza las frecuencias acumuladas. La cuestión es que sea el modelo el que determine la probabilidad del símbolo de acuerdo con su *antiguo* recuento e incremente el contador sólo después de que el símbolo se haya codificado. Esto hace posible que el decodificador pueda reflejar las operaciones del codificador. El codificador sabe qué símbolo es antes incluso de que se codifique, pero el decodificador tiene que decodificar el símbolo con el fin de averiguar cuál es. Por lo tanto el decodificador, cuando decodifica un símbolo, sólo puede utilizar los viejos recuentos. Una vez que el símbolo ha sido descifrado, el decodificador incrementa su contador y actualiza las frecuencias acumuladas exactamente de la misma manera que el codificador.

El modelo debe mantener los símbolos, sus frecuencias de aparición (*recuentos*) y sus frecuencias acumuladas en una matriz. Esta matriz se debe mantener de forma ordenada de acuerdo con los *recuentos*. Cada vez que un símbolo se lee y se incrementa su contador de ocurrencias, el modelo actualiza las frecuencias acumuladas; a continuación, comprueba si es necesario intercambiar el símbolo con otro para mantener los recuentos en orden.

Existe una estructura de datos simple, que permite tanto facilitar la búsqueda como la actualización. Esta estructura es un árbol binario balanceado alojado en una matriz. (Un árbol binario balanceado es un árbol binario completo en el que algunos de los nodos de la parte inferior derecha pueden faltar.) El árbol debe tener un nodo por cada símbolo del alfabeto y puesto que es balanceado, su altura es $[\log_2 n]$, donde n es el tamaño del alfabeto. Para $n = 256$, la altura del árbol binario balanceado es de 8, por lo que partiendo de la raíz, la búsqueda de un nodo requiere un máximo de

a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	(a)
11	12	12	2	5	1	2	19	12	8	
a_8	a_2	a_3	a_9	a_1	a_{10}	a_5	a_4	a_7	a_6	(b)
19	12	12	12	11	8	5	2	2	1	

Tabla 2.58: Un alfabeto de 10 símbolos con contadores.

a_8	a_2	a_3	a_9	a_1	a_{10}	a_5	a_4	a_7	a_6	
19	12	12	12	11	8	5	2	2	1	(a)
40	16	8	2	1	0	0	0	0	0	
a_8	a_9	a_3	a_2	a_1	a_{10}	a_5	a_4	a_7	a_6	
19	13	12	12	11	8	5	2	2	1	(b)
41	16	8	2	1	0	0	0	0	0	

Tabla 2.59: Un alfabeto de 10 símbolos con contadores.

ocho pasos. El árbol está organizado de tal manera que los símbolos más probables (aquellos con los contadores de aparición más altos) se encuentran cerca de la raíz, lo que acelera las búsquedas. La Tabla 2.58a muestra un ejemplo con un alfabeto de diez símbolos con contadores[48]. La Tabla 2.58b muestra los mismos símbolos, ordenados por su frecuencia de aparición.

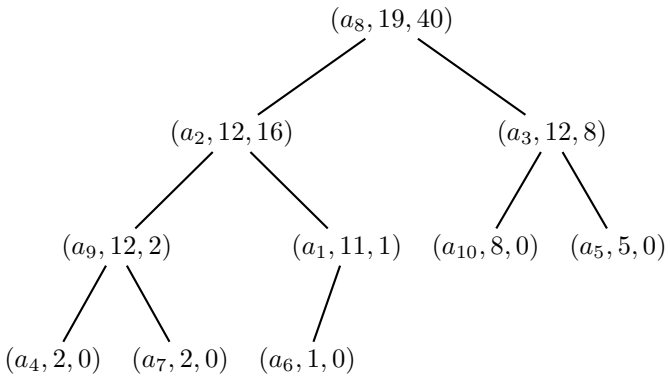
La matriz ordenada "aloja" el árbol binario balanceado de la Figura 2.60a. Esta es una manera elegante y sencilla de construir un árbol. Un árbol binario balanceado puede ser alojado en una matriz sin necesidad de utilizar punteros. La regla es que la ubicación de la primera matriz (con índice 1) contiene la raíz; los dos hijos del nodo situado en la posición i de la matriz están alojados en las localizaciones $2i$ y $2i+1$, y el padre del nodo residente en la posición j de la matriz se encuentra en la posición $[j/2]$. Es fácil ver cómo la ordenación de la matriz coloca los símbolos con los contadores mayores cerca de la raíz.

Además de un símbolo y su contador, se añade otro valor a cada nodo del árbol: la suma total de los contadores de su subárbol izquierdo. Esto se utilizará para calcular las frecuencias acumuladas. La matriz correspondiente se muestra en la Tabla 2.59a.

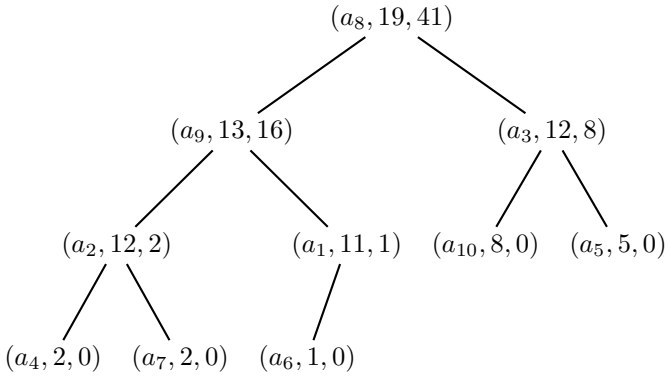
Suponemos que el siguiente símbolo de la cadena de entrada a leer es a_9. Su contador se incrementa desde 12 hasta 13. El modelo mantiene la matriz ordenada mediante la búsqueda del elemento de la matriz más alejado, a la izquierda de a_9, que tenga un contador más pequeño que el de a_9. Esta búsqueda puede ser directamente lineal si la matriz es lo suficientemente corta, o binaria si la matriz es larga. En nuestro caso, los símbolos a_9 y a_2 deben ser intercambiados (Tabla 2.59b). La Figura 2.60b muestra el árbol después del intercambio. Observe cómo se han actualizado los contadores del subárbol de la izquierda.

Finalmente, mostramos cómo se calculan las frecuencias acumuladas a partir de este árbol. Cuando se necesita la frecuencia acumulada de un símbolo `X`, el modelo sigue las ramas de los árboles desde la raíz hasta el nodo que contiene a `X`, mientras recoge la suma de los números en un entero `af`. Cada vez que recorre una rama derecha desde un nodo interior `N`, `fa` se incrementa con los dos números que se encuentran en ese nodo (el contador de la frecuencia de aparición del elemento del nodo y la suma de los contadores del subárbol izquierdo). Si se recorre la rama izquierda, `af` no se modifica. Cuando se alcanza el nodo que contiene `X`, el contador del subárbol izquierdo de `X` se agrega a `af`; en ese instante, `af` contiene la cantidad `LowCumFreq[X]`.

[48] Cada contador indica el número de repeticiones del símbolo asociado al mismo.



(a)



(b)

a_4	2	$0-1$
a_9	12	$2-13$
a_7	2	$14-15$
a_2	12	$16-27$
a_6	1	$28-28$
a_1	11	$29-39$
a_8	19	$40-58$
a_{10}	8	$59-66$
a_3	12	$67-78$
a_5	5	$79-83$

(c)

Figura 2.60: Codificación aritmética adaptativa.

A modo de ejemplo, vamos a recorrer el árbol de la Figura 2.60a desde la raíz hasta el símbolo a_6, cuya frecuencia acumulada es 28. Se sigue la rama derecha desde el nodo a_2, incrementando en 12 y 16 `af`. La rama izquierda del nodo a_1, no añade nada a `af`. Al llegar a a_6, su contador correspondiente a su subárbol izquierdo —0— se agrega a `af`. Como resultado, `af` contiene la suma $12 + 16 = 28$, como se puede comprobar en la Figura 2.60c. La cantidad `HighCumFreq[X]` se obtiene sumando el contador de a_6 (que es 1) a `LowCumFreq[X]`.

Para recorrer el árbol y encontrar el camino desde la raíz hasta a_6, el algoritmo realiza la siguientes pasos:

1. Encontrar a_6 en la matriz que aloja el árbol por medio de una búsqueda binaria. En nuestro ejemplo el nodo con a_6 se encuentra en la posición 10.

2. Hacer la división entera: 10 entre 2. El resto es 0, lo que significa que a_6 es el hijo izquierdo de su padre. El cociente es 5, que es el lugar de la matriz donde está localizado el padre.

3. El elemento 5 de la matriz contiene a_1. Hacer la división entera: 5 entre 2. El resto es 1, lo que significa que a_1 es el hijo derecho de su padre. El cociente es 2, que es la ubicación en la matriz del padre de a_1.

4. La posición 2 de la matriz contiene a_2. Hacer la división entera: 2 entre 2. El resto es 0, lo que significa que a_2 es el hijo izquierdo de su padre. El cociente es 1, que es la ubicación de la raíz en la matriz, por lo que el proceso se detiene.

El método de compresión PPM, Sección 2.18, es un buen ejemplo de un modelo estadístico que invoca un codificador aritmético de la manera aquí descrita.

> El conductor le tendió una carta. Boldwood la cogió y la abrió, la tensa espera de otro anónimo responde a las ideas populares sobre la probabilidad de tener una sensación para que lo precedente se repita de nuevo. "Yo no creo que sea para usted, señor", dijo el hombre cuando vio la acción de Boldwood. "Aunque no hay un nombre creo que es para su pastor."
>
> —Thomas Hardy, Far From The Madding Crowd[a]
>
> [a]Lejos del mundanal ruido.

2.15.1. Codificación del rango

El uso de enteros en la codificación aritmética es una necesidad en cualquier implementación práctica, pero resulta lenta en la codificación debido a la necesidad de renormalizaciones frecuentes. Los principales pasos en cualquier implementación de una codificación aritmética basada en enteros son: (1) reducción proporcional del rango y (2) expansión del rango (renormalización).

La modificación de la codificación del rango[49] es una mejora de la codificación aritmética que reduce el número de renormalizaciones y por lo tanto acelera la codificación aritmética de enteros en un factor de hasta 2. Las principales referencias son [Schindler 98] y [Campos 06]; la descripción aquí expuesta se basa en la primera.

La idea principal consiste en no tratar la salida como un número binario, sino como un número de otra base (normalmente se utiliza la base 256, para que cada dígito se represente en un byte). Esto requiere menos renormalizaciones y elimina las operaciones a nivel de bit. El siguiente análisis puede arrojar luz sobre este método.

[49]Range encoding ó range coding.

En cualquier momento durante la codificación aritmética, la salida se compone de las siguientes cuatro partes:

1. La parte ya escrita en la salida. Esta parte no cambia.

2. Un dígito (bit, byte o un dígito en otra base) que puede ser modificado por como mucho un acarreo al sumar por el extremo inferior del intervalo. (No puede haber dos acarreos porque cuando este dígito quedó determinado originalmente, el rango era menor o igual que uno. Dos acarreos exigen un mayor rango que una unidad.)

3. Un bloque (posiblemente vacío) de dígitos que producen acarreo (1 en binario, 9 en decimal, 255 en base 256, etc) y que están representados por un contador que registra su número.

4. La variable `low` del codificador.

Pueden ocurrir los siguientes estados mientras se codifican los datos:

- No es necesaria ninguna renormalización porque el rango está dentro del intervalo deseado.

- El extremo inferior (`low`) más el rango (éste es el extremo superior del intervalo) nunca producirá acarreos. En este caso, las partes segunda y tercera pueden escribirse en la salida, ya que nunca cambiarán.

- El dígito producido se convertirá en la segunda parte, y la tercera parte estará vacía. El extremo inferior ya ha producido un acarreo. En este caso, las partes segunda (modificada) y tercera pueden pasar a la salida; no habrá otro acarreo. La segunda y tercera parte se actualiza como antes.

- El dígito producido ocasionará un posible futuro acarreo, por lo que se añade al bloque de dígitos de la tercera parte.

La diferencia entre la codificación aritmética de enteros convencional y la codificación del rango es, que en el último, la segunda parte —que puede ser modificada por un acarreo— tiene que ser almacenada de forma explícita. Con salida binaria esta parte es siempre 0, ya que los unos siempre se añaden al bloque de posibles acarreos. Implementación que es sencilla.

En [Campos 06] se puede encontrar más información y el también el código. La codificación del rango se utiliza en LZMA (Sección 3.24).

2.16. El codificador QM

JPEG (Sección 4.8) es un importante método de compresión de imágenes. Utiliza la codificación aritmética, pero no en la forma descrita en la Sección 2.14. El codificador aritmético de JPEG se llama codificador QM y se describe en esta sección. Está diseñado para ser sencillo y veloz, por lo que está limitado a símbolos de entrada que son simples bits, y utiliza una aproximación en lugar de una multiplicación. También utiliza la aritmética de enteros de precisión fija, por lo que tiene que recurrir a la *renormalización* del intervalo de probabilidad de vez en cuando, con el fin de que la aproximación permanezca cercana a la multiplicación real. Para obtener más información de este método, véase [IBM 88], [Pennebaker y Mitchell 88a] y [Pennebaker y Mitchell 88b].

Surge una ligera confusión porque el codificador aritmético JPEG 2000 (Sección 5.19) y JBIG2 (Sección 4.12) se llama codificador MQ y no es el mismo que el codificador QM (El autor está en deuda con Christopher M. Brislawn por señalar esto).

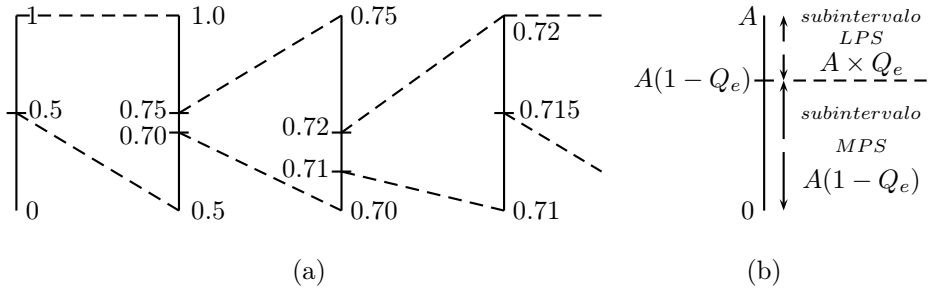


Figura 2.61: División del intervalo de probabilidad.

◇ **Ejercicio 2.38 (sol. en pág. 1066):** El codificador QM está limitado a símbolos de entrada que son simples bits. Sugiérase una forma de convertir un conjunto arbitrario de símbolos en una secuencia de bits.

La idea principal tras el codificador QM es clasificar cada símbolo de entrada (que es un solo bit), ya sea como el símbolo más probable (MPS), ya sea como el símbolo menos probable (LPS). Antes de introducir el bit siguiente, el codificador QM utiliza un modelo estadístico para determinar qué es más probable en este punto: un 0 ó un 1. A continuación, introduce el siguiente bit y lo clasifica de acuerdo con su valor actual. Si el modelo predice, por ejemplo, que un 0 es más probable y el siguiente bit resulta ser un 1, el codificador lo clasifica como un LPS. Es importante entender que la única información codificada en la secuencia comprimida es si el próximo bit es MPS o LPS. Cuando se descifra la cadena, todo lo que el decodificador sabe es si el bit que acaba de decodificar es un MPS o un LPS. La decodificador tiene que usar el mismo modelo estadístico para determinar la relación actual entre MPS/LPS y 0/1. Esta relación cambia —por supuesto— con cada bit, ya que el modelo es actualizado de manera idéntica (en sincronía a cada paso) por el codificador y el decodificador cada vez que un bit es introducido por el primero o decodificado por el último.

El modelo estadístico también calcula una probabilidad Q_e para el LPS, por lo que la probabilidad del MPS es $(1-Q_e)$. Puesto que Q_e es la probabilidad del *símbolo menos probable*, está dentro del rango [0,0,5]. El codificador divide la probabilidad del intervalo A en dos subintervalos de acuerdo con Q_e y coloca el subintervalo LPS (cuyo tamaño es $A \times Q_e$) por encima del subintervalo MPS (cuyo tamaño es $A\,(1-Q_e)$), como se muestra en la Figura 2.61b. Tenga en cuenta que los dos subintervalos de la figura están cerrados en la parte inferior y abiertos en la parte superior. Esto debería compararse con la forma en que un codificador aritmético convencional divide el mismo intervalo (véase la Figura 2.61a, donde las cifras se han tomado de la Tabla 2.48).

En la codificación aritmética tradicional, el intervalo se va estrechando todo el tiempo y el resultado final es un número perteneciente al último subintervalo. En el codificador QM, por simplicidad, en cada paso se añade la parte inferior del subintervalo seleccionado para la salida calculada hasta ese instante. Denotamos C a la cadena de salida. Si el bit actual leído de la entrada es el MPS, se añade a C la parte inferior del subintervalo MPS (i.e., el número 0) . Si el bit actual es el LPS, se añade a C la parte inferior del subintervalo LPS (i.e., el número $A\,(1-Q_e)$). Después de actualizar C de este modo, el intervalo de probabilidad A se reduce al tamaño del subintervalo seleccionado. El intervalo está siempre dentro del rango $[0,A)$, y A se hace más pequeño en cada paso. Este es el principio fundamental del codificador QM y que se expresa con las siguientes reglas:

$$\begin{aligned} &\text{Después de MPS}: C \text{ no cambia},\ A \leftarrow A\,(1-Q_e)\,, \\ &\text{Después de LPS}: C \leftarrow C + A\,(1-Q_e)\,,\ A \leftarrow A \times Q_e. \end{aligned} \tag{2.7}$$

Estas normas hacen que C apunte a la parte inferior del subintervalo MPS o LPS, dependiendo de la

Símbolo	C	A
Inicialmente	0	1
s_1 (LPS)	$0 + 1(1{-}0{,}5) = 0{,}5$	$1 \times 0{,}5 = 0{,}5$
s_2 (MPS)	sin cambios	$0{,}5 \times (1 - 0{,}5) = 0{,}25$
s_3 (LPS)	$0{,}5 + 0{,}25(1{-}0{,}5) = 0{,}625$	$0{,}25 \times 0{,}5 = 0{,}125$
s_4 (MPS)	sin cambios	$0{,}125 \times (1 - 0{,}5) = 0{,}0625$

Tabla 2.62: Codificación de cuatro símbolos con $Q_e = 0{,}5$.

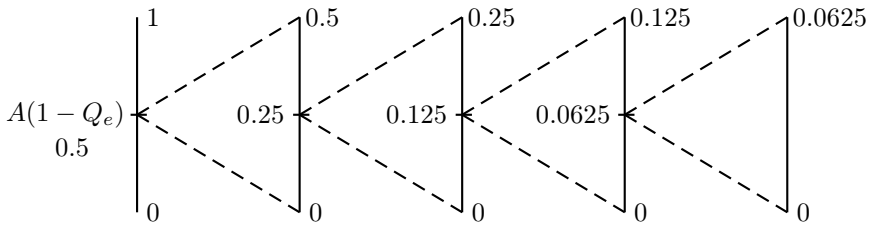


Figura 2.63: División del intervalo de probabilidad.

clasificación del bit leído de la entrada en ese instante. Asimismo, actualiza A con el nuevo tamaño del subintervalo.

La Tabla 2.62 muestra los valores de A y C, cuando se codifican cuatro símbolos, cada uno de un bit. Suponemos que son un LPS y un MPS alternativamente y que $Q_e = 0{,}5$ en los cuatro pasos (normalmente, por supuesto, el modelo estadístico proporciona valores diferentes de Q_e continuamente). Es fácil ver cómo el intervalo de probabilidad A se reduce desde 1 hasta 0,0625 y cómo la salida C crece desde 0 hasta 0,625. La Tabla 2.64 es similar, pero utiliza $Q_e = 0{,}1$ en los cuatro pasos. Una vez más A se reduce a 0,0081 y C crece hasta 0,981. Las Figuras 2.63 y 2.65 ilustran gráficamente la división del intervalo de probabilidad A dentro de un LPS y de un MPS.

◇ **Ejercicio 2.39 (sol. en pág. 1066):** Repítanse estos cálculos para el caso en que los cuatro símbolos son LPS y $Q_e = 0{,}5$; a continuación, para el caso en que sean MPS y $Q_e = 0{,}1$.

La idea principal del codificador QM es sencilla y fácil de comprender, pero ocasiona dos problemas. El primero es el hecho de que el intervalo A, que comienza en 1, se reduce continuamente y requiere una gran precisión para distinguirlo del cero. La solución a este problema es mantener A como un entero y doblarlo cada vez que sea demasiado pequeño. A esto se le llama *renormalización*. Es rápido, ya que se realiza mediante un desplazamiento lógico de bits y la multiplicación no es necesaria. Cada vez que A se duplica, C también se duplica. El segundo problema es la multiplicación $A \times Q_e$ utilizada para subdividir el intervalo de probabilidad A. Un método de compresión rápido debe evitar las multiplicaciones y las divisiones y tratar de reemplazarlos con adiciones, sustracciones y desplazamientos. Resulta que el segundo problema también se resuelve con la renormalización. La idea es mantener el valor de A cercano a 1, de modo que Q_e no sea muy diferente al producto $A \times Q_e$. La multiplicación

Símbolo	C	A
Inicialmente	0	1
s_1 (LPS)	$0 + 1(1{-}0{,}1) = 0{,}9$	$1 \times 0{,}1 = 0{,}1$
s_2 (MPS)	sin cambios	$0{,}1 \times (1 - 0{,}1) = 0{,}09$
s_3 (LPS)	$0{,}9 + 0{,}09(1{-}0{,}1) = 0{,}981$	$0{,}09 \times 0{,}1 = 0{,}009$
s_4 (MPS)	sin cambios	$0{,}009 \times (1 - 0{,}1) = 0{,}0081$

Tabla 2.64: Codificación de cuatro símbolos con $Q_e = 0{,}1$.

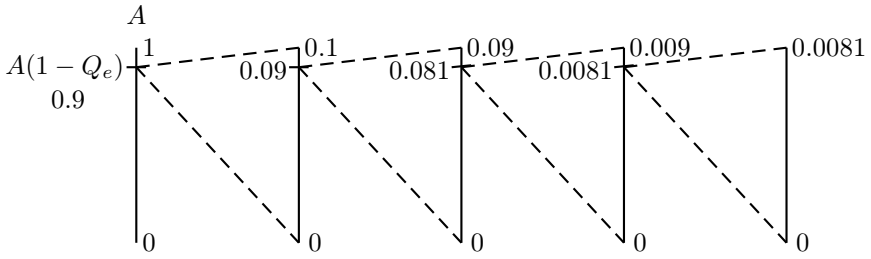


Figura 2.65: División del intervalo de probabilidad.

se *aproxima* usando Q_e.

¿Cómo podemos utilizar renormalización para mantener A cercano a 1? La primera idea que viene a la mente es doblar A cuando alcanza un valor un poco por debajo de 1, digamos 0,9. El problema está en que el doble de 0,9 es 1,8, más cercano a 2 que a 1. Si hacemos que A esté por debajo de 0,5 antes de duplicarlo, el resultado será menor que 1. No se tarda mucho en darse cuenta de que 0,75 es un buen valor mínimo para la renormalización. Si A alcanza este valor en un determinado paso, se duplica a 1,5. Si llega a un valor menor, tal como 0,6 ó 0,55, genera un valor aún más cercano a 1 cuando se duplica.

Si A alcanza un valor inferior a 0,5 en un paso determinado, tiene que ser renormalizado duplicándolo varias veces, C también se dobla cada vez. Un ejemplo es la segunda fila de la Tabla 2.64, donde A se reduce de 1 a 0,1 en un solo paso, debido a una probabilidad muy pequeña de Q_e. En este caso, A tiene que doblarse tres veces, de 0,1 a 0,2, a 0,4 y a 0,8, con el fin de conseguir el rango deseado [0,75,1,5). Concluimos entonces que A puede llegar a 0 (o muy cercano a 0) y puede tener un máximo de 1,5 (en realidad, menos de 1,5, ya que nuestros intervalos son siempre abiertos en el extremo superior).

◇ **Ejercicio 2.40 (sol. en pág. 1066):** ¿En qué caso A tiene que ser renormalizado siempre?

La aproximación de la multiplicación $A \times Q_e$ como Q_e cambia las reglas principales de la codificación QM:

$$\begin{aligned}\text{Después de MPS}: C \text{ no cambia},\ A \leftarrow A\,(1-Q_e) \approx A - Q_e,\\ \text{Después de LPS}: C \leftarrow C + A\,(1-Q_e) \approx C + A - Q_e,\ A \leftarrow A \times Q_e \approx Q_e.\end{aligned}$$

Con el fin de incluir renormalización en estas reglas, tenemos que elegir una representación entera para A, donde los valores reales pertenecientes al intervalo [0,1,5) se representen como números enteros. Ya que muchos equipos antiguos y actuales[50] trabajan con palabras de 16 bits, tiene sentido escoger una representación donde 0 se expresa con una palabra de 16 bits a cero y 1,5 con el número más pequeño de 17 bits, que es:

$$2^{16} = 65536_{10} = 10000_{16} = 1\underbrace{0\ldots0}_{16}{}_2.$$

De este modo, se pueden representar 65536 valores reales en el rango [0,1,5) con enteros de 16 bits, donde el mayor número entero de 16 bits, 65535, representa un valor real ligeramente inferior a 1,5. A

[50] A fecha de hoy, 21-11-2010, la mayor parte de los ordenadores trabajan con 32 bits y una minoría con 64 bits; pero gran cantidad de software aún no está preparado para los de 64 bits. El cálculo con 32 bits es similar al de 16 bits; el número más pequeño de 33 bits es ya bastante grande:

$$2^{32} = 4294967296_{10} = 100000000_{16} = 1\underbrace{0\ldots0}_{32}{}_2.$$

Símbolo	C	A	Renor. A	Renor. C
Inicialmente	0	1		
s_1 (LPS)	$0+1-0{,}5=0{,}5$	0,5	1	1
s_2 (MPS)	sin cambios	$1-0{,}5=0{,}5$	1	2
s_3 (LPS)	$2+1-0{,}5=2{,}5$	0,5	1	5
s_4 (MPS)	sin cambios	$1-0{,}5=0{,}5$	1	10

Tabla 2.66: Renormalización añadida a la Tabla 2.62.

Símbolo	C	A	Ren. A	Ren. C
Inicialmente	0	1		
s_1 (LPS)	$0+1-0{,}1=0{,}9$	0,1	0,8	$0{,}9\cdot 2^3=7{,}2$
s_2 (MPS)	sin cambios 7,2	$0{,}8-0{,}1=0{,}7$	1,4	$7{,}2\cdot 2=14{,}4$
s_3 (LPS)	$14{,}4+1{,}4-0{,}1=15{,}7$	0,1	0,8	$15{,}7\cdot 2^3=125{,}6$
s_4 (MPS)	sin cambios	$0{,}8-0{,}1=0{,}7$	1,4	$125{,}6\cdot 2=251{,}2$

Tabla 2.67: Renormalización añadida a la Tabla 2.64.

continuación damos algunos importantes ejemplos de tales valores:

$$0{,}75=\frac{1{,}5}{2}=2^{15}=32768_{10}=8000_{16},\ 1=0{,}75\left(\frac{4}{3}\right)=43690_{10}=AAAA_{16},$$
$$0{,}5=\frac{43690}{2}=21845_{10}=5555_{16},\ 0{,}25=\frac{21845}{2}=10923_{10}=2AAB_{16}.$$

(El valor óptimo de 1 en esta representación es $AAAA_{16}$, pero la manera de asociar los valores reales de A con los enteros de 16 bits es un tanto arbitraria. Lo importante acerca de esta representación es lograr la subdivisión precisa del intervalo y dicha subdivisión se lleva a cabo con alguna de las asignaciones: $A \leftarrow A-Q_e$ o $A \leftarrow Q_e$. La exactitud de la subdivisión depende, por consiguiente, de los valores relativos de A y de Q_e y se ha comprobado experimentalmente que el valor medio de A es $B55A_{16}$, por lo que este valor, en lugar de $AAAA_{16}$, se asocia en el codificador QM JPEG con $A=1$. La diferencia entre las dos cantidades —$AAAA$ y $B55A$— es $AB0_{16}=2736_{10}$. El codificador QM JBIG utiliza un valor ligeramente distinto para 1.)

La renormalización ahora puede incluirse en las reglas principales del codificador QM, de la siguiente manera:

$$\begin{aligned}\text{Después de MPS}: &\ C \text{ no cambia},\ A \leftarrow A-Q_e,\\ &\ si\ A<8000_{16} \text{ renormalizar } A \text{ y } C.\\ \text{Después de LPS}: &\ C \leftarrow C+A-Q_e,\ A \leftarrow Q_e,\\ &\ \text{renormalizar } A \text{ y } C.\end{aligned} \tag{2.8}$$

Las Tablas 2.66 y 2.67 muestran los resultados de la aplicación de estas reglas a los ejemplos de las Tablas 2.62 y 2.64, respectivamente.

◇ **Ejercicio 2.41 (sol. en pág. 1066):** Repítanse estos cálculos con renormalización para el caso en el que los cuatro símbolos sean LPS y $Q_e=0{,}5$. A continuación, realícense de nuevo los cálculos considerando todos MPS y $Q_e=0{,}1$. (Compárese con el ejercicio 2.39.)

El siguiente punto a considerar en el diseño del codificador QM es el problema de la *inversión del intervalo*. El tamaño del subintervalo asignado a los MPS puede llegar a ser más pequeño que el subintervalo de los LPS. Este problema puede ocurrir cuando Q_e está cercano a 0,5 y es un resultado de

Símbolo	C	A	Renor. A	Renor. C
Inicialmente	0	1		
s_1 (MPS)	0	$1 - 0{,}45 = 0{,}55$	1,1	0
s_2 (MPS)	0	$1{,}1 - 0{,}45 = 0{,}65$	1,3	0
s_3 (MPS)	0	$1{,}3 - 0{,}45 = 0{,}85$		
s_4 (MPS)	0	$0{,}85 - 0{,}45 = 0{,}40$	0,8	0

Tabla 2.68: Ilustrando la inversión del intervalo.

la aproximación de la multiplicación. Esto se ilustra en la Tabla 2.68, donde se codifican cuatro símbolos MPS con $Q_e = 0{,}45$. En la tercera fila de la tabla, el intervalo A se duplica desde 0,65 hasta 1,3. En la cuarta fila se reduce a 0,85. Este valor es superior a 0,75, por lo que no se realiza la renormalización; sin embargo, el subintervalo asignado a los MPS se convierte en $A - Q_e = 0{,}85 - 0{,}45 = 0{,}40$, que es más pequeño que el subintervalo LPS, el cual es $Q_e = 0{,}45$. Es evidente que el problema se produce cuando $Q_e > A/2$, una relación que también se puede expresar como $Q_e > A - Q_e$.

La solución es intercambiar los dos subintervalos cada vez que el subintervalo LPS se haga mayor que el subintervalo MPS. Esto se llama *intercambio condicional*. La condición para la inversión del intervalo es: $Q_e > A - Q_e$; pero como $Q_e \leq 0{,}5$, tenemos $A - Q_e < Q_e \leq 0{,}5$ y es obvio que tanto Q_e como $A - Q_e$ (i.e., tanto el subintervalo LPS como el MPS) son más pequeños que 0,75, por lo que la renormalización debe llevarse a cabo. Ésta es la razón por la cual la prueba para el intercambio condicional se realiza sólo *después* de que el codificador haya decidido que renormalización es necesaria. Las nuevas normas para el codificador QM se muestran en la Figura 2.69.

```
Después de MPS:
  C no cambia
  A ← A − Qe;          % El subintervalo MPS
  if A < 8000_16 then  % si la renormalización es necesaria
    if A < Qe then     % si es necesaria la inversion
      C ← C + A;       % señalar al extremo inferior de LPS
      A ← Qe           % Actualizar A con el subintervalo LPS
    endif;
    renormalizar A y C;
  endif;


Después de LPS:
  A ← A − Qe;          % El subintervalo MPS
  if A ≥ Qe then       % si el tamaño del intervalo no está invertido
    C ← C + A;         % señalar al extremo inferior de LPS
    A ← Qe             % Actualizar A con el subintervalo LPS
  endif;
  renormalizar A y C;
```

Figura 2.69: Reglas del codificador QM con inversión del intervalo.

El decodificador QM: El decodificador QM es el reverso del codificador QM. Por simplicidad

ignoramos la renormalización y el intercambio condicional y suponemos que el codificador QM opera utilizando las reglas de la ecuación (2.7). Invirtiendo el camino, C se actualiza con estas reglas produciendo las siguientes normas para el decodificador QM (el intervalo A se actualiza de la misma manera):

$$\begin{aligned} &\text{Después de MPS}: C \text{ no cambia},\ A \leftarrow A\,(1-Q_e)\,, \\ &\text{Después de LPS}: C \leftarrow C - A\,(1-Q_e)\,,\ A \leftarrow A \times Q_e. \end{aligned} \tag{2.9}$$

Estas reglas se demuestran con los datos de la Tabla 2.62. Los cuatro pasos de la decodificación son los siguientes:

- *Paso 1:* $C = 0{,}625$, $A = 1$; la línea divisoria es $A\,(1-Q_e) = 1\,(1-0{,}5) = 0{,}5$, por lo que los subintervalos LPS y MPS son $[0, 0{,}5)$ y $[0{,}5, 1)$, respectivamente. Puesto que C apunta al subintervalo superior, se decodifica como un LPS. El nuevo C es $0{,}625 - 1\,(1-0{,}5) = 0{,}125$ y la nueva A es $1 \times 0{,}5 = 0{,}5$.
- *Paso 2:* $C = 0{,}125$, $A = 0{,}5$; la línea divisoria es $A\,(1-Q_e) = 0{,}5\,(1-0{,}5) = 0{,}25$, por lo que los subintervalos LPS y MPS son $[0, 0{,}25)$ y $[0{,}25, 0{,}5)$, respectivamente y se decodifica como un MPS. C no cambia y la nueva A es $0{,}5\,(1-0{,}5) = 0{,}25$.
- *Paso 3:* $C = 0{,}125$, $A = 0{,}25$; la línea divisoria es $A\,(1-Q_e) = 0{,}25\,(1-0{,}5) = 0{,}125$, por lo que los subintervalos LPS y MPS son $[0, 0{,}125)$ y $[0{,}125, 0{,}25)$, respectivamente y se decodifica como un LPS. El nuevo C es $0{,}125 - 0{,}25\,(1-0{,}5) = 0$ y la nueva A es $0{,}25 \times 0{,}5 = 0{,}125$.
- *Paso 4:* $C = 0$, $A = 0{,}125$, la línea divisoria es $A\,(1-Q_e) = 0{,}125\,(1-0{,}5) = 0{,}0625$, por lo que los subintervalos LPS y MPS son $[0, 0{,}0625)$ y $[0{,}0625, 0{,}125)$, respectivamente y se decodifica como un MPS. C no se modifica y el nuevo A es $0{,}125(1-0{,}5) = 0{,}0625$.

◇ **Ejercicio 2.42 (sol. en pág. 1066):** Úsense las reglas de la ecuación (2.9) para descifrar los cuatro símbolos codificados en la Tabla 2.64.

Estimación de la probabilidad: El codificador QM utiliza un novedoso, interesante y poco conocido método para estimar la probabilidad Q_e de los LPS. El primer método que viene a la mente al tratar de estimar la probabilidad del siguiente bit de la entrada es inicializar Q_e a 0,5 y actualizarlo contando el número de ceros y unos que se han introducido hasta el momento. Si, por ejemplo, de los 1000 bits recibidos hasta cierto instante, 700 de ellos eran ceros, entonces el MPS actual es 0, con una probabilidad de $^{700}/_{1000} = 0{,}7$ y la probabilidad de los LPS es $Q_e = 0{,}3$. Tenga en cuenta que Q_e debe actualizarse *antes* de leer y codificar el siguiente bit de la entrada, ya que de otro modo, el decodificador no sería capaz de reflejar esta operación (el decodificador no conoce cuál es el bit siguiente). Este método produce buenos resultados, pero es lento, ya que Q_e debe actualizarse frecuentemente (idealmente, para cada bit entrante) y el cálculo implica una división (dividiendo 700/1000 en nuestro ejemplo).

El método utilizado por el codificador QM se basa en una tabla de valores preestablecidos para Q_e. Éste se inicializa a 0,5 y se modifica cuando la renormalización tenga lugar, no para cada bit entrante. La Tabla 2.70 ilustra el proceso. El índice de Q_e comienza siendo cero, por lo que el primer valor de Q_e es $0AC1_{16}$ o muy cercano a 0,5. Después de la primera renormalización de los MPS, el índice de Q_e se incrementa en 1, como se indica en la columna "Incr. MPS". Un índice de 1 implica un valor para Q_e de $0A81_{16}$ ó 0,49237, un poco más pequeño que el original, lo que refleja el hecho de que la renormalización se produjo a causa de un MPS. Si, por ejemplo, el índice actual de Q_e es 26 y la siguiente renormalización es LPS, el índice se decrementa en 3, como indica la columna "Decr. LPS", reduciendo Q_e a 0,00421. El método no se aplica muy a menudo y sólo necesita búsquedas en la tabla y aumentar o disminuir el índice de Q_e: operaciones rápidas y sencillas.

Índice Q_e	Hex. Q_e	Decr. Q_e	Decr. LPS	Incr. MPS	Cambio MPS	Índice Q_e	Hex. Q_e	Decr. Q_e	Decr. LPS	Incr. MPS	Cambio MPS
0	$0AC1$	0,50409	0	1	1	15	0181	0,07050	2	1	0
1	$0A81$	0,49237	1	1	0	16	0121	0,05295	2	1	0
2	$0A01$	0,46893	1	1	0	17	$00E1$	0,04120	2	1	0
3	0901	0,42206	1	1	0	18	$00A1$	0,02948	2	1	0
4	0701	0,32831	1	1	0	19	0071	0,02069	2	1	0
5	0681	0,30487	1	1	0	20	0059	0,01630	2	1	0
6	0601	0,28143	1	1	0	21	0053	0,01520	2	1	0
7	0501	0,23456	2	1	0	22	0027	0,00714	2	1	0
8	0481	0,21112	2	1	0	23	0017	0,00421	2	1	0
9	0441	0,19940	2	1	0	24	0013	0,00348	3	1	0
10	0381	0,16425	2	1	0	25	$000B$	0,00201	2	1	0
11	0301	0,14081	2	1	0	26	0007	0,00128	3	1	0
12	$02C1$	0,12909	2	1	0	27	0005	0,00092	2	1	0
13	0281	0,11737	2	1	0	28	0003	0,00055	3	1	0
14	0241	0,10565	2	1	0	29	0001	0,00018	2	0	0

Tabla 2.70: Tabla (ilustrativa) de estimación de probabilidades.

La columna etiquetada "Cambio MPS" de la Tabla 2.70 contiene información para el de cambio condicional de las definiciones de MPS y LPS con $Q_e = 0{,}5$. Hay que resaltar el valor cero de la parte inferior de la columna "Incr. MPS". Si el índice de Q_e es 29 y se produce una renormalización MPS, este cero hace que el índice permanezca en 29 (que corresponde al valor más pequeño de Q_e).

Las Tabla 2.70 se ha introducido aquí con fines ilustrativos. El codificador QM JPEG utiliza la Tabla 2.71, la cual tiene el mismo formato, pero es más difícil de comprender, ya que los valores de Q_e no se muestran en orden. Esta tabla fue elaborada mediante conceptos de estimación de probabilidad basados en la estadística Bayesiana.

Ahora vamos a justificar este método de estimación de probabilidad con un cálculo aproximado que sugiere que los valores de Q_e obtenidos con este método se adaptarán y aproximarán estrechamente a la probabilidad correcta de los LPS del flujo de datos binarios de la entrada. El método actualiza Q_e cada vez que se produce una renormalización y sabemos, por la ecuación (2.8), que esto sucede siempre que se introduce un LPS, pero no ocurre con todos los valores MPS. Por lo tanto, imaginamos un flujo de entrada ideal equilibrado en el que para cada bit LPS hay una secuencia de bits MPS consecutivos. Denotamos la verdadera (pero desconocida) probabilidad de LPS por q y tratamos de mostrar que los valores Q_e producidos por el método para este caso ideal están cercanos a q.

La Ecuación (2.8) expone las reglas principales del codificador QM y muestra cómo la probabilidad de intervalo A se reduce en Q_e cada vez que se introduce y codifica un MPS. Imagine una renormalización que hace que A alcance un valor A_1 (entre 1 y 1,5), seguida de una secuencia de N bits MPS consecutivos que reducen A en bloques de longitud Q_e desde A_1 hasta un valor A_2 que requiere otra renormalización (i.e., A_2 es menor que 0,75). Es evidente que:

$$N = \left[\frac{\Delta A}{Q_e}\right],$$

donde $\Delta A = A_1 - A_2$. Puesto que q es la probabilidad real de un LPS, la probabilidad de tener N bits MPS en una fila es $P = (1-q)^N$. Esto implica $\ln P = N \cdot \ln(1-q)$, que, para q pequeños, se

Índ. Q_e	Hex Q_e	Índ. siguiente LPS	Índ. siguiente MPS	Cambio MPS	Índ. Q_e	Hex Q_e	Índ. siguiente LPS	Índ. siguiente MPS	Cambio MPS
0	$5A1D$	1	1	1	57	$01A4$	55	58	0
1	2586	14	2	0	58	0160	56	59	0
2	1114	16	3	0	59	0125	57	60	0
3	$080B$	18	4	0	60	$00F6$	58	61	0
4	$03D8$	20	5	0	61	$00CB$	59	62	0
5	$01DA$	23	6	0	62	$00AB$	61	63	0
6	$00E5$	25	7	0	63	$008F$	61	32	0
7	$006F$	28	8	0	64	$5B12$	65	65	1
8	0036	30	9	0	65	$4D04$	80	66	0
9	$001A$	33	10	0	66	$412C$	81	67	0
10	$000D$	35	11	0	67	$37D8$	82	68	0
11	0006	9	12	0	68	$2FE8$	83	69	0
12	0003	10	13	0	69	$293C$	84	70	0
13	0001	12	13	0	70	2379	86	71	0
14	$5A7F$	15	15	1	71	$1EDF$	87	72	0
15	$3F25$	36	16	0	72	$1AA9$	87	73	0
16	$2CF2$	38	17	0	73	$174E$	72	74	0
17	$207C$	39	18	0	74	1424	72	75	0
18	$17B9$	40	19	0	75	$119C$	74	76	0
19	1182	42	20	0	76	$0F6B$	74	77	0
20	$0CEF$	43	21	0	77	$0D51$	75	78	0
21	$09A1$	45	22	0	78	$0BB6$	77	79	0
22	$072F$	46	23	0	79	$0A40$	77	48	0
23	$055C$	48	24	0	80	5832	80	81	1
24	0406	49	25	0	81	$4D1C$	88	82	0
25	0303	51	26	0	82	$438E$	89	83	0
26	0240	52	27	0	83	$3BDD$	90	84	0
27	$01B1$	54	28	0	84	$34EE$	91	85	0
28	0144	56	29	0	85	$2EAE$	92	86	0
29	$00F5$	57	30	0	86	$299A$	93	87	0
30	$00B7$	59	31	0	87	2516	86	88	0
31	$008A$	60	32	0	88	5570	88	89	1
32	0068	62	33	0	89	$4CA9$	95	90	0
33	$004E$	63	34	0	90	$44D9$	96	91	0
34	$003B$	32	35	0	91	$3E22$	97	92	0
35	$002C$	33	9	0	92	3824	99	93	0
36	$5AE1$	37	37	1	93	$32B4$	99	94	0
37	$484C$	64	38	0	94	$2E17$	93	86	0
38	$3A0D$	65	39	0	95	$56A8$	95	96	1
39	$2EF1$	67	40	0	96	$4F46$	101	97	0
40	$261F$	68	41	0	97	$47E5$	102	98	0
41	$1F33$	69	42	0	98	$41CF$	103	99	0
42	$19A8$	70	43	0	99	$3C3D$	104	100	0
43	1518	72	44	0	100	$375E$	99	93	0
44	1177	73	45	0	101	5231	105	102	0
45	$0E74$	74	46	0	102	$4C0F$	106	103	0
46	$0BFB$	75	47	0	103	4639	107	104	0
47	$09F8$	77	48	0	104	$415E$	103	99	0
48	0861	78	49	0	105	5627	105	106	1
49	0706	79	50	0	106	$50E7$	108	107	0
50	$05CD$	48	51	0	107	$4B85$	109	103	0
51	$04DE$	50	52	0	108	5597	110	109	0
52	$040F$	50	53	0	109	$504F$	111	107	0
53	0363	51	54	0	110	$5A10$	110	111	1
54	$02D4$	52	55	0	111	5522	112	109	0
55	$025C$	53	56	0	112	$59EB$	112	111	1
56	$01F8$	54	57	0					

Tabla 2.71: Tabla de estimación de probabilidades del codificador QM.

puede aproximar por:

$$\ln P \approx N(-q) = -\frac{\Delta A}{Q_e}q, \quad ó \quad P \approx \exp\left(-\frac{\Delta A}{Q_e}q\right). \tag{2.10}$$

Dado que se trata de un flujo de entrada ideal y equilibrado, nos interesa que $P = 0{,}5$, ya que ésto implica un número igual de renormalizaciones LPS y MPS. De $P = 0{,}5 = 2^{-1}$ obtenemos $\ln P = -\ln 2$, que, cuando se combina con la ecuación (2.10), produce:

$$Q_e = \frac{\Delta A}{\ln 2}q.$$

Esto es fortuito, porque $\ln 2 \approx 0{,}693$ y ΔA suele ser un poco menor que 0,75. Podemos decir que para nuestro flujo de datos de entrada ideal y balanceado, $Q_e \approx q$, proporcionando una justificación para nuestro método de estimación. Otra justificación procede del hecho de que P depende de Q_e [como muestra la Ecuación (2.10)]. Si Q_e se hace mayor que q, P también aumenta y la tabla tiende a acercarse a valores pequeños de Q_e. En caso contrario, la tabla tiende a seleccionar valores más grandes de Q_e.

2.17. Compresión de Texto

Antes de profundizar en los detalles del siguiente método, veamos unos conceptos generales sobre la compresión de texto. La mayor parte de los métodos de compresión de texto son: o bien estadísticos, o bien basados en un diccionario. Si pertenecen a la última clase, el texto se divide en fragmentos que se guardan en una estructura de datos llamada diccionario. Cuando un fragmento del nuevo texto resulta ser idéntica a una de las entradas del diccionario, se escribe en la cadena comprimida el puntero que señala a esa entrada, lo que produce la compresión del nuevo fragmento. La primera clase, por el contrario, está formada por métodos que desarrollan *modelos* estadísticos del texto.

Un método estadístico común consta de una fase de modelado seguida de otra de codificación. El modelo asigna probabilidades a los símbolos de la entrada y en la etapa de codificación realiza la codificación real de los símbolos basándose en dichas probabilidades. El modelo puede ser estático o dinámico (adaptativo). La mayoría de los modelos se basan en uno de los dos enfoques siguientes:

- **Frecuencia:** El modelo asigna probabilidades a los símbolos del texto en función de sus frecuencias de ocurrencia, de tal manera que a los símbolos que más aparecen se les asignan códigos más cortos. Un modelo estático utiliza probabilidades fijas, mientras que un modelo dinámico modifica las probabilidades "sobre la marcha" a medida que el texto es introducido y comprimido.

- **Contexto:** El modelo considera el contexto de un símbolo cuando le asigna una probabilidad. Puesto que el decodificador no tiene acceso al texto futuro, tanto el codificador como el decodificador deben limitarse al contexto del texto pasado, i.e., a los símbolos que ya han sido introducidos y procesados. En la práctica, el contexto de un símbolo son los N símbolos que le preceden. Por lo tanto, decimos que un método de compresión de texto basado en el contexto utiliza el contexto de un símbolo para *predecirlo* (i.e., para asignarle una probabilidad). Técnicamente, se dice que este método utiliza un modelo de Markov de "orden N". El método PPM, Sección 2.18, es un excelente ejemplo de un método de compresión basado en el contexto, si bien el concepto de contexto también se puede utilizar para comprimir imágenes.

Algunos de los métodos modernos de compresión de texto basados en el contexto realizan una transformación en los datos de entrada y luego aplican un modelo estadístico para asignar probabilidades a los símbolos transformados. Buenos ejemplos de estos métodos son el método de Burrows-Wheeler,

Sección 8.1, también conocido como la transformada de Burrows-Wheeler o *block sorting*[51]; la técnica de la clasificación ordenada de los símbolos[52], Sección 8.2; y el método ACB, Sección 8.3, que utiliza un diccionario asociativo.

2.18. PPM

PPM es un método de compresión sofisticado, derivado de la técnica de compresión de datos originalmente desarrollada por J. Cleary y I. Witten (véase [Cleary y Witten 84]), con extensiones y una implementación realizada por A. Moffat [Moffat 90]. El método se basa en un codificador que mantiene un modelo estadístico del texto. El codificador introduce el símbolo siguiente S, le asigna una probabilidad P y envía S a un codificador aritmético adaptativo, para codificarlo con la probabilidad P.

El *modelo estadístico* más sencillo cuenta el número de veces que ha aparecido cada símbolo en el pasado y les asigna a cada uno de ellos una probabilidad basada en eso. Supongamos que se han introducido y codificado 1217 símbolos hasta el instante actual y que 34 de ellos eran la letra `q`. Si el símbolo siguiente es una `q`, se le asigna una probabilidad de $^{34}/_{1217}$ y su contador se incrementa en 1. La próxima vez que `q` aparezca, se le asignará una probabilidad de $^{35}/_{t}$, donde t es el número total de símbolos introducidos hasta ese momento (sin incluir la última `q`).

El siguiente es un modelo estadístico *basado en el contexto*. La idea es asignar una probabilidad al símbolo S que dependa no sólo de la frecuencia del símbolo, sino del contexto en el que se ha producido hasta ese instante. La letra `h`, por ejemplo, aparece en un texto inglés "típico" (Tabla Intr.1) con una probabilidad de cercana al 5 %. En promedio, se espera ver una `h` aproximadamente el 5 % del tiempo. Sin embargo, si el símbolo actual es `t`, existe una alta probabilidad (aproximadamente un 30 %) de que el siguiente símbolo sea `h`, ya que el digrama `th` es corriente en inglés. Decimos que el modelo del inglés típico ***predice*** una `h` en tal caso. Si el siguiente símbolo es una `h` en realidad, se le asigna una probabilidad grande. En aquellos casos en los que `h` sea la segunda letra de un digrama poco probable, como `xh`, a la `h` se le asigna una probabilidad menor. Tenga en cuenta que la palabra "predice" se utiliza aquí en el sentido de "estimar la probabilidad de". Un ejemplo similar es la letra `u`, que tiene una probabilidad cercana a un 2 %. Cuando se encuentra una `q`, sin embargo, hay una probabilidad de más del 99 % de que la siguiente letra sea una `u`.

◇ **Ejercicio 2.43 (sol. en pág. 1068):** Sabemos que en inglés, que tras una `q` debe haber una `u`. ¿Por qué no decir que la probabilidad de aparición del digrama `qu` es del 100 %?

Un modelo *estático* basado en el contexto utiliza siempre las mismas probabilidades. Contiene tablas estáticas con las probabilidades de todos los diagramas posibles (o trigramas) del alfabeto y las utiliza para asignar una probabilidad al siguiente símbolo S en función del símbolo (o, en general, del contexto) C que le precede. Podemos imaginar que S y C se utilizan como índices, determinando una fila y una columna de una tabla de frecuencias estáticas. La tabla puede construirse a sí misma usando la acumulación de frecuencias de digramas o trigramas encontrados en grandes fragmentos de texto. Tal modelador es simple y produce buenos resultados en promedio, pero tiene dos problemas: El primero es que algunas secuencias de entrada pueden ser estadísticamente muy diferentes de los datos originalmente utilizados para preparar la tabla. Un codificador estático puede provocar una expansión considerable en tal caso. El segundo problema son las probabilidades cero.

¿Qué sucede si después de leer y analizar grandes cantidades de texto en inglés, aún no se ha encontrado el trigrama `qqz`? La celda correspondiente a `qqz` en la tabla de frecuencias de trigramas contendrá un cero. El codificador aritmético, Secciones 2.14 y 2.15, exige que todos los símbolos tengan probabilidades distintas de cero. Incluso si se utiliza un codificador diferente, como el de Huffman, todas

[51] Clasificación u ordenación de bloques.
[52] Symbol ranking.

las probabilidades implicadas deben ser no nulas. (Recordemos que el método de Huffman funciona combinando dos símbolos con baja probabilidad en un símbolo con alta probabilidad. Si se combinan dos símbolos con probabilidad cero, el símbolo resultante tendrá la misma probabilidad: cero.) Otra de las razones por las que un símbolo debe tener probabilidad distinta de cero es que su entropía (el menor número de bits en el que puede ser codificado) depende de $\log_2 P$, que no está definido para $P = 0$ (pero se hace muy grande cuando $P \to 0$). Este *problema de probabilidad cero* lo tiene cualquier modelo, estático o adaptativo, que utiliza las probabilidades de ocurrencia de los símbolos para lograr la compresión. Tradicionalmente se han adoptado dos sencillas soluciones para este problema, pero ninguna tiene una justificación teórica:

1. Después de analizar una gran cantidad de datos y contar las frecuencias, se repasa la tabla de frecuencias, en busca de las celdas vacías. A cada celda vacía se le asigna como contador de frecuencia un 1 y el contador total también se incrementa en 1. Este método pretende que todos los digramas y trigramas aparezcan al menos una vez.

2. Sumar 1 al contador total y dividir este único 1 entre todas las celdas vacías. Cada una recibirá un valor de contador menor que 1 y, como resultado, una baja probabilidad. Esto asigna una probabilidad muy pequeña a todo lo que no se haya visto en los datos de entrenamiento utilizados para el análisis.

Un modelador *adaptativo* basado en el contexto también mantiene tablas con las probabilidades de todos los digramas posibles (o trigramas o incluso largos contextos) del alfabeto y utiliza las tablas para asignar una probabilidad al siguiente símbolo S en función de unos pocos símbolos que le preceden inmediatamente (su contexto C). Las tablas se actualizan continuamente a medida que se van introduciendo más datos, lo que adapta las probabilidades a los datos particulares que se están comprimiendo.

Este modelo es más lento y más complejo que el estático, pero produce una mejor compresión, ya que utiliza las probabilidades correctas incluso cuando la entrada tiene datos con probabilidades muy diferentes de la media.

Un texto que distorsiona las probabilidades de una carta se llama *lipograma*. (¿Puede un programa de ordenador sin sentencias `goto` ser considerado un lipograma?) La palabra viene de la raíz griega $\lambda\epsilon\acute{\iota}\pi\omega$ (lipo o leipo) que significa perder, carecer de, en combinación con la palabra griega $\gamma\rho\acute{\alpha}\mu\mu\alpha$ (gramma), que significa "escrito" o "gráfico". Juntos forman $\lambda\iota\pi\sigma\gamma\rho\acute{\alpha}\mu\mu\alpha\tau\sigma\sigma$. No hay muchos ejemplos de obras literarias que sean lipogramas:

Una cita del Prefacio de *Gadsby*

La gente normalmente no se para a pensar que una tarea es como emular la realidad. Según escribí, al principio del manuscrito, todo un ejército de pequeñas E's se reunieron alrededor de mi escritorio, todas con impaciencia esperando ser llamadas. Sin embargo, gradualmente, a medida que me vieron escribiendo sin cesar, sin siquiera notarlas, se inquietaron; y, con susurros emocionados entre ellas, comenzaron a saltar y a cabalgar en mi pluma, buscando constantemente la oportunidad de sumergirse en una palabra; ¡todas ellas encaramadas como las aves marinas, en busca del paso de un pez! Pero cuando vieron que había cubierto 138 páginas de papel tipo máquina de escribir, se deslizaron hasta el suelo, caminando tristemente lejos, cogidas del brazo; pero gritaban nuevamente: "¡Indudablemente debes tener un baturrillo de historias sin nosotras! Porque, ¡hombre! Estamos en todas las historias escritas, ¡*cientos y miles de veces*! ¡Esta es la primera vez que hemos sido excluídas!"

—Ernest V. Wright

1. Tal vez el lipograma más conocido en Inglés es *Gadsby*, una novela de gran longitud [Wright 39], de Ernest V. Wright, que no contiene apenas ocurrencias de la letra E.

2. *Alphabetical Africa*[53] de Walter Abish (W. W. Norton, 1974) es un lipograma legible donde se supone que el lector debe descubrir el inusual estilo de escritura durante la lectura. Este estilo tiene que ver con las letras iniciales de las palabras. El libro consta de 52 capítulos. En el primero, todas las palabras que comienzan con una `a`, en el segundo, comienzan con `a` o `b`, etc., hasta que, en el Capítulo 26, están permitidas todas las letras al inicio de una palabra. En los restantes 26 capítulos, las letras son eliminadas una a una. Varios lectores han comentado sobre lo poco o lo mucho que han perdido la palabra "`the`" y cómo se sintieron cuando finalmente la vieron (en el Capítulo 20).

3. La novela *La Disparition* (la desaparición) es un lipograma francés de 1969 de Georges Perec, que no contiene la letra `E` (en realidad esta letra aparece varias veces, fuera del texto principal, en palabras que el editor tuvo que incluir, y todas ellas están impresas en rojo). *La Disparition* ha sido traducida al inglés, con el título *A Void*[54], de Gilbert Adair. Perec también escribió un *univocalic* (texto que emplea una sola vocal) titulado *Les Revenentes*[55], empleando sólo la vocal `E`. El título de la traducción en inglés (por Ian Monk) es *The Exeter Text, Jewels, Secrets, Sex*[56]. (Perec escribió también una breve historia de lipograms; véase [Motte 98].)

4. Gottlob Burmann, un poeta alemán, creó el siguiente ejemplo de un lipograma. Escribió 130 poemas, compuestos por cerca de 20 000 palabras, sin utilizar la letra `R`. También se cree que durante los últimos 17 años de su vida, omitió incluso esta letra de su conversación diaria.

5. Un lipograma portugués se encuentra en cinco historias escritas en 1941 por Alonso Alcalá y Herrera, un escritor portugués; cada una suprime una vocal.

6. Otros ejemplos, en español, se encuentran en los escritos de Francisco Navarrete y Ribera (1659), Fernando Jacinto de Zurita y Haro (1654), y Manuel Lorenzo de Lizarazu y Berbinzana (también 1654).

Un modelador adaptativo de orden N basado en el contexto lee el siguiente símbolo S de la cadena de la entrada y considera los N símbolos que preceden a S para obtener el contexto actual C de orden N de S. A continuación, el modelo estima la probabilidad P de que S aparezca en los siguientes datos de entrada con el contexto particular C. En teoría, cuanto mayor sea N, mejor será la estimación de la probabilidad (la *predicción*). Para tener una idea intuitiva, imagine el caso $N = 20\,000$. Es difícil imaginar una situación en la que a un grupo de 20 000 símbolos de la secuencia de entrada le siga un símbolo S; pero otro grupo con los mismos 20 000 símbolos, se encuentre más adelante en el mismo flujo de entrada, seguido por un símbolo diferente. Por lo tanto, $N = 20\,000$ permite al modelo predecir el siguiente símbolo (estimar su probabilidad) con una alta precisión. Sin embargo, los grandes valores de N tienen tres desventajas:

> Tecleaba yo con tanta fuerza esa noche que la letra `e` voló por la parte de la máquina que golpea el papel. No queriendo perder la noche, fui hasta la puerta de un vecino que, sabía, tenía un taller de elaboración en su bodega. Él trató de soldar mi `e` en su lugar, pero cuando empecé a trabajar de nuevo, voló como un abejorro. El resto de la noche inserté cada `e` a mano, y por la mañana tomé los últimos dólares de nuestra cuenta de ahorros para comprar una nueva máquina de escribir. Nada puede permitir retrasar la llegada de mi mayor triunfo.
>
> —Sloan Wilson, *What Shall We Wear to This Party*[a], (1976)

[a] ¿Qué nos pondremos para esta fiesta?

[53] África alfabética.
[54] Un vacío.
[55] Seguramente la palabra procede de *Revenantes*: Aparecidos, espectro, sombra, fantasma; cambiando la `a` por la `e`.
[56] El texto oculto, joyas, secretos, sexo.

1. Si codificamos un símbolo a partir de los 20 000 símbolos que le preceden, ¿cómo podemos codificar los primeros 20 000 símbolos de la secuencia de entrada? Tal vez tengan que escribirse en la cadena de salida como códigos ASCII, lo que reduce la compresión global.

2. Para valores grandes de N, puede haber demasiados posibles contextos. Si nuestros símbolos son códigos ASCII de 7 bits, el tamaño del alfabeto es de $2^7 = 128$ símbolos. Hay, pues, $128^2 = 16\,384$ contextos de orden 2, $128^3 = 2\,097\,152$ contextos de orden 3, y así sucesivamente. El número de contextos crece de manera exponencial, ya que es 128^N o, en general, A^N, donde A es el tamaño del alfabeto.

   ◇ **Ejercicio 2.44 (sol. en pág. 1068):** ¿Cuál es el número de contextos de orden 2 y de orden 3 para un alfabeto de tamaño $2^8 = 256$?

   ◇ **Ejercicio 2.45 (sol. en pág. 1068):** ¿Cuál sería un ejemplo práctico de un alfabeto de 16 símbolos?

3. Un contexto muy largo conserva la información sobre la naturaleza de los datos antiguos. La experiencia muestra que los archivos de datos grandes que contengan diferentes distribuciones de los símbolos en diferentes partes (un buen ejemplo es un libro de historia, donde es común que un capítulo utilice palabras como "Griego", "Atenas" y "Troya", mientras que el capítulo siguiente emplee "Romano", "imperio" y "legión"). La mejor compresión, por lo tanto, se puede lograr si el modelo asigna menos importancia a la información recogida de los datos antiguos y más a los más recientes. Tal efecto se consigue en un contexto corto.

◇ **Ejercicio 2.46 (sol. en pág. 1068):** Muéstrese un ejemplo de un archivo binario ordinario donde las diferentes partes pueden tener diferentes distribuciones de bits.

Como resultado, en la práctica se utilizan contextos relativamente cortos, del orden de 2 a 10. Cualquier algoritmo práctico requiere una estructura de datos cuidadosamente diseñada que proporcione una búsqueda rápida y una fácil actualización, mientras mantiene a muchos miles de símbolos y cadenas (Sección 2.18.5).

Pasamos ahora al siguiente punto en la discusión. Supongamos que tenemos un codificador basado en el contexto que utiliza contextos de orden 3. No tardando mucho, en el proceso de compresión, aparece la palabra `here` varias veces, pero la palabra `there` se ve ahora por primera vez. Supongamos que el siguiente símbolo es la `r` de `there`. El codificador no va a encontrar ninguna instancia de contexto de orden 3, `the` seguido por `r` (`r` tiene probabilidad 0 en este contexto). El codificador puede simplemente escribir `r` en la cadena comprimida como un literal, lo que no produce compresión, pero sabemos `r` que fue visto varias veces en el pasado, siguiendo el contexto de orden 2, (`r` tiene una probabilidad distinta de cero en este contexto). El método PPM se aprovecha de este conocimiento.

"*uvulapalatopharangoplasty*" es el nombre de un procedimiento quirúrgico para corregir la apnea del sueño. Se rumorea que es la palabra más larga (en ¿inglés?) sin ninguna `e`.

2.18.1. Principios del PPM

La idea central del PPM es utilizar este conocimiento. El codificador PPM cambia a un contexto más corto cuando uno más largo da lugar a una probabilidad de 0. Así, PPM comienza con un contexto de orden N. Busca en su estructura de datos una aparición previa del contexto actual C seguido por el

siguiente símbolo S; si no encuentra tal ocurrencia (i.e., si la probabilidad de hallar este C particular seguido por este S es 0), cambia al orden $N-1$ y comprueba de nuevo lo mismo. Sea C' la cadena formada por los $N-1$ símbolos del extremo derecho de C. El codificador PPM busca en su estructura de datos una ocurrencia previa del contexto actual C' seguida por el símbolo S. PPM, por lo tanto, trata de usar las piezas más pequeñas del contexto C, que es la razón de su nombre. Las siglas PPM proceden de "Prediction by Partial Matching[57]". A continuación se indica el proceso en detalle.

El codificador lee el siguiente símbolo S de la secuencia de entrada, mira el contexto actual C de orden N (los últimos N símbolos leídos), y basándose en los datos introducidos en el pasado, determina la probabilidad P de que S aparezca a continuación del contexto particular C. Después, el codificador invoca un algoritmo de codificación aritmética adaptativo para codificar el símbolo S con una probabilidad P. En la práctica, el codificador aritmético adaptativo es un procedimiento que recibe las cantidades `HighCumFreq[X]` y `LowCumFreq[X]` (Sección 2.15) como parámetros del codificador PPM.

A modo de ejemplo, supongamos que el contexto actual de orden 3 es la cadena `the`, la cual ha aparecido ya 27 veces en el pasado seguida por las letras: `r` (11 veces), `s` (9 veces), `n` (6 veces) y `m` (una sola vez). El codificador asigna a estos casos las probabilidades $^{11}/_{27}$, $^{9}/_{27}$, $^{6}/_{27}$ y $^{1}/_{27}$ respectivamente. Si el siguiente símbolo leído es `r`, éste se envía al codificador aritmético con una probabilidad de $^{11}/_{27}$, y las probabilidades se actualizan a $^{12}/_{28}$, $^{9}/_{28}$, $^{6}/_{28}$ y $^{1}/_{28}$.

¿Qué sucede si el siguiente símbolo leído es `a`? El contexto `the` nunca se ha visto seguido por una `a`, por lo que la probabilidad de este caso es 0. Este problema de la probabilidad cero se resuelve en PPM cambiando a un contexto más corto. El codificador PPM pregunta cuántas veces aparece `he` en el pasado en un contexto de orden 2, y qué símbolos le siguen. La respuesta puede ser la siguiente: Aparece 54 veces, seguido por `a` (26 veces), por `r` (12 veces), etc. El codificador PPM ahora envía la `a` al codificador aritmético con una probabilidad de $^{26}/_{54}$.

Si el siguiente símbolo `S` nunca ha sido visto antes siguiendo el contexto de orden 2 `he`, el codificador PPM cambia a un contexto de orden 1. ¿Fue la `S` vista a continuación de la cadena `e`? En caso afirmativo, se le asigna una probabilidad distinta de cero a `S` en función de cuántas veces ella (y otros símbolos) se han encontrado tras la `e`. De lo contrario, el codificador PPM cambia a un contexto de orden 0. Se pregunta a sí mismo cuántas veces ha aparecido el símbolo S en el pasado, independientemente de cualquier contexto. Si ha sido visto 87 veces de cada 574 símbolos leídos, se le asigna una probabilidad de $^{87}/_{574}$. Si el símbolo `S` nunca se ha encontrado antes (una situación común al comienzo de cualquier proceso de compresión), el codificador PPM cambia a un modo llamado contexto de orden -1, donde a `S` se le asigna la probabilidad fija de $^{1}/_{\text{(tamaño del alfabeto)}}$.

> Predecir es una cosa. Predecir correctamente es otra.
>
> —Desconocido

Las Tablas 2.72a y 2.72b muestran los contextos y los recuentos de frecuencia para los órdenes, desde 4 hasta 0, después de introducir y codificar la cadena de 11 símbolos `xyzzxyxyzzx`. Para comprender la operación del codificador PPM, vamos a suponer que el 12^{avo} símbolo es `z`. El contexto de orden 4 es ahora `yzzx`, que antes se ha visto seguido por `y`, pero nunca por `z`. El codificador por lo tanto cambia al contexto de orden 3, que es `zzx`, pero incluso éste no aparece antes seguido por `z`. El siguiente contexto, `zx`, es de orden 2, y también falla. Entonces el codificador cambia al orden 1, donde comprueba el contexto `x`. El símbolo `x` se encuentra tres veces en el pasado, pero siempre seguido por `y`. El orden 0 es el siguiente que se verifica, donde `z` tiene un contador de frecuencia de 4 (de un total de 11). Por lo tanto, se envía el símbolo `z` al codificador aritmético adaptativo, para ser codificado con una probabilidad de $^{4}/_{11}$ (el codificador PPM "predice" que aparecerá un $^{4}/_{11}$ del tiempo).

[57] Predicción mediante localización de cadenas parciales.

Orden 4		Orden 3		Orden 2		Orden 1		Orden 0	
xyzz → x	2	xyz → z	2	xy → z	2	x → y	3	x	4
yzzx → y	1	yzz → x	2	→ x	1	y → z	2	y	3
zzxy → x	1	zzx → y	1	yz → z	2	→ x	1	z	4
zxyx → y	1	zxy → x	1	zz → x	2	z → z	2		
xyxy → z	1	xyx → y	1	zx → y	1	→ x	2		
yxyz → z	1	yxy → z	1	yx → y	1				

(a): Contextos y contadores.

Orden 4		Orden 3		Orden 2		Orden 1		Orden 0	
xyzz → x	2	xyz → z	2	xy → z	2	x → y	3	x	4
yzzx → y	1	yzz → x	2	xy → x	1	→ z	1	y	3
→ z	1	zzx → y	1	yz → z	2	y → z	2	z	5
zzxy → x	1	→ z	1	zz → x	2	→ x	1		
zxyx → y	1	zxy → x	1	zx → y	1	z → z	2		
xyxy → z	1	xyx → y	1	→ z	1	→ x	2		
yxyz → z	1	yxy → z	1	yx → y	1				

(b): Tabla actualizada después de introducir el símbolo z.

Tabla 2.72: Contextos y contadores para "xyzzxyxyzzx".

A continuación, consideramos el decodificador PPM. Hay una diferencia fundamental en la forma en que trabajan el codificador y el decodificador PPM. El codificador siempre puede mirar el símbolo siguiente y basa su próximo paso en ese símbolo. El trabajo del decodificador es identificar el siguiente símbolo. El codificador decide cambiar a un contexto más corto basándose en qué símbolo es el siguiente. El decodificador no puede reflejar esto, ya que no conoce dicho símbolo. El algoritmo necesita una característica adicional que permitirá al decodificador permanecer sincronizado con el codificador. La característica utilizada por PPM es reservar un símbolo del alfabeto como *símbolo de escape*. Cuando el codificador decide para cambiar a un contexto más corto, escribe primero el símbolo de escape (codificado aritméticamente) en la secuencia de salida. El decodificador puede decodificar el símbolo de escape, ya que se codifica en el contexto actual. Después de decodificar un escape, el decodificador también cambia a un contexto más corto.

Lo peor que puede suceder con un codificador de orden N es encontrarse con un símbolo S por primera vez (esto sucede sobre todo al comienzo del proceso de compresión). El símbolo no se ha visto antes en ningún contexto, ni siquiera en el contexto de orden 0 (i.e., por sí mismo). En tal caso, el codificador acaba enviando $N+1$ escapes consecutivos para que se codifiquen aritméticamente y formen parte de la salida, bajando hasta el orden -1, seguidos por el símbolo S codificado con la probabilidad fija de $1/(\text{tamaño del alfabeto})$. Puesto que el codificador puede producir muchas veces un símbolo de escape, es importante asignarle una probabilidad razonable. Inicialmente, esta probabilidad debe ser alta, pero debería bajar a medida que los símbolos son introducidos y decodificados y el modelador recoge más información sobre los contextos en los datos en particular que se están comprimiendo.

◇ **Ejercicio 2.47 (sol. en pág. 1068):** El *carácter de escape* es sólo un símbolo del alfabeto, reservado para indicar un cambio de contexto. ¿Qué pasa si los datos utilizan cada símbolo del alfabeto y ninguno puede reservarse? Un ejemplo común es la compresión de imágenes, donde se representa un píxel por un byte (256 escalas de grises o colores). Ya que los píxeles pueden tener cualquier valor entre 0 y 255, ¿qué valor puede reservarse para el símbolo de *escape* en este caso?

La Tabla 2.73 muestra una forma de asignar probabilidades a los símbolos de escape (ésta es la variante PPMC de PPM). La tabla muestra los contextos (hasta el orden 2) recogidos durante la lectura y codificación de la cadena de 14 símbolos `assanissimassa`. (En la película "8 1/2", los niños italianos emplean esta cadena como palabra mágica. Ellos la pronuncian `assa-neesee-massa`.) Suponemos que el alfabeto se compone de las 26 letras, el espacio en blanco y el símbolo escape; un total de 28 símbolos. La probabilidad de encontrar un símbolo con orden -1 es, por tanto, de $1/28$. Tenga en cuenta que, sin compresión, se necesitan 5 bits para codificar uno de los 28 símbolos.

Orden 2			Orden 1			Orden 0		
Contexto	f	p	Contexto	f	p	Símbolo	f	p
as → s	2	2/3	a → s	2	2/5	a	4	4/19
esc	1	1/3	a → n	1	1/5	s	6	6/19
			esc →	2	2/5	n	1	1/19
ss → a	2	2/5				i	2	2/19
ss → i	1	1/5	s → s	3	3/9	m	1	1/19
esc	2	2/5	s → a	2	2/9	esc	5	5/19
			s → i	1	1/9			
sa → n	1	1/2	esc	3	3/9			
esc	1	1/2						
			n → i	1	1/2			
an → i	1	1/2	esc	1	1/2			
esc	1	1/2						
			i → s	1	1/4			
ni → s	1	1/2	i → m	1	1/4			
esc	1	1/2	esc	2	2/4			
is → s	1	1/2	m → a	1	1/2			
esc	1	1/2	esc	1	1/2			
si → m	1	1/2						
esc	1	1/2						
im → a	1	1/2						
esc	1	1/2						
ma → s	1	1/2						
esc	1	1/2						

Tabla 2.73: Contextos, contadores (f), y probabilidades (p) de "`assanissimassa`".

Cada contexto visto en el pasado se coloca en la tabla en un grupo aparte junto con el símbolo escape. El contexto de orden 2 `as`, e.g., aparece dos veces en el pasado, seguido por `s` en ambas ocasiones. Se le asigna una frecuencia de 2 y se coloca en un grupo junto con el símbolo escape, al que se le asigna la frecuencia 1. Las probabilidades de `as` y de escape en este grupo son, por lo tanto, de 2/3 y 1/3, respectivamente. El contexto `ss` aparece tres veces: dos seguidos por `a` y uno por `i`. A estos dos casos se les asignan las frecuencias 2 y 1, respectivamente, y se colocan en un grupo junto con el símbolo escape, cuya frecuencia es ahora de 2 (porque está en un grupo de dos elementos). Las probabilidades de los tres miembros de este grupo son, por lo tanto: 2/5, 1/5 y 2/5, respectivamente.

La justificación de este método para asignar probabilidades al símbolo escape es la siguiente: Supongamos que el contexto `abc` ha sido visto diez veces en el pasado y siempre seguido por `x`. Esto

sugiere que el mismo contexto deberá encontrarse seguido por la misma `x` en el futuro, por lo que el codificador sólo tiene que cambiar en raras ocasiones a un contexto más bajo. Al símbolo escape por lo tanto se le puede asignar la pequeña probabilidad de $1/11$. Sin embargo, si cada ocurrencia de contexto `abc` apareció seguido en el pasado por un símbolo diferente (lo que sugiere que los datos varían mucho), entonces hay una buena probabilidad de que la siguiente aparición esté también seguida por un símbolo diferente, lo que obliga al codificador a cambiar a un contexto más bajo (y por ello, a emitir un escape) con más frecuencia. Al símbolo escape se le asigna, por lo tanto una mayor probabilidad: $10/20$.

◇ **Ejercicio 2.48 (sol. en pág. 1068):** Explíquense los números $1/11$ y $10/20$.

El orden 0 consta de los cinco símbolos diferentes `asnim` vistos en la cadena de entrada, seguidos por un escape, al que se le asigna una frecuencia de 5. Por lo tanto, la probabilidad asociada a cada elemento depende de su número: $4/19$ (para `a`) , $6/19$ (para `s`), $2/19$ (para `i`), $1/19$ (para `n` y `m`), y $5/19$ (para el símbolo escape).

> Los índices de Wall Street han predicho nueve de las últimas cinco recesiones.
>
> —Paul A. Samuelson, *Newsweek* (19 de septiembre de 1966)

2.18.2. Ejemplos

Ahora estamos listos para ver ejemplos reales de lectura y codificación de nuevos símbolos. Suponemos que la cadena de 14 elementos `assanissimassa` ha sido introducida y codificada completamente, por lo que el contexto actual de orden 2 es "`sa`". A continuación indicamos cuatro casos típicos:

1. El siguiente símbolo es `n`. El codificador PPM encuentra que `sa` seguido de `n` ha aparecido antes y tiene una probabilidad de $1/2$. El codificador aritmético codifica la `n` con esta probabilidad, que conoce, ya que la codificación aritmética normalmente comprime en o cercano al valor de la entropía: $-\log_2(1/2) = 1$ bit.

2. El siguiente símbolo es `s`. El codificador PPM observa que `sa` nunca ha sido vista antes seguida de una `s`. Por ello, el codificador envía el símbolo `escape` al codificador aritmético, junto con la probabilidad ($1/2$) predicha por el contexto de orden 2 de `sa`. Por lo tanto, gasta un bit para codificar este `escape`. Cambiando al orden 1, el contexto actual se convierte en `a`, y el codificador PPM encuentra que una `a` seguida por una `s` ya ha aparecido antes y actualmente tiene una probabilidad asignada de $2/5$. La `s` se envía al codificador aritmético para que la codifique con una probabilidad de $2/5$, lo que produce otros $-\log_2(2/5) = 1{,}32$ bits. En total se generan $1 + 1{,}32 = 2{,}32$ bits para codificar la `s`.

3. El siguiente símbolo es `m`. El codificador PPM observa que `sa` nunca se ha visto antes seguida por una `m`. Por tanto, envía el símbolo `escape` al codificador aritmético, como en el caso 2, produciendo el gasto de un bit, hasta el momento. A continuación, cambia al orden 1; determina que `a` nunca ha aparecido antes seguida por una `m`, por lo que envía otro `escape`, esta vez utilizando la probabilidad del símbolo `escape` para el orden 1 `a`, que es $2/5$. Esto se codifica en 1,32 bits. Cambiando al orden 0, el codificador PPM encuentra `m`, con una probabilidad de $1/19$ y lo envía para ser codificados en $-\log_2(1/19) = 4{,}25$ bits. El número total de bits producidos es, por tanto, de $1 + 1{,}32 + 4{,}25 = 6{,}57$.

4. El siguiente símbolo es `d`. El codificador PPM codifica los cambios desde el orden 2 hasta el orden 0, enviando dos `escapes`, como en el caso 3. Puesto que `d` no se ha visto antes, no se encuentra en el orden 0, y el codificador PPM cambia al orden -1 después de enviar un tercer `escape` a la salida con la probabilidad del símbolo `escape` para el orden 0, $5/19$ (esto gasta $-\log_2(5/19) = 1{,}93$ bits). La `d` misma se envía al codificador aritmético junto con su probabilidad de orden -1, que es $1/28$, por lo que se codifica en 4,8 bits. El número total de bits necesarios para codificar esta primera `d` es: $1 + 1{,}32 + 1{,}93 + 4{,}8 = 9{,}05$; más que los cinco bits que habrían sido necesarios sin ningún tipo de compresión.

◇ **Ejercicio 2.49 (sol. en pág. 1068):** Supóngase que se ha producido el caso 4 (i.e., el 15^{avo} símbolo introducido ha sido una `d`). Muéstrese el nuevo estado de los contextos de orden 0.

◇ **Ejercicio 2.50 (sol. en pág. 1068):** Supóngase que se ha producido el caso 4 y el 16^{avo} símbolo es también una `d`. ¿Cuántos bits se necesitarían para codificar este segundo símbolo `d`?

◇ **Ejercicio 2.51 (sol. en pág. 1069):** Muéstrese cómo los resultados de los anteriores cuatro casos se ven afectados si se asume un tamaño del alfabeto de 256 símbolos.

2.18.3. Exclusión

Cuando se baja desde el orden 2 hasta el orden 1, el codificador PPM puede utilizar la información que se encuentra en el orden 2 con el fin de excluir a ciertos casos de orden 1, que ahora se sabe que imposible. Esto incrementa las probabilidades del orden 1 y por lo tanto mejora la compresión. Lo mismo se puede hacer cuando se baja desde cualquier otro orden. A continuación ofrecemos dos ejemplos detallados.

En el caso 2, el siguiente símbolo es `s`. El codificador PPM observa que `sa` se ha visto antes seguido por una `n`, pero no por una `s`. El codificador envía un símbolo `escape` y cambia al orden 1. El contexto actual se convierte en `a`, y el codificador busca la aparición anterior de una `a` seguida por una `s`. La respuesta es sí (con frecuencia 2), pero el hecho de que `sa` haya aparecido antes seguido por una `n` implica que el símbolo actual no puede ser `n` (si lo fuera, sería codificado en el orden 2).

El codificador puede *excluir* el caso de una `a` seguida por una `n` en los contextos de orden 1 [se puede decir que no hay necesidad de reservar "lugar" (o "espacio") para la probabilidad de este caso, ya que es imposible que suceda]. Esto reduce la frecuencia total del grupo de orden 1 "$a \rightarrow$" desde 5 hasta 4, lo que incrementa la probabilidad asignada para `s` de $2/5$ a $2/4$. Basándose en nuestro conocimiento del orden 2, la `s` ahora se puede codificar en $-\log_2(2/4) = 1$ bit, en lugar de 1,32 (se gasta un total de dos bits, ya que el símbolo `escape` también requiere un bit).

Otro ejemplo es el caso 4, modificado para la exclusión. Cuando se desciende desde el orden 2 hasta el orden 1, la probabilidad del símbolo `escape` es, como antes, de $1/2$. Cuando en el orden 1, se excluye el caso de `a` seguida de `n`, incrementa la probabilidad de `escape` de $2/5$ a $2/4$. Después de cambiar al orden 0, tanto `s` como `n`, representan casos imposibles y pueden ser excluidos. Esto deja al orden 0 con los cuatro símbolos `a`, `i`, `m` y `escape`, con frecuencias de 4, 2, 1 y 5, respectivamente. La frecuencia total es de 12, por lo que al símbolo `escape` se le asigna una probabilidad de $5/12$ (1,26 bits) en lugar de la original $5/19$ (1,93 bits). Este `escape` se envía al codificador aritmético, y el codificador PPM cambia al orden -1. Aquí excluye a los cinco símbolos `asnim` que ya se han visto en el orden 1 y por lo tanto son imposibles para el orden -1. La `d` ahora se puede codificar con una probabilidad de $1/(28-5) \approx 0{,}043$ (4,52 bits en lugar de 4,8) ó $1/(256-5) \approx 0{,}004$ (7,97 bits en lugar de 8), dependiendo del tamaño del alfabeto.

La construcción exacta y cuidadosa de modelos debe incorporar restricciones
que satisfagan en cualquier caso la respuesta final.

—Francis Crick, *What Mad Pursuit*[a], (1988)

[a] ¡Qué loca búsqueda!

2.18.4. Cuatro variantes del PPM

El método particular descrito anteriormente para la asignación de las probabilidades del símbolo `escape` se llama PPMC. Se han desarrolado cuatro métodos más: PMMA, PPMB, PPMP y PPMX, para intentar asignar probabilidades exactas al símbolo `escape` en PPM. Los cinco métodos han sido seleccionados en base a la amplia experiencia que los desarrolladores han obtenido con la compresión de datos. Los dos últimos se basan en la distribución de Poisson [Witten y Bell 91], que es el razón de la "P" en PPMP (la "X" viene de "aproximación", ya que es un PPMX variante aproximada de PPMP).

Supongamos que un grupo de contextos en la Tabla 2.73 tiene como frecuencia total n (excluyendo el símbolo `escape`). PPMA asigna al símbolo `escape` una probabilidad de $1/(n+1)$. Esto es equivalente a asignar siempre un contador con valor 1. Los otros miembros del grupo tienen aún asignadas sus probabilidades originales de x/n, y estas probabilidades se suman para obtener 1 (sin incluir la probabilidad del símbolo escape).

PPMB es similar a PPMC con una diferencia: Asigna una probabilidad al símbolo `S` siguiendo el contexto C sólo después de que `S` se haya visto dos veces en el contexto C. Esto se consigue restando 1 de los contadores de frecuencia. Si, por ejemplo, el contexto `abc` se ha visto tres veces: dos delante de `x` y una de `y`, entonces a `x` se le asigna una probabilidad de $(2-1)/3$, y a `y` (que tendría que adquirir una probabilidad de $(1-1)/3 = 0$) no se le asigna ninguna probabilidad (i.e., no se incluye en la tabla 2.73 o su equivalente). En cambio, el símbolo `escape` "obtiene" los dos valores sustraídos de `x` e `y`, lo que significa una probabilidad de $2/3$. Este método se basa en la creencia de que "ver dos veces es creer."

PPMP se basa en un principio diferente. Se considera el aspecto de cada símbolo un proceso de Poisson separado. Supongamos que se tienen q símbolos diferentes en la secuencia de entrada. En un momento dado durante la compresión, se han leído n símbolos, y el símbolo `i` se ha introducido c_i veces (por lo que $\sum c_i = n$). Algunos de los c_is son cero (este es el problema de la probabilidad cero). PPMP se basa en el supuesto de que el símbolo `i` aparezca siguiendo una distribución de Poisson con un valor esperado (media) λ_i. El problema estadístico considerado por PPMP es estimar q extrapolando a partir de muestras de n símbolos de los datos introducidos hasta el momento, para determinar la cadena de entrada total de N símbolos (o, en general, una muestra más grande). Si expresamos N en términos de n en la forma $N = (1+\theta)\,n$, entonces un largo análisis muestra que el número de símbolos que no han aparecido en la muestra de n elementos está dada por $t_1\theta - t_2\theta^2 + t_3\theta^3 - \cdots$, donde t_1 es el número de símbolos que se han encontrado exactamente una vez en nuestra muestra, t_2 es el número de símbolos que se han encontrado dos veces, y así sucesivamente.

Hapax legomena: palabras o formas que se presentan sólo una vez en los escritos de un determinado lenguaje; tales palabras son muy difíciles, si no imposibles, de traducir.

En el caso concreto en el que N no es la secuencia de entrada completa sino la muestra de tamaño algo mayor $n+1$, el número de nuevos símbolos esperados es $t_1\frac{1}{n} - t_2\frac{1}{n^2} + t_3\frac{1}{n^3} - \cdots$. Esta expresión se convierte en la probabilidad de que el símbolo siguiente sea nuevo, por lo que se utiliza en PPMP como la probabilidad de `escape`. Observe que cuando t_1 pasa a ser cero, esta expresión es

normalmente negativa y no se puede utilizar como una probabilidad. Además, el caso $t_1 = n$ produce una probabilidad de aparición del símbolo `escape` de 1 y debe evitarse. En ambos casos se requieren correcciones a la suma anterior.

PPMX utiliza el valor aproximado t_1/n (el primer término de la suma) como probabilidad del símbolo `escape`. Esta expresión también falla cuando t_1 pasa a ser 0 ó n, por lo que en estos casos PPMX se modifica para PPMXC, que utiliza la misma probabilidad para el símbolo `escape` que PPMC.

Los experimentos con las cinco variantes muestran que las diferencias entre ellas son pequeñas. La versión X es indistinguible de la P, y ambas son un poco mejores que las A-B-C. La versión C es ligeramente mejor y en consecuencia preferible a las versiones A y B.

De nuevo hay que señalar que las probabilidades para el símbolo `escape` asignadas en las variantes A-B-C se basan en la experiencia y la intuición, no en una teoría subyacente. La experiencia con estas variantes indica que el algoritmo PPM es robusto y no se ve afectado demasiado por la precisión del camino seguido para el cálculo de las probabilidades del símbolo `escape`. Las variantes P y X se basan en la teoría, pero incluso estas no mejoran significativamente el rendimiento del método PPM.

2.18.5. Detalles de la implementación

El principal problema en cualquier aplicación práctica del PPM es mantener una estructura de datos donde todos los contextos (órdenes de 0 a N) de cada símbolo leído de la secuencia de entrada se almacenen y puedan localizarse rápidamente. La estructura aquí descrita es un tipo especial de árbol, llamado *trie*. Se trata de un árbol en el que la estructura de las ramas en cualquier nivel se determina sólo por parte de un elemento de datos, no por el elemento completo (Figura 3.11). En el caso del PPM, un contexto de orden N es una cadena que incluye todos los contextos más cortos de los órdenes, desde $N-1$ hasta 0; por lo que cada contexto añade, efectivamente, sólo un símbolo para el trie.

La Figura 2.74 muestra cómo se construye un trie para la cadena "`zxzyzxxyzx`" suponiendo $N = 2$. Basta un rápido vistazo para apreciar que el árbol crece en anchura, pero no en profundidad. Su profundidad permanece en $N + 1 = 3$, sin importar la cantidad de datos de entrada ya leídos. Su anchura crece a medida que se introducen más y más símbolos, pero no a un ritmo constante. A veces, no se añaden nuevos nodos, como en el caso 10, cuando se lee la última `x`. En otras ocasiones, se añaden hasta tres nodos, como en los casos 3 y 4, cuando se agregan la segunda `z` y la primera `y`.

El nivel 1 del trie (justo bajo la raíz) contiene un nodo para cada símbolo leído hasta el momento. Estos son los contextos de orden 1. El nivel 2 contiene todos los contextos de orden 2, y así sucesivamente. Cada contexto se puede encontrar a partir de la raíz y desplazándose hacia una de las hojas. En el caso 3, por ejemplo, los dos contextos son `xz` (el símbolo `z` precedido por el contexto de orden 1 de `x`) y `zxz` (el símbolo `z` precedido por el contexto de orden 2 de `zx`). En el caso 10, hay siete contextos que van desde `xxy` y `xyz` —a la izquierda—, hasta `zxz` y `zyz` —a la derecha—.

Los números en los nodos son contadores del contexto. "`z`,4" en la rama derecha del caso 10 implica que `z` se ha visto cuatro veces. "`x`,3" e "`y`,1", de más abajo significan que estas cuatro ocurrencias se han encontrado seguidas por `x` tres veces, y por `y` una vez. Los nodos circulares muestran los distintos órdenes del contexto del último símbolo añadido al trie. En el caso 3, por ejemplo, el segundo `z` acaba de ser leído y añadido al trie. Se ha introducido dos veces: debajo de la `x` de la rama izquierda y bajo la `x` de la rama derecha (éste último se indica mediante una flecha). Además, el contador del `z` original se ha incrementado a 2. Esto demuestra que el nuevo `z` sigue los dos contextos: `x` (de orden 1) y `zx` (de orden 2).

Ahora debería ser fácil para el lector a seguir los diez pasos de la construcción del árbol y comprender intuitivamente cómo se agregan los nodos y se actualizan los contadores. Tenga en cuenta que, en cada paso, están involucrados tres nodos —o, en general, $N + 1$ nodos, uno en cada nivel del trie—

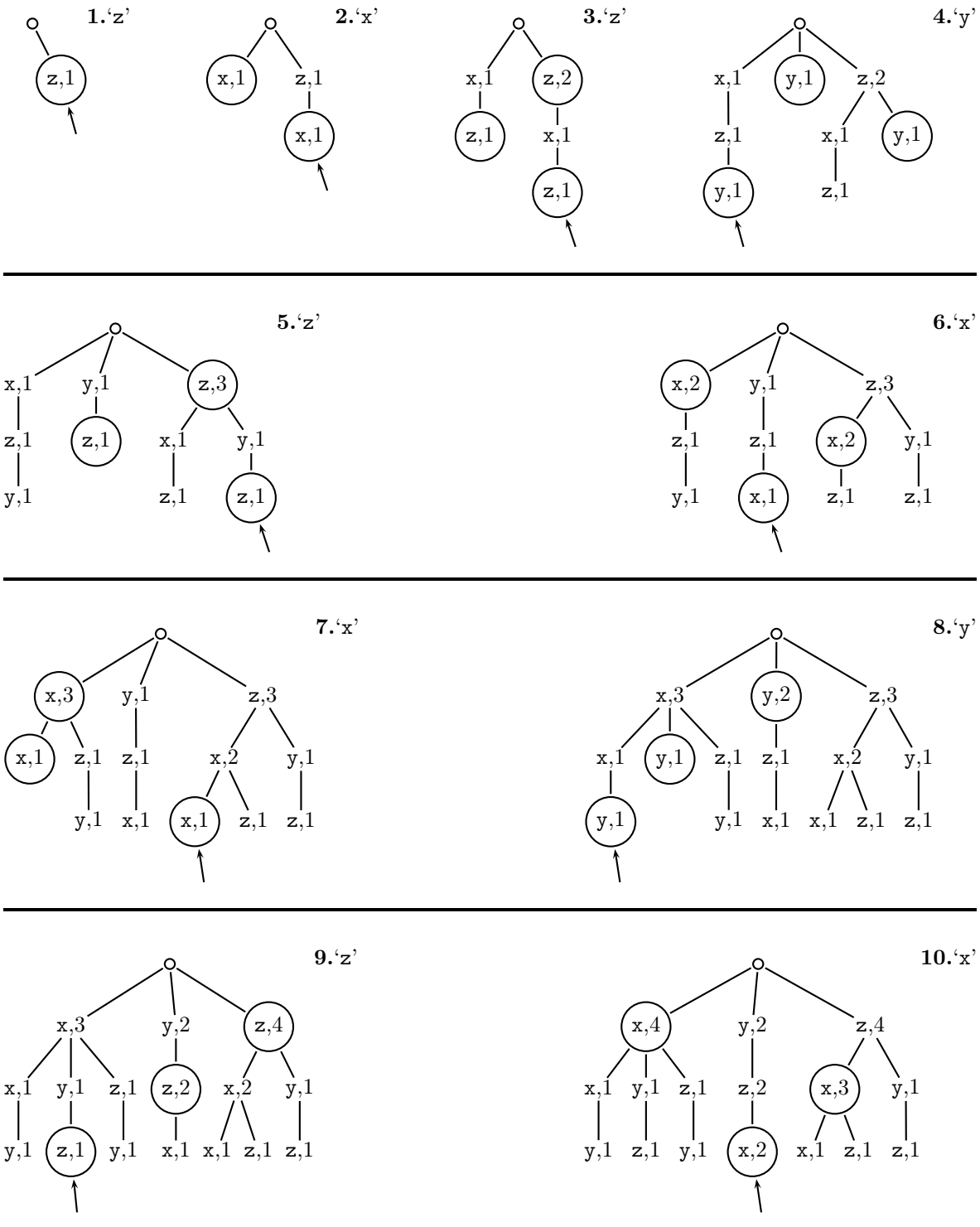


Figura 2.74: Diez tries de "zxzyzxxyzx".

(excepto en los primeros pasos, en los que el trie no ha alcanzado aún su altura final). De los tres, algunos son nuevos nodos que se añaden al trie; otros han incrementado sus contadores.

El siguiente punto que se debe discutir es cómo decide el algoritmo qué nodos actualiza y cuáles agrega. Para simplificar el algoritmo, se añade un puntero más a cada nodo, que apunta hacia atrás hasta el nodo que representa el siguiente contexto más corto. Un puntero que apunta hacia atrás en un árbol se llama "*vine pointer*".

Las Figuras 2.75a, 2.75b y 2.75c muestran los diez primeros pasos en la construcción del trie PPM para la cadena de 14 símbolos "`assanissimassa`". Cada uno de los diez pasos muestra los punteros *vine* (las líneas discontinuas en la figura) construidos por el algoritmo de actualización del trie, mientras ejecutaba el paso correspondiente. Tenga en cuenta que los punteros *vine* no se eliminan; simplemente no se muestran en los diagramas posteriores. En general, un puntero *vine* apunta desde un nodo `X` en el nivel n a un nodo con el mismo símbolo `X` en el nivel $n-1$. Todos los nodos del nivel 1 apuntan a la raíz.

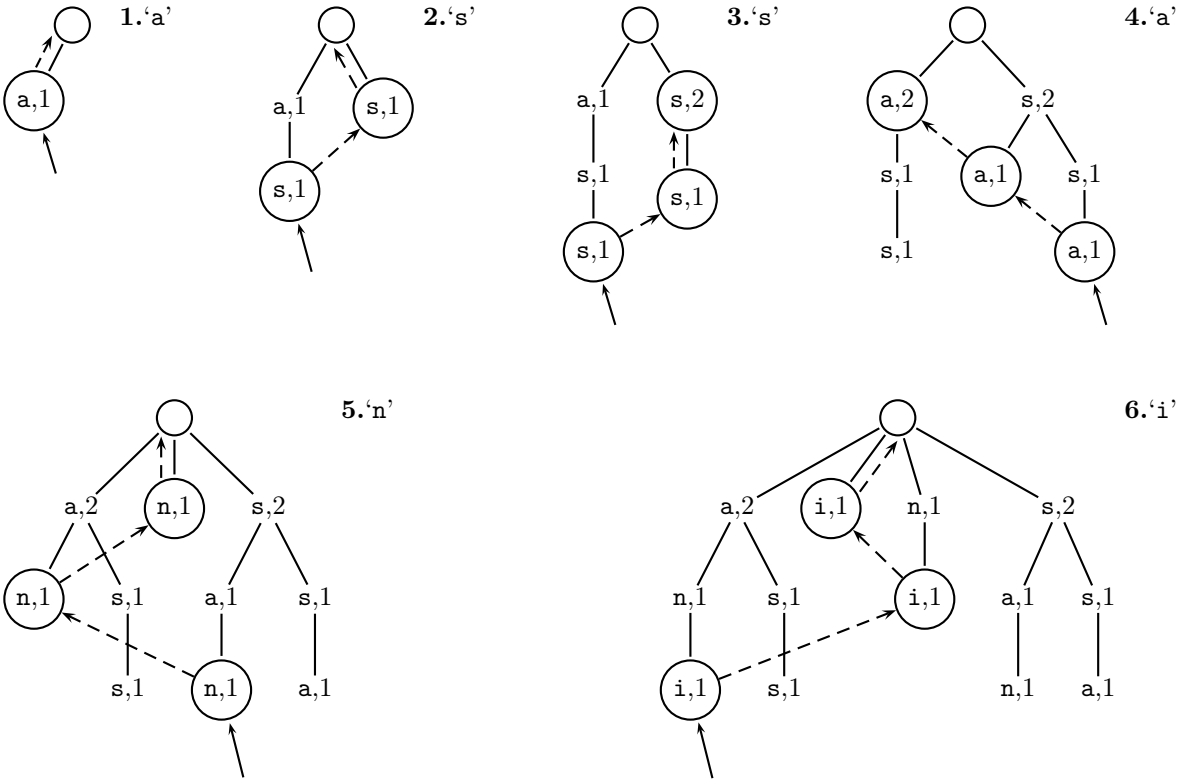


Figura 2.75: Parte I. Seis primeros tries de "`assanissimassa`".

Un nodo en el trie PPM, por lo tanto, consta de los siguientes campos:

1. El código (ASCII u otro) del símbolo.
2. El contador.
3. Un puntero hacia abajo, que señala al hijo más a la izquierda del nodo. En la Figura 2.75b, caso 10, por ejemplo, el hijo más a la izquierda de la raíz es "**a,2**". El de "**a,2**" es "**n,1**" y el de "**s,4**" es "**a,1**".

4. Un puntero derecho, que apunta al hermano siguiente del nodo. La raíz no tiene hermanos derechos. El hermano siguiente del nodo "**a,2**" es "**i,2**" y el de "**i,2**" es "**m,1**".

5. Un puntero *vine*. Éstos se muestran en la Figura 2.75 como flechas discontínuas.

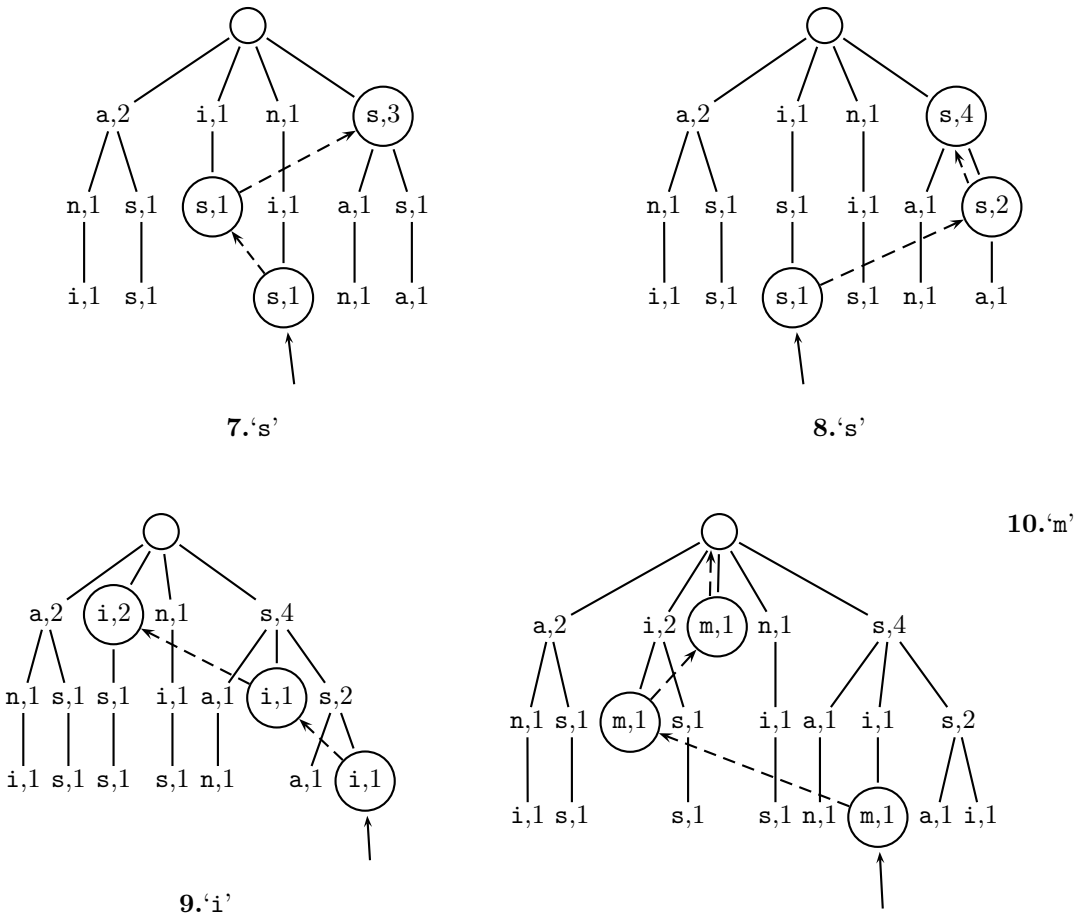


Figura 2.75: (Continuación) Los cuatro tries siguientes de "`assanissimassa`".

◇ **Ejercicio 2.52 (sol. en pág. 1069):** Complétese la construcción de este trie y muéstrese su estructura tras la introducción de los 14 caracteres.

En cualquier paso durante la construcción del trie, se mantiene un puntero llamado *base*, que apunta al último nodo añadido o actualizado en el paso anterior. Este puntero se muestra como una flecha, en trazo continuo, en la Figura 2.75. Supongamos que el símbolo `S` ya se ha introducido y el trie debe actualizarse en este momento. El algoritmo para agregar y/o actualizar los nodos es el siguiente:

1. Siga el puntero *base* hasta el nodo `X`. Siga el puntero *vine* desde `X` hasta `Y` (nótese que `Y` puede ser la raíz). Añada `S` como un nuevo nodo hijo de `Y` actualizando el puntero base para que apunte a él. No obstante, si `Y` ya tiene un nodo hijo conteniendo a `S`, se incrementará el contador de este nodo en 1 (y también se actualizará el puntero base para que apunte a él). Llame a este nodo A.

2. Repita el mismo paso, pero sin la actualización del puntero *base*. Siga el puntero *vine* desde `Y` hasta `Z`. Añada `S` como un nuevo nodo hijo de `Z`, o actualice un hijo existente. Llame a este nodo `B`. Si no existe un puntero *vine* desde `A` hasta `B`, cree uno. (Si tanto `A` como `B` son nodos viejos, ya tendrán un puntero *vine* desde `A` hasta `B`.)

3. Repita hasta que haya añadido (o incrementado) un nodo en el nivel 1.

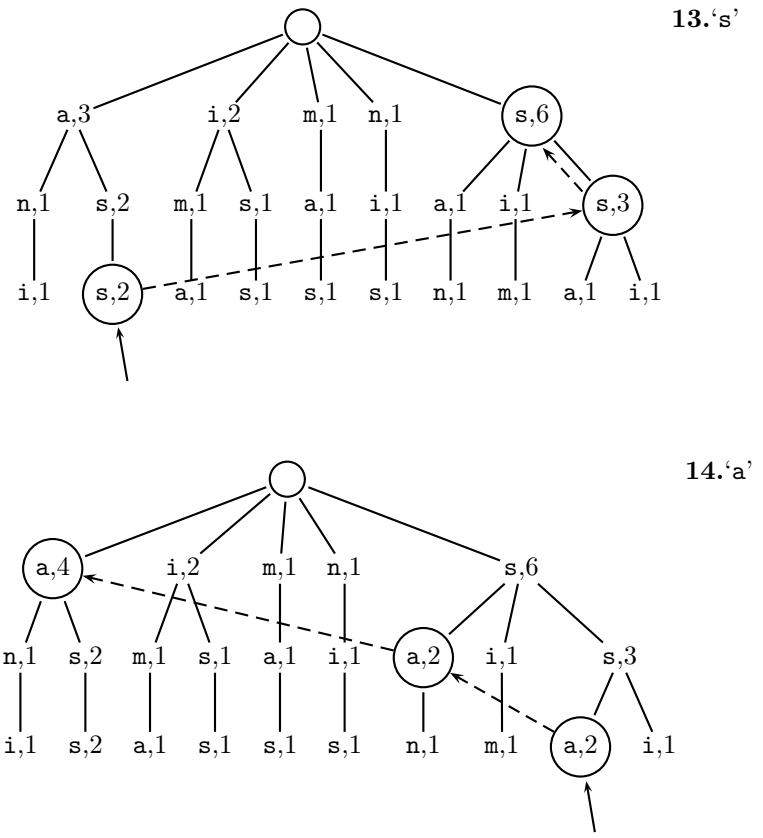


Figura 2.75: (Continuación) Los dos tries finales de "`assanissimassa`".

Durante estos pasos, el codificador PPM también recoge el valor de los contadores que se necesitan para calcular la probabilidad de aparición del nuevo símbolo `S`. La Figura 2.75c muestra el trie después de añadir los dos últimos símbolos, `s` y `a`. En la Figura 2.75, caso 13, se ha seguido un puntero *vine* desde el nodo "**s, 2**", al nodo "**s, 3**", que ya contaba con los dos hijos: "**a, 1**" e "**i, 1**". El primer hijo se ha incrementado a "**a, 2**". En la Figura 2.75, caso 14, el subárbol con los tres nodos "**s, 3**", "**a, 2**" e "**i, 1**", informa al codificador de que `a` se ha visto siguiendo al contexto `ss` dos veces, e `i` se ha visto siguiendo el mismo contexto una vez. Puesto que el árbol tiene dos hijos, el símbolo `escape` recibe un incremento en su contador de 2, quedando éste en un valor de 5. La probabilidad de `a` es, por tanto $^2/_5$ (compárese con la Tabla 2.73). Tenga en cuenta que los pasos 11 y 12 no se muestran. El lector serio debería dibujar los árboles de estos pasos como ejercicio voluntario (i.e, sin una respuesta).

Ahora es fácil entender la razón por la cual este particular trie, es tan útil. Cada vez que un símbolo es introducido, el algoritmo necesita, como mucho, $N + 1$ pasos para actualizar el trie y recoger los

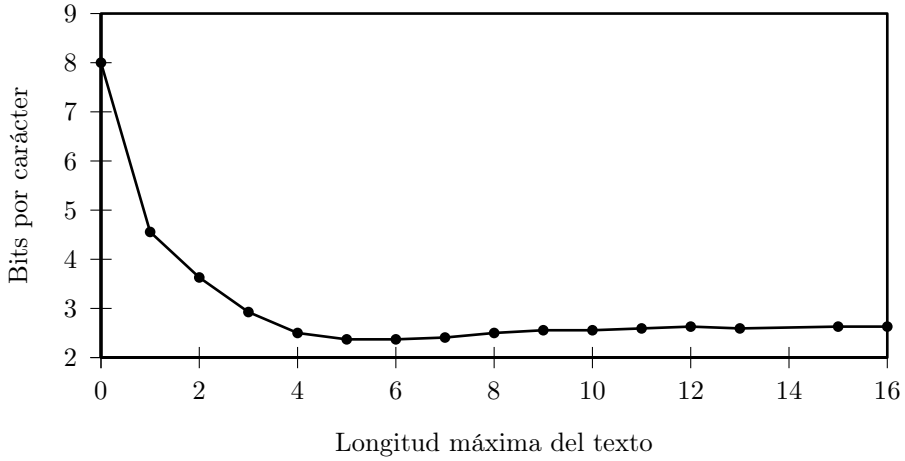


Figura 2.76: Razón de compresión en función de la longitud máxima del contexto.

valores de los contadores necesarios al recorrerlo desde el puntero base a la raíz. La adición de un símbolo al trie y su codificación necesita $O(N)$ pasos, independientemente del tamaño del trie. Puesto que N es pequeño (típicamente 4 ó 5), una implementación puede hacerse lo suficientemente rápida como para ser práctica, incluso si el trie es muy grande. Si el usuario especifica que el algoritmo debería utilizar exclusiones, se hace más complejo, ya que se tienen que mantener, en cada paso, una lista de los símbolos que deben excluirse.

Como se ha señalado, se añaden entre 0 y 3 nodos al trie por cada símbolo introducido y codificado (en general, entre 0 y $N+1$ nodos). El trie, por lo tanto, puede hacerse muy grande y llenar cualquier espacio de memoria disponible. Una solución elegante, adoptada en [Moffat 90], es descartar el trie cuando se llena y empezar a construir uno nuevo. Con la finalidad de "poner al día" al nuevo tree rápidamente, los últimos 2048 símbolos introducidos se guardan siempre en un buffer circular en la memoria y se utilizan para construir el nuevo trie. Esto reduce la cantidad de código ineficiente generado cuando se sustituyen los tries.

2.18.6. PPM*

Una característica importante del método original PPM es el uso de una longitud fija, limitada al contexto inicial. El método selecciona un valor N para la longitud de contexto y trata siempre de predecir (i.e., asignar probabilidades a) el siguiente símbolo `S` comenzando con un contexto C de orden N. Si `S` no ha aparecido hasta ahora en el contexto C, PPM cambia a un contexto menor. Intuitivamente, parece que un contexto largo (con un valor de N alto) puede producir una mejor predicción; no obstante, en la Sección 2.18 se explican los inconvenientes de los contextos largos. En la práctica, las implementaciones del método PPM tienden a utilizar 5 ó 6 como valores de N (Figura 2.76).

El método PPM*, debido a [Cleary et al. 95] y [Cleary y Teahan 97], trata de aumentar el valor de N indefinidamente. Los desarrolladores trataron de encontrar maneras de utilizar valores de N ilimitados con el fin de mejorar la compresión. El método resultante requiere una nueva estructura de datos para el tree y más recursos de cálculo que el PPM original; pero a cambio, proporciona una mejora de la compresión de alrededor del 6 % sobre el PPMC.

(En matemáticas, cuando un conjunto S se compone de a_i símbolos, se utiliza la notación S^* para representar al conjunto de todas las cadenas de símbolos a_i.)

Uno de los problemas con los contextos largos son los símbolos de `escape`. Si el codificador introduce el símbolo siguiente `S`, comienza con un contexto de orden 100, y no encuentra ninguna ocurrencia de

la cadena de referencia de 100 elementos —seguida por S—, entonces tiene que emitir un símbolo de `escape` y probar con un contexto de orden 99. Este algoritmo puede dar lugar a una secuencia de 100 símbolos de `escape` consecutivos que son producidos por el codificador, los cuales pueden causar una expansión considerable. Por tanto, es importante permitir contextos de diferentes longitudes —no sólo los contextos largos— y decidir la longitud de un contexto en función de la situación actual. La única restricción es que el decodificador debe ser capaz de averiguar la longitud del contexto utilizado por el codificador en cada símbolo codificado. Esta idea es el principio fundamental subyacente en el diseño del método PPM *.

Una idea inicial fue mantener un registro, para cada contexto, de su rendimiento anterior. Este registro se reflejaba por el decodificador; por ello, tanto el codificador como el decodificador podía utilizar, en cualquier momento, el contexto que mejor comportamiento había tenido en el pasado. Esta idea no parecía funcionar y los desarrolladores de PPM* tuvieron que enfrentarse también con la tarea de tener que explicar por qué no trabajaba como se esperaba.

El algoritmo finalmente seleccionado para PPM* depende del concepto de contexto determinista. Un contexto se define como determinista cuando ofrece sólo una predicción. Por ejemplo, el contexto `this␣is␣my` es determinista si todas las apariciones de éste introducidas hasta el momento van seguidas por el mismo símbolo. Los experimentos indican que si un contexto C es determinista, la posibilidad de que cuando se detecte la próxima vez, esté seguido por un nuevo símbolo es más pequeña de lo que se espera en una distribución uniforme de los símbolos precedentes. Esta característica sugiere el uso de contextos deterministas para la predicción en la nueva versión del método PPM.

Basándose en la experiencia con los contextos deterministas, los desarrolladores han llegado al siguiente algoritmo para PPM*: Cuando se introduce el siguiente símbolo `S`, se buscan todos sus contextos tratando de encontrar contextos deterministas de `S`. Si se encuentran tales contextos, se utiliza el más corto de ellos. Si no se detectan contextos deterministas, se usa el contexto no determinista más largo.

El resultado de esta estrategia para PPM* es que los contextos no deterministas se utilizan la mayor parte del tiempo, y casi siempre son de 5-6 símbolos de longitud; los mismos que en el PPM tradicional. Sin embargo, de vez en cuando se utilizan contextos deterministas que se alargan a medida que se leen y se procesan más datos. (En experimentos realizados por los desarrolladores, los contextos deterministas comenzaron con una longitud de 10 y llegaron a una longitud de 20-25 elementos tras la introducción de cerca de 30 000 símbolos). El uso de contextos deterministas produce una predicción muy exacta, que es el principal contribuyente al ligeramente mejor rendimiento de PPM* sobre PPMC.

Una implementación práctica de PPM* tiene que resolver el problema de mantener la pista de los contextos largos. Cada vez que se introduce un símbolo `S`, deben comprobarse todas sus apariciones pasadas, junto con todos los contextos —cortos y largos, deterministas o no—, para cada ocurrencia. En principio, esto se puede hacer simplemente manteniendo el archivo de datos completo en la memoria y revisándolo de nuevo para cada símbolo. Imagine el símbolo en la posición i en el archivo de entrada. Está precedido de $i-1$ símbolos, por lo que se necesitan $i-1$ pasos para buscar y encontrar todos sus contextos. El número total de pasos para n símbolos es, por lo tanto, $1+2+\cdots+(n-1) = n\,(n-1)\,/2$. Para n grande, esto equivale a una complejidad de $O\left(n^2\right)$; demasiado lento para las implementaciones prácticas. Este problema fue resuelto con un trie especial, llamado *trie de contexto*, en el que un nodo hoja apunta hacia atrás a la cadena de entrada siempre que un contexto es único. Cada nodo corresponde a un símbolo que sigue algún contexto y el contador de la frecuencia del símbolo se almacena en el nodo.

PPM* utiliza el mismo mecanismo de símbolos escape que el PPM original. La implementación mostrada en las publicaciones PPM utiliza el algoritmo PPMC para asignar probabilidades a los diferentes símbolos `escape`. Tenga en cuenta que el PPM original utiliza cada vez menos `escape`s a medida que pasa el tiempo, se introducen más datos y se disponen de más predicciones de contexto. Por el contrario, PPM* tiene que utilizar `escape`s muy a menudo, sin importar la cantidad de datos ya introducidos; sobre todo por el uso de contextos deterministas. Este hecho hace que el problema

del cálculo de las probabilidades de los símbolos de escape sea especialmente grave.

La compresión de todo el *Calgary corpus* por PPM* produjo un promedio de 2,34 bpc, en comparación con los 2,48 bpc alcanzados por PPMC. Esto representa una mejora de la compresión de alrededor del 6 % porque 2,34 es el 94,4 % de 2,48.

2.18.7. PPMZ

La variante PPMZ, creada e implementada por Charles Bloom [Bloom 98], es un intento de mejorar el algoritmo original PPM. Parte de la premisa de que PPM es un potente algoritmo que puede, en principio, comprimir los datos hasta su valor de entropía, en especial cuando trata grandes cantidades de datos; pero es menos óptimo —en la práctica— debido a una deficiente manipulación de características como: los contextos deterministas, contextos sin límites de longitud y la estimación del orden local. PPMZ intenta manejar estas características de forma óptima y acaba logrando un rendimiento superior.

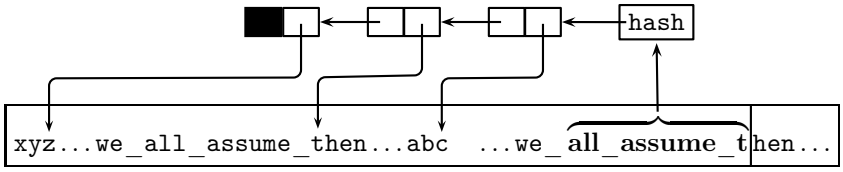
El algoritmo PPMZ comienza, similarmente a PPM*, explorando el contexto máximo determinista del símbolo actual. Si no se encuentra ningún contexto determinista, el codificador PPMZ ejecuta un procedimiento de estimación de orden local (LOE), para calcular un orden en el intervalo [0,12] y usarlo para predecir el símbolo actual como hace el algoritmo original PPM. Además, PPMZ utiliza un modelo secundario para predecir las probabilidades de los distintos símbolos `escape`.

El creador del método se dio cuenta de que las distintas implementaciones del método PPM comprimen datos a unos 2 bpc; de los cuáles la mayoría de los caracteres están comprimidos a 1 bpc cada uno, y los caracteres restantes representan, ya el inicio de la secuencia de entrada, ya datos aleatorios. La conclusión natural es que cualquier pequeña mejora en la estimación de probabilidad de los caracteres más comunes, puede conducir a mejoras significativas en el rendimiento global del método. Empezamos hablando de cómo maneja PPMZ los contextos sin límites.

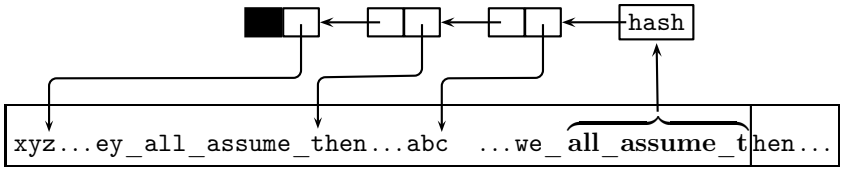
La Figura 2.77a muestra una situación en la que el carácter actual es `e`, y su contexto de orden 12 es `ll␣assume␣th`. El contexto es controlado por un puntero P, que señala a un lista enlazada. Los nodos de la lista apuntan a todas las cadenas de 12 caracteres de la secuencia de entrada que pasan a ser información dependiente del mismo puntero P. (Cada nodo tiene un campo contador que indica el número de veces que la cadena apuntada por el nodo ha sido una coincidencia.) El codificador sigue los punteros, en busca de una ocurrencia cuya longitud mínima varíe de un contexto a otro. Suponiendo que la longitud mínima de coincidencia en nuestro caso es 15, el codificador localizará la cadena de 15 caracteres `e␣all␣assume␣th` (la `w` anterior hace que coincidan 16 caracteres, e incluso puede ser más larga). El carácter actual `e`, se codifica con una probabilidad determinada por el número de veces que éste se ha encontrado en el pasado, y el contador de coincidencias (en el nodo correspondiente de la lista) se actualiza.

La Figura 2.77b muestra una situación en la que no se encuentra ninguna coincidencia determinista. El carácter actual es de nuevo `e` y su contexto de orden 12 es el mismo: `ll␣assume␣th`, pero la única cadena `ll␣assume␣th` en el archivo de datos está precedido por los tres caracteres `y␣a`. El codificador no encuentra una coincidencia de 15 caracteres y procede de la siguiente manera:

1. Produce un símbolo `escape` para indicar que no hay coincidencias deterministas.
2. Invoca al procedimiento de LOE para calcular un orden.
3. Utiliza el orden para predecir el símbolo actual de forma similar a como lo haría PPM.
4. Realiza unos pasos para asegurar que este caso se produzca de nuevo. En el primero de estos pasos, el codificador añade un nodo a la lista y hace que apunte al nuevo contexto de 12 caracteres: `ll␣assume␣th`. El segundo paso, incrementa la longitud mínima de coincidencia de ambos contextos en 1 (i.e., a 16 caracteres). Esto asegura que estos dos contextos se utilizarán en el



(a)



(b)

Figura 2.77: Contextos deterministas sin límites de longitud en PPMZ.

futuro sólo cuando el codificador pueda emparentar suficientes caracteres para distinguir entre ellos.

Este complejo procedimiento se ejecuta por el codificador PPMZ para garantizar que todos los contextos sin límites de longitud sean deterministas.

La estimación del orden local es otra innovación de PPMZ. El PPM tradicional utiliza el mismo valor de N (para la longitud máxima del orden, típicamente 5-6) para todas las cadenas introducidas, pero una versión más sofisticada debe tratar de estimar distintos valores de N para los diferentes archivos de entrada o incluso para los diferentes contextos existentes dentro de esos archivos. El cálculo de la LOE realizada por PPMZ trata de decidir qué contextos de orden superior no son fiables. La LOE encuentra un contexto de coincidencias, las examina en cada orden, y calcula un *índice de fiabilidad*[58] para cada orden.

Al principio, parece que la mejor y más fiable es la entropía del contexto; porque la entropía estima la longitud de la salida en bits. En la práctica, sin embargo, esta medida subestima la fiabilidad de los contextos largos. La razón mencionada por el desarrollador del método es que un cierto símbolo `X` puede ser común en una cadena de entrada; sin embargo, un contexto específico puede incluir `X` sólo una vez.

La medida de la confianza finalmente seleccionada para la LOE se basa en la probabilidad P de los caracteres más probables en el contexto. Se estudiaron varias fórmulas que involucraban a P y todas ofrecían aproximadamente el mismo rendimiento. La conclusión fue que la mejor medida de confianza para la LOE es simplemente P mismo.

La última característica importante de PPMZ es la forma en que calcula las probabilidades del símbolo `escape`. Esto se conoce como estimación de escape secundaria o SEE[59]. La idea principal es tener un algoritmo adaptativo que calcule y determine para la secuencia de entrada, no sólo el número de `escape`s, sino también el camino seguido para calcular ese número. En cada contexto, se aplica primero el método PPMC para contar los `escape`s. Este método cuenta el número de nuevos caracteres (i.e., caracteres existentes que no han sido predichos) en el contexto. Esta información se utiliza entonces para construir un *contexto de escape* que, a su vez, se utiliza para buscar la probabilidad

[58] Confidence rating.
[59] Procede de "Secondary Escape Estimation".

del símbolo `escape` en una tabla.

El *contexto de escape* es un número construido a partir de los cuatro campos expuestos aquí, cada uno cuantificado a unos pocos bits. Se han examinado tanto la cuantización lineal como la logarítmica. La cuantificación lineal simplemente trunca los bits menos significativos. La cuantificación logarítmica calcula el logaritmo del número que va a ser cuantificado. Ésta cuantifica más los números grandes que los pequeños, con el resultado de que los valores pequeños permanecen distinguibles, mientras que los grandes valores pueden llegar a ser iguales. Los cuatro componentes del *contexto de escape* son los siguientes:

1. El orden PPM (que está entre 0 y 8), cuantificado a dos bits.
2. El contador de `escape`s, cuantificado a dos bits.
3. El número de ocurrencias encontradas con éxito (el recuento total menos el número de símbolos `escape`), cuantificado a tres bits.
4. Diez bits de los dos caracteres `xy` que preceden al símbolo actual `S`. Se toman siete bits para `x` y tres para `y`.

Este número se convierte en el contexto de orden dos de `escape`. Después de eliminar algunos bits del mismo, PPMZ también crea contextos de orden 1 y de orden 0 (15 y 7 bits de largo, respectivamente). Los contextos de `escape` se utilizan para actualizar una tabla de contadores de `escape`. Cada entrada en esta tabla corresponde a los aciertos de búsqueda codificados de los contextos pasados PPM que tenían los mismos contextos de escape. La información de la tabla se utiliza de una manera compleja para construir la probabilidad de `escape` que se envía al codificador aritmético para codificar el símbolo `escape` en sí mismo.

La ventaja de este método es que combina las estadísticas recogidas de los contextos largos (de orden alto). Estos contextos proporcionan una compresión alta, pero dispersa, haciendo que el PPM original sobreestime sus probabilidades de `escape`.

Aplicada al *Calgary corpus*, PPMZ produjo un promedio de 2,119 bpc. Esto es un 10 % mejor que 2,34 bpc, obtenido por PPM* y un 17 % mejor que 2,48 bpc, alcanzado por PPMC.

2.18.8. PPM rápido

El método PPM rápido es una variante del PPM desarrollado e implementado por [Howard y Vitter 94b], como un compromiso entre la velocidad y el rendimiento del PPM. A pesar de que es reconocido como uno de los mejores (si no el mejor) método de compresión estadístico, PPM no es muy popular porque es lento. La mayor parte de las implementaciones de software de compresión sin pérdidas de propósito general eligen un método basado en diccionarios. PPM rápido intenta aislar esas características de PPM, que contribuyen sólo marginalmente al rendimiento de la compresión y las sustituye por aproximaciones. Se esperaba que esto iba a acelerar la ejecución y convertir esta versión en competitiva con los productos comerciales de compresión sin pérdidas más comunes.

PPM tiene dos aspectos principales: el modelado y la codificación. El modelado busca los contextos del símbolo actual para calcular su probabilidad. La versión rápida simplifica esta parte eliminando el uso explícito de los símbolos `escape`, calculando las probabilidades aproximadas y simplificando el mecanismo de exclusión. PPM utiliza la codificación aritmética adaptativa. La versión rápida incrementa la velocidad de esta parte mediante el uso de la codificación cuasi-aritmética, un método desarrollado por los mismos investigadores [Howard y Vitter 92c] y no que no se trata aquí.

La parte de modelado rápido de PPM se ilustra en las tablas de la Figura 2.78. Suponemos que el flujo de entrada comienza con la cadena `abcbabdbaeabbabe` y que el símbolo actual es el segundo `e` (el último carácter de la cadena). En la parte (a) de la tabla se muestran los contextos de este carácter comenzando por el orden 3. El contexto de orden 3 de esta `e` es `bab`, que se ha visto una sóla vez en

Orden	Contexto	Símbolo	Contador	Acción
3	`bab`	`d`	1	$NF, \rightarrow 2$
2	`ab`	`c`	1	NF
		`d`	1	*excluído*
		`b`	1	$NF, \rightarrow 1$
1	`b`	`c`	1	*excluído*
		`a`	3	NF
		`d`	1	*excluído*
		`b`	1	*excluído*, $\rightarrow 0$
0		`a`	5	*excluído*
		`b`	7	*excluído*
		`c`	1	*excluído*
		`d`	1	*excluído*
		`e`	1	*encontrado*

(a)

Orden	Contexto	Símbolo	Contador	Acción
3	`eab`	—	—	$\rightarrow 2$
2	`ab`	`c`	1	NF
		`d`	1	$NF, \rightarrow 1$
1	`b`	`c`	1	*excluído*
		`a`	2	NF
		`d`	1	*excluído*, $\rightarrow 0$
0		`a`	4	*excluído*
		`b`	5	*encontrado*

(b)

Figura 2.78: Dos ejemplos del método PPM rápido para `abcbabdbaeabbabe`.

el pasado, pero estaba seguida por una `d`, por lo que no se puede utilizar para predecir el `e` actual. El codificador, por tanto, pasa al orden 2, donde el contexto `ab` se ha encontrado tres veces, pero nunca seguido por una `e` (nótese que la `d` que está a continuación de `ab` tiene que ser excluida). Saltando al orden 1, el codificador encuentra cuatro símbolos diferentes, siguiendo el contexto de orden 1 de `b`: `c`, `a`, `d` y `b`. De éstos, `c`, `d` y `b` ya han sido vistos tras los contextos largos, por lo que se excluyen, y la `d` es designada NF (*Not Found*[60]), porque estamos buscando una `e`. Bajando hasta el orden 0, el codificador finalmente encuentra una `e`, tras `a`, `b`, `c` y `d`, que están todos los excluidos. La cuestión es que tanto el codificador como el decodificador del PPM rápido pueden generar esta tabla con la información disponible sobre ellos. Todo lo que el decodificador necesita para decodificar la `e` es el número de NFs (4 en el ejemplo) existentes en la tabla.

La parte (b) de la tabla ilustra la situación cuando la sexta `b` (hay siete `b`s en total) es el carácter actual. Esto demuestra que este carácter puede ser identificado por el decodificador codificando tres NFs y escribiéndolos en la cadena comprimida.

Una forma diferente de ver esta parte del PPM rápido es imaginar que el codificador genera una lista de símbolos, empezando por el nivel más alto, eliminando los duplicados. La lista para la parte (a) de la tabla está formada por los elementos `dcbae` (cuatro NFs seguidos por un F[61]), mientras que la lista para la parte (b) es `cdab` (tres NFs seguidos por un F).

De este modo, el método PPM rápido codifica cada carácter mediante la codificación de una secuencia de NFs, seguidos por un F (found). Por lo tanto, emplea un codificador aritmético binario.

[60] No encontrado.
[61] Proviene de "Found": encontrado.

Para incrementar la velocidad, utiliza la cuasi-codificación aritmética, en lugar del codificador QM, más común, de la Sección 2.16. Para un funcionamiento aún más rápido, se usa el codificador cuasi-aritmético para codificar los NFs sólo para los símbolos con mayor probabilidad; a continuación, utiliza un código Rice (Sección 7.9) para codificar la posición de los símbolos (`e` o `b` en nuestro ejemplo) en el resto de la lista. Las variantes del PPM rápido pueden eliminar el codificador cuasi-aritmético por completo (para obtener la máxima velocidad) o emplearlo durante todo el proceso (lo que determina la compresión máxima).

Los desarrolladores han proporcionado los resultados de la aplicación del PPM rápido al *Calgary corpus* y parecen justificar sus esfuerzos de desarrollo. El rendimiento de la compresión es de 2,341 bpc (para la versión con un solo codificador cuasi-aritmético) y 2,287 bpc (para la versión con ambos codificadores: el cuasi-aritmético y el código Rice). Esto es algo peor que los 2,074 bpc alcanzados por PPMC. Sin embargo, la velocidad del PPM rápido es de unos 25 000 a 30 000 caracteres por segundo, que comparado con los cerca de 16 000 caracteres por segundo de PPMC-a, ¡se obtiene un factor de aceleración de 2!

> El razonamiento temporal implica tanto la predicción como la explicación. La predicción es una proyección hacia delante desde las causas hasta los efectos; mientras que la explicación es la proyección hacia atrás desde los efectos hasta las causas. Esto es, la predicción es el razonamiento desde los eventos hasta las propiedades y los acontecimientos que los producen; mientras que la explicación, es el razonamiento desde las propiedades y los eventos hasta los hechos que puedan haberlos causado. Aunque es evidente que un marco completo para razonamientos temporales debería dar facilidades para resolver tanto los problemas de predicción como los de explicación; la predicción ha recibido mucha más atención en la literatura de razonamiento temporal que la explicación.
>
> —Murray Shanahan, *Proceedings*[a] IJCAI 1989

[a]Procedimientos.

2.19. Ponderación del árbol del contexto

Sea lo que sea la secuencia de entrada: texto, píxeles, sonido o cualquier otra cosa, se puede considerar una cadena binaria. La compresión ideal (i.e., la compresión en o muy cercana a la entropía de la cadena) podría lograrse si pudiéramos utilizar los bits que se han introducido hasta el momento actual para predecir con certeza (i.e., con probabilidad 1) el valor del siguiente. En la práctica, lo mejor que podemos esperar es utilizar la historia para estimar la probabilidad de que el siguiente bit sea 1. El método [Willems et al. 95] de ponderación del árbol del contexto[62] (CTW) se inicia con una cadena de bits determinada $b_1^t = b_1 b_2 \ldots b_t$ y los d bits que la preceden $c_d = b_{-d} \ldots b_{-2} b_{-1}$ (el contexto de b_1^t). Las dos cadenas, c_d y b_1^t, constituyen la secuencia de entrada. El método utiliza un sencillo algoritmo para construir un árbol de profundidad d basado en el contexto, en el que cada nodo corresponde a una subcadena de c_d. Después, se introduce y se examina el primer bit b_1. Si es 1, el árbol se actualiza para incluir las subcadenas de $c_b b_1$ y se utiliza para calcular (o estimar) la probabilidad de que b_1 sea 1, dando el contexto c_d. Si b_1 es cero, el árbol se actualiza de manera diferente y el algoritmo calcula de nuevo (o estima) la probabilidad de que b_1 sea cero, dado el mismo contexto. El bit b_1 y su probabilidad se envían a un codificador aritmético, y el proceso continúa con b_2. Los bits del contexto en sí mismos se escriben en la cadena comprimida en formato puro.

La profundidad d del árbol del contexto es fija durante todo el proceso de compresión, y debe depender de la correlación esperada entre los bits de entrada. Si se espera que los bits estén altamente

[62]Context-Tree Weighting.

$\boldsymbol{\theta}$:	0	$1/5$	$2/5$	$1/2$	$3/5$	$4/5$	$5/5$
$\boldsymbol{P_a\,(2,3)}$:	0	0,00512	0,02304	0,03125	0,03456	0,02048	0

Tabla 2.79: Siete valores de $P_a\,(2,3)$.

correlacionados, un d pequeño puede ser suficiente para obtener buenas predicciones de probabilidad y por lo tanto una buena compresión.

Considerando la entrada como una cadena binaria, es costumbre usar el término "fuente". Vemos los bits de datos entrantes como provenientes de una fuente de información determinada. La fuente puede ser *sin memoria* o puede tener memoria. En el primer caso, cada bit es independiente de sus predecesores; en el último, cada bit depende de algunos de sus predecesores (y, quizás, también de sus sucesores, pero éstos no pueden utilizarse porque no están disponibles para el decodificador), por lo que están correlacionados.

Empezamos viendo una fuente sin memoria, donde cada bit tiene una probabilidad $P_a\,(1)$ de ser un 1 y una probabilidad $P_a\,(0)$ de ser un 0. Llamamos $\theta = P_a\,(1)$, por tanto, $P_a\,(0) = 1-\theta$ (el subíndice a procede de "actual probability" —probabilidad efectiva o real—). La probabilidad de una cadena determinada b_1^t que es generada por la fuente se denota por $P_a\,(b_1^t)$, y es igual al producto:

$$P_a\left(b_1^t\right) = \prod_{i=1}^{t} P_a\,(b_i)\,.$$

Si la cadena b_1^t contiene a ceros y b unos, entonces $P_a\,(b_1^t) = (1-\theta)^a\,\theta^b$.

Ejemplo: Sea $t=5$, $a=2$ y $b=3$. La probabilidad de generar una cadena binaria de 5 bits con dos ceros y tres unos es: $P_a\left(b_1^5\right) = (1-\theta)^2\,\theta^3$. La Tabla 2.79 muestra los valores de $P_a\left(b_1^5\right)$ para siete valores de θ, desde 0 hasta 1. Es fácil ver que el máximo se obtiene cuando $\theta = 3/5$. Para comprender estos valores de forma intuitiva examinemos los 32 posibles números de 5 bits. Diez de ellos están formados por dos ceros y tres unos, por lo que la probabilidad de generar tales cadenas es de $10/32 = 0{,}3125$ y la probabilidad de generar una cadena concreta de estas 10 es 0,03125. Este número se obtiene para $\theta = 1/2$.

En situaciones de la vida real no conocemos el valor de θ, así que tenemos que estimarlo basándonos en lo que se ha introducido en el pasado. Suponiendo que la cadena inmediata pasada b_1^t consiste en a ceros y b unos, tiene sentido estimar la probabilidad de que el siguiente bit sea 1 con:

$$P_e\left(b_{t+1}=1 \mid b_1^t\right) = \frac{b}{a+b},$$

donde el subíndice e significa "estimación" (la expresión anterior se lee: "la estimación de la probabilidad de que el siguiente bit b_{t+1} sea un 1, dado que sabemos que la cadena b_1^t es..."). La estimación anterior es intuitiva y no puede manejar el caso $a=b=0$. Una mejor estimación, debida a Krichevsky y Trofimov [Krichevsky 81], se llama KT y se expresa asi:

$$P_e\left(b_{t+1}=1 \mid b_1^t\right) = \frac{b+1/2}{a+b+1}.$$

La estimación KT, como la estimación intuitiva, predice una probabilidad de $1/2$ para cualquier caso en el que $a=b$. A diferencia de la estimación intuitiva, sin embargo, también funciona para el caso $a=b=0$.

El límite KT

Todo el globo terraqueo está cubierto por una capa oscura arcillosa que se depositó hace unos 65 millones de años. Esta capa está enriquecida con el raro elemento iridio. Los fósiles más antiguos encontrados en esta capa KT incluyen muchas especies de dinosaurios. Por encima de esta capa (fósiles más jóvenes), no se encuentran especies de dinosaurios. Esto sugiere que algo que sucedió alrededor al mismo tiempo que se formó el límite KT, matando a los dinosaurios. El iridio se encuentra en los meteoritos, por lo que es posible que un gran meteorito de iridio enriquecido golpeara la tierra, elevando mucho polvo de iridio hasta la estratosfera. Este polvo luego se extendió alrededor de la tierra a través de las corrientes de aire y se depositó en el suelo muy lentamente, formando posteriormente el límite KT.
Este evento se ha llamado el "Impacto KT" porque marca el final del Período Cretácico y el comienzo del Terciario. Se utiliza la letra "K" porque "C" representa el período carbonífero, que finalizó 215 millones de años antes.

◇ **Ejercicio 2.53 (sol. en pág. 1069):** Úsese la estimación KT para calcular la probabilidad de que el siguiente bit sea un cero, dada la cadena b_1^t como contexto.

Ejemplo: Utilicemos la estimación KT para calcular la probabilidad de aparición de la cadena de 5 bits 01110. La probabilidad de que el primer bit sea un cero es (ya que no hay ningún contexto) :

$$P_e\left(0 \mid \text{nulo}\right) = P_e\left(0 \mid_{a=b=0}\right) = \left(1 - \frac{0 + 1/2}{0 + 0 + 1}\right) = \frac{1}{2}.$$

La probabilidad de que sea toda la cadena es el producto:

$$\begin{aligned} P_e\left(01110\right) &= P_e\left(2,3\right) \\ &= P_e\left(0 \mid \text{nulo}\right) P_e\left(1 \mid 0\right) P_e\left(1 \mid 01\right) P_e\left(1 \mid 011\right) P_e\left(0 \mid 0111\right) \\ &= P_e\left(0 \mid_{a=b=0}\right) P_e\left(1 \mid_{a=1,b=0}\right) P_e\left(1 \mid_{a=b=1}\right) P_e\left(1 \mid_{a=1,b=2}\right) P_e\left(0 \mid_{a=1,b=3}\right) \\ &= \left(1 - \frac{0 + 1/2}{0 + 0 + 1}\right) \cdot \frac{0 + 1/2}{1 + 0 + 1} \cdot \frac{1 + 1/2}{1 + 1 + 1} \cdot \frac{2 + 1/2}{1 + 2 + 1} \cdot \left(1 - \frac{3 + 1/2}{1 + 3 + 1}\right) \\ &= \frac{1}{2} \cdot \frac{1}{4} \cdot \frac{3}{6} \cdot \frac{5}{8} \cdot \frac{3}{10} = \frac{3}{256} \approx 0{,}01172. \end{aligned}$$

En general, la probabilidad de la estimación KT para una cadena con a ceros y b unos es:

$$P_e\left(a,b\right) = \frac{1/2 \cdot 3/2 \cdots \left(a - 1/2\right) \cdot 1/2 \cdot 3/2 \cdots \left(b - 1/2\right)}{1 \cdot 2 \cdot 3 \cdots \left(a + b\right)}. \tag{2.11}$$

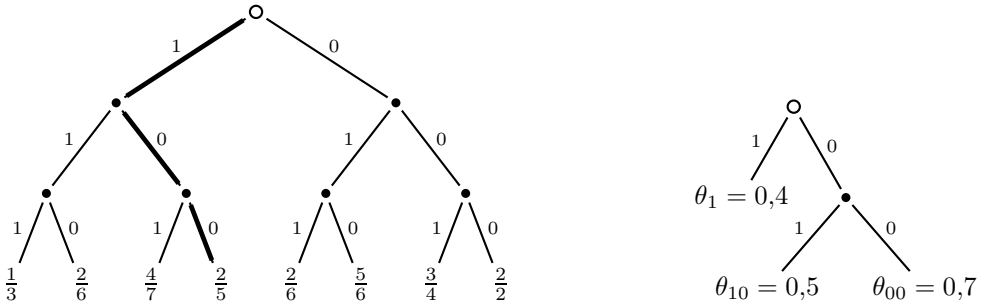
La Tabla 2.80 muestra algunos valores de $P_e\left(a,b\right)$, calculados por la ecuación (2.11). Observe que $P_e\left(a,b\right) = P_e\left(b,a\right)$, por lo que la tabla es simétrica.

Hasta ahora hemos supuesto que la fuente no tiene memoria. En una fuente de este tipo, la probabilidad θ de que el siguiente bit sea un 1 es fija. Cualquier cadena binaria, incluso si es aleatoria, se genera por la fuente con la misma probabilidad. Las cadenas binarias que tienen que ser comprimidas en situaciones reales no suelen ser aleatorias y se generan por una fuente con memoria. En éstas, θ no es fijo; varía de un bit a otro, y depende del contexto pasado del bit. Puesto que un contexto es una cadena binaria, todos los posibles contextos pasados de un bit pueden representarse con un árbol binario. Como un contexto puede ser muy largo, el árbol puede incluir sólo algunos de los últimos bits del contexto, que llamaremos *sufijo*. Como ejemplo, consideremos la siguiente cadena de 42 bits:

$$S = 000101100111010110001101001011110010101100.$$

	0	1	2	3	4	5
0	–	$1/2$	$3/8$	$5/16$	$35/128$	$63/256$
1	$1/2$	$1/8$	$1/16$	$5/128$	$7/256$	$21/1024$
2	$3/8$	$1/16$	$3/128$	$3/256$	$7/1024$	$9/2048$
3	$5/16$	$5/128$	$3/256$	$5/1024$	$5/2048$	$45/32768$
4	$35/128$	$7/256$	$7/1024$	$5/2048$	$35/32768$	$35/65536$
5	$63/256$	$21/1024$	$9/2048$	$45/32768$	$35/65536$	$63/262144$

Tabla 2.80: Estimación de KT para algunos $P_e(a, b)$.



(a) (b)

Figura 2.81: Dos árboles de sufijos.

Supongamos que estamos interesados en los sufijos de longitud 3. Los 3 primeros bits de S no tienen sufijos de longitud suficiente, por lo que se escriben tal cual son en la cadena comprimida. A continuación se examina cada sufijo de 3 bits de los últimos 39 bits de S y se cuentan cuántas veces es seguido, cada sufijo, por un 1 y cuántas, por un 0. El sufijo 001, por ejemplo, es seguido dos veces por un 1 y tres por un 0. La Figura 2.81a muestra el árbol completo de los sufijos de profundidad 3 para este caso (en general, no es un árbol binario completo). Los sufijos se leen desde las hojas hasta la raíz, y cada hoja está etiquetada con la probabilidad de que el sufijo aparezca seguido por un bit 1. Cada vez que los tres bits más leídos recientemente son 001, el codificador comienza en la raíz del árbol y sigue las ramas asociadas a 1, 0 y 0; encuentra $2/5$ en la hoja[63], por lo que debe suponer una probabilidad de $2/5$ de que el siguiente bit sea un 1, y una probabilidad de $1 - 2/5$ de que sea un 0. Entonces, el codificador introduce el bit siguiente, lo examina y lo envía, con la probabilidad correcta, para que sea codificado aritméticamente.

La Figura 2.81b muestra otro sencillo árbol de profundidad 2 que corresponde al conjunto de sufijos 00, 10 y 1. Cada sufijo (i.e., cada hoja del árbol) está etiquetado con una probabilidad θ. Así, la probabilidad de que un bit 1 aparezca[64] tras el sufijo $\ldots 10$ es de $7/12 = 0{,}58$; de encontrarlo tras el sufijo $\ldots 00$, es de $5/7 = 0{,}71$; y de hallarlo tras el sufijo $\ldots 1$, es de $9/21 = 0{,}42$.

El árbol es el *modelo* de la fuente y las probabilidades son los *parámetros*. En la práctica, ni el modelo ni los parámetros son conocidos, por lo que el algoritmo CTW debe estimarlos.

A continuación nos acercamos a situaciones de la vida real. Suponemos que el modelo es conocido y los parámetros son desconocidos, y utilizamos el estimador KT para estimar los parámetros. Como ejemplo, podemos utilizar el modelo de la Figura 2.81b, pero sin las probabilidades. Usamos la cadena $10 \mid 0100110 = 10 \mid b_1b_2b_3b_4b_5b_6b_7$, donde los dos primeros bits son el sufijo, para ilustrar cómo

[63] De las 5 apariciones de 001, dos de ellas van seguidas por un 1.
[64] De las 12 apariciones de 10, siete de ellas van seguidas por un 1.

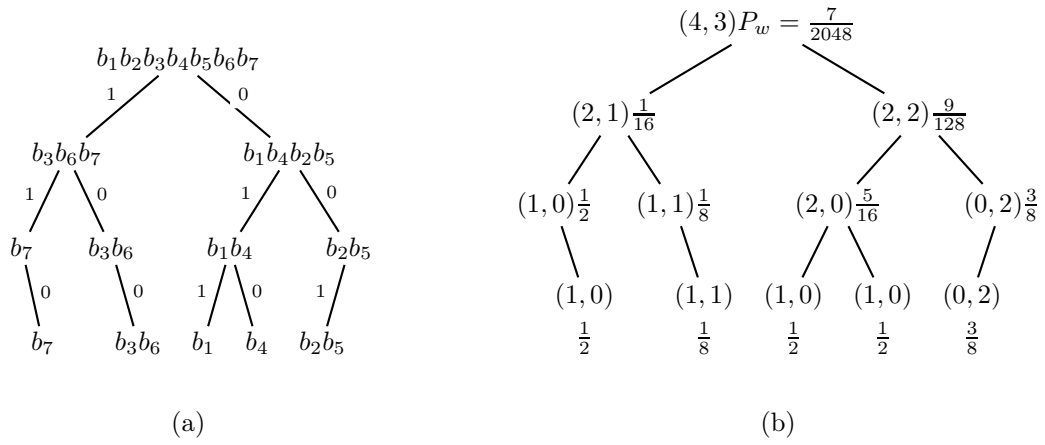


Figura 2.82: (a) Un árbol de contexto. (b) Un árbol de contexto ponderado.

se calculan las probabilidades con el estimador KT. Los bits b_1 y b_4 tienen el sufijo 10, así que la probabilidad para la hoja 10 del árbol se calcula como la probabilidad KT de aparición de la subcadena $b_1b_4 = 00$, que es el valor $P_e(2,0) = 3/8$ (dos ceros y ningun uno) de la Tabla 2.80. Los bits b_2 y b_5 tienen el sufijo 00, por lo que la probabilidad para la hoja 00 del árbol se calcula como la probabilidad KT de la subcadena $b_2b_5 = 11$, que es el valor $P_e(0,2) = 3/8$ (ningún cero y dos unos) de la Tabla 2.80. Los bits $b_3 = 0$, $b_6 = 1$ y $b_7 = 0$ tienen sufijo 1, por lo tanto, la probabilidad de la hoja 1 del árbol se estima con el valor $P_e(2,1) = 1/16$ (dos ceros y un solo uno) de la Tabla 2.80. La probabilidad de toda la cadena, 0100110, dado el sufijo 10 es, por consiguiente, el producto:

$$\frac{3}{8}\cdot\frac{3}{8}\cdot\frac{1}{16} = \frac{9}{1024} \approx 0{,}0088.$$

◇ **Ejercicio 2.54 (sol. en pág. 1069):** Úsese este ejemplo para estimar las probabilidades de las cinco cadenas: 0, 00, 000, 0000 y 00000, asumiendo que cada una va precedida por el sufijo 00.

En el último paso se supone que el modelo, así como los parámetros, se desconocen. Construimos un árbol binario de profundidad d. La raíz se corresponde con el contexto nulo, y cada nodo s corresponde a la subcadena de bits que ha sido introducida tras el contexto s. De este modo, la cadena se divide entre los nodos. La Figura 2.82a muestra un ejemplo de un árbol de contexto para la cadena $10 \mid 0100110 = 10 \mid b_1b_2b_3b_4b_5b_6b_7$. En él se ve, por ejemplo, que la subcadena b_7 es la única que sigue al contexto 011; o que tras el sufijo $\ldots 10$ sólo aparecen b_1 o b_4.

La Figura 2.82b muestra cómo cada nodo s contiene el par (a_s, b_s), el número de ceros y el número de unos en la cadena asociada con s. La raíz, por ejemplo, está asociada con toda la cadena, por lo que contiene el par $(4,3)$. Todavía tenemos que calcular o estimar una probabilidad ponderada P_w^s para cada nodo s: la probabilidad que se debe enviar al codificador aritmético para codificar la cadena asociada con s. Este cálculo es, de hecho, la parte central del algoritmo CTW. Empezamos por las hojas, porque lo único disponible en una hoja es el par (a_s, b_s); no hay sufijo. Por ello, la mejor hipótesis que se puede hacer es que la subcadena formada por a_s ceros y b_s unos que está asociada a la hoja s no tiene memoria, y el mejor peso que se puede definir para la probabilidad del nodo es la estimación KT determinada por $P_e(a_s, b_s)$. Por lo tanto, se define así:

$$P_w^s \stackrel{\text{def}}{=} P_e(a_s, b_s) \quad \text{si profundidad(s)} = d. \tag{2.12}$$

Utilizando las probabilidades ponderadas para las hojas, recorremos el árbol hacia arriba de forma recursiva y calculamos las probabilidades ponderadas para los nodos internos. Para un nodo interno s, conocemos algo más de información que para una hoja, ya que dicho nodo tiene uno o dos hijos. Los hijos, que denotamos por s_0 y s_1, ya disponen de las probabilidades ponderadas. Consideramos dos casos: si la subcadena asociada con el sufijo s no tiene memoria, entonces $P_e\left(a_s, b_s\right)$ es una buena ponderación de la probabilidad para ella; de lo contrario, el método CTW expone que el producto $P_w^{s_0} P_w^{s_1}$, de las probabilidades ponderadas de los nodos secundarios, es una buena codificación de la probabilidad (un nodo hijo desaparecido se considera, en tal caso, ponderado con la probabilidad 1).

Puesto que no sabemos cuál de los casos anteriores es cierto para un nodo interno dado s, lo mejor que podemos hacer es asignar una ponderación de probabilidades, que es la media de los dos casos anteriores, i.e.,

$$P_w^s \stackrel{\text{def}}{=} \frac{P_e\left(a_s, b_s\right) + P_w^{s_0} P_w^{s_1}}{2} \quad \text{si profundidad}\,(\text{s}) < d. \tag{2.13}$$

El último paso que necesita ser descrito es cómo se actualiza el camino del árbol de contexto cuando se recibe el bit siguiente. Supongamos que la cadena $b_1 b_2 \ldots b_{t-1}$ ya ha sido introducida y codificada. Por lo tanto, ya hemos construido un árbol de contexto de profundidad d para esta cadena; hemos utilizado las ecuaciones (2.12) y (2.13) para calcular las probabilidades ponderadas para el árbol entero, y la raíz del árbol ya contiene una probabilidad ponderada determinada. Ahora introducimos el bit siguiente b_t y lo examinamos. De acuerdo con lo que es, tenemos que actualizar el árbol de contexto para la cadena $b_1 b_2 \ldots b_{t-1} b_t$. La probabilidad ponderada en la raíz de el nuevo árbol será enviada al codificador aritmético, junto con el bit b_t, y se utilizará para codificar b_t.

Si $b_t = 0$, la actualización del árbol se realiza así:

1. Se incrementan los contadores a_s, para todos los nodos s.

2. Se actualizan las probabilidades estimadas, $P_e\left(a_s, b_s\right)$, para todos los nodos.

3. Finalmente se actualizan las probabilidades ponderadas, $P_w\left(a_s, b_s\right)$, para todos los nodos.

Si $b_t = 1$, entonces todos los b_s deben ser incrementados, tras lo cual se realiza la actualización de las probabilidades estimadas y ponderadas, procediendo como antes.

La Figura 2.83a muestra cómo se actualiza el árbol de contexto de la Figura 2.82b cuando $b_t = 0$.

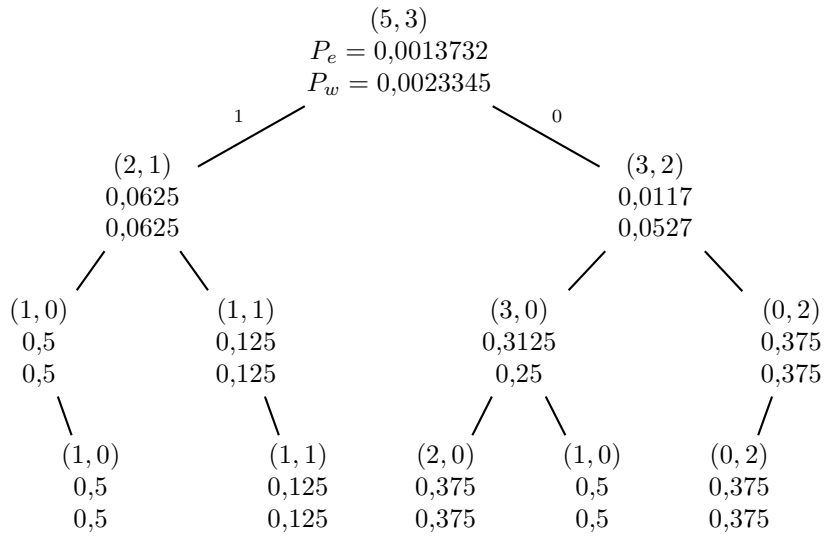
La Figura 2.83b muestra cómo se actualiza el árbol de contexto de la Figura 2.82b cuando $b_t = 1$.

◇ **Ejercicio 2.55 (sol. en pág. 1070):** Constrúyanse los árboles contexto de profundidad 3 para las cadenas 000 | 0, 000 | 00, 000 | 1 y 000 | 11.

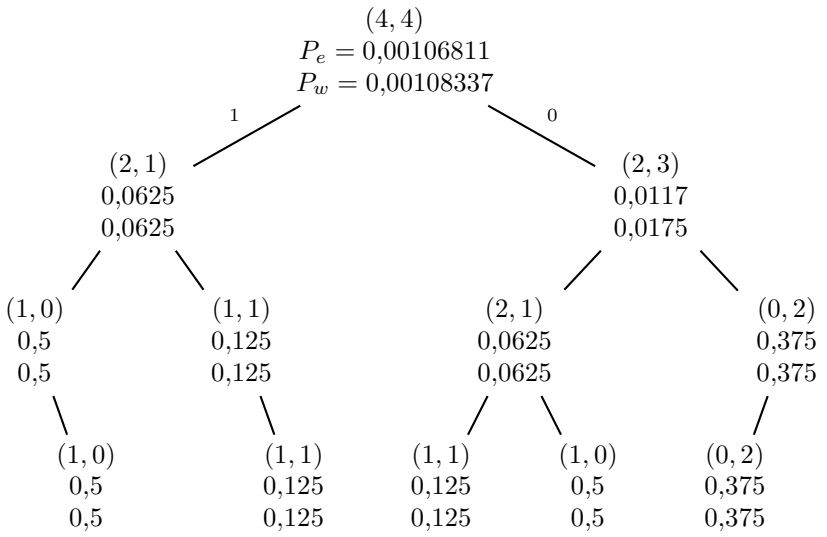
La profundidad d del árbol contexto es seleccionada por el usuario (o se construye internamente tanto en el codificador como en el decodificador) y no cambia mientras se realiza la compresión. El árbol tiene que ser actualizado para cada bit entrante procesado, pero esta actualización requiere a lo sumo $d + 1$ nodos. El número de operaciones necesarias para procesar los n bits de la entrada es, por lo tanto, lineal en n.

Tenía la esperanza de que el contenido de sus bolsillos podría ayudarme a llegar a una conclusión.

—Arthur Conan Doyle, *Memorias de Sherlock Holmes.*



(a)



(b)

Figura 2.83: Árboles de contexto para $b_t = 0,\ 1$.

2.19.1. CTW para la compresión de texto

Hasta ahora, el método CTW ha sido desarrollado para la compresión de cadenas binarias. En la práctica, normalmente estamos interesados en la compresión de texto, imagen, sonido y cadenas; y esta sección analiza un método para la aplicación de CTW a la compresión de texto.

Cada carácter ASCII se compone de siete bits, y se utilizan las 128 combinaciones posibles de los 7 bits. Sin embargo, algunas combinaciones (como `E` y `T`) son más comunes que otras (como `Z`, `<`, y ciertos caracteres de control). Además, ciertos pares de caracteres y tripletes (por ejemplo, `TH` y `THE`) aparecen con más frecuencia que otros. Por eso, pretendemos que si b_t es un bit en un determinado carácter ASCII, `X`, entonces los $t-1$ bits, $b_1 b_2 \dots b_{t-1}$, que le preceden puedan actuar como contexto (incluso si algunos de ellos ni siquiera son parte de `X`, sino que pertenecen a los caracteres anteriores a `X`). La experiencia demuestra que se obtienen buenos resultados:

1. En contextos de tamaño 12.
2. Cuando se utilizan siete árboles de contexto, cada uno para construir un modelo para uno de los siete bits.
3. Y si la estimación KT original es modificada para estimar la *redundancia cero* definida por

$$P_e^z(a,b) \stackrel{\text{def}}{=} \frac{1}{2}P_e(a,b) + \frac{1}{4}\vartheta\,(a=0) + \frac{1}{4}\vartheta(b=0),$$

donde $\vartheta\,(\text{verdadero}) \stackrel{\text{def}}{=} 1$ y $\vartheta\,(\text{falso}) \stackrel{\text{def}}{=} 0$.

Otra característica experimental es un cambio en la definición de las probabilidades ponderadas. La definición original —la Ecuación (2.13)— se utiliza para los dos árboles en las fronteras ASCII (i.e., los bits 1 y 7 de cada código ASCII). Las probabilidades ponderadas para los cinco árboles de contexto asociados a los bits, de 2 a 6, se definen: $P_s^w = P_w^{s_0} P_w^{s_1}$.

Esto produce la compresión típica de 1,8 a 2,3 $^{bits}/_{carácter}$ en los documentos del *Calgary Corpus*.

Paul Volf [Volf 97] ha propuesto otros enfoques para la compresión de texto usando CTW.

> La emoción que un jugador siente al hacer una apuesta es igual a la cantidad que podría ganar por la probabilidad de ganarlo.
>
> —Blaise Pascal.

Capítulo 3

Métodos de diccionario

Los métodos de compresión estadísticos usan un modelo estadístico de los datos, por lo que la calidad de la compresión que alcanzan depende de lo bueno que es el modelo. Los métodos de compresión basados en diccionarios no emplean un modelo estadístico, ni tampoco códigos de tamaño variable. En su lugar, seleccionan cadenas de símbolos y codifican cada cadena como una muestra (*token*) utilizando un diccionario. El diccionario contiene cadenas de símbolos, y puede ser estático o dinámico (adaptativo). En el primer caso, es permanente; a veces permite la adición de cadenas, pero no su eliminación. En el segundo caso se conservan las cadenas que se han introducido anteriormente, lo que permite adiciones y supresiones de cadenas a medida que se leen nuevos datos de entrada.

Dada una cadena de n símbolos, un compresor basado en diccionarios, en principio, comprime hasta nH bits donde H es la entropía de la cadena. Por lo tanto, los compresores basados en diccionarios son codificadores de entropía, pero sólo si el archivo de entrada es muy grande. En la práctica, los compresores basados en diccionarios, producen resultados bastante buenos con la mayoría de los archivos, lo que hace este tipo de codificador muy popular. Además, estos codificadores son de propósito general, pudiéndose utilizar tanto como para imágenes y datos de audio, como para texto.

El ejemplo más simple de "diccionario estático" es un diccionario de inglés utilizado para comprimir texto en inglés. Imagine un diccionario que contiene, digamos, medio millón de palabras (sin sus definiciones). Una palabra (una cadena de símbolos terminada por un espacio o un signo de puntuacion) se lee en la secuencia de entrada y se busca en el diccionario. Si se encuentra una coincidencia, se escribe una entrada de índice para el diccionario en la secuencia de salida. De lo contrario, se escribe la palabra sin comprimir. (Este es un ejemplo de *compresión lógica*.)

Como resultado, la secuencia de salida contiene índices y palabras literales; y es importante distinguirlos. Una forma de lograrlo es reservar un bit extra para cada entidad escrita. En principio, un índice de 19 bits es suficiente para especificar un elemento en un diccionario de $2^{19} = 524{,}288$ palabras. De este modo, cuando se encuentra una coincidencia, podemos escribir un símbolo de 20 bits formado por un bit indicador (tal vez un cero), seguido de una entrada de índice de 19 bits. Cuando no hay coincidencias, se escribe como bit indicador un 1, seguido por el tamaño de la palabra no encontrada y de dicha palabra.

Por ejemplo: Si suponemos que la palabra `bet` se encuentra en la entrada 1025 del diccionario, se codifica como el número de 20 bits: 0 | 00000001000000000001. Si suponemos que la palabra `xet` no se encuentra, se codifica como 1 | 0000011 | 01111000 | 01100101 | 01110100; éste es un número de 4 bytes en el que el campo de 7 bits, 0000011, indica que tras él siguen tres bytes más.

Suponiendo que el tamaño se escribe como un número de 7 bits, y que el promedio de tamaño de una palabra es de cinco caracteres, una palabra sin comprimir ocupa, en promedio, seis bytes ($= 48\,bits$) en la cadena de salida. La compresión de 48 bits en 20 es excelente, siempre y cuando ocurra con la frecuencia suficiente. Por lo tanto, tenemos que responder a la pregunta: ¿cuántas

coincidencias se necesitan para producir una compresión en la totalidad de los datos? Denotamos la probabilidad de que se produzca una coincidencia (cuando la palabra se encuentra en el diccionario), por P. Después de la lectura y la compresión de N palabras, el tamaño de la secuencia de salida será de $N\,[20P + 48\,(1 - P)] = N\,[48 - 28P]$ bits. El tamaño de la cadena de entrada es (suponiendo cinco caracteres por palabra), $40N$ bits. La compresión se logra cuando $N\,[48 - 28P] < 40N$, lo que implica $P > 0{,}29$. Necesitamos un nivel de coincidencia del 29 % ó más, para lograr compresión.

◇ **Ejercicio 3.1 (sol. en pág. 1070):** ¿Qué factor de compresión podemos obtener con P = 0.9?

Siempre y cuando la secuencia de entrada esté formada por texto en inglés, la mayor parte de las palabras se encontrarán en un diccionario con 500 000 entradas. Con otros tipos de datos, sin embargo, no se puede hacer lo mismo. Un archivo que contiene el código fuente de un programa de ordenador puede contener "palabras" como `cout`, `xor` y `malloc`, que no se pueden encontrar en un diccionario de inglés. Un archivo binario normalmente está compuesto por una algarabía de elementos cuando se muestra en ASCII, por lo que se pueden encontrar muy pocas coincidencias, lo que produce una expansión considerable en lugar de compresión.

Esto demuestra que un diccionario estático no es una buena opción para un compresor de propósito general. Puede, sin embargo, ser una buena opción para uno que trate un tipo de información en particular. Considere una cadena de ferreterías inglesas, por ejemplo; sus archivos pueden contener palabras como `nut`, `bolt` y `paint`[1] muchas veces, pero raramente palabras como `peanut`, `lightning` y `painting`[2]. El software de compresión adaptado a una empresa puede beneficiarse de un pequeño y especializado diccionario que contiene, tal vez, sólo unos pocos cientos de palabras. Los ordenadores de cada sucursal, deben tener una copia del diccionario, para facilitar la compresión de los archivos y enviarlos entre las tiendas y las oficinas de la cadena.

En general, es preferible un método basado en diccionarios adaptativos. Tal método puede comenzar con un diccionario vacío o uno pequeño por defecto, añadirle palabras a medida que aparecen en la secuencia de entrada y eliminarle las que han quedado obsoletas, ya que un diccionario extenso retrasa la búsqueda. Dicho método consta de un bucle en el que cada iteración se inicia con la lectura de una cadena de la entrada y la fragmenta (la analiza) en palabras o frases. A continuación, debe buscar en el diccionario cada palabra y, si encuentra una coincidencia, genera un símbolo en la secuencia de la salida. De lo contrario, escribe la palabra sin comprimir y además, la agrega al diccionario. El último paso de cada iteración comprueba si se debe eliminar alguna palabra obsoleta del diccionario. Esto puede sonar complicado, pero tiene dos ventajas:

1. Implica operaciones de búsqueda y emparejamiento de cadenas, en vez de cálculos numéricos. Muchos programadores prefieren eso.

2. El decodificador es sencillo (es un método de compresión asimétrico). En los métodos de compresión estadísticos, el decodificador es normalmente el opuesto exacto del codificador (compresión simétrica). En un método basado en diccionarios adaptativos, sin embargo, el decodificador tiene que leer su secuencia de entrada, determinar si el elemento actual es un símbolo o datos sin comprimir, utilizar los elementos para obtener los datos del diccionario, y ofrecer en la salida los datos finales, sin comprimir. No tiene que analizar la secuencia de entrada de una manera compleja, y no tiene que buscar en el diccionario para encontrar coincidencias. Muchos programadores también lo prefieren así.

Tener un nombre está unido a un descubrimiento científico, técnico, o a un fenómeno que se considera un honor especial para la ciencia. Tener un nombre asociado a un campo entero de la ciencia es incluso mejor. Esto es lo que les sucedió a Jacob Ziv y Abraham Lempel. En la década de 1970 estos dos investigadores desarrollaron los primeros métodos, LZ77 y LZ78, para la compresión de datos

[1] Tuercas, pernos y pintura.
[2] Cacahuete, relámpago y cuadro.

basada en diccionarios. Sus ideas han sido una fuente de inspiración para muchos investigadores, que lo generalizaron, lo mejoraron y lo combinaron con algoritmos RLE y estadísticos para formar los métodos de compresión sin pérdidas más comunes para texto, imágenes y audio. En este capítulo se describen los métodos de compresión LZ más comunes utilizados hoy en día y muestra la forma en que se desarrolló a partir de las ideas básicas de Ziv y Lempel.

> Me encanta el diccionario, Kenny, es el único libro con las palabras en el lugar correcto.
>
> —Paul Reynolds como Colin Mathews en *Press Gang* (1989)

3.1. Compresión de cadenas

En general, los métodos de compresión basados en cadenas de símbolos pueden ser más eficientes que los métodos que comprimen símbolos individuales. Para comprender esto, el lector debe primero revisar el Ejercicio 2.1 (pág. 55). Este ejercicio muestra que, en principio, es posible una mejor compresión si los símbolos del alfabeto tienen probabilidades de ocurrencia muy diferentes. Usemos un sencillo ejemplo para mostrar que las probabilidades de las cadenas de símbolos varían más que las probabilidades de los símbolos individuales que constituyen las cadenas.

Comenzamos con un alfabeto de 2 símbolos —a_1 y a_2—, con probabilidades $P_1 = 0{,}8$ y $P_2 = 0{,}2$, respectivamente. La probabilidad, en promedio, es 0,5, y podemos tener una idea de la varianza (en cuánto se desvían de la media las probabilidades individuales) calculando la suma de las diferencias absolutas: $\mid 0{,}8-0{,}5 \mid + \mid 0{,}2-0{,}5 \mid = 0{,}6$. Cualquier código de tamaño variable debería asignar códigos de 1 bit a los dos símbolos, por lo que el tamaño medio del código es un bit por símbolo.

Ahora generemos todas las cadenas de dos símbolos. Hay cuatro de ellas, se muestran en la Tabla 3.1a, junto con sus probabilidades y un conjunto de códigos de Huffman. La probabilidad media es de 0,25, por consiguiente, calculamos una suma de diferencias absolutas semejante a la realizada anteriormente:

$$\mid 0{,}64 - 0{,}25 \mid + \mid 0{,}16 - 0{,}25 \mid + \mid 0{,}16 - 0{,}25 \mid + \mid 0{,}04 - 0{,}25 \mid = 0{,}78.$$

El tamaño medio del código Huffman es de $1 \times 0{,}64 + 2 \times 0{,}16 + 3 \times 0{,}16 + 3 \times 0{,}04 = 1{,}56$ bits por cadena, lo que representa $1{,}56/2 = 0{,}78$ bits por símbolo.

En el siguiente paso creamos, de manera similar, las ocho posibles cadenas de tres símbolos. Éstas se muestran en la Tabla 3.1b, junto con sus probabilidades y un conjunto de códigos de Huffman. La probabilidad media es de 0,125, por tanto, calculamos una suma de diferencias absolutas semejante a la efectuada más arriba:

$$\mid 0{,}512 - 0{,}125 \mid +3 \mid 0{,}128 - 0{,}125 \mid +3 \mid 0{,}032 - 0{,}125 \mid + \mid 0{,}008 - 0{,}125 \mid = 0{,}792.$$

El tamaño medio del código de Huffman, en este caso, es de $1 \times 0{,}512 + 3 \times 3 \times 0{,}128 + 3 \times 5 \times 0{,}032 + 5 \times 0{,}008 = 2{,}184$ bits por cadena, lo que equivale a $2{,}184/3 = 0{,}728$ bits por símbolo.

A medida que generamos cadenas más largas, las probabilidades de las mismas difieren cada vez más de su promedio, y el tamaño medio del código mejora (Tabla 3.1c). Ésta es la razón por la cual, un método de compresión que comprime cadenas, en lugar de símbolos individuales, puede, en principio, obtener mejores resultados. Esta es también la razón por la que los distintos métodos basados en diccionarios son, en general, mejores y más populares que el método de Huffman y sus variantes (véase también la Sección 4.14). La conclusión anterior es un resultado fundamental de la teoría de la tasa de distorsión, que procede de la teoría de la información que se ocupa de la compresión de datos.

Cadena	Probabilidad	Código
a_1a_1	$0{,}8 \times 0{,}8 = 0{,}64$	0
a_1a_2	$0{,}8 \times 0{,}2 = 0{,}16$	11
a_2a_1	$0{,}2 \times 0{,}8 = 0{,}16$	100
a_2a_2	$0{,}2 \times 0{,}2 = 0{,}04$	101

(a)

Tamaño cadena	Varianza de la prob.	Tam. medio del código
1	0,6	1
2	0,78	0,78
3	0,792	0,728

(c)

Cadena	Probabilidad	Código
$a_1a_1a_1$	$0{,}8 \times 0{,}8 \times 0{,}8 = 0{,}512$	0
$a_1a_1a_2$	$0{,}8 \times 0{,}8 \times 0{,}2 = 0{,}128$	100
$a_1a_2a_1$	$0{,}8 \times 0{,}2 \times 0{,}8 = 0{,}128$	101
$a_1a_2a_2$	$0{,}8 \times 0{,}2 \times 0{,}2 = 0{,}032$	11100
$a_2a_1a_1$	$0{,}2 \times 0{,}8 \times 0{,}8 = 0{,}128$	110
$a_2a_1a_2$	$0{,}2 \times 0{,}8 \times 0{,}2 = 0{,}032$	11101
$a_2a_2a_1$	$0{,}2 \times 0{,}2 \times 0{,}8 = 0{,}032$	11110
$a_2a_2a_2$	$0{,}2 \times 0{,}2 \times 0{,}2 = 0{,}008$	11111

(b)

Tabla 3.1: Probabilidades y códigos de Huffman para un alfabeto de 2 símbolos.

3.2. Una sencilla compresión de diccionario

El propósito de esta sección es un sencillo método, de dos pasos, que me contó Ismail Mohamed (véase el prefacio a la 3 ª edición). El primer paso, lee el archivo de origen y prepara una lista de todos los bytes diferentes que encuentre. El segundo paso, utiliza esta lista para comprimir realmente los bytes de datos. Estos son los pasos en detalle:

1. Se lee el archivo fuente y se prepara una lista de los distintos bytes encontrados. Para cada byte, se incluye también en la lista el número de veces que aparece en dicho archivo (su frecuencia).

2. Se ordena la lista en orden decreciente de las frecuencias. Por lo tanto, comienza con valores de los bytes más comunes en el archivo, y termina con los bytes que aparecen raramente. Puesto que la lista consta de bytes distintos, no puede tener más de 256 elementos.

3. La lista ordenada se convierte en el diccionario; se escribe en el archivo comprimido, precedido por su longitud (un entero de 1 byte).

4. Se lee de nuevo el archivo de origen, byte a byte. se localiza cada byte en el diccionario (por una búsqueda directa) y se observa su índice; el índice es un número dentro del intervalo $[0, 255]$, por lo que requiere entre 1 y 8 bits (apréciese, sin embargo, que la mayoría de los índices son normalmente números pequeños, porque los valores de los bytes más comunes se almacenan al principio del diccionario). En el archivo comprimido, se escribe el índice, precedido por un código de 3 bits, que indica la longitud del índice. En consecuencia, el código 000 denota un índice de 1 bit, el código 001 indica un índice de 2 bits, y así sucesivamente hasta el código 111, que denota un índice de 8 bits.

El compresor mantiene un pequeño *buffer* de 2 bytes, en el que recoge los bits que van a formar parte del archivo comprimido. Cuando la primera celda de la memoria intermedia se ha llenado con un byte, éste se escribe en el archivo, al mismo tiempo que se mueve el segundo byte a la posición del primero.

La descompresión es sencilla. El descompresor comienza leyendo la longitud del diccionario; a continuación, el propio diccionario. Después, decodifica cada byte leyendo su información asociada: el código de 3 bits, seguido por el valor del índice; el índice se utiliza para localizar el siguiente byte de datos en el diccionario.

Se logra compresión porque el diccionario está ordenado por la frecuencia de aparición de los bytes. Cada byte se sustituye por un valor que ocupa entre 4 y 11 bits (un código de 3 bits seguido de 1 a 8 bits). Con 4 bits se obtiene a una razón de compresión de $4/8 = 0{,}5$, mientras que con 11 bits, la razón de compresión llega a $11/8 = 1{,}375$, una expansión. El peor caso es un archivo en el que aparezcan los 256 valores posibles de un byte y tengan una distribución uniforme. La razón de compresión, en este caso, es el promedio[3]:

$$\frac{2 \cdot 4 + 2 \cdot 5 + 4 \cdot 6 + 8 \cdot 7 + 16 \cdot 8 + 32 \cdot 9 + 64 \cdot 10 + 128 \cdot 11}{256 \cdot 8} = 1{,}2509765625,$$

¡lo que indica una expansión! (En realidad, es algo peor que 1,25, ya que el archivo comprimido también incluye el diccionario (256 bytes) y su tamaño (1 byte), por lo que se produce un incremento total de 257 bytes.) La experiencia indica que se obtiene una razón de compresión típica de alrededor de 0,5.

Las probabilidades calculadas aquí se obtienen contando el número de códigos de varios tamaños. Así, hay dos códigos de 4 bits —000 | 0 y 000 | 1—, dos de 5 bits —001 | 10 y 001 | 11—, cuatro de 6 bits —010 | 100, 010 | 101, 010 | 110 y 010 | 111—, ocho de 7 bits —011 | 1000, 011 | 1001, 011 | 1010, 011 | 1011, 011 | 1100, 011 | 1101, 011 | 1110 y 011 | 1111—, y así sucesivamente, hasta 128 códigos de 11 bits.

La desventaja del método es que la compresión es lenta; es una consecuencia de las dos pasadas que realiza el compresor, combinadas con la lenta búsqueda (lenta, porque el diccionario no está ordenado por el valor de cada byte, por lo que no se puede utilizar una búsqueda binaria). La descompresión, en cambio, es más rápida.

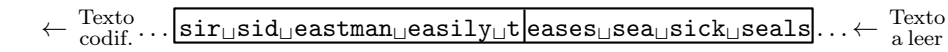
◇ **Ejercicio 3.2 (sol. en pág. 1070):** Diseñe de una estructura razonable para el mantenimiento de la lista generada por este método.

> De éstos, el monte Genèvre y el Brenner fueron los más frecuentados,
> mientras que se apreciará que en los Alpes Centrales sólo dos pasos
> (el Splugen y el Septimer) fueron ciertamente conocidos por los romanos.
>
> (1911 Entrada de la Enciclopedia para “Pasos Principales”)

[3]Para representar los números: de 0 a 1, de 2 a 3, de 4 a 7, de 8 a 15, de 16 a 31, de 32 a 63, de 64 a 127 y de 128 a 255, se necesita un mínimo de 1, 2, 3, 4, 5, 6, 7 y 8 bits, respectivamente; sumando los 3 bits del código que indica la longitud del índice, tenemos: 2 números de 4 bits, otros 2 de 4 bits, 4 de 6 bits,... y 128 de 8 bits.

3.3. LZ77 (Ventana deslizante)

El principio de este método (que a veces es conocido como LZ1) [Ziv y Lempel 78] es utilizar parte de la secuencia de entrada previa a la actual como diccionario. El codificador mantiene, para la cadena de entrada, una memoria intermedia (o buffer) llamada "ventana" y desplaza los datos introducidos en ésta, de derecha a izquierda, a medida que se van codificando los símbolos. En consecuencia, el método se basa en un *deslizamiento de ventana*. La siguiente ventana se divide en dos partes. La parte de la izquierda es el *buffer de búsqueda*. Éste, es el diccionario actual, e incluye los símbolos que se han introducido y codificado recientemente. La parte de la derecha, es el *buffer de lectura anticipada o buffer de preanálisis*[4], que contiene el texto que todavía no se ha codificado. En implementaciones prácticas del búfer de búsqueda tiene un tamaño de unos miles de bytes, mientras que el buffer de lectura anticipada consta sólo de unas decenas de bytes. La barra vertical entre `t` y `e`, en el ejemplo que viene a continuación, representa la línea divisoria actual entre los dos buffers. Suponemos que el texto `sir␣sid␣eastman␣easily␣t` ya se ha comprimido, mientras que el texto `eases␣sea␣sick␣seals` todavía necesita ser comprimido:



El codificador analiza el buffer de búsqueda hacia atrás (de derecha a izquierda) para localizar alguna coincidencia con el símbolo `e`, del buffer de preanálisis. Encuentra uno en la `e` de la palabra `easily`. Esta `e` está a una distancia (desplazamiento) de 8 elementos desde el final del buffer de búsqueda. A continuación, el codificador busca coincidencias con tantos símbolos, respectivamente adyacentes y a la derecha de las dos `e`s como sea posible. En este caso, encuentra tres símbolos —`eas`— , por lo que la longitud de la cadena de caracteres emparejados es 3. El codificador continúa la exploración hacia atrás, tratando de encontrar más coincidencias. En nuestro caso, hay una más; en la palabra `eastman`, con un desplazamiento de 16 y con la misma longitud. El codificador selecciona la cadena de coincidencias más larga o, si son todas tienen la misma longitud, la última de ellas, y prepara el *token* (16, 3, `e`).

La selección del último patrón de coincidencias, en lugar del primero, simplifica la tarea del codificador, debido a que sólo tiene que hacer un seguimiento de la última cadena encontrada. Es interesante apreciar que la selección del primer patrón, aunque hace que el programa sea algo más complejo, también tiene una ventaja. Selecciona el desplazamiento más pequeño. Aparentemente esto no es una ventaja, porque un token debería tener espacio suficiente para almacenar el mayor desplazamiento posible. Sin embargo, es posible combinar secuencialmente LZ77 con Huffman, o alguna otra codificación estadística de tokens, donde a los desplazamientos pequeños se les asignan los códigos más cortos. Este método se llama LZH —el nombre procede de la combinación de LZ y codificación Huffman—. Fue desarrollado en 1988 por Haruyasu Yoshizaki[5], quien creó el paquete lha (véase "http://packages.debian.org/lenny/lha" para más información), que Bernd Herd utilizó durante bastante tiempo para desarrollar programas de instalación para aplicaciones Windows y un CD con bases de datos de guías telefónicas. La existencia de muchos desplazamientos pequeños implica una mejor compresión en LZH.

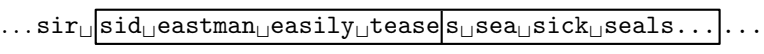
◇ **Ejercicio 3.3 (sol. en pág. 1070):** ¿Cómo hace el decodificador para saber si el codificador selecciona el primer patrón de coincidencia o el segundo?

En general, un token LZ77 consta tres partes: desplazamiento, longitud y el símbolo siguiente del buffer de preanálisis (que, en nuestro caso, es el segundo `e` de la palabra `teases`). Este token, se escribe en la secuencia de salida, y la ventana se desplaza hacia la derecha (o, alternativamente, la cadena de

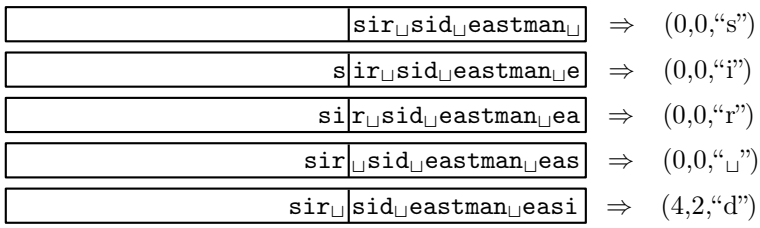
[4] The look-ahead buffer.

[5] Y lo utilizó en el compresor de datos LHARC, que posteriormente cambió el nombre a LHA.

entrada se mueve a la izquierda), cuatro posiciones: tres para el patrón de coincidencia y una para el símbolo siguiente:



Si la búsqueda hacia atrás no ofrece ninguna coincidencia, se escribe un token LZ77 con desplazamiento y longitud cero, y con el símbolo no encontrado. Esta es también la razón por la que un token tiene un tercer componente. Los tokens con desplazamiento y longitud cero, son comunes al inicio de cualquier trabajo de compresión, cuando el buffer de búsqueda está vacío o casi vacío. Los cinco primeros pasos en la codificación de nuestro ejemplo son los siguientes:

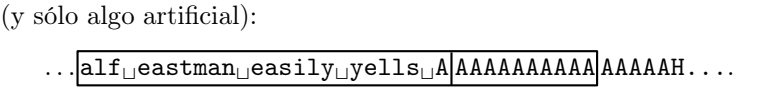


◇ **Ejercicio 3.4 (sol. en pág. 1070):** ¿Cuáles son los dos pasos siguientes?

Claramente, un token de la forma $(0, 0, \ldots)$, que codifica un solo símbolo, no proporciona una buena compresión. Es fácil calcular su longitud. El tamaño del desplazamiento es $[\log_2 S]$, donde S es la longitud del búfer de búsqueda[6]. En la práctica, el buffer de búsqueda puede tener unos pocos miles de bytes de longitud, por lo que el tamaño del desplazamiento es normalmente de 10 a 12 bits. El tamaño del campo "longitud" es similar —$[\log_2 (L-1)]$—, donde L es la longitud del buffer de preanálisis (véase más abajo para explicar el -1). En la práctica, el buffer de lectura anticipada consta sólo de unas pocas decenas de bytes, por lo que el tamaño del campo "longitud" es de sólo unos pocos bits. El tamaño del campo "símbolo" es típicamente de 8 bits, pero en general, es $[\log_2 A]$, donde A es el número de elementos del alfabeto. El tamaño total del token para un símbolo —$(0, 0, \ldots)$—, es típicamente de $11 + 5 + 8 = 24$ bits, mucho mayor que los 8 bits que se necesitan para codificar únicamente el símbolo.

Este es un ejemplo que muestra por qué el campo "longitud" puede mayor que el tamaño del buffer de preanálisis:

Se busca la coincidencia más larga entre la cadena que comienza por el primer símbolo `a` del buffer de preanálisis y cada una de las cinco que comienza por `a`, en el búfer de búsqueda. Resulta que las que empiezan por las dos `a`es de los extremos tienen la misma longitud —3— y el codificador debe seleccionar la última (la situada más a la izquierda) de ellas y crear el token (28,3,"`a`"). De hecho, crea el token (3,4,"␣"). La cadena de cuatro símbolos `alfa` del buffer de preanálisis se empareja con los últimos tres símbolos `alf` del buffer de búsqueda y el primer símbolo `a` del buffer de preanálisis. La razón de esto es que el decodificador puede usarlo como un token con naturalidad, sin ninguna modificación. Comienza en la posición 3 de su buffer de búsqueda y copia los cuatro símbolos siguientes, uno a uno, ampliando su buffer hacia la derecha. Los tres primeros símbolos son copias de los antiguos contenidos del búfer, y el cuarto es una copia del primero de esos tres. El siguiente ejemplo es aún más convincente (y sólo algo artificial):



[6]Los corchetes indican "parte entera de".

El codificador crea el token (1,9,A), relacionando las nueve primeras copias de A en el buffer de preanálisis e incluyendo la décima A. Por eso, en principio, la longitud del patrón de búsqueda puede llegar hasta el tamaño del buffer de preanálisis menos 1.

El decodificador es mucho más sencillo que el codificador (LZ77 es, por lo tanto, un método de compresión asimétrico). Tiene que mantener un buffer, lo que equivale en tamaño a la ventana del codificador. El decodificador introduce un token, encuentra el patrón de búsqueda en su buffer, escribe la cadena localizada y el tercer campo del token en la secuencia de salida, y dentro del buffer, desplaza dicha cadena y el tercer campo. Esto implica que LZ77, o cualquiera de sus variantes, es útil en aquellos casos en que un archivo se comprime una vez (o apenas unas pocas veces) y se descomprime a menudo. Un archivo que raramente se utiliza, compuesto por archivos comprimidos es un buen ejemplo.

Al principio parece que este método no hace ninguna suposición sobre los datos de entrada. En concreto, no presta atención a la frecuencia de los símbolos. Pensando un poco, sin embargo, se ve que, debido a la naturaleza de la ventana deslizante, el método LZ77 siempre se compara el buffer de preanálisis con el texto introducido recientemente en el buffer de búsqueda y nunca con el texto que se ha introducido hace mucho tiempo (y, por consiguiente, ha sido extraído del buffer de búsqueda). Así, el método supone implícitamente que los patrones se encuentran juntos en los datos de entrada. Los datos que cumplen este supuesto se comprimirán bien.

El método LZ77 básico se ha perfeccionado de varias maneras por los investigadores y programadores durante las décadas de 1980 y de 1990. Una forma de mejorarlo es el uso de campos "desplazamiento" y "longitud" de tamaño variable en los tokens. Otra forma, es aumentar el tamaño de los dos búferes. Un buffer de búsqueda de mayor tamaño permite encontrar mejores patrones de coincidencia, pero el trasiego de datos incrementa el tiempo de búsqueda. Un buffer de búsqueda grande, por lo tanto, requiere una estructura de datos más sofisticada que permita una búsqueda rápida (Sección 3.12.2). Una tercera mejora tiene que ver con el desplazamiento de la ventana. El enfoque más simple es mover todo el texto de la ventana hacia la izquierda, después de cada etapa de emparejamiento. Un método más rápido consiste en sustituir la ventana lineal con una *cola circular*, en la que el desplazamiento de la ventana se realiza inicializando dos punteros (Sección 3.3.1). Otra mejora, es la adición de un bit más (un flag) a cada token, eliminando así el tercer campo (Sección 3.4). Un apunte especial es la tabla hash (tabla de dispersión) empleada por el algoritmo de deflación (*Deflate*) (Sección 3.23.3) para buscar coincidencias.

3.3.1. Una cola circular

La cola circular, es una estructura de datos básica. Físicamente, es un array[7] lineal, pero se utiliza como un array circular. La Figura 3.2, ilustra un sencillo ejemplo. Muestra un array de 16 bytes, en el que se añaden caracteres por el "final" y se borran por el "principio". Ambos extremos —principio y fin—, van cambiando de posición; dos punteros —p y f—, apuntan a ellos continuamente. En (a), la cola se compone de los ocho caracteres sid␣east—con el resto del buffer vacío—. En (b), se han ocupado los 16 bytes, y f apunta al final del buffer. En (c), la primera letra —s— ha sido eliminada y se ha insertado la l del futuro easily. Nótese cómo el puntero f está ahora situado *a la izquierda* de p. En (d), se han eliminado las dos letras id con sólo mover el puntero p; los caracteres siguen presentes en el array, pero han sido eliminados eficazmente. En (e), se han añadido los dos caracteres $y_{\sqcup}$ y se ha movido el puntero f. En (f), los punteros muestran que el buffer termina en teas y comienza en tman. La inserción de nuevos caracteres en la cola circular y el movimiento de los punteros, es por tanto equivalente a los desplazamientos de los datos contenidos en la cola. Sin embargo, no es necesario hacer ningún desplazamiento o movimiento real.

Se puede encontrar más información sobre las colas circulares en la mayor parte de los textos sobre estructuras de datos.

[7] Secuencia de elementos adyacentes que forman una unidad.

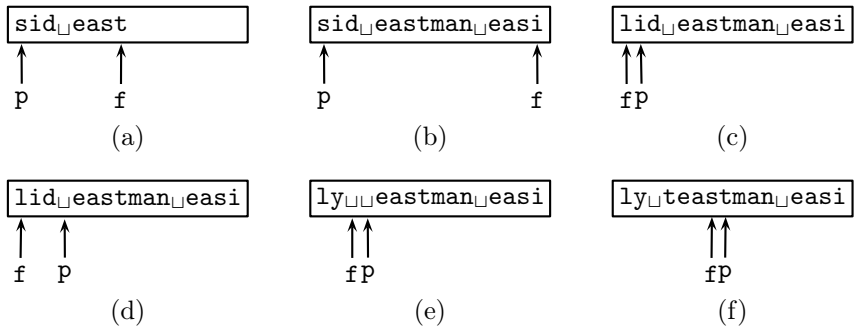


Figura 3.2: Una cola circular.

Consulta del diccionario

Circular. (1) De forma similar o casi como un círculo. (2) Definición de una palabra en términos de otra que se define a sí misma en términos de la primera palabra. (3) Dirigido o distribuido a un gran número de personas.

3.4. LZSS

El método LZSS es una variante eficiente de LZ77, desarrollado por Storer y Szymanski en 1982 [Storer y Szymanski 82]. Mejora LZ77 en tres direcciones: (1) el buffer de preanálisis tiene la estructura de una cola circular, (2) el buffer de búsqueda —el diccionario—, es un árbol binario de búsqueda, y (3) los tokens se crean con dos campos, en lugar de tres.

Un árbol binario de búsqueda, es un árbol binario en el que el subárbol izquierdo de cada nodo A, contiene nodos más pequeños que A, y el subárbol derecho, contiene nodos mayores que A. Puesto que los nodos de nuestros árboles binarios de búsqueda contienen cadenas, primero necesitamos saber cómo comparar dos cadenas y decidir cuál es "más grande". Esto se comprende fácilmente imaginando que las cadenas aparecen en un diccionario o un lexicón, en el que se ordenan alfabéticamente. Es evidente que la cadena `rote` precede a la cadena `said`, ya que la `r` precede a la `s` (a pesar de que la `o` sigue a la `a`), por lo que consideramos `rote` más pequeña que `said`. Este concepto se denomina *orden lexicográfico* (la ordenación de cadenas lexicográficamente).

¿Qué sucede con la cadena `␣abc`? La mayoría de las computadoras modernas utilizan códigos ASCII para representar los caracteres (aunque cada vez más se usa el *Unicode*, que se trata en la Sección 8.12; y algunas computadoras más antiguas de IBM, Amdahl, Fujitsu, y las supercomputadoras de Siemens, usan el viejo código EBCDIC —de 8 bits— desarrollado por IBM), y el código de un espacio en blanco en ASCII, precede a los de las letras, por lo que una cadena que comienza con un espacio, será más pequeño que cualquier cadena que comience con una letra. En general, el *orden de clasificación de caracteres*[8] de la computadora, determina la secuencia de caracteres patrón para la ordenación lexicográfica —los elementos situados más a la izquierda, son menores que los situados más a la derecha—. La Figura 3.3 muestra dos ejemplos de árboles binarios de búsqueda.

Nótese la diferencia entre el árbol (casi) balanceado de la Figura 3.3a, y el asimétrico de la Figura 3.3b. Contienen los mismos 14 nodos, pero su aspecto y su comportamiento es muy diferente. En el árbol balanceado, cualquier nodo se puede encontrar en no más de cuatro pasos. En el asimétrico, se pueden necesitar hasta 14 pasos. En cualquier caso, el número máximo de pasos necesarios para

[8] *Collating sequence* o secuencia de cotejo.

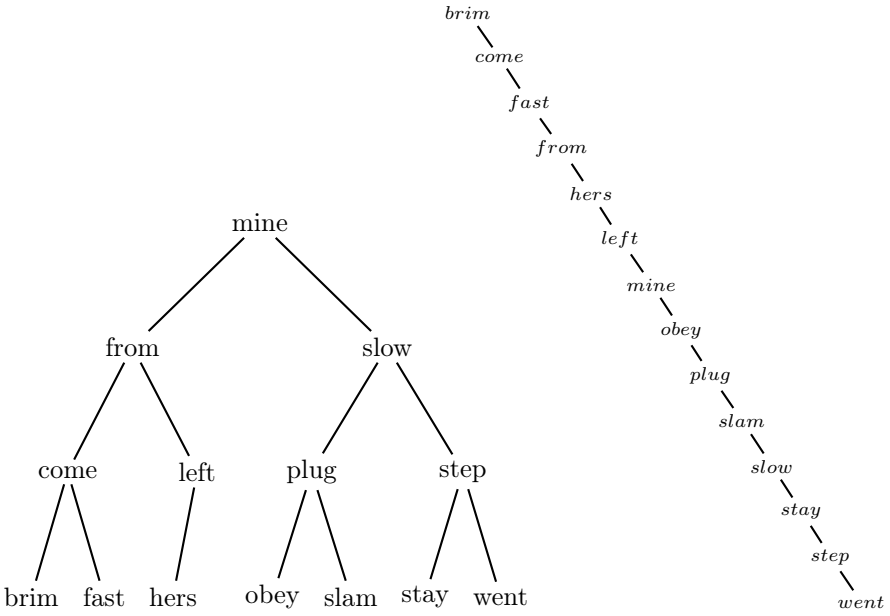


Figura 3.3: Dos árboles binarios de búsqueda.

localizar un nodo es igual a la altura del árbol. Para un árbol asimétrico (que es en realidad lo mismo que una lista enlazada), la altura es el número de elementos —n—; para un árbol balanceado, la altura es $[\log_2 n]$ —un número mucho menor—. Se puede encontrar más información sobre las propiedades de los árboles binarios de búsqueda en cualquier texto sobre estructuras de datos.

En el siguiente ejemplo, se muestra cómo se puede utilizar un árbol binario de búsqueda para acelerar la búsqueda en el diccionario. Suponemos que se ha introducido una cadena con la corta frase:

`sid␣eastman␣clumsily␣teases␣sea␣sick␣seals`.

Para no complicar el ejemplo, asumimos una ventana de 16 bytes para buffer de búsqueda, seguida de una de 5 bytes para el buffer de preanálisis. Después de la introducción de los primeros $16 + 5$ caracteres, la ventana deslizante es:

`sid␣eastman␣clum|sily␣`teases␣sea␣sick␣seals

con la cadena `teases␣sea␣sick␣seals` esperando aún en la entrada.

El codificador analiza el búfer de búsqueda, creando las $16-5+1 = 12$ cadenas de cinco caracteres de la Tabla 3.4, que se insertan en el árbol binario de búsqueda —cada uno con su desplazamiento—.

El primer símbolo en el buffer de preanálisis es `s`, por lo que el codificador busca en el árbol las cadenas que comienzan con una `s`. Se localizan dos —en los desplazamientos 16 y 10—, y el primero de ellos —`sid␣e` (en el desplazamiento 16)— proporciona una coincidencia mayor.

(Ahora tenemos que hacer inciso para discutir el caso en el que una cadena del árbol coincide por completo con el contenido del buffer de preanálisis. En este caso, el codificador debe volver al buffer

`sid␣e`	16
`id␣ea`	15
`d␣eas`	14
`␣east`	13
`eastm`	12
`astma`	11
`stman`	10
`tman␣`	09
`man␣c`	08
`an␣cl`	07
`n␣clu`	06
`␣clum`	05

Tabla 3.4: Cadenas de cinco caracteres.

de búsqueda, para tratar de encontrar coincidencias con cadenas más largas. En principio, la longitud máxima de la cadena coincidente puede llegar a ser $L - 1$.)

En nuestro ejemplo, el número de elementos adyacentes con coincidencia es 2; se emite el token de 2 campos $(16, 2)$. El codificador ahora tiene que deslizar la ventana dos posiciones a la derecha, y actualizar el árbol. La nueva ventana es:

`si`|`d␣eastman␣clumsi`|`ly␣te`|`ases␣sea␣sick␣seals`

El árbol debe ser actualizado suprimiendo las cadenas `sid␣e` e `id␣ea`, e insertando las nuevas cadenas `clums` y `lumsi`. Si se consigue emparejar una cadena de k letras, la ventana tiene que desplazarse k posiciones, y el árbol debe actualizarse eliminando k cadenas y añadiendo k nuevas cadenas; pero ¿cuáles?

Pensando un poco, vemos que las k cadenas que se van a suprimir son las primeras en el buffer de búsqueda —antes del desplazamiento—; y las k cadenas que se van agregar son las últimas en dicho buffer —después del desplazamiento—. Un procedimiento sencillo para actualizar el árbol, es preparar una cadena formada por las cinco primeras letras del búfer de búsqueda, encontrarla en el árbol, y eliminarla. A continuación, deslizar el buffer una posición a la derecha —o desplazar los datos a la izquierda—, preparar una cadena con las últimas cinco letras del búfer de búsqueda, y añadirla al árbol. Esto debe repetirse k veces.

Puesto que cada actualización elimina y añade el mismo número de cadenas, el tamaño del árbol nunca cambia. Siempre contiene T nodos, donde T es la longitud del buffer de búsqueda menos la longitud del buffer de preanálisis más 1 ($T = S - L + 1$). La forma del árbol, sin embargo, puede cambiar significativamente. Como los nodos se agregan y eliminan, el árbol puede cambiar su forma entre un árbol completamente asimétrico —el peor de los casos para la búsqueda— y uno balanceado —la forma ideal para la búsqueda—.

La tercera mejora de LZSS sobre LZ77 está en los tokens creados por el codificador. Un token LZSS contiene sólo un desplazamiento y una longitud. Si no se encuentra ninguna coincidencia, el codificador emite el código sin comprimir del símbolo siguiente, en vez del nada económico token de tres campos $(0, 0, \ldots)$. Para distinguir entre los tokens y los códigos sin comprimir, cada uno va precedido por un solo bit —un flag o bandera—.

En la práctica, el búfer de búsqueda puede estar formado por unos pocos miles de bytes de longitud, por lo que el campo del desplazamiento consta normalmente de 11 a 13 bits. El tamaño del buffer de preanálisis se debe seleccionar de tal manera que el tamaño total de un token sea de 16 bits (2 bytes). Por ejemplo, si el tamaño del buffer de búsqueda es de 2 Kbytes ($= 2^{11}$), entonces, el buffer de preanálisis deberá ser de 32 bytes de longitud ($= 2^5$). El campo del desplazamiento será de 11 bits y el

campo de la longitud, 5 bits —el tamaño del buffer de preanálisis—. Con esta elección de tamaños de buffer, el codificador emitirá tokens de 2 bytes, o códigos ASCII sin comprimir de 1 byte . Pero ¿qué pasa con los bits de flag? Una buena idea en la práctica, es recolectar ocho ítems de salida —tokens y códigos ASCII— en un pequeño buffer, y luego sacar un byte formado por los ocho flags, seguido por los ocho ítems —que son de 1 ó 2 bytes de longitud cada uno—.

3.4.1. LZARI

El siguiente párrafo es una cita procedente de [Okumura 98]:

Durante el verano de 1988, Yo [Haruhiko Okumura] escribí otro programa de compresión de datos, LZARI. Este programa se basa en la siguiente observación: Cada salida de LZSS es, o bien un único carácter o bien un par $< posición, longitud >$. Un solo carácter se puede codificar como un número entero entre 0 y 255. En cuanto al campo $< longitud >$, si el rango de la longitud es, digamos, de 2 a 257, puede codificarse como un entero entre 256 y 511. Por lo tanto, puedo decir que hay 512 tipos de "caracteres", y que los "caracteres" de 256 a 511 van acompañados por un campo $< posición >$. Estos 512 "caracteres" pueden codificarse con el método de Huffman, o aún mejor, algebraicamente. El campo $< posición >$ puede codificarse de la misma manera. En LZARI, he utilizado una compresión algebraica adaptativa para codificar los "caracteres", y una compresión algebraica estática para codificar el campo $< posición >$. (Hubo varias versiones de LZARI; algunas de ellas eran ligeramente diferentes de la descripción anterior). La compresión obtenida por LZARI era muy ajustada, aunque bastante lenta.

3.4.2. Deficiencias

Antes de discutir LZ78, vamos a resumir las deficiencias de LZ77 y sus variantes. Ya se ha mencionado que LZ77 utiliza la suposición implícita integrada de que los patrones aparecen juntos en la entrada de datos. Los flujos de datos que no cumplen esta suposición ofrecen una compresión pobre. Un ejemplo común es un texto en el que una determinada palabra —digamos `economy`— se produce a menudo, pero se distribuye uniformemente en todo el texto. Cuando esta palabra se desplaza en el buffer de preanálisis, su ocurrencia anterior puede haber sido desplazada fuera del buffer de búsqueda. Un algoritmo mejor podría guardar las cadenas que se producen con más frecuencia en el diccionario y no simplemente deslizarlas continuamente.

Otra desventaja de LZ77 es el limitado tamaño —L— del buffer de preanálisis. El tamaño de las cadenas coincidentes se limita a $L - 1$; pero L debe ser pequeña, porque el proceso de búsqueda de cadenas implica la comparación de símbolos individuales. Si se duplicara L en tamaño, la compresión mejoraría, puesto que haría posible las búsquedas largas, pero el codificador sería mucho más lento cuando intentara encontrar una coincidencia larga. El tamaño —S— del buffer de búsqueda también es limitado. Un buffer de búsqueda grande se traduce en una mejor compresión, pero ralentiza al codificador, debido a que la búsqueda conlleva más tiempo (incluso con un árbol binario de búsqueda). El aumento de tamaño de ambos búferes, también significa la creación de más tokens, reduciendo así la eficiencia de la compresión. Con tokens de 2 bytes, la compresión de una cadena de dos caracteres en un token produce como resultado: 2 bytes más 1 de flag. La escritura de los dos caracteres como códigos ASCII en bruto, ocupa 2 bytes más 2 flags —una diferencia muy pequeña en tamaño—. El codificador debe —en tal caso— utilizar la última opción y escribir los dos caracteres sin comprimir, ahorrando tiempo y perdiendo sólo un bit. Decimos que el codificador tiene un punto de equilibrio —o *breakeven* point— de 2 bytes. Con largos tokens, el punto de equilibrio se incrementa a tres bytes.

3.5. Tiempos de repetición

Frans Willems, uno de los desarrolladores de la ponderación del árbol de contexto (Sección 2.19), es también el desarrollador de este original —aunque no muy eficiente— método basado en diccionarios. La entrada puede constar de cualquier símbolo, pero el método que aquí se describe —y también en [Willems 89]—, está enfocado para datos binarios. Los símbolos de la entrada se agrupan en palabras de longitud L, que se introducen en un buffer de deslizamiento. El buffer se divide en un buffer de preanálisis —con palabras que aún no se han comprimido—, y un buffer de búsqueda —que contiene las B palabras procesadas más recientemente. El codificador trata de emparejar las palabras del buffer de preanálisis —empezando por las situadas a izquierda—, con el contenido del buffer de búsqueda. Sólo se empareja una palabra del buffer de preanálisis en cada paso. Si se encuentra una coincidencia, la distancia (desplazamiento) entre el principio de la misma —en el buffer de búsqueda— hasta la palabra —en el buffer de preanálisis—, se denota por m, y se codifica con un código prefijo —que consta de 2 partes— que se escribe en la secuencia de datos comprimidos. Nótese que no es necesario codificar el número de símbolos emparejados, porque se corresponde exactamente con una palabra. Si no hay coincidencias, se escribe un código especial, seguido por los L símbolos de la palabra no localizada en bruto.

Ilustremos el método con un sencillo ejemplo. Supongamos que los símbolos de la entrada son bits. Seleccionamos $L = 3$ para la longitud de las palabras, y un buffer de búsqueda de longitud $B = 2^L - 1 = 7$, que contiene los siete bits procesados más recientemente. El buffer de preanálisis contiene sólo los datos binarios, y las comas aquí mostradas se utilizan solamente para indicar los límites de las palabras.

← Datos codificados... 0100100|100,000,011,111,011,101,001... ← Datos a leer

Es obvio que la palabra situada más a la izquierda en el buffer de preanálisis —100—, coincide con los tres bits más a la derecha del buffer de búsqueda. El tiempo de repetición (el desplazamiento) para esta palabra es, por lo tanto, $m = 3$. (El mayor tiempo de repetición es la longitud B del buffer de búsqueda —7 en nuestro ejemplo—). El buffer se desplaza ahora una palabra —tres bits— a la izquierda para convertirse en:

← ...010 0100100|000,011,111,011,101,001,... ... ← Datos a leer

El tiempo de repetición de la palabra actual —000— es $m = 1$, ya que cada bit de esta palabra se corresponde con el bit inmediato de su izquierda. Nótese que es posible emparejar el 0 de más a la izquierda de la siguiente palabra —011— con el bit de su izquierda, pero este método permite el emparejamiento de exactamente una palabra en cada paso. El buffer se desplaza nuevamente L posiciones para convertirse en:

← ...010010 0100000|011,111,011,101,001,..... ... ← Datos a leer

No hay coincidencia para la siguiente palabra —011— en el buffer de búsqueda, por lo que m se establece en un valor especial que denotaremos 8^* (significa: es mayor o igual que 8). Es fácil verificar que los tiempos de repetición de las tres últimas palabras —011, 101 y 001— son, respectivamente: 6, 4 y 8^*.

En primer lugar, se codifica cada tiempo de repetición, determinando dos números enteros p y q. Si $m = 8^*$, entonces p se establece en L; de lo contrario, se selecciona p como el número entero que satisface $2^p < m < 2^{p+1}$. Nótese que p se encuentra dentro del intervalo $[0, L-1]$. El entero q se determina calculando $q = m - 2p$, lo que lo sitúa en el intervalo $[0, 2^p - 1]$. La Tabla 3.5 muestra los valores de m, p, q, y los códigos prefijo utilizados para $L = 3$.

Una vez que se conocen p y q, se construye un código prefijo para m y se escribe en la secuencia de datos comprimidos. El código se compone de dos partes —un prefijo y un sufijo— que son los valores

m	p	q	Prefijo	Sufijo	Longitud
1	0	0	00	ninguno	2
2	1	0	01	0	3
3	1	1	01	1	3
4	2	0	10	00	4
5	2	1	10	01	4
6	2	2	10	10	4
7	2	3	10	11	4
8*	3	—	11	palabra	5

Tabla 3.5: Codificación de los tiempos de repetición para $L = 3$.

binarios de p y q, respectivamente. Dado que p está dentro del intervalo $[0, L-1]$, el prefijo requiere $\log(L+1)$ bits. La longitud del sufijo es de p bits. El caso $p = L$ es diferente. Aquí, el sufijo es el valor en bruto —L bits— de la palabra que se está comprimiendo.

La secuencia de datos comprimidos para las siete palabras de nuestro ejemplo, se compone de los siete códigos:

$$01|1,\ \ 00,\ \ 11|011,\ \ 00,\ \ 10|10,\ \ 10|00,\ \ 11|001,\ldots,$$

adonde las barras verticales separan el prefijo y el sufijo de un código. Nótese que las palabras tercera y séptima —011 y 001— están incluidas en los códigos en bruto —sin comprimir—.

Es fácil ver por qué este método genera códigos prefijo. Una vez que se ha asignado un código (como $01 \mid 0$, el código de $m = 2$), ese código no puede ser prefijo de cualquier otro código porque: (1) algunos de los otros códigos son para diferentes valores de p, y por lo tanto, no comienzan con 01; y (2) los códigos para el mismo p comienzan con 01, pero se deben tener diferentes valores de q, por lo que tienen sufijos diferentes.

El rendimiento de la compresión producida por este método es inferior al de LZ77, pero es interesante por las siguientes razones:

1. Es universal y óptimo. No utiliza las estadísticas de los datos de la secuencia de entrada, y su rendimiento se aproxima asintóticamente a la entropía de dichos datos a medida que éstos crecen más y más.

2. Se demuestra en [Cachin 98], que este método puede ser modificado para incluir datos ocultos (esteganografía).

3.6. QIC-122

QIC es una asociación comercial internacional, creada en 1987, cuya misión es estimular y fomentar el uso generalizado de la tecnología del cartucho de cinta de un cuarto de pulgada[9] (de ahí el acrónimo QIC; véase también http://www.qic.org/).

El estándar de compresión QIC-122 es una variante de LZ77 que ha sido desarrollado por QIC para la compresión de texto de los datos guardados en unidades de cartucho de cinta de $1/4$ de pulgada. Los datos se leen y se desplazan en un buffer de entrada de 2048 bytes ($= 2^{11}$) —de derecha a izquierda—, de tal manera que el primer carácter es el del extremo izquierdo. Cuando el buffer está lleno —o cuando se han leído todos sus datos—, el algoritmo busca de izquierda a derecha repeticiones de cadenas. La salida consiste en caracteres en bruto y tokens que representan a las cadenas localizadas en el búfer. Como ejemplo, supongamos que se han leído los siguientes datos y se han introducido por desplazamiento en el buffer:

[9] En inglés, *quarter-inch*.

Bytes	Longitud			Bytes	Longitud				
2	00			17	11	11	1001		
3	01			18	11	11	1010		
4	10			19	11	11	1011		
5	11	00		20	11	11	1100		
6	11	01		21	11	11	1101		
7	11	10		22	11	11	1110		
8	11	11	0000	23	11	11	1111	0000	
9	11	11	0001	24	11	11	1111	0001	
10	11	11	0010	25	11	11	1111	0010	
11	11	11	0011	⋮					
12	11	11	0100	37	11	11	1111	1110	
13	11	11	0101	38	11	11	1111	1111	0000
14	11	11	0110	39	11	11	1111	1111	0001
15	11	11	0111		etc.				
16	11	11	1000						

Tabla 3.6: Valores del campo < *longitud* >.

```
ABAAAAAACABABABA..............
```

El primer carácter —`A`— obviamente no es una repetición de cualquier cadena previa, por lo que se codifica —en bruto (formato *raw*)— como un carácter ASCII (véase más abajo). El siguiente carácter —`B`— se codifica también tal cual es. El tercer carácter —`A`— es idéntico al primero, pero también se codifica en bruto, ya que las secuencias repetidas deben ser de —por lo menos— dos caracteres de longitud. Sólo con el cuarto carácter —`A`— tenemos una repetición de cadena. La cadena de cinco`A`*es* —de la posición 4 a la 8— es idéntica a aquella de la posición 3 a la 7. Por lo tanto, se codifica como una cadena de longitud 5 y desplazamiento 1. El desplazamiento —u *offset*— en este método, es la distancia entre el comienzo de la cadena repetida y el principio de la original.

El siguiente carácter —`C`, en la posición 9— se codifica en bruto. La cadena `ABA` —en las posiciones 10 a 12—, es una repetición de la cadena situada en las posiciones 1 a 3, por lo que se codifica como una cadena de longitud 3 y desplazamiento $10 - 1 = 9$. Por último, la cadena`BABA` —en las posiciones 13 a 16—, se codifica como longitud 4 y desplazamiento 2, ya que es una repetición de la cadena localizada en las posiciones 10 a 13.

◇ **Ejercicio 3.5 (sol. en pág. 1072):** Supóngase que los siguientes cuatro caracteres de datos son `CAAC`:

```
ABAAAAAACABABABACAAC............
```

¿Cómo serán codificados?

Un carácter en bruto se codifica como un 0, seguido de los 8 bits del carácter en ASCII. Una cadena se codifica como un token que comienza con 1, seguido por las codificaciones del desplazamiento y de la longitud —en este orden—. Los desplazamientos pequeños, se codifican como 1, seguido de 7 bits de desplazamiento; los grandes, se codifican como 0 seguido de 11 bits de desplazamiento (recuérdese que el tamaño del buffer es 2^{11}). La longitud se codifica conforme a la Tabla 3.6. La cadena de 9 bits —110000000— se escribe, como marcador de finalización, tras el último elemento de la secuencia de salida.

```
(QIC-122 Descripción BNF)

<Compressed-Stream> ::= [<Compressed-String>] <End-Marker>
<Compressed-String> ::= 0<Raw-Byte> | 1<Compressed-Bytes>
<Raw-Byte> ::= <b><b><b><b><b><b><b><b> (Byte 8-bits)
<Compressed-Bytes> ::= <offset><length>
<offset> ::= 1<b><b><b><b><b><b><b> (Desplazamiento de 7 bits)
             |
             0<b><b><b><b><b><b><b><b><b><b><b><b> (Desplazamiento de 11 bits)
<length> ::= (Conforme con la tabla de longitudes)
<End-Marker> ::= 110000000 (Bytes comprimidos con offset=0)
<b> ::= 0 | 1
```

Figura 3.7: Definición BNF de QIC-122.

◇ **Ejercicio 3.6 (sol. en pág. 1072):** ¿Cómo puede identificar el decodificador el marcador de finalización?

Cuando el algoritmo de búsqueda llega al extremo derecho del buffer, desplaza el contenido del mismo a la izquierda, e introduce el siguiente carácter en la posición de más a la derecha del buffer. El decodificador realiza el proceso inverso que el codificador (compresión simétrica).

> Cuando observé cada luminosa criatura de perfil —desde el punto de vista de su cuerpo—, su forma ahuevada era como un yoyó asimétrico gigante que permanecía en pie de canto, o como una olla redonda descansando de costado con su tapa abierta. La parte similar a una tapa era la cubierta frontal; abarcaba —tal vez— una quinta parte del grosor total del capullo.
>
> —Carlos Castaneda, *El fuego desde dentro* (1984)

La Figura 3.7 es una descripción precisa del proceso de compresión, expresada en BNF —que es un metalenguaje utilizado para describir los procesos y los lenguajes formales sin ambigüedades—. BNF usa los siguientes *metasímbolos*:

::=	El símbolo a la izquierda se define con la expresión a la derecha.
< `expresión` >	Una expresión que todavía no se ha definido.
|	Un OR lógico.
[ ]	Opcional. La expresión entre corchetes sucede cero o más veces.
( )	Un comentario.
`0, 1`	Los bits 0 y 1.

(Las aplicaciones especiales de BNF pueden requerir más símbolos.)

La Tabla 3.8 muestra los resultados de la codificación de la cadena de 16 símbolos: `ABAAAAAACABABABA`. El lector puede comprobar fácilmente[10] que la secuencia de salida está formada por los 10 bytes siguientes —en hexadecimal—:

$$20 \quad 90 \quad 88 \quad 38 \quad 1C \quad 21 \quad E2 \quad 5C \quad 15 \quad 80.$$

[10] Recuérdese que para obtener el valor hexadecimal de un número binario basta con dividirlo en grupos de 4 bits y generar el hexadecimal —un número de 0 a F— correspondiente de cada uno de ellos . Por ejemplo, $0001\,1100_2 = 1C_{16}$, pues $0001_2 = 1_{16}$ y $1100_2 = 12_{10} = C_{16}$.

Byte en bruto (Raw-Byte) "`A`"	0 01000001
Byte en bruto (Raw-Byte) "`B`"	0 01000010
Byte en bruto (Raw-Byte) "`A`"	0 01000001
Cadena "`AAAAA`", `offset` = 1	1 1 0000001 1100
Byte en bruto (Raw-Byte) "`C`"	0 01000011
Cadena "`ABA`" , `offset` = 9	1 1 0001001 01
Cadena "`BABA`", `offset` = 2	1 1 0000010 10
Marca de finalización (End-Marker)	1 1 0000000

Tabla 3.8: Ejemplo de codificación.

3.7. LZX

En 1995, Jonathan Forbes y Tomi Poutanen desarrollaron una variante de LZ (posiblemente influenciada por *Deflate*) que llamaron LZX. La característica principal de la primera versión de LZX fue la forma en que codificaba los desplazamientos de las coincidencias —que pueden ser grandes—, segmentando el tamaño del búfer de búsqueda. Implementaron LZX en un ordenador personal *Amiga* e incluyeron una característica con la que los datos se agrupaban en grandes bloques, en lugar de comprirse como una sola unidad.

Casi al mismo tiempo, Microsoft diseñó un nuevo formato de instalación de medios que denominó —en analogía con un archivador[11]— ficheros *cabinet*. Un archivo *cabinet* tiene como extensión de nombre `.cab` y puede estar formado por varios archivos de datos concatenados en una sola unidad y comprimidos. Inicialmente, Microsoft ha utilizado dos métodos de compresión para comprimir los archivos *cabinet*, MSZIP —que es sólo otro nombre para *Deflate*— y Quantum —un codificador basado en diccionarios de grandes ventanas que utiliza la codificación aritmética—. Quantum fue desarrollado por David Stafford.

Más tarde Microsoft utilizó los archivos *cabinet* en su Cabinet Software Development Kit[12] (SDK). Se trata de un paquete de software que proporciona a los desarrolladores de software las herramientas necesarias para emplear los archivos *cabinet* en cualquier aplicación que implementen.

En 1997, Jonathan Forbes empezó a trabajar para Microsoft y modificó LZX para comprimir los archivos *cabinet*. Microsoft publicó una especificación oficial para los ficheros *cabinet*, incluyendo los métodos MSZIP y LZX, pero excluyendo el codificador Quantum. La descripción LZX contenía errores de tal envergadura que no era posible crear ninguna implementación basándose en ella.

La documentación de LZX está disponible en el archivo ejecutable `cabsdk.exe` que podía obtenerse[13] en la siguiente dirección de internet:
http://download.microsoft.com/download/platformsdk/cab/2.0/w98nt42kmexp/en-us/.
Después de desempaquetar el archivo ejecutable, la documentación se encuentra en el archivo `LZXFMT.DOC`.

LZX es una variante de LZ77 que escribe en la secuencia comprimida, o bien caracteres sin localizar o bien pares (desplazamiento, longitud). Actualmente, los datos que forman parte de dicha secuencia son: códigos de tamaño variable para los caracteres no localizados, desplazamientos, y longitudes. El tamaño del buffer de búsqueda es una potencia de 2 —entre 2^{15} y 2^{21}—. LZX utiliza árboles estáticos de Huffman canónicos —Sección 2.8.6—, para proporcionar códigos prefijo de tamaño variable para los tres tipos de datos. Hay 256 valores posibles para los caracteres, pero los desplazamientos pueden ser numerosos y diferentes, así como las longitudes . Como consecuencia, los árboles de Huffman

[11] File cabinet.
[12] Kit de desarrollo de software cabinet.
[13] El Microsoft Cabinet SDK original, ya no está disponible en el sitio web de Microsoft (probablemente forme parte de algún otro SDK de Microsoft), pero aún se puede obtener buscando en internet: `cabsdk.exe`.

son grandes; pero cualquier archivo *cabinet* comprimido mediante LZX necesita sólo una pequeña parte de cada árbol. Estas partes están integradas en los datos comprimidos. Debido a que un único archivo *cabinet* puede estar formado por varios ficheros de datos, cada uno de ellos se comprime por separado y se escribe en la secuencia comprimida como un bloque —incluyendo las partes de los árboles requeridas—. A continuación comentamos las demás características importantes de LZX:

- **Desplazamientos repetidos**. Los desarrolladores de LZX observaron que ciertos desplazamientos tienden a repetirse; i.e., si una cadena determinada se comprime con un par $(74, \text{longitud})$, entonces existe una alta probabilidad de que el desplazamiento 74 se vaya a utilizar de nuevo pronto. En consecuencia, se asignaron los tres códigos especiales —0, 1 y 2— para codificar los tres desplazamientos más recientes. El desplazamiento actual asociado con cada uno de los códigos varía continuamente. Denotamos por $R0$ —el desplazamiento más reciente—, $R1$ —el segundo más reciente— y $R2$ —el tercero más reciente— (estos desplazamientos no deben repetirse a sí mismos; i.e., ninguno debe ser 0, 1 ó 2). Consideramos a $R0$, $R1$ y $R2$ una lista corta con un algoritmo de actualización similar a una cola LRU (*Least-Recently Used* —menos recientemente utilizado—). Las tres cantidades se inicializan a 1 y se actualizan como sigue. Suponiendo que el desplazamiento actual es X, entonces:

$$\begin{gathered}\texttt{si } X \neq R0 \,y\, X \neq R1 \,y\, X \neq R2, \texttt{ entonces } R2 \leftarrow R1,\ R1 \leftarrow R0,\ R0 \leftarrow X,\\ \texttt{si } X = R0, \texttt{ entonces no hacer nada},\\ \texttt{si } X = R1, \texttt{ entonces intercambiar } R0 \text{ y } R1,\\ \texttt{si } X = R2, \texttt{ entonces intercambiar } R0 \text{ y } R2.\end{gathered}$$

  Puesto que los códigos 0, 1 y 2 están asignados a los tres últimos desplazamientos, a un desplazamiento de 3 se le asigna el código 5, y —en general— a un desplazamiento x, se le asigna el código $x+2$. El mayor desplazamiento es el tamaño del buffer de búsqueda menos 3, y su código asignado es el tamaño del buffer de búsqueda menos 1.

- **Codificador de preprocesamiento**. LZX fue diseñado para comprimir archivos *cabinet* de Microsoft, que forman parte del sistema operativo Windows. Los ordenadores que utilizan este sistema están generalmente basados en microprocesadores fabricados por Intel, y entre 1980 y 1990 —antes de la introducción de los Pentium—, estos microprocesadores pertenecían a la conocida familia 80x86. El modo de preprocesamiento del codificador LZX es seleccionado por el usuario cuando la secuencia de datos de entrada es un archivo ejecutable para un equipo 80x86. Este modo convierte las instrucciones `CALL` del 80x86 para utilizar direcciones absolutas en lugar de relativas.

- **Formato de salida en bloques**. LZX saca los datos comprimidos en bloques, donde cada bloque contiene caracteres en bruto, desplazamientos, longitudes de coincidencias —aquellos elementos que se han conseguido emparejar con otros existentes—, y los árboles de Huffman canónicos utilizados para codificar los tres tipos de datos. Un árbol de Huffman canónico puede ser reconstruido a partir de la longitud del camino seguido por cada uno de sus nodos. Por lo tanto, sólo se tienen que guardar en la salida las longitudes de trayectoria de cada árbol Huffman. LZX limita la profundidad de un árbol de Huffman a 16, por lo que cada nodo del árbol se representa en la salida por un número en el rango, de 0 a 16. Un 0 indica un nodo que falta —un código Huffman que no es utilizado por este bloque—. Si el árbol tiene que ser más grande, se escribe en la salida el bloque actual y la compresión se reanuda con árboles nuevos. Los nodos del árbol se almacenan en forma comprimida. Si varios nodos consecutivos del árbol son idénticos, entonces se utiliza la codificación *run-length*. Los tres números —17, 18 y 19— se utilizan para este propósito. De lo contrario, se guarda la diferencia (módulo 17) entre las longitudes de trayectoria del nodo actual y del nodo anterior. Este diferencia pertenece al

intervalo $[0, 16]$. Por lo tanto, lo que se escribe en la salida son los 20 enteros de 5 bits —de 0 a 19—; estos números enteros se codifican en un árbol de Huffman llamado pre-árbol. El pre-árbol se genera de forma dinámica de acuerdo a las frecuencias de ocurrencia de los 20 valores. El pre-árbol en sí tiene que formar parte de la salida, y se escribe como 20 enteros de 4 bits (un total de 80 bits), en la que cada número entero indica la longitud de la trayectoria de uno de los 20 nodos de árbol. Un camino de longitud cero indica que uno de los 20 valores no se utiliza en el bloque actual.

Los desplazamientos y las longitudes de las coincidencias se comprimen en un proceso complejo que implica varios pasos y se resume en la Figura 3.9. Los pasos individuales involucran muchas operaciones y el uso de varias tablas que están integradas tanto en el codificador como en el decodificador. Sin embargo, debido a que LZX no es un método de compresión importante, estos pasos no se han discutido aquí.

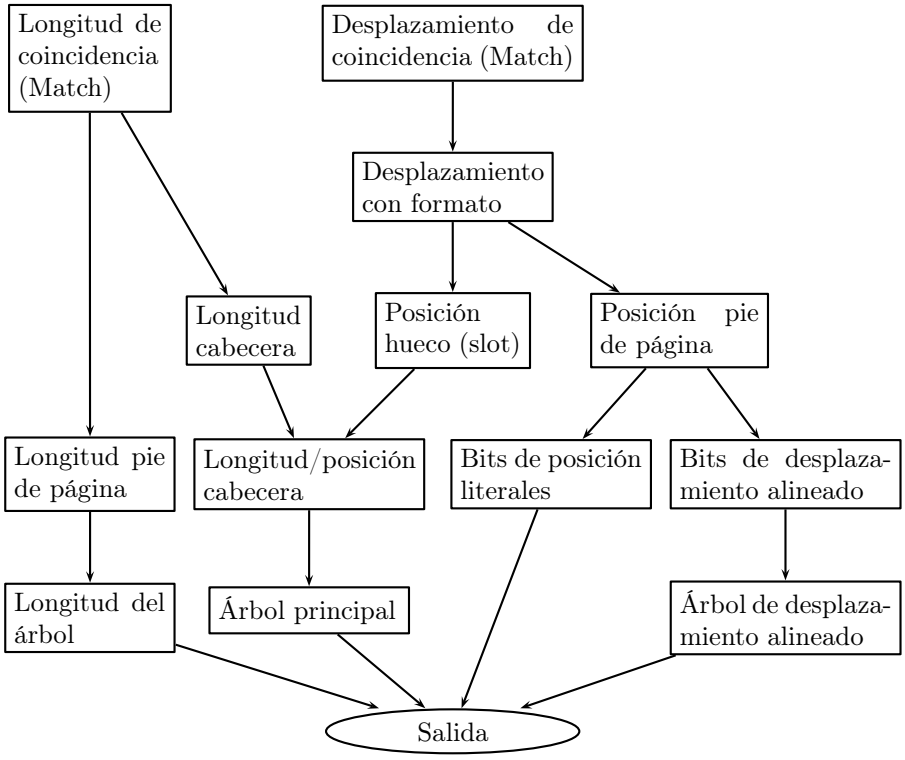


Figura 3.9: LZX: Procesamiento de los desplazamientos y las longitudes.

3.8. LZ78

El método LZ78 (referido a veces como LZ2) [Ziv y Lempel 78], no utiliza ningún buffer de búsqueda, buffer de preanálisis, o ventana deslizante. En cambio, hay un diccionario de cadenas previamente localizadas. Este diccionario comienza vacío —o casi vacío—, y su tamaño está limitado sólo por la cantidad de memoria disponible. El codificador saca tokens de dos campos: el primer campo, es un puntero al diccionario; el segundo, es el código de un símbolo. Los tokens no contienen la longitud

Diccionario		Token	Diccionario		Token
0	null				
1	“s”	(0, “s”)	8	“a”	(0, “a”)
2	“i”	(0, “i”)	9	“st”	(1, “t”)
3	“r”	(0, “r”)	10	“m”	(0, “m”)
4	“␣”	(0, “␣”)	11	“an”	(8, “n”)
5	“si”	(1, “i”)	12	“␣ea”	(7, “a”)
6	“d”	(0, “d”)	13	“sil”	(5, “l”)
7	“␣e”	(4, “e”)	14	“y”	(0, “y”)

Tabla 3.10: Primeros 14 pasos de codificación en LZ78.

de una cadena, ya que está implícita en el diccionario. Cada token se corresponde con una cadena de símbolos de entrada; y esa cadena se añade al diccionario, después de escribir el token en la secuencia de datos comprimidos. Nunca se elimina nada del diccionario, lo que es al mismo tiempo una ventaja sobre LZ77 (ya que las futuras cadenas pueden ser comprimidas incluso por cadenas encontradas en el pasado lejano) y una desventaja (porque el diccionario tiende a crecer rápido y a ocupar toda la memoria disponible).

El diccionario comienza con la cadena nula en la posición cero. A medida que los símbolos son introducidos y codificados, se añaden las cadenas al diccionario en las posiciones 1,2,... y así sucesivamente. Cuando se lee el siguiente símbolo —x— de la secuencia de entrada, se busca en el diccionario una entrada con la cadena de un solo símbolo x. Si no se encuentra ninguna, se añade x a la siguiente posición disponible en el diccionario, y se produce el token $(0, \mathtt{x})$. Este token indica la cadena “null x” (una concatenación de la cadena nula y x). Si se encuentra una entrada con x —en, digamos, la posición 37—, se lee el siguiente símbolo —y—, y se busca en el diccionario una entrada que contenga la cadena de dos símbolos xy. Si no se encuentra ninguna, entonces se añade la cadena xy en la siguiente posición disponible en el diccionario, y se genera el token $(37, \mathtt{y})$ como salida. Este token indica la cadena xy, ya que 37 es la posición en el diccionario de la cadena x. El proceso continúa hasta alcanzar el final de la secuencia de datos de entrada.

En general, se lee el símbolo actual y se convierte en una cadena de un elemento. A continuación, el codificador intenta localizarlo en el diccionario. Si lo encuentra, se lee el símbolo siguiente y se concatena con el primero para formar una cadena de dos elementos que el codificador —a continuación— trata de localizar en el diccionario. Mientras esas cadenas se encuentren en el diccionario, se siguen leyendo y concatenando más símbolos con la cadena. En un cierto punto, la cadena no se encontrará en el diccionario, por lo que el codificador la añadirá al diccionario y generará un token con la posición del último emparejamiento en el diccionario como su primer campo, y el último símbolo de la cadena —el que causó el fallo en la búsqueda— como su segundo campo. La Tabla 3.10 muestra los primeros 14 pasos en la codificación de la cadena:

“sir␣sid␣eastman␣easily␣teases␣sea␣sick␣seals”.

◇ **Ejercicio 3.7 (sol. en pág. 1072):** Complétese la Tabla 3.10.

En cada paso, la cadena agregada al diccionario —menos su último símbolo— es la que se está codificando. En una etapa típica de compresión, el diccionario comienza con cadenas cortas, pero a medida que el texto es introducido y procesado, se añaden cadenas cada vez más largas. El tamaño del diccionario puede ser fijo o puede determinarse por el tamaño de la memoria disponible cada vez que se ejecuta el programa de compresión LZ78 . Un gran diccionario puede contener más cadenas y permitir así la localización de largas secuencias de datos adyacentes; pero a cambio, los punteros son más largos —y por consiguiente, los tokens más grandes—, y la búsqueda en el diccionario, más lenta.

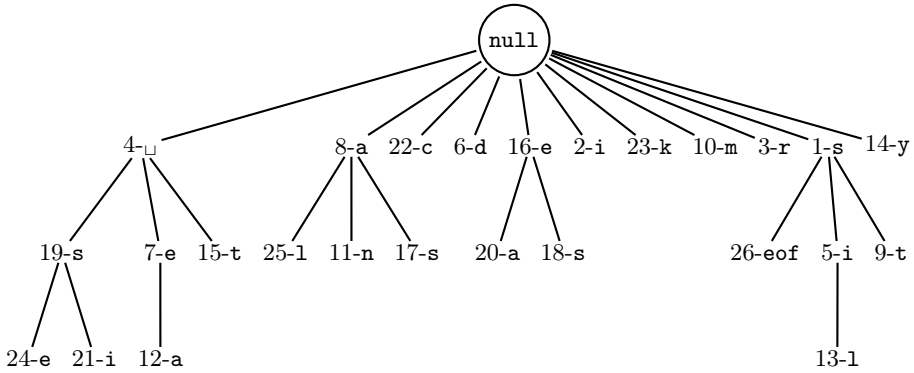


Figura 3.11: Un diccionario de árbol LZ78.

Una buena estructura de datos para el diccionario es un árbol, pero no un árbol binario. El árbol comienza con la cadena nula como raíz. Todas las cadenas que comienzan con la cadena nula —aquellas para las que el puntero del token es cero— se añaden al árbol, como hijos de la raíz. En el ejemplo anterior, son: `s`, `i`, `r`, ␣, `d`, `a`, `m`, `y`, `e`, `c` y `k`. Cada uno de ellos se convierte en la raíz de un árbol, como se muestra en la Figura 3.11. Por ejemplo, todas las cadenas que comienzan con `s` —las cuatro cadenas: `si`, `sil`, `st` y `s(eof)`— constituyen el subárbol del nodo `s`.

Asumiendo un alfabeto de símbolos de 8 bits, hay 256 símbolos diferentes, por lo que en principio, cada nodo del árbol puede tener hasta 256 hijos. El proceso de añadir un hijo a un nodo del árbol debe ser, por lo tanto, dinámico. Cuando el nodo se crea por primera vez, no tiene hijos —y no es necesario reservar memoria para ellos—. Al añadir un hijo al nodo, se le debe asignar un espacio de memoria. Dado que nunca se elimina ningún nodo, no hay necesidad de recuperar ese espacio, lo que simplifica un poco la gestión de la memoria.

Este árbol hace que resulte fácil buscar o añadir una cadena. Para buscar `sil` —por ejemplo— el programa busca el hijo `s` de la raíz; luego, el hijo `i` de `s`; y así sucesivamente, bajando por el árbol. A continuación mostramos algunos ejemplos:

1. Cuando se introduce la `s` de `sid` en el paso 5, el codificador encuentra el nodo "1-`s`" en el árbol —como un hijo de "`null`"—. Entonces, se introduce el siguiente símbolo —`i`—, pero el nodo `s` no tiene un hijo `i` (de hecho —en este instante— no tiene ninguno), por lo que el codificador agrega el nodo "5-`i`", como un hijo de "1-`s`" —lo que almacena eficazmente la cadena `si` en el árbol—.

2. Cuando se introduce el espacio en blanco entre `eastman` y `easily` en el paso 12, ocurre una situación similar. El codificador encuentra el nodo "4-␣", introduce `e`, localiza "7-`e`", introduce `a`; pero "7-`e`" no tiene un hijo "`a`", por lo que el codificador agrega el nodo "12-`a`" —lo que almacena eficazmente la cadena "␣`ea`" en el árbol—.

Un árbol del tipo aquí descrito se llama *trie*. En general, un trie es un árbol en el que la estructura de ramificación en cualquier nivel se determina sólo por parte de un ítem de datos, y no por el ítem entero (Sección 2.18.5). En el caso de LZ78, cada cadena agrega al árbol de manera eficaz exactamente un símbolo —y no una rama—.

Puesto que el tamaño total del árbol es limitado, se puede llenar durante la compresión. Esto, de hecho, sucede constantemente —excepto cuando la secuencia de datos de entrada es inusualmente pequeña—. El método original LZ78, no especifica qué hacer en tal caso, por lo que ofrecemos una lista de algunas soluciones posibles:

1. La solución más sencilla es congelar el diccionario en ese punto. No debe agregarse ningún nodo nuevo; el árbol se convierte en un diccionario estático, pero todavía se puede utilizar para codificar las cadenas.

2. Eliminar todo el árbol una vez que se llena, y comenzar con un árbol nuevo y vacío. Esta solución divide con eficacia los datos de entrada en bloques, cada uno con su propio diccionario. Si el contenido de la entrada varía de un bloque a otro, esta solución produce una buena compresión, ya que que eliminará un diccionario con cadenas que apenas se utilizarán en el futuro. Se puede decir que esta solución supone implícitamente, que los símbolos futuros se beneficiarán más de los nuevos datos que de los antiguos (la misma suposición implícita utilizada por LZ77).

3. La utilidad de compresión —`compress`— de UNIX (Sección 3.18) utiliza una solución más compleja. Cuando el diccionario está lleno, elimina algunas de las entradas *menos recientemente utilizadas*, para dejar espacio para las nuevas. Desafortunadamente, no existe un buen algoritmo para decidir qué entradas eliminar, y cuántas (sin embago, puede verse el procedimiento de *reutilización*, en la Sección 3.21).

El decodificador LZ78 efectúa la construcción y el mantenimiento del diccionario de la misma manera que el codificador. Por tanto, es más complejo que el decodificador LZ77.

3.9. LZFG

Edward Fiala y Daniel Greene han desarrollado varios métodos de compresión relacionados que son híbridos de LZ77 y LZ78 [Fiala y Greene 89]. Todos sus métodos se basan en el siguiente esquema. El codificador genera un archivo comprimido con tokens y literales (códigos ASCII puros) entremezclados. Hay dos tipos de tokens: *literal* y *copia*. Un token literal indica que a continuación se encuentra una cadena de literales; un token copia apunta a una cadena encontrada anteriormente en los datos. La cadena "`the␣boy␣on␣my␣right␣is␣the␣right␣boy`" produce, cuando se codifica,

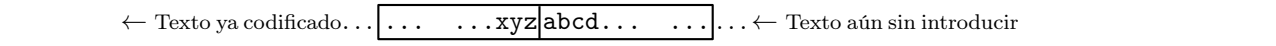
(literal 23)`the␣boy␣on␣my␣right␣is␣`(copia 4 23) (copia 6 13) (copia 3 29),

donde los tres tokens copia hacen referencia a las cadenas `the␣`, `right␣` y `boy`, respectivamente. Los métodos LZFG se comprenden mejor considerando cómo funciona el decodificador. El decodificador comienza con un gran buffer vacío, en el que se generan y desplazan las cadenas descomprimidas. Cuando el decodificador recibe un token (literal 23), introduce los 23 bytes siguientes —como códigos en ASCII puro— en el buffer, desplazando el contenido del buffer de manera que el último byte introducido sea el situado más a la derecha. Cuando el decodificador recibe (copia 4 23), copia la cadena de longitud 4 que comienza 23 posiciones antes del extremo derecho del buffer. Luego añade la cadena al buffer, al mismo tiempo que desplaza su contenido. Aquí se describen dos variantes de LZFG, denotadas por $A1$ y $A2$.

El esquema $A1$ emplea tokens literales de 8 bits y tokens copia de 16 bits. Un token literal tiene el formato `0000nnnn`, donde `nnnn` indica el número de bytes ASCII que siguen al token. Puesto que el campo de 4 bits `nnnn`, puede tener valores entre 0 y 15, éstos se interpretan con el significado "de 1 a 16". La cadena más larga posible de literales es, por tanto, de 16 bytes. El formato de un token copia es "`sssspp...p`", donde el campo `ssss` —4 bits distintos de cero— indica la longitud de la cadena a copiar, y el campo de 12 bits "`pp...p`", es un valor de desplazamiento que determina la localización del comienzo de la cadena en el buffer. Puesto que el campo `ssss` no puede ser cero, sólo se admiten valores entre 1 y 15 —que se interpretan como longitudes de cadena entre 2 y 16—. Los valores de desplazamiento pertenecen al intervalo $[0, 4095]$, y se interpretan como $[1, 4096]$.

El codificador comienza con un buffer de búsqueda vacío —de 4096 bytes de longitud—, y llena el buffer de preanálisis con los datos de entrada. En cada paso subsiguiente trata de crear un token

copia. Si no encuentra coincidencias en ese paso, el codificador crea un token literal. Supongamos que en un momento dado el buffer contiene:



El codificador trata de emparejar la cadena "`abc`..." del buffer de preanálisis, con las distintas cadenas del buffer de búsqueda. Si se encuentra una coincidencia (de al menos dos símbolos), escribe un token copia en la secuencia de datos comprimidos y desplaza a la izquierda los datos en los buffers tantas posiciones como el número de elementos coincidentes. Si no se encuentra una pareja, el codificador inicia un literal con `a`, y desplaza los datos a la izquierda una posición. A continuación, intenta emparejar la cadena "`bcd`..." con alguna del buffer de búsqueda. Si encuentra una coincidencia, produce un token literal como salida, seguido por un byte con la `a`, seguida de un token de localización. De lo contrario, el codificador anexa `b` al literal e intenta encontrar una coincidencia para "`cd`...". Los literales pueden ser de hasta 16 bytes de longitud, por lo tanto, la cadena "`the␣boy␣on␣my`..." —de más arriba— se codifica como[14]:

(lit. 16)`the␣boy␣on␣my␣ri`(lit. 7)`ght␣is␣`(cop. 4 23) (cop. 6 13) (cop. 3 29).

El método $A1$ se apropia de la idea del buffer de desplazamiento de LZ77; pero también comporta como LZ78, porque crea tokens de dos campos. Ésta es la razón por la que se puede considerar un híbrido de los dos métodos LZ originales. Cuando $A1$ comienza, crea principalmente literales; pero cuando alcanza velocidad (llena el buffer de búsqueda), cuenta con la adaptación fuerte, por lo que se generan más y más tokens copia en la secuencia de datos comprimidos.

El método $A2$ utiliza un buffer de búsqueda más grande (de hasta 21K bytes de longitud). Esto mejora de compresión, porque se pueden encontrar copias largas, pero plantea el problema del tamaño del token. Un buffer de búsqueda grande implica grandes desplazamientos en tokens copia; copias largas implican grandes campos "longitud" en esos tokens. Al mismo tiempo, esperamos que ambos campos —desplazamiento y "longitud"— de un token copia típico sean pequeños, ya que la mayor parte de las coincidencias se encuentran cercanas al principio del buffer de búsqueda. La solución es utilizar un código de tamaño variable para esos campos, y $A2$ utiliza los códigos unarios generales de la Sección 2.3.1. El campo "longitud" de un token copia se codifica con un código $(2, 1, 10)$ (Tabla 2.5), lo que hace posible localizar cadenas coincidentes de hasta 2 044 símbolos de longitud. Obsérvese que la longitud del código $(2, 1, 10)$ está comprendida entre 3 y 18 bits.

Los cuatro primeros códigos del unario general $(2, 1, 10)$ son 000, 001, 010 y 011. Los tres últimos, representan longitudes de búsqueda de dos, tres y cuatro, respectivamente (recuérdese que la longitud mínima de coincidencia es de 2). El primero —código 000— está reservado para denotar un literal. La longitud del literal se anexa a continuación, y se codifica con el código $(0, 1, 5)$. Un literal puede tener —por tanto— hasta 63 bytes de longitud, y el campo longitud del token literal se codifica en 1 a 10 bits. En caso de coincidencia, el campo "longitud" no es 000 y va seguido por el campo de desplazamiento, que se codifica con el código $(2, 1, 10)$ (Tabla 3.12). Este código consta de 21K valores, y el tamaño máximo del código es de 16 bits (sin embargo, véanse los puntos 2 y 3 más adelante).

A continuación se comentan tres refinamientos más, empleados por el método $A2$, que logran una ligera mejoría — un 1 % ó 2 %— en la compresión:

1. Un literal de longitud máxima —63 bytes— puede ir immediatamente seguido por otro literal, o un por un token copia de cualquier longitud; pero un literal de menos de 63 bytes debe ir seguido por un token copia de *al menos tres símbolos* (o por el indicador fin-de-archivo). Este hecho se utiliza para aprovechar los códigos $(2, 1, 10)$ más pequeños, para indicar la longitud de la cadena emparejada. Normalmente, los códigos 000, 001, 010 y 011 indican que no hay coincidencias, y que se han logrado emparejamientos de longitud 2, 3 y 4, respectivamente. Sin embargo, un

[14]Las palabras "lit." y "cop.", son abreviaturas de "literal" y "copia", respectivamente.

n	$a = 10 + n \cdot 2$	$n-ésima$ *palabra clave*	*Número de palabras clave*	*Rango de enteros*
0	10	$0\underbrace{x \ldots x}_{10}$	$2^{10} =$ 1K	0-1023
1	12	$10\underbrace{xx \ldots x}_{12}$	$2^{12} =$ 4K	1024-5119
2	14	$11\underbrace{xx \ldots xx}_{14}$	$2^{14} =$ 16K	5120-21503
		Total	21 504	

Tabla 3.12: El código unario general $(10, 2, 14)$.

token copia tras de un token literal de menos de 63 bytes utiliza esos códigos —000, 001, 010 y 011— para indicar emparejamientos de longitud 3, 4, 5 y 6, respectivamente. De esta manera, la longitud máxima de coincidencia puede ampliarse a 2 046 símbolos, en vez de 2 044.

2. El campo desplazamiento se codifica con el código $(10, 2, 14)$, que consta de 21K valores y cuyos códigos individuales varían en tamaño desde 11 hasta 16 bits. Para archivos más pequeños, los grandes desplazamientos pueden ser innecesarios, y se pueden utilizar otros códigos del unario general —con códigos individuales más cortos—. El método $A2$, por lo tanto, utiliza códigos de la forma $(10 - d, 2, 14 - d)$ para $d = 10, 9, 8, \ldots, 0$. Para $d = 1$, el código $(9, 2, 13)$ tiene $2^9 + 2^{11} + 2^{13} = 10\,752$ valores, y los códigos individuales varían en tamaño, de 9 a 15 bits. Para $d = 10$, el código $(0, 2, 4)$ contiene $2^0 + 2^2 + 2^4 = 21$ valores, y los códigos tienen entre 1 y 6 bits de longitud. El método $A2$ comienza con $d = 10$ —lo que significa que inicialmente utiliza código $(0, 2, 4)$— y un buffer de búsqueda de 21 bytes. Cuando el buffer se llena (lo que indica una cadena de entrada de más de 21 bytes), el algoritmo $A2$ cambia a $d = 9$ —código $(1, 2, 5)$— y aumenta el tamaño de búfer de búsqueda a 42 bytes. Este proceso continúa hasta que se ha codificado la cadena de entrada completamente o hasta que se alcanza $d = 0$ —a partir de este momento, utiliza el código $(10, 2, 14)$ hasta el final—. ¡Es mucho trabajo para conseguir un pequeño aumento de la compresión! (Véase el análisis sobre la disminución del rendimiento —*una palabra para el sabio* (pág. 7)— en el Prefacio).

3. Cada uno de los códigos $(10 - d, 2, 14 - d)$ requiere un buffer de búsqueda de un cierto tamaño —desde 21 hasta $21K = 21\,504$ bytes, según el número de códigos que contenga—. Si el usuario quiere, por alguna razón, para asignar al buffer de búsqueda un tamaño diferente, entonces, no se pueden utilizar algunos de los códigos largos, lo que permite reducir un poco el tamaño de los códigos individuales. Por ejemplo, si el usuario decide utilizar un buffer de búsqueda $16K = 16\,384$ bytes, entonces debe utilizarse el código $(10, 2, 14)$ [porque el siguiente código —$(9, 2, 13)$ — contiene sólo 10 752 valores]. El código $(10, 2, 14)$ contiene $21K = 21\,504$ códigos individuales, por lo que los 5 120 códigos más largos nunca se van a utilizar. El último grupo de códigos —"11" seguido por 14 bits— en $(10, 2, 14)$, contiene $2^{14} = 16\,384$ códigos individuales diferentes, de los cuales sólo se van a utilizar 11 264. De los 11 264 códigos, los primeros 8 192 se pueden representar como "11" seguido por $[\log_2 11\,264] = 13$ bits, y sólo los 3 072 códigos restantes requieren $[\log_2 11\,264] = 14$ bits a continuación del primer "11". Así, terminamos con 8 192 códigos de 15 bits y 3 072 códigos de 16 bits, en lugar de 11 264 códigos de 16 bits, una mejora muy pequeña.

Estas tres mejoras ilustran las grandes longitudes que los investigadores están dispuestos admitir a fin de mejorar sus algoritmos muy ligeramente.

> La experiencia demuestra que afinar un algoritmo de compresión hasta los últimos bits de redundancia de los datos desemboca en rendimientos decrecientes. La modificación de un algoritmo para mejorar la compresión en un 1 %, puede aumentar el tiempo de ejecución en un 10 % (de la Introducción).

El LZFG "corpus" de algoritmos contiene cuatro métodos más. $B1$ y $B2$ son similares a $A1$ y $A2$, pero más rápidos —debido a la forma en que calculan los desplazamientos—. Sin embargo, se sacrifica algo la razón de compresión. $C1$ y $C2$ van en la dirección opuesta: logran una compresión ligeramente mejor que $A1$ y $A2$, perdiendo velocidad de procesamiento. (LZFG ha sido patentado, un tema que se discute en la Sección 3.30.)

3.10. LZRW1

Desarrollado por Ross Williams [Williams 91a] y [Williams 91b] como una sencilla y rápida variante de LZ77, LZRW1 también está relacionado con el método $A1$ de LZFG (Sección 3.9). La idea principal es encontrar un emparejamiento en un solo paso, utilizando una tabla hash. Esto es rápido, pero no muy eficiente, ya que la cadena localizada no es siempre la más larga. Comenzaremos con una descripción del algoritmo, seguida del formato de la secuencia de datos comprimidos , y concluiremos con un ejemplo.

El método utiliza toda la memoria disponible como un buffer y codifica los datos de entrada en bloques. Se introduce un bloque en el buffer, se lee, y se codifica por completo; después se procede de igual manera con el siguiente bloque, y así sucesivamente. La longitud del buffer de búsqueda es de $4K$ y la del buffer de preanálisis es de 16 bytes. Estos dos buffers se deslizan por el bloque de entrada, en la memoria —de izquierda a derecha—. Sólo es necesario mantener un puntero —`p_src`— que apunta al inicio del buffer de preanálisis. El puntero `p_src` se inicializa a 1 y se incrementa después de la codificación de cada frase; por lo tanto, ambos buffers se desplazan hacia la derecha tantas posiciones como la longitud de la frase. La Figura 3.13 muestra cómo el buffer de búsqueda comienza vacío, crece hasta $4K$, y entonces comienza a deslizarse hacia la derecha, siguiendo al buffer de preanálisis.

Los tres caracteres del extremo izquierdo del buffer de lectura anticipada se convierten en un número I de 12 bits —un puntero índice—, que se utiliza para indexar un array[15] de $2^{12} = 4\,096$ punteros. Se recupera un puntero P y se sustituye inmediatamente en la matriz por `p_src`. Si P apunta fuera del buffer de búsqueda, no hay ninguna coincidencia; el primer carácter en el buffer de lectura anticipada se saca como un literal, y `p_src` se incrementa en 1. Lo mismo se hace si P apunta al interior del buffer de búsqueda, pero para una cadena que no coincide con la del buffer de preanálisis. si P apunta a un emparejamiento de al menos tres caracteres, el codificador encuentra la cadena coincidente más larga (un máximo de 16 caracteres), saca un ítem de localización e incrementa `p_src` tantas veces como la longitud de la cadena emparejada. Este proceso se muestra en la Figura 3.15. Un punto interesante a destacar es que la matriz de punteros, no tiene que ser inicializada cuando el codificador comienza a funcionar, ya que éste comprueba cada puntero. Inicialmente, todos los punteros son aleatorios, pero a medida que son reemplazados, apuntan —cada vez más— a los emparejamientos reales.

La salida del codificador LZRW1 (Figura 3.16) se compone de grupos, cada uno de los cuales comienza con una *palabra de control* de 16 bits, seguida de 16 ítems. Cada ítem es, o bien un literal de 8 bits o bien un ítem de copia de 16 bits (una localización de una cadena encontrada), que consta de un campo b de 4 bits (donde la longitud es $b + 1$) y un desplazamiento de 12 bits (los campos a y c). La longitud del campo admite valores entre 3 y 16. Los 16 bits de la palabra de control son flags de cada uno de los 16 ítems siguientes (un flag a 0 denota un literal, y un flag a 1 denota un ítem

[15] Recuérdese que un array —arreglo o matriz— es una secuencia de datos adyacentes del mismo tipo.

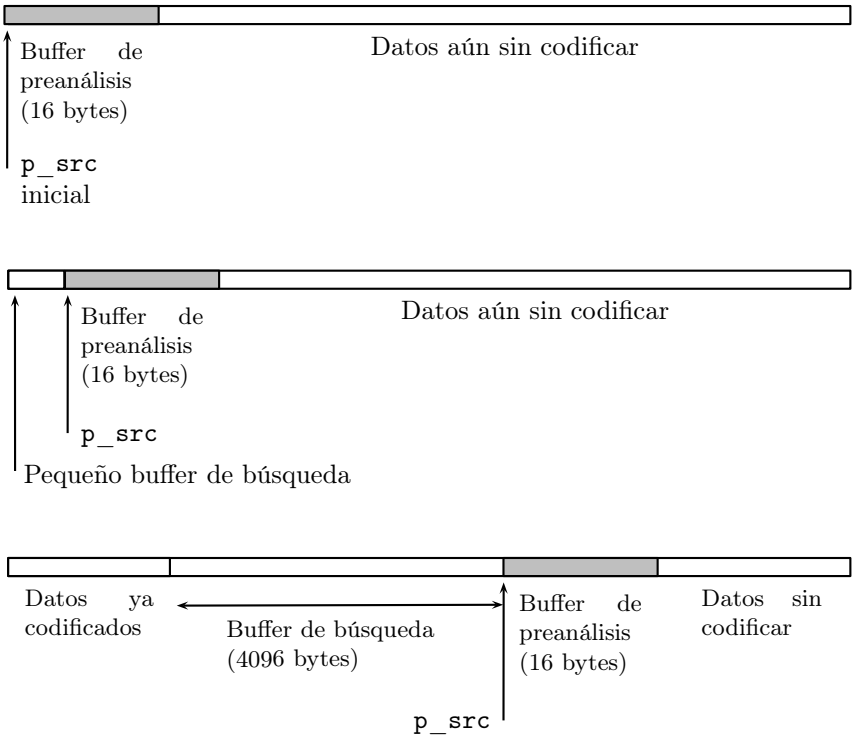


Figura 3.13: Desplazamiento de los búferes de búsqueda y preanálisis en LZRW1.

de localización). Obviamente, los grupos tienen longitudes diferentes. El último grupo puede contener menos de 16 ítems.

El decodificador es incluso más simple que el codificador, ya que no necesita la matriz de punteros. Mantiene un buffer de gran tamaño utilizando un puntero `p_src` de la misma forma que el codificador. El decodificador lee una palabra de control de la secuencia de datos comprimidos y utiliza sus 16 bits para leer 16 ítems. Un ítem literal se decodifica añadiéndolo al buffer e incrementando `p_src` en 1. Un ítem de copia se decodifica restando el desplazamiento de `p_src`, localiza una cadena del buffer de búsqueda —de longitud, la indicada por el campo longitud—, y la agrega al buffer. Luego incrementa `p_src` tantas veces como indique la longitud.

La Tabla 3.14 ilustra los primeros siete pasos de la codificación de "`that␣thatch␣thaws`". los valores producidos por la función hash (función de dispersión) son arbitrarios —nótese que los índices hash iguales corresponden a cadenas semejantes—. Inicialmente, todos los punteros son aleatorios (indicado por "*nada*"), pero son reemplazados por otros útiles rápidamente.

◇ **Ejercicio 3.8 (sol. en pág. 1072):** Resúmanse los últimos pasos en una tabla similar a la Tabla 3.14, y escríbase la cadena de datos comprimidos resultante —en formato binario—.

Las pruebas realizadas por el desarrollador original indican que el rendimiento de LZRW1 es alrededor de un 10 % peor que LZC (la utilidad `compress` de UNIX), pero es cuatro veces más rápido. Asimismo, es alrededor del 4 % peor que LZFG (el método *A*1), pero se ejecuta diez veces más rápido. Por lo tanto, es adecuado para aquellos casos en que la velocidad es más importante que el rendimiento de la compresión. Una implementación del lenguaje ensamblador del 68 000 ha requerido —en promedio— la ejecución de sólo 13 instrucciones máquina para comprimir, y cuatro instrucciones para descomprimir, un byte.

p_src	3 caracteres	Índice hash	P	Salida	Salida binaria
1	tha	4	*nada* → 1	t	01110100
2	hat	6	*nada* → 2	h	01101000
3	at␣	2	*nada* → 3	a	01100001
4	t␣t	1	*nada* → 4	t	01110100
5	␣th	5	*nada* → 5	␣	00100000
6	tha	4	4 → 1	6, 5	0000 | 0011 | 00000101
10	ch␣	3	*nada* → 10	c	01100011

Tabla 3.14: Primeros 7 pasos de la codificación de "that␣thatch␣thaws".

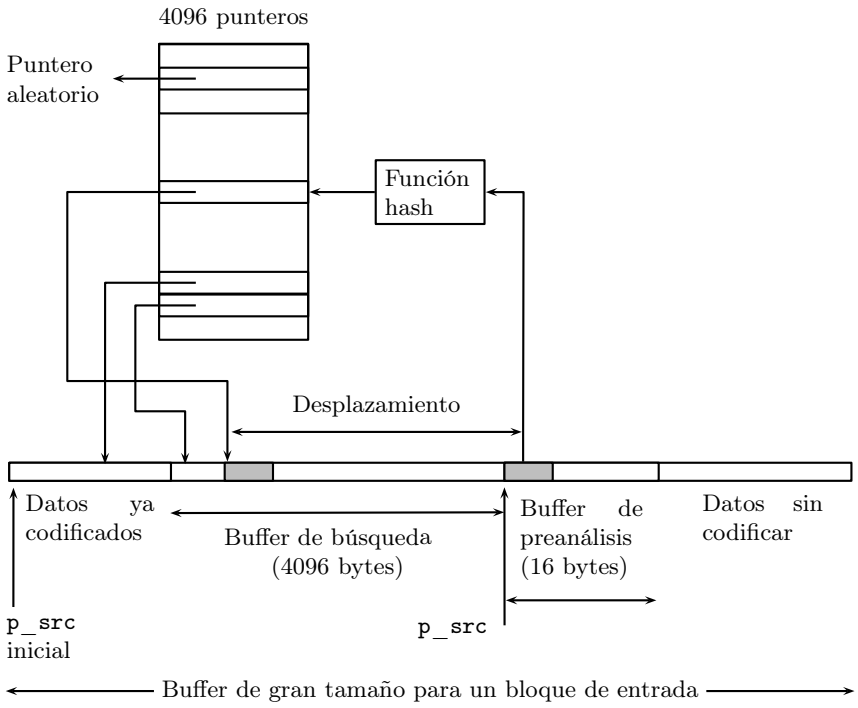


Figura 3.15: El codificador LZRW1.

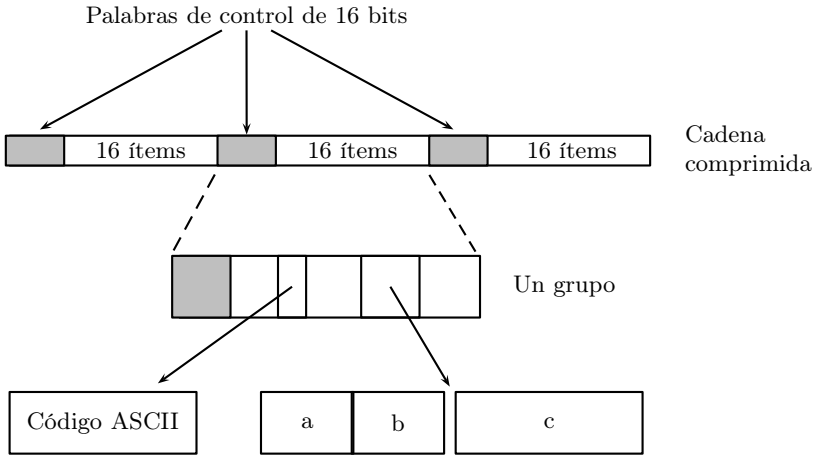


Figura 3.16: Formato de la salida.

3 bits:	000	001	010	011	100	101	110	111
Longitud:	2	3	4	5	6	7	8	16

Tabla 3.17: Codificación de la longitud en LZRW4.

◇ **Ejercicio 3.9 (sol. en pág. 1072):** Muéstrese una situación práctica donde la velocidad de compresión es más importante que la razón de compresión.

3.11. LZRW4

LZRW4 es una variante de LZ77, basado en las ideas de Ross Williams acerca de las posibles formas de combinar un método de diccionario con la predicción (Sección 3.31). LZRW4 también toma algunas ideas de LZRW1. Utiliza un buffer de 1 Mbyte, donde tanto el buffer de búsqueda como el de lectura anticipada se desplazan de izquierda a derecha. En cualquier momento del proceso de codificación, se utiliza el contexto de orden 2 del símbolo actual (los dos símbolos más recientes del buffer de búsqueda) para predecirlo. Con los dos símbolos que constituyen el contexto se construye un número —I— de 12 bits, que se usa como un índice a un array —A— de particiones de $2^{12} = 4\,096$ elementos. Cada partición contiene 32 punteros a los datos de entrada del búfer de 1 Mbyte (cada puntero, por lo tanto, tiene 20 bits de longitud —pues $2^{20} = 1Mbyte$—).

Los 32 punteros de la partición —$A[I]$— se verifican para encontrar el emparejamiento más largo entre el buffer de lectura anticipada y los datos de entrada que han aparecido hasta ahora. Una vez localizado, se selecciona y se codifica en 8 bits. Los 3 primeros bits contienen el número de elementos con pareja, en concordancia con la Tabla 3.17; los 5 bits restantes, identifican el puntero dentro de la partición. Dicho número de 8 bits se denomina *ítem copia*. Si no se ha encontrado un emparejamiento, se codifica un literal en 8 bits. Para cada ítem, se prepara un bit extra: un 0 para un literal y un 1 por un ítem copia. Los bits extra se acumulan en grupos de 16, y cada grupo se saca —como en LZRW1— precediendo a los 16 ítems a los que se refiere.

Las particiones se actualizan continuamente moviendo los punteros "buenos" hacia el principio de su partición. Cuando se encuentra un emparejamiento, el codificador intercambia el puntero seleccionado con aquel situado a medio camino del final de la partición (Figura 3.18a,b). Si no hay cadenas coincidentes, los 32 punteros de la partición se desplazan hacia la izquierda y se introduce el puntero

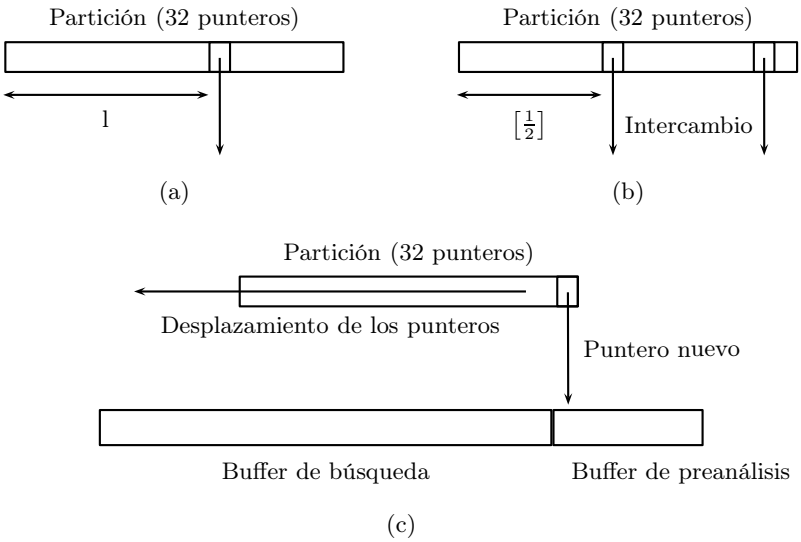


Figura 3.18: Actualización de una partición en LZRW4.

nuevo por la derecha, apuntando al símbolo actual (Figura 3.18c).

> La Reina Roja sacudió la cabeza. "Puedes llamarlo 'disparate' si quieres", dijo ella; "sin embargo, ¡he oído disparate, comparado con lo que sería tan razonable como un diccionario!"
>
> —Lewis Carroll, *A través del espejo* (1872)

3.12. LZW

Esta es una variante popular de LZ78, desarrollado por Terry Welch en 1984 ([Welch 84] y [Phillips 92]). Su característica principal es la eliminación del segundo campo de un token. Un token LZW consta de exactamente un puntero al diccionario. Para comprender mejor LZW, olvidemos temporalmente que el diccionario es un árbol, y pensemos en él como un array de cadenas de tamaño variable. El método LZW comienza inicializando el diccionario con todos los símbolos del alfabeto. En el caso más común —en el que cada símbolo se representa con 8 bits— se ocupan las primeras 256 entradas del diccionario (de la 0 a la 255), antes de la introducción de cualquier dato. Debido a que el diccionario se ha inicializado, el siguiente carácter de entrada se encuentra siempre en el diccionario. Por ello, un token LZW puede consistir en sólo un puntero y no necesita contener un código de carácter, como en LZ77 y LZ78.

(LZW ha sido patentado —y por muchos años—; mientras no expire, su uso requiere una licencia. Este problema se discute en la Sección 3.30.)

El principio del codificador LZW consiste en introducir los símbolos uno por uno y acumularlos en una cadena I. Con cada símbolo de entrada, se concatena el mismo con I, y se busca en el diccionario la nueva cadena resultante. Este proceso se repite mientras I se encuentre en el diccionario. En un momento determinado, la adición del símbolo siguiente —x— provoca un fallo en la búsqueda; la cadena I está en el diccionario, pero cadena Ix (el símbolo x concatenado con I), no. En este punto, el codificador (1) ofrece como salida el puntero del diccionario que apunta a la cadena I, (2) guarda la cadena Ix (que ahora se llama *frase*) en la siguiente entrada del diccionario disponible, y (3) inicializa

```
desde i:=0 hasta 255 hacer
  añadir i como una cadena de 1 símbolo al diccionario;

añadir λ al diccionario;
di:=índice en el diccionario para λ;

repetir
  leer(ch);
  si ≪ di, ch ≫ está en el diccionario entonces
    di:=índice en el diccionario para ≪ di, ch ≫;
  si no
    sacar(di);
    añadir ≪ di, ch ≫ al diccionario;
    di:=índice en el diccionario para ch;
  fin si;
hasta fin de datos de entrada;
```

Figura 3.21: El algoritmo LZW.

la cadena I con el símbolo x. Para ilustrar este proceso, una vez más utilizaremos la cadena de texto "`sir␣sid␣eastman␣easily␣teases␣sea␣sick␣seals`". Los pasos son los siguientes:

1. Se inicializan las entradas —de 0 a 255— del diccionario con los 256 bytes de 8 bits.

2. Se introduce el primer símbolo —`s`— y se localiza en el diccionario (en la entrada 115, ya que este es el código ASCII de `s`). Se introduce el siguiente símbolo —`i`— , pero `si` no se encuentra en el diccionario. El codificador realiza los siguientes pasos: (1) saca 115, (2) guarda la cadena `si` en la siguiente entrada disponible del diccionario —entrada 256—, y (3) inicializa I con el símbolo `i`.

3. Se introduce la `r` de `sir`, pero la cadena `ir` no está en el diccionario. El codificador: (1) saca 105 (el código ASCII de `i`), (2) guarda la cadena `ir` en la siguiente entrada disponible del diccionario —entrada 257—, y (3) inicializa I con el símbolo `r`.

La Tabla 3.19 resume todos los pasos de este proceso. La Tabla 3.20 muestra algunas de las 256 entradas originales en el diccionario LZW, más las añadidas durante la codificación de la cadena anterior. La secuencia de salida completa es (sólo forman parte de la salida los números, no las cadenas entre paréntesis) la siguiente:

115 (`s`), 105 (`i`), 114 (`r`), 32 (`␣`), 256 (`si`), 100 (`d`), 32 (`␣`), 101 (`e`), 97 (`a`), 115 (`s`), 116 (`t`), 109 (`m`), 97 (`a`), 110 (`n`), 262 (`␣e`), 264 (`as`), 105 (`i`), 108 (`l`), 121 (`y`), 32 (`␣`), 116 (`t`), 263 (`ea`), 115 (`s`), 101 (`e`), 115 (`s`), 259 (`␣s`), 263 (`ea`), 259 (`␣s`), 105 (`i`), 99 (`c`), 107 (`k`), 281 (`␣se`), 97 (`a`), 108 (`l`), 115 (`s`), `eof`.

La Figura 3.21 es un listado en pseudocódigo del algoritmo. Denotamos con λ a la cadena vacía, y con $\ll$ `a, b` $\gg$ a la concatenación de las cadenas `a` y `b`.

La línea "`añadir ≪ di, ch ≫ al diccionario`", es de especial interés. Está claro que en la práctica, el diccionario puede llenarse. Por consiguiente, esta línea debe incluir una prueba para determinar si el diccionario está completo, así como ciertas medidas para el caso en que esto suceda.

Puesto que las primeras 256 entradas del diccionario ya están ocupadas desde el comienzo, los punteros que forman parte del diccionario necesitan más de 8 bits. Una sencilla implementación podría utilizar punteros de 16 bits, que permiten un diccionario con $64K$ entradas (donde $64K = 2^{16} = 65\,536$). Tal diccionario, por supuesto, se llena muy rápido en todos los trabajos de compresión —salvo los más pequeños—. El mismo problema existe con LZ78, y algunas de las soluciones utilizadas

I	En dicc.	Nueva entrada	Salida	I	En dicc.	Nueva entrada	Salida
s	S			y	S		
si	N	256-si	115 (s)	y␣	N	274-y␣	121 (y)
i	S			␣	S		
ir	N	257-ir	105 (i)	␣t	N	275-␣t	32 (␣)
r	S			t	S		
r␣	N	258-r␣	114 (r)	te	N	276-te	116 (t)
␣	S			e	S		
␣s	N	259-␣s	32 (␣)	ea	S		
s	S			eas	N	277-eas	263 (ea)
si	S			s	S		
sid	N	260-sid	256 (si)	se	N	278-se	115 (s)
d	S			e	S		
d␣	N	261-d␣	100 (d)	es	N	279-es	101 (e)
␣	S			s	S		
␣e	N	262-␣e	32 (␣)	s␣	N	280-s␣	115 (s)
e	S			␣	S		
ea	N	263-ea	101 (e)	␣s	S		
a	S			␣se	N	281-␣se	259 (␣s)
as	N	264-as	97 (a)	e	S		
s	S			ea	S		
st	N	265-st	115 (s)	ea␣	N	282-ea␣	263 (ea)
t	S			␣	S		
tm	N	266-tm	116 (t)	␣s	S		
m	S			␣si	N	283-␣si	259 (␣s)
ma	N	267-ma	109 (m)	i	S		
a	S			ic	N	284-ic	105 (i)
an	N	268-an	97 (a)	c	S		
n	S			ck	N	285-ck	99 (c)
n␣	N	269-n␣	110 (n)	k	S		
␣	S			k␣	N	286-k␣	107 (k)
␣e	S			␣	S		
␣ea	N	270-␣ea	262 (␣e)	␣s	S		
a	S			␣se	S		
as	S			␣sea	N	287-␣sea	281 (␣se)
asi	N	271-asi	264 (as)	a	S		
i	S			al	N	288-al	97 (a)
il	N	272-il	105 (i)	l	S		
l	S			ls	N	289-ls	108 (l)
ly	N	273-ly	108 (l)	s	Y		
				s, eof	N		115 (s)

Tabla 3.19: Codificación de sir␣sid␣eastman␣easily␣teases␣sea␣sick␣seals.

0	NULL	110	n	262	␣e	276	te
1	SOH	···		263	ea	277	eas
···		115	s	264	as	278	se
32	SP	116	t	265	st	279	es
···		···		266	tm	280	s␣
97	a	121	y	267	ma	281	␣se
98	b	···		268	an	282	ea␣
99	c	255	255	269	n␣	283	␣si
100	d	256	si	270	␣ea	284	ic
101	e	257	ir	271	asi	285	ck
···		258		272	il	286	k␣
107	k	259	␣s	273	ly	287	␣sea
108	l	260	r␣	274	y␣	288	al
109	m	261	d␣	275	␣t	289	ls

Tabla 3.20: Un diccionario LZW.

con LZ78 también son aplicables a LZW. Otro hecho interesante sobre LZW es que las cadenas en el diccionario se convierten continuamente en un sólo carácter. Por lo tanto, toma mucho tiempo explorar completamente las cadenas largas del diccionario, lo que limita la oportunidad de lograr una compresión realmente buena. Podemos decir que LZW se adapta lentamente a sus datos de entrada.

◇ **Ejercicio 3.10 (sol. en pág. 1073):** Utilícese LZW para la codificación de la cadena "alf␣eats␣alfalfa". Muéstrense la salida del codificador y las nuevas entradas añadidas por él al diccionario.

◇ **Ejercicio 3.11 (sol. en pág. 1073):** Analícese la compresión LZW de la cadena "aaaa...".

A dirty icon —Un sucio icono—
(anagrama de "*dictionary* —diccionario—")

3.12.1. Decodificación LZW

Para comprender cómo funciona el decodificador LZW, recordemos las tres etapas que realiza el codificador cada vez que escribe algo en la secuencia de salida: (1) ofrece como salida el puntero del diccionario que apunta a la cadena I, (2) guarda cadena Ix en la siguiente entrada del diccionario disponible, y (3) inicializa la cadena I con el símbolo x.

El decodificador comienza con las primeras entradas de su diccionario inicializado con todos los símbolos del alfabeto (normalmente 256 símbolos). A continuación, lee su cadena de entrada (que consiste en punteros que señalan ítems del diccionario) y utiliza cada puntero para recuperar los símbolos sin comprimir de su diccionario y los escribe en su secuencia de datos de salida. También construye su diccionario del mismo modo que el codificador (este hecho se suele expresar diciendo que el codificador y el decodificador están *sincronizados*, o que trabajan en *lockstep* —al mismo ritmo—).

En la primera etapa de decodificación, el decodificador introduce el primer puntero y lo utiliza para recuperar un ítem del diccionario —I—. Éste es una cadena de símbolos, que se escribe en la salida del decodificador. La cadena Ix necesita ser guardada en el diccionario, pero el símbolo x es aún desconocido; será el primer elemento de la siguiente cadena recuperada del diccionario.

En cada etapa de decodificación posterior a la primera, el decodificador introduce el siguiente puntero, recupera la siguiente cadena — J— del diccionario, la escribe en la salida, aísla su primer

símbolo —`x`—, y guarda la cadena `Ix` en la siguiente entrada del diccionario disponible (después de comprobar —para asegurar— que la cadena `Ix` no está en el diccionario). El decodificador mueve entonces J a I, y está listo para el siguiente paso.

En nuestro ejemplo —"`sir␣sid...`"—, el primer puntero que introduce el decodificador es 115. Esto corresponde a la cadena `s`, que se recupera del diccionario, se almacena en I, y se convierte en el primer ítem escrito en la salida del decodificador. El siguiente puntero es 105, por lo que la cadena `i` se recupera desde J y también se escribe en la salida. El primer símbolo de J se concatena con I, formando la cadena `si` —que no existe en el diccionario— y por consiguiente, se añade en su entrada 256. La variable J se mueve a I, por lo que ahora es la cadena `i`. El siguiente puntero es 114, por lo tanto, recupera la cadena `r` del diccionario en J y también la escribe en la salida. El primer símbolo de J se concatena con I , para formar la cadena `ir` —que no existe en el diccionario— y se añade al mismo como la entrada 257. La variable J se traslada a I, por lo tanto, I es ahora la cadena `r`. El siguiente paso lee el puntero 32, escribe ␣ en la salida, y guarda la cadena r␣.

◇ **Ejercicio 3.12 (sol. en pág. 1074):** Decodifíquese la cadena "`alf␣eats␣alfalfa`" utilizando los resultados de la codificación predecentes del Ejercicio 3.10.

◇ **Ejercicio 3.13 (sol. en pág. 1074):** Supóngase un alfabeto de dos símbolos: `a` y `b`. Muéstrense los primeros pasos para codificar y decodificar la cadena "`ababab...`".

3.12.2. Estructura del diccionario LZW

Hasta ahora, hemos asumido que el diccionario LZW es un array de cadenas de tamaño variable. Para comprender por qué un trie es una estructura de datos mejor para el diccionario necesitamos recordar cómo funciona el codificador. Éste introduce los símbolos y los concatena en una variable `I`, mientras la cadena contenida en `I` se encuentre en el diccionario. En cierto momento el codificador introduce el primer símbolo —`x`— que provoca un fallo en la búsqueda (la cadena `Ix` no está en el diccionario). Luego, agrega `Ix` al diccionario. Esto significa que cada cadena añadida al diccionario le proporciona —en realidad— exactamente un símbolo nuevo: `x`. (Dicho de otra manera: para cada cadena del diccionario de más de un símbolo, no existe una cadena "padre" en el diccionario que sea un símbolo más corta.)

Un árbol similar al utilizado por LZ78 es —por lo tanto— una buena estructura de datos, ya que la agregación de la cadena `Ix` a un árbol de este tipo, se realiza mediante la adición de un nodo con `x`. El principal problema es que cada nodo en el árbol LZW puede tener muchos hijos (es un árbol de caminos múltiples, no un árbol binario). Imaginese que el nodo con la letra `a` está en la entrada 97. Inicialmente no tiene hijos, pero si se añade la cadena `ab` al árbol, el nodo 97 obtiene un hijo. Más tarde, cuando se añade —digamos— la cadena `ae`, el nodo 97 obtiene un segundo hijo, y así sucesivamente. La estructura de datos para el árbol —por lo tanto— debe diseñarse de tal manera que un nodo pueda tener cualquier número de hijos, pero sin necesidad de reservar memoria para ellos de antemano.

Una manera de diseñar tal estructura de datos es alojar el árbol en un array de nodos; cada estructura con dos campos: un símbolo y un puntero al nodo padre. Un nodo no tiene punteros a ningún nodo hijo. El desplazamiento por el árbol, desde un nodo a uno de sus hijos, se realiza mediante un proceso de *hashing* en el que el puntero al nodo y el símbolo del hijo están determinados por un método de dispersión —*hashing*—, para crear un nuevo puntero.

Supongamos que ya se ha introducido la cadena `abc` —símbolo a símbolo—, y se ha almacenado en el árbol en los tres nodos, cuyas direcciones son: 97, 266 y 284. Inmediatamente a continuación, el codificador lee el símbolo siguiente —`d`—; busca la cadena `abcd`, o más específicamente, un nodo que contenga el símbolo `d` cuyo padre está en la posición 284; realiza la operación *hash* sobre 284 (el puntero a la cadena `abc`) y 100 (código ASCII de `d`), para crear un puntero a un nodo —digamos, 299—; y examina el nodo 299. Existen tres posibilidades:

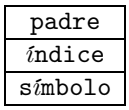
1. El nodo no se utiliza. Esto significa que `abcd` aún no está en el diccionario y debería añadirse al mismo. El codificador lo agrega al árbol, almacenando el puntero padre 284 y el código ASCII 100 en el nodo. El resultado es el siguiente:

Nodo				
Dirección :	97	266	284	299
Contenido :	(− : a)	(97 : b)	(266 : c)	(284 : d)
Cadena representada :	a	ab	abc	abcd

2. El nodo contiene un puntero padre 284 y el código ASCII de `d`. Esto significa que la cadena `abcd` ya está en el árbol. El codificador introduce el símbolo siguiente —digamos `e`—, y busca en el árbol-diccionario la cadena `abcde`.

3. El nodo contiene algo más. Esto significa que otra operación hash de un puntero y un código ASCII ha obtenido como resultado 299; y el nodo 299 ya contiene la información de otra cadena. Esto recibe el nombre de *colisión*, y puede tratarse de varias maneras. La forma más sencilla para hacer frente a una colisión es incrementar el puntero 299 y examinar los nodos 300, 301,... hasta localizar uno que no esté en uso, o hasta encontrar un nodo (284 : d).

En la práctica, construimos nodos cuya estructura consta de tres campos: un puntero a un nodo padre, el puntero (o índice) creado por el proceso de *hashing*, y el código (normalmente ASCII) del símbolo contenido en el nodo. El segundo campo es necesario por las colisiones. Un nodo, por lo tanto, se puede representarse así:



Ilustremos esta estructura de datos utilizando la cadena `abababa` ... del Ejercicio 3.13. El diccionario es un array —`dict`— donde cada entrada es una estructura con los tres campos: `padre`, `indice` y `simbolo`. Nos referimos a un campo, por ejemplo, `dict[puntero].padre`, donde `puntero` es un índice al array. El diccionario se inicializa con las dos entradas: `a` y `b`. (Para mantener la sencillez del ejemplo, utilizaremos códigos ASCII. Suponemos que `a` tiene el código 1 y `b` el código 2.) Los primeros pasos del codificador son los siguientes:

Paso 0: Se marcan todas las posiciones del diccionario —a partir de la 3— como no utilizadas:

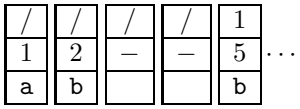
/	/	/	/	/	...
1	2	−	−	−	
a	b				

Paso 1: Se introduce el primer símbolo `a` en la variable `I`. (Lo que realmente se introduce es el código de `a` —que en nuestro ejemplo es 1 —, por consiguiente, `I = 1`.) Puesto que éste es el primer símbolo, el codificador asume que está en el diccionario y, por lo tanto, no realiza ninguna búsqueda.

Paso 2: Se introduce el segundo símbolo —`b`— en `J`, por consiguiente, `J = 2`. El codificador tiene que buscar la cadena `ab` en el diccionario; ejecuta: `puntero := hash(I, J)`. Vamos a suponer que el resultado es 5. El campo `dict[puntero].indice` contiene "no utilizado", ya que la ubicación 5 aún está vacía, por lo que la cadena `ab` no está en el diccionario. Se agrega ejecutando:

```
dict[puntero].padre := I;
dict[puntero].indice := puntero;
dict[puntero].simbolo := J;
```

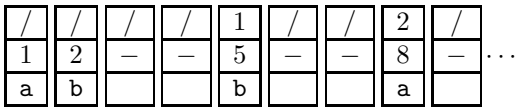
siendo `puntero` = 5. `J` se mueve hasta `I` (ahora, `I` = 2):



Paso 3: Se introduce el tercer símbolo —`a`— en `J`, por consiguiente, `J` = 1. El codificador tiene que buscar la cadena `ba` en el diccionario. Realiza la operación de *hashing*: `puntero` := `hash` (`I`, `J`). Vamos a suponer que el resultado es 8. El campo `dict` [`puntero`] .`indice` contiene "no utilizado", por lo que la cadena `ba` no está en el diccionario. Se añade —como antes— ejecutando:

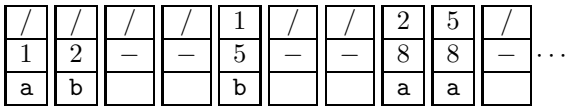
```
dict [puntero] .padre := I;
dict [puntero] .indice := puntero;
dict [puntero] .simbolo := J;
```

siendo `puntero` = 8. `J` se mueve hasta `I` (ahora, `I` = 1):



Paso 4: Se introduce el cuarto símbolo —`b`— en `J`, por consiguiente, `J` = 2. El codificador tiene que buscar la cadena `ab` en el diccionario. Realiza la operación: `puntero` := `hash` (`I`, `J`). Sabemos por el paso 2, que el resultado es 5. El campo `dict` [`puntero`] .`indice` contiene 5, por lo que la cadena `ab` está en el diccionario. El valor de `puntero` se mueve a `I` (ahora, `I` = 5).

Paso 5: Se introduce el quinto símbolo —`a`— en `J`, por consiguiente, `J` = 1. El codificador tiene que buscar la cadena `aba` en el diccionario. Ejecuta, como de costumbre: `puntero` := `hash` (`I`, `J`). Vamos a suponer que el resultado es 8 (una colisión). El campo `dict` [`puntero`] .`indice` contiene 8, que parece bueno, pero el campo `dict` [`puntero`] .`padre` contiene 2 —en vez del esperado 5— por lo que la función hash sabe que se trata de una colisión y que la cadena `aba` no está en la entrada 8 del diccionario. Incrementa `puntero` tantas veces como sea necesario hasta encontrar una entrada de diccionario con `indice` = 8 y `padre` = 5, o hasta que encuentra una entrada sin usar. En el primer caso, la cadena `aba` se encuentra en el diccionario, y `puntero` se mueve a `I`. En el último caso, `aba` no está en el diccionario; el codificador lo guarda en la entrada apuntada por `puntero`, y mueve `J` a `I`:



Ejemplo: A continuación se muestran los 15 pasos de hashing para la codificación de la cadena `alf`␣`eats`␣`alfalfa`. El proceso de codificación está ilustrado en sí mismo —en detalle— en la respuesta al Ejercicio 3.10. Los resultados de las operaciones hash son arbitrarios; no son los resultados obtenidos por una función hash real. Los 12 nodos trie construidos para esta cadena se muestran en las Figuras 3.22-I y 3.22-II. (Hay que leer ambas figuras simultáneamente, por filas. A la izquierda de cada fila se hace referencia a las acciones que se indican a continuación):

1. `Hash` (`l`, `97`) → `278`. La posición 278 del array se establece en (`97`, `278`, `l`).
2. `Hash` (`f`, `108`) → `266`. La posición 266 del array se establece en (`108`, `266`, `f`).
3. `Hash` (␣, `102`) → `269`. La posición 269 del array se establece en (`102`, `269`,␣).

4. `Hash(e, 32)` → `267`. La posición 267 del array se establece en (`32`, `267`, `e`).

5. `Hash(a, 101)` → `265`. La posición 265 del array se establece en (`101`, `265`, `a`).

6. `Hash(t, 97)` → `272`. La posición 272 del array se establece en (`97`, `272`, `t`).

7. `Hash(s, 116)` → `265`. ¡Una colisión! Salta a la siguiente posición disponible —268— y la establece en (`116`, `265`, `s`). Ésta es la razón por la que el índice necesita ser almacenado.

8. `Hash(␣, 115)` → `270`. La posición 270 del array se establece en (`115`, `270`,`␣`).

9. `Hash(a, 32)` → `268`. ¡Una colisión! Salta a la siguiente posición disponible —271— y la establece en (`32`, `268`, `a`).

10. `Hash(l, 97)` → `278`. La posición 278 del array ya contiene el índice 278 y el símbolo `l` desde el paso 1, así que no hay necesidad de almacenar nada o añadir una entrada trie nueva.

11. `Hash(f, 278)` → `276`. La posición 276 del array se establece en (`278`, `276`, `f`).

12. `Hash(a, 102)` → `274`. La posición 274 del array se establece en (`102`, `274`, `a`).

13. `Hash(l, 97)` → `278`. La posición 278 del array ya contiene el índice 278 y el símbolo `l` desde el paso 1, así que no hay necesidad de hacer nada.

14. `Hash(f, 278)` → `276`. La posición 276 del array ya contiene el índice 276 y el símbolo `f` desde el paso 11, así que no hay necesidad de hacer nada.

15. `Hash(a, 276)` → `274`. ¡Una colisión! Salta a la siguiente posición disponible —275— y la establece en (`276`, `274`, `a`).

A los lectores que hayan seguido cuidadosamente esta discusión hasta ahora, les agradará saber que el uso del diccionario-árbol en array del decodificador LZW es sencillo y no necesita operaciones de hashing. El decodificador comienza —al igual que el codificador— inicializando las primeras 256 posiciones del array. A continuación, lee los punteros de la secuencia de datos de entrada y utiliza cada uno de ellos para localizar un símbolo en el diccionario.

En la primera etapa de la decodificación, el decodificador introduce el primer puntero y lo utiliza para recuperar un ítem de diccionario —`I`—. Éste es un símbolo que escribe el decodificador en su secuencia de salida. La cadena `Ix` necesita ser salvada en el diccionario, pero el símbolo `x` es aún desconocido; éste será el primer símbolo de la siguiente cadena recuperada del diccionario.

En cada etapa de decodificación posterior a la primera, el decodificador introduce el siguiente puntero y lo utiliza para recuperar la siguiente cadena —`J`— del diccionario, y la escribe en la secuencia de salida. Si el puntero es —digamos, 8— el decodificador examina el campo `dict[8].indice`. Si este campo es igual a 8, entonces es el nodo correcto. De lo contrario, el decodificador analiza posiciones consecutivas del array, hasta encontrar la primera correcta.

Una vez que se localiza el nodo adecuado del árbol, se utiliza el campo `padre` para volver a ascender por el árbol y recuperar los símbolos individuales de la cadena *en orden inverso*. Los símbolos se colocan entonces en `J` en el orden correcto (véase más abajo); el decodificador aísla el primer símbolo —`x`— de `J`, y guarda la cadena `Ix` en la siguiente ubicación disponible en el array. (La cadena `I` se encontró en la etapa anterior, por lo que sólo necesita añadir un nodo —el que tiene el símbolo `x`—.) Después, el decodificador desplaza `J` a `I`, quedando listo para la siguiente etapa.

La recuperación de una cadena completa del árbol LZW implica, por lo tanto, seguir los punteros de los campos `padre`. Esto equivale a ascender por el árbol, por lo que la función hash ya no es necesaria.

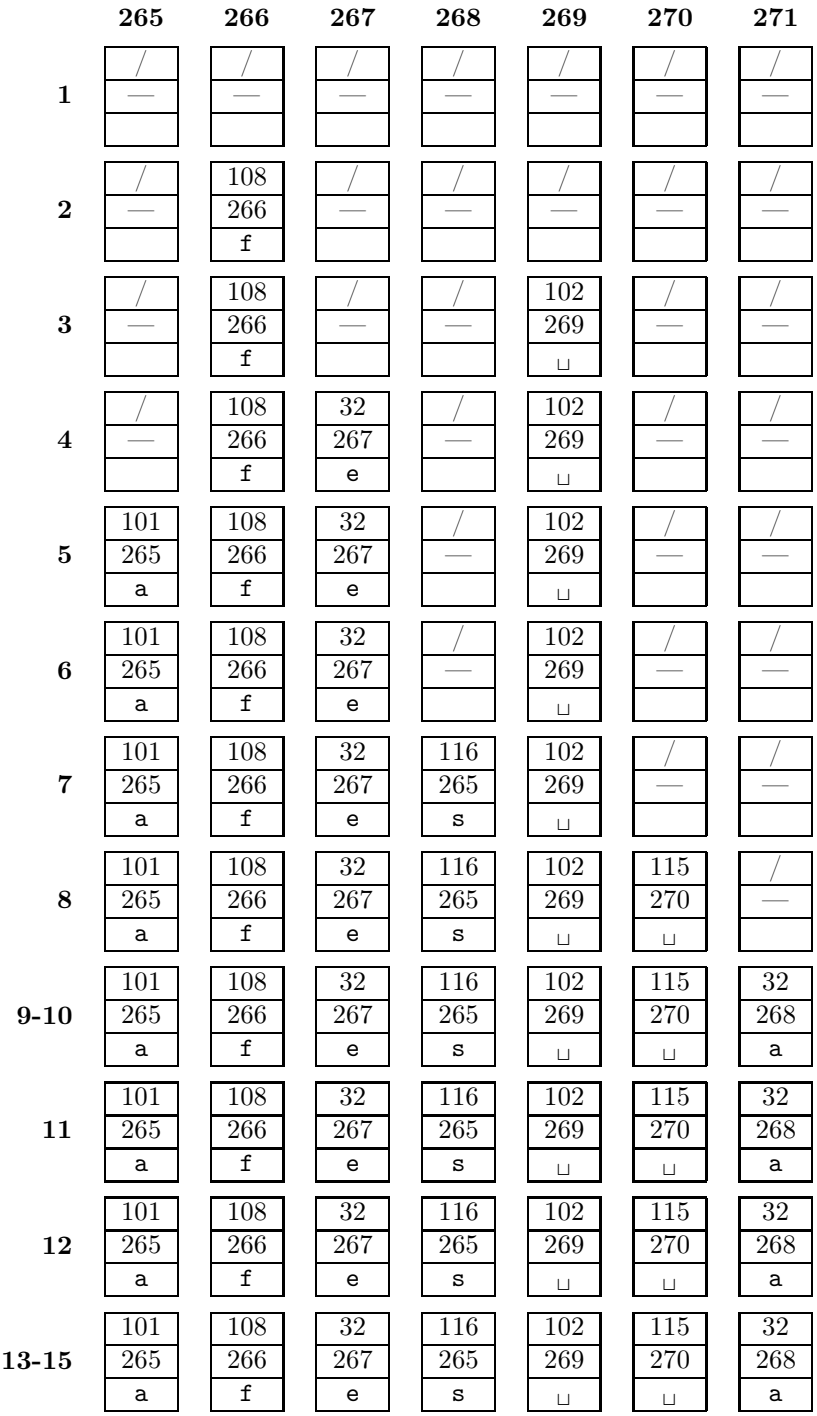


Figura 3.22: Crecimiento de un Trie LZW para "alf␣eats␣alfalfa". (I)

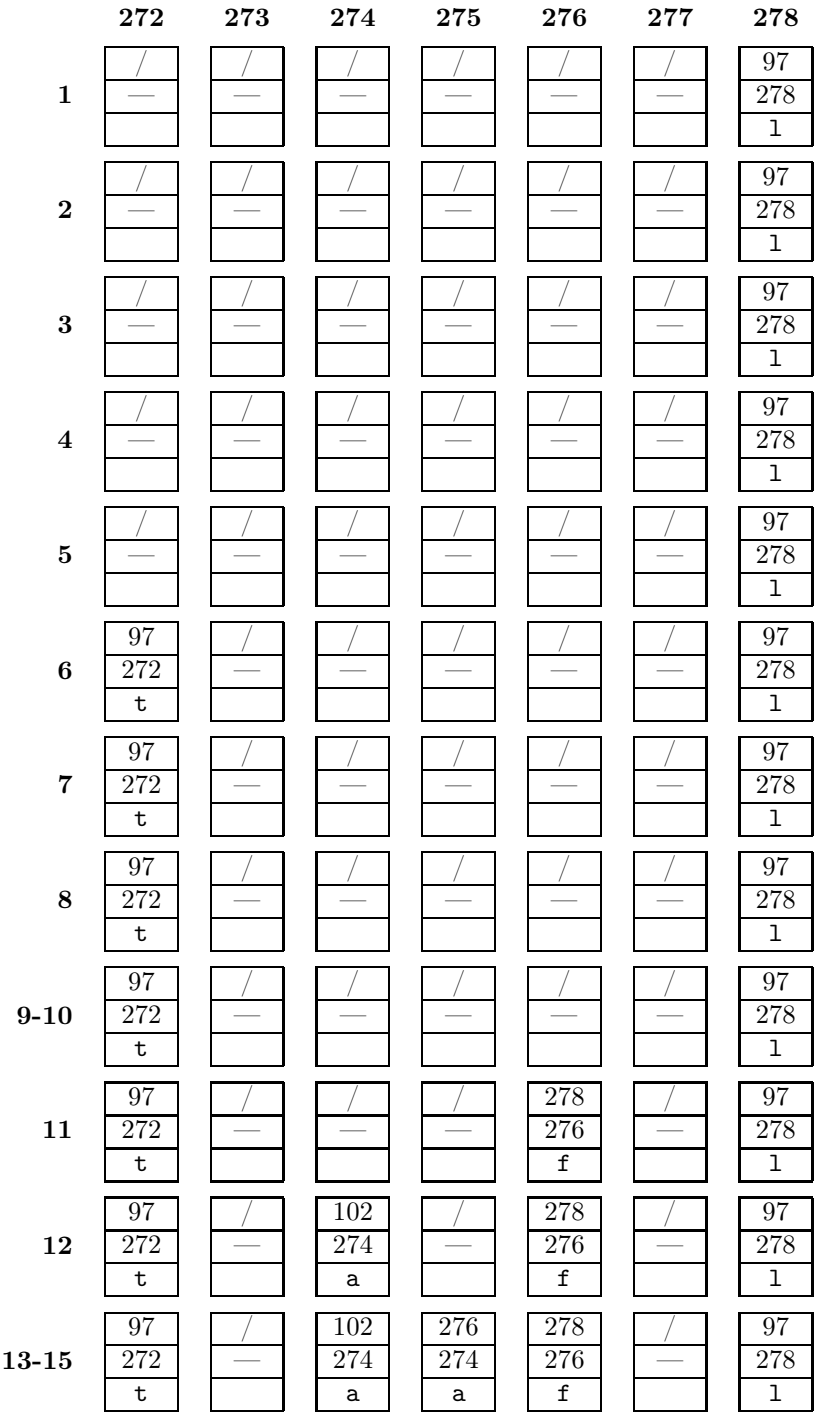


Figura 3.22: Crecimiento de un Trie LZW para "alf␣eats␣alfalfa". (II)

Ejemplo: El ejemplo anterior describe los 15 pasos hash en la codificación de la cadena `alf␣eats␣alfalfa`. El último paso establece la ubicación 275 del array en (276, 274, a) y escribe 275 (un puntero a la posición 275) en la secuencia de datos comprimidos. Cuando el decodificador lee esta secuencia, el puntero 275 es el último ítem introducido y procesado por él. El decodificador encuentra un símbolo en el campo `simbolo` de la posición 275 (indicando que la cadena almacenada en la ubicación 275 termina con una `a`) y un puntero en la posición 276 en el campo `padre`. El decodificador examina la ubicación 276, donde se encuentra el símbolo `f` y el puntero padre —278—. En la posición 278 el decodificador encuentra el símbolo `l` y un puntero al nodo 97. Finalmente, en la posición 97, el decodificador encuentra un símbolo `a` y un puntero nulo. La cadena —invertida— es, por lo tanto, `afla`. El decodificador no necesita hacer ninguna operación hash o utilizar los campos `indice`.

El último punto a discutir es la inversión de cadena. A continuación indicamos dos métodos utilizados corrientemente:

1. Utilizando una pila. Una pila es una estructura de datos común en los ordenadores modernos. Se trata de un array en la memoria, al que se accede sólo por un extremo. En todo momento, el último ítem que se ha introducido en la pila, es el primero que sale (Last-In-First-Out[16], o LIFO). Los símbolos obtenidos del diccionario se introducen en la pila. Cuando el último ha entrado en la misma, ésta se vacía —símbolo a símbolo— en la variable `J`. Cuando el proceso de vaciado termina, toda la cadena se ha invertido. Esta es una forma muy común para invertir una cadena.

2. Recuperando los símbolos del diccionario y concatenándolos en `J` —*de derecha a izquierda*—. Al terminar, la cadena queda almacenada en `J` en el orden correcto. La variable `J` debe ser lo suficientemente amplia para dar cabida a la cadena más larga posible; al menos tanto como cuando se utiliza una pila.

◇ **Ejercicio 3.14 (sol. en pág. 1074):** ¿Cuál es la cadena más larga que se puede recuperar del diccionario LZW durante la decodificación?

(Un recordatorio. El problema de las patentes del software y las licencias se discute en Sección 3.30.)

3.12.3. LZW en la práctica

La publicación del algoritmo LZW —en 1984— ha afectado decisivamente a la comunidad sobre compresión de datos y ha influido en muchas personas a la hora de crear implementaciones y variantes de este método. Aquí se describen algunas de las variantes LZW más importantes y subproductos derivados del mismo (*spin-offs*).

3.12.4. Diferenciación

El concepto de diferenciación —o codificación relativa— ya se ha mencionado en la Sección 1.3.1. Esta idea resulta útil para la compresión de imágenes LZW, ya que la mayoría de los píxeles adyacentes no difieren en mucho. Es posible implementar un codificador LZW que calcule el valor relativo de un píxel con respecto a su predecesor, y luego codifique esta diferencia. El decodificador tiene —por supuesto— que ser compatible, y debe calcular el valor absoluto de un píxel después de la decodificación de su valor relativo.

[16] El último en entrar, es el primero en salir.

Incrementa la cadena en	**Añade una cadena al diccionario** *por frase*	*por carácter de entrada*
Un carácter :	LZW	LZY
Varios caracteres:	LZMW	LZAP

Tabla 3.23: LZW y tres variantes.

3.12.5. Variantes LZW

En la Sección 8.6.2 se describe una variante de LZW basada en palabras.

LZW es un método de compresión de datos adaptativo; pero es lento para adaptarse a su entrada, ya que cada cadena del diccionario consta sólo de un carácter. El Ejercicio 3.11 muestra que una cadena de un millón de `a`es (la cual, por supuesto, tiene una redundancia muy alta) produce frases de diccionario, la más larga de las cuales contiene sólo 1 414 `a`es.

El método LZMW —Sección 3.13— es una variante de LZW que soluciona este problema. Su principio fundamental es el siguiente: En lugar de añadir I más un carácter de la siguiente frase del diccionario, añade I más la frase siguiente completa al diccionario.

El método LZAP —Sección 3.14— es otra variante basada en esta idea: En lugar de concatenar sólo las dos últimas frases y almacenar el resultado en el diccionario, guarda todos los prefijos de la concatenación en el diccionario. Más específicamente, si `S` y `T` son los dos últimos caracteres emparejados, añade `St` al diccionario para cada prefijo no vacío —`t` o `T`—.

La Tabla 3.23 resume los principios de LZW, LZMW y LZAP, y muestra cómo sugieren de forma natural otra variante: LZY.

LZW añade una cadena de diccionario por frase e incrementa la longitud de las cadenas en un símbolo cada vez. LZMW añade una cadena de diccionario por frase e incrementa las cadenas en varios símbolos cada vez. LZAP añade una cadena de diccionario por símbolo de entrada e incrementa las cadenas varios símbolos cada vez. LZY —Sección 3.15— se ajusta a la cuarta celda de la Tabla 3.23. Este método que agrega una cadena de diccionario por símbolo de entrada e incrementa las cadenas un símbolo cada vez.

3.13. LZMW

Esta variante de LZW, desarrollado por V. Miller y M. Wegman (véase [Miller y Wegman 85]), está basada en dos principios:

1. Cuando el diccionario está lleno, se elimina la frase del mismo utilizada menos recientemente. Hay varias formas de seleccionar esta frase; los desarrolladores indicaron que cualquier manera razonable de hacerlo funcionaría. Una posibilidad es identificar todas las frases `S` del diccionario para las cuáles no hay frases `Sa` (nada se ha añadido a `S`, lo que implica que `S` no se ha utilizado desde que empezó a formar parte en el diccionario) y eliminar la más antigua de ellas. En este caso, debe construirse y mantenerse una estructura de datos auxiliar, que apunta a las frases diccionario según su antigüedad (el primer puntero siempre apunta a la frase más vieja). Las primeras 256 frases del diccionario nunca deben ser eliminadas.

2. Cada frase añadida al diccionario es una concatenación de dos cadenas: el emparejamiento anterior —$\mathtt{S}'$ previo— y el actual —`S`—. Esto contrasta con LZW, donde cada frase añadida es la concatenación del emparejamiento actual y el primer símbolo de la siguiente cadena localizada. El siguiente algoritmo en pseudocódigo ilustra esto:

```
Inicializar Dicc a todos los símbolos del alfabeto A;
i:= 1;
S':= null;
Mientras i <= tamaño de la cadena de entrada
    k:= emparejamiento más largo de Entrada[i] de Dicc;
    Salida(k);
    S:= Frase k de Dicc;
    i:= i + longitud(S);
    Si frase S'S no está en Dicc, añadirla a Dicc;
    S':= S;
finMientras;
```

Mediante la adición de la concatenación `S′S` al diccionario LZMW, las frases del diccionario pueden crecer más de un símbolo a la vez. Esto significa que las frases diccionario LZMW son las unidades de la entrada más "naturales" (por ejemplo, si la entrada es texto en lenguaje natural, las frases del diccionario tienden a ser palabras completas, o incluso varias palabras en ese lenguaje). Al mismo tiempo, esto implica que el diccionario LZMW general se adapta a la entrada más rápido que el diccionario LZW.

La Tabla 3.24 ilustra el método LZMW aplicado a la cadena:

`sir␣sid␣eastman␣easily␣teases␣sea␣sick␣seals`.

LZMW se adapta a su entrada más rápido que LZW, pero tiene las tres desventajas siguientes:

1. La estructura del diccionario de datos no puede ser el sencillo trie LZW, porque no todos los prefijos de una frase del diccionario están incluidos en el diccionario. Esto significa que el método utilizado en LZW consistente en la búsqueda de un símbolo cada vez, no va a funcionar. En cambio, cuando se añade al diccionario LZMW una frase `S` , cada prefijo de `S` debe agregarse a la estructura de datos, y cada nodo de la estructura de datos debe tener una etiqueta que indique si el nodo está en el diccionario o no.

2. Encontrar la cadena más larga puede requerir el método de *backtracking* —en el que se retorna hacia atrás para reanudar la búsqueda por otro camino—. Si el diccionario contiene `aaaa` y `aaaaaaaa`, tenemos que alcanzar el octavo símbolo de la frase `aaaaaaab` para darnos cuenta de la necesidad de elegir la frase más corta. Esto implica que las búsquedas en el diccionario LZMW son más lentas que en el LZW. Este problema no se aplica al decodificador LZMW.

3. Una frase puede ser añadida al diccionario dos veces. De nuevo, esto complica la elección de estructura de datos para el diccionario.

◇ **Ejercicio 3.15 (sol. en pág. 1075):** Utilícese el método LZMW para comprimir la cadena `swiss␣miss`.

◇ **Ejercicio 3.16 (sol. en pág. 1075):** Comprímase la cadena `yabbadabbadabbadoo` utilizando LZMW.

3.14. LZAP

LZAP es una extensión de LZMW. "AP" significa "All Prefixes[17]" [Storer 88]. LZAP se adapta a su entrada rápidamente —como LZMW— pero elimina la necesidad del backtracking, una característica

[17] Todos los prefijos.

Paso	Entrada	Salida	S	Añadir al dicc.	S′
	sir␣sid␣eastman␣easily␣teases␣sea␣sick␣seals				
1	s	115	s	—	s
2	i	105	i	256-si	i
3	r	114	r	257-ir	r
4	␣	32	␣	258-r␣	␣
5	si	256	si	259-␣si	si
6	d	100	d	260-sid	d
7	␣	32	␣	261-d␣	␣
8	e	101	e	262-␣e	e
9	a	97	a	263-ea	a
10	s	115	s	264-as	s
11	t	116	t	265-st	t
12	m	109	m	266-tm	m
13	a	97	a	267-ma	a
14	n	110	n	268-an	n
15	␣e	262	␣e	269-n␣e	␣e
16	as	264	as	270-␣eas	as
17	i	105	i	271-asi	i
18	l	108	l	272-il	l
19	y	121	y	273-ly	y
20	␣	32	␣	274-y␣	␣
21	t	116	t	275-␣t	t
22	ea	263	ea	276-tea	ea
23	s	115	s	277-eas	s
24	e	101	e	278-se	e
25	s	115	s	279-es	s
26	␣	32	␣	280-s␣	␣
27	se	278	se	281-␣se	se
28	a	97	a	282-sea	a
29	␣si	259	␣si	283-a␣si	␣si
30	c	99	c	284-␣sic	c
31	k	107	k	285-ck	k
32	␣se	281	␣se	286-k␣se	␣se
33	a	97	a	287-␣sea	a
34	l	108	l	288-al	l
35	s	115	s	289-ls	s

Tabla 3.24: Ejemplo de LZMW.

Paso	Entrada	*Match*	Añadir al diccionario
	yabbadabbadabbadoo		
1	y	y	—
2	a	a	256-ya
3	b	b	257-ab
4	b	b	258-bb
5	a	a	259-ba
6	d	d	260-ad
7	ab	ab	261-da, 262-dab
8	ba	ba	263-abb, 264-abba
9	dab	dab	265-bad, 266-bada, 267-badab
10	bad	bad	268-dabb, 269-dabba, 270-dabbad
11	o	o	271-bado
12	o	o	272-oo

Tabla 3.25: Ejemplo de LZAP.

que lo hace más rápido que LZMW. El principio es el siguiente: En lugar de añadir la concatenación de las dos últimas frases —S′S— al diccionario, añade todas las cadenas S′t donde t es un prefijo de S (incluyendo S mismo). En consecuencia, si S′ = a y S = bcd, agrega las frases ab, abc y abcd al diccionario LZAP. La Tabla 3.25 muestra los emparejamientos —*matches*— y las frases añadidas al diccionario para la cadena yabbadabbadabbadoo.

En el paso 7, el codificador concatena d con los dos prefijos de ab y añade las dos frases —da y dab— al diccionario. En el paso 9, se concatena ba con los tres prefijos de dab y añade las tres frases resultantes —bad, bada y badab— al diccionario.

LZAP agrega más frases a su diccionario que LZMW, por lo que necesita más bits para representar la posición de una frase. Al mismo tiempo, LZAP proporciona una mayor selección de frases de diccionario para comparar con la cadena de entrada, por lo que la compresión es algo mejor que la de LZMW, además de ser más rápido (debido a la sencilla estructura de datos del diccionario, lo que elimina la necesidad del *backtracking*). Este tipo de compensación entre ventajas y desventajas es común en los algoritmos informáticos.

"¡Cielos, Andrew!", dijo su esposa, "¿qué es un *rustler*?"
No se encontraba en ningún diccionario, y las traducciones actuales del mismo eran inconsistentes. Un hombre en Hoosic Falls, dijo que él había pasado por Cheyenne, y oído el término aplicado de manera complementaria a personas vivas y emprendedoras. Otro hombre había siempre supuesto que hacía referencia a una especie de caballo. Sin embargo, la versión más alarmante de todas era que un *rustler* era un ladrón de ganado.

—Owen Wister, *The Virginian— A Horseman of the Plains*[a]

[a]Un jinete de las Llanuras.

3.15. LZY

El método LZY se debe a Dan Bernstein. La Y es por Yabba, que procede de la cadena de entrada originalmente utilizada para probar el algoritmo. El diccionario LZY se inicializa con todos

los símbolos individuales del alfabeto. Por cada símbolo C de la secuencia de entrada, el decodificador busca la cadena más larga –P— que precede a C y que ya está incluida en el diccionario. Si la cadena PC no está en el diccionario, ésta se añade al mismo como una nueva frase.

A modo de ejemplo, la entrada yabbadabbadabbadoo genera las frases ya, ab, bb, ba, ad, da, abb, bba, bad, ada, dab, abba, bbad, bado, ado, do y oo que se añaden al diccionario.

Mientras codifica los datos de entrada, el codificador realiza un seguimiento de la lista —L— de coincidencias hasta el momento actual (*matches-so-far*). Inicialmente, L está vacía. Si C es el símbolo actual de la secuencia de entrada, el codificador (antes de añadir nada al diccionario) comprueba —para cada cadena M en L— si la cadena MC está en el diccionario. Si es así, entonces MC se convierte en un nuevo elemento de L. De lo contrario, el codificador genera como salida el número de L (su posición en el diccionario) y agrega C, como un nuevo *match-so-far*, a L.

A continuación ofrecemos un algoritmo en pseudocódigo para construir el diccionario LZY. La experiencia personal del autor indica que la implementación de este tipo de algoritmo en una programación real, lleva a una comprensión más profunda de su funcionamiento:

```
Comenzar con un diccionario que contiene todos los símbolos del alfabeto, cada uno asignado
a un número entero único.
M: = cadena vacía.
Repetir
    Añadir el símbolo siguiente C a la cadena de entrada de M.
    Si M no está en el diccionario, agregarlo al mismo; eliminar
    el primer carácter de M, y repetir este paso.
Hasta fin de datos de entrada.
```

La salida de LZY no está sincronizada con las adiciones al diccionario. Además, el codificador debe tener cuidado para que el emparejamiento de salida más largo no se superponga a sí mismo. A causa de esto, el diccionario debe constar de dos partes: S y T, donde sólo se utiliza la primera para la salida. El algoritmo es el siguiente:

```
Comenzar con S, asignando a cada carácter individual un entero único;
Establecer T vacío; M vacío; y O vacío.
Repetir
    Introducir el siguiente símbolo C.
    Si OC se encuentra en S,
        actualizar O := OC;
    de lo contrario,
        sacar S(O), actualizar O := C, añadir T a S, y vaciar T.
    Mientras MC no esté en S ó T,
        añadir MC a T (asignándolo al siguiente entero disponible),
        y eliminar el primer carácter de M.
    Después de que M sea lo suficientemente corto para que MC quepa
    en el diccionario, establecer M := MC.
Hasta fin de datos de entrada.
Sacar S(O) y salir.
```

El decodificador lee la secuencia de datos comprimidos. Utiliza cada código para encontrar una frase en el diccionario; saca la frase como una cadena, y luego utiliza cada símbolo de la cadena para añadir una nueva frase al diccionario —tal y como lo hacía el codificador—. Estos son los pasos de la decodificación:

```
Comenzar con un diccionario que contiene todos los símbolos del alfabeto, cada uno asignado
a un número entero único.
M := cadena vacía.
Repetir
```

```
    Leer D(O) de la entrada,
    y tomar la inversa en D para encontrar O.
    Mientras O no sea la cadena vacía,
        encontrar el primer carácter C de O,
        y actualizar (D,M) como antes.
        También sacar C y eliminarlo del inicio de O.
Hasta fin de datos de entrada.
```

Téngase en cuenta que la codificación requiere de dos operaciones rápidas en las cadenas del diccionario: (1) comprobar si la cadena `SC` está en el diccionario, cuando la posición de `S` es conocida y (2) encontrar la posición de `S`, dada la posición de `CS`. La decodificación requiere las mismas operaciones, más una búsqueda rápida para encontrar el primer carácter de una cadena cuando se conoce su posición en el diccionario.

La Tabla 3.26 ilustra el algoritmo LZY para la cadena de entrada "`abcabcabcabcabcabcx`". Muestra las frases añadidas al diccionario en cada etapa, así como la lista de emparejamientos actuales.

Paso	Entrada	Añadir al diccionario.	Emparejamientos actuales
	abcabcabcabcabcabcx		
1	a	—	a
2	b	256-ab	b
3	c	257-bc	c
4	a	258-ca	a
5	b	—	ab, b
6	c	259-abc	bc, c
7	a	260-bca	ca, a
8	b	261-cab	ab, b
9	c	—	abc, bc, c
10	a	262-abca	bca, ca, a
11	b	263-bcab	cab, ab, b
12	c	264-cabc	abc, bc, c
13	a	—	abca, bca, ca, a
14	b	265-abcab	bcab, cab, ab, b
15	c	266-bcabc	cabc, abc, bc, c
16	a	267-cabca	abca, bca, ca, a
17	b	—	abcab, bcab, cab, ab, b
18	c	268-abcabc	bcabc, cabc, abc, bc, c
19	a	269-bcabca	cabca, abca, bca, ca, a
20	b	270-cabcab	abcab, bcab, cab, ab, b
21	c	—	abcabc, bcabc, cabc, abc, bc, c
22	x	271-abcabcx	x
23		272-bcabcx	
24		273-cabcx	
25		274-abcx	
26		275-bcx	
27		276-cx	

Tabla 3.26: Ejemplo de LZY.

El codificador comienza sin elementos emparejados. Cuando se introduce un símbolo, lo añade

a cada elemento disponible en los emparejamientos realizados hasta el momento actual; cualquier resultado que ya esté en el diccionario se convierte la nueva lista de emparejamientos actuales (el símbolo en sí mismo se convierte en otro elemento de esta lista). Cualquier resultado que no esté en el diccionario, se elimina de la lista y se añade al diccionario.

Antes de leer el quinto `c` —por ejemplo— el conjunto de emparejamientos actuales es: `bcab`, `cab`, `ab`, y `b`. El codificador agrega `c` a cada elemento. No se encuentra coincidencia para `bcabc`, por lo que el codificador lo agrega al diccionario. El resto aún está en el diccionario, por lo que la nueva lista emparejamientos actuales es: `cabc`, `abc`, `bc`, y `c`.

Cuando se introduce la `x`, la lista de emparejamientos actuales es: `abcabc`, `bcabc`, `cabc`, `abc`, `bc` y `c`. Ninguno de los elementos —`abcabcx`, `bcabcx`, `cabcx`, `abcx`, `bcx` y `cx`— está en el diccionario, por lo que todos ellos se añaden al mismo, y la lista de emparejamientos actuales se reduce a sólo un elemento: `x`.

Airman stops coed Anagrama de “data compression”

3.16. LZP

LZP es una variante LZ77 desarrolada por Charles Bloom [Bloom 96] (La P se refiere a “predicción”). Se basa en el principio de la predicción de contexto, que dice: “si una cadena determinada `abcde` ha aparecido en la secuencia de entrada en el pasado e iba seguida por `fg...`, entonces, cuando `abcde` aparece de nuevo en la secuencia de datos de entrada, hay una alta probabilidad de que aparezca seguida por el mismo `fg...`”. Debería consultarse la Sección 3.31 para observar la relación entre los algoritmos basados en un diccionario y los de predicción.

La Figura 3.27-I, muestra un buffer de desplazamiento LZ77 con `fgh...` como símbolos actuales (esta cadena es denotada por S) en el buffer de preanálisis, precedido inmediatamente por `abcde` en el buffer de búsqueda. La cadena `abcde` se llama *contexto* de `fgh...` y se denota por C. En general, el contexto de una cadena S está formado por los N símbolos de la cadena C inmediatamente a la izquierda de S. Un contexto puede ser de cualquier longitud N, y las variantes de LZP, discutidas en las Secciones 3.16.3 y 3.16.4, utilizan diferentes valores de N. El algoritmo pasa el contexto a través de una función hash y utiliza el resultado —H— como un índice para una tabla de punteros llamada *tabla índice*.

La tabla índice, contiene puneros a varios símbolos del buffer de búsqueda. El índice H se usa para seleccionar un puntero P. En un caso típico, P apunta a una cadena vista anteriormente cuyo contexto también es `abcde` (véanse más abajo los casos atípicos). El algoritmo, pues, realiza los siguientes pasos:

- *Paso 1*: Guarda P y lo reemplaza en la tabla índice con un nuevo puntero Q que apunta a `fgh...` en el buffer de preanálisis (Figura 3.27-II). Se inicializa a cero una variable entera L. Ésta se utiliza más adelante para indicar la longitud de la cadena emparejada.

- *Paso 2*: Si `P` no es un puntero nulo, el algoritmo lo sigue y compara la cadena apuntada por `P` (“`fgi...`”, en el búfer de búsqueda) con la cadena “`fgh...`” —en el buffer de preanálisis—. Sólo hay dos símbolos iguales en nuestro ejemplo, por lo que la longitud de los elementos emparejados —L— se establece en 2.

- *Paso 3*: Si $L = 0$ (ningún símbolo tiene correspondencia), el buffer se desplaza hacia la derecha (o, equivalente, la entrada se desplaza a la izquierda) **una posición** y el primer símbolo de la cadena `S` —la `f`— se escribe en la secuencia de datos comprimidos como un código ASCII puro (un literal).

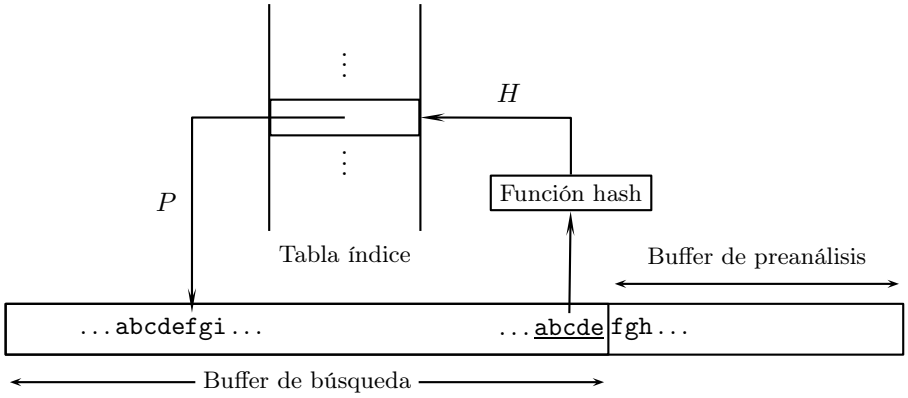


Figura 3.27: El principio de LZP. Parte I

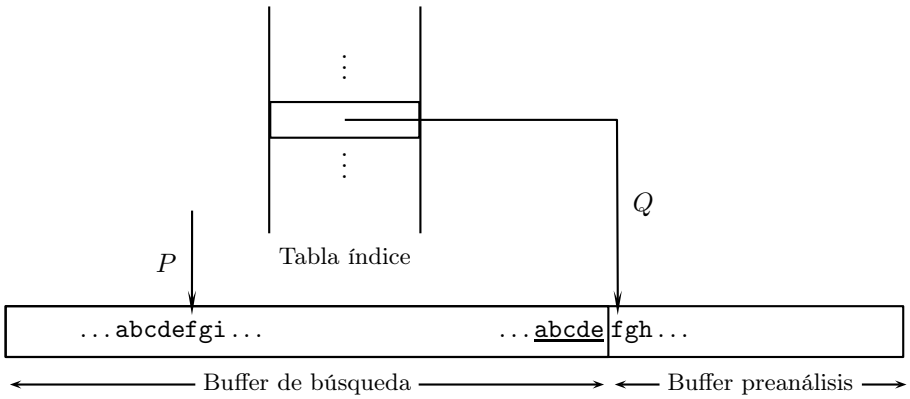


Figura 3.27: El principio de LZP. Parte II

- *Paso 4*: Si $L > 0$ (L símbolos han sido emparejados), el buffer se desliza L posiciones a la derecha y el valor de L se escribe en la secuencia de datos comprimidos (después de haber sido debidamente codificado).

En nuestro ejemplo, el único valor codificado $L = 2$ pasa a formar parte de la secuencia comprimida, representando a los dos símbolos `fg`, y es este paso el que produce la compresión. Es evidente que a mayor valor de L, mayor será la compresión. Se producen grandes valores de L cuando un contexto de `C` —de N símbolos— en los datos de entrada es seguido por la misma cadena larga —S—, que precedía a una ocurrencia previa de `C`. Esto puede ocurrir cuando la secuencia de entrada cuenta con una alta redundancia. Con un flujo de datos de entrada aleatorios, cada vez que aparezca el mismo contexto `C`, es probable que se encuentre seguido por otra S —llevando a $L = 0$— y por lo tanto, ninguna compresión. En "promedio", las cadenas de entrada producen más literales que L valores escritos en la secuencia de salida (véase también el Ejercicio 3.18).

El decodificador introduce los datos comprimidos ítem a ítem y crea la salida descomprimida en un buffer `B`. Los pasos son los siguientes:

- *Paso 1*: Introduce el siguiente ítem —`I`— desde la cadena comprimida.
- *Paso 2*: Si es un código ASCII puro (el literal `a`), se añade al buffer `B`, y desplaza los datos de `B` a la izquierda una posición.
- *Paso 3*: Si `I` es una longitud de emparejamiento codificada, se decodifica, para obtener L. El contexto actual `C` (los N símbolos de más a la derecha, en `B`) se pasa a una función hash para obtener un índice `H`, que se utiliza para seleccionar un puntero `P` de la tabla de índices. El decodificador copia la cadena de L símbolos de que comienza en `B[P]` y la añade al extremo derecho de `B`. También desplaza los datos en `B` a la izquierda —L posiciones— y sustituye `P` en la tabla de índices con un nuevo puntero, para mantenerse en sintonía con el codificador.

Quedan por discutir dos puntos antes de que estemos listos para ver un ejemplo detallado:

1. Cuando el codificador comienza, coloca los N primeros símbolos de la cadena de entrada en el buffer de búsqueda, para formar el primer contexto. A continuación, escribe estos símbolos —como literales— en la secuencia de datos comprimidos. Esta es la única medida especial que se necesita para iniciar la compresión. El decodificador comienza leyendo los primeros N ítems de los datos comprimidos (que deben ser literales), y los coloca en el extremo derecho del buffer `B`, para servir como primer contexto.
2. Se ha mencionado antes que en el caso típico, `P` apunta a una cadena vista previamente cuyo contexto es idéntico al contexto actual —`C`—. En un caso atípico, `P` puede apuntar a una cadena cuyo contexto es diferente. El algoritmo, sin embargo, no revisa el contexto y siempre se comporta de la misma manera. Simplemente intenta hacer coincidir el mayor número símbolos como sea posible. En el peor de los casos, no se logra emparejar ningún símbolo, dando lugar a la escritura de un literal en el flujo de datos comprimidos.

3.16.1. Ejemplo

Para ilustrar el funcionamiento del codificador LZP, suponemos que la cadena de entrada es: `xyabcabcabxy`. Para simplificar el ejemplo, utilizaremos $N = 2$.

1. Para comenzar la operación, el codificador desplaza los dos primeros símbolos —`xy`— en el buffer de búsqueda y los saca como literales. También inicializa todas las posiciones de la tabla de índices a *puntero nulo*.

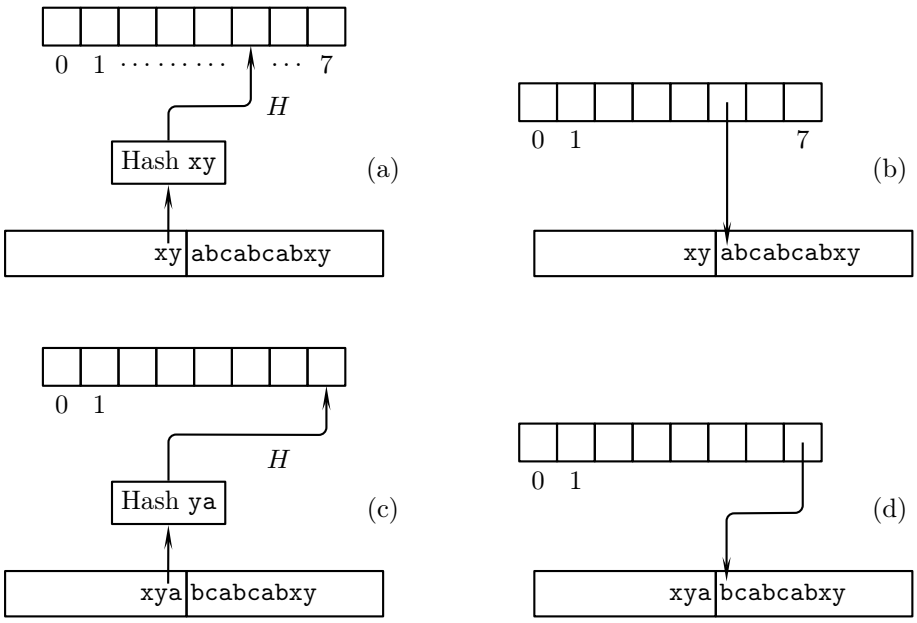


Figura 3.28: Compresión LZP de `xyabcabcabxy`. Parte I.

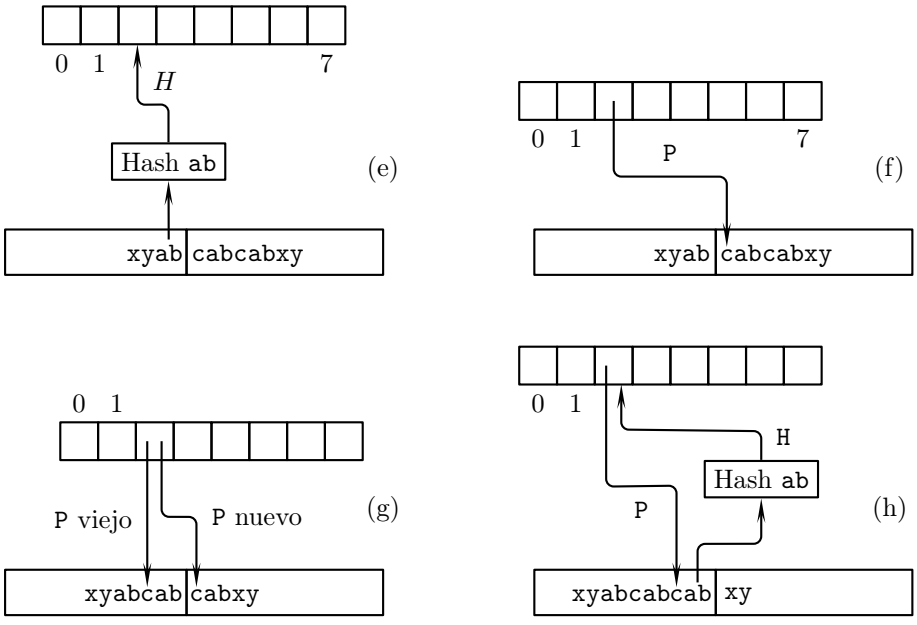


Figura 3.28: (Continuación): Compresión LZP de `xyabcabcabxy`. Parte II.

2. El símbolo actual es `a` (la primera `a`), y el contexto es `xy`. Éste se pasa a la función hash, que genera un valor —digamos, 5—; pero la posición 5 de la tabla de índices contiene un puntero nulo, por lo que `P` es nulo (Figura 3.28a-I). Se actualiza el contenido de la ubicación 5 de manera que apunte a la primera `a` (Figura 3.28b-I), el cual sale entonces como un literal. Los datos en el buffer del codificador se desplazan hacia la izquierda.

3. El símbolo actual es la primera `b`, y el contexto es `ya`. Éste se pasa a la función hash, que genera un valor —digamos, 7—, pero la posición 7 de la tabla de índices contiene un puntero nulo, por lo que `P` es nulo (Figura 3.28c-I). Se actualiza el contenido de la ubicación 7 para que apunte a la primera `b` (Figura 3.28d-I), la cual sale entonces como un literal. Los datos en el buffer del codificador se desplazan hacia la izquierda.

4. El símbolo actual es la primera `c`, y el contexto es `ab`. Éste se pasa a la función hash, que genera un valor —digamos, 2—, pero la posición 2 de la tabla de índices contiene un puntero nulo, por lo que `P` es nulo (Figura 3.28e-II). Se actualiza el contenido de la ubicación 2 para que apunte a la primera `c` (Figura 3.28f-II), la cual sale entonces como un literal. Los datos en el buffer del codificador se desplazan hacia la izquierda.

5. Lo mismo ocurre dos veces más, escribiendo los literales `a` y `b` en la secuencia de datos comprimidos. El símbolo actual es ahora —la segunda— `c`, y el contexto es `ab`. Éste se envía a la función hash, que —como en el paso 4— produce el valor 2, por lo que `P` apunta a "`cabc`...". La ubicación 2 se actualiza para que señale al símbolo actual (Figura 3.28g-II), y el codificador intenta emparejar las cadenas `cabcabxy` y `cabxy`. El número de caracteres iguales en aspecto y posición en ambas cadenas es de $L = 3$. El número 3 se escribe en la salida —codificado— y los datos se desplazan tres posiciones hacia la izquierda.

6. El símbolo actual es la segunda `x`, y el contexto es `ab`. Éste se envía a la función hash, que genera el valor 2; pero la ubicación de 2 en la tabla de índices apunta a la segunda `c` (Figura 3.28h-II). La posición 2 se actualiza para apuntar al símbolo actual, y el codificador intenta emparejar las cadenas `cabxy` y `xy`. El resultado de la operación es, por supuesto, ninguna coincidencia ($L = 0$), por lo que el codificador escribe `x` en la secuencia de datos comprimidos como un literal y desplaza los datos una posición.

7. El símbolo actual es la segunda `y`, y el contexto es `bx`. Éste se envía a la función hash, que genera un valor —digamos, 7—. Esto es una colisión hash, ya que el contexto `ya` tenía asignada mediante la función hash la posición 7 —en el paso 3—; pero el algoritmo no comprueba las colisiones. Continúa como de costumbre. La ubicación 7 de la tabla de índices apunta a la primera `b` (o más bien a la cadena `bcabcabxy`). Se actualiza de forma que señale al símbolo actual, y el codificador intenta emparejar las cadenas `bcabcabxy` e `y`, lo que produce $L = 0$. El codificador escribe `y` en la secuencia de datos comprimidos como un literal y desplaza los datos una posición.

8. El símbolo actual es el carácter *fin-de-datos*, por lo que el algoritmo termina.

◇ **Ejercicio 3.17 (sol. en pág. 1075):** Escríbanse los pasos de la codificación LZP de la cadena de entrada `xyaaaa`...

> Las estructuras son las armas de los matemáticos.
>
> —Nicolás Bourbaki.

3.16.2. Consideraciones prácticas

El desplazamiento de los datos en el buffer requiere la actualización de todos los punteros en la tabla de índices. Una implementación eficiente, por tanto, debería adoptar una mejor solución. A continuación se describen dos enfoques, pero otros pueden funcionar también:

1. Se reserva de un buffer lo más grande posible, y se comprime la secuencia de entrada en bloques. Cada bloque se introduce en el buufer y nunca se desplaza. En cambio, se mueve un puntero de izquierda a derecha, para apuntar en todo momento al símbolo actual en el buffer. Cuando se ha codificado el buffer completo, éste se llena de nuevo con datos de entrada recientes. Este enfoque simplifica el programa, pero tiene la desventaja de que los datos del final de un bloque no pueden utilizarse para predecir algo del bloque siguiente. Cada bloque se codifica de manera independiente de los otros, lo que produce una compresión más pobre.

2. Se reserva un buffer de gran tamaño, y se utiliza como una cola circular (Sección 3.3.1). Los propios datos no se mueven, pero se simula su desplazamiento eficazmente mediante la actualización de los punteros inicio y fin —después de la codificación del símbolo actual—, y se introduce y almacena un nuevo símbolo en el buffer. El algoritmo es algo más complicado, pero este enfoque tiene la ventaja de que todos los datos entrada se codifican como una secuencia. Cada símbolo se beneficia de los D símbolos que le preceden (donde D es la longitud total del buffer).

Imaginese un puntero `P` en la tabla de índices que apunta a un símbolo `X` en el buffer. Cuando el movimiento de los dos punteros en la cola circular saca `X` fuera de la cola, algunos símbolos nuevos —`Y`— se introducen en la posición ocupada por `X`, y `P` ahora apunta a `Y`. Cuando `P` es seleccionado de nuevo por la función hash y se utiliza para el emparejamiento de las cadenas, el resultado del mismo probablemente producirá $L = 0$. Sin embargo, el algoritmo siempre reemplaza el puntero que se utiliza, por lo que este caso no debería afectar al rendimiento del algoritmo de manera significativa.

3.16.3. LZP1 y LZP2

Actualmente existen cuatro versiones de LZP, llamadas LZP1 a LZP4. En esta sección se discuten los detalles de los dos primeros. El contexto utilizado por LZP1 es de orden 3, i.e., son los tres bytes que preceden al actual. La función hash produce un índice `H` de 12 bits y se describe mejor mediante el siguiente código en C:

$$\texttt{H} = \big((\texttt{C} >> \texttt{11})^{\wedge}\ \texttt{C}\big)\ \&\texttt{0xFFF}.$$

Puesto que `H` es de 12 bits, la tabla de índices debe tener un tamaño de $2^{12} = 4096$ entradas. Cada entrada es de dos bytes (16 bits), pero sólo se utilizan 14 de los 16 bits. En consecuencia, un puntero `P` seleccionado en la tabla de índices apunta a un buffer de tamaño $2^{14} = 16K$.

El codificador LZP1 crea una cadena comprimida compuesta por una mezcla de literales y valores de L. Por lo tanto, cada ítem debe ir precedido por un *flag* que indica su naturaleza. Ya que sólo se necesita discernir entre dos posibilidades, la opción más sencilla es elegir flags de 1 bit. Sin embargo, ya hemos mencionado que un "promedio" de los datos de entrada produce más literales que L valores, así que tiene sentido asignar un flag corto (menor que un bit) para indicar un literal, y un flag largo (un poquito más que un bit) para indicar una longitud. El esquema utilizado por LZP1 utiliza 1 para indicar dos literales, 01 para indicar un literal seguido de una longitud de emparejamiento, y 00 para indicar una longitud de emparejamiento.

◇ **Ejercicio 3.18 (sol. en pág. 1076):** Sea T la probabilidad de aparición de un literal en la secuencia de datos comprimidos. ¿Para qué valor de T el esquema anterior produce —en promedio— flags de 1 bit de longitud?

Un literal se escribe en la cadena comprimida como un código ASCII de 8 bits. Las longitudes de emparejamiento (*match*) se codifican en concordancia con la Tabla 3.29. Inicialmente, los códigos son

Longitud	Código	Longitud	Código
1	00	11	11 | 111 | 0000 0
2	01	12	11 | 111 | 0000 1
3	10	⋮	
4	11 | 000	41	11 | 111 | 1111 0
5	11 | 001	42	11 | 111 | 1111 1 | 0000 0000
6	11 | 010	⋮	
⋮		296	11 | 111 | 1111 1 | 1111 1110
10	11 | 110	297	11 | 111 | 1111 1 | 1111 1111 | 0000 0000

Tabla 3.29: Códigos usados por LZP1 y LZP2 para longitudes de *match*.

de 2 bits. Cuando todos ellos se han utilizado, se añaden 3 bits más —lo que hace un total de 5 bits— donde los 2 primeros bits son unos. Cuando éstos también se han utilizado, se añaden 5 bits más —lo que produce códigos de 10 bits— donde los primeros 5 bits son unos. A partir de entonces, se añade otro grupo de 8 bits al código cada vez que todos los códigos anteriores han sido utilizados. Observe que el patrón "todos unos" nunca se usa como código y está reservado para indicar códigos cada vez más largos. Nótese también, que un código unario o un código unario general (Sección 2.3.1) podría haber sido una mejor opción.

La secuencia de datos comprimidos consta de una mezcla de literales (bytes en código ASCII) y bytes de control —que contienen flags y longitudes codificadas—. Esto se ilustra en la salida del ejemplo de la Sección 3.16.1. La entrada de este ejemplo es la cadena `xyabcabcabxy`, y los elementos de salida son `x`, `y`, `a`, `b`, `c`, `a`, `b`, 3, `x` e `y`. La secuencia de salida real consiste en el único byte de control 111 01 | 10 1 seguido por nueve bytes con los códigos ASCII de `x`, `y`, `a`, `b`, `c`, `a`, `b`, `x` e `y`.

◇ **Ejercicio 3.19 (sol. en pág. 1076):** Explíquese el contenido del byte de control 111 01 | 10 1.

Otro ejemplo de cadena comprimida son los tres literales —`x`, `y`, y `a`— seguidos por las cuatro longitudes de emparejamiento 12, 12, 12 y 10. En primer lugar, preparamos los flags:

$$1\,(\texttt{x},\texttt{y})\;01\,(\texttt{a},12)\;00\,(12)\;00\,(12)\;00\,(12)\;00\,(10)\,,$$

luego sustituimos los códigos de 12 y 10:

$$1\texttt{xy}\,01\texttt{a}11|111|00001\,00|11|111|00001\,00|11|111|00001\,00|11|111|00001\,00|11|110,$$

y finalmente agrupamos los bits que forman los bytes de control. El resultado es:

$$10111111\,\texttt{x},\texttt{y},\texttt{a},\,00001001\,11110000\,10011111\,00001001\,11110000\,10011110.$$

Nótese que al primer byte de control le siguen tres literales.

El último punto que debe mencionarse es el caso $\ldots 0\,1yyyyyyy\,zzzzzzzz$. El primer byte de control termina con el 0 de un par 01, y el segundo byte comienza con el 1 del mismo par. Esto indica un literal seguido de una longitud de emparejamiento. La longitud de emparejamiento está formado por yyy bits (o por lo menos algunos de ellos) del segundo byte de control. Si el código de esa longitud es largo, entonces los bits zzz —o algunos de ellos— pueden formar parte del código. El literal es, o bien el byte zzz, o bien el byte siguiente.

LZP2 es idéntico a LZP1 salvo que los literales se codifican utilizando códigos de Huffman no adaptativos. Idealmente, sólo son necesarias dos pasadas; la primera cuenta la frecuencia de aparición de los símbolos en la secuencia de datos de entrada y la segunda realiza la compresión real. Entre ambas pasadas, se construye la tabla de códigos de Huffman.

3.16.4. LZP3 y LZP4

LZP3 es similar a LZP1 y a LZP2. Utiliza contextos de orden 4 y códigos de Huffman más sofisticados para codificar tanto las longitudes de emparejamiento como los literales. La función hash LZP3 es:

$$\texttt{H} = \big((\texttt{C} >> 15)^{\wedge}\, \texttt{C}\big)\ \&\texttt{0xFFFF},$$

por lo que `H` es un índice de 16 bits, que se va a utilizar con una tabla de índices de tamaño $2^{16} = 64K$. Además de los `P` punteros, la tabla de índices contiene también el `C` contextos. Por consiguiente, si el resultado proporcionado por la función hash —al pasarle un contexto `C`— es un índice `H`, el codificador espera encontrar el mismo contexto `C` en la ubicación `H` de la tabla de índices. Esto se conoce como *confirmación de contexto*. Si el codificador encuentra algo más, o si encuentra un puntero nulo, establece `P` a nulo y pasa a un contexto de orden 3. Si la confirmación del contexto de orden 3 también falla, el algoritmo pasa a un contexto de orden 2, y si eso también falla, el algoritmo establece `P` a nulo y escribe un literal en la secuencia de datos comprimidos. Este método intenta encontrar el contexto de mayor orden visto hasta el momento actual.

LZP4 utiliza contextos de orden 5 y un proceso de varios pasos de búsqueda. En el paso 1, los cuatro bytes de más a la derecha —`I`— del contexto, se pasan a la función hash para crear un índice `H` de 16 bits acorde con la siguiente expresión:

$$\texttt{H} = \big((\texttt{I} >> 15)^{\wedge}\, \texttt{I}\big)\ \&\texttt{0xFFFF}.$$

Entonces `H` se utiliza como un índice para la tabla de índices que tiene $64K$ entradas, cada una correspondiente a un valor de `H`. Cada entrada apunta al principio de una lista que une los nodos que tienen el mismo valor hash. Supongamos que los contextos `abcde`, `xbcde` y `mnopq` proporcionan el mismo índice —`H` = 13— al pasarlos por la función hash (es decir, la función hash calcula el mismo índice —13— cuando se aplica a `bcde` y a `nopq`) y que estamos buscando el contexto `xbcde`. La ubicación 13 de la tabla de índices apunta a una lista con los nodos de estos contextos (y posiblemente otros que también han obtenido el mismo valor —13— al pasarlos por la función hash). Se recorre la lista hasta encontrar un nodo con `bcde`. Este nodo apunta a una segunda lista que vincula los elementos `a` y `x`, y tal vez otros símbolos que preceden la secuencia `bcde`. La segunda lista se recorre hasta encontrar un nodo con `x`. Este nodo contiene un puntero a la aparición más reciente de `xbcde` en el buffer de búsqueda. Si no se localiza ningún nodo conteniendo `x`, se escribe un literal en la secuencia de datos comprimidos.

LZP4 utiliza este complejo procedimiento de búsqueda porque un contexto de 5 bytes no completa holgadamente una única palabra en la mayoría de los ordenadores actuales.

3.17. Buscador de repetición

Todos los métodos basados en diccionarios descritos hasta ahora, tienen una cosa en común: el uso un buffer de memoria de gran tamaño como un diccionario que mantiene los fragmentos de texto encontrados hasta el momento. El diccionario se utiliza para localizar las cadenas de símbolos que se repiten. El método aquí descrito es diferente. En lugar de un diccionario, emplea un array —matriz o arreglo— de enteros de tamaño fijo para encontrar apariciones previas de las cadenas de texto. El tamaño del array es igual al cuadrado del tamaño del alfabeto, por lo que no es muy grande. El método se debe a Hidetoshi Yokoo [Yokoo 91], que no eligió llamarlo LZHY, sino que lo dejó sin nombre. El no haber utilizado un nombre de la forma LZxx se debe a que el método no emplea el diccionario Ziv-Lempel tradicional. La razón por la que se quedó sin nombre, es que no comprime muy bien y por lo tanto, debe considerarse el primer paso en un nuevo campo de investigación, más que un método maduro y práctico.

	A	B	C	D	E	...	X	Y	Z
A		3							
B			4						
C				5					
D					6				
E								7	
⋮									
X	2								
Y	8								
Z									

	A	B	C	D	E	...	X	Y	Z
A		9							
B			10						
C				11					
D					12				
E									13
⋮									
X	2								
Y	8								
Z									

Tabla 3.30: (a) REP cuando $i = 8$. (b) REP cuando $i = 13$.

El método alterna entre dos modos: normal y de repetición. Se inicia en el modo normal, donde introduce los símbolos y los codifica —utilizando el método de Huffman adaptativo—. Cuando identifica una repetición de cadena, cambia al modo "de repetición", en el que produce un símbolo `escape`, seguido por la longitud de la cadena repetida.

Supongamos que la secuencia de datos de entrada se compone de los símbolos $x_1x_2\ldots$, pertenecientes a un alfabeto A. Ambos —codificador y decodificador— mantienen un array REP de dimensiones $|A| \times |A|$que se inicializa a todo ceros. Para cada símbolo de entrada x_i, el codificador (y el decodificador) calcula un valor y_i en concordancia con $y_i = i - \texttt{REP}\,[x_{i-1}, x_i]$, y luego actualiza $\texttt{REP}\,[x_{i-1}, x_i] := i$. La cadena de 13 símbolos:

x_i :	X	A	B	C	D	E	Y	A	B	C	D	E	Z
i :	1		3		5		7		9		11		13

genera los siguientes valores para y:

$i =$	1	2	3	4	5	6	7	8	9	10	11	12	13
$y_i =$	1	2	3	4	5	6	7	8	6	6	6	6	13
$x_{i-1}x_i$:		XA	AB	BC	CD	DE	EY	YA	AB	BC	CD	DE	EZ

La Tabla 3.30a muestra el estado del array REP después de la introducción codificación del octavo símbolo. La Tabla 3.30b muestra el estado de REP después haber introducido y codificado los 13 símbolos.

Quizás una mejor manera de explicar los valores que va tomando y es calcularlos con la siguiente expresión:

$$y_i = \begin{cases} 1, & \text{para } i = 1, \\ i, & \text{para } i > 1 \text{ y la primera aparición de } x_{i-1}x_i, \\ \min(k) & \text{para } i > 1 \text{ y } x_{i-1}x_i \text{ idéntico a } x_{i-k-1}x_{i-k}. \end{cases}$$

Esto demuestra que y es: o bien i, o bien la distancia k entre la cadena actual —$x_{i-1}x_i$— y la copia más reciente —$x_{i-k}x_{i-k}$—. Sin embargo, el reconocimiento de una repetición de una cadena se realiza por medio del array REP y sin utilizar ningún diccionario (que es la cualidad principal de este método).

Cuando una cadena de longitud l se repite en la entrada, se generan l valores consecutivos idénticos de y; ésta es la señal destinada al codificador para que pase al modo "de repetición". Siempre y cuando se generen valores diferentes y consecutivos de y, el codificador permanece en el modo "normal", en el que codifica x_i en Huffman adaptativo y lo saca. Cuando el codificador detecte que $y_{i+1} = y_i$, saca x_i

en modo normal, y luego entra en el modo "de repetición". En el ejemplo anterior, esto sucede cuando $i = 9$, por lo que la cadena $XABCDEYAB$ se saca en modo normal.

Una vez en el modo "de repetición", el codificador introduce más símbolos, y calcula y valores hasta que encuentra el primero que difiere de y_i. En nuestro ejemplo, esto sucede en $i = 13$, cuando se introduce la `Z`. El codificador comprime la cadena "`CDE`" (que corresponde a $i = 10, 11, 12$) en el modo "de repetición" mediante la emisión de un símbolo de cambio de código (codificado), seguido por la longitud (codificada) de la cadena repetida (3 en nuestro ejemplo). Entonces, el codificador vuelve al modo normal, donde se guarda el valor de y la `Z` como y_i, e introduce el siguiente símbolo.

El símbolo de cambio de código debe ser un símbolo adicional, que no está incluido en el alfabeto A. Nótese que en todo momento sólo se necesitan guardar dos valores de y —y_{i-1} e y_i—. Además, el método no es muy eficiente, ya que detecta la repetición de la cadena "demasiado tarde" y codifica las dos primeras repitiendo los símbolos en el modo normal. En nuestro ejemplo, sólo tres de los cinco símbolos repetidos se codifican en el modo "de repetición".

El decodificador introduce y decodifica los primeros nueve símbolos —formando la cadena `XABCDEYAB`— mientras actualiza el array `REP` y calcula los valores de y. Cuando el símbolo de cambio de código es introducido, los valores respectivos de i e y_i son: 9 y 6. El decodificador introduce y decodifica la longitud —3—, y ahora tiene que averiguar la cadena repetida de longitud 3 con sólo los datos disponibles en el array `REP` —sin tener en cuenta cualquiera de las entradas previamente decodificadas—. Como $i = 9$ e y_i es la distancia entre esta cadena y su copia, el decodificador sabe que la copia comienza en la posición $i - y_i = 9 - 6 = 3$ de la entrada. Analiza `REP`, en busca de un 3. Lo localiza en la posición `REP` [`A`, `B`], por lo que empieza a buscar un 4 en la fila `B` de `REP`. Lo encuentra en `REP` [`B`, `C`], por lo que el primer símbolo de la cadena buscada es `C`. Buscando un 5 en la fila `C`, el decodificador lo localiza en `REP` [`C`, `D`], por lo que el segundo símbolo es `D`. Buscando ahora un 6 en la fila `D`, el decodificador lo encuentra en `REP` [`D`, `E`].

Así es cómo una cadena repetida puede ser decodificada sin mantener un diccionario.

Ambos —codificador y decodificador— almacenan los valores de i en `REP`, por lo que una entrada de `REP` debe tener, por lo menos, dos bytes de longitud. De esta manera i puede tomar valores de hasta $64\text{K} - 1 \approx 65\,500$, por lo que la entrada ha de ser codificada en bloques de 64K. Para un alfabeto de 256 símbolos, el tamaño de `REP`, por lo tanto, debe ser de $256 \times 256 \times 2 = 128$ Kbytes — no muy grande—. Para alfabetos mayores, `REP` puede necesitar ser incómodamente grande.

En el modo normal, símbolos (como el de cambio de código) se codifican utilizando el método de Huffman adaptativo (Sección 2.9). En el modo de repetición, las longitudes se codifican en un código prefijo recursivo denominado $Q_k(i)$, donde k es un entero positivo (véase la Sección 2.3 para los códigos prefijos). Suponiendo que i es un entero cuya representación binaria es 1α, el prefijo $Q_k(i)$ de i se define como:

$$Q_0(i) = 1^{|\alpha|}0\alpha, \qquad Q_k(i) = \begin{cases} 0, & i = 1, \\ 1Q_{k-1}(i-1), & i > 1, \end{cases}$$

donde $|\alpha|$ es la longitud de α, y $1^{|\alpha|}$ es una cadena de $|\alpha|$ unos. La Tabla 3.31 muestra algunos de los códigos propuestos; sin embargo, ninguno de los códigos prefijo de la Sección 2.3 puede utilizarse en sustitución de los códigos $Q_k(i)$ aquí propuestos.

El desarrollador de este método, Hidetoshi Yokoo, indica que el rendimiento de la compresión no es muy sensible al valor exacto de k, y propone $k = 2$ para obtener un mejor rendimiento total.

Como se mencionó anteriormente, el método no es muy eficiente, por lo que debe ser considerado el comienzo de un nuevo campo de investigación, donde se identifican las cadenas repetidas sin la necesidad de un gran diccionario.

i	α	$Q_0(i)$	$Q_1(i)$	$Q_2(i)$
1	*nulo*	0	0	0
2	0	100	10	10
3	1	101	1100	110
4	00	11000	1101	11100
5	01	11001	111000	11101
6	10	11010	111001	1111000
7	11	11011	111010	1111001
8	000	1110000	111011	1111010
9	001	1110001	11110000	1111011

Tabla 3.31: Código prefijo propuesto.

3.18. Compresión UNIX

En el amplio mundo de UNIX, se utiliza *compress* como la utilidad de compresión más común (a pesar de que GNU `gzip` se ha vuelto más popular, ya que está libre de solicitudes de patentes, es más rápido, y ofrece una mejor compresión). Esta utilidad (también llamada LZC) utiliza LZW con un diccionario creciente. Comienza con un pequeño diccionario de sólo $2^9 = 512$ entradas (con las primeras 256 de ellas ya llenas). Mientras está utilizando este diccionario, escribe punteros de 9 bits como datos de salida. Cuando el diccionario inicial se llena, su tamaño se duplica —hasta 1024 entradas— y a partir de entonces, utiliza punteros de 10 bits. Este proceso continúa hasta que el tamaño del puntero llega a un máximo establecido por el usuario (puede variar entre 9 y 16 bits, con 16 como valor por defecto). Cuando el diccionario más grande permitido se llena, el programa sigue sin cambiarlo (y en ese momento se convierte en estático), pero controlando la razón de compresión. Si ésta es inferior a un umbral predefinido, el diccionario es eliminado, y se inicia uno nuevo de 512 entradas. De esta manera, el diccionario nunca está "demasiado anticuado".

La decodificación se realiza mediante el comando *uncompress*, que implementa el decodificador LZC. Su tarea principal es mantener el diccionario en sintonía con el codificador.

A continuación se comentan dos mejoras de LZC, propuestas por [Horspool 91]:

1. La codificación de los punteros del diccionario con los *códigos binarios por etapas*[18] de la Sección 2.9.1. Así, si el tamaño del diccionario es de $2^9 = 512$ entradas, los punteros pueden ser codificados en 8 ó 9 bits.

2. La determinación de todas las cadenas imposibles en todo momento. Supongamos que la cadena actual en el buffer de lectura anticipada es "`abcd`..." y que el diccionario contiene las cadenas `abc` y `abca` —pero no `abcd`—. El codificador sacará, en este caso, el puntero que señala a la cadena `abc` e iniciará la codificación de una nueva cadena que comienza con `d`. La cuestión es, que después de la decodificación de `abc`, el decodificador sabe que la siguiente cadena no puede empezar con `a` (si así lo hiciera, se habría codificado una cadena `abca`, en lugar de `abc`). En general, si S es la cadena actual, entonces la siguiente cadena no puede comenzar con cualquier símbolo x que satisfaga que "Sx esté en el diccionario".

Este conocimiento puede ser utilizado —tanto por el codificador como por el decodificador— para reducir la redundancia aún más. Cuando un puntero a una cadena ha de salir, tiene que ser codificado, y el método utilizado para asignar el código debe eliminar todas las cadenas que se sabe que son imposibles en ese momento. Esto puede producir un código algo más corto, pero probablemente demasiado complejo como para justificar su uso en la práctica.

[18] Phased-in binary codes.

3.19. Imágenes GIF

GIF —el **f**ormato para el **i**ntercambio de **g**ráficos— fue desarrollado por los Servicios de Información de CompuServe en 1987 como un eficiente *archivo de gráficos comprimido*, que permite que las imágenes puedan ser enviadas entre distintos ordenadores. La versión original de GIF se conoce como GIF 87a. El estándar actual es GIF 89a y —en el momento de la escritura de este texto— puede obtenerse gratuitamente en el archivo: `http://www.w3.org/Graphics/GIF/spec-gif89a.txt`. GIF no es un método de compresión de datos; es un formato de archivo de gráficos que utiliza una variante de LZW para comprimir los datos gráficos (véase [Blackstock 87 y 89]). En esta sección se analizan sólo desde el punto de vista de la compresión de datos GIF.

En la compresión de datos, GIF es muy similar a `compress` y utiliza un diccionario de crecimiento dinámico. Se inicia con el número de bits por pixel —b— como un parámetro. Para una imagen monocromática, $b = 2$; para una imagen con 256 colores o tonos de gris, $b = 8$. El diccionario comienza con 2^{b+1} entradas y se duplica en tamaño cada vez que se llena, hasta que alcanza un tamaño de $2^{12} = 4096$ entradas, donde permanece estática. En este punto, el codificador controla la razón de compresión y puede decidir descartar el diccionario en cualquier momento y comenzar uno nuevo, vacío. Cuando toma esta decisión, el codificador emite el valor 2^b como código de borrado (*clear code*), que es la señal dirigida al decodificador para que éste descarte su diccionario.

Los punteros, que aumentan en un bit de un diccionario a otro, se agrupan y se emiten en bloques de bytes de 8 bits. Cada bloque está precedido por un encabezado que contiene el tamaño de los bloques (255 bytes como máximo) y termina con un byte de ocho ceros. El último bloque contiene —justo antes del terminador— el valor `eof`, que es $2^b + 1$. Una característica interesante es que los punteros se almacenan con su bit menos significativo a la izquierda. Consideremos, por ejemplo, los siguientes punteros de 3 bits: 3, 7, 4, 1, 6, 2 y 5. Sus valores binarios son: 011, 111, 100, 001, 110, 010 y 101, por lo que se empaquetan en 3 bytes: | 10101001 | 11000011 | 11110 . . . |.

El formato GIF es hoy de uso común en los navegadores web, pero no es un eficiente compresor de imágenes. GIF explora la imagen fila a fila, por lo que descubre correlaciones entre píxeles dentro de una fila, pero no entre ellas. Podemos decir que la compresión GIF es ineficiente, porque es unidimensional, mientras que una imagen es bidimensional. Un ejemplo ilustrativo son las dos sencillas imágenes de la Figura 4.3a,b (Sección 4.1). Cuando se guardan ambas en formato GIF89 producen tamaños de archivo de 1053 y 1527 bytes, respectivamente.

La mayoría de los formatos de archivos de gráficos utilizan algún tipo de compresión. Se puede obtener más información sobre estos archivos en la bibliografía: [Murray y vanRyper 94].

3.20. RAR y WinRAR

El popular software RAR es la creación de Eugene Roshal, el cual lo comenzó como su tesis doctoral universitaria. RAR es un acrónimo que procede de Roshal ARchive (o Roshal ARchiver)[19]. El desarrollador actual es el hermano de Eugenio: Alexander Roshal. Lo siguiente es una lista de sus características más importantes:

- RAR se encuentra disponible en [rarlab 06], como shareware, para Windows (WinRAR), Pocket PC, Macintosh OS X, Linux, DOS y FreeBSD. WinRAR tiene una interfaz gráfica de usuario, mientras que las otras versiones sólo admiten una interfaz de línea de comandos.

- WinRAR proporciona soporte completo para archivos RAR y ZIP y puede descomprimirlos (pero no comprimir) archivos CAB, ARJ, LZH, TAR, GZip, ACE, UUE, BZ2, JAR, ISO, 7-Zip y Z.

[19] ARchivo Roshal o ARchivador Roshal.

- Además de la compresión de datos, WinRAR puede encriptar los datos con el estándar de encriptación avanzada (AES-128).

- El tamaño de un archivo RAR está limitado a $2^{33} - 1 = 8\,589\,934\,591$ GB ($2^{33} \times 1024^3 - 1 = 9\,223\,372\,036\,854\,775\,807$ bytes —octetos—); aunque los archivos mayores de 4 GB sólo se pueden crear si el sistema operativo instalado lo permite —con sistema de archivos NT (NTFS)—.

- El tamaño de un archivo ZIP está limitado a 4 GB.

- Los archivos comprimidos con WinRAR pueden ser auto-extraíbles (SFX), y pueden dividirse en diferentes volúmenes.

- Es posible combinar muchos archivos —grandes y pequeños— en el llamado archivo "sólido", donde se comprimen juntos, como si fuesen una sóla entidad. Esto puede ahorrar un 10-50 % en comparación con la compresión de cada archivo individual; pero tiene el inconveniente de que no se pueden recuperar los archivos que estén a continuación de una zona dañada del archivo sólido.

- El software RAR puede opcionalmente generar registros de recuperación y volúmenes de recuperación que hacen que sea posible reconstruir los archivos dañados. Los datos redundantes, en formato de un código Reed-Solomon de corrección de errores, puede añadirse de forma opcional al archivo comprimido para mayor fiabilidad.
  El usuario puede especificar la cantidad de redundancia (como un porcentaje del tamaño original de los datos) que se construirá en los registros de recuperación de un archivo RAR. Una gran redundancia hace el archivo RAR más resistente a la corrupción de datos, lo que permite la recuperación de los daños más graves. Sin embargo, los datos redundantes reducen la eficiencia de la compresión, que es por lo que la compresión y la fiabilidad son conceptos opuestos.

- Puede incluirse información de autenticidad para obtener una mayor seguridad. El software RAR guarda información sobre la última actualización y el nombre del archivo.

- Tiene un asistente intuitivo diseñado especialmente para usuarios principiantes. El asistente hace que todas las funciones de RAR sean fáciles de utilizar a través de un sencillo procedimiento de preguntas y respuestas.

- Consta de excelentes características del tipo *arrastrar y soltar*. El usuario puede arrastrar archivos desde WinRAR a otros programas, hasta el escritorio, o hasta cualquier carpeta. Un archivo soltado en WinRAR se abre inmediatamente para mostrar sus archivos. Al dejar caer un archivo sobre otro que está abierto en WinRAR, aquel se agrega inmediatamente al archivo comprimido. Un grupo de archivos o carpetas arrastradas sobre WinRAR se convierten automáticamente en un nuevo archivo.

- RAR ofrece NTFS y el soporte Unicode (véase [ntfs 06] para NTFS).

- WinRAR está disponible en más de 35 idiomas.

Estas características, combinadas con una excelente compresión, una buena velocidad compresión, una atractiva interfaz gráfica de usuario, y soporte para copias de seguridad, hacen de RAR una de las mejores herramientas de compresión global disponibles en la actualidad.

El algoritmo RAR generalmente se considera propietario, y la siguiente cita (que procede de [donationcoder 06]) arroja alguna luz sobre lo que esto implica:

El hecho de que el algoritmo de codificación RAR sea propietario es una cuestión que vale la pena considerar. Esto significa que, a diferencia de ZIP y 7z, y casi todos los algoritmos de compresión, sólo los programas oficiales WinRAR pueden crear archivos RAR (si bien otros programas los pueden descomprimir). Algunos podrían argumentar que esto es saludable para la industria del software y que la estandarización de un formato abierto sería mejor para todos a largo plazo. Pero para la mayoría de los usuarios, estos asuntos son académicos; WinRAR ofrece un gran apoyo tanto al formato ZIP como al RAR.

Propietario

Un término que indica que una de las partes —o titular— controla la posesión privada, gestión, o utilización de un elemento de su propiedad, normalmente para la exclusión de las otras partes.

—De http://www.wikipedia.org/

La siguiente descripción esclarecedora se obtiene directamente de Eugene Roshal, el diseñador de RAR. (El código fuente del decodificador RAR está disponible en [unrarsrc 06], pero sólo a condición de que no se utilice para realizar ingeniería inversa del codificador.)

RAR tiene dos modos de compresión: general y especial. El modo general emplea un algoritmo basado en LZSS similar a *Deflate* —de ZIP— (Sección 3.23). El tamaño del diccionario de desplazamiento en RAR puede variar de 64 KB a 4 MB (con un valor predeterminado de 4 MB), y la longitud mínima de emparejamiento es de 2. Literales, desplazamientos, y longitudes de emparejamiento se comprimen adicionalmente mediante una codificación de Huffman (recordemos que *Deflate* funciona de manera similar).

A partir de la versión 3.0, RAR también utiliza un modo de compresión especial para mejorar la compresión de datos de tipo texto. Este modo se basa en el algoritmo PPMD (también conocido como PPMII) por Dmitry Shkarin.

RAR contiene un conjunto de filtros para preprocesar audio (en formatos `wav` o `au`), datos de color verdadero, tablas, y ejecutables (x86 de 32 bits e Itanium, véase la nota —más abajo—). Un módulo de análisis de datos selecciona el filtro adecuado y el mejor algoritmo (general o PPMD), según los datos que se compriman.

(*Nota*: La familia de procesadores 80x86 fue originalmente desarrollada por Intel con una longitud de palabra de 16 bits. Debido a su enorme éxito, su arquitectura se extendió a palabras de 32 bits, y actualmente es conocida como IA-32 [arquitectura Intel de 32 bits]. Véase [IA-32 06] para obtener más información. El Itanium es un microprocesador IA-64 desarrollado por Hewlett-Packard e Intel conjuntamente.)

Además de utilizarse como ARchivo Roshal, el acrónimo RAR tiene muchos otros significados, algunos de los cuales se listan a continuación:

(1) Router de Acceso Remoto (un dispositivo de red utilizado para conectar sitios remotos a través de líneas privadas o portadores públicos).(2) Royal Anglian Regiment, una unidad del ejército británico. (3) Royal Australian Regiment[a], una unidad del ejército australiano. (4) Adaptador de Recursos, un módulo específico de la plataforma Java EE. (5) El código del aeropuerto de Rarotonga, Islas Cook. (6) Revise As Required[b] (el favrito de los editores). (7) Road Accident Rescue[c].

Véanse más ejemplos en [acronyms 06].

[a] Regimiento Real de Australia.
[b] Revisar según sea necesario.
[c] Ruta de Rescate de Accidentes.

Rarissimo, de [softexperience 06], es una utilidad de archivo cuyo propósito es comprimir y descomprimir automáticamente archivos en WinRAR. Rarissimo por sí misma no sirve para nada. Se puede utilizar sólo si WinRAR ya está instalado en el ordenador. El usuario de Rarissimo especifica un número de carpetas que Rarissimo debe controlar; la utilidad comprime o descomprime los archivos que hayan sido modificados en estas carpetas. También puede mover los archivos modificados a las carpetas elegidas. Además, el usuario puede especificar con qué frecuencia (en múltiplos de 10 segundos) Rarissimo debe comprobar las carpetas.

Para cada carpeta que deba ser inspeccionada, el usuario tiene que especificar RAR o UnRAR y una carpeta objetivo. Si se especifica RAR, entonces Rarissimo emplea a WinRAR para comprimir cada archivo que haya sido modificado desde la última comprobación, y luego se mueve ese archivo a la carpeta objetivo. Si dicha carpeta se encuentra en otro equipo, esta operación equivale a una transferencia FTP. Si se especifica UnRAR, entonces Rarissimo utiliza WinRAR para descomprimir todos los archivos RAR comprimidos que se encuentren en la carpeta y luego los mueve a la carpeta objetivo.

Una característica importante de Rarissimo es que conserva los flujos de datos alternos NTFS (véase [ntfs 06]). Esto significa que Rarissimo puede manejar archivos de Macintosh que existan en el PC, ya que puede comprimirlos y descomprimirlos preservando los datos y las resource forks.[20]

3.21. El protocolo V.42bis

El protocolo V.42bis es un conjunto de reglas —o un estándar— que fue publicado por la UIT-T para su uso en módems rápidos. Este documento se puede obtener —en castellano— en la dirección:

`http://www.itu.int/rec/T-REC-V.42bis-199001-I/es`.

Se basa en el actual protocolo V.32bis, y se supone que se utilizará para las tasas de transmisión rápida —hasta $57{,}6K$ baudios—. Thomborson [Thomborson 92] ofrece una referencia detallada de esta norma para iniciados. Las normas del UIT-T son recomendaciones, pero normalmente son seguidas por los principales fabricantes de módems. La norma contiene especificaciones para la compresión de datos y la corrección de errores, pero aquí sólo se comenta la primera.

V.42bis especifica dos modos: un modo *transparente*, donde no se usa la compresión, y un modo *con compresión*, que utiliza una variante de LZW. El primero, se utiliza para flujos de datos que no se comprimen bien y pueden incluso causar expansión. Un buen ejemplo es un archivo ya comprimido.

[20] La bifurcación de recursos es una constucción del sistema operativo Mac OS. Para obtener más información, véase http://en.wikipedia.org/wiki/Resource_fork.

Este tipo de archivo se asemeja a datos aleatorios; no tiene ningún patrón repetitivo, y al tratar de comprimirlo con LZW llenará el diccionario con frases cortas de dos símbolos.

El modo comprimido utiliza un diccionario creciente, cuyo tamaño inicial se negocia entre los modems cuando se inicia la conexión. V.42bis recomienda un tamaño del diccionario de 2048 entradas. El tamaño mínimo es de 512 entradas. De ellas, las tres primeras —correspondientes a los punteros 0, 1 y 2— no contienen frases y sirven como códigos especiales. El código 0 (paso al modo transparente, PTM)[21] se envía cuando el codificador nota una baja razón de compresión, y decide comenzar a enviar datos sin comprimir. (Por desgracia, V.42bis no sabe cómo debe —el codificador— probar que la compresión disminuirá.) El código 1 es EVACUACIÓN, evacuación de datos[22]. El código 2 (incremento de la longitud de palabra de código, INCREMENTO)[23] se envía cuando el diccionario está casi lleno y el codificador decide duplicar su tamaño. Un diccionario se considera casi lleno cuando su tamaño excede de un umbral especial (que también es negociado por los módems).

Cuando el diccionario se encuentra ya en su tamaño máximo y se llena, V.42bis recomienda un procedimiento de *reutilización*. Se encuentra y se elimina la frase menos recientemente utilizada, para dejar sitio a una nueva frase. Esto se consigue buscando en el diccionario —desde la entrada 256— la primera frase que no sea un prefijo de cualquier otra. Como ejemplo, supongamos que se encuentra la cadena `abcd`, y que no hay frases de la forma `abcdx` para ningún `x`. Esto significa que `abcd` no se ha utilizado desde su creación, y que es la más antigua de esas frases. Por lo tanto, tiene sentido eliminarla, ya que refleja un viejo patrón de la secuencia de datos de entrada. De esta manera, el diccionario siempre hace patentes las tendencias recientes en la entrada.

> ... hay un grupo de gente cada vez mayor que sostiene la opinión de que *The Ultra-Complete Maximegalon Dictionary*[a] no vale la flota de camiones que se necesita para transportar su edición completa *microstored*. Por extraño que parezca, el diccionario omite la palabra "floopily", que realmente significa "in the manner of something which is floopy".[b]
>
> —Douglas Adams, Life, the Universe, and Everything[c] (1982)

[a] El diccionario ultra-completo Maximegalon.

[b] Floopy significa confuso, perturbado... . Floopily se usa para describir algo que se comporta como si estuviera en ese estado de entontecimiento. Al buscar Flooply en el diccionario urbano —en la dirección de internet http://www.urbandictionary.com/— se muestran algunos ejemplos.

[c] La Vida, el Universo y Todo.

3.22. Varias aplicaciones de LZ

ARC es un programa de compresión/archivo/catalogación desarrollado por Robert A. Freed a mediados de 1980. Ofrece una buena compresión y la capacidad de combinar varios archivos en uno solo, denominado *archive*. Ya a principios de 2003, ARC quedó obsoleta y fue sustituida por aplicaciones PK más recientes.

PKArc es una versión mejorada de ARC. Fue desarrollada por Philip Katz, quien fundó la compañía PKWare [PKWare 03], que todavía comercializa el software PKZip, PKunzip y PKlite. Los programas PK son más rápidos y más generales que ARC y también proporcionan un mayor control al usuario. Pasadas ediciones de este libro —en inglés— tienen más información sobre estas aplicaciones.

LHArc —de Haruyasu Yoshizaki— y LHA —de Haruhiko Okumura y Haruyasu Yoshizaki—, utilizan una codificación Huffman adaptativa con características extraídas de LZSS. El tamaño de la

[21] En inglés, ETM —Enter Transparent Mode—.

[22] En inglés, FLUSH —Flush data— .

[23] En inglés, STEPUP —Step up codeword size—.

ventana de LZ puede ser de hasta $16K$ bytes. ARJ es otra herramienta de compresión —de Robert K. Jung— que utiliza el mismo algoritmo, pero aumenta el tamaño de la ventana de LZ a 26 624 bytes. Un programa similar, conocido como ICE (por el viejo sistema operativo MS-DOS), parece ser una versión falsificada de LHarc o LHA (cualesquiera de ellos). [Okumura 98] tiene alguna información sobre LHarc y LHA como así como una historia sobre la compresión de datos en Japón.

Dos nuevas aplicaciones, muy populares entre los usuarios de Microsoft Windows, son RAR/WinRAR [rarlab 06] (Sección 3.20) y ACE/WinAce [WinAce 03]. Utilizan LZ con un gran buffer de búsqueda (de hasta 4 Mb) en combinación con códigos de Huffman. Están disponibles en varias plataformas y ofrecen muchas características útiles.

3.23. Deflate: Zip y Gzip

Deflate es un popular método de compresión que se utilizó originalmente en los famosos softwares ZIP y Gzip; desde entonces ha sido adoptado por muchas aplicaciones, la más importante de las cuales son: (1) el protocolo HTTP ([RFC1945 96] y [RFC2616 99]), (2) el protocolo de control de compresión PPP ([RFC1962 96] y [RFC1979 96]), (3) los formatos de archivo de gráficos PNG —*Portable Network Graphics*[24], Sección 3.25— y MNG —*Multiple-Image Network Graphics*[25]— ([PNG 03] y [MNG 03]), y (4) el PDF de Adobe (Portable Document Format[26], Sección 8.13) [PDF 01].

Deflate fue diseñado por Philip Katz como parte del formato de archivo Zip e implementado en su software PKZIP [PKWare 03]. Tanto el formato ZIP como el método *Deflate* son de dominio público, lo ha permitido que implementaciones como Zip —de Info-ZIP— y Unzip (esencialmente, los clones PKZIP y PKUNZIP) aparezcan en múltiples plataformas. *Deflate* se describe en [RFC1951 96].

Phillip W. Katz nació en 1962. Obtuvo una licenciatura en ciencias de la computación de la Universidad de Wisconsin, en Madison. Siempre interesado en la escritura de software, comenzó a trabajar en 1984 como programador para Allen-Bradley Co. desarrollando controladores lógicos programables para la industria de la automatización industrial. Más tarde trabajó para Graysoft —otra compañía de software— en Milwaukee, Wisconsin. En ese tiempo se interesó por la compresión de datos y fundó PKWare en 1987 para desarrollar, implementar y comercializar productos de software como PKarc y PKzip. Durante un tiempo, la empresa tuvo un gran éxito vendiendo los programas como shareware.

Siempre un solitario, Katz sufría de problemas personales y legales; comenzó a beber demasiado, y murió el 14 de abril de 2000 por complicaciones relacionadas con el alcoholismo crónico. Tenía 37 años de edad.

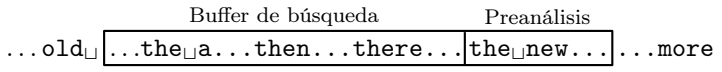
Después de su muerte, PKWare fue vendida —en marzo de 2001— a un grupo de inversionistas. Ellos cambiaron la dirección y el objetivo de su negocio. PKWare actualmente tiene como objetivo el mercado corporativo, y enfatiza la compresión combinada con la encriptación. Su línea de productos se ejecuta en una amplia variedad de plataformas.

La más notable implementación de *Deflate* es **zlib**, una librería de compresión portátil y libre ([zlib 03] y [RFC1950 96]) de Jean-Loup Gailly y Mark Adler, quienes la diseñaron e implementaron para que estuviera libre de requerimientos de patentes y licencias. Esta librería (el código fuente está disponible en [Deflate 03]) implementa los formatos de archivo ZLIB y GZIP ([RFC1950 96] y [RFC1952 96]), que son el núcleo de la mayoría de las aplicaciones de Deflate, incluyendo el popular software Gzip.

[24] Gráficos portátiles para la red.
[25] Gráficos multi-imagen para la red.
[26] Formato de archivo portátil.

Deflate se basa en una variación de LZ77 combinado con códigos de Huffman. Comenzamos con un simple vistazo sobre la base de [Feldspar 03] y continuamos con una descripción completa basada en [RFC1951 96].

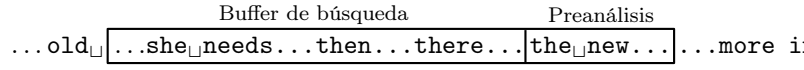
El método original LZ77 (Sección 3.3) intenta emparejar el texto en el buffer de preanálisis con las cadenas que ya están en el buffer de búsqueda. En el ejemplo:



el buffer de preanálisis comienza con la cadena `the␣`, que puede localizarse en una de las tres cadenas en el buffer de búsqueda. El mayor número de caracteres coincidentes es 4. LZ77 escribe tokens en la secuencia de datos comprimidos, cuya estructura es un triplete: (desplazamiento, longitud, símbolo siguiente). El tercer componente es necesario en aquellos casos en que no se ha conseguido localizar ninguna cadena coincidente (imagínese tener `che` en lugar de `the` en el buffer de preanálisis); pero es parte de cada token, lo que reduce el rendimiento de LZ77. La variación del algoritmo LZ77 utilizado en *Deflate* elimina el tercer componente y escribe un par (longitud, desplazamiento) en la secuencia de datos comprimidos. Cuando no se encuentra ninguna coincidencia, en lugar del token, se escribe el carácter sin pareja en el flujo comprimido. Por lo tanto, una cadena comprimida consta de tres tipos de entidades: los **literales** (caracteres no emparejados), los **desplazamientos** (denominados "distancias" en la literatura *Deflate*), y **longitudes**. *Deflate* en realidad escribe los códigos Huffman para estas entidades en el *stream* comprimido, y utiliza dos tablas de códigos: una para los literales y la longitud, y la otra para las distancias. Esto tiene sentido porque los literales son normalmente bytes —y por lo tanto, están dentro del intervalo $[0, 255]$—, y las longitudes están limitadas por *Deflate* a 258. Las distancias, sin embargo, pueden ser números grandes debido a que *Deflate* permite un buffer de búsqueda de hasta $32Kbytes$.

Cuando se determina un par (longitud, desplazamiento), el codificador busca en la tabla de códigos literal/longitud para encontrar el código de la longitud. Este código (que más adelante denominaremos "edoc") se reemplaza por un código Huffman que se escribe en la secuencia de datos comprimidos. El codificador entonces busca en la tabla de códigos de distancias, el código de la distancia y lo escribe (un código prefijo especial con un prefijo fijo de 5 bits) en el flujo comprimido. El decodificador sabe cuando se espera un código de distancia, ya que siempre sigue a un código de longitud.

La variante utilizada por LZ77 por *Deflate* aplaza la elección de un emparejamiento de la siguiente manera. Supongamos que los dos buffers contienen[27]



La secuencia de caracteres emparejados más larga es de 3. Antes de seleccionar este patrón, el codificador guarda la `t` del buffer de preanálisis e inicia la búsqueda de un patrón de coincidencia secundario donde intenta localizar la cadena `he␣new...` en el buffer de búsqueda. Si encuentra un emparejamiento más largo, saca `t` como un literal, seguido por la longitud de la secuencia coincidente de más extensión. Además, hay un parámetro que *admite 3 valores* para controlar este intento de búsqueda secundaria. En el modo "**normal**" del mismo, si el patrón de búsqueda principal era suficiente largo (más extenso que un valor preestablecido), reduce el patrón secundario (le corresponde al implementador decidir cómo hacerlo). En el modo de "**alta compresión**", el codificador siempre lleva a cabo una búsqueda secundaria completa —lo que mejora la compresión— pero consume más tiempo en la selección de un patrón. En el modo "**rápido**", se omite la búsqueda secundaria.

Deflate comprime un archivo de datos de entrada en bloques, donde cada bloque se comprime por separado. Los bloques pueden tener diferentes longitudes y el decodificador determina la longitud de cada bloque basándose en el tamaño de los distintos códigos prefijo utilizados (su longitud está

[27] *Old* es "viejo, antiguo"; *more input* es "más entradas".

limitada a 15 bits) y por la memoria disponible para el codificador (con la excepción de que los bloques en el modo 1 se limitan a 65 535 bytes de datos sin comprimir). El decodificador *Deflate* debe ser capaz de decodificar bloques de cualquier tamaño. *Deflate* ofrece tres modos de compresión, y cada bloque puede estar en cualquier modo. Los modos, son los siguientes:

1. **Sin compresión**. Este modo tiene sentido para archivos o partes de archivos que son incomprimibles (i.e., aleatorios) o ya están comprimidos, o para aquellos casos en que el software de compresión pide dividir un archivo sin comprimirlo. Un caso típico es un usuario que quiere mover un archivo de 8 Gb a otro equipo, pero sólo tiene un DVD para grabar. El usuario puede desear dividir el archivo en dos partes de 4 Gb sin compresión. El software de compresión comercial basado en *Deflate* puede utilizar este modo de operación para segmentar el archivo. Este modo no utiliza las tablas de códigos. Un bloque escrito en el *stream* comprimido —en este modo— comienza con una cabecera especial que indica el modo 1, seguido por la longitud —LEN— de los datos, y tras ella, por LEN bytes de datos literales. Téngase en cuenta que el valor máximo de LEN es 65 535.

2. **Compresión con tablas de códigos fijas**. Dos tablas de códigos están integradas en el codificador y el decodificador *Deflate*, y se utilizan siempre. Esto acelera la compresión y la descompresión y tiene la ventaja añadida de que las tablas de códigos no tienen que ser por guardadas en la secuencia de datos comprimidos. El rendimiento de la compresión, sin embargo, puede verse afectado si los datos que se comprimen son estadísticamente diferentes de los datos utilizados para establecer las tablas de códigos. Los literales y las longitudes de los patrones emparejados se encuentran en la primera tabla y se reemplazan por un código (denominado "**edoc**"); es decir, son sustituidos sucesivamente por un código prefijo que se envía al flujo de datos comprimidos. Las distancias se encuentran en la segunda tabla y se sustituyen por códigos prefijo especiales que se van al stream de compresión. Un bloque escrito en la secuencia de datos comprimidos —en este modo— comienza con una cabecera especial que indica el modo 2, seguido por los datos comprimidos en forma de códigos prefijo de los literales y longitudes, y códigos prefijo especiales para las distancias. El bloque termina con un código prefijo único para indicar el final el bloque.

3. **Compresión con tablas de códigos individuales** generados por el codificador para el conjunto de datos particular que está siendo comprimido. Un codificador *Deflate* sofisticado puede obtener estadísticas acerca de los datos, ya que comprime los bloques, y puede ser capaz de construir tablas de códigos que mejoran a medida que se procesan los bloques. Existen dos tablas de códigos: para los literales/longitudes y para las distancias. Éstas tienen que guardarse de nuevo en el *stream* comprimido, y están escritas en formato comprimido. Un bloque escrito por el codificador en el flujo comprimido —en este modo— se inicia con una cabecera especial, seguida por (1) una tabla de códigos Huffman comprimida y (2) las dos tablas de códigos, cada una de ellas comprimida con los códigos de Huffman que preceden a ambas. Tras esto se guardan los datos comprimidos en forma de códigos prefijo para literales, longitudes y distancias, y termina con un código único para indicar el final del bloque.

3.23.1. Los detalles

Cada bloque comienza con una cabecera de 3 bits, donde el primer bit es 1 para el último bloque del archivo y 0 para los demás bloques. Los otros dos bits son 00, 01 ó 10, para indicar los modos 1, 2 ó 3, respectivamente. Téngase en cuenta que un bloque de datos comprimidos no siempre termina en un byte frontera. La información contenida en el bloque es suficiente para que el decodificador pueda leer todos los bits del bloque comprimido y reconocer el final del bloque. La cabecera de 3 bits del

bloque siguiente está inmediatamente a continuación del bloque actual, y por lo tanto, puede estar situada en cualquier posición —en un byte de la secuencia de datos comprimidos—.
El formato de un bloque en el modo 1 es el siguiente:

1. La cabecera de 3 bits: 000 ó 100.
2. El resto de los bytes actuales se pasa por alto, y los siguientes cuatro bytes contienen LEN y el complemento a uno de LEN (como entero de 16 bits sin signo), donde LEN es el número de bytes de datos en el bloque. Por ello, el tamaño del bloque —en este modo— está limitado a 65 535 bytes.
3. LEN bytes de datos.

El formato de un bloque en el modo 2 es diferente:

1. La cabecera de 3 bits: 001 ó 101.
2. Inmediatamente después, los códigos prefijo fijados para *literales/longitudes* y los códigos prefijo especiales de las distancias.
3. El código 256 (mejor dicho, su código prefijo) que designa el final del bloque.

El modo 2 utiliza dos tablas de códigos: una para los literales y las longitudes, y otra para las distancias. Los códigos de la primera tabla no son lo que se escribe en la cadena comprimida, por lo que para eliminar la ambigüedad, aquí se utiliza el término "edoc" para referirse a ellos. Cada edoc se convierte en un prefijo que se emite a la secuencia comprimida. La primera tabla asigna: los edocs 0 a 255 a los literales, el edoc 256 a fin de bloque, y los edocs 257 a 285 a las longitudes. Los últimos 29 edocs no son suficientes para representar las 256 longitudes de emparejamiento —de 3 a 258—, por lo que se añaden bits adicionales a algunos de esos edocs. La Tabla 3.32 muestra los 29 edocs, los bits extra, y las longitudes que representan. Lo que se escribe en la secuencia comprimida son los códigos prefijo de los edocs (Tabla 3.33). Obsérvese que los edocs 286 y 287 nunca se crean, por lo que sus códigos prefijo nunca se utilizan. Mostraremos más adelante que la Tabla 3.33 que puede ser representada por la secuencia de longitudes de códigos siguiente:

$$\underbrace{8,8,\ldots,8}_{144}\ \underbrace{9,9,\ldots,9}_{112}\ \underbrace{7,7,\ldots,7}_{24}\ \underbrace{8,8,\ldots,8}_{8}, \tag{3.1}$$

pero cualquier codificador y decodificador *Deflate* incluye la tabla completa en lugar de la secuencia de longitudes de códigos. Hay edocs para las longitudes de emparejamiento de hasta 258, por lo que el buffer de preanálisis de un codificador *Deflate* puede tener un tamaño máximo de 258, pero también puede ser más pequeño.

Ejemplos. Si el algoritmo LZ77 ha logrado emparejar una cadena de 10 símbolos, *Deflate* prepara un par $(longitud, distancia)$, donde la longitud 10 se convierte en el edoc 264, que se escribe como el código prefijo de 7 bits: 0001000. Una longitud de 12 se convierte en el edoc 265 seguido por el único bit 1. Ésto se escribe como el código prefijo de 7 bits: 0001010, seguido por 1. Una longitud de 20 se convierte en el edoc 269 seguido de los dos bits 01. Ésto se escribe como los nueve bits: 0001101 | 01. Una longitud de 256 se convierte en el edoc 284 seguido por los cinco bits: 11110. Ésto se escribe como: 11000101 | 11110. Una longitud de emparejamiento mayor que 258 se indica por el edoc 285, cuyo prefijo de 8 bits es: 11000110. El edoc de fin indicador del fin de un bloque es el 256, que se escribe como siete bits cero.

Los 30 códigos de distancia que se muestran en la Tabla 3.34, son códigos prefijo especiales —con un tamaño fijo de 5 bits— que van seguidos de bits adicionales a fin de representar las distancias en el intervalo $[1, 32768]$. El tamaño máximo del buffer de búsqueda es, por lo tanto, 32 768, pero

Código	Bits extra	Longitud	Código	Bits extra	Longitud	Código	Bits extra	Longitud
257	0	3	267	1	15, 16	277	4	67 − 82
258	0	4	268	1	17, 18	278	4	83 − 98
259	0	5	269	2	19 − 22	279	4	99 − 114
260	0	6	270	2	23 − 26	280	4	115 − 130
261	0	7	271	2	27 − 30	281	5	131 − 162
262	0	8	272	2	31 − 34	282	5	163 − 194
263	0	9	273	3	35 − 42	283	5	195 − 226
264	0	10	274	3	43 − 50	284	5	227 − 257
265	1	11, 12	275	3	51 − 58	285	0	258
266	1	13, 14	276	3	59 − 66			

Tabla 3.32: Edocs Literal/Longitud para el modo 2.

Edoc	Bits	Códigos prefijo
0 − 143	8	00110000 − 10111111
144 − 255	9	110010000 − 111111111
256 − 279	7	0000000 − 0010111
280 − 287	8	11000000 − 11000111

Tabla 3.33: Códigos Huffman para edocs en el modo 2.

puede ser más pequeño. La tabla muestra que una distancia de 6 está representada por 00100 | 1, una distancia de 21 se convierte en el código 01000 | 101, y una distancia de 8195 corresponde al código 11010 | 000000000010.

3.23.2. El formato de los bloques en el modo 3

En el modo 3, el codificador genera dos tablas de códigos prefijo: una para *literales/longitudes* y la otra para las distancias. Utiliza las tablas para codificar los datos que constituyen el bloque. El codificador puede generar las tablas con cualquier método. La idea es que un sofisticado codificador *Deflate* puede recoger estadísticas a medida que introduce los datos y comprime los bloques. Las estadísticas se utilizan para construir mejores tablas de códigos para los bloques posteriores. Un codificador sencillo puede omitir el uso de tablas de códigos similares a las del modo 2 o tal vez incluso no generar en absoluto bloques en modo 3. Las tablas de códigos tienen que ser escritas en el flujo comprimido, y en un formato altamente comprimido. Como resultado, una parte importante de *Deflate* es la forma en que comprime las tablas de códigos y las emite. Los pasos principales son (1) Cada tabla comienza como un árbol de Huffman. (2) El árbol se reorganiza para conducirlo a un formato estándar en el que pueda ser representado por una secuencia de longitudes de código. (3) La secuencia es comprimida mediante codificación *run-length* para obtener otra más corta. (4) Se aplica el algoritmo Huffman a los elementos de esa corta secuencia para asignarles códigos de Huffman. Esto crea un árbol de Huffman que se reorganiza de nuevo para convertirlo al formato estándar. (5) Este árbol estándar está representado por una secuencia de longitudes de código que se escriben, después de ser permutadas y posiblemente truncadas, en la salida. Describimos estos pasos en detalle debido a la originalidad de este inusual método.

Recordemos que el árbol de código Huffman generado por el algoritmo básico de la Sección 2.8 no es único. El codificador *Deflate* aplica este algoritmo para generar un árbol de código Huffman, y

Código	Bits extra	Distancia	Código	Bits extra	Distancia	Código	Bits extra	Distancia
0	0	1	10	4	33 − 48	20	9	1025 − 1536
1	0	2	11	4	49 − 64	21	9	1537 − 2048
2	0	3	12	5	65 − 96	22	10	2049 − 3072
3	0	4	13	5	97 − 128	23	10	3073 − 4096
4	1	5, 6	14	6	129 − 192	24	11	4097 − 6144
5	1	7, 8	15	6	193 − 256	25	11	6145 − 8192
6	2	9 − 12	16	7	257 − 384	26	12	8193 − 12288
7	2	13 − 16	17	7	385 − 512	27	12	12289 − 16384
8	3	17, 24	18	8	513 − 768	28	13	16385 − 24576
9	3	25, 32	19	8	769 − 1024	29	13	24577 − 32768

Tabla 3.34: Treinta códigos prefijo para distancias, en el modo 2.

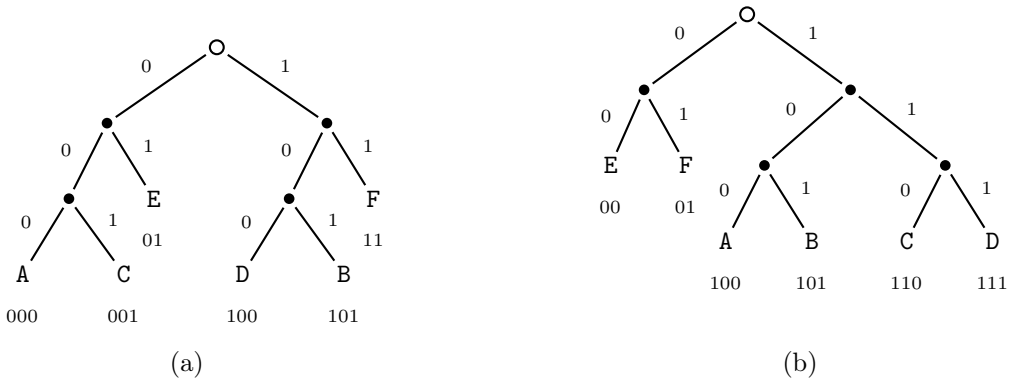


Figura 3.35: Dos árboles de Huffman.

luego reorganiza el árbol y reasigna los códigos para conducirlo a un formato estándar, en el que pueda expresarse de forma compacta por una secuencia de longitudes de código. (El resultado recuerda a los códigos de Huffman canónicos de la Sección 2.8.6.) El nuevo árbol satisface las dos propiedades siguientes:

1. Los códigos más cortos aparecen a la izquierda, y los más largos, a la derecha del árbol de código Huffman.
2. Cuando varios símbolos tienen códigos de la misma longitud, los símbolos (lexicográficamente) más pequeños se colocan a la izquierda.

El primer ejemplo utiliza un conjunto de seis símbolos `A-F` con probabilidades: 0,11, 0,14, 0,12, 0,13, 0,24 y 0,26, respectivamente. La aplicación del algoritmo de Huffman produce un árbol similar al mostrado en la Figura 3.35a. Los códigos de Huffman de los seis símbolos son: 000, 101, 001, 100, 01 y 11. El árbol se reorganiza, y se reasignan los códigos para cumplir con los dos requisitos anteriores, lo que nos lleva al árbol de la Figura 3.35b. Los nuevos códigos de los símbolos son: 100, 101, 110, 111, 00 y 01. El último árbol tiene la ventaja de que puede expresarse completamente por la secuencia: 3, 3, 3, 3, 2, 2 de longitudes de los códigos de los seis símbolos. La tarea del codificador en el modo 3 consiste en generar esta secuencia, comprimirla, y enviarla al flujo de datos comprimidos.

Las longitudes de los códigos se limitan a un máximo de cuatro bits cada uno. Por lo tanto, son números enteros dentro del intervalo $[0, 15]$, lo que implica que el código puede tener, a lo sumo, 15

bits de longitud (esto es un factor que influye en la elección de los tamaños de cada bloque en el modo 3 por parte del codificador *Deflate*).

La secuencia de las longitudes de los códigos representados en un árbol de Huffman tiende a formar rachas de valores idénticos y puede contener varias de ellas con el mismo valor. Por ejemplo, si asignamos las probabilidades: 0,26, 0,11, 0,14, 0,12, 0,24 y 0,13 al conjunto de seis símbolos `A-F`, el algoritmo de Huffman produce códigos de dos bits para `A` y `E`, y códigos de 3 bits para los cuatro símbolos restantes. La secuencia de estas longitudes de código es: 2, 3, 3, 3, 2, 3.

El decodificador lee una secuencia comprimida, la descomprime y la utiliza para reproducir el árbol de código Huffman estándar para los símbolos. En primer lugar, mostraremos cómo dicha secuencia es utilizada por el decodificador para generar una tabla de códigos, y luego cómo ésta es comprimida por el codificador.

Dada la secuencia: 3, 3, 3, 3, 2, 2, el decodificador *Deflate* actúa en tres pasos como sigue:

1. Cuenta el número de códigos para cada longitud de código en la secuencia. En nuestro ejemplo, no hay códigos de longitud 1, pero sí dos de longitud 2 y cuatro de longitud 3.

2. Asigna un valor base para cada longitud de código. No hay códigos de longitud 1, por lo que se les asigna un valor base de 0 y no requieren ningún bit. Los dos códigos de longitud 2, por lo tanto, comienzan con el mismo valor base —0—. A los códigos de longitud 3 se les asigna un valor base de 4 (el doble del número de códigos de longitud 2). El código en *C* que se muestra aquí (siguiendo las especificaciones de [RFC1951 96]) fue escrito por Peter Deutsch. Supone que el paso 1 deja el número de códigos para cada longitud `n` en `bl_count[n]`.

```
code = 0;
bl_count[0] = 0;
for (bits = 1; bits <= MAX_BITS; bits++){
  code = (code + bl_count[bits-1]) << 1;
  next code[bits] = code;
}
```

3. Utiliza el valor base de cada longitud para asignar valores numéricos consecutivos a todos los códigos de esa longitud. Los dos códigos de longitud 2 empiezan en 0, y por lo tanto, son: 00 y 01. Éstos se asignan a los símbolos quinto y sexto: `E` y `F`. Los cuatro códigos de longitud 3 comenzarán a partir de 4, y por lo tanto, son: 100, 101, 110 y 111. Éstos se asignan a los primeros cuatro símbolos: `A-D`. El código en *C* que se muestra aquí (por Peter Deutsch) supone que las longitudes de los códigos están en `tree[I].Len` y que guarda los códigos en `tree[I].Codes`.

```
for (n = 0; n <= max code; n++){
  len = tree[n].Len;
  if (len != 0){
    tree[n].Code = next_code[len];
    next_code[len]++;
  }
}
```

En el siguiente ejemplo, se proporciona la secuencia: 3, 3, 3, 3, 3, 2, 4, 4 y se utiliza para generar una tabla de ocho códigos prefijo. En el paso 1 descubre que no hay códigos de longitud 1, hay un código de longitud 2, cinco de longitud 3, y dos de longitud 4. A los códigos de longitud 1, les asigna el valor base 0. No existen tales códigos, por lo que al siguiente grupo le asigna el valor base "doble 0". Este grupo contiene un código, para que al grupo siguiente (con códigos de longitud 3) le asigne el valor base 2 (dos veces la suma $0 + 1$). Este grupo está formado por cinco códigos, por lo que al

último grupo se le asigna el valor base 14 (el doble de la suma $2+5$). En el paso 3 simplemente genera los cinco códigos de 3 bits —010, 011, 100, 101 y 110— y les asigna los primeros cinco símbolos. A continuación, genera el único código de 2 bits —00— y lo asigna al sexto símbolo. Por último, genera los dos códigos de 4 bits —1110 y 1111— y los asigna a los dos últimos (séptimo y octavo) símbolos.

Dada la secuencia de longitudes de código de la Ecuación (3.1), aplicamos este método para generar su árbol de código Huffman estándar (mostrados en la Tabla 3.33).

El paso 1 descubre que no hay códigos de longitudes de 1 a 6, que hay 24 códigos de longitud 7, 152 códigos de longitud 8, y 112 códigos de longitud 9. A los códigos de longitud 7 les asigna un valor base de 0. Hay 24 de dichos códigos, por lo que al siguiente grupo se le asigna el valor base $2 \cdot (0 + 24) = 48$. Éste grupo contiene 152 códigos, por lo que al último grupo (los códigos de longitud 9) le asigna un valor base de $2 \cdot (48 + 152) = 400$. El paso 3 sólo genera los 24 códigos de 7 bits —del 0 al 23—, los 152 códigos de 8 bits —del 48 al 199— y los 112 códigos de 9 bits —del 400 al 511—. Los valores binarios de estos códigos se enumeran en la Tabla 3.33.

"¿Hay que desalentar la retórica romántica? Además, los Astabigans tienen numerosos visitantes de otros mundos que irán a verla."

—Roger Zelazny, *Doorways in the Sand*[a]

[a]Portales en la arena.

Ahora queda claro que una tabla de códigos de Huffman puede ser representada por una corta secuencia (llamada SQ) de longitudes de código (en lo sucesivo, llamadas CLs). Esta secuencia es especial, ya que tiende a tener rachas de elementos idénticos, por lo que puede ser altamente comprimida por codificador *run-length*. El codificador *Deflate* comprime esta secuencia en un proceso de tres pasos, donde el primer paso emplea la codificación *run-length*; el segundo paso calcula los códigos de Huffman para las longitudes de los *runs* y genera otra serie de longitudes de código (que se llamarán CCLs) para esos códigos de Huffman. El tercer paso, emite una secuencia permutada —posiblemente truncada— de las CCLs con destino a la secuencia de datos comprimidos.

- **Paso 1**. Cuando CL se repite más de tres veces, el codificador lo considera un *run*. Añade CL a una nueva secuencia (denominada SSQ), seguida por el *flag* especial 16 y por un factor de repetición de 2 bits —que indica 3-6 repeticiones—. Un flag 16, por lo tanto, va precedido por CL y seguido por un factor —que indica cuántas veces copiar CL—. Así, por ejemplo, si la secuencia a comprimir contiene seis 7s consecutivos, ésta se comprime como: 7, 16, 10_2 (el factor de repetición 10_2 indica que hay cinco apariciones consecutivas de la misma longitud de código). Si la secuencia contiene 10 longitudes de código consecutivas de 6, éstas se comprimen como: 6, 16, 11_2, 16, 00_2 (los factores de repetición 11_2 y 00_2 indican seis y tres apariciones consecutivas, respectivamente, de la misma longitud de código).

  La experiencia indica que CLs de cero son muy comunes y suelen formar largos *run*s. (Recordemos que los códigos en cuestión son códigos de *literales/longitudes* y distancias. Cualquier archivo de datos a comprimir puede carecer de muchos literales, longitudes y distancias.) Esto es por lo que a los *runs* de ceros se le asignan dos flags especiales: 17 y 18. Un flag 17 va seguido por un factor de repetición de 3 bits, que indica 3-10 repeticiones de CL 0. Un flag 18 va seguido por un factor de repetición de 7 bits, que indica 11-138 repeticiones de CL 0. Por lo tanto, seis ceros consecutivos en una secuencia de CLs se comprimen como: 17, 11_2, y 12 ceros consecutivos en una SQ se comprimen como: 18, 01_2.

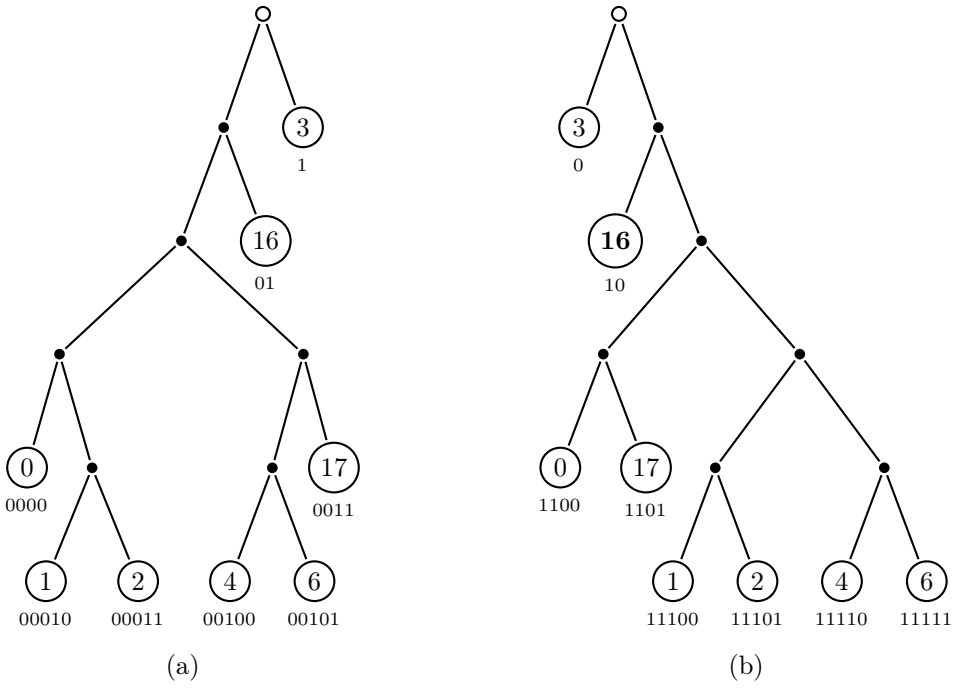


Figura 3.36: Dos árboles de Huffman para longitudes de código.

La secuencia de CLs se comprime de esta manera en una secuencia más corta (que se denomina SSQ) de enteros en el intervalo $[0, 18]$. Un ejemplo puede ser la secuencia de 28 CLs:

$$4, 4, 4, 4, 4, 3, 3, 3, 6, 6, 6, 6, 6, 6, 6, 6, 6, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2$$

que se comprime como la SSQ de 16 números:

$$4, 16, 01_2, 3, 3, 3, 6, 16, 11_2, 16, 00_2, 17, 11_2, 2, 16, 00_2,$$

o, en decimal: $4, 16, 1, 3, 3, 3, 6, 16, 3, 16, 0, 17, 3, 2, 16, 0$.

- **Paso 2.** Prepara los códigos de Huffman para las SSQ —ordenándolos— con el fin de comprimirlos aún más. En nuestro ejemplo, SSQ contiene los siguientes números (con sus frecuencias entre paréntesis): 0 (2), 1 (1), 2 (1), 3 (5), 4 (1), 6 (1), 16 (4), 17 (1). Sus árboles de Huffman inicial y estándar se muestran en la Figura 3.36a,b. El árbol estándar puede representarse por la SSQ de ocho longitudes: 4, 5, 5, 1, 5, 5, 2 y 4. Éstas son las longitudes de los códigos de Huffman asignados a los ocho números: 0, 1, 2, 3, 4, 6, 16 y 17, respectivamente.
- **Paso 3.** Esta SSQ de ocho longitudes se amplía a 19 números insertando ceros en las posiciones que corresponden a las CCLs no utilizadas.

Posición:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
CCL:	4	5	5	1	5	0	5	0	0	0	0	0	0	0	0	0	2	4	0

Las siguientes 19 CCLs se permutan conforme a:

Posición:	16	17	18	0	8	7	9	6	10	5	11	4	12	3	13	2	14	1	15
CCL:	2	4	0	4	0	0	0	5	0	0	0	5	0	1	0	5	0	5	0

La razón de la permutación es terminar con una secuencia de 19 CCLs con una alta probabilidad de tener ceros a la derecha. La SSQ de 19 CCLs —sin incluir sus ceros de la derecha— se envían a la secuencia de datos comprimidos, precedida por su longitud real, que puede ser de 4 a 19. Cada CCL se escribe como un número de 3 bits. En nuestro ejemplo, sólo hay un cero al final, por lo que la secuencia de 18 números: 2, 4, 0, 4, 0, 0, 0, 5, 0, 0, 0, 5, 0, 1, 0, 5, 0, 5 se integra en la cadena comprimida para ser el último código comprimido de una tabla de códigos prefijo. En el modo 3, cada bloque de datos comprimidos requiere dos tablas de códigos prefijo, por lo que se escriben dos de dichas secuencias en el flujo de datos comprimidos.

Finalmente, un lector que llega a este punto (sudando profusamente debido a una profunda concentración en tantos detalles) puede responder con la palabra "demencial". Este esquema de Phil Katz para la compresión de las dos tablas de códigos prefijo es endiabladamente complejo y difícil de seguir, ¡pero funciona!

El formato de un bloque en el modo 3 es el siguiente:

1. La cabecera de 3 bits: 010 ó 110.
2. Un parámetro de 5 bits $HLIT$, que indica el número de códigos existentes en la tabla de códigos *literal/longitud*. Esta tabla tiene códigos 0-256 para los literales, el código 256 para indicar el final de bloque, y los 30 códigos 257-286 para las longitudes. Algunos de los 30 códigos de longitud pueden no aparecer, por lo que este parámetro indica el número de códigos de longitud que existen en realidad en la tabla.
3. Un parámetro de 5 bits $HDIST$, que indica el tamaño de la tabla de códigos para las distancias. Hay 30 códigos en esta tabla, pero pueden faltar algunos.
4. Un parámetro de 4 bits $HCLEN$, que indica el número de CCLs (puede haber entre 4 y 19 CCLs).
5. Una secuencia de $HCLEN + 4$ CCLs: cada uno, un número de 3 bits.
6. Una secuencia SQ de $HLIT + 257$ CLs para la tabla de códigos *literal/longitud*. Esta SQ se comprime como se explicó anteriormente.
7. Una secuencia SQ de $HDIST + 1$ CLs para la tabla de códigos de distancias. Esta SQ se comprime como se explicó anteriormente.
8. Los datos comprimidos, codificados con las dos tablas de códigos prefijos.
9. El código de fin de bloque (el código prefijo de edoc 256).

Cada CCL se envía a la salida como un número de 3 bits, pero las CCLs son códigos de Huffman de hasta 19 símbolos. Cuando el algoritmo de Huffman se aplica a un conjunto de 19 símbolos, los códigos resultantes pueden ser de hasta 18 bits de longitud. Es responsabilidad del codificador asegurar que cada CCL es un número de 3 bits y ninguno excede de 7. La definición formal [RFC1951 96] de *Deflate* no especifica cómo llevar a cabo esta restricción en los CCLs.

3.23.3. La tabla hash

Esta breve sección analiza el problema de la localización de una cadena coincidente —o *match*— en el buffer de búsqueda. El buffer es de 32 Kb de longitud, por lo que una búsqueda lineal es demasiado lenta. La búsqueda lineal para un emparejamiento de cualquier cadena requiere una exploración completa del buffer de búsqueda. Si *Deflate* debe ser capaz de comprimir grandes archivos de datos

en un tiempo razonable, debe utilizar un método de búsqueda sofisticado. El método propuesto por el estándar *Deflate* se basa en una tabla hash. Este método está recomendado con vehemencia por el estándar, pero no es indispensable. Un codificador que utiliza diferentes métodos de búsqueda, aún es compatible y puede llamarse a sí mismo un codificador *Deflate*. Quienes no están familiarizados con las tablas hash, deberían consultar cualquier texto sobre estructuras de datos.

En lugar de separar los buffers de preanálisis y de búsqueda, el codificador debe tener uno de 32 Kb. El buffer se rellena con datos de la entrada e, inicialmente, todo es buffer de preanálisis. En el método original LZ77, una vez que los símbolos han sido examinados, se desplazan dentro del buffer de búsqueda. El codificador *Deflate*, por el contrario, no mueve los datos en su buffer y, en su lugar, desplaza un puntero (o separador) de izquierda a derecha, para indicar el punto donde termina el buffer de preanálisis y comienza el buffer de búsqueda. En poco tiempo, cadenas de 3 símbolos residentes en el buffer de preanálisis se envían a la función hash para ser añadidas a la tabla hash. Después de calcular el valor hash para una cadena, el codificador examina la tabla hash en busca de coincidencias. Asumiendo que un símbolo ocupa n bits, una cadena de tres símbolos puede producir valores comprendidos en el intervalo $\left[0, 2^{3n} - 1\right]$. Si $2^{3n} - 1$ no es demasiado grande, la función hash puede devolver valores dentro este intervalo, lo que tiende a minimizar el número de colisiones. De lo contrario, la función hash puede devolver valores comprendidos en un intervalo más pequeño, como 32 Kb (el tamaño del buffer de *Deflate*).

Demostremos los principios del *hashing* de *Deflate* con la siguiente cadena de 17 símbolos:

```
a b b a a b b a a b a a b a a a a
1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7
```

Inicialmente, las 17 posiciones del buffer son *buffer de preanálisis* y la tabla hash está vacía:

0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	...

Suponemos que el primer triplete `abb` produce un valor de hash de 7. El codificador saca el símbolo `a` tal cual es, desplazándolo al buffer de búsqueda (moviendo el separador entre los dos buffers hacia la derecha), y actualizando la celda 7 de la tabla hash a 1:

```
a | b b a a b b a a b a a b a a a a
1 | 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7
```

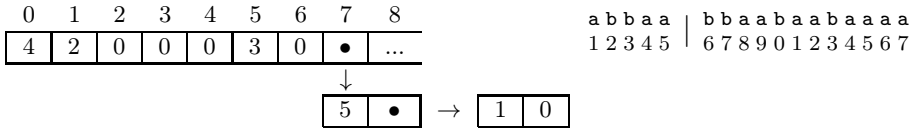
0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	1	...

En los tres pasos siguientes, las cadenas `bba`, `baa` y `aab` se pasan a la función hash para obtener —digamos— los valores 1, 5 y 0, respectivamente. El codificador saca los tres símbolos `b`, `b` y `a` sin modificar, mueve el separador, y actualiza la tabla hash de la siguiente manera:

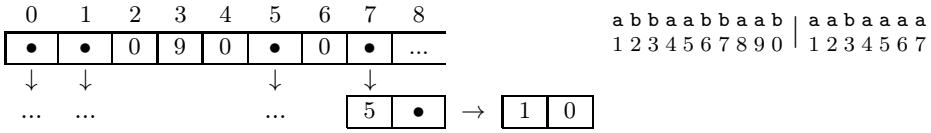
```
a b b a | a b b a a b a a b a a a a
1 2 3 4 | 5 6 7 8 9 0 1 2 3 4 5 6 7
```

0	1	2	3	4	5	6	7	8
4	2	0	0	0	3	0	1	...

A continuación, se pasa el triplete `abb` a la función hash, y como ya sabemos, ésta proporciona el valor 7. El codificador encuentra un 1 en la celda número 7 de la tabla hash, así que busca una cadena que comience por `abb` en la posición 1 de su buffer. Encuentra una coincidencia de tamaño 6, por lo que envía a la salida el par $(5 - 1, 6)$. El desplazamiento (4), es la diferencia entre el inicio de la cadena actual (5) y el inicio de la cadena emparejada (1). Ahora hay dos cadenas que comienzan por `abb`, por lo que la celda 7 debe apuntar a ambas. Por lo tanto, se convierte en el inicio de una lista enlazada (o cadena), cuyos ítems son 5 y 1. Téngase en cuenta que el 5 precede a l en esta cadena, por lo que las búsquedas posteriores de la cadena encontrarán el 5 antes, y por lo tanto, se tiende a encontrar coincidencias con los desplazamientos más pequeños; debido a eso tiene códigos de Huffman cortos.



Los seis símbolos han sido emparejados con la cadena situada a partir de la posición 5, por lo que la siguiente posición a considerar es la $6 + 5 = 11$. Mientras se desplaza a la posición 11, el codificador envía a la función hash las cinco cadenas de 3 símbolos que encuentra en el camino (las que comienzan en las posiciones 6 a 10). Éstas son: `bba`, `baa`, `aab`, `aba` y `baa`. La función hash proporciona para ellas los valores respectivos: 1, 5, 0, 3 y 5 (asumimos arbitrariamente que la función hash proporciona el valor 3 para la cadena `aba`). La celda 3 de la tabla hash se actualiza a 9, y las celdas 0, 1 y 5 se convierten en los inicios de cadenas vinculadas.



Continuando desde la posición 11, la cadena `aab` proporciona —a través de la función hash— el valor 0. Siguiendo la cadena desde la celda 0, encontramos coincidencias en las posiciones 4 y 8. El último emparejamiento es más largo y coincide con la cadena de 5 símbolos: `aabaa`. El codificador envía a la salida el par $(11 - 8, 5)$ y se desplaza a la posición $11 + 5 = 16$. Al hacerlo, también manda a la función hash las cadenas de 3 símbolos que se comienzan en las posiciones 12, 13, 14 y 15. Cada valor hash se agrega a la tabla hash. (Fin del ejemplo.)

Está claro que las cadenas pueden llegar a ser muy largas. Un ejemplo es un archivo gráfico con grandes áreas uniformes, donde muchas cadenas de 3 símbolos serán idénticas; la función hash proporcionará el misma valor, y se añadirán a la misma celda en la tabla hash. Puesto que la búsqueda de una cadena debe ser lineal, una cadena larga frustra el propósito de una tabla hash. Ésta es la razón por la que *Deflate* tiene un parámetro que limita el tamaño de una cadena. Si una cadena supera este tamaño, sus elementos más antiguos deben ser truncados. El estándar *Deflate* no especifica cómo se debe hacer esto y lo deja a la discreción del implementador. La limitación del tamaño de una cadena va en detrimento de la calidad de la compresión, pero puede reducir significativamente el tiempo de compresión. En aquellas situaciones donde el tiempo de compresión no es importante, el usuario puede especificar cadenas largas.

Además, la selección de largos patrones de búsqueda puede no ser siempre la mejor estrategia; el desplazamiento también debe tenerse en cuenta. Un patrón de búsqueda de 3 símbolos puede llegar a utilizar menos bits —con un desplazamiento pequeño— (una vez que el desplazamiento se sustituya por un código de tamaño variable) que un patrón de búsqueda de 4 símbolos —con un desplazamiento grande—.

3.23.4. Conclusiones

Deflate es un algoritmo de propósito general para la compresión sin pérdidas que ha demostrado su valía, año tras año, formando parte de varios programas de compresión populares. El método requiere memoria para los buffers de preanálisis y de búsqueda, y para las dos tablas de códigos prefijos. Sin embargo, el tamaño de la memoria que necesita el codificador y decodificador es independiente del tamaño de los datos o los bloques. La implementación no es trivial, pero es más simple que la de algunos métodos modernos —tales como JPEG 2000 o MPEG—. Los algoritmos de compresión que están dirigidos a tipos específicos de datos, como audio o video, pueden funcionar mejor que *Deflate* con tales datos, pero *Deflate* normalmente produce factores de compresión de 2,5 a 3 con texto,

ligeramente menor con los archivos ejecutables, y algo mayor con las imágenes. Lo más importante es —incluso en el peor de los casos— que *Deflate* expande los datos en sólo 5 bytes por bloque de 32 Kb. Finalmente, existen implementaciones libres evitan las patentes. Nótese que el método original, diseñado por Phil Katz, ha sido patentado (patente estadounidense 5 051 745, 24 de septiembre de 1991) y está asignada a PKWARE.

3.24. LZMA y 7-ZIP

LZMA es el principal algoritmo (así como el método por defecto) utilizado en el popular software de compresión 7z (o 7-Zip) [7z 06]. Ambos —7z y LZMA— son creaciones de Igor Pavlov. El software se ejecuta en Windows y es gratis. Ambos —LZMA y 7z— fueron diseñados para proporcionar una elevada compresión, rápida descompresión, y bajos requerimientos de memoria para la descompresión.

La característica principal de 7z es su arquitectura abierta. El software actualmente puede comprimir datos en uno de los seis algoritmos, y puede —en principio— soportar cualquier método de compresión nuevo. Los algoritmos actuales son los siguientes:

1. LZMA. Es una sofisticada extensión de LZ77
2. PPMD. Una variante de PPMdH, de Dmitry Shkarin
3. BCJ. Un conversor para ejecutables x86 de 32 bits (véase la nota de más abajo)
4. BCJ2. Un conversor similar
5. BZip2. Una implementación del método de Burrows-Wheeler (Sección 8.1)
6. *Deflate*. Un algoritmo basado en LZ77 (Sección 3.23)

(**Nota:** La familia de procesadores 80x86 fue originalmente desarrollada por Intel con una longitud de palabra de 16 bits. Debido a su enorme éxito, su arquitectura se ha ampliado a palabras de 32 bits y actualmente es conocida como IA-32 [Arquitectura Intel de 32-bits]. Véase [IA-32 06] para obtener más información.)

Otras características importantes de 7z son las siguientes:

- Además de la compresión y descompresión de datos en formato LZMA, el software 7z puede comprimir y descomprimir datos en formato ZIP, GZIP y BZIP2; puede empaquetar y desempaquetar datos en formato TAR; y puede descomprimir datos originalmente comprimidos en RAR, CAB, ARJ, LZH, CHM, Z, CPIO, RPM y DEB.
- Cuando comprime datos en los formatos ZIP o GZIP, 7z proporciona razones de compresión un 2-10 % mejores que las obtenidas por PKZip y WinZip.
- Un archivo de datos comprimido en 7z incluye un descompresor; es autoextraíble.
- El software 7z se integra con Windows Shell [Horstmann 06].
- Constituye un potente gestor de archivos.
- Ofrece una versión en línea de comandos de gran potencia.
- Cuenta con un *plugin* para *FAR Manager*.[28] (*FAR Manager* es un programa para la gestión de carpetas y archivos en sistemas operativos Windows. Funciona en modo de texto y permite:

[28] Véase http://www.farmanager.com/.

  - Visualización de archivos y directorios;
  - Editar, copiar y renombrar los archivos;
  - y muchas otras acciones.)

- Incluye localizaciones para 60 idiomas.
- Se pueden encriptar los datos comprimidos con el algoritmo estándar de encriptación avanzado (AES-256), basado en una clave de cifrado de 256 bits [FIPS197 01]. (El algoritmo original de AES —también conocido como Rijndael— se basa en claves de 128 bits.) El usuario introduce una cadena de texto como contraseña, y 7z emplea una función de derivación de clave para obtener los 256 bits de la clave a partir de la contraseña. Esta función se basa en el algoritmo hash SHA-256 [SHA256 02], y calcula la clave generando una cadena muy larga derivada de la contraseña y que sirve como entrada a la función hash. Los 256 bits emitidos por la función hash se convierten en la clave de cifrado.

Para generar la cadena, 7z comienza con la contraseña, codificada con el esquema de codificación UTF-16 (dos bytes por carácter, véase [UTF16 06]). A continuación, genera y concatena $256K = 2^{18}$ pares $(contraseña, entero)$, con enteros de 64 bits comprendidos entre 0 y $2^{18}-1$ —ambos inclusive—. Si la contraseña es de p símbolos de longitud, entonces cada par ocupa $2p+8$ bytes (los bytes están dispuestos en *little endian* —esto es, de menor a mayor valor posicional—), y la longitud total de la cadena final es de $2^{18}\,(2p+8)$ bytes; ¡muy larga!

El término *Little Endian* significa que el byte de orden inferior (el extremo de valor posicional más pequeño) de un número o una cadena se almacena en la memoria en la dirección más baja (la primera que se encuentre). Por ejemplo, dado el número de 4 bytes $b_3b_2b_1b_0$, si el byte menos significativo —b_0— se almacena en la dirección A, el byte más significativo —b_3— se almacenará en la dirección $A+3$.

LZMA (que procede de Lempel-Ziv-Markov Algoritmo-cadena) es el algoritmo de compresión por defecto de 7z. Es una variante de LZ77 diseñada para obtener razones de compresión altas y una descompresión rápida. Un kit de desarrollo de software gratuito (SDK) con e código fuente de LZMA —en C, C++, C# y Java— está disponible en [7z 06] para cualquiera que quiera estudiar, entender, o modificar este algoritmo. Las principales características de LZMA son los siguientes:

- Velocidades de compresión de alrededor de $1^{Mb}/_{s}$ en un procesador de 2 GHz. Se obtienen típicamente velocidades de descompresión de aproximadamente 10-20$^{Mb}/_{s}$ en una CPU de 2 GHz.
- El tamaño del descompresor puede ser tan sólo de $2Kb$ (si es optimizado para el tamaño del código), y sólo requiere de 8-32Kb (además del tamaño del diccionario) para sus datos.
- El tamaño del diccionario es variable y puede ser de hasta $4Gb$, pero la implementación actual lo limita a $1Gb$.
- Soporta multihilo —*multithreading*— y el hiperhilo —*Hyperthreading*— del Pentium 4. (*Hyperthreading* es una tecnología que permite compartir y repartir los recursos mientras se proporciona un entorno multiprocesador con una única CPU). La versión actual del codificador LZMA puede utilizar uno o dos hilos, y el decodificador LZMA puede utilizar un solo hilo.
- El decodificador LZMA utiliza únicamente operaciones con enteros y se puede implementar fácilmente en cualquier procesador de 32 bits (su implementación en una CPU de 16 bits es más complicada).

El principio de compresión de LZMA es similar al de *Deflate* (Sección 3.23), pero utiliza la codificación del rango (Sección 2.15.1) en lugar de la codificación de Huffman. Esto complica el codificador, pero produce una mejor compresión (recordemos que la codificación del rango es una versión basada en enteros de la codificación aritmética y puede obtener compresiones muy cercanas a la entropía de los datos), mientras reduce al mínimo el número de renormalizaciones necesarias. La codificación del rango está implementada en binario, de tal manera que los desplazamientos se utilizan para dividir números enteros, evitando así la lenta operación de "la división".

Recuérdese que LZ77 busca en el buffer de búsqueda de la cadena más larga que coincide con aquella del buffer de preanálisis, y luego escribe en la secuencia comprimida un triplete (distancia, longitud, símbolo siguiente) donde "distancia" es la separación entre la cadena en el buffer de preanálisis y su pareja coincidente en el buffer de búsqueda. Por lo tanto, se escriben tres tipos de datos en la salida: *literales* (el símbolo siguiente, a menudo un código ASCII), *longitudes* (números relativamente pequeños), y *distancias* (que puede ser un número grande si el buffer de búsqueda también lo es).

LZMA también proporciona como salida estos tres tipos. Si no logra emparejar ninguna cadena del buffer de preanálisis, saca un literal (un valor dentro del intervalo $[0, 255]$). Si encuentra una coincidencia, entonces saca un par (longitud, distancia) (después de haber sido codificado por el codificador del rango). Debido al gran tamaño del buffer de búsqueda, se utiliza un array —pequeño, de 4 entradas— *histórico-de-distancias* que siempre contiene las cuatro distancias más recientes que ha determinado y sacado. Si la distancia del emparejamiento actual es igual a una de las cuatro distancias de esta matriz, entonces saca un par (longitud, índice) (después de haber sido codificado por el codificador del rango), donde "índice" es un índice de 2 bits al arreglo *histórico-de-distancias.*

La localización de los emparejamientos del buffer de búsqueda se realiza mediante una operación hash de dos bytes: el byte actual en el buffer de preanálisis y el byte inmediatamente a su derecha (véase el siguiente párrafo para obtener más detalles). La salida de la función hash es un índice a un array (el *hash-array*). El tamaño de este arreglo se selecciona eligiendo la potencia de 2 más cercana a la mitad del tamaño del diccionario, así que la salida de la función hash depende del tamaño del diccionario. Por ejemplo, si el tamaño del diccionario es de 256 Mb (ó 2^{28}), el tamaño del array es de 2^{27} y la función hash debe calcular y producir números de 27 bits. El gran tamaño del arreglo minimiza las colisiones hash.

Los lectores con experiencia pueden haber detectado un problema con el párrafo anterior. Si sólo se realiza la operación hash sobre dos bytes, entonces la entrada a la función hash es de sólo 16 bits, por lo que el número de entradas diferentes posibles es de $2^{16} = 65\,536$. La función hash, por lo tanto, puede calcular sólo 65 536 salidas distintas (los índices del *hash-array*), independientemente del tamaño del arreglo. La historia completa es, por consiguiente, más compleja. El codificador LZMA puede hacer operaciones hash sobre 2, 3 ó 4 bytes, y seleccionar el número de ítems en el *hash-array*, según en el tamaño del diccionario. Por ejemplo, un diccionario de 1 Gb (= 2^{30} bytes) produce un *hash-array* de $512M = 2^{29}$ ítems (donde cada ítem en el *hash-array* es un entero de 32 bits). Con el fin de aprovechar la ventaja de un arreglo tan grande, el codificador realiza las operaciones hash sobre cuatro bytes. Cuatro bytes constituyen 32 bits, que proporcionan a la función de hash 2^{32} entradas distintas. La función hash convierte cada entrada en un índice de 29 bits. (Por lo tanto, muchas entradas se convierten en el mismo índice.)

La Tabla 3.38 muestra varias opciones para el usuario e indica cómo éste puede controlar el codificador estableciendo el parámetro *Match Finder* a ciertos valores. El valor `bt4`, por ejemplo, especifica el modo de árbol binario con operaciones hash de 2-3-4 bytes. Para simplificar, el resto de esta descripción trata sobre las operaciones hash de dos bytes (véanse las diferentes opciones de hashing en la Tabla 3.38).

La salida de la función hash se utiliza como un índice al *hash-array* y el usuario puede elegir uno

de los dos métodos de búsqueda, *hash-chain*[29] (el método rápido) o *binary tree*[30] (el método eficiente).

En el método rápido, la salida de la función hash es un índice a un *hash-array* de listas. Cada posición del array es el comienzo de una lista de distancias. Por lo tanto, si los dos bytes a pasar a la función hash son `XY` y el resultado de dicha función es el índice 123, entonces la ubicación 123 del *hash-array* es un lista de las distancias a los pares `XY` en el buffer de búsqueda. Estas listas pueden ser muy extensas, por lo que LZMA sólo comprueba las primeras 24 distancias. Éstas corresponden a las 24 ocurrencias más recientes de `XY` (el número 24 es un parámetro controlado por el usuario). Se selecciona el mejor emparejamiento de los 24, se codifica y se envía a la salida. La distancia de este emparejamiento se añade al principio de la lista, para convertirse en la primera búsqueda con éxito cuando la ubicación 123 del arreglo se compruebe de nuevo. Este método es una reminiscencia de la búsqueda de cadenas coincidentes en LZRW4 (Sección 3.11).

En el método del árbol binario, la salida de la función hash es un índice a un *hash-array* de árboles binarios de búsqueda (Sección 3.4). Inicialmente, el *hash-array* está vacío y no hay árboles. A medida que se leen y codifican los datos, los árboles se generan y crecen, y cada byte de datos se convierte en un nodo de uno de los árboles. Se crea un árbol binario para cada par de bytes consecutivos en los datos originales. Por lo tanto, si los datos contienen `good␣day`, entonces se generan árboles para los pares, `go`, `oo`, `od`, `d␣`, `␣d`, `da`, y `ay`. El número total de nodos en esos árboles es ocho (el tamaño de los datos). Téngase en cuenta que las dos ocurrencias de `o` acaban formando parte de nodos de árboles diferentes. La primera[31] reside en el árbol para `oo`, y la segunda, en el árbol para `od`.

El siguiente ejemplo ilustra cómo —este método— utiliza los árboles binarios para encontrar coincidencias de cadenas y cómo se actualizan los árboles. El ejemplo asume que el parámetro *Match Finder* (Tabla 3.38) está ajustado a `bt2`, lo que indica: 2 bytes para operaciones hash.

Suponemos que los datos a codificar ya están completamente almacenados en un gran buffer (el diccionario). Las cinco cadenas que comienzan con el par `ab` están localizadas en varios puntos como se muestra a continuación:

...abm...	abcd2...	abcx...	abcd1...	aby...
1	2	3	5	7
1	4	0	7	8

Denotamos por `p(n)` el índice (ubicación) de la cadena `n` en el diccionario. Por lo tanto, `p(abm...)` es 11 y `p(abcd2...)` es 24. Cada vez que se busca un árbol binario T y se selecciona una cadena emparejada, T se reorganiza y se actualiza de acuerdo con las dos reglas siguientes:

1. El árbol debe seguir siendo siempre un árbol binario de búsqueda.
2. Si `p(n1)` $<$ `p(n2)`, entonces `n2` no puede estar en ningún subárbol de `n1`. Esto implica que: (a) la última cadena (la que tiene el mayor índice) es siempre la raíz del árbol y (b) los índices disminuyen a medida que se deslizan hacia abajo por el árbol. El resultado es un árbol donde las cadenas más recientes están situadas cerca de la raíz.

También suponemos que la función hash proporciona el valor 62 para el par de bytes `ab`. La Figura 3.37 ilustra cómo se crea el árbol binario para el par `ab`, cómo se actualiza, y cómo se realiza una búsqueda en el mismo. Los siguientes párrafos numerados se refieren a las seis partes de esta figura:

1. Cuando el codificador LZMA llega a la posición 11 y se encuentra el carácter `a`, realiza la operación hash sobre este byte y el que le sigue —`b`—, para obtener 62. Examina la ubicación 62 del *hash-array* y la encuentra vacía. A continuación, genera un árbol binario (para el par `ab`) con un nodo que contiene el puntero 11, y actualiza la ubicación 62 del *hash-array* para que apunte a este árbol. No hay ninguna coincidencia, el byte `a` de la posición 11 se emite como un literal, y el codificador procede a realizar otra operación hash con el siguiente par —`bm`—.

[29] Búsqueda de cadena mediante hashing.
[30] Búsqueda mediante un árbol binario.
[31] También reside en el árbol para `go`; pero cuando se creó este árbol, aún no había aparecido la segunda `o`.

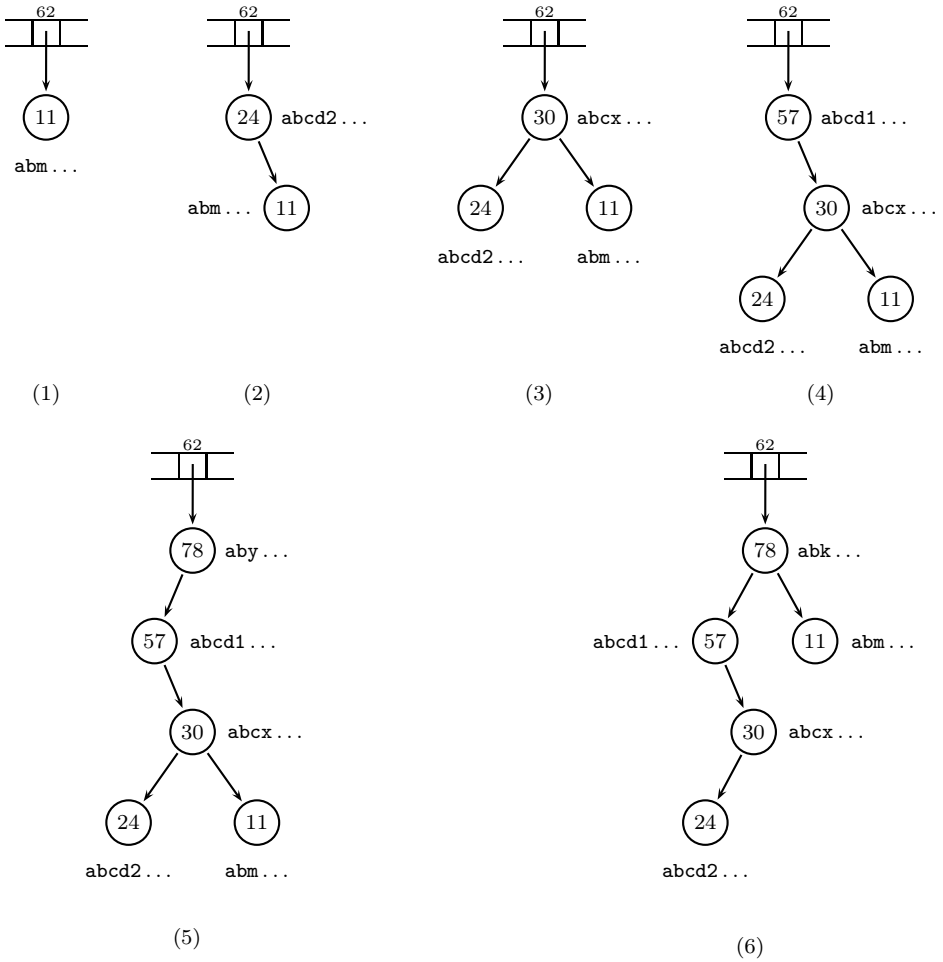


Figura 3.37: Árboles binarios de búsqueda en LZMA.

2. El codificador LZMA encuentra el siguiente par —`ab`— cuando llega a la posición 24. La operación hash produce el mismo 62, y la ubicación 62 del *hash-array* está apuntando a un árbol binario cuya raíz (que es hasta ahora su único nodo) contiene el valor 11. El codificador modifica el árbol, de manera que su nueva raíz sea 24 y su subárbol derecho 11 —porque `abcd2...` es más pequeño que `abm...` (las cadenas se comparan lexicográficamente)—. El codificador busca el número de elementos iguales[32] entre `abcd2...` y `abm...` y determina que es 2. El codificador continúa con `cd2...`, pero antes de comenzar el proceso, pasa a la función hash el par `bc` y, o bien lo añade al árbol binario para `bc` (si es que existe tal árbol) o bien genera un nuevo árbol.

3. El siguiente par `ab` se encuentra en la posición 30. La operación hash produce el mismo 62. El codificador establece 30 (`abcx...`) como la nueva raíz, con 24 (`abcd2...`) como su subárbol izquierdo y 11 (`abm...`) como su subárbol derecho. La cadena con más elementos coincidentes es `abcd2...` y la longitud de emparejamiento es 3. Los siguientes dos pares —`bc` y `cx`— se añaden a sus árboles respectivos (si tales árboles existen), y el codificador continúa con el par `x...`.

4. La siguiente aparición de `ab` se encuentra en la posición 57 y que función hash asocia al valor

[32]Lo que anteriormente hemos llamado longitud de emparejamiento, longitud de coincidencia o longitud de *match*.

62. Ésta se convierte en la raíz del árbol, con 30 (`abcx`...) como su subárbol derecho. La mejor opción de emparejamiento es con `abcd2`..., donde la longitud de coincidencia es 4. El codificador continúa con la cadena 1... que se encuentra en la posición 61.

5. En el último paso de este ejemplo, el codificador encuentra un par `ab` en la posición 78. Esta cadena (`aby`...) se convierte en la nueva raíz, con 57 como su subárbol izquierdo. La longitud de coincidencia 2.

6. Ahora supongamos que la posición 78 contiene la cadena `abk`... en lugar de `aby`.... Puesto que `abm`... es mayor que `abk`..., debe ser desplazado al subárbol derecho de `abk`..., lo que produce un árbol completamente diferente.

Las versiones anteriores de LZMA utilizan una estructura de datos llamada *trie Patricia* (véase la página 195 para la definición de trie), pero esta estructura resultó ineficaz y ha sido eliminada.

Terminamos esta descripción con algunas opciones que pueden ser especificadas por el usuario cuando invoca LZMA.

- –`a`{`N`}. El modo de compresión. 0, 1 y 2 especifican los modos: rápido, normal y máximo, con 2 como opción predeterminada. (La última versión, actualmente en su fase beta, no es compatible con el modo máximo.)

- –`d`{`N`}. El tamaño del diccionario. N es un número entero entre 0 y 30, y el tamaño del diccionario se establece en 2^N. El valor predeterminado es $N = 23$ (un diccionario de 8 Mb), y el máximo actual es $N = 30$ (un diccionario de 1 Gb).

- –`mf`{`MF_ID`}. El buscador de emparejamientos (*Match Finder*). Especifica el método para encontrar coincidencias y limita el número de bytes que se pasan a la función hash. Los requisitos de memoria dependen de esta elección y del tamaño del diccionario —d—. La Tabla 3.38 muestra las opciones. El valor predeterminado de `MF_ID` en los modos normal, máximo y ultra es `bt4` y en los modos "rápido" y "el más rápido" es `hc4`.

`MF_ID`	Memoria	Descripción
`bt2`	$d \times 9{,}5 + 1\,Mb$	Árbol binario con 2 bytes para operaciones hash
`bt3`	$d \times 11{,}5 + 4\,Mb$	Árbol binario con 3 bytes para operaciones hash
`bt4`	$d \times 11{,}5 + 4\,Mb$	Árbol binario con 4 bytes para operaciones hash
`hc4`	$d \times 7{,}5 + 4\,Mb$	Árbol binario con 4 bytes para operaciones hash

Tabla 3.38: Opciones LZMA para el *Match Finder*.

Notas (para la Tabla 3.38):

1. `bt4` utiliza tres tablas hash: con 2^{10} ítems para operaciones hash de dos bytes, con 2^{16} ítems para operaciones hash de tres bytes, y con un tamaño variable para operaciones hash de cuatro bytes. Sólo la última tabla apunta a los árboles binarios. Las otras tablas apuntan a posiciones en el buffer de entrada.

2. `bt3` utiliza dos tablas hash: una con 2^{10} ítems para operaciones hash de dos bytes y la otra con un tamaño variable para operaciones hash de tres bytes.

3. `bt2` utiliza sólo una tabla hash con 2^{16} ítems.

4. `bt2` y `bt3` también pueden encontrar la mayor parte de todos los emparejamientos, pero `bt4` es más rápido.

El autor desea agradecer a Igor Pavlov su contribución con información importante y detalles sobre el material de esta sección.

3.25. PNG

El formato de archivo *Portable Network Graphics*[33](PNG) fue desarrollado a mediados de 1990 por un grupo (el grupo de desarrollo de PNG [PNG 03]), encabezado por Thomas Boutell. El proyecto se inició como respuesta a las cuestiones legales que rodean el formato de archivo GIF (Sección 3.19). El objetivo de este proyecto era desarrollar un formato de archivo de gráficos sofisticado que fuera flexible, que soportara muchos tipos diferentes de imágenes, que se pudiera transmitir fácilmente mediante Internet, y que estuviera libre patentes. El diseño se finalizó en octubre de 1996, y sus principales características son las siguientes:

1. Soporta imágenes de 1, 2, 4, 8, y 16 planos de bits.
2. Juego de colores sofisticado.
3. Una característica de transparencia con un control muy fino proporcionado por un canal alfa.
4. Compresión sin pérdidas mediante el método *Deflate* combinado con la predicción de píxeles.
5. Extensibilidad: Los nuevos tipos de *meta-información* pueden agregarse a un archivo de imagen sin crear incompatibilidades con las aplicaciones existentes.

En la actualidad, PNG es soportado por muchos visores de imágenes y navegadores web de varias plataformas. Esta sección es una descripción general del formato PNG, seguido por los detalles del método de compresión que utiliza.

Un archivo PNG se compone de fragmentos que pueden ser de varios tipos y tamaños. algunos contienen información que es esencial para la visualización de la imagen, y los decodificadores deben ser capaces de reconocerlos y procesarlos. Estos fragmentos se conocen con el nombre de "partes críticas". Otras partes son complementarias. Pueden mejorar la visualización de la imagen o pueden contener metadatos como el título de la imagen, el nombre del autor, la fecha y hora de creación y modificación, etc. (nótese, sin embargo, que los decodificadores pueden optar por no procesar los trozos de este tipo). Se pueden registrar en el grupo de desarrollo de PNG nuevos tipos de bloques que resulten de utilidad.

Una fragmento se compone de las siguientes partes: (1) el tamaño del campo de datos, (2) el nombre del bloque, (3) el campo de datos, y (4) un código de redundancia cíclica de 32 bits (CRC, Sección 3.28). Cada parte tiene un nombre de 4 letras de las cuales: (1) la primera letra es mayúscula para una parte crítica y en minúscula para una parte complementaria, (2) la segunda letra es mayúscula para los fragmentos estándar (los que definidos o registrados por el grupo PNG) y minúscula para un fragmento de una empresa privada (una extensión de PNG), (3) la tercera letra es siempre en mayúsculas, y (4) la cuarta letra es mayúscula si el fragmento es "inseguro para copiar" y en minúsculas, si es "seguro para copiar".

Cualquier aplicación *PNG-aware* procesará todos los fragmentos que reconozca. Puede ignorar con seguridad cualquier parte complementaria que no reconozca, pero si encuentra una parte crítica que no puede reconocer, tiene que parar y emitir un mensaje de error. Si una parte no puede ser reconocida,

[33]Era un acrónimo de "PNG No es GIF"; pero lo más común es identificar las siglas con un formato gráfico portátil para enviar archivos de imágenes a través de la red.

pero su propio nombre indica que es segura para copiar, la aplicación puede leer y reescribir de forma segura incluso si se ha alterado la imagen. Sin embargo, si la aplicación no puede reconocer un fragmento "inseguro para copiar", debe desprenderse del mismo. Éste no debe reescribirse sobre el nuevo archivo PNG. Ejemplos de "seguro para copiar" son fragmentos con comentarios de texto o aquellos que indican el tamaño físico de un píxel. Ejemplos de "inseguro para copiar" son fragmentos de datos con corrección $^{gamma}/_{color}$ o con histogramas de la paleta.

Los cuatro bloques fundamentales definidos por el estándar PNG son IHDR (la cabecera de la imagen), PLTE (la paleta de colores), IDAT (los datos de la imagen, como una secuencia comprimida muestras filtradas), e IEND (el avance —trailer— de la imagen). El estándar también define varios fragmentos auxiliares que se consideran de interés general. Cualquier persona con un fragmento nuevo que sea de interés general puede registrarlo en el grupo de desarrollo de PNG y se le asignará un nombre público (una segunda letra en mayúsculas).

El formato de archivo PNG utiliza un CRC de 32 bits (Sección 3.28) tal como se define en la norma ISO 3309 [ISO 84] o ITU-T V.42 [ITU-T 94]. El polinomio CRC es:

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1.$$

El cálculo específico propuesto en el estándar PNG emplea una tabla precalculada que acelera la velocidad de procesamiento significativamente.

Un archivo PNG comienza con una firma de 8 bytes que ayuda a identificar el software como PNG. Ésta es inmediatamente seguida por un fragmento IHDR con las dimensiones de la imagen, el número de planos de bits, el tipo de color, y el filtrado y entrelazado de datos. Los trozos restantes deben incluir un fragmento PLTE —si el tipo de color es paleta—, y uno o más fragmentos IDAT adyacentes con los píxeles comprimidos. El archivo debe terminar con un fragmento IEND. El estándar PNG define el orden de los fragmentos públicos, mientras que los privados pueden tener su propias restricciones de orden.

Una imagen en formato PNG puede tener uno de los cinco tipos de color siguientes: RGB con 8 ó 16 planos de bits, paleta con 1, 2, 4 u 8 planos de bits, escala de grises con 1, 2, 4, 8 ó 16 planos de bits, RGB con canal alfa (con 8 ó 16 planos de bits), y escala de grises con canal alfa (también con 8 ó 16 planos de bits). Un canal alfa implementa el concepto de color transparente. Un color puede ser designado transparente, y los píxeles de ese color no se dibujan en la pantalla (o se imprimen). En lugar de ver los píxeles, un espectador ve el fondo detrás de la imagen. El canal alfa es un número perteneciente al intervalo $\left[0, 2^{bp} - 1\right]$, donde bp es el número de planos de bits. Suponiendo que el color de fondo es B, un píxel de color transparente P se dibuja en color $(1 - \alpha)\, B + \alpha P$. Éste es una mezcla de un $\alpha\,\%$ del color de fondo y un $(1 - \alpha)$ % del color del píxel.

Quizás la característica más interesante del formato PNG es la forma en que maneja el entrelazado. El entrelazado permite mostrar una versión preliminar de la imagen en la pantalla, y luego mejorarla gradualmente hasta llegar a su estado final —en formato de alta resolución—. El entrelazado especial que se utiliza en PNG se llama Adam 7 en pos de su creador, Adam M. Costello. PNG divide la imagen en bloques de 8×8 píxeles cada una, y muestra cada bloque en siete pasos. En el paso 1, el bloque entero se llena con las copias del píxel de la parte superior izquierda (el marcado con un "1" en la Figura 3.39a). En cada etapa posterior, la resolución del bloque se duplica mediante la modificación de la mitad de sus píxeles de acuerdo con el siguiente número de la Figura 3.39a. Este proceso es más fácil de entender con un ejemplo, como el que se muestra en la Figura 3.39b.

La compresión PNG es sin pérdidas —*lossless*— y se realiza en dos pasos: El primer paso, denominado *filtrado delta* (o simplemente de *filtrado*), convierte el valor de cada píxel a un número mediante un proceso similar a la predicción utilizada en el modo sin pérdidas de imagen de JPEG (Sección 4.8.5). La etapa de filtrado calcula un valor "predeterminado" para cada píxel, y reemplaza cada píxel por la diferencia entre dicho píxel y su valor previsto. El segundo paso emplea *Deflate* para codificar las diferencias. *Deflate* es el tema de la Sección 3.23, por lo que aquí sólo necesitamos describir el filtrado.

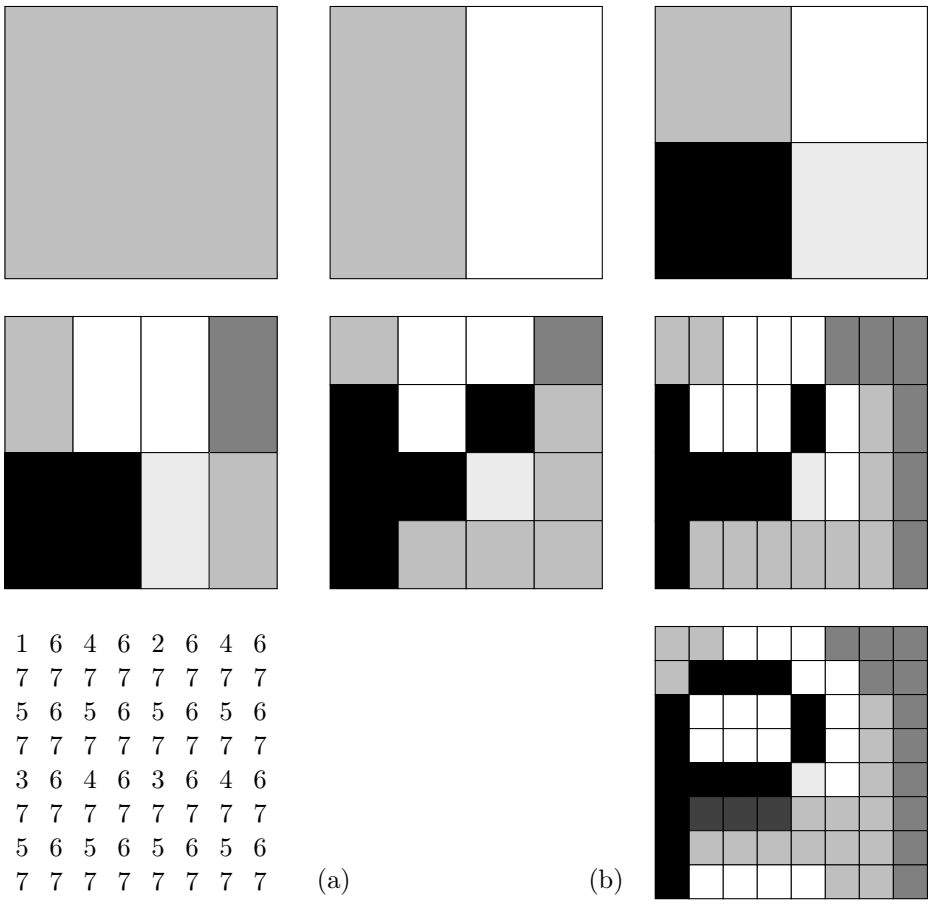


Figura 3.39: Entrelazado en PNG.

El filtrado no comprime los datos. Sólo transforma los datos de cada píxel en un formato más compresible. El filtrado se realiza sobre cada fila de la imagen por separado, por lo que un codificador inteligente puede cambiar los filtros de una fila de la imagen a la siguiente (ésto se llama *filtrado adaptativo*). PNG especifica cinco métodos de filtrado (la primera es simplemente: sin filtrado) y recomienda una heurística simple para la elección de un método de filtrado para cada fila de la imagen. Cada fila comienza con un byte indicando el método de filtrado utilizado en ella. El filtrado se realiza en bytes, no en valores de píxel completo. Ésto es fácil de entender cuando un píxel se compone de tres bytes, especificando los tres componentes de color del píxel. Denotando los tres bytes por a, b y c, podemos esperar que a_i y a_{i+1} estén correlacionados (y también b_i y b_{i+1}, y c_i y c_{i+1}); pero no hay ninguna correlación entre a_i y b_i. Además, en una imagen en escala de grises con píxeles de 16 bits, tiene sentido comparar los bytes más significativos de los dos píxeles adyacentes y luego los bytes menos significativos. Los experimentos sugieren que el filtrado de no es efectivo para las imágenes con menos de ocho planos de bits, y también para las imágenes de paleta, por lo que PNG no recomienda el filtrado en tales casos.

La heurística recomendada por PNG para el filtrado adaptativo es la aplicación de los cinco tipos de filtros a la fila de píxeles y seleccionar aquel que produce la menor suma de valores absolutos de salida. (para esta prueba, los bytes de filtrado deben considerase diferencias con signo).

A continuación, describimos los cinco tipos de filtros:

Tipo de imagen	Planos de bits	t
Escala de grises (*Grayscale*)	1,2,4,8	1
Escala de grises (*Grayscale*)	16	2
Escala de grises con alfa (*Grayscale with alpha*)	8	2
Escala de grises con alfa (*Grayscale with alpha*)	16	4
Paleta (*Palette*)	1,2,4,8	1
Rojo Verde Azul (*RGB*)	8	3
Rojo Verde Azul (*RGB*)	16	6
Rojo Verde Azul con alfa (*RGB with alpha*)	8	4
Rojo Verde Azul con alfa (*RGB with alpha*)	16	8

Figura 3.40: Intervalo (t) entre bytes.

- El primer tipo (tipo 0) es *sin filtrado*.

- El filtrado de tipo 1 (*sub*) establece el byte $B_{i,j}$ —en la fila i y la columna j— en la diferencia $B_{i,j} - B_{i-t,j}$, donde t es el intervalo entre un byte y su predecesor correlacionado (el número de bytes en un píxel). Los valores de t para los diferentes tipos de imagen y planos de bits se muestran en la Tabla 3.40. Si $i - t$ es negativo, entonces no se resta nada, lo que equivale a tener un píxel cero a la izquierda de la fila. La resta se hace módulo 256, y los bytes restados se consideran sin signo.

- El filtrado de tipo 2 (*up* —hasta—) establece el byte $B_{i,j}$ en la diferencia $B_{i,j} - B_{i,j-1}$. La resta es similar a la del tipo 1, pero si j es la fila de la imagen superior, no se realiza ninguna resta.

- El filtrado de tipo 3 (*average* —promedio—) establece el byte $B_{i,j}$ en la diferencia $B_{i,j} - [B_{i-t,j} + B_{i,j-1}] \div 2$. Calcula el promedio del vecino de la izquierda y el superior, y lo resta del byte. Cualquier vecino perdido a la izquierda o abajo se considera cero. Nótese que la suma $B_{i-t,j} + B_{i,j-1}$ puede ser de hasta nueve bits de longitud. Para garantizar que el filtrado es sin pérdidas y puede ser revertido por el decodificador, la suma debe calcularse con exactitud. La división por 2 es equivalente a un desplazamiento a la derecha y traslada la suma de 9 bits a 8 bits. Después de eso, el promedio de 8 bits se resta del byte actual $B_{i,j}$ módulo 256 y sin signo.

  Ejemplo: Supongamos que el byte actual es $B_{i,j} = 112$, su vecino de la izquierda $B_{i-t,j} = 182$, y su vecino superior $B_{i,j-1} = 195$. El promedio es de $(182 + 195) \div 2 = 188$. La resta $(112 - 188) \bmod 256$ produce $-76 \bmod 256$, o lo que es lo mismo: $256 - 76 = 180$. Por lo tanto, el codificador establece $B_{i,j}$ a 180. El decodificador introduce el valor 180 para el byte actual, calcula la media de la misma manera que el codificador para obtener 188, y añade $(180 + 188) \bmod 256$ para obtener 112.

- El filtrado de tipo 4 (*Paeth* —trazado—) establece el byte $B_{i,j}$ en:

$$B_{i,j} - PaethPredictor\left[B_{i-t,j}, B_{i,j-1}, B_{i-t,j-1}\right].$$

  PaethPredictor es una función que utiliza unas sencillas reglas para seleccionar uno de los tres parámetros; luego devuelve ese parámetro. Estos parámetros son los vecinos de la izquierda, de arriba y superior-izquierda. El vecino seleccionado se resta entonces del byte actual —módulo 256 y sin signo—. La función PaethPredictor se define con el siguiente pseudocódigo:

```
function PaethPredictor (a, b, c)
begin
  ; a=izquierda, b=arriba, c=superior izquierda
  p:=a+b-c ; estimación inicial
  pa := abs(p-a) ; calcula la distancia desde la
  pb := abs(p-b) ; estimación inicial,
  pc := abs(p-c) ; hasta a,b,c.
  ; devuelve el elemento más cercano a p
  ; siguiendo el criterio de orden a,b,c.
  if pa<=pb AND pa<=pc then return a
  else if pb<=pc then return b
  else return c
end
```

PaethPredictor debe realizar sus cálculos con exactitud, sin desbordamiento. El criterio de orden seguido por PaethPredictor para elegir el parámetro es importante y no debe ser alterado. Éste orden (que es diferente al dado en [Paeth 91]) es: vecino izquierdo, vecino de arriba y vecino superior izquierdo.

PNG es un formato para una sola imagen, pero el grupo de desarrollo de PNG ha diseñado también un formato de ayuda a la animación llamado formato de red de imágenes múltiples (MNG o *multiple-image network format*), que, propiamente dicho, es un superconjunto de PNG.

El autor está en deuda con Cosmin Truţa por la revisión y rectificación de esta subsección.

> ¿El mundo realmente necesita un nuevo formato de gráficos? Creemos que sí. GIF ya no es de libre uso,... no sería mucho más fácil de implementar completamente que un nuevo formato de archivo. (PNG está diseñado para ser sencillo de implementar, con la excepción del motor de compresión, que sería necesario en cualquier caso.) Creemos que ésta es una excelente oportunidad para diseñar un nuevo formato que fija algunas de las limitaciones conocidas del formato GIF.
>
> Del estándar PNG, RFC 2083, 1999

3.26. Compresión XML: XMill

XMill es una eficiente aplicación software de propósito general para la compresión de documentos XML (Extensible Markup Language[34]). Su descripción en esta breve sección se basa en [Liefke y Suciu 99], pero se pueden obtener más detalles y una implementación libre en [XMill 03]. Primeramente, unas palabras acerca de XML.

XML es un lenguaje de etiquetado para documentos que contengan información estructurada. Un lenguaje de marcas es un mecanismo para identificar las estructuras en un documento. Una corta historia o una novela pueden tener una estructura muy pobre. Puede dividirse en capítulos y también puede incluir notas al pie, una introducción y un epílogo. Una receta de cocina está más estructurado. Se inicia con el nombre del plato y su clase (por ejemplo, ensaladas, sopas, etc.); seguido por los dos elementos estructurales principales: los ingredientes y la preparación. La receta puede describir la manera correcta de servir el plato, y puede terminar con notas, revisiones y comentarios. Una tarjeta de visita está igualmente dividida en una serie de cortos ítems.

El estándar XML [XML 03] define una forma de añadir etiquetas a los documentos, y ha demostrado ser muy popular y exitoso. Un archivo XML contiene datos representados en forma de texto y también

[34] Lenguaje de Marcado (de etiquetado) eXtensible.

incluye etiquetas que identifican los tipos de (o que proporcionan significado a) diferentes ítems de datos. HTML es un lenguaje de marcas —conocido por muchos— pero es restrictivo, ya que define sólo ciertas etiquetas y su significado. La etiqueta `<H3>`, por ejemplo, está definida en HTML y tiene un significado determinado (un cierto encabezado); pero cualquiera que desee usar una etiqueta —digamos, `<blah>`—, primero tiene que convencer al consorcio WWW para definirlo y asignarle un significado, y luego esperar a que los distintos navegadores web lo implementen. XML, por el contrario, se no especifica un conjunto de etiquetas, pero permite definir etiquetas y las relaciones estructurales entre ellas. El significado de las etiquetas (su semántica) queda definido luego por las aplicaciones que procesan los documentos XML o mediante hojas de estilo.

He aquí un sencillo ejemplo de una tarjeta de visita en XML. La mayoría de las etiquetas especifica el inicio y el final de un cierto ítem de datos; son contenedores. Una etiqueta como `<red_backgrnd/>` que termina con un "/" es considerada vacía. Especifica algo a la aplicación que lee y procesa el archivo XML, pero que haga algo que no requiere ningún dato adicional.

```
<card xmlns="http://businesscard.org">
    <name>Melvin Schwartzkopf</name>
    <title>Chief person, Monster Inc.</title>
    <email>mschwa@monster.com</email>
    <phone>(212)555-1414</phone>
    <logo url="widget.gif"/>
    <red_backgrnd/>
</card>
```

En resumen, un archivo XML se compone de las etiquetas y del contenido. Hay seis tipos de etiquetas que pueden producirse en un documento XML: elementos, referencias de entidad, comentarios, instrucciones de procesamiento, secciones marcadas, y declaraciones sobre el tipo de documento. El contenido puede ser cualquier dato digital.

El principal objetivo de los desarrolladores de XMill fue diseñar e implementar un codificador XML de propósito general capaz de comprimir un archivo XML mejor que un compresor típico que comprime sólo los datos de ese archivo. Dado un archivo de datos A, supongamos que un compresor típico —tal como gzip— lo comprime a X. Ahora agrega etiquetas XML en A, convirtiéndolo en B e incrementando su tamaño en el proceso. Cuando B es comprimido por XMill, el archivo resultante —Y— debe ser más pequeño que X. Por ejemplo, los desarrolladores habían probado XMill con un archivo de datos de 98 Mb tomados de *SwissProt*, una base de datos sobre la estructura de las proteínas. El archivo inicialmente fue comprimido con gzip a 16 Mb. El archivo original se convirtió entonces a XML, incrementando su tamaño a 165 Mb y fue comprimido con gzip (a 19 Mb) y con XMill (a 8,6 Mb, aunque después de ajustar XMill para los datos específicos de ese archivo). Sin embargo, ya que XMill es un codificador de propósito especial, no se espera un buen rendimiento en archivos de datos arbitrarios. El diseño de XMill se basa en los siguientes principios:

1. Por sí mismo, XMill no es un compresor. Más bien, es una herramienta extensible para especificar y aplicar los compresores existentes a los ítems de datos XML. XMill analiza el archivo XML, luego invoca a diferentes compresores para comprimir distintas partes del archivo. El principal compresor utilizado por XMill es gzip, pero XMill incluye otros compresores y puede ser también asociado por el usuario a cualquier compresor existente.

2. Separar la estructura de los datos mismos. Las etiquetas XML y los atributos son separados por XMill a partir de los datos del archivo de entrada y se comprimen por separado. Los datos son contenido de elementos XML y los valores de los atributos.

3. Agrupar los elementos que están relacionados. XMill utiliza el concepto de *contenedor*. En el ejemplo anterior de las tarjetas de visita, todas las URLs se agrupan en un solo contenedor,

todos los nombres se agrupan en un segundo, y así sucesivamente. Además, todas las etiquetas XML y los atributos se agrupan en el contenedor de estructuras. El usuario puede controlar el contenido de los contenedores proporcionando expresiones de contenedor en XMill, en la línea de comandos.

4. El uso de compresores de semántica. Un contenedor puede incluir ítems de datos de un tipo determinado, tales como números de teléfono o códigos de aeropuertos. Un usuario sofisticado puede tener un codificador que comprime dichos ítems eficientemente. Por lo tanto, el usuario puede dirigir a XMill para que utilice determinados codificadores para ciertos contenedores, logrando con ello una compresión excelente para todo el archivo de entrada XML. XMill cuenta con varios codificadores, pero cualquier codificador disponible por el usuario puede relacionarse con XMill y éste lo utilizará para comprimir todo contenedor especificado.

Una parte importante de XMill es un lenguaje conciso, denominado *expresiones de contenedores*, que se utiliza para agrupar ítems de datos en los contenedores y especificar los codificadores adecuados para los diversos contenedores.

XMill fue diseñado para preparar los archivos XML para el almacenamiento o la transmisión. A veces, un archivo XML se utiliza conectado con un procesador de consultas, de manera que el archivo tiene que ser buscado a menudo. XMill no es una buena opción para este tipo de aplicación. Otra limitación de XMill es que es eficiente sólo con archivos de gran tamaño. Cualquier archivo XML menor que unos 20 Kb se comprimirá escasamente por XMill, porque XMill añade un consumo de recursos en forma de contabilidad del archivo comprimido. Los archivos pequeños XML, sin embargo, son comunes en situaciones en las que los mensajes en formato XML se intercambian entre los usuarios o entre las aplicaciones.

> No lo sé —no tiene nombre— es una palabra no dicha, no está en ningún diccionario, declaración, símbolo.
>
> —Walt Whitman, *Leaves of Grass*[a], (1900)
>
> [a]Hojas de hierba.

3.27. Compresores EXE

El programa LZEXE es freeware y fue escrito originalmente en la década de 1980 por Fabrice Bellard como una utilidad de propósito especial para comprimir archivos EXE (archivos ejecutables del PC). La idea es que un archivo EXE comprimido por LZEXE puede ser descomprimido y ejecutado con un comando. El descompresor no escribe el archivo descomprimido en el disco, pero lo carga en la memoria, reubica las direcciones, y ¡lo ejecuta! El descompresor utiliza la memoria utilizada eventualmente por el programa que se está descomprimiendo, por lo que no requiere ninguna RAM extra. Además, el descompresor es muy pequeño comparado con descompresores de archivos auto extraíbles.

El algoritmo se basa en LZ. Utiliza una cola circular y un diccionario de árbol para la localización de cadenas coincidentes. La posición y el tamaño del emparejamiento están codificados por un algoritmo auxiliar basado en el método de Huffman. Los bytes sin comprimir se mantienen sin cambios, ya que tratar de comprimirlos aún más habría supuesto un descompresor mucho más complejo y de mayor tamaño. El descompresor se encuentra al final del archivo EXE comprimido y ocupa 330 bytes de longitud (en la versión 0.91). Los principales pasos que realiza el decodificador son los siguientes:

1. Comprobar el CRC (Sección 3.28) para garantizar la fiabilidad de los datos.

2. Localizarse a sí mismo en la parte alta de la memoria RAM; después mover el código comprimido con el fin de dejar suficiente espacio para el archivo EXE.
3. Descomprimir el código, comprobar que es correcto, y ajustar los segmentos si son más grandes que 64K.
4. Descomprimir la tabla de reubicación y actualización de las direcciones reubicables del archivo EXE.
5. Ejecutar el programa, actualizando los registros CS, IP, SS, y SP.

La idea de los compresores de archivos EXE, introducida por LZEXE, era atractiva tanto para los usuarios como para los desarrolladores de software, por lo que se han desarrollado algunos más:

1. PKlite —de PKWare— es un compresor de archivos EXE similar que también puede comprimir archivos COM.
2. DIET —de Teddy Matsumoto— es un compresor de archivos EXE más general que puede comprimir archivos de datos. DIET puede actuar como un monitor, que reside permanentemente en la memoria RAM, en busca de aplicaciones que tratan de leer archivos del disco. Cuando una aplicación intenta leer un archivo de datos comprimido con DIET, DIET lo detecta y realiza la lectura y la descompresión en un proceso que es transparente para la aplicación.

UPX es un compresor de archivos EXE creado en 1996 por Markus Oberhumer y László Molnár. La versión actual (a partir de junio de 2006) es la 2.01. Indicamos aquí una breve cita de [UPX 03]:

> UPX es un empaquetador ejecutable, libre, portátil, extensible y de alto rendimiento para varios formatos de ejecutables diferentes. Logra una razón de compresión excelente y ofrece una descompresión muy rápida. Sus ejecutables no sufren sobrecarga de la memoria u otros inconvenientes.

3.28. CRC

La idea de bit de paridad es simple, vieja y familiar para la mayoría de los profesionales de la informática. Un bit de paridad es el tipo más sencillo de código de detección de errores. Añade fiabilidad a un grupo de bits, lo que permite que el hardware pueda detectar ciertos errores que pueden ocurrir cuando el grupo se almacena en la memoria, se escribe en un disco, o se transmite a través de las líneas de comunicación entre computadoras. Un solo bit de paridad no hace el grupo completamente fiable. Hay ciertos errores que no se pueden detectar con un bit de paridad, pero la experiencia demuestra que incluso un solo bit de paridad puede hacer que la transmisión de datos sea segura en los casos más prácticos.

El bit de paridad se calcula a partir de un grupo de $n-1$ bits, luego se añade al grupo, haciéndolo n bits de longitud. Un ejemplo común es el código ASCII de 7 bits que se convierte en 8 bits después de añadir un bit de paridad. El bit de paridad p se calcula contando el número de unos en el grupo original, y estableciendo p para completar ese número a impar o par. El primero se llama paridad impar, y el segundo, paridad par.

Ejemplos: Dado el grupo de 7 bits 1010111, el número de unos es de cinco —que es impar—. Asumiendo paridad impar, el valor de p debe ser 0, dejando el número total de unos impar. Del mismo modo, el grupo 1010101 tiene cuatro unos, por lo que su bit de paridad impar debe ser un 1, llevando a un total de cinco unos.

Imagine un bloque de datos donde el bit más significativo[35] (MSB) de cada byte es un bit de paridad impar, y los bytes se escriben verticalmente (Tabla 3.41a).

[35] Most Significant Bit.

1 01101001	1 01101001	1 01101001	1 01101001
0 00001011	0 00001011	0 00001011	0 00001011
0 11110010	0 11**0**10010	0 11**0**10**1**10	0 11**0**10**1**10
0 01101110	0 01101110	0 01101110	0 01101110
1 11101101	1 11101101	1 11101101	1 11101101
1 01001110	1 01001110	1 01001110	1 01001110
0 11101001	0 11101001	0 11101001	0 11101001
1 11010111	1 11010111	1 11010111	1 11010111
			0 00011100
(a)	(b)	(c)	(d)

Tabla 3.41: Paridades horizontal y vertical.

Cuando se lee este bloque desde un disco o se recibe desde un ordenador, puede contener errores de transmisión —errores que han sido causados por hardware imperfecto o por interferencias eléctricas durante la transmisión—. Podemos pensar en los bits de paridad horizontal como la *fiabilidad horizontal*. Cuando el bloque es leído, el hardware puede comprobar cada byte, verificando la paridad. Ésta se realiza simplemente contando el número de unos del byte. Si este número es impar, el hardware asume que el byte es bueno. Esta suposición no siempre es correcta, ya que se pueden corromper dos bits durante la transmisión (Tabla 3.41c). Un solo bit de paridad es por lo tanto útil (Tabla 3.41b), pero no proporciona plena capacidad de detección de errores[36].

Una forma sencilla de aumentar la fiabilidad de un bloque de datos consiste en calcular las paridades verticales. El bloque se considera como ocho columnas verticales, y se calcula un bit de paridad impar por cada columna (Tabla 3.41d). Si se corrompen dos bits en un misno byte, la paridad horizontal no lo detecta, pero dos bits de paridad vertical sí. Incluso los bits verticales no proporcionan la capacidad plena de detección de errores, pero son una forma sencilla de mejorar la fiabilidad de los datos significativamente.

Un CRC es una paridad vertical glorificada. CRC procede de Control de Redundancia Cíclica[37] (o Código de Redundancia Cíclica[38]) y es una regla que muestra cómo calcular los bits de verificación vertical (se les llama ahora bits de verificación[39], no son sólo simples bits de paridad) de todos los bits de los datos. Así es como se calcula el CRC-32 (uno de los muchos estándares desarrollados por el CCITT):

El bloque de datos se escribe como un número binario largo. En nuestro ejemplo éste será el número de 64 bits:

$$101101001 \mid 000001011 \mid 011110010 \mid 001101110 \mid 111101101 \mid 101001110 \mid 011101001 \mid 111010111.$$

Los bits individuales se consideran los coeficientes de un *polinomio* (véase más abajo la definición). En nuestro ejemplo, éste será el polinomio de grado 63:

$$\begin{aligned} P(x) &= 1 \times x^{63} + 0 \times x^{62} + 1 \times x^{61} + 1 \times x^{60} + \cdots + 1 \times x^2 + 1 \times x^1 + 1 \times x^0 \\ &= x^{63} + x^{61} + x^{60} + \cdots + x^2 + x + 1. \end{aligned}$$

[36] En la Tabla 3.41b, la paridad horizontal del tercer byte —con el bit corrupto— es 1, que no coincide con el bit de paridad original (el mostrado en la tabla); por lo que se detecta que hay un error en ese byte. En la Tabla 3.42c, la paridad horizontal coincide con la original y no se detecta el error; pero al calcular la paridad vertical, ésta no coincide con la esperada (la mostrada en la Tabla 3.42d), indicando dos bits erróneos.

[37] Cyclical Redundancy Check.

[38] Cyclical Redundancy Code.

[39] Check bits.

Este polinomio se divide por el *polinomio generador* estándar CRC-32*:*

$$CRC_{32}(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1.$$

Cuando un número M es dividido por un entero N, el resultado es un cociente Q (que será ignorado) y un resto R, que se encuentra en el intervalo $[0, N-1]$. Del mismo modo, cuando un polinomio $P(x)$ se divide por un polinomio de grado 32, el resultado es de dos polinomios: un cociente y un resto. El resto es un polinomio de grado 31, lo que significa que tiene 32 coeficientes, cada uno de un sólo bit. Estos 32 bits, son el código CRC-32, que se añade al bloque de datos como cuatro bytes. A modo de ejemplo, el CRC-32 de una versión reciente del archivo con el texto de este capítulo (el original, en inglés) es $\texttt{586DE4FE}_{16}$.

El CRC se llama a veces "huella digital[40]" del archivo. Por supuesto, puesto que es un número de 32 bits, sólo hay 2^{32} CRCs diferentes. Esta cifra equivale aproximadamente a 4 300 millones, por lo que, en teoría, puede haber otros ficheros con el mismo CRC; pero en la práctica, esto es raro. El CRC es útil como un detector de errores de código, ya que tiene las siguientes propiedades:

1. Cada bit del bloque de datos se utiliza para calcular el CRC. Esto significa que el cambio de incluso un sólo bit produce un CRC diferente.
2. Incluso pequeños cambios en los datos, normalmente producen CRCs muy diferentes. La experiencia con CRC-32 muestra que es muy raro que introducir errores en los datos sin modificar el CRC.
3. Cualquier histograma del CRC-32 evaluado en distintos bloques de datos es plano (o muy cercano a plano). Para un bloque de datos dado, la probabilidad de que cualquiera de los 2^{32} posibles CRCs se produzca es prácticamente el mismo.

Otros polinomios generadores comunes son[41]: $CRC_{12}(x) = x^{12} + x^{11} + x^3 + x^2 + x + 1$ y $CRC_{16}(x) = x^{16} + x^{15} + x^2 + 1$. Éstos generan los códigos CRC-12 y CRC-16, que son de 12 y 16 bits de longitud, respectivamente.

Definición: Un polinomio de grado n en x es la función:

$$P_n(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n,$$

donde los a_i son los $n+1$ coeficientes (en nuestro caso, números reales).

En la bibliografía se pueden encontrar dos referencias sencillas y legibles sobre el CRC, en [Williams 93] y en [Ramabadran y Gaitonde 88].

3.29. Resumen

Los métodos basados en diccionarios aquí presentados son diferentes pero se basan en el mismo principio. Leen el símbolo de la secuencia de datos de entrada símbolo a símbolo y añaden frases al diccionario. Las frases son símbolos o cadenas de símbolos de la entrada. La principal diferencia entre los métodos se manifiesta a la hora de decidir qué frases añadir al diccionario. Cuando una cadena del flujo de entrada coincide con una frase del diccionario, el codificador saca la posición de la cadena emparejada en el diccionario. Si esta posición requiere menos bits que la cadena coincidente, se produce la compresión.

En general, los métodos basados en diccionario —cuando están cuidadosamente implementados— dan una mejor compresión que los métodos estadísticos. Por ello, muchos programas de compresión populares están basados o emplean un diccionario como una de sus etapas de compresión.

[40] Fingerprint.

[41] En el libro original, en inglés $CRC_{12}(x) = x^{12} + x^3 + x + 1$, pero éste no es el que he encontrado como $CRC_{12}(x)$.

3.30. Las patentes de compresión de datos

En general se acepta que una invención o un proceso es patentable, pero un concepto matemático, cálculo, o una demostración no lo es. Un algoritmo se asemeja a un concepto de matemática abstracta, que no debe ser patentable. Sin embargo, una vez que el algoritmo se integra en *software* (programas para ordenadores) o en *firmware* (microprogramas para distintos dispositivos) puede que no sea posible separar el algoritmo de su implementación. Una vez que se utiliza la implementación en un nuevo producto (i.e., una invención), dicho producto —incluidos la implementación (software o firmware) y el algoritmo tras ella— puede ser patentable. [Zalta 88] es una discusión general sobre la patentabilidad de algoritmos. Varios algoritmos de compresión de datos comunes, sobre todo LZW, han sido patentados; y la patente LZW se discute aquí con cierto detalle.

A la Corporación Sperry le fue concedida una patente (4 558 302) sobre LZW en diciembre de 1985 (a pesar de que el inventor —Terry Welch—, dejó a Sperry antes de esa fecha). cuando Unisys adquirió Sperry en 1986 se convirtió en el dueño de esta patente y todavía se requiere a los usuarios obtener (y pagar por) una licencia para poder usarlo.

Cuando CompuServe diseñó el formato GIF en 1987, decidió utilizar LZW como método de compresión para los archivos GIF. Parece ser que CompuServe no era consciente en ese momento de que LZW estaba patentado, Unisys no sabía que el formato GIF utiliza LZW. Después de 1987, algunos desarrolladores de software se sintieron atraídos por GIF y publicaron programas para crear y mostrar imágenes en este formato; el cual fue ampliamente aceptado y ahora se utiliza comúnmente en la *World Wide Web*, donde es uno de los formatos de imagen principal para las páginas web y los navegadores.

No fue hasta que GIF se convirtió de facto en un estándar mundial que Unisys contactó con CompuServe para obtener una licencia. Naturalmente, CompuServe y otros usuarios de LZW han tratado de impugnar la patente. Solicitaron a la Oficina de Patentes de Estados Unidos un nuevo examen de la patente LZW, con el resultado (tal vez sorprendente) de que el 4 de enero de 1994, la patente fue reconfirmada y CompuServe tuvo que obtener una licencia (por una suma no revelada) a Unisys ese mismo año. Otros titulares de licencias importantes de LZW (véase [Rodriguez 95]) son: Aldus (en 1991, por el formato de archivo de gráficos TIFF), Adobe (en 1990, por el nivel II de PostScript), y America Online y Prodigy (en 1995).

La patente LZW de Unisys tiene implicaciones significativas para la *World Wide Web*, donde uso de imágenes en formato GIF está muy extendida. Similarmente, la utilidad *compress* de Unix utiliza LZW y por lo tanto requiere una licencia. En los Estados Unidos, la patente expiró el 20 de junio de 2003 (20 años a partir de la fecha de la primera presentación). En Europa (patente `EP0129439`) ha expirado el 18 de junio de 2004. En Japón, las patentes 2 123 602 y 2 610 084 expiraron el 20 de junio de 2004, y en Canadá, la patente `CA1223965` expiró el 7 de julio de 2004.

Actualmente Unisys exime a los productos de software antiguo (los escritos o modificados antes del 1 de enero de 1995) de una licencia de patente. También está exento cualquier software —antiguo o nuevo— no comercial y sin fines de lucro. El software comercial (incluso el shareware) o el firmware creado después del 31 de diciembre de 1994, debe tener una licencia si soporta el formato GIF o implementa LZW. Una política similar se aplica con respecto a TIFF, donde la fecha límite es el 1 de julio de 1995. Téngase en cuenta que los usuarios de computadoras pueden mantener y transferir legalmente archivos GIF y cualquier otro archivo comprimido con LZW; sólo el software de *compresión/descompresión* requiere una licencia.

Para más información sobre la patente y la licencia LZW de Unisys véase [Unisys 03].

Una alternativa a GIF es el formato de archivo gráfico Portable Network Graphics, PNG (pronunciado "ping", Sección 3.25) [Crocker 95], que fue desarrollado expresamente para sustituir a GIF, y evitar las reivindicaciones de patentes. PNG es simple, portátil, con código el fuente disponible libremente, y sin el estorbo de licencias de patentes. Tiene potencial se espera que sustituya a GIF. Sin embargo, cualquier software de conversión de GIF a PNG aún requiere de una licencia de Unisys.

El software de compresión GNU gzip (Sección 3.18) también debe mencionarse aquí como un

sustituto popular para *compress*, ya que está libre de las reivindicaciones de patentes, es más rápido, y proporciona una mejor compresión.

El número de patente LZW de U.S. es 4 558 302 —emitida el 10 de diciembre de 1985—. Aquí tenemos un extracto presentado de parte de la misma (el documento entero constituye 50 páginas).

> Un compresor comprime un flujo de señales de datos de carácter de entrada, almacenando en una tabla de cadenas, las cadenas de señales de datos de carácter encontrados en el flujo de entrada. El compresor busca en la secuencia de entrada para determinar la coincidencia más larga con una cadena almacenada. Cada cadena almacenada consta de una cadena prefijo y un carácter de extensión —donde el carácter de extensión es el último carácter de la cadena— y la cadena prefijo comprende todo salvo el carácter de extensión. Cada cadena tiene una señal de código asociado con la misma y se guarda una cadena en la tabla de cadenas almacenando —al menos implícitamente— la señal de código para la cadena, la señal de código para la cadena prefijo y el carácter de extensión. Cuando se determina la coincidencia más larga entre el flujo de datos de caracteres de entrada y las cadenas almacenadas, se transmite la señal de código para el emparejamiento más largo como la señal de código comprimido para la cadena de caracteres encontrada y se almacena una cadena de extensión en la tabla de cadenas. El prefijo de la cadena extendida es el emparejamiento más largo y el carácter de extensión de la candena extendida es la siguiente señal de datos de carácter de entrada que sigue al emparejamiento más largo. La búsqueda dentro de la tabla de cadenas y la introducción en ella cadenas extendidas se realiza mediante un procedimiento hash de búsqueda limitada. La descompresión se realiza mediante un descompresor que recibe las señales de código comprimido y genera una tabla de cadenas similares a la construída por el compresor para llevar a cabo consultas de las señales de código recibidas con el fin de recuperar las señales de datos de carácter que comprende una cadena almacenada. La tabla de cadenas del descompresor se actualiza mediante el almacenamiento de una cadena que tiene un prefijo de acuerdo con una señal de código recibida anteriormente y un carácter de extensión que concuerda con el primer carácter de la actual cadena recuperada.

Estos son otros métodos de compresión patentados, algunos de ellos mencionados en otra parte de este libro:

1. "Compresión de datos de sustitución de texto con ventanas de búsqueda de longitud finita[42]", Patente de U.S. 4 906 991, emitida el 6 de marzo 1990 (el método LZFG).

2. "Codificación de estructura de datos en árbol de búsqueda para sistemas de compresión de datos de sustitución de texto[43]", patente de U.S. 5 058 144, emitida el 15 de octubre 1991.

   Las dos patentes anteriores fueron emitidas a Edward Fiala y Daniel Greene.

3. "Equipos y método para la compresión de señales de datos y la recuperación de señales de datos comprimidas[44]". Ésta es la patente LZ78, asignada a Sperry Corporation por los inventores Willard L. Eastman, Abraham Lempel, Jacob Ziv, y Martin Cohn. Patente de U.S. 4 464 650, emitida en agosto de 1984.

Lo siguiente, de Ross Williams http://www.ross.net/compression/ ilustra cuán espinoso es el tema de las patentes.

[42] Textual Substitution Data Compression with Finite Length Search Windows.
[43] Search Tree Data Structure Encoding for Textual Substitution Data Compression Systems.
[44] Apparatus and Method for Compressing Data Signals and Restoring the Compressed Data Signals.

> Entonces, justo cuando pensaba que toda esperanza se había ido, llegaron algunas patentes de software que clavó una estaca en el corazón de los algoritmos LZRW haciéndolos inutilizables. ¡Por fin me curé! Dejé la compresión y me embarqué en una nueva vida, dejando atrás el mundo de la compresión de datos para siempre.

Oportunamente, el Dr. Williams mantiene una lista [patents 06] de patentes relacionadas con la compresión de datos.

> Su solicitud de patente será rechazada. Sus permisos se retrasarán. Algo le obligará a entrar en razón y vender sus medicamentos a un costo menor.
> Wu había oído el argumento de antes. Y sabía que Hammond estaba en lo cierto, algunos de los nuevos productos farmacéuticos de bioingeniería habían sufrido retrasos inexplicables y problemas con las patentes.
> Ni siquiera saben exactamente lo que has hecho, pero ya has informado de ello, lo has patentado y lo has vendido.
>
> —Michael Crichton, Parque Jurásico (1991)

3.31. Una unificación

Los métodos basados en diccionarios y los métodos basados en predicciones abordan el problema de la compresión de datos desde dos direcciones diferentes. Cualquier método basado en la predicción, predice (i.e., asigna probabilidades a) el símbolo actual basándose en su contexto de orden N (los N símbolos que le preceden). Este tipo de método normalmente almacena muchos contextos de diferentes tamaños en una estructura de datos y tiene que lidiar con los recuentos de frecuencia, las probabilidades y los rangos de probabilidad. Luego, utiliza la codificación aritmética para codificar la cadena de entrada completa en un número grande. Un método basado en diccionario, por el contrario, funciona de forma diferente. Identifica la frase siguiente en el flujo de entrada, la almacena en su diccionario, le asigna un código, y continúa con la frase siguiente. Ambos enfoques pueden ser utilizados para comprimir datos, porque cada uno obedece a la ley general de la compresión de datos, es decir: asignar códigos cortos a eventos comunes (símbolos o frases) y códigos largos a eventos raros.

Superficialmente, los dos enfoques son totalmente diferentes. Un predictor considera las probabilidades, por lo que puede ser altamente eficiente. Al mismo tiempo, se puede esperar que sea lento, ya que trata con símbolos individuales. Un método basado en diccionario trata con cadenas de símbolos (frases), por lo que absorbe el flujo de entrada más rápido, pero ignora las correlaciones entre las frases, resultando típicamente en una compresión más pobre.

Los dos enfoques son similares debido a que un método basado en diccionario *hace uso de* contextos y probabilidades (aunque de manera implícita) sólo mediante el almacenamiento de frases en su diccionario y la búsqueda de las mismas. La siguiente discusión utiliza el trie LZW para ilustrar este concepto; pero el argumento es válido para cualquier método basado en diccionario, no importan los detalles de su algoritmo ni la estructura de datos de su diccionario.

Imagine la frase `abcdef...` almacenada en un trie LZW (Figura 3.42a). Podemos pensar en la subcadena `abcd` como el contexto de orden 4 de `e`. Cuando el codificador encuentra otra ocurrencia de `abcde...` en el flujo de entrada, localiza nuestra frase en el diccionario, la analiza símbolo a símbolo comenzando por la raíz, obtiene el nodo `e`, y continuar desde allí, tratando de emparejar más símbolos. Con el tiempo, el codificador llegará a una hoja, donde añadirá otro símbolo y asignará otro código. Podemos pensar en este proceso como la adición de una nueva página para el subárbol cuya raíz es la `e` de `abcde...`. Cada vez que la cadena `abcde` se convierte en el prefijo de un análisis, tanto su

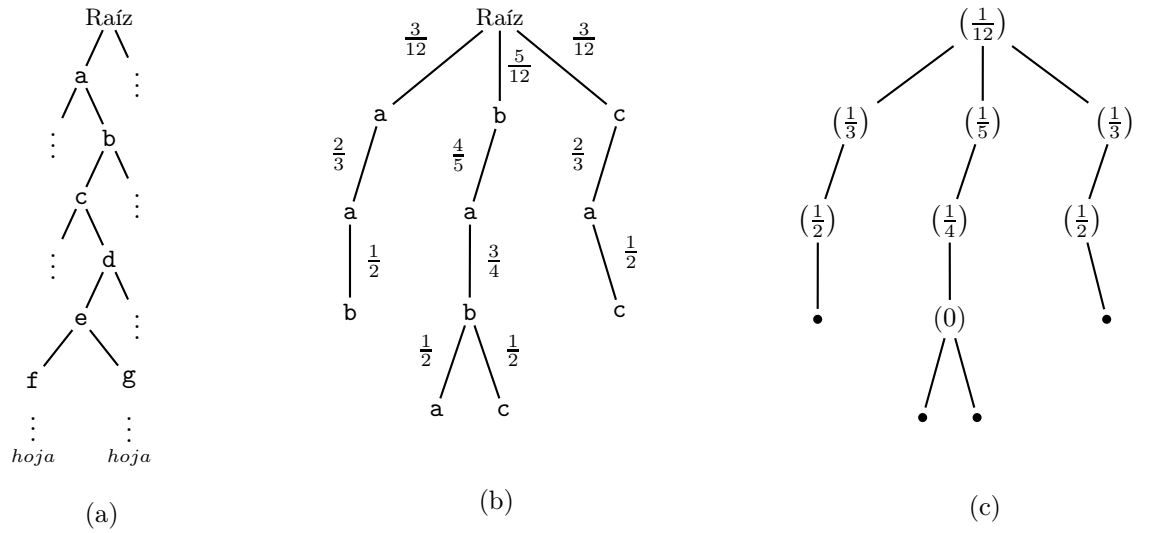


Figura 3.42: Definición de las probabilidades en un árbol de diccionario.

subárbol como su espacio de código (el número de los códigos asociados a ella) crecen en 1. Por lo tanto, tiene sentido asignar al nodo `e` una probabilidad en función del tamaño de su espacio de código, y la discusión anterior muestra que el tamaño del espacio de código de nodo `e` (o, equivalentemente, la cadena `abcde`) se puede medir contando el número de nodos del subárbol cuya raíz es `e`. Así es cómo las probabilidades pueden ser asignadas a los nodos en cualquier árbol de diccionario.

Las ideas de Glen Langdon en la década de 1980 (véase [Langdon 83], pero nótese que la ecuación (8) es errónea —debería decir: $P(y \mid s) = {}^{c(s)}/_{c(s \cdot y)}$—; [Langdon 84] es quizás más útil) dieron lugar a una forma simple de asociar probabilidades no sólo a los nodos, sino también a las ramas de un árbol de diccionario. La asignación de probabilidades a las ramas es más útil, ya que la rama desde el nodo `e` al nodo `f`, por ejemplo, significa una `f` cuyo contexto es `abcde`. La probabilidad de esta rama es, pues, la probabilidad de que una `f` siga a `abcde` en el flujo de entrada. El hecho de que estas probabilidades se pueden calcular en un árbol de diccionario demuestra que todos los algoritmos de compresión de datos basados en diccionarios pueden ser "simulados" por un algoritmo de predicción (sin embargo, nótese que el recíproco no es cierto). Los algoritmos basados en la predicción son, en este sentido, más generales, pero el hecho importante es que estos dos tipos de métodos de compresión aparentemente diferentes pueden ser unificados por las observaciones aquí listadas.

El proceso por el cual un codificador de diccionario se desliza hacia abajo desde la raíz de su árbol de diccionario, analizando una cadena de símbolos, puede ahora tener una interpretación diferente. Podemos visualizarlo como una secuencia para hacer predicciones de los símbolos individuales, los códigos calculados para ellos, y la combinación de los códigos en uno más largo, que eventualmente será escrito en la secuencia de datos comprimidos. Es como si el código generado por un codificador de diccionario para una frase es en realidad un conjunto de pequeños trozos: cada uno, un código para un símbolo.

La regla para calcular la probabilidad de la rama `e` → `f` es contar el número de nodos en el subárbol cuya raíz es `f` (incluyendo el propio nodo `f`) y dividirlo por el número de nodos en el subárbol de `e`. La Figura 3.42b muestra un árbol de diccionario típico con las cadenas `aab`, `baba`, `babc` y `cac`. También se muestran las probabilidades asociadas con cada rama y debería ser fácil para el lector para verificarlas. Téngase en cuenta que las probabilidades de subárboles hermanos no suman 1. Las probabilidades de los tres subárboles de la raíz, por ejemplo, se suman ${}^{11}/_{12}$. El restante ${}^{1}/_{12}$ se le asigna a la raíz misma y representa la probabilidad de que una cuarta cadena, eventualmente, comience en la raíz.

Estas "probabilidades ausentes" se muestran entre paréntesis —en cada nodo correspondiente— en la Figura 3.42c.

Los dos enfoques —diccionario y predicción—, se pueden combinar en un sóla método de compresión. El método LZP de la Sección 3.16 es un ejemplo; el método LZRW4 (Sección 3.11), es otro. Estos métodos trabajan teniendo en cuenta el contexto de un símbolo antes de buscar en el diccionario.

Comparisons date. Adoption screams. Co-ordinates maps.
Darn! as composite. Is mono-spaced art. Composed as train.
Promised as on act. Oops and matrices. Promised to a scan.

—Anagramas de "data compression".

Capítulo 4

Compresión de imágenes

La primera parte de este capítulo describe los tipos de imágenes digitales y sus características básicas, así como los principales enfoques para la compresión de imágenes. Seguidamente, se proporciona una descripción de aproximadamente 30 métodos diferentes de compresión. El autor desea comenzar con las siguientes observaciones:

1. ¿Por qué se incluyen éstos métodos en particular en el libro, mientras que otros fueron excluídos? La respuesta es simple: debido a la documentación a disposición del autor. Se han incluido los métodos de compresión de imágenes que están bien documentados. Los métodos propietarios (patentados), o cuya documentación no estaba clara para el autor, se rechazaron.

2. El tratamiento de los diversos métodos es desigual. Esto, de nuevo, refleja la documentación a disposición del autor. Algunos métodos han sido documentados por sus desarrolladores con gran detalle, y esto se refleja en este capítulo. Cuando no estaba disponible ninguna documentación detallada para un algoritmo de compresión, aquí sólo se han descrito sus principios básicos.

3. No se intentan comparar los distintos métodos aquí descritos. Ésto es debido a que la mayoría de los métodos de compresión de imágenes han sido diseñadas para un tipo específico de imagen, y también a causa de las dificultades prácticas de obtener todo el software y adaptarlo para funcionar en la misma plataforma.

4. Los métodos de compresión descritos en este capítulo no están dispuestos en un orden en particular. Después de mucho pensar y de muchas pruebas, el autor renunció a toda esperanza de clasificación de los métodos de compresión de cualquier manera razonable. Los lectores que busquen un método en particular pueden consultar el índice de contenidos y el índice analítico para localizarlo fácilmente.

Una imagen digital es una matriz rectangular de puntos o elementos de imagen, dispuestos en m filas y n columnas. La expresión $m \times n$ se llama *resolución* de la imagen, y los puntos se denominan *píxeles* (excepto en los casos de imágenes de fax y de compresión de vídeo, donde se conocen como *pels*). El término "resolución" a veces también se utiliza para indicar el número de píxeles por unidad de longitud de la imagen. Por lo tanto, dpi[1] significa puntos por pulgada. Para el propósito de la compresión de imágenes es útil distinguir los siguientes tipos de imágenes:

1. Una imagen *binivel* (o monocromática). Ésta es una imagen donde los píxeles pueden tener uno de dos valores, normalmente referenciados como negro y blanco. Cada píxel de dicha imagen se representa mediante un bit, haciendo de éste, el tipo de imagen más sencilla.

[1]Dots per inch.

2. Una imagen en *escala de grises*. Un píxel en dicha imagen puede representarse con n bits; admitiría 2^n posibles tonos de gris (o tonos de otro color) y normalmente los valores de los píxeles estarían comprendidos entre 0 y $2^n - 1$. No obstante, si las imágenes son con signo, los valores de los píxeles estarían en el rango, de $-\left(2^{n-1} - 1\right)$ a $+2^{n-1}$. El valor de n es normalmente compatible con un tamaño de byte; i.e., es 4, 8, 12, 16, 24, o algún otro múltiplo conveniente de 4 ó de 8. El conjunto de los bits más significativos de todos los píxeles es el mapa de bits más significativo. Así, una imagen en escala de grises tiene n planos de bits.[2]

3. Una imagen de *tonos continuos*. Este tipo de imagen puede tener muchos colores similares (o escalas de grises). Cuando los píxeles adyacentes difieren en una sola unidad, es difícil o incluso imposible para el ojo para distinguir sus colores. Como resultado, tal imagen puede contener zonas con colores que parecen variar continuamente a medida que el ojo se mueve a lo largo del área. Un píxel en dicha imagen está representado por un único número cualquiera de gran tamaño (en el caso de muchas escalas de grises) o tres componentes (en el caso de una imagen en color). Una imagen de tono continuo es normalmente una imagen natural (natural en oposición de artificial) y se obtiene al tomar una fotografía con una cámara digital, o mediante el escaneo de una fotografía o una pintura. Las Figuras 4.53 a 4.56 son ejemplos típicos de imágenes de tono continuo. Un estudio general de la compresión sin pérdidas de este tipo de imágenes es [Carpentieri et al. 00].

4. Una imagen de *tonos discretos* (también llamada imagen gráfica o imagen sintética). Normalmente es una imagen artificial. Puede tener pocos o muchos colores, pero no tiene el ruido y el difuminado de una imagen natural. Ejemplos son: un objeto artificial o máquina, una página de texto, un diagrama, un dibujo animado, o el contenido de una pantalla de ordenador. (No toda imagen artificial es de tonos discretos. Una imagen generada por ordenador que está destinada a parecer natural es una imagen de tonos continuos, a pesar de su ser generada artificialmente.) Los objetos artificiales, textos y dibujos de líneas tienen bordes nítidos y bien definidos, por lo que tienen un alto contraste con el resto de la imagen (el fondo). Los píxeles adyacentes en una imagen de tonos discretos, a menudo son idénticos o varían de forma significativa en valor. Tal imagen provoca que no se comprima bien con los métodos con pérdidas, porque la pérdida de tan sólo unos pocos píxeles pueden hacer una carta ilegible, o cambiar un patrón familiar a uno irreconocible. Los métodos de compresión para imágenes de tonos continuos a menudo no manejan los bordes agudos muy bien, por lo que se necesitan métodos especiales para la compresión eficiente de estas imágenes. Tenga en cuenta que una imagen de tonos discretos puede tener una elevada redundancia, ya que el mismo carácter o patrón puede aparecer muchas veces en la imagen. La Figura 4.57 es un ejemplo típico de una imagen de tonos discretos.

5. Una imagen *cartoon-like* (de *dibujos animados*). Ésta es una imagen en color que se compone de áreas uniformes. Cada área tiene un color uniforme, salvo las zonas adyacentes, que pueden tener colores muy diferentes. Esta característica puede ser explotada para obtener una excelente compresión.

Que una imagen sea tratada como discreta o continua suele determinarse por la profundidad de los datos. Sin embargo, es posible forzar a una imagen a ser continua incluso si se ajustara a la categoría discreta. (De `www.genaware.com`)

[2]Este párrafo han sido modificado con respecto al original, siguiendo la recomendación de Miguel Hernández Cabronero, que se prestó a revisar la traducción.

Es intuitivamente claro que cada tipo de imagen puede presentar redundancia, pero son redundantes de diferentes maneras. Ésta es la razón por la que cualquier método de compresión dado, no puede funcionar bien para todas las imágenes, y por qué son necesarios diferentes métodos para comprimir distintos tipos de imagen. Hay métodos de compresión para imágenes binivel, para las de tono continuo, y para las de tono discreto. También hay métodos que intentan descomponer una imagen en partes con tonos continuos y partes con tonos discretos, y comprimir cada una por separado.

4.1. Introducción

Las computadoras modernas utilizan gráficos de forma extensa. Los sistemas operativos basados en ventanas muestran el directorio de archivos del disco de forma gráfica. El progreso de muchas operaciones del sistema, tales como descargar un archivo, también se pueden mostrar gráficamente. Muchas aplicaciones proporcionan una interfaz gráfica de usuario (GUI o *Graphical User Interface*), lo que hace que sea más fácil utilizar el programa e interpretar los resultados mostrados. Los gráficos de ordenador se utilizan en muchas áreas de la vida cotidiana para convertir muchos tipos de información compleja en imágenes. En consecuencia, las imágenes son importantes, ¡pero tienden a ser grandes! El hardware de hoy puede mostrar muchos colores, por lo que es común tener un píxel representado internamente como un número de 24 bits, donde los porcentajes de rojo, verde y azul ocupan 8 bits cada uno. Estos píxeles de 24 bits pueden especificar uno de los $2^{24} \approx 16{,}78$ millones de colores. Como resultado, una imagen con una resolución de 512×512 que consta de tales píxeles ocupa 786 432 bytes. En una resolución de 1024×1024, llega a ser cuatro veces más grande, requiriendo 3 145 728 bytes. Los videos son también de uso común en los ordenadores, elaborando imágenes incluso más grandes. Ésta es la razón por la que la compresión de imágenes es tan importante. Un característica a destacar en la compresión de imágenes es que puede ser con pérdidas. Una imagen, después de todo, existe para que la gente la vea; así, cuando se comprime, es aceptable perder características de la imagen a la que el ojo no es sensible. Ésta es una de las ideas principales subyacente en muchos métodos de compresión con pérdida que se describen en este capítulo.

En general, la información se puede comprimir si es redundante. Se ha mencionado varias veces que la compresión de datos equivale a reducir o eliminar la redundancia en los datos. Con la compresión con pérdida, sin embargo, tenemos un nuevo concepto, denominado compresión mediante la eliminación de *irrelevancia*. Una imagen puede ser comprimida con pérdidas quitando información irrelevante incluso si la imagen original no tiene ningún redundancia.

◇ **Ejercicio 4.1 (sol. en pág. 1077):** Al parecer, una imagen sin redundancia es siempre aleatoria (y por lo tanto, carente de interés). ¿No es así?

La idea de perder información de la imagen se vuelve más aceptable si tenemos en cuenta cómo se crean las imágenes digitales. He aquí tres ejemplos: (1) Una imagen de la vida real puede ser escaneada de una fotografía o una pintura y digitalizada (convertida en píxeles). (2) Una imagen puede ser grabada por una cámara digital que crea píxeles y los almacena directamente en la memoria. (3) Una imagen puede ser dibujada en la pantalla por medio de un programa de gráficos. En todos estos casos, se pierde alguna información cuando la imagen se digitaliza. El hecho de que el espectador está dispuesto a aceptar ésta pérdida sugiere que la pérdida posterior de la información podría ser tolerable si se hace correctamente.

(La digitalización de una imagen implica dos pasos: *muestreo* y *cuantificación*. El muestreo de una imagen es el proceso de dividir la imagen original bidimensional en pequeñas regiones: píxeles. La cuantificación es el proceso de asignar un valor entero a cada píxel. Nótese que la digitalización del sonido implica los mismos dos pasos, con la diferencia de que el sonido es unidimensional.)

Presentamos un proceso simple que puede ser empleado para determinar cualitativamente la cantidad de datos perdidos en una imagen comprimida: Dada una imagen A, (1) ésta se comprime en B,

(2) se descomprime B en C, y (3) se resta $D = C - A$. Si A se comprimió sin ninguna pérdida y se descomprime correctamente, entonces C debe ser idéntica a A y la imagen D debe ser de color blanco uniforme. Cuantos más datos se hayan perdido en la compresión, más lejos estará D del color blanco uniforme.

¿Cómo debe ser una imagen comprimida? Las técnicas de compresión discutidas en capítulos anteriores son: RLE, cuantificación escalar, métodos estadísticos, y métodos basados en diccionarios. Por sí mismos, no son muy satisfactorios para imágenes en color o en escala de grises (aunque se pueden utilizar en combinación con otros métodos). He aquí por qué:

La Sección 1.4.1 muestra cómo puede ser utilizado RLE para (sin pérdidas o con pérdidas) la compresión de una imagen. Éste es sencillo, y se usa en ciertas partes de JPEG, especialmente por su modo sin pérdidas. En general, sin embargo, los otros principios utilizados por JPEG producen una compresión mucho mejor que la proporcionada sólo por RLE. La compresión de faxes (Sección 2.13) utiliza RLE combinado con la codificación de Huffman y obtiene buenos resultados, pero sólo para las imágenes de dos niveles.

La cuantificación escalar ha sido mencionada en la Sección 1.6. Se puede usar para comprimir imágenes, pero su rendimiento es mediocre. Imagine una imagen con 8 bits por píxel. Puede ser comprimida con cuantificación escalar eliminando los cuatro bits menos significativos de cada píxel. Esto produce una razón de compresión de 0,5 —lo que no es muy impresionante—, y al mismo tiempo reduce el número de colores (o escala de grises) de 256 a sólo 16. Tal reducción no sólo degrada la calidad general de la imagen reconstruida, sino que también puede crear bandas de diferentes colores, un efecto perceptible y molesto que se ilustra aquí.

Imagine una fila de 12 píxeles de colores similares, que van desde 202 hasta 215. En notación binaria, estos valores son:

$$11010111\,11010110\,11010101\,11010011\,11010010\,11010001$$
$$11001111\,11001110\,11001101\,11001100\,11001011\,11001010.$$

La cuantificación resultante produce los siguientes 12 valores de 4 bits:

$$1101\,1101\,1101\,1101\,1101\,1101\,1100\,1100\,1100\,1100\,1100\,1100,$$

que se reconstruirán como los 12 píxeles:

$$11010000\,11010000\,11010000\,11010000\,11010000\,11010000$$
$$11000000\,11000000\,11000000\,11000000\,11000000\,11000000.$$

Los primeros seis píxeles de la fila ahora tienen el valor $11010000_2 = 208$, mientras que los siguientes seis píxeles son $11000000_2 = 192$. Si las filas adyacentes, tienen píxeles similares, las primeras seis columnas formarán una banda, claramente diferente de la banda formada por las seis columnas siguientes. El efecto de esta banda —o contorno— es muy perceptible para el ojo, ya que nuestros ojos son sensibles a los bordes y los cambios bruscos en una imagen.

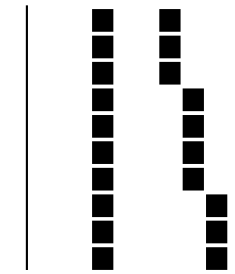
Una forma de eliminar este efecto se llama *mejora de la cuantificación de la escala de grises* (IGS o *Improved GrayScale quantization*). Funciona mediante la adición a cada píxel de un número aleatorio generado a partir de los cuatro bits del extremo derecho de los píxeles previos. La Sección 4.2.1 muestra que los bits menos significativos de un píxel son casi aleatorios, por lo que IGS funciona añadiendo a cada píxel una aleatoriedad que depende de la vecindad del píxel.

El método mantiene una variable de 8 bits, que se denota `rsm`, que está inicialmente a cero. Para cada píxel P de 8 bits que se cuantifica (excepto el primero), el método IGS realiza los siguientes pasos:

1. Guarda en `rsm` la suma de los ocho bits de P más los cuatro bits de más a la derecha de `rsm`. Sin embargo, si P tiene la forma $1111xxxx$, establece `rsm` en P.

Píxel	Valor	`rsm`	Valor comprimido
1	1010 0110	0000 0000	1010
2	1101 0010	1101 0010	1101
3	1011 0101	1011 0111	1011
4	1001 1100	1010 0011	1010
5	1111 0100	1111 0100	1111
6	1011 0011	1011 0111	1011

Tabla 4.1: Ilustración del método IGS.



(a) (b) (c)

En (a) se muestra una regla vertical ideal. En (b), se supone que la regla ha sido digitalizada perfectamente en diez píxeles, en vertical. Sin embargo, si la imagen se coloca en el escáner ligeramente inclinada, la exploración puede ser embargo, si la imagen se coloca en el escáner imperfecta, y los píxeles resultantes podrían parecerse a los de (c).

Figura 4.2: Digitalización perfecta e imperfecta.

2. Escribe los cuatro bits de la izquierda de `rsm` en la secuencia de datos comprimidos. Este es el valor comprimido de P. IGS no es, por tanto, exactamente un método de cuantificación, sino una variación de la cuantización escalar.

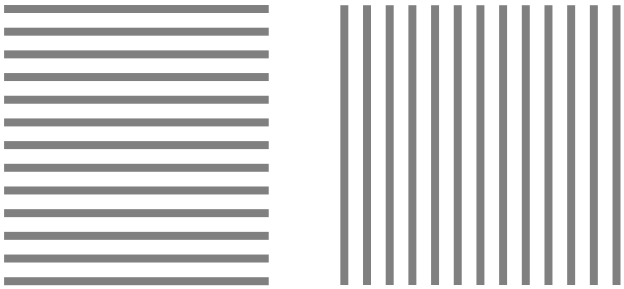
El primer píxel se cuantifica en la forma habitual, desechando sus cuatro bits de más a la derecha. La Tabla 4.1 ilustra el funcionamiento de IGS.

La cuantificación vectorial se puede utilizar con más éxito para comprimir imágenes. Se discute en la Sección 4.14.

Los métodos estadísticos funcionan mejor cuando los símbolos que se están comprimiendo tienen probabilidades diferentes. Un flujo de entrada, donde todos los símbolos tienen la misma probabilidad no se comprimirá, a pesar de no ser aleatorio. Resulta que en una imagen en tonos continuos o en escala de grises, los diferentes colores o tonos de gris, a menudo pueden tener más o menos la misma probabilidad. Por ésto, los métodos estadísticos no son una buena opción para comprimir tales imágenes, y se hacen necesarios nuevos enfoques. Las imágenes con discontinuidades de color, donde los píxeles adyacentes tienen colores muy diferentes, se comprimen mejor con métodos estadísticos; pero no es fácil predecir, sólo mirando a una imagen, si tiene suficientes discontinuidades de color.

Los métodos de compresión basados en diccionarios también tienden a no tener éxito en el tratamiento de imágenes de tonos continuos. Dicha imagen contiene típicamente píxeles adyacentes con colores similares, pero no contiene patrones repetitivos. Incluso una imagen que contiene patrones repetidos, tales como líneas verticales, puede perderlos cuando se digitaliza. Una línea vertical en la imagen original puede llegar a ser ligeramente inclinada cuando la imagen se digitaliza (Figura 4.2), por lo que los píxeles de una fila escaneada pueden llegar a tener colores ligeramente diferentes de los de las filas adyacentes, dando lugar a un diccionario con cadenas cortas. (Este problema también puede afectar a los bordes curvos.)

Otro problema con la compresión de diccionario de imágenes radica en que tales métodos escanean



(a) (b)

Figura 4.3: Compresión de diccionario de líneas paralelas.

Figura 4.4: Valores y diferencias de 24 píxeles adyacentes.

la imagen fila por fila y, por lo tanto, pueden pasar por alto las correlaciones entre los píxeles verticales. Un ejemplo son las dos sencillas imágenes de la Figura 4.3a,b. Guardando ambas en GIF89 —un formato de archivo de gráficos basado en diccionarios (Sección 3.19)— han proporcionado tamaños de archivo de 1053 y 1527 bytes, respectivamente, en el ordenador del autor.

Los métodos tradicionales son, por lo tanto, insatisfactorios para la compresión de imágenes, por lo que este capítulo trata nuevos enfoques. Todos son diferentes, pero eliminan la redundancia de una imagen utilizando el siguiente principio (véase también la Sección 1.4):

Principio de la compresión imágenes. Si seleccionamos un píxel de la imagen al azar, hay una buena probabilidad de que sus vecinos tengan el mismo color o colores muy similares.

La compresión de imágenes se basa, por tanto, en el hecho de que píxeles vecinos están *altamente correlacionados*. Esta correlación también se llama *redundancia espacial*.

A continuación hay un sencillo ejemplo que ilustra lo que puede hacerse con píxeles correlacionados. La siguiente secuencia de valores da las intensidades de 24 píxeles adyacentes en una fila de una imagen de tonos continuos:

$$12, 17, 14, 19, 21, 26, 23, 29, 41, 38, 31, 44, 46, 57, 53, 50, 60, 58, 55, 54, 52, 51, 56, 60.$$

Sólo dos de los 24 píxeles son idénticos. Su valor promedio es de 40,3. Restando pares de píxeles adyacentes se genera la secuencia:

$$12, 5, -3, 5, 2, 4, -3, 6, 11, -3, -7, 13, 4, 11, -4, -3, 10, -2, -3, 1, -2, -1, 5, 4$$

Las dos secuencias se ilustran en la Figura 4.4.

La secuencia de valores diferencia tiene tres propiedades que ilustran su compresión potencial: (1) Los valores diferencia son menores que los valores de los píxeles originales. Su promedio es 2,58. (2) Se repiten. Hay sólo 15 valores diferencia distintos, por lo que —en principio— pueden ser codificados

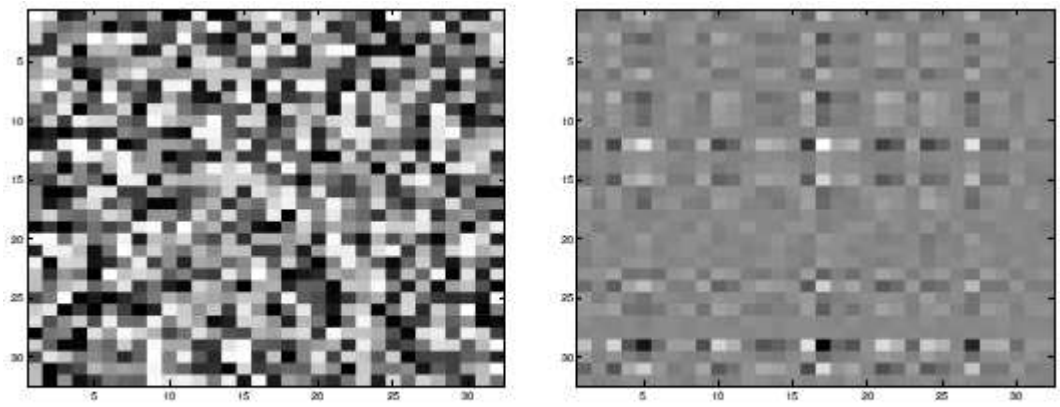


Figura 4.5: Mapas de (a) una matriz aleatoria y (b) su inversa.

```
n=32; a=rand(n); imagesc(a); colormap(gray)
b=inv(a); imagesc(b)
```

Código Matlab para la Figura 4.5.

con cuatro bits cada uno. (3) Están *descorrelacionados*: los valores diferencia adyacentes tienden a ser diferentes. Ésto puede verse restándolos, lo que produce la secuencia de 24 segundas diferencias:

$$12, -7, -8, 8, -3, 2, -7, 9, 5, -14, -4, 20, -11, 7, -15, 1, 13, -12, -1, 4, -3, 1, 6, 1.$$

Éstas son más grandes que las propias diferencias.

La Figura 4.5 proporciona otro ejemplo del significado de las palabras "cantidades correlacionadas". Se construye una matriz A de 32×32 con números aleatorios, y sus elementos aparecen en la parte (a) como cuadrados sombreados. La naturaleza aleatoria de los elementos es evidente. La matriz se invierte y se almacena en B, que se muestra en la parte (b). Esta vez, los 32×32 cuadrados parecen más estructurados. Un cálculo directo utilizando la Ecuación (4.1) muestra que la correlación cruzada entre las dos filas de A es 0,0412, mientras que la correlación cruzada entre las dos primeras filas de B es $-0{,}9831$. Los elementos de B están correlacionados, ya que todos dependen de los elementos de A

$$R = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{\sqrt{\left[n\sum x_i^2 - \left(\sum x_i\right)^2\right]\left[n\sum y_i^2 - \left(\sum y_i\right)^2\right]}}. \tag{4.1}$$

◇ **Ejercicio 4.2 (sol. en pág. 1077):** Utilícese software matemático para ilustrar las matrices de covarianza de (1) una matriz con valores correlacionados y (2) una matriz con valores descorrelacionados.

Una vez que el concepto de las cantidades correlacionadas nos es familiar, empezamos a buscar una prueba de correlación. Dada una matriz M, se necesita una prueba estadística para determinar si sus elementos están correlacionados o no. El test se basa en el concepto estadístico de covarianza. Si los

elementos de M están descorrelacionados (i.e., son independientes), entonces la covarianza de dos filas diferentes cualesquiera y dos columnas diferentes cualesquiera de M será igual a cero (la covarianza de una fila o de una columna con sí misma es siempre 1). Como resultado, la matriz de covarianza de M (ya sea la covarianza de filas, ya sea la de columnas) será diagonal. Si la matriz de covarianza de M no es diagonal, entonces los elementos de M están correlacionados. Los conceptos estadísticos de varianza, covarianza y correlación se discuten en cualquier texto relativo a las estadísticas.

El principio de la compresión de imágenes tiene otro aspecto. Sabemos por experiencia que la luminosidad de los píxeles vecinos también está correlacionada. Dos píxeles adyacentes pueden tener diferentes colores. Uno puede ser en su mayor parte de color rojo, y el otro puede ser predominantemente de color verde. Sin embargo, si el componente de rojo de la primera es brillante, el componente verde de su vecino , en la mayoría de los casos, también será brillante. Esta propiedad puede ser explotada en la conversión de las representaciones de los píxeles en RGB a otros tres componentes, uno de los cuales es el brillo, y los otros dos representan el color. Un formato de este tipo (o *espacio de color*) es YCbCr, donde Y (el componente de "luminancia") representa el brillo de un píxel, y Cb y Cr define su color. Este formato se discute en la Sección 4.8.1, pero su ventaja es fácil de comprender. El ojo es sensible a pequeños cambios en el brillo, pero no a pequeños cambios en el color. Por lo tanto, la pérdida de información en los componentes Cb y Cr comprime la imagen al tiempo que introduce distorsiones a las que el ojo no es sensible. La pérdida de información en la componente Y, por otro lado, es muy sensible al ojo.

4.2. Métodos para la compresión de imágenes

Un método de compresión de imágenes está normalmente diseñado para un tipo específico de imagen, y en esta sección se enumeran varios enfoques para la compresión de imágenes de diferentes tipos. Sólo se discuten aquí los principios generales; los métodos específicos se describen en el resto del este capítulo.

Método 1: Éste es apropiado para las imágenes binivel. Un píxel en tal imagen está representado por un bit. La aplicación del principio de la compresión de imágenes a una imagen binivel significa, por lo tanto, que los vecinos inmediatos de un píxel P tienden a ser *idénticos* a P. Así, tiene sentido utilizar la codificación *run-length* (RLE) para comprimir una imagen de este tipo. Un método de compresión para dicha imagen puede escanearla en zigzag, de arriba abajo[3] (fila por fila) y calcular las longitudes de las rachas (*runs*) de píxeles blancos y negros. Las longitudes se codifican mediante códigos de tamaño variable (prefijos) y se escriben en la secuencia de datos comprimidos. Un ejemplo de tal método es la compresión de faxes, Sección 2.13.

Cabe destacar que ésto es sólo una aproximación a la compresión de imágenes binivel. Los detalles de los métodos específicos varían. Por ejemplo, un método puede escanear la imagen columna por columna o en zigzag (Figura 1.8b), puede convertir la imagen a un quadtree (Sección 4.30), o puede escanearla zona por zona utilizando una curva de relleno de espacio (Sección 4.32).

Método 2: También para las imágenes binivel. El principio de la compresión de imágenes nos dice que los vecinos de un píxel tienden a ser similares al mismo. Podemos extender este principio y concluir que si el píxel actual tiene un color c (donde c es o bien negro, o bien blanco), entonces los píxeles del mismo color vistos en el pasado (y también aquellos que aparecerán en el futuro) tienden a tener los mismos vecinos inmediatos.

Este enfoque observa los n vecinos más cercanos del píxel actual y los considera un número de n bits. Este número es el *contexto* del píxel. En principio, pueden ser contextos de 2^n bits, pero debido a la redundancia de la imagen esperamos que se distribuyan de manera no uniforme. Algunos contextos deberían ser frecuentes, mientras que otros raramente aparecerán.

[3]Raster order.

El codificador cuenta cuántas veces se ha encontrado ya cada contexto para un píxel de color c, y asigna probabilidades a los contextos en consecuencia. Si el píxel actual tiene un color c y su contexto tiene una probabilidad p, el codificador puede utilizar la codificación aritmética adaptativa para codificar el píxel con esa probabilidad. Este enfoque es utilizado por JBIG (Sección 4.11).

A continuación, consideramos las imágenes en escala de grises. Un píxel en dicha imagen está representado por n bits y puede tener uno de 2^n valores. La aplicación del principio de la compresión de imágenes a una imagen en escala de grises implica que los vecinos inmediatos de un píxel P tienden a ser similares a P, pero no son necesariamente idénticos. En consecuencia, RLE no debe ser utilizado para comprimir tales imagenes. En su lugar, se comentan otros dos métodos.

Método 3: Separar la imagen en escala de grises en n imágenes binivel y comprimir cada una de ellas con RLE y códigos prefijo. El principio de la compresión de imágenes parece implicar intuitivamente que dos píxeles adyacentes que son similares en la imagen en escala de grises será idéntico en la mayoría de las n imágenes binivel. Ésto, sin embargo, no es cierto, como deja en claro el siguiente ejemplo. Imagine una imagen en escala de grises con $n = 4$ (i.e., 4 bits por píxel ó 16 tonos de gris). La imagen se puede separar en cuatro imágenes binivel. Si dos píxeles adyacentes en la imagen original en escala de grises tienen valores de 0000 y 0001, entonces son similares. Ellos son también idénticos en tres de las cuatro imágenes binivel. Sin embargo, dos píxeles adyacentes con valores de 0111 y 1000 son también similares en la imagen en escala de grises (sus valores son 7 y 8, respectivamente), pero difieren en las cuatro imágenes binivel.

Este problema se produce porque los códigos binarios de los números enteros adyacentes pueden diferir en varios bits. Los códigos binarios de 0 y 1 difieren en 1 bit, los de 1 y 2 difieren en dos bits, y los de 7 y 8 difieren en cuatro bits. La solución es diseñar códigos binarios especiales, de tal manera que los códigos de los enteros consecutivos i e $i+1$ difieran en un sólo bit. Un ejemplo de dicho código son los *reflected Gray codes* (*códigos de Gray reflejados*) de la Sección 4.2.1.

Método 4: Utilizar el contexto de un píxel para predecir su valor. El contexto de un píxel son los valores de algunos de sus vecinos. Podemos examinar algunos vecinos de un píxel P, calcular un promedio A de una de sus valores, y predecir que P tendrá un valor de A. El principio de la compresión de imágenes nos dice que nuestra predicción será correcta en la mayoría de los casos; casi correcta, en muchos casos; y totalmente equivocada, en unos pocos casos. Podemos decir que el valor pronosticado del píxel P representa la información redundante en P. Ahora calculamos la diferencia:

$$\Delta \stackrel{\text{def}}{=} P - A,$$

y asignar códigos (prefijo) de tamaño variable a los diferentes valores de Δ tales que a los valores pequeños (que esperamos sean frecuentes) se les asignen los códigos cortos y a los valores grandes (que esperamos que aparezcan raramente) se les asignen códigos largos. Si P puede tener los valores, de 0 a $m-1$, entonces los valores de Δ están en el intervalo $[-(m-1),(m-1)]$, y el número de códigos necesarios es $2(m-1)+1$ ó $2m-1$.

Los experimentos con un gran número de imágenes sugieren que los valores de Δ tienden a distribuirse de acuerdo con la distribución de Laplace (Figura 4.128b —pág. 430—). Un método de compresión puede, por tanto, utilizar esta distribución para asignar una probabilidad a cada valor de Δ, y usar la codificación aritmética para codificar los valores Δ de manera muy eficiente. Éste es el principio del método MLP (Sección 4.21).

El contexto de un píxel puede consistir en sólo uno o dos de sus vecinos inmediatos. Sin embargo, los mejores resultados pueden obtenerse cuando varios píxeles vecinos están incluidos en el contexto. El promedio de A, en tal caso, debe ser ponderado, asignando a los vecinos más cercanos mayores pesos (véase, por ejemplo, la Tabla 4.126 —pág. 429—). Otra consideración importante es el decodificador. A fin de que pueda decodificar la imagen, debe ser capaz de calcular el contexto de cada píxel. Ésto significa que el contexto debe emplear solamente píxeles que ya han sido codificados. Si la imagen es

escaneada en zig zag, de arriba abajo (orden *raster*), el contexto debe incluir sólo los píxeles situados por encima del píxel actual o en la misma fila y a su izquierda.

Método 5: Transformar los valores de los píxeles y codificar los valores transformados. El concepto de una transformación, así como las transformaciones más importantes que se utilizan en la compresión de imágenes, se discuten en la Sección 4.4. El Capítulo 5 está dedicado a la transformada wavelet. Recuérdese que la compresión se logra mediante la reducción o eliminación de la redundancia. La redundancia de una imagen está causada por la correlación entre los píxeles, por lo que la transformación de los píxeles a una representación donde estén descorrelacionados elimina la redundancia. También es posible pensar en una transformación en términos de la entropía de la imagen. En una imagen altamente correlacionada, los píxeles tienden a tener valores equiprobables, lo que produce una entropía máxima. Si los píxeles transformados están descorrelacionados, ciertos valores de píxel serán comunes, lo que proporciona grandes probabilidades, mientras que otros serán poco frecuentes. Ésto da lugar a pequeñas entropías. La cuantificación de los valores transformados puede producir una compresión con pérdida, pero eficiente de las imágenes. Queremos que los valores transformados sean independientes, porque la codificación independiente de los valores hace que sea más sencilla la construcción de un modelo estadístico.

Consideramos ahora las imágenes en color. Un píxel en dicha imagen está formada por tres componentes de color, a saber: rojo, verde y azul. La mayoría de las imágenes en color están formadas, o bien por tonos continuos o bien por tonos discretos.

Método 6: El principio de este enfoque es separar una imagen en color de tonos continuos en tres imágenes en escala de grises y comprimir cada una de las tres por separado, utilizando los métodos 3, 4 ó 5.

Para una imagen de tonos continuos, el principio de la compresión de imágenes implica que píxeles adyacentes sean similares, aunque tal vez no sean idénticos, en cuanto al color. Sin embargo, que los colores sean similares no significa que los valores de los píxeles también lo sean. Consideremos, por ejemplo, valores de píxel de 12 bits, en los que cada componente de color se expresa con cuatro bits. Así, los 12 bits 1000|0100|0000 representan un píxel cuyo color es una mezcla de ocho unidades de color rojo (aproximadamente el 50 %, ya que el máximo es de 15 unidades), cuatro unidades de verde (aproximadamente el 25 %) y nada de azul. Ahora imagine dos píxeles adyacentes con valores 0011|0101|0011 y 0010|0101|0011. Ellos tienen colores similares, ya que sólo difieren en sus componentes de rojo, y sólo por una unidad. Sin embargo, cuando se consideran como números de 12 bits, los dos números 001101010011 y 001001010011 son muy distintos, ya que difieren en uno de sus bits más significativos.

Una característica importante de este método es utilizar una representación del color de crominancia y luminancia, en lugar de la más común RGB. Los conceptos de luminancia y crominancia se discuten en la Sección 4.8.1 y en [Salomon 99]. La ventaja de la representación del color en luminancia y crominancia está en que el ojo es sensible a los pequeños cambios en la luminancia pero no en la crominancia. Ésto permite pérdidas de datos considerables en los componentes de crominancia, al tiempo que permite decodificar la imagen sin una pérdida significativa visible de calidad.

Método 7: Se necesita un enfoque diferente para imágenes de tonos discretos. Recordemos que tales imágenes contienen regiones uniformes, y una región puede aparecer varias veces en la imagen. Un buen ejemplo es un volcado de pantalla. Tal imagen se compone de texto e iconos. Cada carácter de texto y cada icono es una región, y cualquier región puede aparecer varias veces en la imagen. Una posible forma de comprimir una imagen así consiste en analizarla, identificando las regiones, y encontrando las que estén repetidas. Si una región B es idéntica a una región A —ya encontrada anteriormente—, entonces B puede comprirse escribiendo un puntero a A en la secuencia de datos comprimidos. El método de descomposición en bloques (FABD, Sección 4.28) es un ejemplo de cómo se puede implementar este enfoque.

Método 8: La partición de la imagen en partes (superpuestas o no) y su compresión mediante el

procesamiento de cada parte, una a una. Supongamos que la siguiente parte de la imagen sin procesar es la parte número 15. Se trata de hacerla coincidir con las partes 1–14, que ya han sido procesadas. Si la parte 15 puede ser expresada, por ejemplo, como una combinación de las partes 5 (en escala) y 11 (rotada), entonces sólo necesitan guardarse los pocos números que especifican dicha combinación, y la parte 15 puede ser desechada. Si la parte 15 no puede expresarse como una combinación de partes ya procesadas, se declara procesada y se guarda en bruto (formato raw).

Este enfoque es la base de diversos métodos *fractales* para la compresión de imágenes. Se aplica el principio de la compresión de imágenes a partes de la imagen en vez de a píxeles individuales. Aplicado de esta manera, el principio nos dice que "interesantes" imágenes (i.e., aquellas que se pueden comprimir, en la práctica) tienen una cierta cantidad de *auto-similitud*. Partes de la imagen son idénticas o similares a la imagen completa o a otras partes.

Los métodos de compresión de imágenes no se limitan a estos enfoques básicos. Este libro analiza los métodos que utilizan los conceptos de árboles de contexto, modelos de Markov (Sección 8.8), y wavelets, entre otros. Además, debe mencionarse el concepto de *compresión progresiva imágenes* (Sección 4.10), ya que añade otra dimensión al campo de la compresión de imágenes.

4.2.1. Códigos de Gray

Un método de compresión de imágenes que ha sido desarrollado específicamente para un determinado tipo imagen, a veces puede ser utilizado para otros tipos. Cualquier método para comprimir imágenes binivel, por ejemplo, puede ser utilizado para comprimir imágenes en escala de grises, separando los planos de bits y comprimiendo cada uno individualmente, como si se tratara de una imagen de dos niveles. Imaginemos, por ejemplo, una imagen con 16 valores en escala de grises . Cada píxel está definido por cuatro bits, por lo que la imagen se puede separar en cuatro imágenes binivel. El problema con este enfoque, es que viola el principio general de la compresión de imágenes. Imagínese dos píxeles adyacentes de 4 bits con valores: $7 = 0111_2$ y $8 = 1000_2$. Estos píxeles tienen valores cercanos, pero cuando son separados en cuatro planos de bits, los píxeles resultantes de 1 bit ¡son diferentes en cada plano de bits! Ésto sucede porque las representaciones binarias de los números enteros consecutivos 7 y 8 difieren en las cuatro posiciones de bit. Con el fin de aplicar cualquier método de compresión binivel a imágenes en escala de grises, se necesita una representación binaria de los enteros donde los números consecutivos tengan códigos que difieran en sólo un bit. Tal representación existe y se llama *código de Gray reflejado* (RGC o *Reflected Gray Code*). Este código es fácil de generar mediante la siguiente construcción recursiva:

Se comienza con los dos códigos de 1 bit $(0, 1)$. Se construyen dos conjuntos de códigos de 2 bits, duplicando $(0, 1)$ y añadiendo al conjunto original —o bien por la izquierda, o bien por la derecha— primero un cero; luego un uno. El resultado es: $(00, 01)$ y $(10, 11)$. Ahora se invierte (refleja) el segundo conjunto[4], y se concatenan ambos. El resultado es el RGC de 2 bits $(00, 01, 11, 10)$; un código binario formado por los enteros del 0 al 3, donde los códigos consecutivos difieren en exactamente un bit. Aplicando la regla de nuevo, se generan los dos conjuntos $(000, 001, 011, 010)$ y $(110, 111, 101, 100)$, que se concatenan para formar el RGC de 3 bits. Obsérvese que los códigos primero y último de cualquier RGC también difieren en un bit. Estos son los tres primeros pasos para el cálculo del RGC de 4 bits:

$$\begin{aligned} \text{Añadir un cero } & (0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100), \\ \text{Añadir un uno } & (1000, 1001, 1011, 1010, 1110, 1111, 1101, 1100), \\ \text{y reflejar } & (1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000). \end{aligned}$$

La Tabla 4.6 muestra cómo cambian los bits individuales cuando pasan a través de los códigos binarios de los primeros 32 enteros. Las columnas impares (43210) muestran los códigos binarios de

[4]Se forma un nuevo conjunto cogiendo los elementos del conjunto original, del último al primero.

43210	Gray	43210	Gray	43210	Gray	43210	Gray
00000	00000	0**1000**	10010	**10000**	00011	1**1000**	10001
0000**1**	00100	0100**1**	10110	1000**1**	00111	1100**1**	10101
000**10**	01100	010**10**	11110	100**10**	01111	110**10**	11101
0001**1**	01000	0101**1**	11010	1001**1**	01011	1101**1**	11001
00**100**	11000	01**100**	01010	10**100**	11011	11**100**	01001
0010**1**	11100	0110**1**	01110	1010**1**	11111	1110**1**	01101
0011**10**	10100	0111**10**	00110	1011**10**	10111	1111**10**	00101
0011**1**	10000	0111**1**	00010	1011**1**	10011	1111**1**	00001

Tabla 4.6: Primeros 32 binarios y códigos de Gray reflejados.

```
function b=rgc(a,i)
[r,c]=size(a);
b=[zeros(r,1),a; ones(r,1),flipud(a)];
if i>1, b=rgc(b,i-1); end;
```

Código en Matlab para la Tabla 4.6.

5 bits de estos enteros, con los bits del entero i que difieren de los del $i-1$, en negrita. Es fácil ver que el bit menos significativo (bit b_0) cambia continuamente, el bit b_1 cambia cada $2^1 = 2$ números, y, en general, el bit b_k cambia cada 2^k enteros. Las columnas pares (Gray) muestran uno de los posibles códigos de Gray reflejados de estos enteros. La tabla también muestra una función recursiva en Matlab para calcular el RGC. [Los parámetros son: la matriz inicial `a` $= [0; 1]$, y el número de bits deseado para el RGC menos uno, `i`. El código `[r,c]=size(a);` proporciona el tamaño de la matriz `a` en las variables `r` y `c`. En la tercera línea, se crean los códigos de Gray a partir de `a`: la mitad de ellos, añadiendo un cero a la izquierda de cada elemento de `a`; y la otra mitad, añadiendo un uno a la izquierda de los elementos de `a` (pero esta vez tomados en orden inverso —`flipud(a)`—). La última línea realiza la recursión (cuando `i>1`), o la detiene (en cualquier otro caso).]

◇ **Ejercicio 4.3 (sol. en pág. 1077):** También es posible generar el código de Gray reflejado de un entero n con la regla recursiva siguiente: realizar la operación OR-eXclusivo (XOR) entre n y una copia de sí mismo con un desplazamiento lógico (a nivel de bits) de una posición hacia la derecha. En el lenguaje de programación C, ésto se denota por `n^(n>>1)`. Utilícese esta expresión para construir una tabla similar a la Tabla 4.6.

La conclusión es, que los planos de bits (*bitplanes*) más significativos de una imagen obedecen el principio de la compresión de imágenes más que los menos significativos. Cuando píxeles adyacentes tienen valores que difieren en una unidad (tal como, p y $p+1$), lo más probable es que los bits menos significativos sean diferentes, y los más significativos sean idénticos. Cualquier método de compresión de imágenes que comprima de forma individual planos de bits, debería, por tanto, tratar los planos de bits menos significativos de forma distinta a los más significativos, o utilizar el RGC en lugar del código binario para representar los píxeles. Las Figuras 4.9, 4.10 y 4.11 (generadas por el código en Matlab de la Figura 4.7) muestran los ocho planos de bits de la conocida imagen de los loros, tanto en código binario (la columna de la izquierda) como en RGC (la columna de la derecha). Los planos de bits están numerados desde 8 (los bits de la izquierda o más significativos) hasta 1 (los bits de la derecha o menos significativos). Es obvio que el plano de bits menos significativo no muestra ninguna correlación entre los píxeles; es aleatorio o casi aleatorio tanto en binario como en RGC. Los planos de bits 2 a 5, sin embargo, presentan una mejor correlación entre los píxeles en el código de Gray. Los

```
clear;
filename='parrots128'; dim=128;
fid=fopen(filename,'r');
img=fread(fid,[dim,dim])';
mask=1;% entre 1 y 8


nimg=bitget(img,mask);
imagesc(nimg), colormap(gray)
```

Código binario

```
clear;
filename='parrots128'; dim=128;
fid=fopen(filename,'r');
img=fread(fid,[dim,dim])';
mask=1;% entre 1 y 8
a=bitshift(img,-1);
b=bitxor(img,a);
nimg=bitget(b,mask);
imagesc(nimg), colormap(gray)
```

Código de Gray

Figura 4.7: Código en Matlab para separar los planos de bits de la imagen.

43210	Gray	43210	Gray	43210	Gray	43210	Gray
00000	00000	0**1000**	01100	**10000**	11000	1**1000**	10100
0000**1**	00001	0100**1**	01101	1000**1**	11001	1100**1**	10101
000**10**	00011	010**10**	01111	100**10**	11011	110**10**	10111
0001**1**	00010	0101**1**	01110	1001**1**	11010	1101**1**	10110
00**100**	00110	01**100**	01010	10**100**	11110	11**100**	10010
0010**1**	00111	0110**1**	01011	1010**1**	11111	1110**1**	10011
001**10**	00101	011**10**	01001	101**10**	11101	111**10**	10001
0011**1**	00100	0111**1**	01000	1011**1**	11100	1111**1**	10000

Tabla 4.8: Primeros 32 binarios y códigos de Gray.

```
a=linspace(0,31,32); b=bitshift(a,-1);
b=bitxor(a,b);dec2bin(b)
```

Código en Matlab para la Tabla 4.8.

planos de bits 6 a 8 tienen un aspecto diferente en el código de Gray que en el binario, pero parecen estar altamente correlacionados en cualquiera de las dos representaciones.

La Figura 4.12 es una representación gráfica de dos versiones de los primeros 32 códigos de Gray reflejados. La parte (b) muestra los códigos de la Tabla 4.6, y la parte (c) muestra los códigos de la Tabla 4.8. A pesar de que ambos son códigos de Gray, difieren en la forma en que los bits alternan entre 0 y 1, en cada plano de bits. En la parte (b), los bits de los planos de bits más significativos alternan cuatro veces entre 0 y 1. Los del segundo plano de bits más significativo, alternan ocho veces entre 0 y 1; y los bits de los otros tres planos de bits, alternan 16, dos, y una veces entre 0 y 1. Cuando los planos de bits se separan, el plano de bits central presenta la menor correlación entre los píxeles, ya que los códigos de Gray de enteros adyacentes tienden a tener diferentes bits en este plano de bits. Los códigos de Gray se muestran en la Figura 4.12c, por otro lado, alternan más y más entre 0 y 1, a medida que se mueven de los planos de bits más significativos a los menos significativos. Los planos de bits menos significativos de esta versión proporcionan una correlación cada vez menor entre los píxeles y, por tanto, tienden a ser aleatorios. Para comparar, la Figura 4.12a muestra el código binario. Es obvio que los bits, en este código, alternan con mayor frecuencia entre 0 y 1.

◇ **Ejercicio 4.4 (sol. en pág. 1077):** Incluso una mirada superficial a los códigos de Gray de la

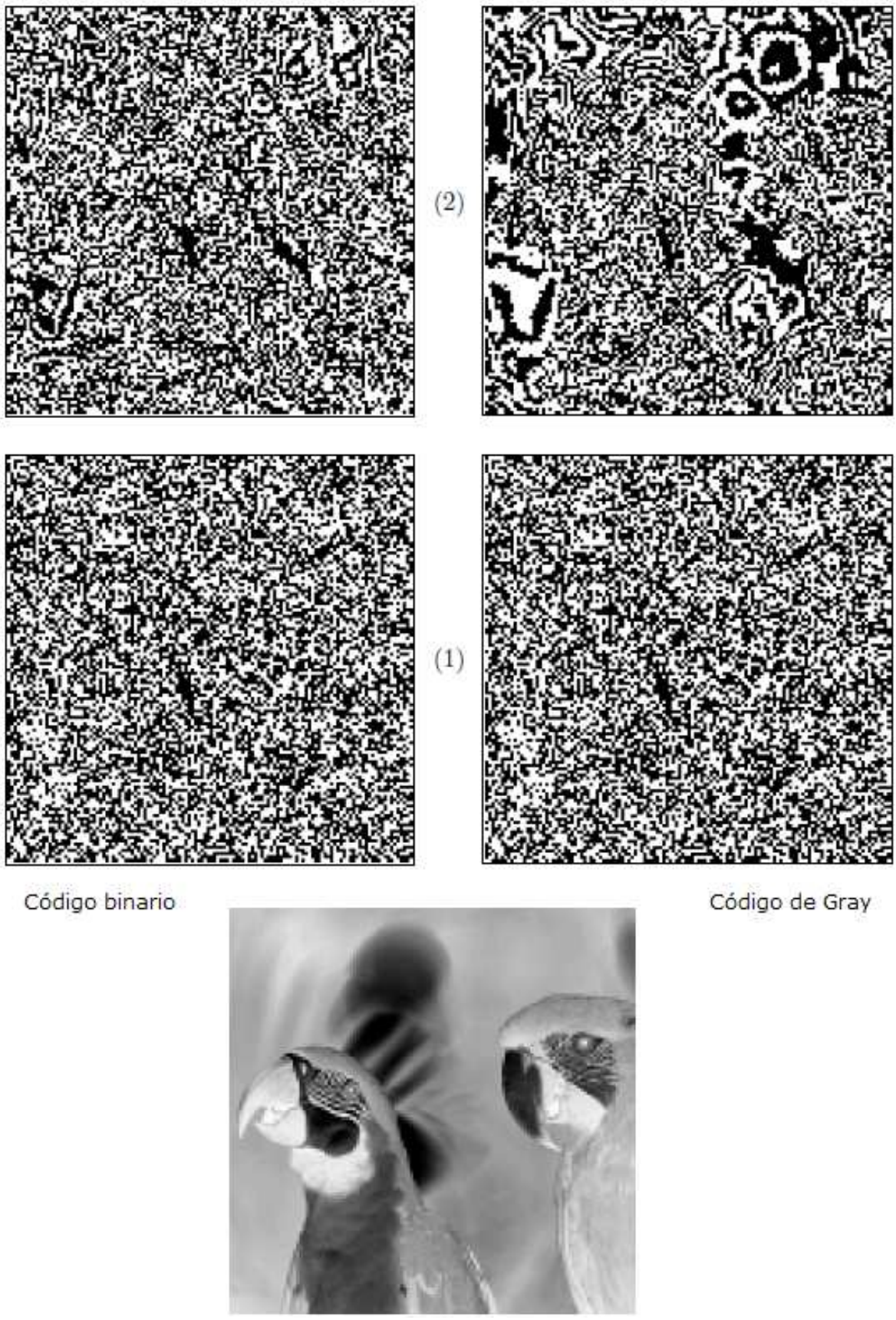


Figura 4.9: Planos de bits 1 y 2 de la imagen de los loros.

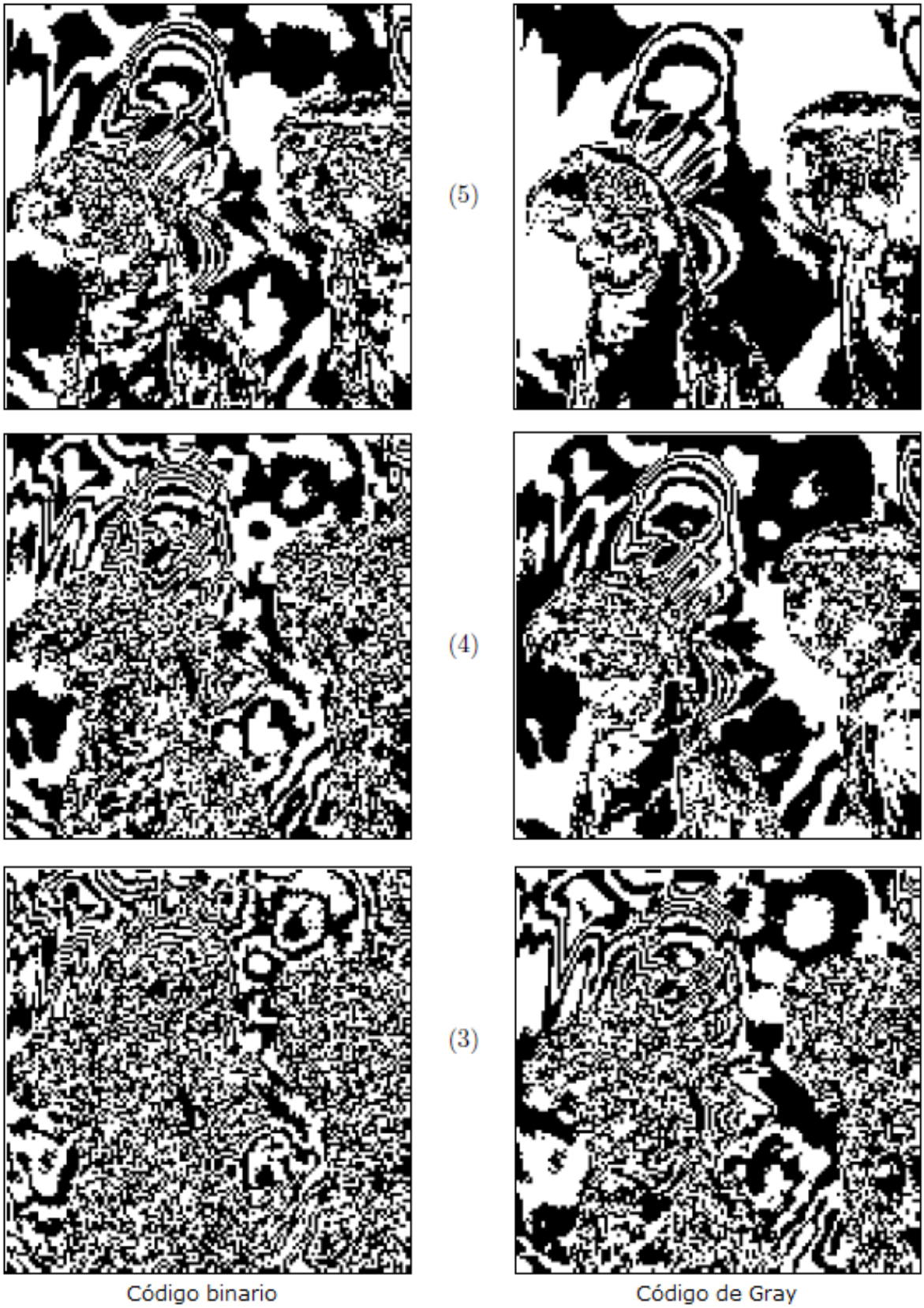


Figura 4.10: Planos de bits 3, 4 y 5 de la imagen de los loros.

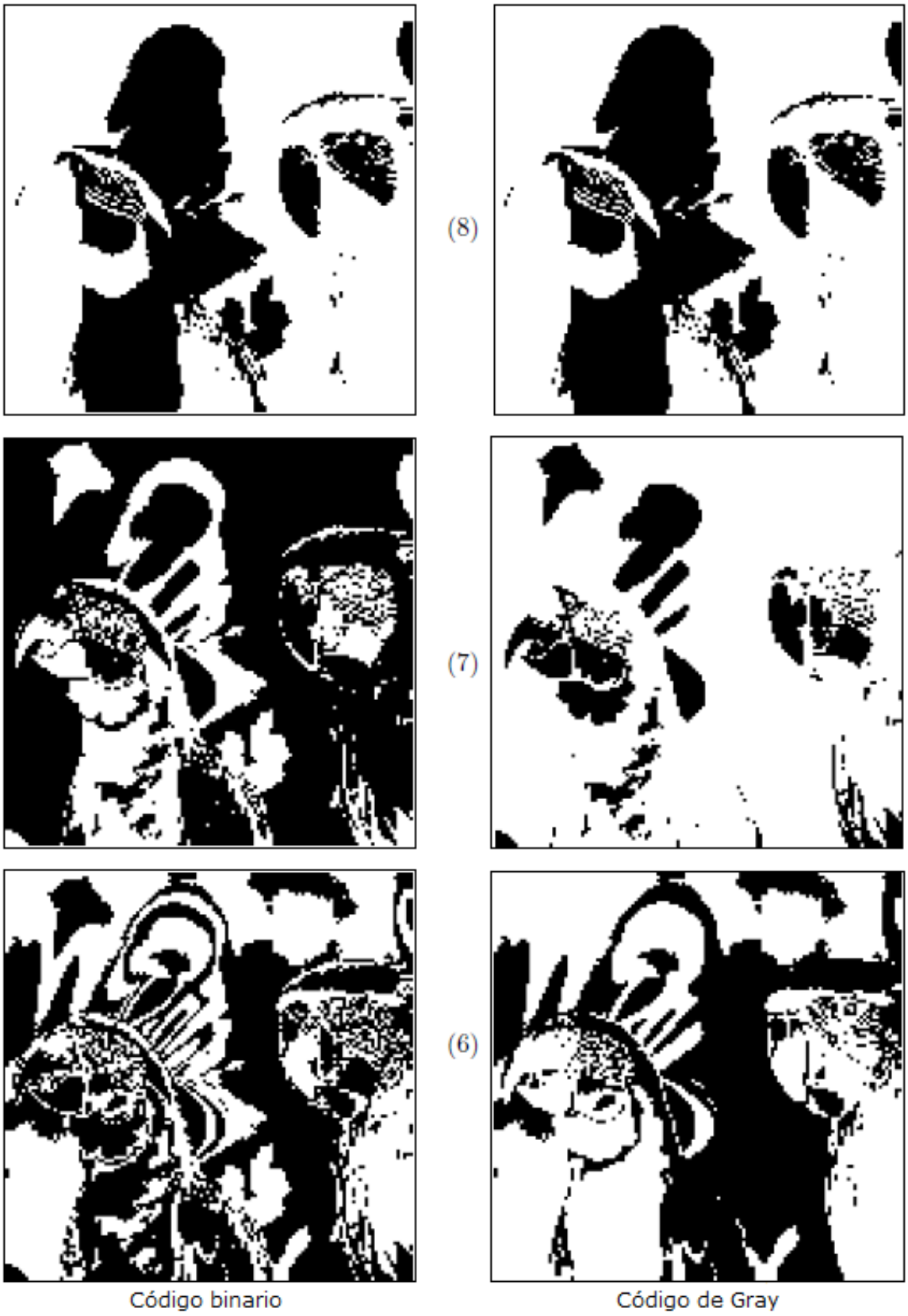


Figura 4.11: Planos de bits 6, 7 y 8 de la imagen de los loros.

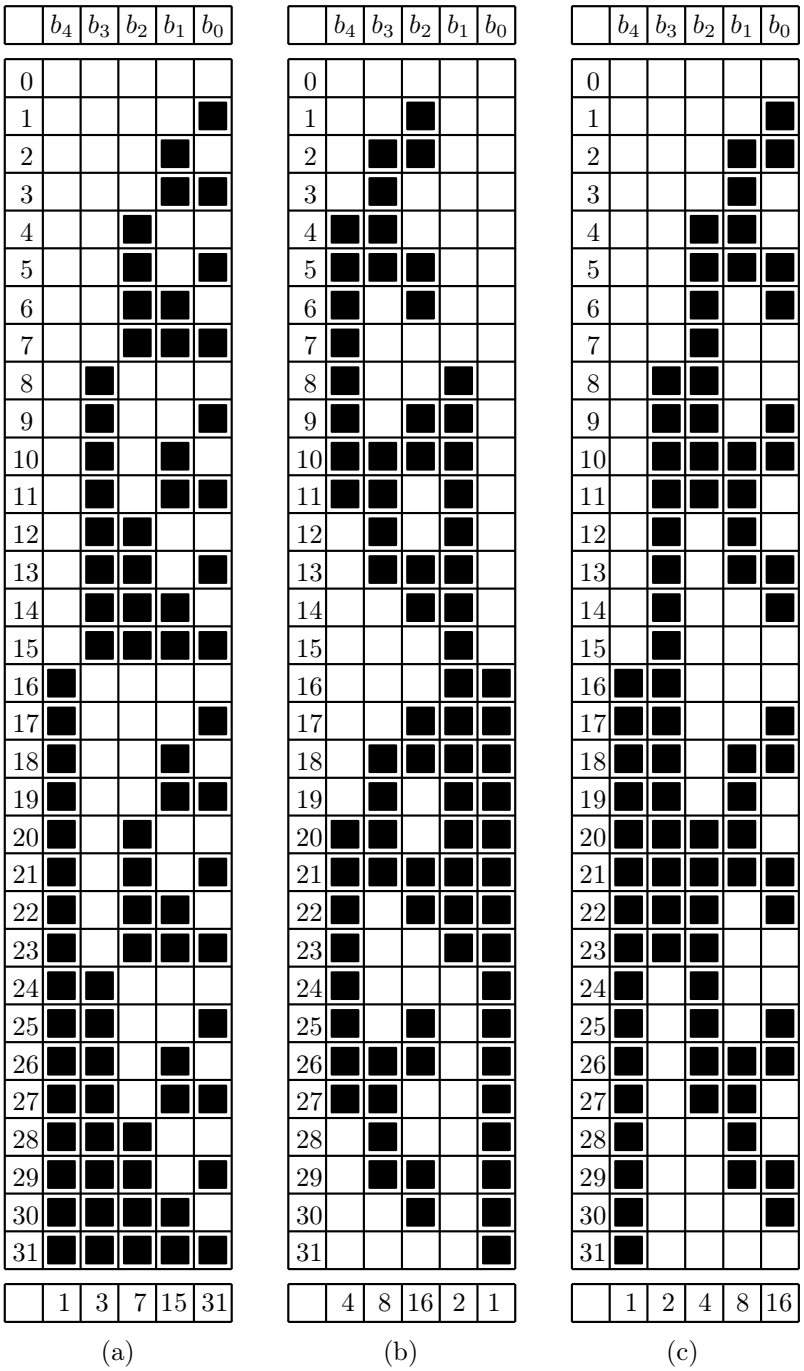


Figura 4.12: Primeros 32 binarios y códigos de Gray reflejados.

Figura 4.12c muestra que presentan una cierta regularidad. Examine cuidadosamente estos códigos e identifique dos características que pueden ser utilizadas para calcular los códigos.

◇ **Ejercicio 4.5 (sol. en pág. 1077):** La Figura 4.12 es una representación gráfica de los códigos binarios y los códigos de Gray reflejados. Encuéntrese una representación gráfica similar de los mismos códigos que ilustre el hecho de que los códigos primero y último también difieren en un bit.

Las imágenes en color proporcionan otro ejemplo del uso del mismo método de compresión en distintos tipos de imágenes. Cualquier método de compresión para imágenes en escala de grises se puede utilizar para comprimir imágenes en color. En una imagen en color, cada píxel está representado por tres componentes de color (tales como RGB). Imagínese una imagen en color, donde se representa cada componente de color por un byte. Un píxel se representa por tres bytes —ó 24 bits—; pero estos bits no deben considerarse un solo número. Los dos píxeles —118|206|12 y 117|206|12— difieren sólo una unidad en el primer componente, por lo que tienen colores muy similares. Considerados como números de 24 bits, sin embargo, estos píxeles son muy diferentes, ya que difieren en uno de sus bits más significativos. Cualquier método de compresión que trate estos píxeles como números de 24 bits, considerará dichos píxeles muy diferentes y, como resultado, su rendimiento va a resentirse. Un método de compresión para imágenes en escala de grises puede aplicarse para comprimir imágenes en color; pero la imagen en color primero debe ser separada en los tres componentes de color, y cada componente comprimido de forma individual como en una imagen en escala de grises.

Como ejemplo de uso del RGC para la compresión de imágenes véase la Sección 4.27.

El código de Gray binario es divertido,
Por las cosas extrañas que puede hacer.
Quince, como usted sabe,
es uno, oh, oh, oh[a],
Y diez es uno, uno, uno, uno.

—Anónimo.

[a] oh es una manera de decir *cero* en inglés.

4.2.2. Métricas de error

Los desarrolladores e implementadores de los métodos de compresión de imágenes con pérdidas necesitan un estándar métrico para medir la calidad de las imágenes reconstruidas en comparación con las originales. La imagen reconstruida que más se asemeja a la original, es la que debe tener el mayor valor producido por esta métrica. Tal métrica también debe producir un número adimensional, y ese número no debe ser muy sensible a las pequeñas variaciones en la imagen reconstruida. Una medida común utilizada para éste propósito es la *relación entre el pico máximo de la señal y el ruido* (*peak signal to noise ratio* o PSNR). Es familiar a los trabajadores en el campo; también es fácil de calcular, pero tiene al menos una limitación: la relación con los errores de percepción observados por el sistema visual humano no es estrecha. Por esta razón, valores más altos de la PSNR implican un mayor parecido numérico entre las imagenes reconstruidas y las originales, pero no necesariamente perceptual; por lo que no proporcionan una garantía de que a los espectadores les guste la imagen reconstruida.[5]

Denotando los píxeles de la imagen original por P_i y los píxeles de la imagen reconstruida por Q_i (donde $1 \leq i \leq n$), definimos en primer lugar el *error cuadrático medio* (*mean square error* o MSE) entre las dos imágenes como:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(P_i - Q_i)^2 . \tag{4.2}$$

Éste es el promedio del cuadrado de los errores (diferencias de píxeles) de las dos imágenes. La *raíz del error cuadrático medio* (*root mean square error* o RMSE) se define como la raíz cuadrada del MSE y la PSNR se define como:

$$\text{PSNR} = 20\,\log_{10}\frac{\max_i |P_i|}{\text{RMSE}}.$$

El valor absoluto normalmente no es necesario, ya que los valores de píxel rara vez son negativos. Para una imagen binivel, el numerador es 1. Para una imagen en escala de grises con ocho bits por píxel, el numerador es $2^8 - 1 = 255$. Para imágenes en color, solamente se utiliza la componente de luminancia.

Historia de los Códigos de Gray

Los códigos de Gray llevan el nombre de Frank Gray, quien patentó su uso para los codificadores de árbol en 1953 [Gray 53]. No obstante, el trabajo se llevó a cabo mucho antes, pues la patente se solicitó en 1947. Gray era un investigador en los laboratorios Bell. Durante las décadas de 1930 y 1940 le otorgaron numerosas patentes por el trabajo relacionado con la televisión. Según [Heath 72] el código fue, de hecho, utilizado primero por J.M.E. Baudot para la telegrafía, en la década de 1870 (Sección 1.1.3), aunque es sólo desde el advenimiento de las computadoras que el código ha sido ampliamente conocido.

El código de Baudot utiliza cinco bits por símbolo. Puede representar $32 \times 2 - 2 = 62$ caracteres (cada código puede tener dos significados; el significado se indica mediante los códigos `LS` y `FS`). Se hizo popular y, en 1950, se designó el Código Internacional Telegráfico Nº 1. Fue utilizado por muchos ordenadores de primera y segunda generación.

El número de agosto de 1972 de *Scientific American* contiene dos artículos de interés: uno sobre el origen de códigos binarios [Heath 72], y otro [Gardner 72] sobre algunos aspectos entretenidos de los códigos de Gray.

[5] Las dos últimas frases han sido modificadas con respecto al original, siguiendo la recomendación de Miguel Hernández Cabronero, que se prestó a revisar la traducción.

Una mayor semejanza entre las imágenes implica una RMSE menor y, como resultado, una mayor PSNR. La PSNR no tiene dimensiones, ya que las unidades, tanto del numerador como del denominador, son valores de píxel. Sin embargo, debido al uso del logaritmo, se dice que la PSNR se expresa en *decibelios* (dB, Sección 7.1). El uso del logaritmo también implica menos sensibilidad a los cambios en la RMSE. Por ejemplo, la división por 10 de la RMSE, multiplica por 2 la PSNR. Observe que la PSNR no tiene ningún significado absoluto. No tiene sentido decir que una PSNR de, digamos, 25 dB es buena. Los valores de la PSNR sólo se utilizan para comparar el rendimiento de los diferentes métodos de compresión con pérdida, o los efectos de diferentes valores de los parámetros en el rendimiento de un algoritmo. El comité de MPEG, por ejemplo, utiliza un umbral informal de PSNR = 0,5 dB para decidir si incorporar una optimización de la codificación, puesto que creen que una mejora de ésa magnitud sería visible para el ojo.

Los valores típicos de la PSNR están en el rango, entre 20 y 40. Suponiendo que los valores de los píxeles están dentro del intervalo $[0, 255]$, una RMSE de 25,5 produce una PSNR de 20, y una RMSE de 2,55 genera una PSNR de 40. Una RMSE de cero (i.e., imágenes idénticas) tiene como efecto una PSNR infinita (o, más precisamente, indefinida). Una RMSE de 255 produce una PSNR de cero, y una RMSE de valores mayores que 255, provoca rendimientos negativos en las PSNRs.

◇ **Ejercicio 4.6 (sol. en pág. 1079):** Si el valor de píxel máximo es de 255, ¿los valores de la RMSE pueden ser mayores que 255?

Algunos autores definen la PSNR como:

$$\text{PSNR} = 10 \log_{10} \frac{\max_i |P_i|^2}{\text{MSE}}.$$

A fin de que las dos fórmulas produzcan el mismo resultado, el logaritmo se multiplica en este caso por 10 —en lugar de por 20—, ya que $\log_{10} A^2 = 2 \log_{10} A$. Cualquiera de las dos definiciones es útil, porque en la práctica, solamente se utilizan valores relativos de la PSNR. Sin embargo, el uso de dos factores diferentes es confuso.

Una medida relacionada es la *relación señal–ruido* (*signal to noise ratio* o SNR). Ésta se define como:

$$\text{SNR} = 20 \log_{10} \frac{\sqrt{\frac{1}{n} \sum_{i=1}^{n} P_i^2}}{\text{RMSE}}.$$

El numerador es la raíz de la media cuadrática de la imagen original.

La Figura 4.13 es una función en *Matlab* para calcular la PSNR de dos imágenes. Una llamada típica es `PSNR(A,B)`, donde `A` y `B` son archivos de imagen. Éstos deben tener la misma resolución y tienen valores de píxeles dentro del rango $[0, 1]$.

Otra expresión relacionada con la PSNR es la *relación entre la señal y el ruido de cuantificación* (*signal to quantization noise ratio* o SQNR). Éste es una medida del efecto de la cuantificación en la calidad de la señal. Se define como:

$$\text{SQNR} = 10 \log_{10} \frac{\text{potencia de la señal}}{\text{error de cuantificación}},$$

donde el error de cuantificación es la diferencia entre la señal cuantificada y el señal original.

Otro enfoque para la comparación de una imagen original y una imagen reconstruida es generar la imagen de la diferencia y juzgarla visualmente. Intuitivamente, la imagen de la diferencia es $D_i = P_i - Q_i$, pero tal imagen es difícil de juzgar visualmente debido a que sus valores de píxel D_i tienden a ser números pequeños. Si un valor de píxel de cero representa el blanco, tal diferencia de la imagen sería casi invisible. En el caso opuesto, donde un valores de píxel de cero representa el negro, dicha diferencia sería demasiado oscura como para juzgarla. Se obtienen mejores resultados mediante el cálculo:

$$D_i = a\,(P_i - Q_i) + b,$$

```
function PSNR(A,B)
if A==B
  error('Las imágenes son idénticas; la PSNR está indefinida')
end
max2_A=max(max(A)); max2_B=max(max(B));
min2_A=min(min(A)); min2_B=min(min(B));
if max2_A>1 | max2_B>1 | min2_A<0 | min2_B<0
  error('los píxeles deben pertenecer al intervalo [0,1]')
end
differ=A-B;
decib=20*log10(1/(sqrt(mean(mean(differ.^2)))));
disp(sprintf('PSNR = +%5.2f dB',decib))
```

Figura 4.13: Una función en Matlab para calcular la PSNR.

donde a es un parámetro de magnificación (típicamente un número tan pequeño como 2) y b es la mitad del valor máximo de un píxel (típicamente 128). El parámetro a sirve para magnificar pequeñas diferencias, mientras que b desplaza la imagen diferencia desde el extremo blanco (o extremo negro) a un gris mucho más cómodo.

4.3. Métodos intuitivos

Es fácil llegar a métodos sencillos e intuitivos para la compresión de imágenes. Pero son ineficientes, y sólo se describen aquí con el fin de complementar.

4.3.1. Submuestreo

El submuestreo es quizás la forma más sencilla para comprimir una imagen. Un enfoque para el submuestreo consiste simplemente en eliminar algunos de los píxeles. El codificador puede, por ejemplo, ignorar una fila de cada dos y una columna de cada dos de la imagen, y escribir el resto de los píxeles (que constituyen el 25 % de la imagen) en la cadena comprimida. El decodificador introduce los datos comprimidos y utiliza cada píxel para generar cuatro píxeles idénticos de la imagen reconstruida. Esto, por supuesto, implica la pérdida de muchos detalles de la imagen y raramente es aceptable. Obsérvese que la razón de compresión se conoce de antemano.

Una ligera mejora se obtiene cuando el codificador calcula el promedio de cada bloque de cuatro píxeles y escribe este promedio en la secuencia de datos comprimidos. Ningún píxel es totalmente eliminado; pero el método es todavía primitivo, ya que un buen método de compresión de imágenes con pérdidas debería perder sólo los datos a los que el ojo no es sensible.

Mejores resultados (pero con peor compresión) se obtienen cuando se cambia la representación del color de la imagen original (normalmente RGB) en componentes de luminancia y crominancia. El codificador submuestrea los dos componentes de crominancia de un píxel, pero no su componente de luminancia. Suponiendo que cada componente utiliza el mismo número de bits, los dos componentes de crominancia acaparan $2/3$ del tamaño de la imagen. Un nuevo submuestreo reduce ésto al 25 % de $2/3$, ó $1/6$. El tamaño de la imagen comprimida es, por lo tanto, $1/3$ (para el componente de luminancia sin comprimir), más $1/6$ (para los dos componentes de crominancia); ó $1/2$ del tamaño original.

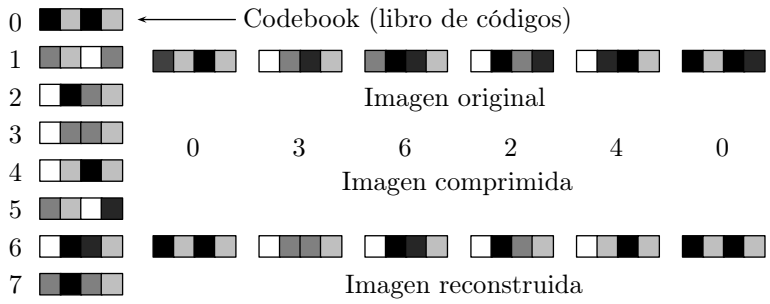


Figura 4.14: Cuantificación vectorial intuitiva.

4.3.2. Cuantificación

La cuantificación escalar se ha mencionado en la Sección 4.1. Se trata de un método intuitivo con pérdidas, donde la información perdida no es necesariamente la menos importante. La cuantificación vectorial puede obtener mejores resultados, y aquí se describe una versión intuitiva del mismo.

La imagen se divide en bloques de píxeles de igual tamaño (llamados *vectores*), y el codificador tiene una lista (llamada *libro de códigos* o *codebook*) de bloques del mismo tamaño. Cada bloque B de la imagen se compara con todos los bloques del llibro de códigos y se establece una correspondencia con el "más cercano".

Si B se ha emparejado con el bloque C del libro de códigos, entonces el codificador escribe un puntero a C en la cadena comprimida. Si el puntero es más pequeño que el tamaño del bloque, se logra la compresión. La Figura 4.14 muestra un ejemplo.

Los detalles para seleccionar y mantener el libro de códigos y de bloques emparejados se discute en la Sección 4.14. Observe que la cuantificación vectorial es un método donde la razón de compresión se conoce de antemano.

4.4. Transformadas de imágenes

El concepto matemático de transformada es una poderosa herramienta que se emplea en muchas áreas y también puede servir como una aproximación a la compresión de imágenes. La Sección 5.1 discute este concepto en general, así como la transformada de Fourier. Una imagen se puede comprimir transformando sus píxeles (que están correlacionados) en una representación donde estén *descorrelacionados*. La compresión se logra si los nuevos valores son más pequeños, en promedio, que los originales. La compresión con pérdida puede lograrse mediante la cuantificación de los valores de la transformada. El decodificador introduce los valores transformados de la secuencia de datos comprimidos y efectúa la reconstrucción (precisa o aproximada) de los datos originales mediante la aplicación de la transformada inversa. Las transformaciones discutidas en esta sección son *ortogonales*. La Sección 5.6.1 explica las *transformadas de subbanda*.

El término *descorrelacionado* significa que los valores transformados son estadísticamente independientes. Como resultado, pueden ser codificados independientemente, lo que hace más sencillo construir un modelo estadístico. Una imagen se puede comprimir si su representación tiene redundancia. La redundancia en las imágenes proviene de la correlación entre los píxeles. Si transformamos la imagen a una representación donde los píxeles estén descorrelacionados, hemos eliminado la redundancia y la imagen ha sido totalmente comprimida.

Empezamos con un sencillo ejemplo, donde escaneamos una imagen en orden *raster* —de arriba abajo y de izquierda a derecha— y agrupamos pares de píxeles adyacentes. Debido a que los píxeles

```
p={{5,5},{6, 7},{12.1,13.2},{23,25},{32,29}};
rot={{0.7071,-0.7071},{0.7071,0.7071}};
Sum[p[[i,1]]p[[i,2]], {i,5}]
q=p.rot
Sum[q[[i,1]]q[[i,2]], {i,5}]
```

Figura 4.15: Código para la rotación de cinco puntos.

están correlacionados, los dos pixeles (x, y) de un par normalmente tienen valores similares. Consideramos cada par de píxeles un punto del espacio bidimensional, y dibujamos los puntos. Sabemos que todos los puntos de la forma (x, x) se encuentran en la recta de 45º $y = x$, por lo que esperamos que nuestros puntos que se concentren en torno a esta línea. La Figura 4.16a muestra los resultados del trazado de los píxeles de una imagen típica —donde un píxel tiene valores en el intervalo $[0, 255]$— de esa manera. La mayoría de los puntos forman una nube alrededor de esta línea, y sólo unos pocos puntos se encuentran lejos de ella. Ahora transformamos la imagen girando todos los puntos 45º en el sentido de las agujas de un reloj alrededor del origen, de tal manera que la recta de 45º coincida ahora con el eje x (Figura 4.16b). Ésto se hace mediante la sencilla transformación [véase la Ecuación (4.62)]:

$$(x^*, y^*) = (x, y) \begin{pmatrix} \cos 45^o & -\text{sen}45^o \\ \text{sen}45^o & \cos 45^o \end{pmatrix} = (x, y) \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix} = (x, y)\, \mathbf{R}, \tag{4.3}$$

donde la matriz de rotación $\mathbf{R}$ es ortonormal (i.e., el producto escalar de una fila con sí misma es 1, el producto escalar de las filas diferentes es 0, y lo mismo es cierto para las columnas). La transformación inversa es:

$$(x, y) = (x^*, y^*)\, \mathbf{R}^{-1} = (x*, y*)\, \mathbf{R}^T = (x^*, y^*) \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix}. \tag{4.4}$$

(La inversa de una matriz ortonormal es su transpuesta.)

Es obvio que la mayor parte de los puntos terminan con coordenadas y que son cero o cercanas a cero, mientras que las coordenadas x no cambian mucho. La Figura 4.17a,b, muestra la distribución de las coordenadas x e y (i.e., los píxeles con números impares y con números pares) de la imagen de Lena en escala de grises de $128 \times 128 \times 8$, antes de la rotación. Es evidente que las dos distribuciones no difieren en mucho. La Figura 4.17c,d muestra que la distribución de las coordenadas x se mantiene igual (con una varianza mayor), pero las coordenadas y se concentra en torno a cero. También se muestra el código en Matlab que generó estos resultados. (La Figura 4.17d muestra que las coordenadas x e y se concentran alrededor de 100, pero ésto se debe a que unos pocos eran tan pequeños como -101, por lo que tuvo que ser escalado a 101 para alojarlos en un array de Matlab, que siempre comienza en el índice 1.)

Una vez que se conocen las coordenadas de los puntos antes y después de la rotación, es fácil medir la reducción en la correlación. Una medida sencilla es la suma $\sum_i x_i y_i$, también llamada *correlación cruzada* de los puntos (x_i, y_i).

◇ **Ejercicio 4.7 (sol. en pág. 1079):** Dados los cinco puntos $(5, 5)$, $(6, 7)$, (12,1, 13,2), $(23, 25)$ y $(32, 29)$, gírense 45º en el sentido de las agujas del reloj y calcúlense sus correlaciones cruzadas antes y después de la rotación.

Ahora podemos comprimir la imagen simplemente escribiendo los píxeles transformados en la cadena comprimida. Si una compresión con pérdida es aceptable, entonces todos los píxeles pueden ser cuantificados (Secciones 1.6 y 4.14), produciendo números aún más pequeños. También podemos escribir todos los píxeles impares (los que forman las coordenadas x de los pares) en la secuencia de datos comprimidos, seguidos por todos los píxeles pares. Estas dos secuencias reciben el nombre de

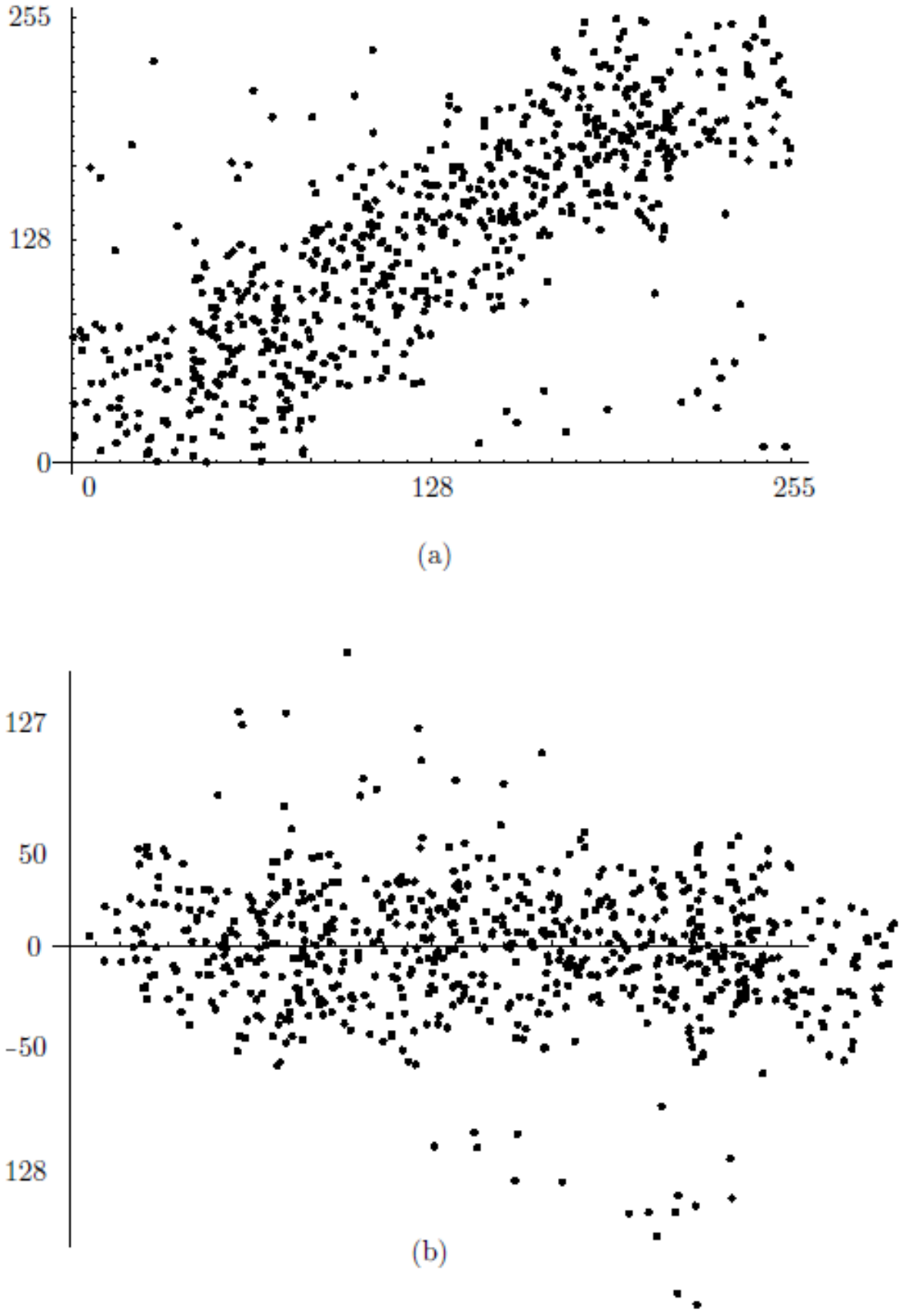
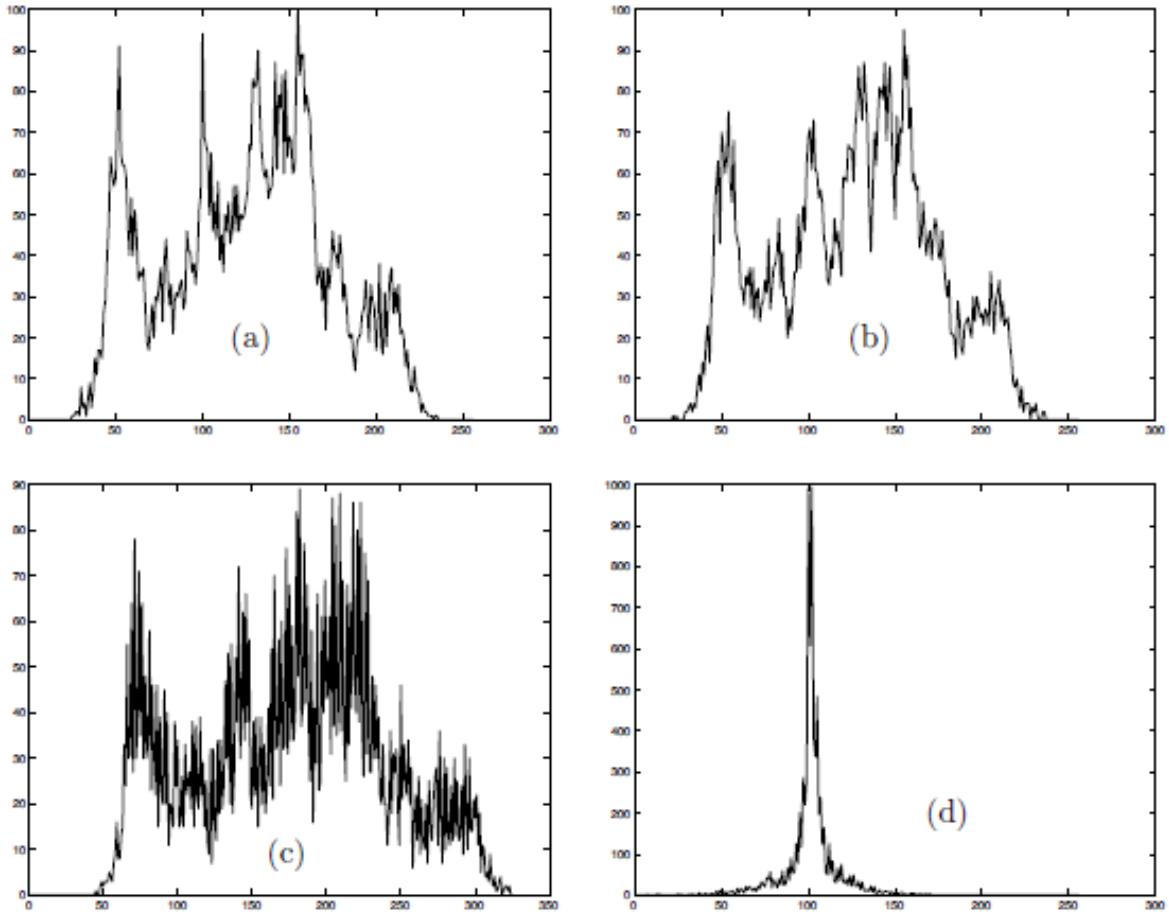


Figura 4.16: Rotación de una nube de puntos.




```
filename='lena128'; dim=128;
xdist=zeros(256,1); ydist=zeros(256,1);
fid=fopen(filename,'r');
img=fread(fid,[dim,dim])';
for col=1:2:dim-1
  for row=1:dim
    x=img(row,col)+1; y=img(row,col+1)+1;
    xdist(x)=xdist(x)+1; ydist(y)=ydist(y)+1;
  end
end
figure(1), plot(xdist), colormap(gray)%valores de dist. de x&y
figure(2), plot(ydist), colormap(gray)%antes de la rotación
xdist=zeros(325,1);% inicializa a cero los arrays
ydist=zeros(256,1);
for col=1:2:dim-1
  for row=1:dim
    x=round((img(row,col)+img(row,col+1))*0.7071);
    y=round((-img(row,col)+img(row,col+1))*0.7071)+101;
    xdist(x)=xdist(x)+1; ydist(y)=ydist(y)+1;
  end
end
figure(3), plot(xdist), colormap(gray)%valores de dist. de x&y
figure(4), plot(ydist), colormap(gray)%después de la rotación
```

Figura 4.17: Distribución de píxeles de la imagen antes y después de la rotación.

vectores de coeficientes de la transformada. La última secuencia está formada por números pequeños y puede, después de la cuantificación, tener *runs* de ceros, lo que mejora aún más la compresión.

Se puede demostrar que la varianza total de los píxeles no cambia por la rotación, porque una matriz de rotación es ortonormal. Sin embargo, puesto que la varianza de las nuevas coordenadas y es pequeña, la mayor parte de la diferencia se concentra ahora en las coordenadas x. La varianza se denomina a veces *energía* de la distribución de los píxeles, por lo que podemos decir que la rotación ha concentrado (o compactado) la energía en la coordenada x y ha obtenido compresión de esta manera.

La concentración de la energía en una coordenada tiene otra ventaja. Hace posible cuantificar esas coordenadas con más precisión que las otras. Este tipo de cuantificación produce una mejor compresión (con pérdidas).

El siguiente ejemplo ilustra el poder de esta transformación básica. Comenzamos con el punto $(4, 5)$, cuyas dos coordenadas son similares. Utilizando la Ecuación (4.3), el punto se transforma en $(4, 5)\,\mathbf{R} = {}^{(9,1)}/_{\sqrt{2}} \approx (6{,}36396,\ 0{,}7071)$. Las energías del punto y de su transformada son: $4^2 + 5^2 = 41 = {}^{(9^2+1^2)}/_2$. Si eliminamos la coordenada más pequeña (4) del punto, acabamos con un error de ${}^{4^2}/_{41} = 0{,}39$. Si, por otro lado, eliminamos el menor de los dos coeficientes de la transformación (0,7071), el error resultante es de sólo ${}^{0{,}7071^2}/_{41} = 0{,}012$. Otra forma de obtener el mismo error es considerar el punto reconstruido. El paso de $\frac{1}{\sqrt{2}}\,(9, 1)$ a través de la transformada inversa [Ecuación (4.4)] produce el punto original $(4, 5)$. Haciendo lo mismo con $\frac{1}{\sqrt{2}}\,(9, 0)$ da como resultado el punto de reconstrucción aproximada (4,5 4,5). La diferencia de energía entre los puntos originales y el reconstruido es la misma pequeña cantidad:

$$\frac{\left[\left(4^2 + 5^2\right) - \left(4{,}5^2 + 4{,}5^2\right)\right]}{4^2 + 5^2} = \frac{41 - 40{,}5}{41} = 0{,}0012.$$

Esta sencilla transformación se puede extender fácilmente a cualquier número de dimensiones. En lugar de seleccionar pares de píxeles adyacentes, podemos seleccionar tripletes. Cada triplete se convierte en un punto en el espacio tridimensional, y estos puntos forman una nube concentrada alrededor de la línea que forma ángulos iguales (aunque no de 45°) con los tres ejes de coordenadas. Cuando esta línea se hace girar de tal manera que coincida con el eje x, las coordenadas y y z de los puntos transformados se convierten en números pequeños. La transformación se realiza multiplicando cada punto por una matriz de rotación de 3×3, y tal matriz es, por supuesto, ortonormal. Los puntos transformados se separa en tres vectores de coeficientes, de los cuales los dos últimos están formados por números pequeños. Para una compresión máxima, cada vector de coeficientes debe cuantificarse por separado.

Ésto puede extenderse a más de tres dimensiones, con la única diferencia de que no podemos visualizar espacios de dimensiones superiores a tres. Sin embargo, las matemáticas se pueden ampliar fácilmente. Algunos métodos de compresión, tales como JPEG, dividen una imagen en bloques de 8×8 píxeles cada uno, y rotan primero cada fila, luego cada columna, mediante la Ecuación (4.15), como se muestra en la Sección 4.6. Esta doble rotación produce un conjunto de 64 valores transformados, de los cuales el primero —denominado "coeficiente DC"— es grande, y los otros 63 (llamados "coeficientes AC") son normalmente pequeños. En consecuencia, esta transformación concentra la energía en la primera de las 64 dimensiones. El conjunto de coeficientes DC y cada uno de los conjuntos de 63 coeficientes AC, en principio, se cuantifican por separado (aunque JPEG hace ésto de manera algo diferente; consulte la Sección 4.8.3).

4.5. Transformadas ortogonales

Las transformadas de una imagen están diseñadas para tener dos propiedades: (1) disminuir la redundancia de la imagen mediante la reducción de los tamaños de la mayor parte de los píxeles y (2) identificar las partes menos importantes de la imagen aislando las diversas frecuencias de la imagen. Por

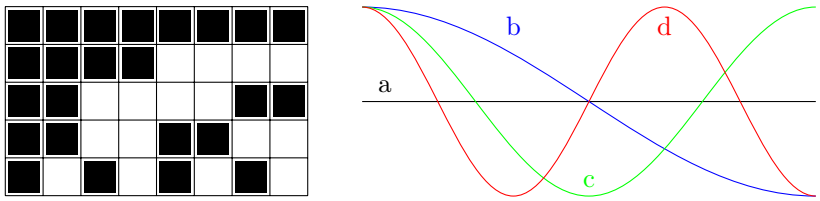


Figura 4.18: Frecuencias de una imagen.

consiguiente, esta sección se inicia con una breve discusión de las frecuencias. Intuitivamente, asociamos una frecuencia a una onda. Las ondas en el agua, las ondas sonoras y las ondas electromagnéticas tienen frecuencias, pero los píxeles de una imagen también pueden poseer frecuencias. La Figura 4.18 muestra una pequeña imagen binivel de 5×8 que ilustra este concepto. La fila superior es uniforme, por lo que se le puede asignar la frecuencia cero. Las filas inferiores van aumentando las frecuencias de píxel medidas según el número de cambios de color a lo largo de una fila. Los cuatro ondas de la derecha corresponden aproximadamente a las frecuencias de las cuatro filas superiores de la imagen.

Las frecuencias de la imagen son importantes por el siguiente hecho básico: las frecuencias bajas corresponden a características importantes de la imagen, mientras que las frecuencias altas corresponden a los detalles de la imagen, que son menos importantes. Así, cuando una transformada aísla las diferentes frecuencias de la imagen, los píxeles que corresponden a las frecuencias altas se pueden cuantificar fuertemente, mientras que los píxeles que corresponden a las frecuencias bajas deberían cuantificarse ligeramente o nada en absoluto. Así es cómo una transformada puede comprimir una imagen muy eficazmente perdiendo información; pero sólo la información asociada con detalles de menor importancia de la imagen.

En la práctica, las transformaciones de las imágenes deben ser rápidas, y preferiblemente también sencillas de implementar. Ésto sugiere el uso de *transformadas lineales*. En tales transformadas, cada valor transformado (o coeficiente de la transformada) c_i es una suma ponderada de los ítems de datos (los píxeles) d_j que están siendo transformados, donde cada ítem se multiplica por un peso ω_{ij}. Por lo tanto, $c_i = \sum_j d_j \omega_{ij}$, para $i, j = 1, 2, \ldots, n$. Para $n = 4$, ésto se expresa en notación matricial:

$$\begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{pmatrix} = \begin{pmatrix} \omega_{11} & \omega_{12} & \omega_{13} & \omega_{14} \\ \omega_{21} & \omega_{22} & \omega_{23} & \omega_{24} \\ \omega_{31} & \omega_{32} & \omega_{33} & \omega_{34} \\ \omega_{41} & \omega_{42} & \omega_{43} & \omega_{44} \end{pmatrix} \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \end{pmatrix}.$$

Para el caso general, podemos escribir $\mathbf{C} = \mathbf{W} \cdot \mathbf{D}$. Cada fila de $\mathbf{W}$ se llama "vector de la base".

Las únicas cantidades que tienen que ser calculadas son los pesos ω_{ij}. Los principios guía son los siguientes:

1. La reducción de la redundancia. El primer coeficiente c_1 de la transformada puede ser grande, pero el resto de los valores —$c_2, c_3, \ldots$— deben ser pequeños.

2. El aislamiento de las frecuencias. El primer coeficiente c_1 de la transformada debe corresponder a la frecuencia de píxel cero, y los coeficientes restantes deben corresponder a frecuencias cada vez más altas.

La clave para determinar los pesos ω_{ij} es el hecho de que nuestros ítems de datos d_j no son números arbitrarios sino valores de píxeles, que son no negativos y correlacionados.

La relación base $c_i = \sum_j d_j \omega_{ij}$ sugiere que el primer coeficiente c_i será grande si todos los pesos de la forma ω_{1j} son positivos. Para hacer que los otros coeficientes c_i sean pequeños, es suficiente hacer que la mitad de los pesos ω_{ij} sean positivos y la otra mitad negativos. Una sencilla elección es asignar

a la mitad de los pesos el valor +1 y a la otra mitad del valor −1. En el caso extremo, en el que todos los píxeles d_j sean idénticos, ésto dará como resultado $c_i = 0$. Cuando los d_j's sean similares, c_i será pequeño (positivo o negativo).

Esta elección de ω_{ij} satisface la primera condición: reducir la redundancia de los píxeles mediante una transformación. Con el fin de satisfacer el segundo requisito, los pesos ω_{ij} de la fila i deben representar frecuencias cada vez más altas con cada incremento de i. Los pesos ω_{1j} deben tener frecuencia cero; todos ellos deben ser +1'. Los pesos ω_{2j} deben tener un cambio de signo; i.e., deben ser: $+1, +1, \ldots, +1, -1, -1, \ldots, -1$. Ésto continúa hasta la última fila, cuyos pesos ω_{nj} deben tener la frecuencia más alta : $+1, -1, +1, -1, \ldots, +1, -1$. La disciplina matemática de los espacios vectoriales otorga el término "vectores de la base" a nuestras filas de pesos.

Además de aislar las distintas frecuencias de los píxeles d_j, esta elección produce vectores de la base que son ortogonales. Los vectores de la base son las filas de la matriz $\mathbf{W}$, que es por lo que esta matriz y, en consecuencia, la transformada completa también se llama ortogonal.

Estas consideraciones se ven satisfechas con la matriz ortogonal:

$$\mathbf{W} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{pmatrix}. \tag{4.5}$$

El primer vector de la base (la fila superior de $\mathbf{W}$) está formado únicamente por 1's, por lo que su frecuencia es cero. Cada uno de los vectores subsiguientes tiene dos +1's y dos −1's, produciendo valores transformados pequeños, y sus frecuencias (medidas como el número de cambios de signo a lo largo del vector de la base) más altas. Esta matriz es similar a la transformada de Walsh-Hadamard [Ecuación (4.6)].

Para ilustrar cómo identifica esta matriz las frecuencias en un vector de datos, la multiplicamos por cuatro vectores como sigue:

$$\mathbf{W} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \\ 2 \\ 0 \end{bmatrix}, \quad \mathbf{W} \cdot \begin{bmatrix} 0 \\ 0{,}33 \\ -0{,}33 \\ -1 \end{bmatrix} = \begin{bmatrix} 0 \\ 2{,}66 \\ 0 \\ 1{,}33 \end{bmatrix},$$

$$\mathbf{W} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \quad \mathbf{W} \cdot \begin{bmatrix} 1 \\ -0{,}8 \\ 1 \\ -0{,}8 \end{bmatrix} = \begin{bmatrix} 0{,}4 \\ 0 \\ 0 \\ 3{,}6 \end{bmatrix}.$$

Los resultados tienen sentido cuando descubrimos cómo se han determinado los cuatro vectores de prueba:

$$\begin{gathered} (1,0,0,1) = 0{,}5\,(1,1,1,1) + 0{,}5\,(1,-1,-1,1)\,, \\ (1,\ 0{,}33,\ -0{,}33,\ -1) = 0{,}66\,(1,1,-1,-1) + 0{,}33\,(1,-1,1,-1)\,, \\ (1,0,0,0) = 0{,}25\,(1,1,1,1) + 0{,}25\,(1,1,-1,-1) + 0{,}25\,(1,-1,-1,1) + 0{,}25\,(1,-1,1,-1)\,, \\ (1,-0{,}8,1,-0{,}8) = 0{,}1\,(1,1,1,1) + 0{,}9\,(1,-1,1,-1)\,. \end{gathered}$$

El producto de $\mathbf{W}$ y el primer vector muestra cómo ése vector consta de cantidades iguales en las frecuencias primera y tercera. De manera similar, la transformación $(0{,}4, 0, 0, 3{,}6)$ muestra que el vector $(1, -0{,}8, 1, -0{,}8)$ es una mezcla de una pequeña cantidad de la primera frecuencia y nueve veces la cuarta frecuencia.

También es posible modificar esta transformada para conservar la energía de los vectores de datos. Todo lo que se necesita es multiplicar la matriz de la transformada $\mathbf{W}$ por el factor de escala $1/2$. Así, el

producto $(\mathbf{W}/2)\times(a,b,c,d)$ tiene la misma energía $a^2+b^2+c^2+d^2$ que el vector de datos (a,b,c,d). Un ejemplo es, el producto de $\mathbf{W}/2$ y el vector correlacionado $(5,6,7,8)$. Éste produce los coeficientes de la transformada $(13,-2,0,-1)$, donde el primer coeficiente es grande, y los restantes son más pequeños que los ítems de datos originales. La energía de ambos —$(5,6,7,8)$ y $(13,-2,0,-1)$— es 174, pero mientras que en el primer vector las primeras cantidades de los componentes abarcan sólo el 14 % de la energía, en el vector transformado el primer componente representa el 97 % de la energía. Así es como nuestra sencilla transformada ortogonal compacta la energía del vector de datos.

Otra ventaja de $\mathbf{W}$ es que también realiza la transformada inversa. El producto $(\mathbf{W}/2)\cdot(13,-2,0,-1)^T$ reconstruye los datos originales $(5,6,7,8)$.

Ahora estamos en posición de apreciar el potencial de compresión de esta transformada. Usamos la matriz $\mathbf{W}/2$ para transformar el vector de datos (no muy correlacionados) $d=(4,6,5,2)$. El resultado es $t=$ (8,5, 1,5, −2,5, 0,5). Es fácil transformar t de nuevo en d, pero t en sí mismo no proporciona ningún tipo de compresión. A fin de lograr compresión, cuantificamos las componentes de t, y lo importante es que incluso después de una fuerte cuantificación, todavía es posible obtener un vector muy similar al d original.

En primer lugar, cuantificamos t para los números enteros $(9,1,-3,0)$ y realizamos la transformada inversa para obtener de nuevo (3,5, 6,5, 5,5, 2,5). En un experimento similar, eliminamos por completo los dos elementos más pequeños y calculamos la transformada inversa del vector toscamente cuantificado (8,5, 0, −2,5, 0). Ésto produce los datos reconstruidos (3, 5,5, 5,5, 3), todavía muy cercanos a los valores originales de d. La conclusión es, que incluso en esta sencilla e intuitiva transformación, es una herramienta poderosa para "expulsar" la redundancia en los datos. Transformaciones más sofisticadas producen resultados que pueden ser fuertemente cuantificados y todavía utilizados para reconstruir los datos originales en su mayor parte.

4.5.1. Transformadas bidimensionales

Dando los datos bidimensionales, como la matriz de 4×4:

$$\mathbf{D}=\begin{pmatrix}5&6&7&4\\6&5&7&5\\7&7&6&6\\8&8&8&8\end{pmatrix},$$

donde cada una de las columnas está altamente correlacionada, podemos aplicar nuestra sencilla transformada unidimensional a las columnas de $\mathbf{D}$. El resultado es:[6]

$$\mathbf{C}'=\mathbf{W}\cdot\mathbf{D}=\begin{pmatrix}1&1&1&1\\1&1&-1&-1\\1&-1&-1&1\\1&-1&1&-1\end{pmatrix}\cdot\mathbf{D}=\begin{pmatrix}26&26&28&23\\-4&-4&0&-5\\0&2&2&1\\-2&0&-2&-3\end{pmatrix}.$$

Cada columna de $\mathbf{C}'$ es la transformada de una columna de $\mathbf{D}$. Observe cómo el elemento superior de cada columna de $\mathbf{C}'$ es dominante, ya que los datos en la columna correspondiente de $\mathbf{D}$ están correlacionados. Nótese también que las filas de $\mathbf{C}'$ siguen aún correlacionados. $\mathbf{C}'$ es la primera etapa en un proceso de dos etapas que produce la transformada bidimensional de la matriz $\mathbf{D}$. La segunda etapa debe transformar cada *fila* de $\mathbf{C}'$, y ésto se hace multiplicando $\mathbf{C}'$ por la transpuesta

[6] Recuérdese la mutiplicación de matrices. Ejemplo para matrices de orden 2: $\begin{pmatrix}a_{11}&a_{12}\\a_{21}&a_{22}\end{pmatrix}\begin{pmatrix}b_{11}&b_{12}\\b_{21}&b_{22}\end{pmatrix}=\begin{pmatrix}a_{11}b_{11}+a_{12}b_{21}&a_{11}b_{12}+a_{12}b_{22}\\a_{21}b_{11}+a_{22}b_{21}&a_{21}b_{12}+a_{22}b_{22}\end{pmatrix}$.

$\mathbf{W}^T$. Nuestra particular $\mathbf{W}$, sin embargo, es simétrica, por lo que terminamos con $\mathbf{C} = \mathbf{C}' \cdot \mathbf{W}^T = \mathbf{W} \cdot \mathbf{D} \cdot \mathbf{W}^T = \mathbf{W} \cdot \mathbf{D} \cdot \mathbf{W}$ o

$$\mathbf{C} = \begin{pmatrix} 26 & 26 & 28 & 23 \\ -4 & -4 & 0 & -5 \\ 0 & 2 & 2 & 1 \\ -2 & 0 & -2 & -3 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{pmatrix} = \begin{pmatrix} 103 & 1 & -5 & 5 \\ -13 & -3 & -5 & 5 \\ 5 & 1 & -3 & -1 \\ -7 & 3 & -3 & -1 \end{pmatrix}.$$

Los elementos de $\mathbf{C}$ están descorrelacionados. El elemento superior izquierdo es el dominante. Contiene más energía que la total del original $\mathbf{D}$. Los elementos de la fila superior y de la columna del extremo izquierdo son algo grandes, mientras que los elementos restantes son más pequeños que los ítems de datos originales. La doble etapa de la transformada bidimensional, ha reducido la correlación tanto en las dimensiones horizontales como en las verticales. Como en el caso unidimensional, puede lograrse una excelente compresión mediante la cuantificación de los elementos de $\mathbf{C}$, especialmente aquellos que corresponden a las frecuencias más altas (i.e., los ubicados hacia la esquina inferior derecha de $\mathbf{C}$).

Ésta es la esencia de las transformaciones ortogonales. El resto de esta sección analiza las siguientes transformaciones importantes:

1. La transformada de Walsh-Hadamard (WHT, Sección 4.5.2) es rápida y fácil de calcular (requiere solamente adiciones y sustracciones); pero su rendimiento, en términos de compactación de energía, es menor que la de la DCT.

2. La transformada de Haar [Stollnitz et al. 96] es una transformada simple y rápida. Es la transformada wavelet más sencilla, y se discute en la Sección 4.5.3 y en el Capítulo 5.

3. La transformada de Karhunen-Loève (KLT, Sección 4.5.4) es la mejor —en teoría—, en el sentido de la compactación de la energía (o, equivalentemente, descorrelación de píxeles). Sin embargo, su coeficientes no están fijados; dependen de los datos que van a ser comprimidos. El cálculo de estos coeficientes (la base de la transformada) es lento, así como el cálculo de los valores de la transformada en sí mismos. Dado que los coeficientes dependen de los datos, tienen que ser incluidos en la cadena comprimida. Por estas razones y debido a que la DCT tiene un rendimiento casi tan bueno, la KLT no se utiliza generalmente en la práctica.

4. La transformada discreta del coseno (DCT) se discute en detalle en la Sección 4.6. Esta importante transformada es casi tan eficiente como la KLT en términos de compactación de la energía; pero utiliza una base fija, independiente de los datos. También existen métodos rápidos para el cálculo de la DCT. Este método es utilizado por JPEG[7] y MPEG de audio.

4.5.2. Transformada de Walsh–Hadamard

Como se mencionó anteriormente (página 298), esta transformación tiene una eficiencia de compresión baja, por lo que no se utiliza mucho en la práctica. Sin embargo, es rápida, ya que puede ser calculada sólo con adiciones, sustracciones, y ocasionalmente desplazamientos a la derecha (en sustitución de una división por una potencia de 2).

Dado un bloque de $N \times N$ píxeles P_{xy} (donde N debe ser una potencia de 2, $N = 2^n$), su WHT

[7] Miguel Hernández Cabronero aclaró que JPEG no comprime audio, pero emplea la DCT en la compresión de imágenes.

bidimensional y su WHT inversa se define mediante las Ecuaciones (4.6) y (4.7):

$$\begin{aligned} H(u,v) &= \sum_{x=0}^{N-1}\sum_{y=0}^{N-1} p_{xy}\, g(x,y,u,v) \\ &= \frac{1}{N}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} p_{xy}\,(-1)^{\sum_{i=0}^{n-1}[b_i(x)p_i(u)+b_i(y)p_i(v)]}, \end{aligned} \tag{4.6}$$

$$\begin{aligned} P_{xy} &= \sum_{u=0}^{N-1}\sum_{v=0}^{N-1} H(u,v)\, h(x,y,u,v) \\ &= \frac{1}{N}\sum_{u=0}^{N-1}\sum_{v=0}^{N-1} H(u,v)\,(-1)^{\sum_{i=0}^{n-1}[b_i(x)p_i(u)+b_i(y)p_i(v)]}, \end{aligned} \tag{4.7}$$

donde $H(u,v)$ es el resultado de la transformada (i.e., los coeficientes de la WHT), la cantidad $b_i(u)$ es el bit i de la representación binaria del entero u, y $p_i(u)$ se define en términos de $b_j(u)$ mediante la Ecuación (4.8):

$$\begin{aligned} p_0(u) &= b_{n-1}(u), \\ p_1(u) &= b_{n-1}(u) + b_{n-2}(u), \\ p_2(u) &= b_{n-2}(u) + b_{n-3}(u), \\ &\vdots \\ p_{n-1}(u) &= b_1(u) + b_0(u). \end{aligned} \tag{4.8}$$

(Recuérdese que n se define arriba como $N = 2^n$.) Como ejemplo, consideremos $u = 6 = 110_2$. Los bits cero, uno, y dos de 6 son: 0, 1 y 1, respectivamente, de modo que $b_0(6) = 0$, $b_1(6) = 1$, y $b_2(6) = 1$.

Las cantidades $g(x,y,u,v)$ y $h(x,y,u,v)$ reciben el nombre de *kernels (núcleos)* (o *imágenes base*) de la WHT. Estas matrices son idénticas. Sus elementos son sólo $+1$ y -1, y se multiplican por el factor $\frac{1}{N}$. Como resultado, la transformada WHT consiste en multiplicar cada píxel de imagen por $+1$ o -1, sumando, y dividiendo la suma por N. Como $N = 2^n$ es una potencia de 2, la división por aquél se puede hacer mediante un desplazamiento de n posiciones a la derecha.

Los kernels de WHT se muestran en forma gráfica, para $N = 4$, en la Figura 4.19, donde el blanco denota $+1$ y el negro denota -1 (el factor $\frac{1}{N}$ es ignorado). Las filas y columnas de los bloques en esta figura corresponden a valores de u y v, de 0 a 3, respectivamente. Las filas y columnas dentro de cada bloque corresponde a los valores de x e y, de 0 a 3, respectivamente. El número de cambios de signo dentro de una fila o una columna de una matriz se llama *secuencia* de la fila o de la columna. Las filas y columnas de la figura están ordenadas según el incremento de la secuencia. Algunos autores muestran figuras similares —pero sin ordenar—, porque esta transformación fue definida por Walsh y por Hadamard por caminos ligeramente diferentes (véase [Gonzalez y Woods 92], para obtener más información).

La compresión de una imagen con la WHT se realiza de manera similar a la DCT, excepto que se utilizan las Ecuaciones (4.6) y (4.7), en lugar de las Ecuaciones (4.15) y (4.16).

◇ **Ejercicio 4.8 (sol. en pág. 1079):** Utilícese el software matemático adecuado para calcular y mostrar las imágenes base de la WHT para $N = 8$.

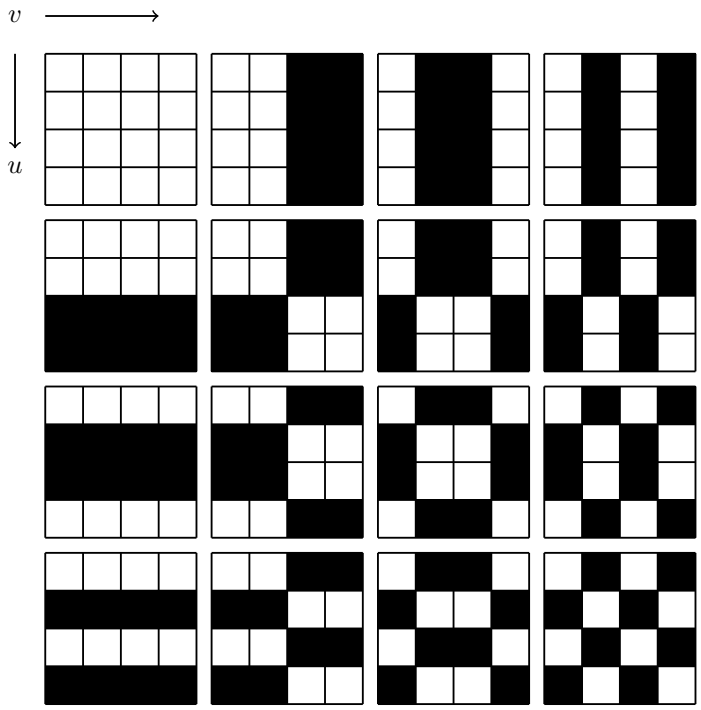


Figura 4.19: El kernel de la WHT ordenado para $N = 4$.

4.5.3. La transformada de Haar

La transformada de Haar [Stollnitz et al. 95] y [Stollnitz et al. 96] se basa en las funciones de Haar $h_k(x)$, que están definidas para $x \in [0,1]$ y para $k = 0, 1, \ldots, N-1$, donde $N = 2^n$. Su aplicación se discute también en el Capítulo 5.

Antes de discutir la transformación misma, tenemos que mencionar que cualquier entero k puede expresarse como la suma $k = 2^p + q - 1$, donde $0 \le p \le n-1$, $q = 0$ ó 1 (para $p = 0$), y $1 \le q \le 2^p$ (para $p \ne 0$). Para $N = 4 = 2^2$, por ejemplo, obtenemos $0 = 2^0 + 0 - 1$, $1 = 2^0 + 1 - 1$, $2 = 2^1 + 1 - 1$, y $3 = 2^1 + 2 - 1$.

Las funciones base de Haar se definen ahora como:

$$h_0(x) \stackrel{\text{def}}{=} h_{00}(x) = \frac{1}{\sqrt{N}}, \quad \text{para } 0 \le x \le 1, \tag{4.9}$$

y

$$h_k(x) \stackrel{\text{def}}{=} h_{pq}(x) = \frac{1}{\sqrt{N}} \begin{cases} 2^{p/2}, & \frac{q-1}{2^p} \le x < \frac{q-1/2}{2^p}, \\ -2^{p/2}, & \frac{q-1/2}{2^p} \le x < \frac{q}{2^p}, \\ 0, & \text{en otro caso para } x \in [0,1]. \end{cases} \tag{4.10}$$

Ahora puede construirse la matriz de la transformada de Haar —$\mathbf{A}_N$— de orden $N \times N$. Un elemento general i, j de esta matriz es la función base $h_i(j)$, donde $i = 0, 1, \ldots, N-1$ y $j = {}^0/_N, {}^1/_N, \ldots {}^{(N-1)}/_N$. Por ejemplo:

$$\mathbf{A}_2 = \begin{pmatrix} h_0({}^0/_2) & h_0({}^1/_2) \\ h_1({}^0/_2) & h_1({}^1/_2) \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{4.11}$$

(recordemos que $i = 1$ implica que $p = 0$ y $q = 1$). La Figura 4.20 muestra el código para calcular esta matriz para cualquier N, y también las imágenes base de Haar para $N = 8$.

◇ **Ejercicio 4.9 (sol. en pág. 1079):** Calcúlense los coeficientes de las matrices de Haar $\mathbf{A}_4$ y $\mathbf{A}_8$.

Dado un bloque de imágenes $\mathbf{P}$ de orden $N \times N$, donde $N = 2^n$, su transformada de Haar es la matriz producto $\mathbf{A}_N\mathbf{P}\mathbf{A}_N$ (Sección 5.6).

4.5.4. La transformada de Karhunen-Loève

La transformada de Karhunen–Loève (también llamada transformada de Hotelling) tiene la mejor eficiencia en el sentido de la compactación de la energía; pero por las razones mencionadas anteriormente, tiene un valor más teórico que práctico. Dada una imagen, la dividimos en k bloques de n píxeles cada uno; donde n es típicamente 64 pero puede tener otros valores, y k depende del tamaño de la imagen. Consideramos los vectores de los bloques y los denotamos por $\mathbf{b}^{(i)}$, para $i = 1, 2, \ldots, k$. El vector promedio es $\overline{\mathbf{b}} = \left(\sum_i \mathbf{b}^{(i)}\right)/k$. Se define un nuevo conjunto de vectores $\mathbf{v}^{(i)} = \mathbf{b}^{(i)} - \overline{\mathbf{b}}$, provocando que el promedio $\left(\sum \mathbf{v}^{(i)}\right)/k$ sea cero. Denotamos la matriz de la transformada KLT de $n \times n$ que estamos buscando por $\mathbf{A}$. El resultado de la transformación de un vector $\mathbf{v}^{(i)}$ es el vector de pesos $\mathbf{w}^{(i)} = \mathbf{A}\mathbf{v}^{(i)}$. El promedio de los $\mathbf{w}^{(i)}$ es también cero. Ahora construimos una matriz $\mathbf{V}$ cuyas columnas son los vectores $\mathbf{v}^{(i)}$ y otra matriz $\mathbf{W}$ cuyas columnas son los vectores de pesos $\mathbf{w}^{(i)}$:

$$\mathbf{V} = \left(\mathbf{v}^{(1)}, \mathbf{v}^{(2)}, \ldots, \mathbf{v}^{(k)}\right), \quad \mathbf{W} = \left(\mathbf{w}^{(1)}, \mathbf{w}^{(2)}, \ldots, \mathbf{w}^{(k)}\right).$$

Las matrices $\mathbf{V}$ y $\mathbf{W}$ tienen n filas y k columnas cada una. De la definición de $\mathbf{w}^{(i)}$, obtenemos: $\mathbf{W} = \mathbf{A} \cdot \mathbf{V}$.

Los n vectores de coeficientes $\mathbf{c}^{(j)}$ de la transformada de Karhunen–Loève están dados por:

$$\mathbf{c}^{(j)} = \left(\omega_j^{(1)}, \omega_j^{(2)}, \ldots, \omega_j^{(k)}\right), \quad j = 1, 2, \ldots, n.$$

En consecuencia, el vector $\mathbf{c}^{(j)}$ se compone de los j-avos elementos de todos los vectores de peso $\mathbf{w}^{(i)}$, para $i = 1, 2, \ldots, k$ ($\mathbf{c}^{(j)}$ es la coordenada j-ésima de los vectores $\mathbf{w}^{(i)}$).

Ahora vamos a examinar los elementos de la matriz del producto $\mathbf{W} \cdot \mathbf{W}^T$ (esta es una matriz de $n \times n$). Un elemento general en una fila a y una columna b de esta matriz es la suma de los productos:

$$\left(\mathbf{W} \cdot \mathbf{W}^T\right)_{ab} = \sum_{i=1}^{k} \omega_a^{(i)} \omega_b^{(i)} = \sum_{i=1}^{k} c_i^{(a)} c_i^{(b)} = \mathbf{c}^{(a)} \bullet \mathbf{c}^{(b)}, \quad \text{para } a, b \in [1, n]. \tag{4.12}$$

El hecho de que el promedio de cada $\mathbf{w}^{(i)}$ sea cero implica que un elemento genérico de la diagonal $\left(\mathbf{W} \cdot \mathbf{W}^T\right)_{jj}$ de la matriz del producto es la varianza (hasta un factor k) del j-ésimo elemento (o j-ésima coordenada) de los vectores $\mathbf{w}^{(i)}$. Ésto, por supuesto, es la varianza del vector de coeficientes $\mathbf{c}^{(j)}$.

◇ **Ejercicio 4.10 (sol. en pág. 1081):** Demuéstrese por qué es ésto verdad.

Los elementos de fuera de la diagonal de $\left(\mathbf{W} \cdot \mathbf{W}^T\right)$ son las covarianzas de los vectores $\mathbf{w}^{(i)}$ tales que dicho elemento $\left(\mathbf{W} \cdot \mathbf{W}^T\right)_{ab}$ es la covarianza de las a-ésima y b-ésima coordenadas de los $\mathbf{w}^{(i)}$'s. La Ecuación (4.12) muestra que esto es también el producto punto $\mathbf{c}^{(a)} \cdot \mathbf{c}^{(b)}$. Uno de los principales objetivos de la imagen transformada es descorrelacionar las coordenadas de los vectores, y la teoría de la probabilidad nos dice que dos coordenadas están descorrelacionadas si su covarianza es cero (el otro objetivo es la compactación de la energía, pero los dos objetivos van mano a mano). Por lo

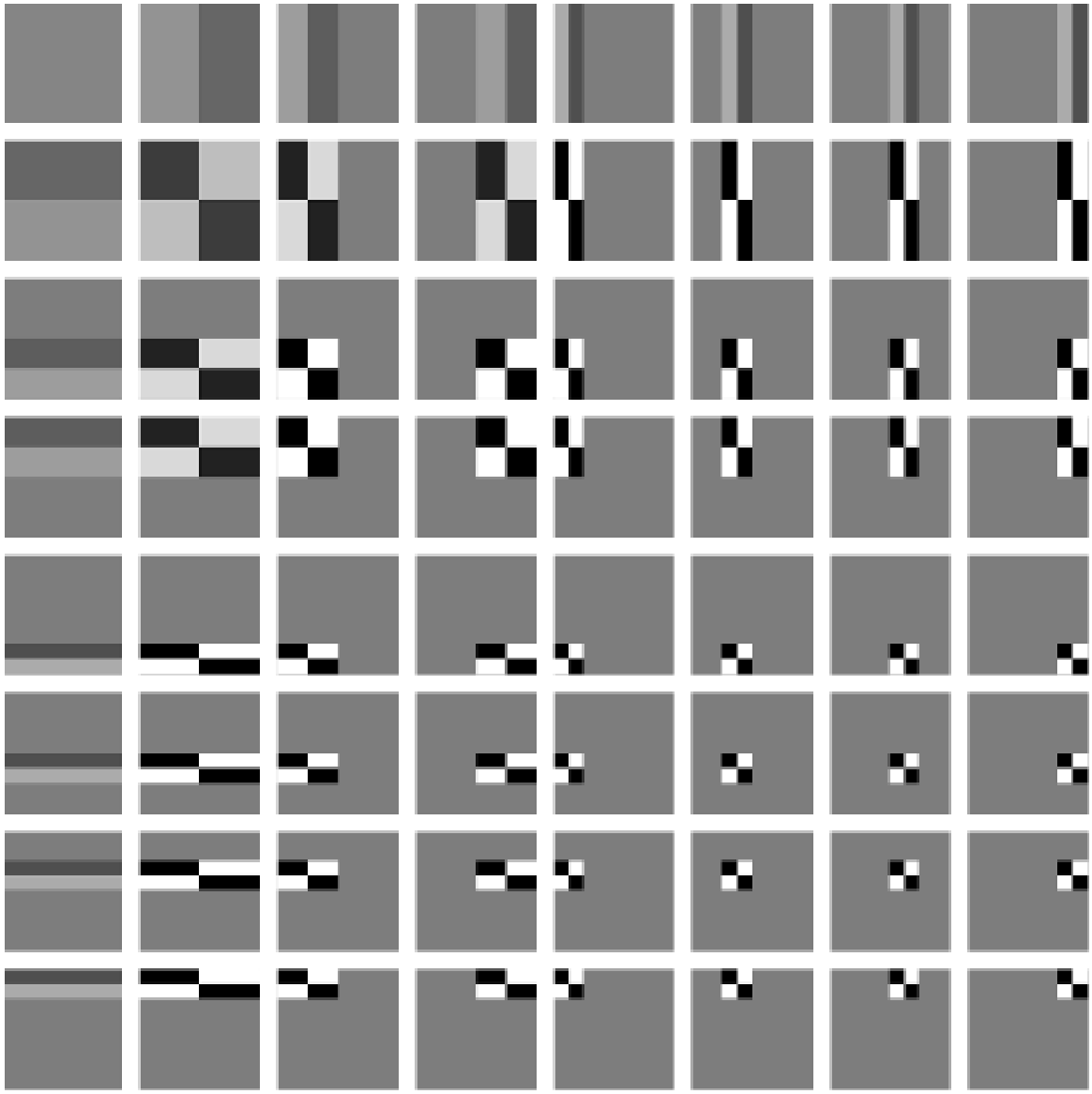


Figura 4.20: Imágenes básicas de la transformada de Haar para $n = 8$.

```
Needs["GraphicsImage‘"] (* Dibuja los coeficientes Haar 2D *)
n=8;
h[k_,x_]:=Module[{p,q}, If[k==0, 1/Sqrt[n], (* h_0(x) *)
 p=0; While[2^p<=k ,p++]; p--; q=k-2^p+1; (* si k>0,calc. p, q *)
 If[(q-1)/(2^p)<=x && x<(q-.5)/(2^p),2^(p/2),
  If[(q-.5)/(2^p)<=x && x<q/(2^p),-2^(p/2),0]]]];
HaarMatrix=Table[h[k,x], {k,0,7}, {x,0,7/n,1/n}] //N;
HaarTensor=Array[Outer[Times, HaarMatrix[[#1]],HaarMatrix[[#2]]]&,
{n,n}];
Show[GraphicsArray[Map[GraphicsImage[#, {-2,2}]&, HaarTensor,{2}]]]
```

Código para la Figura 4.20.

tanto, nuestro objetivo es encontrar una matriz de transformación $\mathbf{A}$ tal que el producto $\mathbf{W}\cdot\mathbf{W}^T$ sea diagonal.

De la definición de la matriz $\mathbf{W}$ obtenemos:

$$\mathbf{W}\cdot\mathbf{W}^T=(\mathbf{AV})\cdot(\mathbf{AV})^T=\mathbf{A}\left(\mathbf{V}\cdot\mathbf{V}^T\right)\mathbf{A}^T.$$

La matriz $\mathbf{V}\cdot\mathbf{V}^T$ es simétrica, y sus elementos son las covarianzas de las coordenadas de los vectores $\mathbf{v}^{(i)}$, i.e.:

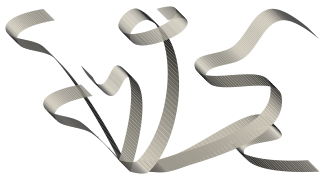
$$\left(\mathbf{V}\cdot\mathbf{V}^T\right)_{ab}=\sum_{i=1}^{k}v_a^{(i)}v_b^{(i)},\quad \text{para } a,b\in[1,n]\,.$$

Puesto que $\mathbf{V}\cdot\mathbf{V}^T$ es simétrica, sus vectores propios son ortogonales. Por lo tanto, normalizamos estos vectores (es decir, los hacemos ortonormales) y los elegimos para que sean las filas de la matriz $\mathbf{A}$. Esto produce el resultado:

$$\mathbf{W}\cdot\mathbf{W}^T=\mathbf{A}\left(\mathbf{V}\cdot\mathbf{V}^T\right)\mathbf{A}^T=\begin{pmatrix}\lambda_1&0&0&\cdots&0\\0&\lambda_2&0&\cdots&0\\0&0&\lambda_3&\cdots&0\\\vdots&\vdots&&\vdots&\vdots\\0&0&\cdots&0&\lambda_n\end{pmatrix}.$$

Esta elección de $\mathbf{A}$ tiene como efecto una matriz diagonal $\mathbf{W}\cdot\mathbf{W}^T$ cuyos elementos diagonales son los valores propios de $\mathbf{V}\cdot\mathbf{V}^T$. La matriz $\mathbf{A}$ es la matriz de la transformada de Karhunen–Loève; sus filas son los vectores base de la KLT, y las energías (varianzas) de los vectores transformados, son los valores propios $\lambda_1,\lambda_2,\ldots,\lambda_n$ de $\mathbf{V}\cdot\mathbf{V}^T$.

Los vectores base de la KLT se calculan a partir de los píxeles de la imagen original y son, por lo tanto, dependientes de los datos. En un método de compresión práctico, estos vectores tienen que ser incluidos en el flujo de datos comprimidos —para el uso del decodificador—, y ésto, combinado con el hecho de que no se ha descubierto ningún método rápido para el cálculo de la KLT, hace esta transformada menos ideal para aplicaciones prácticas.



4.6. La transformada discreta del coseno

Esta importante transformada (DCT, para abreviar) tuvo su origen en el documento [Ahmed et al. 74] y ha sido utilizada y estudiada ampliamente desde entonces. Debido a su importancia para la compresión de datos, la DCT se trata aquí en detalle. La Sección 4.6.1 introduce las expresiones matemáticas para la DCT en una dimensión y dos dimensiones sin ninguna base teórica ni justificaciones. A continuación, se muestra con varios ejemplos el uso de la transformada y sus ventajas para la compresión de datos. Las Secciones 4.6.2 y 4.6.3 cubren la teoría de la DCT y discuten sus dos interpretaciones como una rotación y como una base de vectores del espacio. La Sección 4.6.4 introduce los cuatro tipos de DCT, y la Sección 8.15.2 trata la DCT tridimensional. La Sección 4.6.5 describe formas de acelerar el

cálculo de la DCT, y la Sección 4.6.7 es un breve análisis de la simetría de la DCT y de cómo puede ser explotada para una implementación en hardware. Seguidamente, hay varias secciones de antecedentes importantes en la materia. La Sección 4.6.8 explica la descomposición QR de las matrices. La Sección 4.6.9 introduce el concepto de espacios vectoriales y sus bases. La Sección 4.6.10 muestra cómo se relaciona la rotación realizada por la DCT con las rotaciones generales en tres dimensiones. Por último, se introduce la transformada discreta del seno en la Sección 4.6.11, junto con las razones que lo hacen inadecuado para la compresión de datos.

4.6.1. Introducción

La DCT unidimensional viene dada por:

$$G_f = \sqrt{\frac{2}{n}} C_f \sum_{t=0}^{n-1} p_t \cos\left[\frac{(2t+1)\, f\pi}{2n}\right], \tag{4.13}$$

donde

$$C_f = \begin{cases} \frac{1}{\sqrt{2}}, & f = 0, \\ 1, & f > 0, \end{cases} \qquad \text{para } f = 0, 1, \ldots, n-1.$$

La entrada es un conjunto de n valores de datos p_t (píxeles, muestras de audio, u otros datos), y la salida es un conjunto de n *coeficientes de la transformada* DCT (o pesos) G_f. El primer coeficiente G_0 se llama el coeficiente DC, y el resto se conocen como coeficientes AC (éstos términos se han heredado de la ingeniería eléctrica, donde se utilizan para "corriente continua —*direct current*—" y "corriente alterna —*alternating current*—"). Tenga en cuenta que los coeficientes son números reales, incluso si los datos de entrada están formados por enteros. De manera similar, los coeficientes pueden ser positivos o negativos incluso si los datos de entrada se componen solamente de números no negativos. Este cálculo es sencillo, pero lento (la Sección 4.6.5 discute versiones más rápidas). El decodificador introduce los coeficientes de la DCT en conjuntos de n elementos y utiliza la DCT *inversa* (IDCT) para reconstruir los valores de los datos originales (también en grupos de n). La IDCT unidimensional viene dada por:

$$p_t = \sqrt{\frac{2}{n}} \sum_{j=0}^{n-1} C_j G_j \cos\left[\frac{(2t+1)\, j\pi}{2n}\right], \qquad \text{para } t = 0, 1, \ldots, n-1. \tag{4.14}$$

La característica más importante de la DCT, aquella que hace que sea tan útil en la compresión de datos, es que toma datos correlacionados de la entrada y concentra su energía en sólo unos pocos (los primeros) coeficientes de transformada. Si los datos de entrada están formados por cantidades correlacionadas, entonces la mayoría de los n coeficientes de transformada producidos por la DCT son ceros o números pequeños, y sólo unos pocos son grandes (por lo general los primeros). Veremos que el coeficientes de los primeros contienen la información importante de la imagen (frecuencias bajas) y los coeficientes posteriores contienen la información menos importante (frecuencias altas) de la imagen. La compresión de datos con la DCT, por lo tanto, se realiza mediante la cuantificación de los coeficientes. Los valores pequeños se cuantifican toscamente (posiblemente siempre a cero), y los grandes pueden ser cuantificados finamente al entero más cercano. Después de la cuantificación, los coeficientes (o códigos de tamaño variable asignados a los coeficientes) se escriben en la secuencia de datos comprimidos. La descompresión se realiza mediante el cálculo de la DCT inversa de los coeficientes cuantificados. Ésto produce ítems de datos que no son idénticos a los originales, pero no son muy diferentes.

En la práctica, los datos a ser comprimidos se dividen en conjuntos de n elementos cada uno, calculándose la transformada DCT y la cuantificación de la misma para cada conjunto, individualmente. El valor de n es crítico. Los valores pequeños de n, tales como 3, 4 ó 6 producen muchos grupos

```
n=8;
p={12.,10.,8.,10.,12.,10.,8.,11.};
c=Table[If[t==1, 0.7071, 1], {t,1,n}];
dct[i_]:=Sqrt[2/n]c[[i+1]]Sum[p[[t+1]]Cos[(2t+1)i Pi/16],{t,0,n-1}];
q=Table[dct[i],{i,0,n-1}] (* usa coeficientes DCT precisos *)
q={28,0,0,2,3,-2,0,0}; (* o usa coeficientes DCT cuantificados *)
idct[t_]:=Sqrt[2/n]Sum[c[[j+1]]q[[j+1]]Cos[(2t+1)j Pi/16],{j,0,n-1}];
ip=Table[idct[t],{t,0,n-1}]
```

Figura 4.21: Experimentos con la DCT unidimensional.

pequeños de ítems de datos. Estos conjuntos tan pequeños se transforman en un pequeño conjunto de coeficientes, donde la energía de los datos originales se concentra en unos pocos coeficientes, ¡pero sólo hay unos pocos coeficientes en dicho conjunto! Por consiguiente, no hay suficientes coeficientes pequeños para cuantificar. Los valores grandes de n dan lugar a unas pocas series grandes de datos. El problema en este caso es, que los ítems de datos individuales de un conjunto grande normalmente no están correlacionados y, por lo tanto, dan lugar a un conjunto de coeficientes de la transformada donde todos los ellos son grandes. La experiencia indica que $n = 8$ es un buen valor, y la mayor parte de los métodos de compresión de datos que emplean la DCT utilizan este valor de n.

El experimento siguiente ilustra el poder de la DCT en una dimensión. Comenzamos con el conjunto: $p = (12, 10, 8, 10, 12, 10, 8, 11)$, de ocho ítems de datos correlacionados; les aplicamos la DCT unidimensional, y encontramos que produce los ocho coeficientes:

$$28{,}6375,\ 0{,}571202,\ 0{,}46194,\ 1{,}757,\ 3{,}18198,\ -1{,}72956,\ 0{,}191342,\ -0{,}308709.$$

Estos pueden ser introducidos en la IDCT y transformados por ella para reconstruir con precisión los datos originales (excepto por los pequeños errores causados por los límites de precisión de la máquina). Nuestro objetivo, sin embargo, es comprimir los datos mediante la cuantificación de los coeficientes. En primer lugar, los cuantificamos a 28,6, 0,6, 0,5, 1,8, 3,2, $-1{,}8$, 0,2, $-0{,}3$, y aplicamos la IDCT obteniendo:

$$12{,}0254,\ 10{,}0233,\ 7{,}96054,\ 9{,}93097,\ 12{,}0164,\ 9{,}99321,\ 7{,}94354,\ 10{,}9989.$$

Entonces cuantificamos los coeficientes aún más, a 28, 1, 1, 2, 3, 2, 0, 0, y aplicamos la IDCT obteniendo:

$$12{,}1883,\ 10{,}2315,\ 7{,}74931,\ 9{,}20863,\ 11{,}7876,\ 9{,}54549,\ 7{,}82865,\ 10{,}6557.$$

Finalmente cuantificamos los coeficientes a 28, 0, 0, 2, 3, -2, 0, 0, y se los proporcionamos de nuevo a la IDCT, que produce la secuencia:

$$11{,}236,\ 9{,}62443,\ 7{,}66286,\ 9{,}57302,\ 12{,}3471,\ 10{,}0146,\ 8{,}05304,\ 10{,}6842.$$

donde la mayor diferencia entre un valor original (12) y uno reconstruido (11,236) es 0,764 (ó el 6,4 % de 12). El código que hace todo esto se muestra en la Figura 4.21.

Parece mágico que los ocho elementos de datos originales puedan ser reconstruidos con una precisión tan alta, con tan sólo cuatro coeficientes de la transformada. La explicación, sin embargo, se basa en los siguientes argumentos en lugar de en la magia: (1) A la IDCT se le proporcionan los ocho coeficientes de la transformada, por lo que conoce las posiciones —no sólo los valores— de los coeficientes distintos de cero. (2) Los primeros pocos coeficientes (los grandes) contienen la información importante de los ítems de datos originales. Los coeficientes pequeños —los que están cuantificados más duramente—, contienen información menos importante (en el caso de las imágenes, contienen los detalles de imagen). (3) Los datos originales están correlacionados.

(a) (b)

Figura 4.22: (a) Entrada unidimensional. (b) Su DFT inversa.

El siguiente experimento ilustra el rendimiento de la DCT cuando se aplica a ítems de datos descorrelacionados. Dados los ocho elementos de datos descorrelacionados: −12, 24, −181, 209, 57,8, 3, −184, y −250 , su DCT produce

$$-117{,}803,\ 166{,}823,\ -240{,}83,\ 126{,}887,\ 121{,}198,\ 9{,}02198,\ -109{,}496,\ -185{,}206.$$

Cuando estos coeficientes se cuantifican a (−120, 170, −240, 125, 120, 9, −110, −185) y se introducen en la IDCT, el resultado es:

$$-12{,}1249,\ 25{,}4974,\ -179{,}852,\ 208{,}237,\ 55{,}5898,\ 0{,}364874,\ -185,42,\ -251{,}701,$$

donde la diferencia máxima (entre 3 y 0,364874) es 2,63513 ó el 88 % de 3. Obviamente, incluso con una cuantificación fina semejante la reconstrucción no es tan buena como con los datos correlacionados.

◇ **Ejercicio 4.11 (sol. en pág. 1081):** Calcúlese la DCT unidimensional [Ecuación (4.13)] de los ocho valores correlacionados: 11, 22, 33, 44, 55, 66, 77 y 88. Muéstrese cómo cuantificarlos y calcúlese su IDCT a partir de la Ecuación (4.14).

Un familiar importante de la DCT es la transformada de Fourier (Sección 5.1), que también tiene una versión discreta denominada DFT. La DFT tiene importantes aplicaciones, pero no logra buenos resultados en la compresión de datos, ya que asume que los datos a transformar son periódicos.

El siguiente ejemplo ilustra la diferencia en rendimiento entre la DCT y la DFT. Empezamos con la sencilla secuencia de ocho números altamente correlacionados: (8, 16, 24, 32, 40, 48, 56, 64). Se muestran gráficamente en la Figura 4.22a. La aplicación de la DCT a los mismos produce: (100, −52, 0, −5, 0, −2, 0, 0,4). Cuando ésto se cuantifica a (100, −52, 0, −5, 0, 0, 0, 0) y se realiza la transformación inversa, produce (8, 15, 24, 32, 40, 48, 57, 63), una secuencia casi idéntica a la entrada original. Aplicando la DFT a la misma de entrada, por el contrario, se obtiene (36, 10, 10, 6, 6, 4, 4, 4). Cuando ésto se cuantifica a (36, 10, 10, 6, 0, 0, 0, 0) y se realiza la transformación inversa, produce (24, 12, 20, 32, 40, 51, 59, 48). Esta salida se muestra en la Figura 4.22b, e ilustra la tendencia de la transformada de Fourier a producir un resultado periódico.

La DCT unidimensinal puede utilizarse para comprimir datos unidimensionales, tales como muestras de audio. Este capítulo, sin embargo, discute la compresión de imágenes que se basa en la correlación bidimensional de píxeles (un píxel tiende a parecerse a todos sus vecinos más próximos, no sólo a los de su fila). Ésta es la razón por la que —en la práctica— los métodos de compresión de imágenes utilizan la DCT bidimensional. Esta versión de la DCT se aplica a pedazitos (bloques de datos) de la imagen. Se calcula aplicando la DCT unidimensional para cada fila de un bloque de datos, y después para cada columna del resultado. A causa de la manera especial como se calcula la DCT bidimensional, se dice que es separable en las dos dimensiones. Debido a que se aplica a los bloques de una imagen, la denominamos "transformada por bloques". Se define así:

$$G_{ij} = \sqrt{\frac{2}{m}}\sqrt{\frac{2}{n}} C_i C_j \sum_{x=0}^{n-1}\sum_{y=0}^{m-1} p_{xy} \cos\left[\frac{(2y+1)\,j\pi}{2m}\right] \cos\left[\frac{(2x+1)\,i\pi}{2n}\right], \tag{4.15}$$

para $0 \leq i \leq n-1$ y $0 \leq j \leq m-1$, y para C_i y C_j definidos en la Ecuación (4.13). El primer coeficiente —G_{00}— se denomina, de nuevo, "coeficiente DC", y el resto de los coeficientes reciben el nombre de "coeficientes AC".

La imagen se divide en bloques de $n \times m$ píxeles p_{xy} (con típicamente $n = m = 8$), y se utiliza la Ecuación (4.15) para producir un bloque de $n \times m$ coeficientes G_{ij} de la DCT, para cada bloque de píxeles. Los coeficientes son entonces cuantificados, lo que produce ciertas pérdidas, pero una compresión altamente eficiente. El decodificador reconstruye un bloque de valores de datos cuantificados calculando la IDCT cuya definición es:

$$p_{xy} = \sqrt{\frac{2}{m}}\sqrt{\frac{2}{n}}\sum_{x=0}^{n-1}\sum_{y=0}^{m-1} C_i C_j G_{ij} \cos\left[\frac{(2x+1)\, i\pi}{2n}\right] \cos\left[\frac{(2y+1)\, j\pi}{2m}\right], \tag{4.16}$$

donde

$$C_f = \begin{cases} \frac{1}{\sqrt{2}}, & f = 0, \\ 1, & f > 0, \end{cases}$$

para $0 \leq x \leq n-1$ y $0 \leq y \leq m-1$. A continuación mostramos una manera de comprimir una imagen entera con la DCT en varias etapas o pasos:

1. Se divide la imagen en k bloques de 8×8 píxeles cada uno. Los píxeles se denotan p_{xy}. Si el número de filas de la imagen (columnas) no es divisible por 8, la fila más inferior (la columna del extremo derecho) se duplica tantas veces como sea necesario.

2. La DCT bidimensional [Ecuación (4.15)] se aplica a cada bloque B_i. El resultado es un bloque (que llamaremos vector) $W^{(i)}$ formado por los 64 coeficientes $w_j^{(i)}$ de la transformada (donde $j = 0, 1, \ldots, 63$). Los k vectores $W^{(i)}$ se convierten en las filas de la matriz $\mathbf{W}$:

$$\mathbf{W} = \begin{bmatrix} \omega_0^{(1)} & \omega_1^{(1)} & \ldots & \omega_{63}^{(1)} \\ \omega_0^{(2)} & \omega_1^{(2)} & \ldots & \omega_{63}^{(2)} \\ \vdots & \vdots & & \\ \omega_0^{(k)} & \omega_1^{(k)} & \ldots & \omega_{63}^{(k)} \end{bmatrix}.$$

3. Las 64 columnas de $\mathbf{W}$ se denotan por $C^{(0)}$, $C^{(1)}$,..., $C^{(63)}$. Los k elementos de $C^{(j)}$ son: $\left(\omega_j^{(1)}, \omega_j^{(2)}, \ldots, \omega_j^{(k)}\right)$. El primer vector coeficiente $C^{(0)}$ está formado por los k coeficientes DC.

4. Cada vector $C^{(j)}$ se cuantifica por separado para producir un vector $Q^{(j)}$ de coeficientes cuantificados (JPEG hace ésto de manera diferente; véase la Sección 4.8.3). A continuación se escriben los elementos de $Q^{(j)}$ en la cadena comprimida. En la práctica, se asignan códigos de tamaño variable a los elementos, y se escriben los códigos, en lugar de los propios elementos, en el flujo comprimido. A veces, como en el caso de JPEG, se asignan códigos de tamaño variable a *runs* de coeficientes cero, para lograr una mejor compresión.

En la práctica, la DCT se utiliza para la compresión con pérdida. Para la compresión sin pérdidas (donde los coeficientes DCT no se cuantifican) la DCT es ineficiente, pero todavía puede utilizarse —al menos en teoría—, porque: (1) la mayor parte de los coeficientes son números pequeños y (2) las rachas (*runs*) de coeficientes cero son frecuentes. Sin embargo, los coeficientes pequeños son números reales, no enteros, por lo que no queda claro cómo escribirlos con la máxima precisión en la secuencia de datos comprimidos y aún obtener compresión. Otros métodos de compresión de imágenes se adaptan mejor a la compresión de imágenes sin pérdidas.

12	10	8	10	12	10	8	11
11	12	10	8	10	12	10	8
8	11	12	10	8	10	12	10
10	8	11	12	10	8	10	12
12	10	8	11	12	10	8	10
10	12	10	8	11	12	10	8
8	10	12	10	8	11	12	10
10	8	10	12	10	8	11	12

(a) Datos originales

81	0	0	0	0	0	0	0
0	1,57	0,61	1,90	0,38	1,81	0,20	0,32
0	0,61	0,71	0,35	0	0,07	0	0,02
0	1,90	0,35	4,76	0,77	3,39	0,25	0,54
0	0,38	0	0,77	8,00	0,51	0	0,07
0	1,81	0,07	3,39	0,51	1,57	0,56	0,25
0	0,20	0	0,25	0	0,56	0,71	0,29
0	0,32	0,02	0,54	0,07	0,25	0,29	0,90

(b) Coeficientes DCT

81	0	0	0	0	0	0	0
0	2	1	2	0	2	0	0
0	1	1	0	0	0	0	0
0	2	0	5	1	3	0	1
0	0	0	1	8	1	0	0
0	2	0	3	1	2	1	0
0	0	0	0	0	1	1	0
0	0	0	1	0	0	0	1

(c) Cuantificados

12,29	10,26	7,92	9,93	11,51	9,94	8,18	10,97
10,90	12,06	10,07	7,68	10,30	11,64	10,17	8,18
7,83	11,39	12,19	9,62	8,28	10,10	11,64	9,94
10,15	7,74	11,16	11,96	9,90	8,28	10,30	11,51
12,21	10,08	8,15	11,38	11,96	9,62	7,68	9,93
10,09	12,10	9,30	8,15	11,16	12,19	10,07	7,92
7,87	9,50	12,10	10,08	7,74	11,39	12,06	10,26
9,66	7,87	10,09	12,21	10,15	7,83	10,90	12,29

(d) Datos reconstruidos (bueno)

Tabla 4.23: DCT bidimensional de un bloque de valores correlacionados.

El decodificador lee los 64 coeficientes cuantificados de los vectores $Q^{(j)}$ de k elementos cada uno, los almacena como las columnas de una matriz, y considera las k filas de la matriz vectores peso $W^{(i)}$ de 64 elementos cada uno (nótese que estos $W^{(i)}$'s no son idénticos a los originales $W^{(i)}$'s debido a la cuantificación). A continuación, aplica la IDCT [Ecuación (4.16)] a cada vector peso, para reconstruir (aproximadamente) los 64 píxeles del bloque B_i. (De nuevo, JPEG hace ésto de forma diferente.)

Ilustramos el rendimiento de la DCT bidimensional, aplicándola a dos bloques de 8×8 valores. El primer bloque (Tabla 4.23a) está formado por valores enteros altamente correlacionados en el intervalo $[8, 12]$; y el segundo bloque, consta de valores aleatorios en el mismo rango. El primer bloque produce un gran coeficiente DC, seguido de pequeños coeficientes AC [incluyendo 20 ceros (Tabla 4.23b), donde los números negativos están subrayados]. Cuando los coeficientes son cuantificados (Tabla 4.23c), el resultado —que se muestra en la Tabla 4.23d— es muy similar a los valores originales. Por el contrario, los coeficientes para el segundo bloque (Tabla 4.24b) son exactamente un cero. Cuando es cuantificado (Tabla 4.24c) y se transforma de nuevo, muchos de los 64 resultados son muy diferentes de los valores originales (Tabla 4.24d).

◇ **Ejercicio 4.12 (sol. en pág. 1081):** Muéstrese por qué los 64 valores de la Tabla 4.23 están correlacionados.

El siguiente ejemplo ilustra la diferencia en el rendimiento de la DCT cuando se aplica a una imagen de tonos continuos y a una imagen de tonos discretos. Comenzamos con el patrón altamente correlacionado de la Tabla 4.25. Éste es un ejemplo idealizado de una imagen de tonos continuos, ya que los píxeles adyacentes difieren en una cantidad constante excepto el píxel (subrayado) de la fila 7, columna 7. Los 64 coeficientes de la DCT de este patrón se muestran en la Tabla 4.26. Es evidente que sólo hay unos pocos coeficientes dominantes. La Tabla 4.27 muestra los coeficientes después de hacer sido toscamente cuantificados, por lo que ¡sólo permanecen cuatro coeficientes no nulos! Los resultados tras aplicar la IDCT en estos coeficientes cuantificados se muestran en la Tabla 4.28. Es obvio que los cuatro coeficientes no nulos han reconstruido el patrón original en un alto grado. La única diferencia visible es en la fila 7, columna 7, que ha cambiado de 12 a 17,55 (marcado en ambas figuras). El código

8	10	9	11	11	9	9	12
11	8	12	8	11	10	11	10
9	11	9	10	12	9	9	8
9	12	10	8	8	9	8	9
12	8	9	9	12	10	8	11
8	11	10	12	9	12	12	10
10	10	12	10	12	10	10	12
12	9	11	11	9	8	8	12

(a) Datos originales

79,12	0,98	0,64	1,51	0,62	0,86	1,22	0,32
0,15	1,64	0,09	1,23	0,10	3,29	1,08	2,97
1,26	0,29	3,27	1,69	0,51	1,13	1,52	1,33
1,27	0,25	0,67	0,15	1,63	1,94	0,47	1,30
2,12	0,67	0,07	0,79	1,13	1,40	0,16	0,15
2,68	1,08	1,99	1,93	1,77	0,35	0	0,80
1,20	2,10	0,98	0,87	1,55	0,59	0,98	2,76
2,24	0,55	0,29	0,75	2,40	0,05	0,06	1,14

(b) Coeficientes DCT

79	1	1	2	1	1	1	0
0	2	0	1	0	3	1	3
1	0	3	2	0	1	2	1
1	0	1	0	2	2	0	10
20	1	0	1	0	10	0	0
3	1	2	2	2	0	0	1
1	2	1	1	2	1	1	3
2	1	0	1	2	0	0	1

(c) Cuantificados

7,59	9,23	8,33	11,88	7,12	12,47	6,98	8,56
12,09	7,97	9,3	11,52	9,28	11,62	10,98	12,39
11,02	10,06	13,81	6,5	10,82	8,28	13,02	7,54
8,46	10,22	11,16	9,57	8,45	7,77	10,28	11,89
9,71	11,93	8,04	9,59	8,04	9,7	8,59	12,14
10,27	13,58	9,21	11,83	9,99	10,66	7,84	11,27
8,34	10,32	10,53	9,9	8,31	9,34	7,47	8,93
10,61	9,04	13,66	6,04	13,47	7,65	10,97	8,89

(d) Datos reconstruidos (malo)

Tabla 4.24: DCT bidimensional de un bloque de valores aleatorios.

00	10	20	30	30	20	10	00
10	20	30	40	40	30	20	10
20	30	40	50	50	40	30	20
30	40	50	60	60	50	40	30
30	40	50	60	60	50	40	30
20	30	40	50	50	40	30	20
10	20	30	40	40	30	12	10
00	10	20	30	30	20	10	00

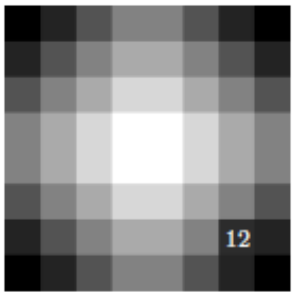


Tabla 4.25: Un patrón de tonos continuos.

239	1,19	−89,76	−0,28	1,00	−1,39	−5,03	−0,79
1,18	−1,39	0,64	0,32	−1,18	1,63	−1,54	0,92
−89,76	0,64	−0,29	−0,15	0,54	−0,75	0,71	−0,43
−0,28	0,32	−0,15	−0,08	0,28	−0,38	0,36	−0,22
1,00	−1,18	0,54	0,28	−1,00	1,39	−1,31	0,79
−1,39	1,63	−0,75	−0,38	1,39	−1,92	1,81	−1,09
−5,03	−1,54	0,71	0,36	−1,31	1,81	−1,71	1,03
−0,79	0,92	−0,43	−0,22	0,79	−1,09	1,03	−0,62

Tabla 4.26: Sus coeficientes de la DCT.

239	1	−90	0	0	0	0	0
0	0	0	0	0	0	0	0
−90	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Tabla 4.27: Cuantificación profunda de sólo 4 coeficientes distintos de cero.

0,65	9,23	21,36	29,91	29,84	21,17	8,94	0,30
9,26	17,85	29,97	38,52	38,45	29,78	17,55	8,91
21,44	30,02	42,15	50,70	50,63	41,95	29,73	21,09
30,05	38,63	50,76	59,31	59,24	50,56	38,34	29,70
30,05	38,63	50,76	59,31	59,24	50,56	38,34	29,70
21,44	30,02	42,15	50,70	50,63	41,95	29,73	21,09
9,26	17,85	29,97	38,52	38,45	29,78	17,55	8,91
0,65	9,23	21,36	29,91	29,84	21,17	8,94	0,30

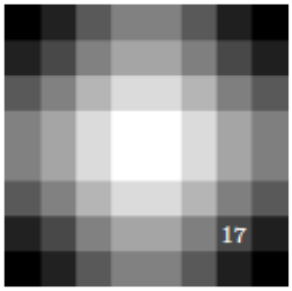


Tabla 4.28: Resultados de la IDCT.

00	10	00	00	00	00	00	10
00	00	10	00	00	00	10	00
00	00	00	10	00	10	00	00
00	00	00	00	10	00	00	00
00	00	00	00	10	00	00	00
00	00	00	00	10	00	00	00
00	00	00	00	10	00	00	00
00	00	00	00	10	00	00	00

Tabla 4.29: Una imagen de tonos discretos.

13,75	−3,11	−8,17	2,46	3,75	−6,86	−3,38	6,59
4,19	−0,29	6,86	−6,85	−7,13	4,48	1,69	−7,28
1,63	0,19	6,40	−4,81	−2,99	−1,11	−0,88	−0,94
−0,61	0,54	5,12	−2,31	1,30	−6,04	−2,78	3,05
−1,25	0,52	2,99	−0,20	3,75	−7,39	−2,59	1,16
−0,41	0,18	0,65	1,03	3,87	−5,19	−0,71	−4,76
0,68	−0,15	−0,88	1,28	2,59	−1,92	1,10	−9,05
0,83	−0,21	−0,99	0,82	1,13	−0,08	1,31	−7,21

Tabla 4.30: Sus coeficientes de la DCT.

13,75	−3	−8	2	3	−6	−3	6
4	−0	6	−6	−7	4	1	−7
1	0	6	−4	−2	−1	−0	−0
−0	0	5	−2	1	−6	−2	3
−1	0	2	−0	3	−7	−2	1
−0	0	0	1	3	−5	−0	−4
0	−0	−0	1	2	−1	1	−9
0	−0	−0	0	1	−0	1	−7

Tabla 4.31: Cuantificación ligera truncando a enteros.

−0,13	8,96	0,55	−0,27	0,27	0,86	0,15	9,22
0,32	0,22	9,10	0,40	0,84	−0,11	9,36	−0,14
0,00	0,62	−0,20	9,71	−1,30	8,57	0,28	−0,33
−0,58	0,44	0,78	0,71	10,11	1,14	0,44	−0,49
−0,39	0,67	0,07	0,38	8,82	0,09	0,28	0,41
0,34	0,11	0,26	0,18	8,93	0,41	0,47	0,37
0,09	−0,32	0,78	−0,20	9,78	0,05	−0,09	0,49
0,16	−0,83	0,09	0,12	9,15	−0,11	−0,08	0,01

Tabla 4.32: La IDCT. Malos resultados.

en Matlab para este cálculo se muestra en Figura 4.33.

Las Tablas 4.29 a 4.32 muestran el mismo proceso aplicado a un patrón en forma de Y, típica de una imagen de tonos discretos. La cuantificación que se muestra en la Tabla 4.31, es ligera. Los coeficientes sólo se han truncado al entero más cercano. Es fácil ver que la reconstrucción —mostrada en la Tabla 4.32— no es tan buena como antes. Las cantidades que deberían haber sido 10 se encuentran entre 8,96 y 10,11. Las cantidades que deberían haber sido cero, son tan grandes como 0,86. La conclusión es, que la DCT rinde bien con imágenes en tonos continuos, pero es menos eficiente cuando se aplica a una imagen de tonos discretos.

4.6.2. La DCT como base

Hasta ahora, la discusión se ha centrado en cómo utilizar la DCT para la compresión unidimensional y bidimensional de datos. El objetivo de esta sección y la siguiente, es mostrar por qué funciona la DCT de la manera que lo hace y cómo se han obtenido las Ecuaciones (4.13) y (4.15). Este tema se aborda desde dos direcciones diferentes. La primera interpretación de la DCT es como una base especial de un espacio vectorial n-dimensional. Mostramos que la transformación de un vector de datos dado $\mathbf{p}$ mediante la DCT es equivalente a representarlo en esta base especial que aísla las diversas frecuencias contenidas en el vector. Por consiguiente, los coeficientes de la DCT resultantes de la aplicación de la transformada DCT al vector $\mathbf{p}$ indican las distintas frecuencias en el vector. Las frecuencias más bajas contienen la información importante en $\mathbf{p}$, mientras que las frecuencias más altas corresponden a los detalles de los datos en $\mathbf{p}$ y son, por lo tanto, menos importantes. Es por ésto, que puede ser cuantificado toscamente.

La segunda interpretación de la DCT es como una rotación, como se muestra intuitivamente para $n = 2$ (puntos bidimensionales) en la Figura 4.16. Esta interpretación considera la DCT una matriz de rotación que hace girar un punto n-dimensional con coordenadas idénticas $(x, x, \ldots, x)$ en su ubicación original en el eje x, donde las coordenadas se convierten en $(\alpha, \epsilon_2, \ldots, \epsilon_n)$, siendo los diversos ϵ_i números

```
% 8x8 valores correlacionados
n=8; p=[00,10,20,30,30,20,10,00; 10,20,30,40,40,30,20,10; 20,30,40,50,50,40,30,20; ...
30,40,50,60,60,50,40,30; 30,40,50,60,60,50,40,30; 20,30,40,50,50,40,30,20; ...
10,20,30,40,40,30,12,10; 00,10,20,30,30,20,10,00];
figure(1), imagesc(p), colormap(gray), axis square, axis off
dct=zeros(n,n);
for j=0:7
  for i=0:7
    for x=0:7
      for y=0:7
dct(i+1,j+1)=dct(i+1,j+1)+p(x+1,y+1)*cos((2*y+1)*j*pi/16)*cos((2*x+1)*i*pi/16);
      end;
    end;
  end;
end;
dct=dct/4; dct(1,:)=dct(1,:)*0.7071; dct(:,1)=dct(:,1)*0.7071;
dct
quant=[239,1,-90,0,0,0,0,0; 0,0,0,0,0,0,0,0; -90,0,0,0,0,0,0,0; 0,0,0,0,0,0,0,0; ...
0,0,0,0,0,0,0,0; 0,0,0,0,0,0,0,0; 0,0,0,0,0,0,0,0; 0,0,0,0,0,0,0,0];
idct=zeros(n,n);
for x=0:7
  for y=0:7
    for i=0:7
      if i==0 ci=0.7071; else ci=1; end;
      for j=0:7
        if j==0 cj=0.7071; else cj=1; end;
        idct(x+1,y+1)=idct(x+1,y+1)+ ...
        ci*cj*quant(i+1,j+1)*cos((2*y+1)*j*pi/16)*cos((2*x+1)*i*pi/16);
      end;
    end;
  end;
end;
idct=idct/4;
idct
figure(2), imagesc(idct), colormap(gray), axis square, axis off
```

Figura 4.33: Código para el patrón altamente correlacionado.

pequeños o ceros. Ambas interpretaciones se ilustran para el caso $n = 3$, ya que éste es el mayor número de dimensiones donde es posible visualizar las transformaciones geométricas.

Para el caso especial $n = 3$, la Ecuación (4.13) se reduce a:

$$G_f = \sqrt{\frac{2}{3}} C_f \sum_{t=0}^{2} p_t \cos\left[\frac{(2t+1)\, f\pi}{6}\right], \quad \text{para } f = 0, 1, 2.$$

Ignorando temporalmente los factores de normalización $\sqrt{2/3}$ y C_f, ésto se puede escribir en notación matricial como:

$$\begin{bmatrix} G_0 \\ G_1 \\ G_2 \end{bmatrix} = \begin{bmatrix} \cos 0 & \cos 0 & \cos 0 \\ \cos \frac{\pi}{6} & \cos \frac{3\pi}{6} & \cos \frac{5\pi}{6} \\ \cos 2\frac{\pi}{6} & \cos 2\frac{3\pi}{6} & \cos 2\frac{5\pi}{6} \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \end{bmatrix} = \mathbf{D} \cdot \mathbf{p}.$$

Por consiguiente, la DCT de los tres valores de datos $\mathbf{p} = (p_0, p_1, p_2)$ se obtiene como el producto de la matriz $\mathbf{D}$ de la DCT y el vector $\mathbf{p}$. Podemos pensar, por lo tanto, en la DCT como el producto de una

θ	0,5236	1,5708	2,618
$\cos 0\theta$	1,0	1,0	1,0
$\cos 1\theta$	0,866	0	−0,866
$\cos 2\theta$	0,5	−1	0,5

Tabla 4.34: La matriz DCT para $n = 3$.

matriz DCT y un vector de datos, donde la matriz se construye de la siguiente manera: Se seleccionan los tres ángulos $\pi/6$, $3\pi/6$ y $5\pi/6$, y se calculan los tres vectores de la base $\cos(f\theta)$ para $f = 0, 1$, y 2, y para los tres ángulos. Los resultados se muestran en la Tabla 4.34 para el beneficio del lector.

Debido a la particular elección de los tres ángulos, estos vectores son ortogonales, pero no ortonormales. Sus magnitudes son: $\sqrt{3}$, $\sqrt{1{,}5}$, y $\sqrt{1{,}5}$, respectivamente. Normalizándolos se obtienen los tres vectores $\mathbf{v}_1 = (0{,}5774, 0{,}5774, 0{,}5774)$, $\mathbf{v}_2 = (0{,}7071, 0, -0{,}7071)$, y $\mathbf{v}_3 = (0{,}4082, -0{,}8165, 0{,}4082)$. Cuando se apilan verticalmente, producen la siguiente matriz de 3×3:

$$\mathbf{M} = \begin{bmatrix} 0{,}5774 & 0{,}5774 & 0{,}5774 \\ 0{,}7071 & 0 & -0{,}7071 \\ 0{,}4082 & -0{,}8165 & 0{,}4082 \end{bmatrix}. \tag{4.17}$$

[La Ecuación 4.13 (pág. 304) nos dice cómo normalizar estos vectores: Se multiplica cada uno por $\sqrt{2/3}$, y luego se multiplica el primero por $1/\sqrt{2}$.] Observe que, como resultado de la normalización, las columnas de $\mathbf{M}$ tienen que ser también ortonormales, por lo tanto, $\mathbf{M}$ es una matriz ortonormal (tales matrices tienen propiedades especiales).

Los pasos para el cálculo de la matriz DCT para un n arbitrario son:

1. Se seleccionan los n ángulos $\theta_j = (j + 0{,}5)\,\pi/n$ para $j = 0, \ldots n-1$. Dividimos el intervalo $[0, \pi]$ en n segmentos de igual tamaño; estos ángulos son los puntos centrales de los segmentos.

2. Se calculan los n vectores $\mathbf{v}_k$ para $k = 0, 1, 2, \ldots, n-1$, cada uno con los n componentes $\cos(k\theta_j)$.

3. Se normaliza cada uno de los n vectores y se disponen como las n filas de una matriz.

Los ángulos seleccionados para la DCT son $\theta_j = (j + 0{,}5)\,\pi/n$, por lo que las componentes de cada vector $\mathbf{v}_k$ son: $\cos[k\,(j + 0{,}5)\,\pi/n]$ o $\cos[k\,(2j + 1)\,\pi/(2n)]$. La Sección 4.6.4 cubre otras tres formas de seleccionar tales ángulos. Esta elección de ángulos tiene las dos siguientes propiedades útiles: (1) Los vectores resultantes son ortogonales, y (2) para valores crecientes de k, los n vectores $\mathbf{v}_k$ contienen frecuencias crecientes (Figura 4.35). Para $n = 3$, la fila superior de $\mathbf{M}$ [Ecuación (4.17)] corresponde a la frecuencia cero, la fila del medio (cuyos elementos se vuelven más pequeños monótonamente) representa la frecuencia baja, y la fila inferior (con tres elementos que primero van hacia abajo, y luego hacia arriba) representa la frecuencia alta. Dado un vector tridimensional $\mathbf{v} = (v_1, v_2, v_3)$, el producto $\mathbf{M} \cdot \mathbf{v}$ es un triplete cuyos componentes indican las magnitudes de las diferentes frecuencias incluidas en $\mathbf{v}$; éstas se denominan *coeficientes de la frecuencia*. [Estrictamente hablando, el producto es $\mathbf{M} \cdot \mathbf{v}^T$, pero ignoramos la transposición en los casos en que el significado es claro.] Los tres ejemplos extremos siguientes ilustran el significado de lo dicho.

El primer ejemplo es $\mathbf{v} = (v, v, v)$. Los tres componentes de $\mathbf{v}$ son idénticos, por lo que corresponden a la frecuencia cero. El producto $\mathbf{M} \cdot \mathbf{v}$ produce los coeficientes de frecuencia $(1{,}7322v, 0, 0)$, indicando que no hay frecuencias altas. El segundo ejemplo es $v = (v, 0, -v)$. Los tres componentes de $\mathbf{v}$ varían lentamente desde v a $-v$, por lo que este vector contiene una frecuencia baja. El producto $\mathbf{M} \cdot \mathbf{v}$ produce los coeficientes $(0,\ 1{,}4142v, 0)$, lo que confirma este resultado. El tercer ejemplo es $v = (v, -v, v)$. Los tres componentes de $\mathbf{v}$ varían desde $\mathbf{v}$ hasta $-\mathbf{v}$, por lo que este vector contiene una frecuencia alta. El producto $\mathbf{M} \cdot \mathbf{v}$ produce $(0, 0,\ 1{,}6329v)$, indicando de nuevo la frecuencia correcta.

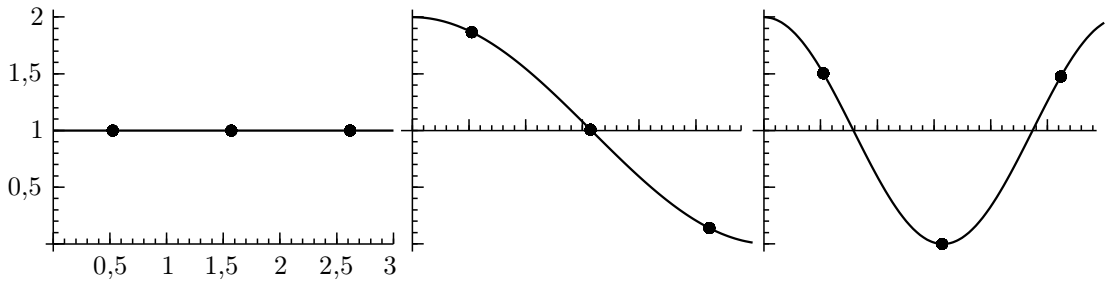


Figura 4.35: Frecuencias crecientes.

Estos ejemplos no son muy realistas, porque los vectores que se están probando son cortos, sencillos, y contienen una sola frecuencia cada uno. La mayor parte de los vectores son más complejos y contienen varias frecuencias, lo que hace este método útil. Un ejemplo sencillo de un vector con dos frecuencias es $\mathbf{v} = (1,\ 0{,}33\ -0{,}34)$. Con el producto $\mathbf{M}\cdot\mathbf{v}$ se obtiene $(0{,}572,\ 0{,}948, 0)$ lo que indica una frecuencia media grande, una frecuencia cero pequeña, y una frecuencia alta inexistente. Ésto tiene sentido cuando nos damos cuenta de que el vector que se está probando procede de la suma $0{,}33\,(1,1,1)+0{,}67\,(1,0,-1)$. Un ejemplo similar es la suma $0{,}9\,(-1,1,-1)+0{,}1\,(1,1,1) = (-0{,}8,\ 1,-0{,}8)$, que cuando se multiplica por $\mathbf{M}$ produce $(-0{,}346,\ 0,-1{,}469)$. Por otro lado, un vector con componentes aleatorios, tales como $(1, 0,\ 0{,}33)$, contiene típicamente cantidades aproximadamente iguales de las tres frecuencias y produce tres coeficientes de frecuencia grandes. El producto $\mathbf{M}\cdot(1, 0,\ 0{,}33)$ produce $(0{,}77,\ 0{,}47\,0{,}54)$, porque $(1, 0,\ 0{,}33)$ es la suma:

$$0{,}33\,(1,1,1) + 0{,}33\,(1,0,-1) + 0{,}33\,(1,-1,1)\,.$$

Observe que si $\mathbf{M}\cdot\mathbf{v} = \mathbf{c}$, entonces $\mathbf{M}^T\cdot\mathbf{c} = \mathbf{M}^{-1}\cdot\mathbf{c} = \mathbf{v}$. El vector original $\mathbf{v}$ puede, por lo tanto, ser reconstruido a partir de los coeficientes de frecuencia (con pequeñas diferencias debidas a la limitación en la precisión de las operaciones aritméticas de la máquina). La inversa $\mathbf{M}^{-1}$ de $\mathbf{M}$ es también su transpuesta $\mathbf{M}^T$, porque $\mathbf{M}$ es ortonormal.

Un vector tridimensional puede tener sólo tres frecuencias: cero, media y alta. De manera similar, un vector de n-dimensional puede tener n frecuencias diferentes, que este método puede identificar. Nos concentramos en el caso $n = 8$ y comenzamos con la DCT en una dimensión. La Figura 4.36 muestra ocho ondas coseno de la forma $\cos(f\theta_j)$, para $0 \le \theta_j \le \pi$, con frecuencias $f = 0, 1, \ldots, 7$. Cada onda es muestreada en los ocho puntos:

$$\theta_j = \frac{\pi}{16},\ \frac{3\pi}{16},\ \frac{5\pi}{16},\ \frac{7\pi}{16},\ \frac{9\pi}{16},\ \frac{11\pi}{16},\ \frac{13\pi}{16},\ \frac{15\pi}{16} \tag{4.18}$$

para formar una base de vectores $\mathbf{v}_f$, y el resultado de los ocho vectores $\mathbf{v}_f$, $f = 0, 1, \ldots, 7$(un total de 64 números) se muestran en la Tabla 4.37. Sirven como la matriz de base de la DCT. Nótese la similitud entre esta tabla y la de la matriz $\mathbf{W}$ de la Ecuación (4.5).

Debido a la elección particular de los ocho puntos de muestra, las $\mathbf{v}_i$'s son ortogonales. Ésto es fácil de comprobar directamente con el software matemático adecuado, pero la Sección 4.6.4 describe una forma más elegante para probar esta propiedad. Después de la normalización, las $\mathbf{v}_i$'s pueden considerarse como una matriz de transformación de 8×8 (específicamente, una matriz de rotación, ya que es ortonormal) o como un conjunto de ocho vectores ortogonales que constituyen la base de un espacio vectorial. Cualquier vector $\mathbf{p}$ en este espacio se puede expresar como una combinación lineal de las $\mathbf{v}_i$'s. Como ejemplo, seleccionamos los ocho números (correlacionados) $\mathbf{p} = (0{,}6,\ 0{,}5,\ 0{,}4,\ 0{,}5,\ 0{,}6,\ 0{,}5,\ 0{,}4,\ 0{,}55)$ como nuestros datos de prueba y expresamos $\mathbf{p}$ como una combinación lineal $\mathbf{p} = \sum\omega_i\mathbf{v}_i$ de los ocho

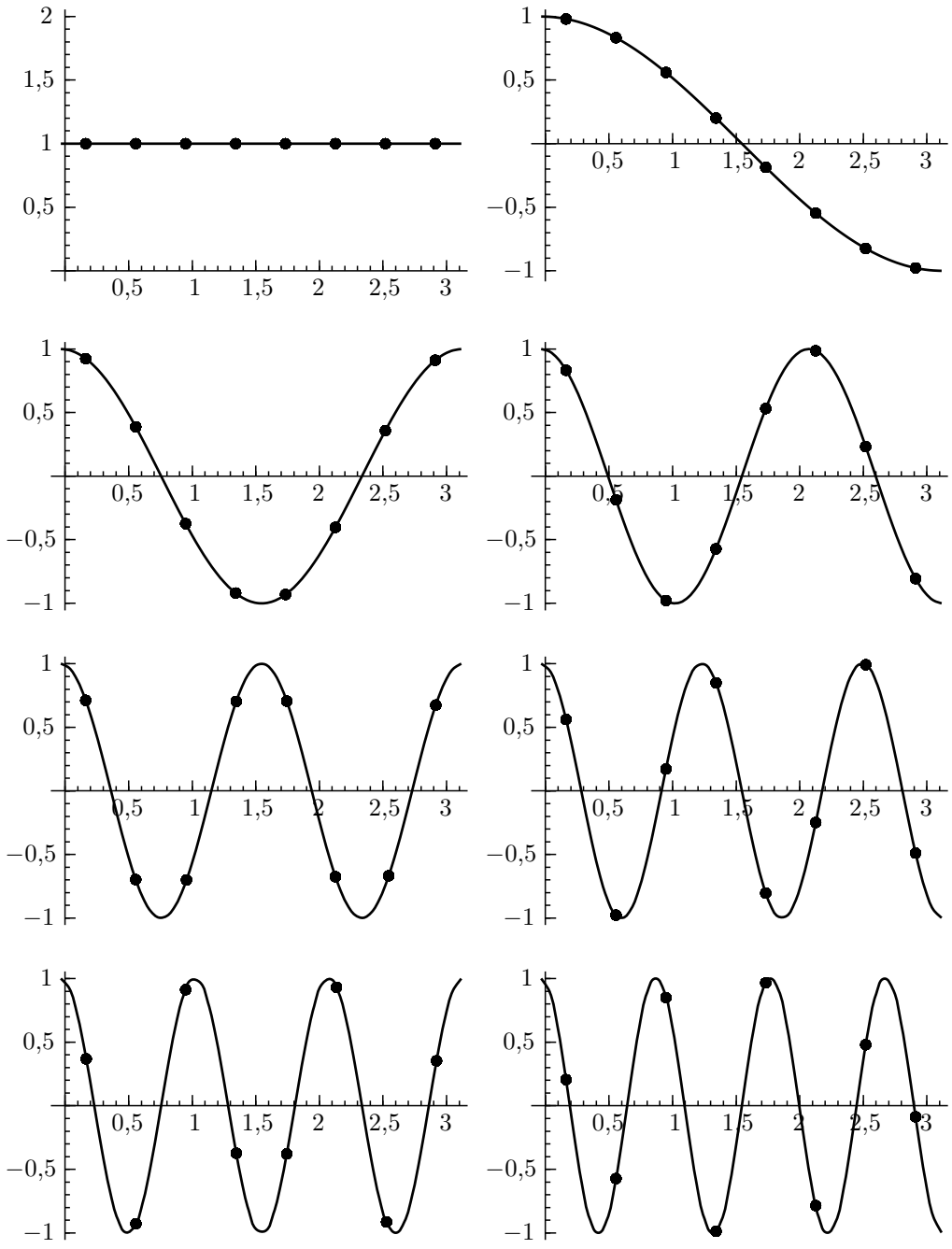


Figura 4.36: Valores de ángulo y coseno para una DCT de 8 puntos.

θ	0,196	0,589	0,982	1,374	1,767	2,160	2,553	2,945
$\cos 0\theta$	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0
$\cos 1\theta$	0,981	0,831	0,556	0,195	−0,195	−0,556	−0,831	−0,981
$\cos 2\theta$	0,924	0,383	−0,383	−0,924	−0,924	−0,383	0,383	0,924
$\cos 3\theta$	0,831	−0,195	−0,981	−0,556	0,556	0,981	0,195	−0,831
$\cos 4\theta$	0,707	−0,707	−0,707	0,707	0,707	−0,707	−0,707	0,707
$\cos 5\theta$	0,556	−0,981	0,195	0,831	−0,831	−0,195	0,981	−0,556
$\cos 6\theta$	0,383	−0,924	0,924	−0,383	−0,383	0,924	−0,924	0,383
$\cos 7\theta$	0,195	−0,556	0,831	−0,981	0,981	−0,831	0,556	−0,195

Tabla 4.37: La matriz DCT sin normalizar en una dimensión para $n = 8$.

```
Table[N[t],{t,Pi/16,15Pi/16,Pi/8}]
dctp[pw_]:=Table[N[Cos[pw t]],{t,Pi/16,15Pi/16,Pi/8}]
dctp[0]
dctp[1]
...
dctp[7]
```

Código para la Tabla 4.37.

```
dct[pw_]:=Plot[Cos[pw t], {t,0,Pi}, DisplayFunction->Identity,
AspectRatio->Automatic];
dcdot[pw_]:=ListPlot[Table[{t,Cos[pw t]},{t,Pi/16,15Pi/16,Pi/8}],
DisplayFunction->Identity]
Show[dct[0],dcdot[0], Prolog->AbsolutePointSize[4],
DisplayFunction->$DisplayFunction]
...
Show[dct[7],dcdot[7], Prolog->AbsolutePointSize[4],
DisplayFunction->$DisplayFunction]
```

Código para la Figura 4.36.

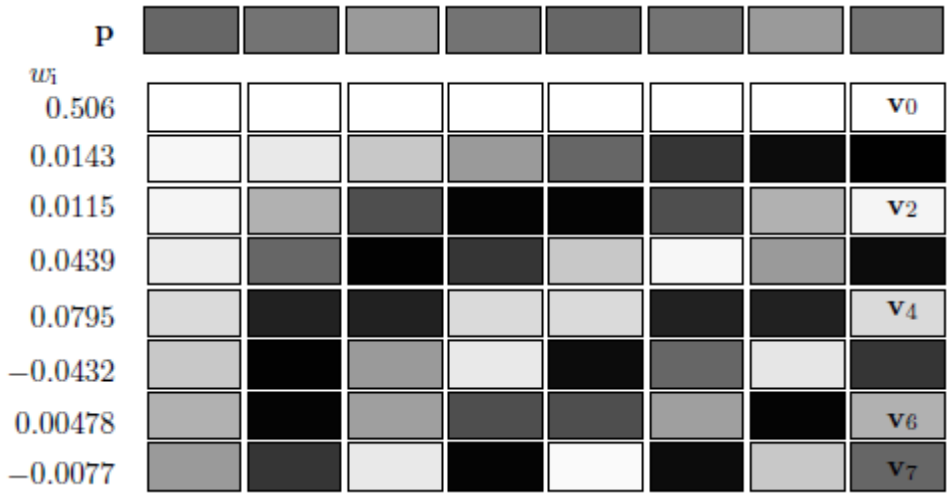


Figura 4.38: Una representación gráfica de la DCT unidimensional.

vectores de la base. La solución de este sistema de ocho ecuaciones produce los ocho pesos:

$$\omega_0 = 0{,}506, \quad \omega_1 = 0{,}0143, \quad \omega_2 = 0{,}0115, \quad \omega_3 = 0{,}0439,$$
$$\omega_4 = 0{,}0795, \quad \omega_5 = -0{,}0432, \quad \omega_6 = 0{,}00478, \quad \omega_7 = -0{,}0077.$$

El peso ω_0 no es muy diferente de los elementos de $\mathbf{p}$, pero los otros siete pesos son mucho más pequeños. Así es cómo la DCT (o cualquier otra transformación ortogonal) puede conducir a la compresión. Los ocho pesos pueden ser cuantificados y escritos en la secuencia de datos comprimidos, donde ocupan menos espacio que los ocho componentes de $\mathbf{p}$.

La Figura 4.38 ilustra esta combinación lineal gráficamente. Cada uno de los ocho $\mathbf{v}_i$'s se muestra como una fila de ocho pequeños rectángulos grises (una imagen base), donde un valor de $+1$ está dibujado en blanco, y uno de -1 es de color negro. Los ocho elementos del vector $\mathbf{p}$ también se muestran como una hilera de ocho píxeles en escala de grises.

En resumen, podemos interpretar la DCT unidimensional como un conjunto de imágenes base que tienen las frecuencias más altas. Dado un vector de datos, la DCT separa las frecuencias contenidas en los datos y representa el vector como una combinación lineal (o una suma ponderada) de las imágenes base. Los pesos son los coeficientes de la DCT. Esta interpretación puede extenderse a la DCT bi dimensional. Aplicamos la Ecuación (4.15) para el caso $n = 8$, para crear 64 pequeñas imágenes base de 8×8 píxeles cada una. Las 64 imágenes se utilizan entonces como una base de un espacio vectorial 64-dimensional. Cualquier imagen B de 8×8 píxeles puede expresarse como una combinación lineal de las imágenes de la base, y los 64 pesos de esta combinación lineal son los coeficientes de la DCT de B.

La Figura 4.39 muestra la representación gráfica de las 64 imágenes base de la DCT bidimensional para $n = 8$. Un elemento general $(i.j)$ de esta figura es la imagen de 8×8 obtenida mediante el cálculo del producto $\cos(i \cdot s)\cos(j \cdot t)$, donde s y t varían de forma independiente de los valores indicados en la Ecuación (4.18) e i y j varían desde 0 hasta 7. Esta figura se puede generar fácilmente usando el código en Mathematica mostrado con ella. El código alternativo que se muestra, es una modificación del código disponible en [Watson 94], y requiere el paquete `GraphicsImage.m`, que no está muy disponible.

Con el uso del software adecuado, es fácil realizar cálculos de la DCT y mostrar los resultados de forma gráfica. La Figura 4.40a muestra una unidad de datos al azar de 8×8 que consta de ceros

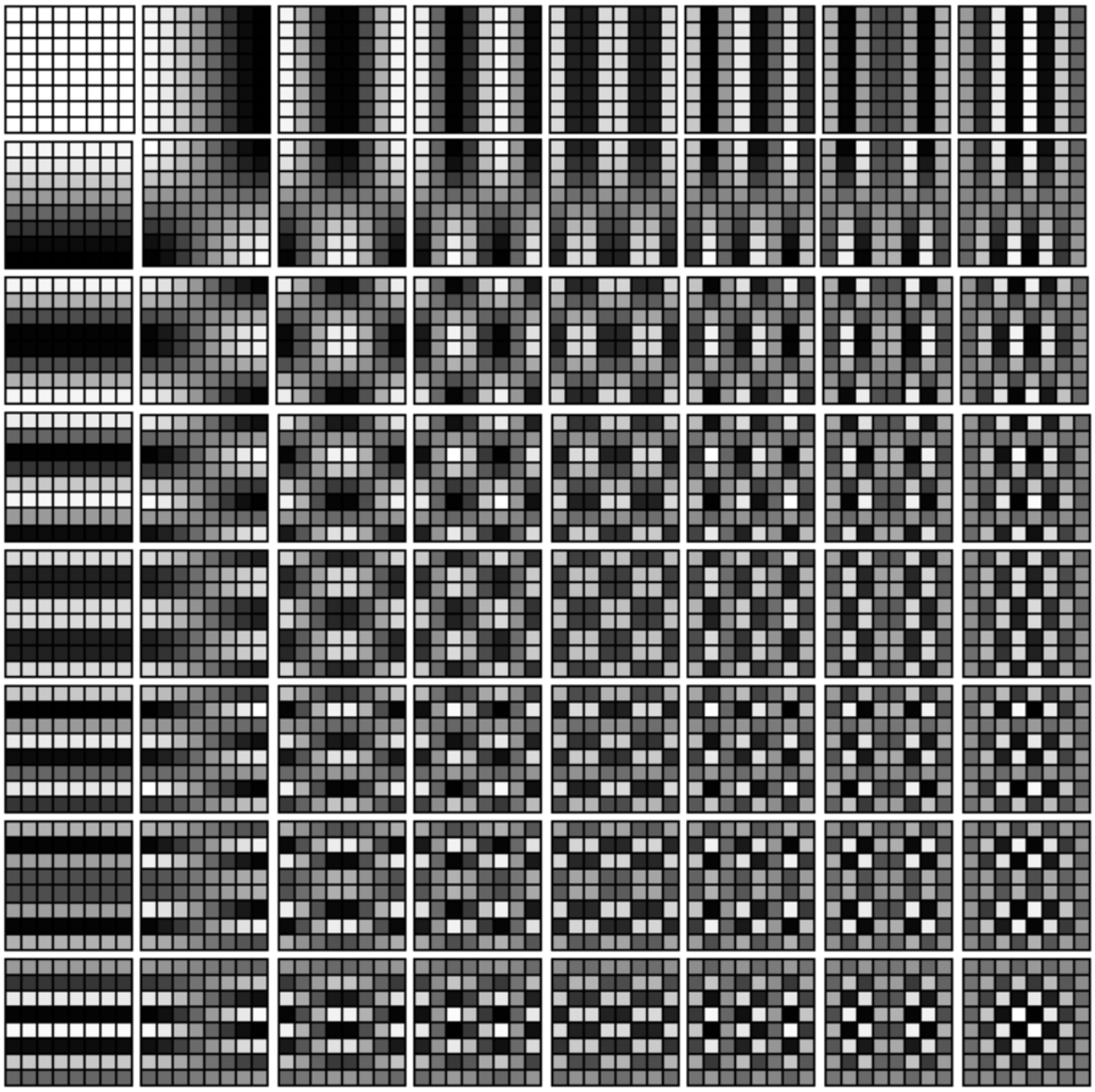


Figura 4.39: Las 64 imágenes base de la DCT bidimensional.

```
dctp[fs_,ft_]:=Table[SetAccuracy[N[(1.-Cos[fs s]Cos[ft t])/2],3],
{s,Pi/16,15Pi/16,Pi/8},{t,Pi/16,15Pi/16,Pi/8}]//TableForm
dctp[0,0]
dctp[0,1]
...
dctp[7,7]
```

Código para la Figura 4.39.

```
Needs["GraphicsImage‘"] (* Dibuja coeficientes DCT 2D *)
DCTMatrix=Table[If[k==0,Sqrt[1/8],Sqrt[1/4]Cos[Pi(2j+1)k/16]],
{k,0,7}, {j,0,7}] //N;
DCTTensor=Array[Outer[Times,DCTMatrix[[#1]],DCTMatrix[[#2]]]&,{8,8}];
Show[GraphicsArray[Map[GraphicsImage[#,{-.25,.25}]&,DCTTensor,{2}]]]
```

Código alternativo para la Figura 4.39.

y unos. La misma unidad se muestra gráficamente en la Figura 4.40b, con 1 como blanco y 0 como negro. La Figura 4.40c muestra los pesos por los que cada una de las imágenes de base 64 DCT tiene que ser multiplicada con el fin de reproducir la unidad de datos original. En esta figura, se muestra el cero en gris neutro, los números positivos con brillo (nótese cuán brillante es el peso DC), y los números negativos se muestran en oscuro. La Figura 4.40d muestra los pesos numéricamente. También muestra el código en *Mathematica* que hace todo ésto. La Figura 4.41 es similar, pero para una unidad de datos regular.

◇ **Ejercicio 4.13 (sol. en pág. 1081):** Imagine un bloque de valores de 8×8 donde todas las filas impares están formadas por 1's, y todas las filas pares contienen ceros. ¿Qué podemos decir acerca de los pesos de la DCT de este bloque?

4.6.3. La DCT como una rotación

La segunda interpretación de la matriz **M** [Ecuación (4.17)] es como una rotación. Ya sabemos que $\mathbf{M} \cdot (v, v, v)$ tiene como resultado (1,7322v, 0, 0) y ésto se puede interpretar como una rotación desde el punto (v, v, v) al punto (1,7322v, 0, 0). El primer punto se encuentra en la línea que forma ángulos iguales con los tres ejes de coordenadas; y el último, está en el eje x. Cuando los consideramos como píxeles adyacentes, esta rotación tiene un significado sencillo. Imagine tres píxeles adyacentes en una imagen. Normalmente son similares, por tanto, empezamos examinando el caso en que son idénticos. Cuando tres píxeles idénticos se consideran las coordenadas de un punto en tres dimensiones, ese punto está situado en la línea $x = y = z$. La rotación de esta línea hacia el eje x acerca nuestro punto a dicho eje; su coordenada x no ha cambiado mucho, y sus coordenadas y y z son cero. Así es cómo tal rotación conduce a la compresión. En general, tres pixeles adyacentes p_1, p_2, y p_3 son similares pero no idénticos, lo que ubica el punto (p_1, p_2, p_3) algo fuera de la línea $x = y = z$. Después de la rotación, el punto terminará cerca del eje x, donde sus coordenadas x e y serán cantidades pequeñas.

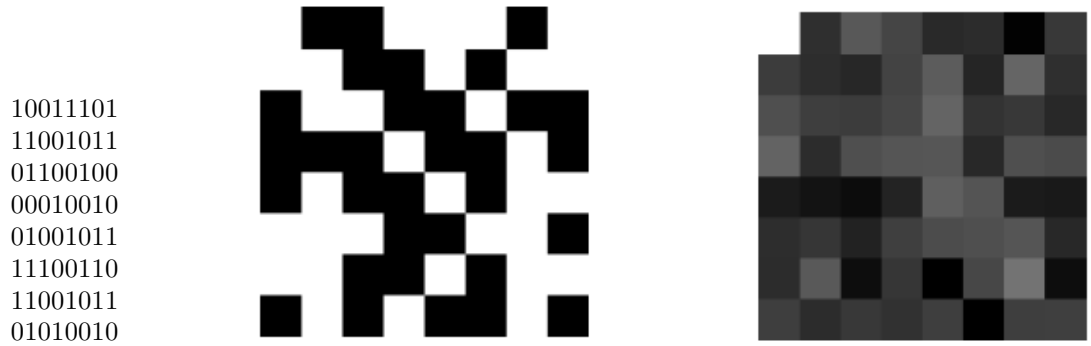
Esta interpretación de **M** como una rotación tiene sentido porque **M** es ortonormal, y cualquier matriz ortonormal es una matriz de rotación. Sin embargo, el determinante de una matriz de rotación es 1, mientras que el determinante de nuestra matriz es -1. Una matriz ortonormal cuyo determinante es -1 realiza una *rotación impropia* (una rotación combinada con una reflexión). Para obtener una mejor comprensión de la transformación realizada por **M**, aplicamos la técnica de descomposición QR de una matriz (Sección 4.6.8) para descomponer la matriz **M** en el producto $T1 \times T2 \times T3 \times T4$, donde:

$$T1 = \begin{bmatrix} 0{,}8165 & 0 & -0{,}5774 \\ 0 & 1 & 0 \\ 0{,}5774 & 0 & 0{,}8165 \end{bmatrix}, \quad T2 = \begin{bmatrix} 0{,}7071 & -0{,}7071 & 0 \\ 0{,}7071 & 0{,}7071 & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

$$T3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix}, \quad T4 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}.$$

Cada una de las matrices $T1$, $T2$ y $T3$ realiza una rotación (llamada *rotación de Givens*) alrededor de uno de los ejes de coordenadas. La matriz $T4$ es una reflexión sobre el eje z. La transformación $\mathbf{M} \cdot (1, 1, 1,)$ ahora puede escribirse como $T1 \times T2 \times T3 \times T4 \times (1, 1, 1)$, donde $T4$ refleja el punto $(1, 1, 1)$ en el $(1, 1, -1)$, $T3$ rota $(1, 1, -1)$ 90º alrededor del eje x hasta el $(1, -1, -1)$, que es rotado 45º por $T2$ alrededor del eje z hasta el (1,4142, 0, -1), que es rotado 35,26º por $T1$ alrededor del eje y hasta el $(1{,}7321, 0, 0)$.

[Esta secuencia particular de transformaciones es un resultado del orden en el que se han realizado los pasos individuales de la descomposición QR. Efectuando los mismos pasos en un orden diferente se producen diferentes secuencias de rotaciones. Un ejemplo es: (1) una reflexión respecto al eje z que



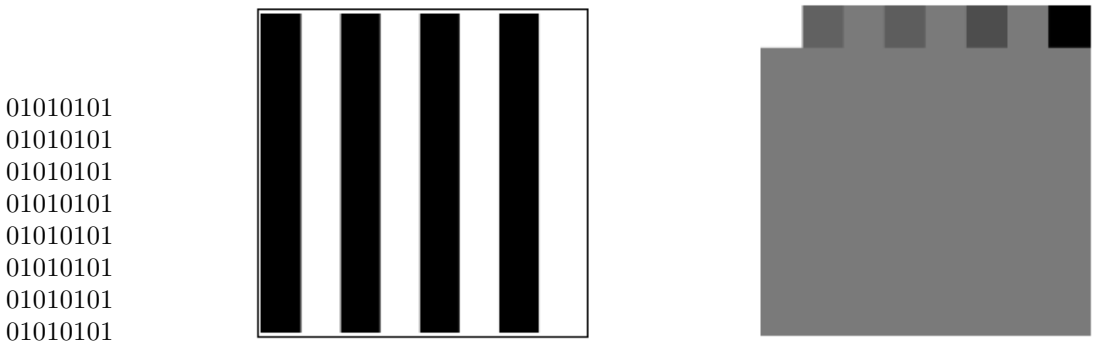
(a) (b) (c)

4,000	−0,133	0,637	0,272	−0,250	−0,181	−1,076	0,026
0,081	−0,178	−0,300	0,230	0,694	−0,309	0,875	−0,127
0,462	0,125	0,095	0,291	0,868	−0,070	0,021	−0,280
0,837	−0,194	0,455	0,583	0,588	−0,281	0,448	0,383
−0,500	−0,635	−0,749	−0,346	0,750	0,557	−0,502	−0,540
−0,167	0	−0,366	0,146	0,393	0,448	0,577	−0,268
−0,191	0,648	−0,729	−0,008	−1,171	0,306	1,155	−0,744
0,122	−0,200	0,038	−0,118	0,138	−1,154	0,134	0,148

Figura 4.40: Un ejemplo de la DCT bidimensional.

```
DCTMatrix=Table[If[k==0,Sqrt[1/8],Sqrt[1/4]Cos[Pi(2j+1)k/16]],
{k,0,7}, {j,0,7}] //N;
DCTTensor=Array[Outer[Times, DCTMatrix[[#1]],DCTMatrix[[#2]]]&,
{8,8}];
img={{1,0,0,1,1,1,0,1},{1,1,0,0,1,0,1,1},
{0,1,1,0,0,1,0,0},{0,0,0,1,0,0,1,0},
{0,1,0,0,1,0,1,1},{1,1,1,0,0,1,1,0},
{1,1,0,0,1,0,1,1},{0,1,0,1,0,0,1,0}};
ShowImage[Reverse[img]]
dctcoeff=Array[(Plus @@ Flatten[DCTTensor[[#1,#2]] img])&,{8,8}];
dctcoeff=SetAccuracy[dctcoeff,4];
dctcoeff=Chop[dctcoeff,.001];
MatrixForm[dctcoeff]
ShowImage[Reverse[dctcoeff]]
```

Código para la Figura 4.40



(a) (b) (c)

4,000	−0,721	0	−0,850	0	−1,273	0	−3,625
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

(d)

> Algunos pintores transforman el sol en una mancha amarilla; otros transforman una mancha amarilla en el sol.
>
> —Pablo Picasso.

```
DCTMatrix=Table[If[k==0,Sqrt[1/8],Sqrt[1/4]Cos[Pi(2j+1)k/16]],
{k,0,7}, {j,0,7}] //N;
DCTTensor=Array[Outer[Times, DCTMatrix[[#1]],DCTMatrix[[#2]]]&,
{8,8}];
img={{0,1,0,1,0,1,0,1},{0,1,0,1,0,1,0,1},
{0,1,0,1,0,1,0,1},{0,1,0,1,0,1,0,1},{0,1,0,1,0,1,0,1},
{0,1,0,1,0,1,0,1},{0,1,0,1,0,1,0,1},{0,1,0,1,0,1,0,1}};
ShowImage[Reverse[img]]
dctcoeff=Array[(Plus @@ Flatten[DCTTensor[[#1,#2]] img])&,{8,8}];
dctcoeff=SetAccuracy[dctcoeff,4];
dctcoeff=Chop[dctcoeff,.001];
MatrixForm[dctcoeff]
ShowImage[Reverse[dctcoeff]]
```

Código para la Figura 4.41

Figura 4.41: Un ejemplo de la DCT bidimensional.

```
L L L L L L L L      L S S S S S S S      L S S S S S S S
L L L L L L L L      L S S S S S S S      S s s s s s s s
L L L L L L L L      L S S S S S S S      S s s s s s s s
L L L L L L L L      L S S S S S S S      S s s s s s s s
L L L L L L L L      L S S S S S S S      S s s s s s s s
L L L L L L L L      L S S S S S S S      S s s s s s s s
L L L L L L L L      L S S S S S S S      S s s s s s s s
L L L L L L L L      L S S S S S S S      S s s s s s s s
```

(a) (b) (c)

Figura 4.42: La DCT bidimensional como una rotación doble.

transforma $(1, 1, 1)$ en el $(1, 1, -1)$, (2) una rotación de $(1, 1, -1)$ de 135° alrededor del eje x hasta el $(1, -1{,}4141, 0)$, y (3) una rotación adicional de 54,74° alrededor del eje z hasta el (1,7321, 0, 0).]

Para un n arbitrario, esta interpretación es similar. Empezamos con un vector de n píxeles adyacentes. Éstos se consideran las coordenadas de un punto en espacio n-dimensional. Si los píxeles son similares, el punto se encuentra cerca de la línea que forma ángulos iguales con todos los ejes de coordenadas. La aplicación de la DCT unidimensional [Ecuación (4.13)] rota el punto y lo aproxima al eje x, de manera que su primera coordenada no ha cambiado mucho y sus restantes $n-1$ coordenadas son números pequeños. Así es como la DCT unidimensional puede considerarse una rotación única en el espacio n-dimensional. La rotación se puede dividir en una reflexión seguida de $n-1$ rotaciones de Givens, pero un usuario de la DCT no necesita preocuparse por estos detalles.

La DCT bidimensional se interpreta de forma similar como una rotación doble. Esta interpretación comienza con un bloque de $n \times n$ píxeles (Figura 4.42a, donde los píxeles están etiquetados como `L`). En primer lugar, se considera cada fila de este bloque como un punto $(p_{x,0}, p_{x,1}, \ldots, p_{x,n-1})$ en el espacio n-dimensional, y hace rotar el punto mediante la suma más interna:

$$G1_{x,j} = \sqrt{\frac{2}{n}} C_j \sum_{y=0}^{n-1} p_{xy} \cos\left(\frac{(2y+1)\, j\pi}{2n}\right)$$

de la Ecuación (4.15). Ésto produce un bloque $G1_{x,j}$ de $n \times n$ coeficientes, donde el primer elemento de cada fila es dominante (etiquetado `L` en la Figura 4.42b) y los elementos restantes son pequeños (etiquetados `S` en esa figura). La suma más externa de la Ecuación (4.15) es:

$$G_{ij} = \sqrt{\frac{2}{n}} C_i \sum_{x=0}^{n-1} G1_{x,j} \cos\left(\frac{(2x+1)\, i\pi}{2n}\right).$$

Aquí, las *columnas* de $G1_{x,j}$ se consideran puntos en espacio n-dimensional y son rotadas. El resultado es un coeficiente grande en la esquina superior izquierda del bloque (`L` en la Figura 4.42c) y $n^2 - 1$ coeficientes pequeños en los demás lugares (`S` y `s` en esa figura). Esta interpretación considera la DCT bidimensional como dos rotaciones separadas en n dimensiones; la primera de ellas rota cada una de las n filas, y la segunda rota cada una de las n columnas. Es interesante observar que $2n$ rotaciones en n dimensiones son más rápidas que una rotación en n^2 dimensiones, ya que la última requiere una matriz de rotación de $n^2 \times n^2$.

4.6.4. Los cuatro tipos de DCT

Hay cuatro formas de seleccionar n ángulos equidistantes que generan vectores ortogonales de cosenos. Corresponden (después de normalizar los vectores por factores de escala) a cuatro transformadas

del coseno discretas, designadas DCT-1 a DCT-4. La más útil es la DCT-2, que normalmente se conoce como *la* DCT. La Ecuación (4.19) muestra las definiciones de los cuatro tipos. En la Tabla 4.43, se muestran (para $n = 8$) los ángulos reales para la DCT-1 y la DCT-2. Dése cuenta de que la DCT-3 es la transpuesta de la DCT-2, y la 4-DCT es una versión desplazada de la DCT-1. Observe que la DCT-1 tiene $n + 1$ vectores de $n + 1$ cosenos cada uno. En cada uno de los cuatro tipos, los n (o $n + 1$) vectores de la DCT son ortogonales y quedan normalizados después de multiplicarlos por el factor de escala adecuado. La Figura 4.44 muestra el código en *Mathematica* para generar los vectores normalizados de los cuatro tipos y un test para la normalización.

$$\begin{aligned}
\mathrm{DCT}^1_{k,j} &= \sqrt{\frac{2}{n}} C_k C_j \cos\left[\frac{kj\pi}{n}\right], \quad k,\, j = 0, 1, \ldots, n, \\
\mathrm{DCT}^2_{k,j} &= \sqrt{\frac{2}{n}} C_k \cos\left[\frac{k\left(j+\frac{1}{2}\right)\pi}{n}\right], \quad k,\, j = 0, 1, \ldots, n-1, \\
\mathrm{DCT}^3_{k,j} &= \sqrt{\frac{2}{n}} C_j \cos\left[\frac{\left(k+\frac{1}{2}\right) j\pi}{n}\right], \quad k,\, j = 0, 1, \ldots, n-1, \\
\mathrm{DCT}^4_{k,j} &= \sqrt{\frac{2}{n}} \cos\left[\frac{\left(k+\frac{1}{2}\right)\left(j+\frac{1}{2}\right)\pi}{n}\right], \quad k,\, j = 0, 1, \ldots, n-1,
\end{aligned} \tag{4.19}$$

donde le factor de escala C_x se define como

$$C_x = \begin{cases} 1/\sqrt{2}, & \text{si } x = 0 \text{ ó } x = n, \\ 1, & \text{en otro caso.} \end{cases}$$

La ortogonalidad puede ser demostrada, ya sea directamente —multiplicando pares de vectores diferentes—, ya sea indirectamente. Este último enfoque se discute en detalle en [Strang 99] y allí demuestra que los vectores DCT son ortogonales, mostrando que son los vectores propios de ciertas matrices simétricas. En el caso de la DCT-2, por ejemplo, la matriz simétrica es

$$\mathbf{A} = \begin{bmatrix} 1 & -1 & & & \\ -1 & 2 & -1 & & \\ & & \vdots & & \\ & & -1 & 2 & -1 \\ & & & -1 & 1 \end{bmatrix}. \quad \text{Para } n = 3, \text{ la matriz } \quad \mathbf{A}_3 = \begin{bmatrix} 1 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{bmatrix}$$

tiene como vectores propios (0,5774, 0,5774, 0,5774), (0,7071, 0, −0,7071), (0,4082, −0,8165, 0,4082), con valores propios 0, 1 y 3, respectivamente. Recordemos que estos vectores propios son las filas de la matriz **M** de la Ecuación (4.17).

4.6.5. La DCT en la práctica

La Ecuación (4.15) puede ser codificada directamente en cualquier lenguaje de alto nivel. Ya que esta ecuación es la base de varios métodos de compresión como JPEG y MPEG, su cálculo rápido es esencial. Se puede acelerar considerablemente haciendo varias mejoras, y esta sección ofrece algunas ideas.

1. Sin importar el tamaño de la imagen, sólo están involucradas 32 funciones coseno. Éstas pueden calcularse previamente una vez y utilizarse cuando sean necesarias para calcular todas las 8×8 unidades de datos. El cálculo de la expresión:

$$p_{xy} \cos\left[\frac{(2x+1)\, i\pi}{16}\right] \cos\left[\frac{(2y+1)\, j\pi}{16}\right]$$

k	escala	Ángulos para la DCT-1 (9×9)								
0	$\frac{1}{2\sqrt{2}}$	0^*	0	0	0	0	0	0	0	0^*
1	$\frac{1}{2}$	0	$\frac{\pi}{8}$	$\frac{2\pi}{8}$	$\frac{3\pi}{8}$	$\frac{4\pi}{8}$	$\frac{5\pi}{8}$	$\frac{6\pi}{8}$	$\frac{7\pi}{8}$	$\frac{8\pi}{8}$
2	$\frac{1}{2}$	0	$\frac{2\pi}{8}$	$\frac{4\pi}{8}$	$\frac{6\pi}{8}$	$\frac{8\pi}{8}$	$\frac{10\pi}{8}$	$\frac{12\pi}{8}$	$\frac{14\pi}{8}$	$\frac{16\pi}{8}$
3	$\frac{1}{2}$	0	$\frac{3\pi}{8}$	$\frac{6\pi}{8}$	$\frac{9\pi}{8}$	$\frac{12\pi}{8}$	$\frac{15\pi}{8}$	$\frac{18\pi}{8}$	$\frac{21\pi}{8}$	$\frac{24\pi}{8}$
4	$\frac{1}{2}$	0	$\frac{4\pi}{8}$	$\frac{8\pi}{8}$	$\frac{12\pi}{8}$	$\frac{16\pi}{8}$	$\frac{20\pi}{8}$	$\frac{24\pi}{8}$	$\frac{28\pi}{8}$	$\frac{32\pi}{8}$
5	$\frac{1}{2}$	0	$\frac{5\pi}{8}$	$\frac{10\pi}{8}$	$\frac{15\pi}{8}$	$\frac{20\pi}{8}$	$\frac{25\pi}{8}$	$\frac{30\pi}{8}$	$\frac{35\pi}{8}$	$\frac{40\pi}{8}$
6	$\frac{1}{2}$	0	$\frac{6\pi}{8}$	$\frac{12\pi}{8}$	$\frac{18\pi}{8}$	$\frac{24\pi}{8}$	$\frac{30\pi}{8}$	$\frac{36\pi}{8}$	$\frac{42\pi}{8}$	$\frac{48\pi}{8}$
7	$\frac{1}{2}$	0	$\frac{7\pi}{8}$	$\frac{14\pi}{8}$	$\frac{21\pi}{8}$	$\frac{28\pi}{8}$	$\frac{35\pi}{8}$	$\frac{42\pi}{8}$	$\frac{49\pi}{8}$	$\frac{56\pi}{8}$
8	$\frac{1}{2\sqrt{2}}$	0^*	$\frac{8\pi}{8}$	$\frac{16\pi}{8}$	$\frac{24\pi}{8}$	$\frac{32\pi}{8}$	$\frac{40\pi}{8}$	$\frac{48\pi}{8}$	$\frac{56\pi}{8}$	$\frac{64\pi}{8}^*$

*El factor de escala para estos ángulos es 4.

k	escala	Ángulos para la DCT-2							
0	$\frac{1}{2\sqrt{2}}$	0	0	0	0	0	0	0	0
1	$\frac{1}{2}$	$\frac{\pi}{16}$	$\frac{3\pi}{16}$	$\frac{5\pi}{16}$	$\frac{7\pi}{16}$	$\frac{9\pi}{16}$	$\frac{11\pi}{16}$	$\frac{13\pi}{16}$	$\frac{15\pi}{16}$
2	$\frac{1}{2}$	$\frac{2\pi}{16}$	$\frac{6\pi}{16}$	$\frac{10\pi}{16}$	$\frac{14\pi}{16}$	$\frac{18\pi}{16}$	$\frac{22\pi}{16}$	$\frac{26\pi}{16}$	$\frac{30\pi}{16}$
3	$\frac{1}{2}$	$\frac{3\pi}{16}$	$\frac{9\pi}{16}$	$\frac{15\pi}{16}$	$\frac{21\pi}{16}$	$\frac{27\pi}{16}$	$\frac{33\pi}{16}$	$\frac{39\pi}{16}$	$\frac{45\pi}{16}$
4	$\frac{1}{2}$	$\frac{4\pi}{16}$	$\frac{12\pi}{16}$	$\frac{20\pi}{16}$	$\frac{28\pi}{16}$	$\frac{36\pi}{16}$	$\frac{44\pi}{16}$	$\frac{52\pi}{16}$	$\frac{60\pi}{16}$
5	$\frac{1}{2}$	$\frac{5\pi}{16}$	$\frac{15\pi}{16}$	$\frac{25\pi}{16}$	$\frac{35\pi}{16}$	$\frac{45\pi}{16}$	$\frac{55\pi}{16}$	$\frac{65\pi}{16}$	$\frac{75\pi}{16}$
6	$\frac{1}{2}$	$\frac{6\pi}{16}$	$\frac{18\pi}{16}$	$\frac{30\pi}{16}$	$\frac{42\pi}{16}$	$\frac{54\pi}{16}$	$\frac{66\pi}{16}$	$\frac{78\pi}{16}$	$\frac{90\pi}{16}$
7	$\frac{1}{2}$	$\frac{7\pi}{16}$	$\frac{21\pi}{16}$	$\frac{35\pi}{16}$	$\frac{49\pi}{16}$	$\frac{63\pi}{16}$	$\frac{77\pi}{16}$	$\frac{91\pi}{16}$	$\frac{105\pi}{16}$

Tabla 4.43: Valores de ángulos para la DCT-1 y la DCT-2.

```
(* DCT-1. Observe: (n+1)x(n+1) *)
Clear[n, nor, kj, DCT1, T1];
n=8; nor=Sqrt[2/n];
kj[i_]:=If[i==0 || i==n, 1/Sqrt[2], 1];
DCT1[k_]:=Table[nor kj[j] kj[k] Cos[j k Pi/n], {j,0,n}]
T1=Table[DCT1[k], {k,0,n}]; (* Calcula nxn cosenos *)
MatrixForm[T1] (* los muestra como una matriz *)
(* multiplica las filas para mostrar la ortonormalidad *)
MatrixForm[Table[Chop[N[T1[[i]].T1[[j]]]], {i,1,n}, {j,1,n}]]

(* DCT-2 *)
Clear[n, nor, kj, DCT2, T2];
n=8; nor=Sqrt[2/n];
kj[i_]:=If[i==0 || i==n, 1/Sqrt[2], 1];
DCT2[k_]:=Table[nor kj[k] Cos[(j+1/2)k Pi/n], {j,0,n-1}]
T2=Table[DCT2[k], {k,0,n-1}]; (* Calcula nxn cosenos *)
MatrixForm[T2] (* los muestra como una matriz *)
(* multiplica las filas para mostrar la ortonormalidad *)
MatrixForm[Table[Chop[N[T2[[i]].T2[[j]]]], {i,1,n}, {j,1,n}]]

(* DCT-3. Ésta es la transpuesta de la DCT-2 *)
Clear[n, nor, kj, DCT3, T3];
n=8; nor=Sqrt[2/n];
kj[i_]:=If[i==0 || i==n, 1/Sqrt[2], 1];
DCT3[k_]:=Table[nor kj[j] Cos[(k+1/2)j Pi/n], {j,0,n-1}]
T3=Table[DCT3[k], {k,0,n-1}]; (* Calcula nxn cosenos *)
MatrixForm[T3] (* los muestra como una matriz *)
(* multiplica las filas para mostrar la ortonormalidad *)
MatrixForm[Table[Chop[N[T3[[i]].T3[[j]]]], {i,1,n}, {j,1,n}]]

(* DCT-4. Ésta es la DCT-1 desplazada *)
Clear[n, nor, DCT4, T4];
n=8; nor=Sqrt[2/n];
DCT4[k_]:=Table[nor Cos[(k+1/2)(j+1/2) Pi/n], {j,0,n-1}]
T4=Table[DCT4[k], {k,0,n-1}]; (* Calcula nxn cosenos *)
MatrixForm[T4] (* los muestra como una matriz *)
(* multiplica las filas para mostrar la ortonormalidad *)
MatrixForm[Table[Chop[N[T4[[i]].T4[[j]]]], {i,1,n}, {j,1,n}]]
```

Figura 4.44: Código para cuatro tipos de DCT.

Procedente del diccionario

Exegete (EK-suh-jeet), noun: A person who explains or interprets difficult parts of written works.[a]

[a]Exegeta o exégeta: Persona que expone o interpreta las partes difíciles de las obras escritas.

tiene como coste la realización de dos multiplicaciones. Así, la suma doble de la Ecuación (4.15) requiere $64 \times 2 = 128$ multiplicaciones y 63 adiciones.

◇ **Ejercicio 4.14 (sol. en pág. 1081):** (Propuesto por V. Saravanan.) ¿Por qué son sólo 32 las diferentes funciones coseno necesarias para la DCT?

2. Unos pocos retoques algebraicos muestran que la suma doble de la Ecuación (4.15) puede ser escrita como la matriz producto $\mathbf{CPC}^T$, donde $\mathbf{P}$ es la matriz de 8×8 de los píxeles, $\mathbf{C}$ es la matriz definida por:

$$C_{ij} = \begin{cases} \frac{1}{\sqrt{8}}, & i = 0 \\ \frac{1}{2}\cos\left[\frac{(2j+1)i\pi}{16}\right], & i > 0, \end{cases} \tag{4.20}$$

y $\mathbf{C}^T$ es la transpuesta de $\mathbf{C}$. (El producto de dos matrices $\mathbf{A}_{mp}$ y $\mathbf{B}_{pn}$ es una matriz $\mathbf{C}_{mn}$ definida por:

$$C_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}.$$

Para otras propiedades de las matrices, consulte cualquier texto sobre álgebra lineal.)

El cálculo de un elemento de la matriz producto $\mathbf{CP}$, por lo tanto, requiere ocho multiplicaciones y siete (pero para simplificar, digamos ocho) adiciones. La multiplicación de las dos matrices $\mathbf{C}$ y $\mathbf{P}$ de 8×8 requiere $64 \times 8 = 8^3$ multiplicaciones y el mismo número de adiciones. La multiplicación del producto $\mathbf{CP}$ por $\mathbf{C}^T$ requiere el mismo número de operaciones, por lo que la DCT de una unidad de datos de 8×8 requiere 2×8^3 multiplicaciones (y el mismo número de adiciones). Suponiendo que la imagen completa se compone de $n \times n$ píxeles y que $n = 8q$, hay $q \times q$ unidades de datos, por lo que la DCT de todas las unidades de datos requieren $2q^2 8^3$ multiplicaciones (y el mismo número de adiciones). En comparación, la realización de una DCT para toda la imagen requeriría $2n^3 = 2q^3 8^3 = \left(2q^2 8^3\right) q$ operaciones. Dividiendo la imagen en unidades de datos, se reduce el número de multiplicaciones (y también de adiciones) por un factor de q. Desafortunadamente, q no puede ser demasiado grande, porque eso significaría que las unidades de datos son muy pequeñas.

Recuérdese que una imagen en color consta de tres componentes (a menudo RGB, pero a veces YCbCr o YPbPr).[8] En JPEG, la DCT se aplica a cada componente por separado, elevando el número total de operaciones aritméticas a $3 \times 2q^2 8^3 = 3072q^2$. Para una imagen de 512×512 píxeles, esto implica $3072 \times 64^2 = 12\,582\,912$ multiplicaciones (y el mismo número de adiciones).

3. Otra forma de acelerar la DCT es llevar a cabo todas las operaciones aritméticas en coma fija (en escala entera) en lugar de números en coma flotante. En muchas computadoras, las operaciones sobre números de coma fija requieren (algunas) técnicas de programación sofisticada, pero son considerablemente más rápidas que las operaciones en coma flotante (salvo en las supercomputadoras, que están optimizadas para la aritmética de coma flotante).

Posiblemente, el mejor algoritmo DCT se describe en [Feig y Linzer 90]. Utiliza 54 multiplicaciones y 468 sumas y desplazamientos. Hoy en día, también hay varios chips VLSI que realizan este cálculo de manera eficiente.

[8] RGB es la cantidad de rojo verde y azul; Y es la luminancia, Cb y Cr son respectivamente los componentes de crominancia para la diferencia en azul (blue) y la diferencia en rojo (red); YPbPr es similar a YCbCr, el primero se utiliza en sistemas analógicos y el último para el video digital.

Matriz	Adiciones	Multiplicaciones
$\mathbf{C}_1$	0	0
$\mathbf{C}_2$	8	12
$\mathbf{C}_3$	4	0
$\mathbf{C}_4$	2	0
$\mathbf{C}_5$	0	2
$\mathbf{C}_6$	4	0
$\mathbf{C}_7$	8	0
Total:	26	14

Tabla 4.45: Número de operaciones aritméticas.

4.6.6. El método LLM

Esta sección describe el método de Loeffler–Ligtenberg–Moschytz (LLM) para la DCT unidimensional [Loeffler et al. 89]. Desarrollado en 1989 por Christoph Loeffler, Adriaan Ligtenberg, y George S. Moschytz, este algoritmo calcula la DCT en una dimensión con un total de 29 adiciones y 11 multiplicaciones. Recuérdese que la DCT unidimensional consiste en multiplicar un vector fila por una matriz. Para $n = 8$, la multiplicación de la fila por una columna de la matriz requiere ocho multiplicaciones y siete adiciones, por lo que el número total de operaciones necesarias para toda la operación es de 64 multiplicaciones y 56 adiciones. La reducción del número de multiplicaciones de 64 a 11 representa un ahorro del 83 %, y la reducción del número de adiciones de 56 a 29 representa un ahorro del 49 % —¡muy significativo!—

Sólo se muestra aquí el resultado final; el lector interesado puede consultar los detalles en la publicación original. Comenzamos con la suma doble de la Ecuación (4.15) y realizamos algunos retoques algebraicos que la reducen a la forma $\mathbf{CPC}^T$, donde $\mathbf{P}$ es la matriz de 8×8 de los píxeles, $\mathbf{C}$ es la matriz definida por la Ecuación (4.20) y $\mathbf{C}^T$ es la transpuesta de $\mathbf{C}$. En el caso unidimensional, sólo es necesaria una multiplicación de la matriz, a saber, $\mathbf{PC}$. Los autores de este método muestran que la matriz $\mathbf{C}$ puede escribirse como el producto de siete sencillas matrices, como se muestra en la Figura 4.46.

Aunque el número de matrices se ha incrementado, el problema se ha simplificado, ya que nuestras siete matrices son dispersas y están formadas en su mayoría por 1's y -1's. La multiplicación por 1 o por -1 no requiere una multiplicación, y la multiplicación de algo por 0 ahorra una adición. La Tabla 4.45 resume el número total de operaciones aritméticas necesarias para multiplicar un vector fila por los siete matrices.

Estos números sorprendentemente pequeños pueden reducirse aún más mediante la siguiente observación. Observamos que la matriz $\mathbf{C}_2$ tiene tres grupos de cuatro cosenos cada uno. Uno de los grupos se compone de (ignoramos la $\sqrt{2}$) dos $\cos\frac{6}{16}\pi$ y dos $\cos\frac{2}{16}\pi$ (uno con un signo negativo). Usamos la identidad trigonométrica $\cos\left(\frac{\pi}{2}-\alpha\right)=\sin\alpha$ para reemplazar los dos $\pm\cos\frac{2}{16}\pi$ por $\pm\sin\frac{6}{16}\pi$. La multiplicación de cualquier matriz por $\mathbf{C}_2$ ahora se traduce en productos de la forma $A\cos\left(\frac{6}{16}\pi\right)-B\sin\left(\frac{6}{16}\pi\right)$ y $B\cos\left(\frac{6}{16}\pi\right)-A\sin\left(\frac{6}{16}\pi\right)$. Aparentemente, el cálculo de estos dos elementos requiere cuatro multiplicaciones y dos adiciones (suponiendo que una sustracción toma el mismo tiempo para ejecutarse que una adición). El cálculo siguiente, sin embargo, obtiene el mismo resultado con tres adiciones y tres multiplicaciones:

$$T=(A+B)\cos\alpha,\quad T-B\,(\cos\alpha-\sin\alpha)\,,\quad -T+A\,(\cos\alpha+\sin\alpha)\,,$$

Por consiguiente, los tres grupos ahora requieren nueve adiciones y nueve multiplicaciones, en lugar de las 6 adiciones y 12 multiplicaciones originales (se necesitan dos adiciones más para los otros elementos distintos de cero de $\mathbf{C}_2$), que modifican los totales de la parte inferior de la Tabla 4.45 a 29 adiciones

$$
\mathbf{C} = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}
$$

$$
\times \begin{bmatrix}
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \sqrt{2}\cos\frac{6\pi}{16} & \sqrt{2}\cos\frac{2\pi}{16} & 0 & 0 & 0 & 0 \\
0 & 0 & -\sqrt{2}\cos\frac{2\pi}{16} & \sqrt{2}\cos\frac{6\pi}{16} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \sqrt{2}\cos\frac{7\pi}{16} & 0 & 0 & \sqrt{2}\cos\frac{\pi}{16} \\
0 & 0 & 0 & 0 & 0 & \sqrt{2}\cos\frac{3\pi}{16} & \sqrt{2}\cos\frac{5\pi}{16} & 0 \\
0 & 0 & 0 & 0 & 0 & -\sqrt{2}\cos\frac{5\pi}{16} & \sqrt{2}\cos\frac{3\pi}{16} & 0 \\
0 & 0 & 0 & 0 & -\sqrt{2}\cos\frac{\pi}{16} & 0 & 0 & \sqrt{2}\cos\frac{7\pi}{16}
\end{bmatrix}
$$

$$
\times \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1
\end{bmatrix}
\times \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}
$$

$$
\times \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1/\sqrt{2} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1/\sqrt{2} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}
\times \begin{bmatrix}
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}
$$

$$
\times \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & -1
\end{bmatrix}
$$

$$
= \mathbf{C}_1\mathbf{C}_2\mathbf{C}_3\mathbf{C}_4\mathbf{C}_5\mathbf{C}_6\mathbf{C}_7.
$$

Figura 4.46: El producto de siete matrices.

$$\begin{array}{rrrrrrrr}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
b_1 & b_3 & b_5 & b_7 & -b_7 & -b_5 & -b_3 & -b_1 \\
c_1 & c_3 & -c_3 & -c_1 & -c_1 & -c_3 & c_3 & c_1 \\
b_3 & -b_7 & -b_1 & -b_5 & b_5 & b_1 & b_7 & -b_3 \\
a & -a & -a & a & a & -a & -a & a \\
b_5 & -b_1 & b_7 & b_3 & -b_3 & -b_7 & b_1 & -b_5 \\
c_3 & -c_1 & c_1 & -c_3 & -c_3 & c_1 & -c_1 & c_3 \\
b_7 & -b_5 & b_3 & -b_1 & b_1 & -b_3 & b_5 & -b_7
\end{array}$$

Tabla 4.47: Seis valores del coseno distintos para la DCT-2.

$$[(p_0-p_7),(p_1-p_6),(p_2-p_5),(p_3-p_4)]\begin{bmatrix} b_1 & b_3 & b_5 & b_7 \\ b_3 & -b_7 & -b_1 & -b_5 \\ b_5 & -b_1 & b_7 & b_3 \\ b_7 & -b_5 & b_3 & -b_1 \end{bmatrix} \rightarrow [G_1,G_3,G_5,G_7], \quad \text{(I)}$$

$$[(p_0+p_7),(p_1+p_6),(p_2+p_5),(p_3+p_4)]\begin{bmatrix} 1 & c_1 & a & c_3 \\ 1 & c_3 & -a & -c_1 \\ 1 & -c_3 & -a & c_1 \\ 1 & -c_1 & a & -c_3 \end{bmatrix} \rightarrow [G_0,G_2,G_4,G_6]. \quad \text{(II)}$$

Figura 4.48: Una implementación hardware de la DCT-2.

y 11 multiplicaciones.

> No hay ciencia nacional, así como no hay una tabla de multiplicar nacional; lo nacional ya no es ciencia.
>
> —Anton Chekhov.

4.6.7. Implementación hardware de la DCT

La Tabla 4.43 muestra los 64 valores de ángulo de la DCT-2 para $n = 8$. Cuando se calculan los cosenos de los ángulos, encontramos que —debido a la simetría de la función coseno—, hay sólo seis valores distintos no triviales del coseno. Éstos se resumen en la Tabla 4.47, donde $a = 1/\sqrt{2}$, $b_i = \cos(i\pi/16)$, y $c_i = \cos(i\pi/8)$. Los seis valores no triviales son: b_1, b_3, b_5, b_7, c_1, y c_3.

Esta característica puede ser explotada en una implementación de software rápida de la DCT o para construir un dispositivo hardware sencillo que calcule los coeficientes G_i de la DCT, para ocho valores de píxel p_i. La Figura 4.48 muestra cómo puede ser organizado tal dispositivo en dos partes —cada una calcula cuatro de los ocho coeficientes G_i—. La parte I se basa en una matriz simétrica de 4×4, cuyos elementos son los cuatro b_i's distintos. Los ocho píxeles están divididos en cuatro grupos de dos píxeles cada uno. Los dos píxeles de cada grupo se restan, y las cuatro diferencias se convierten en un vector fila que se ha multiplicado por las cuatro columnas de la matriz para producir los cuatro coeficientes de la DCT: G_1, G_3, G_5, G_7. La parte II se basa en una matriz similar de 4×4, cuyos elementos no triviales son los dos c_i's. Los cálculos son similares excepto que los dos píxeles de cada grupo se suman, en lugar de restarse.

4.6.8. Descomposición QR de una matriz

En esta sección se proporciona material de apoyo para la técnica de descomposición QR de una matriz. Está dirigido a quienes ya estén familiarizados con las matrices que quieren dominar este método.

Cualquier matriz $\mathbf{A}$ puede ser factorizada en la matriz producto $\mathbf{Q} \times \mathbf{R}$, donde $\mathbf{Q}$ es una matriz ortogonal y $\mathbf{R}$ es triangular superior. Si $\mathbf{A}$ es también ortogonal, entonces $\mathbf{R}$ será también ortogonal. Sin embargo, una matriz triangular superior que es también ortogonal debe ser diagonal. La ortogonalidad de $\mathbf{R}$ implica $\mathbf{R}^{-1} = \mathbf{R}^T$ y que sea diagonal implica $\mathbf{R}^1 \times \mathbf{R} = \mathbf{I}$. La conclusión es que si $\mathbf{A}$ es ortogonal, entonces $\mathbf{R}$ debe satisfacer $\mathbf{R}^T \times \mathbf{R} = \mathbf{I}$, lo que significa que sus elementos de la diagonal deben $+1$ ó -1. Si $\mathbf{A} = \mathbf{Q} \times \mathbf{R}$, y $\mathbf{R}$ tiene esta forma, entonces $\mathbf{A}$ y $\mathbf{Q}$ son idénticas, excepto que las columnas i de $\mathbf{A}$ y $\mathbf{Q}$ tendrán signos opuestos para todos los valores de i, donde $\mathbf{R}_{i,i} = -1$.

La descomposición QR de la matriz $\mathbf{A}$ en $\mathbf{Q}$ y $\mathbf{R}$ se realiza mediante un bucle en el que cada iteración convierte un elemento de $\mathbf{A}$ a cero. Cuando todos los elementos de $\mathbf{A}$ por debajo de la diagonal se han puesto a cero, ésta se convierte en la matriz triangular superior $\mathbf{R}$. Cada elemento $\mathbf{A}_{i,j}$ es cero multiplicando $\mathbf{A}$ por una matriz de *rotación de Givens* $T_{i,j}$. Ésta es una matriz antisimétrica en la que los dos elementos de la diagonal $T_{i,i}$ y $T_{j,j}$ se establecen en el coseno de un cierto ángulo θ, y los dos elementos externos a la diagonal $T_{j,i}$ y $T_{i,j}$ se establecen en el seno y el seno negativo, respectivamente, del mismo θ. El seno y el coseno de θ se definen como:

$$\cos\theta = \frac{A_{j,j}}{D}, \quad \sin\theta = \frac{A_{i,j}}{D}, \quad \text{donde } D = \sqrt{A_{j,j}^2 + A_{i,j}^2}.$$

Los siguientes son algunos ejemplos de matrices de rotación de Givens:

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ 0 & c & s \\ 0 & -s & c \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c & 0 & s \\ 0 & 0 & 1 & 0 \\ 0 & -s & 0 & c \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & c & 0 & s & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & -s & 0 & c & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}. \tag{4.21}$$

Aquellos familiarizados con matrices de rotación reconocerán que una matriz de Givens [Givens 58] gira un punto hasta un ángulo cuyo seno y coseno son la s y la c de la Ecuación (4.21). En dos dimensiones, la rotación se hace sobre el origen. En tres dimensiones, se hace sobre uno de los ejes de coordenadas [el eje x, en la Ecuación (4.21)]. En cuatro dimensiones, la rotación es alrededor de dos de los cuatro ejes de coordenadas, (la primera y la tercera en la Ecuación (4.21)) y no puede ser visualizada. En general, una matriz de Givens de $n \times n$ rota un punto sobre $n-2$ ejes de coordenadas de un espacio n-dimensional.

J. Wallace Givens, Jr. (1910–1993) fue pionero en el uso de rotaciones de planos en los inicios de los cálculos automáticos de matrices. Givens se graduó en el Lynchburg College en 1928, y completó su Ph.D. en la Universidad de Princeton en 1936. Después de pasar tres años en el Institute for Advanced Study (Instituto de Estudios Avanzados) de Princeton como asistente de Oswald Veblen, Givens aceptó un cargo en la Universidad de Cornell, pero más tarde se trasladó a la Universidad de Northwestern. Además de su carrera académica, Givens fue el director de la Applied Mathematics Division (División de Matemáticas Aplicadas) en el Argonne National Lab y, al igual que su homólogo Alston Householder del Oak Ridge National Laboratory, Givens ejerció como uno de los primeros presidentes de SIAM. Publicó su trabajo sobre las rotaciones en el año 1958.

—Carl D. Meyer.

La Figura 4.49 es una función en *Matlab* para la descomposición QR de una matriz $\mathbf{A}$. Observe cómo la matriz $\mathbf{Q}$ se obtiene como el producto de las matrices de Givens individuales y cómo el doble

```
function [Q,R]=QRdecompose(A);
% Calcula la descomposición QR de la matriz A
% R es una matriz triangular superior y Q
% una matriz ortogonal de manera que A=Q*R.
[m,n]=size(A);% determina las dimensiones de A
Q=eye(m);% Q comienza como la matriz identidad mxm
R=A;
for p=1:n
  for q=(1+p):m
    w=sqrt(R(p,p)^2+R(q,p)^2);
    s=-R(q,p)/w; c=R(p,p)/w;
    U=eye(m);% Construye una matriz U para la rotación de Givens
    U(p,p)=c; U(q,p)=-s; U(p,q)=s; U(q,q)=c;
    R=U'*R;% una rotación de Givens
    Q=Q*U;
  end
end
```

Figura 4.49: Una función en Matlab para la descomposición QR de una matriz.

bucle pone a cero todos los elementos por debajo de la diagonal columna por columna de abajo arriba.

> "¡Ordenador!', gritó Zaphod, "rota ángulo de visión a unoochenta grados y ¡no informes de ello!'
>
> —Douglas Adams, *The Hitchhiker's Guide to the Galaxy.*
> *(Guía del autoestopista galáctico)*

4.6.9. Espacios vectoriales

La transformada discreta del coseno también se puede interpretar como un cambio de base en un espacio vectorial desde la base estándar a la base de la DCT, por lo que esta sección es una breve discusión sobre espacios vectoriales, su relación con la compresión de datos y con la DCT, sus bases, y las operaciones importantes para el cambio de base.

Un espacio vectorial n-dimensional es el conjunto de todos los vectores de la forma $(v_1, v_2, \ldots, v_n)$. Limitamos la discusión al caso en el que los v_i son números reales. El atributo de los espacios vectoriales que los hace tan importantes para nosotros es la existencia de *bases*. Cualquier vector (a, b, c) en tres dimensiones puede ser escrito como la combinación lineal:

$$(a, b, c) = a\,(1, 0, 0) + b\,(0, 1, 0) + c\,(0, 0, 1) = a\mathbf{i} + b\mathbf{j} + c\mathbf{k},$$

por lo que decimos que el conjunto de tres vectores **i**, **j**, **k** forma una base del espacio vectorial tridimensional. Observe que los tres vectores de la base son ortogonales; el producto escalar de cualesquier dos de ellos es cero. También son ortonormales; el producto escalar de cada uno con sí mismo es 1. Es conveniente tener una base ortonormal, pero ésto no es un requisito. Las bases ni siquiera tienen que ser ortogonales.

El conjunto de tres vectores **i**, **j**, **k**, puede extenderse a cualquier número de dimensiones. Una base para un espacio vectorial n-dimensional puede estar formada por n vectores v_i para $i = 1, 2, \ldots, n$, donde el elemento j del vector v_i es la función delta de Kronecker δ_{ij}. Esta sencilla base es la *base estándar (o canónica)* del espacio vectorial n-dimensional. Además de esta base, el espacio vectorial n-dimensional puede tener otras bases. Mostramos otras dos bases para $n = 8$.

> Dios creó los números enteros, todo lo demás es obra del hombre.
>
> —Leopold Kronecker.

La base de la DCT (sin normalizar) está formada por los ocho vectores:

$$\begin{aligned}
&(1,1,1,1,1,1,1,1), \quad (1,1,1,1,-1,-1,-1,-1),\\
&(1,1,-1,-1,-1,-1,1,1), \quad (1,-1,-1,-1,1,1,1,-1),\\
&(1,-1,-1,1,1,-1,-1,1), \quad (1,-1,1,1,-1,-1,1,-1),\\
&(0,0,0,0,1,-1,0,0), \quad (0,0,0,0,0,0,1,-1).
\end{aligned}$$

Observe cómo sus elementos corresponden a las frecuencias más altas. La base wavelet (sin normalizar) de Haar (Sección 5.6) se compone de los ocho vectores:

$$\begin{aligned}
&(1,1,1,1,1,1,1,1), \quad (1,1,1,1,-1,-1,-1,-1),\\
&(1,1,-1,-1,0,0,0,0), \quad (0,0,0,0,1,1,-1,-1),\\
&(1,-1,0,0,0,0,0,0), \quad (0,0,1,-1,0,0,0,0),\\
&(0,0,0,0,1,-1,0,0), \quad (0,0,0,0,0,0,1,-1).
\end{aligned}$$

Para entender por qué estas bases son útiles para la compresión de datos, recordemos que nuestros vectores de datos son imágenes o partes de imágenes. Los píxeles de una imagen están normalmente correlacionados, pero la base estándar no adquiere ventajas de ésto. El vector de todos 1's, por el contrario, está incluido en las bases anteriores porque este vector por sí mismo es suficiente para expresar cualquier imagen uniforme. Por consiguiente, un grupo de píxeles idénticos $(v, v, \ldots, v)$ se puede representar como el único coeficiente: v veces el vector de todo 1's. [La transformada discreta del seno de la Sección 4.6.11 es inadecuada para la compresión de datos, principalmente debido a que no incluyen este vector uniforme.] El vector de la base $(1,1,1,1,-1,-1,-1,-1)$ puede representar la energía de un grupo de píxeles que es mitad oscura y mitad brillante. Por consiguiente, el grupo de píxeles $(v, v, \ldots, v, -v, -v, \ldots, -v)$ está representado por el único coeficiente: v veces dicho vector de la base. Los sucesivos vectores de la base representan imágenes de más alta frecuencia, hasta llegar al vector $(1,-1,1,-1,1,-1,1,-1)$. Este vector de la base se asemeja a un tablero de ajedrez y, por lo tanto, aísla los detalles de alta frecuencia de una imagen. Ésos detalles son normalmente los menos importantes y pueden ser fuertemente cuantificados —o incluso reducidos a cero— para lograr una mejor compresión.

Los vectores miembros de una base no tienen que ser ortogonales. Para que un conjunto S de vectores formen una base, tienen que tener las siguientes dos propiedades: (1) Los vectores tienen que ser linealmente independientes y (2) debe ser posible expresar cualquier miembro del espacio vectorial como una combinación lineal de los vectores de S. Por ejemplo, los tres vectores $(1,1,1)$, $(0,1,0)$ y $(0,0,1)$ no son ortogonales, pero forman una base del espacio vectorial tridimensional. (1) son linealmente independientes porque ninguno de ellos puede se expresar como una combinación lineal de los otros dos. (2) Cualquier vector (a,b,c) puede expresarse como la combinación lineal $a\,(1,1,1)+(b-a)\,(0,1,0)+(c-a)\,(0,0,1)$.

Una vez que sabemos que un espacio vectorial puede tener muchas bases, empezamos a buscar buenas bases. Una base buena para la compresión de datos es una donde la inversa de la matriz de la base es fácil de calcular y donde la energía de un vector de datos se concentra en unos pocos coeficientes. Las bases discutidas hasta ahora son sencillas, se basan en ceros y unos. Las bases ortogonales tienen la ventaja añadida de que la inversa de la matriz de la base es simplemente su transpuesta. Ser rápida no es suficiente, porque lo mejor que podemos hacer es permanecer con la base estándar original. La razón para cambiar una base es conseguir compresión. La base de la DCT tiene la ventaja añadida de

que concentra la energía de un vector de valores correlacionados en unos pocos coeficientes. Sin esta propiedad, no habría ninguna razón para cambiar los coeficientes de un vector desde la base estándar a la base de la DCT. Después de cambiar a la base de la DCT, pueden cuantificarse muchos coeficientes —a veces, incluso a cero— con una pérdida de sólo la información menos importante de la imagen. Si cuantificamos los valores de los píxeles originales en la base estándar, también logramos compresión, pero perdemos información de la imagen que puede ser importante.

Una vez que se ha seleccionado una base, es fácil de expresar cualquier vector dado en términos de los vectores de la base. Suponiendo que los vectores de la base son $\mathbf{b}_i$ y dado un vector arbitrario $\mathbf{P} = (p_1, p_2, \ldots, p_n)$, podemos escribir $\mathbf{P}$ como una combinación lineal $\mathbf{P} = c_1\mathbf{b}_1 + c_2\mathbf{b}_2 + \cdots + c_n\mathbf{b}_n$ de los $\mathbf{b}_i$'s con coeficientes desconocidos c_i. Usando la notación de matrices, ésto se escribe: $\mathbf{P} = \mathbf{c} \cdot \mathbf{B}$, donde $\mathbf{c}$ es un vector fila de los coeficientes, y $\mathbf{B}$ es la matriz cuyas filas son los vectores de la base. Los coeficientes desconocidos pueden calcularse con: $c = \mathbf{P} \cdot \mathbf{B}^{-1}$ y ésta es la razón por la cual una buena base es aquella donde la inversa de la matriz de la base es fácil de calcular.

Un ejemplo sencillo son los coeficientes de un vector bajo la base estándar. Hemos visto, que el vector (a, b, c) se puede escribir como la combinación lineal $a\,(1, 0, 0) + b\,(0, 1, 0) + c\,(0, 0, 1)$. Por consiguiente, cuando se utiliza la base estándar, los coeficientes de un vector $\mathbf{P}$ son simplemente sus elementos originales. Si ahora desea comprimir el vector cambiando a la base de la DCT, necesitamos calcular los coeficientes en la nueva base. Éste es un ejemplo de la importancia de la operación de *cambio de base*.

Dadas dos bases $\mathbf{b}_i$ y $\mathbf{v}_i$, y suponiendo que un vector $\mathbf{P}$ dado puede expresarse como $\sum c_i\mathbf{b}_i$ y también como $\sum \omega_i\mathbf{v}_i$, el problema del cambio de base consiste en expresar un conjunto de coeficientes de una base en términos de la otra. Puesto que los vectores $\mathbf{v}_i$ constituyen una base, cualquier vector se puede expresar como una combinación lineal de ellos. Específicamente, cualquier $\mathbf{b}_j$ puede escribirse como $b_j = \sum_i t_{ij}\mathbf{v}_i$ para algunos números t_{ij}. Ahora construimos una matriz $\mathbf{T}$ con los t_{ij} y observamos que satisface $\mathbf{b}_i\mathbf{T} = \mathbf{v}_i$ para $i = 1, 2, \ldots, n$. Por consiguiente, $\mathbf{T}$ es una *transformación lineal* que transforma la base $\mathbf{b}_i$ en la $\mathbf{v}_i$. Los números t_{ij} son los elementos de $\mathbf{T}$ en la base $\mathbf{v}_i$.

Para nuestro vector $\mathbf{P}$, ahora podemos escribir $\left(\sum c_i\mathbf{b}_i\right)\mathbf{T} = \sum c_i\mathbf{v}_i$, lo que implica:

$$\sum_{j=1}^{n} \omega_j\mathbf{v}_j = \sum_j \omega_j\mathbf{b}_j\mathbf{T} = \sum_j \omega_j \sum_i \mathbf{v}_i t_{ij} = \sum_i \left(\sum_j \omega_j t_{ij}\right)\mathbf{v}_i.$$

Ésto demuestra que $c_i = \sum_j t_{ij}\omega_j$; en otras palabras, una base se cambia por medio de una transformación lineal $\mathbf{T}$ y la misma transformación también se refiere a los elementos de un vector en la base antigua y en la nueva.

Una vez que cambiamos a una nueva base en un espacio vectorial, cada vector tiene nuevas coordenadas y cada transformación tiene una matriz diferente.

Una transformación lineal $\mathbf{T}$ opera en un vector n-dimensional y produce un vector m-dimensional; por lo tanto, $\mathbf{T}(\mathbf{v})$ es un vector $\mathbf{u}$. Si $m = 1$, la transformación produce un escalar. Si $m = n - 1$, la transformación es una proyección. Las transformaciones lineales satisfacen las dos importantes propiedades: $\mathbf{T}(\mathbf{u} + \mathbf{v}) = \mathbf{T}(\mathbf{u}) + \mathbf{T}(\mathbf{v})$ y $\mathbf{T}(c\mathbf{v}) = c\mathbf{T}(\mathbf{v})$. En general, la transformación lineal de una combinación lineal $\mathbf{T}(c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \cdots + c_n\mathbf{v}_n)$ es igual a la combinación lineal de las transformaciones individuales: $c_1\mathbf{T}(\mathbf{v}_1) + c_2\mathbf{T}(\mathbf{v}_2) + \cdots + c_n\mathbf{T}(\mathbf{v}_n)$. Ésto implica que el vector cero se transforma en sí mismo bajo cualquier transformación lineal.

Ejemplos de transformaciones lineales son: la proyección, la reflexión, la rotación, y la diferenciación de un polinomio. La derivada de $c_1 + c_2x + c_3x^2$ es $c_2 + 2c_3x$. Ésta es una transformación desde la base (c_1, c_2, c_3) en el espacio tridimensional, a la base (c_2, c_3) en el espacio bidimensional. La matriz

de transformación satisface $(c_1, c_2, c_3)\,\mathbf{T} = (c_2, 2c_3)$, por lo que viene dado por:

$$T = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 2 \end{bmatrix}.$$

Ejemplos de transformaciones no lineales son: la traslación, la longitud de un vector, y la adición de un vector constante v_0. Ésta última no es lineal porque si $\mathbf{T}(\mathbf{v}) = \mathbf{v} + \mathbf{v}_0$ y doblamos el tamaño de $\mathbf{v}$, entonces $\mathbf{T}(2\mathbf{v}) = 2\mathbf{v} + \mathbf{v}_0$ es diferente de $2\mathbf{T}(\mathbf{v}) = 2(\mathbf{v} + \mathbf{v}_0)$. La transformación de un vector $\mathbf{v}$ en su longitud $\parallel v \parallel$ es también no lineal porque $\mathbf{T}(-\mathbf{v}) \neq -\mathbf{T}(\mathbf{v})$. La traslación es no lineal, ya que transforma el vector cero en un vector no nulo.

En general, una transformación lineal se realiza multiplicando el vector transformado $\mathbf{v}$ por la matriz de la transformación $\mathbf{T}$. Por consiguiente, $\mathbf{u} = \mathbf{v} \cdot \mathbf{T}$ o $\mathbf{T}(\mathbf{v}) = \mathbf{v} \cdot \mathbf{T}$. Observe que denotamos por $\mathbf{T}$ tanto a la transformación como a su matriz.

Con el fin de describir una transformación única, es suficiente para describir qué hace a los vectores de una base. Para ver por qué ésto es cierto, observamos lo siguiente. Si para un vector dado $\mathbf{v}_1$ sabemos que es $\mathbf{T}(\mathbf{v}_1)$, entonces sabemos que $\mathbf{T}(a\mathbf{v}_1)$ se cumple para cualquier a. De manera similar, si para un determinado $\mathbf{v}_2$ sabemos que es $\mathbf{T}(\mathbf{v}_2)$, entonces sabemos que $\mathbf{T}(b\mathbf{v}_2)$ se cumple para cualquier b y también es $\mathbf{T}(a\mathbf{v}_1 + b\mathbf{v}_2)$. Por consiguiente, sabemos cómo transforma $\mathbf{T}$ cualquier vector en el plano que contiene $\mathbf{v}_1$ y $\mathbf{v}_2$. Este argumento demuestra, que si sabemos lo que $\mathbf{T}(\mathbf{v}_i)$ es para todos los vectores $\mathbf{v}_i$ de una base, entonces sabemos cómo transforma $\mathbf{T}$ cualquier vector en el espacio vectorial.

Dado una base $\mathbf{b}_i$ de un espacio vectorial, consideramos la transformación especial que afecta la magnitud de cada vector, pero no su dirección. Por consiguiente, $\mathbf{T}(\mathbf{b}_i) = \lambda_i \mathbf{b}_i$ para algunos números λ_i. La base $\mathbf{b}_i$ es la base de *vectores propios* de la transformación $\mathbf{T}$. Como conocemos $\mathbf{T}$ para toda la base, también la conocemos para cualquier otro vector. Cualquier vector $\mathbf{v}$ del espacio vectorial se puede expresar como una combinación lineal $\mathbf{v} = \sum_i c_i \mathbf{b}_i$. Si aplicamos nuestra transformación a ambos lados y utilizamos la propiedad de linealidad, nos encontramos con:

$$\mathbf{T}(\mathbf{v}) = \mathbf{v} \cdot \mathbf{T} = \sum_i c_i \mathbf{b}_i \cdot \mathbf{T}. \tag{4.22}$$

En el caso especial donde $\mathbf{v}$ es el vector de la base $\mathbf{b}_1$, la Ecuación (4.22) implica $\mathbf{T}(\mathbf{b}_1) = \sum_i c_i \mathbf{b}_i \cdot \mathbf{T}$. Por otro lado, $\mathbf{T}(\mathbf{b}_1) = \lambda_1 \mathbf{b}_1$. Por consiguiente, concluimos que $c_1 = \lambda_1$ y, en general, que la transformación de matriz $\mathbf{T}$ es diagonal con λ_i en la posición i de su diagonal.

En la base de vectores propios, la matriz de transformación es diagonal, así que ésta es la base perfecta. Nos encantaría tenerla en la compresión, pero depende de los datos. Se llama transformada de Karhunen–Loève (KLT) y se describe en la Sección 4.5.4.

4.6.10. Rotaciones en tres dimensiones

Para los exégetas que quieren la historia completa, en los párrafos siguientes se muestra cómo un matriz de rotación apropiada (con un determinante de +1) que rota un punto (v, v, v) hacia el eje x se puede deducir desde la matriz de rotación general en tres dimensiones.

Una rotación general en tres dimensiones queda completamente especificada por (1) un eje de rotación $\underline{\mathbf{u}}$, (2) el ángulo de rotación θ, y (3) la dirección (en el sentido de las agujas del reloj o en el sentido contrario, visto desde el origen) de la rotación alrededor de $\mathbf{u}$. Dado un vector unitario $\mathbf{u} = (u_x, u_y, u_z)$, la matriz $\mathbf{M}$ de la Ecuación (4.23) efectúa una rotación de θ^o alrededor de $\mathbf{u}$. La rotación se muestra en el sentido de las agujas del reloj para un observador que mira desde el origen en la dirección de $\mathbf{u}$. Si $\mathbf{P} = (x, y, z)$ es un punto arbitrario, su posición después de la rotación está

dada por el producto $\mathbf{P}\cdot\mathbf{M}$.

$$\mathbf{M}= \tag{4.23}$$
$$\begin{pmatrix} u_x^2+\cos\theta\left(1-u_x^2\right) & u_xu_y\left(1-\cos\theta\right)-u_z\sin\theta & u_xu_z\left(1-\cos\theta\right)+u_y\sin\theta \\ u_xu_y\left(1-\cos\theta\right)+u_z\sin\theta & u_y^2+\cos\theta\left(1-u_y^2\right) & u_yu_z\left(1-\cos\theta\right)+u_x\sin\theta \\ u_xu_z\left(1-\cos\theta\right)-u_y\sin\theta & u_yu_z\left(1-\cos\theta\right)+u_x\sin\theta & u_z^2+\cos\theta\left(1-u_z^2\right) \end{pmatrix}.$$

La rotación general de la Ecuación (4.23) se puede aplicar ahora a nuestro problema, que consiste en rotar el vector $\mathbf{D}=(1,1,1)$ hasta el eje $\mathbf{x}$. La rotación debe hacerse de modo que el vector $\mathbf{u}$ sea perpendicular tanto a $\mathbf{D}$ como a $(1,0,0)$. Este vector se calcula mediante el producto cruzado $\mathbf{u}=\mathbf{D}\times(1,0,0)=(0,1,-1)$. La normalización produce $\mathbf{u}=(0,\alpha,-\alpha)$, donde $\alpha={}^1/\sqrt{2}$.

El siguiente paso consiste en calcular el ángulo θ entre $\mathbf{D}$ y el eje x. Ésto se hace normalizando $\mathbf{D}$ y calculando el producto escalar del mismo y el eje x (recordemos que el producto escalar de dos vectores unidad es el coseno del ángulo que forman). $\mathbf{D}$ normalizado es (β,β,β), donde $\beta={}^1/\sqrt{3}$, y el resultado del producto escalar es $\cos\theta=\beta$, que también produce $\sin\theta=-\sqrt{1-\beta^2}=-\sqrt{2/3}=-\beta/\alpha$. La razón para el signo negativo es que una rotación desde $(1,1,1)$ hasta $(1,0,0)$ arlrededor de $\mathbf{u}$ se muestra en el sentido contrario a las agujas del reloj para un observador que mira desde el origen en la dirección de $\mathbf{u}$ positivo. La matriz de rotación de la Ecuación (4.23) se obtuvo para la dirección opuesta de la rotación. Además, $\cos\theta=\beta$ implica que $\theta=54{,}76^\circ$. Este ángulo —y no 45°—, es el ángulo formado por el vector $\mathbf{D}$ con cada uno de los tres ejes de coordenadas. [Como nota aparte, cuando aumenta el número de dimensiones, el ángulo entre el vector $(1,1,\ldots,1)$ y cualquiera de los ejes de coordenadas se aproxima a 90°.]

Sustituyendo $\mathbf{u}$, $\sin\theta$, y $\cos\theta$ y en la Ecuación (4.23) y usando las relaciones $\alpha^2+\beta\left(1-\alpha^2\right)=(\beta+1)/2$ y $-\alpha^2(1-\beta)=(\beta-1)/2$ se obtiene la senilla matriz de rotación:

$$M=\begin{bmatrix} \beta & -\beta & -\beta \\ \beta & \alpha^2+\beta\left(1-\alpha^2\right) & -\alpha^2(1-\beta) \\ \beta & -\alpha^2(1-\beta) & \alpha^2+\beta\left(1-\alpha^2\right) \end{bmatrix}=\begin{bmatrix} \beta & -\beta & -\beta \\ \beta & (\beta+1)/2 & (\beta-1)/2 \\ \beta & (\beta-1)/2 & (\beta+1)/2 \end{bmatrix}$$
$$\approx\begin{bmatrix} 0{,}5774 & -0{,}5774 & -0{,}5774 \\ 0{,}5774 & 0{,}7886 & -0{,}2115 \\ 0{,}5774 & -0{,}2115 & 0{,}7886 \end{bmatrix}.$$

Ahora es fácil ver que un punto en la recta $x=y=z$, con coordenadas (v,v,v) es rotado $\mathbf{M}$ a $(v,v,v)\,\mathbf{M}=(1{,}7322v,\,0\,0)$. Observe que el determinante de $\mathbf{M}$ es igual a $+1$, por lo que $\mathbf{M}$ es una matriz de rotación, en contraste con la matriz de la Ecuación (4.17), que genera rotaciones impropias.

4.6.11. La transformada discreta del seno

Los lectores que llegaron a este punto pueden plantearse la cuestión: ¿por qué se utiliza en la transformación la función coseno, y no la seno? ¿Es posible utilizar la función seno de forma similar a la DCT para crear una transformada discreta del seno? ¿Existe una DST? y si no ¿Por qué? Esta breve sección analiza las diferencias entre las funciones seno y coseno y muestra por qué estas diferencias llevan a una ineficiente transformada discreta del seno.

Una función $f(x)$ que satisface $f(x)=-f(-x)$ se llama *impar*. De manera similar, una función que cumple $f(x)=f(-x)$ se llama *par*. En una función impar, siempre es cierto que $f(0)=-f(-0)=-f(0)$, por lo que $f(0)$ debe ser 0. La mayor parte de las funciones no son ni impares, ni pares; sin embargo, las funciones trigonométricas seno y coseno constituyen importantes ejemplos de funciones impar y par, respectivamente. La Figura 4.50 muestra que, aunque la única diferencia entre ellas es la fase (i.e., el coseno es una versión desplazada del seno), esta diferencia es suficiente para invertir su paridad. Cuando la curva del seno (impar) se desplaza, se convierte en la curva coseno (par), que tiene la misma forma.

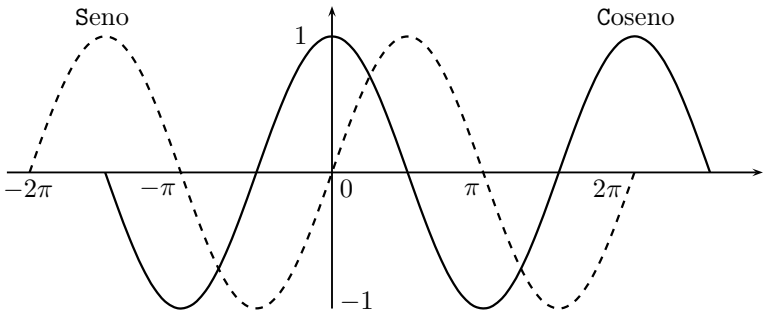


Figura 4.50: El seno y el coseno como funciones impar y par, respectivamente.

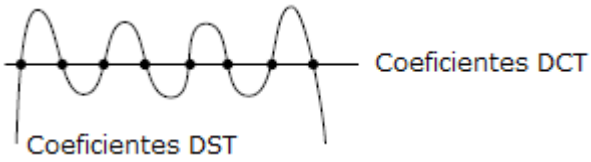


Figura 4.51: La DCT y la DST de ocho valores de datos idénticos.

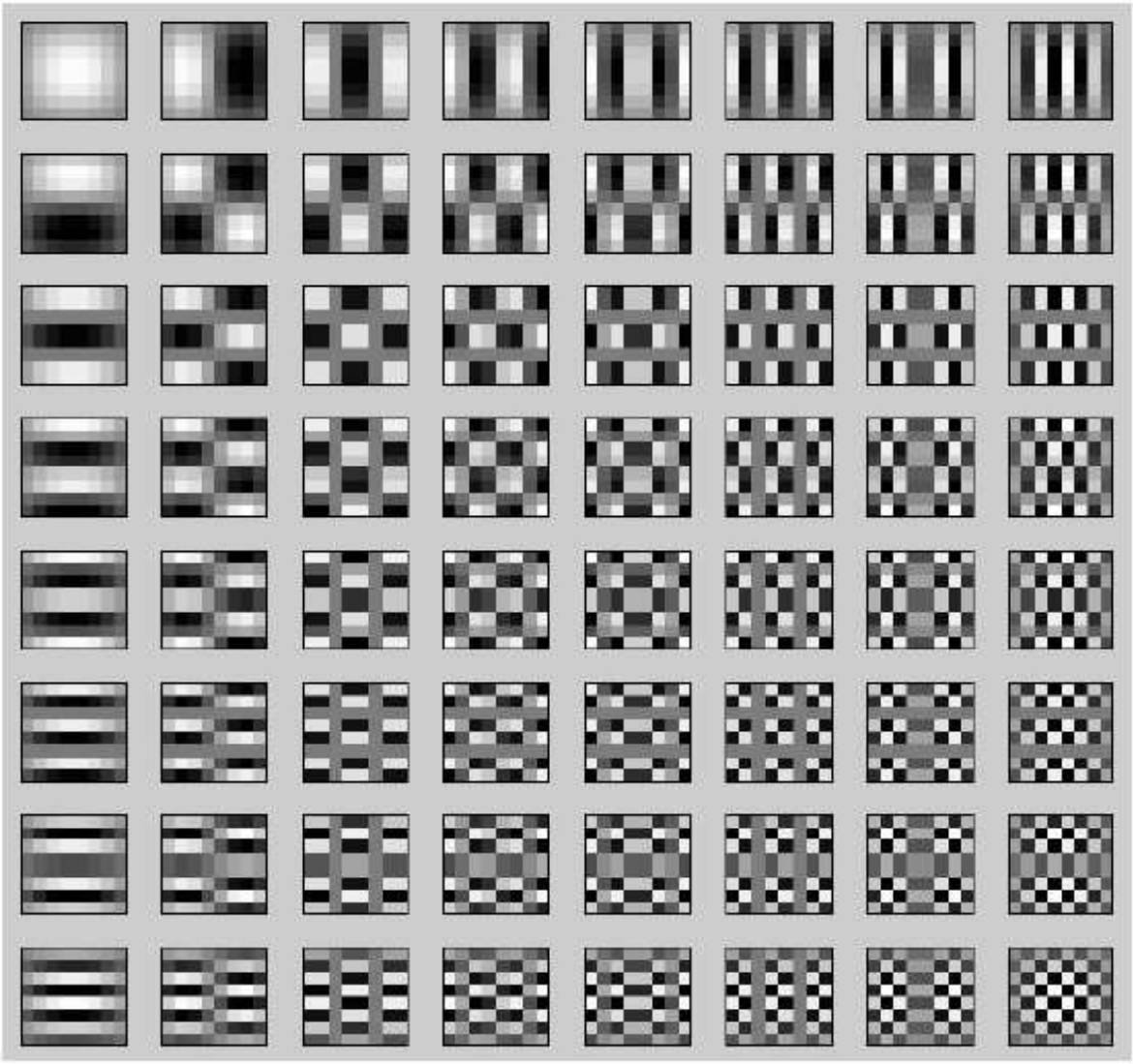
Para comprender la diferencia entre la DCT y la DST, analizamos el caso en una dimensión. La DCT unidimensional —Ecuación (4.13)— emplea la función $\cos[(2t+1)f\pi/16]$ con $f = 0, 1, \ldots, 7$. Para el primer término, donde $f = 0$, esta función convierte en $\cos(0)$, que es 1. Este término es el importante coeficiente DC ya conocido, que es proporcional a la media de los ocho valores de datos que están siendo transformados. La DST se basa similarmente en la función $\sin[(2t+1)f\pi/16]$, resultando su primer término cero [ya que $\sin(0) = 0$]. El primer término no contribuye en nada a la transformación, por lo que la DST no tiene un coeficiente DC.

La desventaja de ésto puede verse cuando consideramos el ejemplo de ocho valores de datos idénticos que están siendo transformados por la DCT y por la DST. Los valores idénticos están, por supuesto, perfectamente correlacionados. Cuando se trazan, se convierten en una línea horizontal. La aplicación de la DCT a estos valores produce sólo un coeficiente DC: Todos los coeficientes AC son cero. La DCT compacta toda la energía de los datos en el único coeficiente DC cuyo valor es idéntico a los valores de los ítems de datos. La IDCT puede reconstruir los ocho valores a la perfección (excepto por pequeños cambios producidos por la limitada precisión de la máquina). Aplicando la DST a los mismos ocho valores, por otro lado, se producen siete coeficientes AC cuya suma es una función de onda que pasa a través de los ocho puntos de datos pero que oscila entre los puntos. Este comportamiento —que se ilustra en la Figura 4.51— tiene tres desventajas, a saber: (1) la energía de los valores de los datos originales no se compacta, (2) los siete coeficientes no están descorrelacionados (ya que los valores de los datos están perfectamente correlacionados), y (3) la cuantificación de los siete coeficientes puede reducir considerablemente la calidad de la reconstrucción realizada por la DST inversa.

Ejemplo: La aplicación de la DST a los ocho valores idénticos 100 produce los ocho coeficientes (0, 256,3, 0, 90, 0, 60,1, 0, 51). Utilizando estos coeficientes, la IDST puede reconstruir los valores originales, pero es fácil ver que los coeficientes AC no se comportan como los de la DCT. No son cada vez más pequeños, y no hay *runs* de ceros entre ellos. La aplicación de la DST a los ocho valores altamente correlacionados 11, 22, 33, 44, 55, 66, 77, y 88 produce un conjunto de coeficientes aún peores:

$$(0,\ 126{,}9, -57{,}5,\ 44{,}5, -31{,}1,\ 29{,}8, -23{,}8,\ 25{,}2)\,.$$

No hay energía compactada en absoluto.



```
N=8;
m=[1:N]'*ones(1,N); n=m';
% puede usarse también el cos en lugar del sin
%A=sqrt(2/N)*cos(pi*(2*(n-1)+1).*(m-1)/(2*N));
A=sqrt(2/N)*sin(pi*(2*(n-1)+1).*(m-1)/(2*N));
A(1,:)=sqrt(1/N);
C=A';
for row=1:N
  for col=1:N
    B=C(:,row)*C(:,col).';%tensor producto
    subplot(N,N,(row-1)*N+col)
    imagesc(B)
    drawnow
  end
end
```

Figura 4.52: Las 64 imágenes base de la DST en dos dimensiones.

Estos argumentos y ejemplos, junto con el hecho (discutido en [Ahmed et al. 74] y [Rao y Yip 90]) de que la DCT produce coeficientes altamente descorrelacionados, sostienen firmemente el uso de la DCT en oposición a la DST en la compresión de datos.

◇ **Ejercicio 4.15 (sol. en pág. 1081):** Utilícese software matemático para calcular y mostrar las 64 imágenes base de la DST en dos dimensiones para $n = 8$.

> Somos la brizna de paja, el juguete de los vientos. Creemos que estamos alcanzando un objetivo deliberadamente elegido; el destino nos conduce a otro. Las matemáticas, la exagerada preocupación de mi juventud, apenas me ha hecho algún servicio; y los animales, que evitaba tanto como podía, son el consuelo de mi vejez. Sin embargo, no guardo rencor al seno y al coseno, que sigo teniendo en alta estima. Me han costado muchas horas seguidas de dedicación, pero siempre me produjeron cierto de entretenimiento de primer nivel: todavía lo hacen, cuando mi la cabeza se encuentra lanzando ideas insomne en su almohada.
>
> —J. Henri Fabre, *La vida de la mosca*.

4.7. Imágenes de prueba

Los nuevos métodos de compresión de datos que se desarrollan y se implementan tienen que ser probados. Probar diferentes métodos con los mismos datos hace posible la comparación de su rendimiento tanto en eficiencia de compresión como en velocidad. Por eso hay colecciones de datos de prueba estándar, como el Calgary Corpus y el Canterbury Corpus (mencionado en el Prefacio), y el conjunto de la ITU-T de ocho documentos de adiestramiento para la compresión de faxes (Sección 2.13.1).

La necesidad de datos de prueba estándar también se ha hecho notar en el campo de la compresión de imágenes, y no existen en la actualidad colecciones de imágenes fijas de uso general para los investigadores e implementadores en este campo. Tres de las cuatro imágenes que aquí se muestran —a saber, "Lena", "mandril", y "pimientos"— son, sin duda, las más conocidas. Éstas son imágenes de tonos continuos, a pesar de que "Lena" tiene algunas características inherentes a una imagen de tonos discretos.

Cada imagen se acompaña de un detalle, mostrando los píxeles individuales. Es fácil ver por qué la imagen "pimientos" es de tonos continuos. Los píxeles adyacentes que difieren mucho en color son bastante raros en esta imagen. La mayoría de píxeles vecinos son muy similares. En contraste, la imagen "mandril", aunque natural, es un mal ejemplo de una imagen de tonos continuos. El detalle (centrado en una parte del ojo derecho y el área alrededor del mismo), muestra que el número de píxeles difieren considerablemente de sus vecinos inmediatos, debido al vello facial de los animales en este área. Esta imagen se comprime mal bajo cualquier método de compresión. Sin embargo, la zona de la nariz, en su mayoría azul y rojo, es de tonos continuos. La imagen de "Lena" está compuesta, en su mayor parte, por tonos continuos puros, sobre todo en la pared y en las áreas de piel al descubierto. El sombrero es una buena zona de tonos continuos, mientras que el pelo y la pluma en el sombrero son malas áreas para tonos continuos. Las líneas rectas en la pared y las partes curvadas del espejo son características de una imagen de tonos discretos.

La imagen de "Lena" es ampliamente utilizada por la comunidad de procesamiento de imágenes, además de ser popular en la compresión de imágenes. Debido al interés en ella, su origen e historia

Figura 4.53: Lena y un detalle.

han sido investigados y están bien documentados. Esta imagen forma parte de la página central de la revista *Playboy* de noviembre de 1972. Presenta a la *playmate*[9] sueca Lena Soderberg (née Sjooblom), y fue descubierta, cortada, y escaneada a principios de 1970 por un investigador desconocido en la Universidad del Sur de California para su uso como imagen de prueba para su investigación en la compresión de imágenes. Desde entonces se ha convertido en la imagen más importante, mejor conocida, y comúnmente usada en la historia de las imágenes y las comunicaciones electrónicas. Como resultado, Lena está considerado por muchos como la Primera Dama de Internet. *Playboy*, que normalmente enjuicia a los usuarios no autorizados de sus imágenes, se ha enterado del uso habitual de una de sus imágenes con derechos de autor, pero decidió dar su bendición a esta particular "aplicación".

La propia Lena vive actualmente en Suecia. Se le informó de su "fama" en 1988, quedó sorprendida por ello y le pareció divertido, y fue invitada a participar en la conferencia del 50º aniversario de la IS&T (la sociedad para la Imaging Science and Technology —ciencia de la imagen y la tecnología—) celebrada en Boston, en mayo de 1997. Allí autografió su foto, posó para nuevas fotos (disponibles en la www), e hizo una presentación (de sí misma, no sobre compresión).

Las tres imágenes están ampliamente disponibles para su descarga en Internet.

La Figura 4.57 muestra una imagen de tonos discretos típica, con un detalle mostrado en la Figura 4.58. Observe las líneas rectas y el texto, donde ciertos caracteres aparecen varias veces (una fuente de redundancia). Esta imagen en particular tiene pocos colores, pero en general, una imagen de tonos discretos puede tener muchos colores.

[9]Es aquella modelo elegida para las páginas centrales de la revista *Playboy*. Literalmente es: "compañera de juegos".

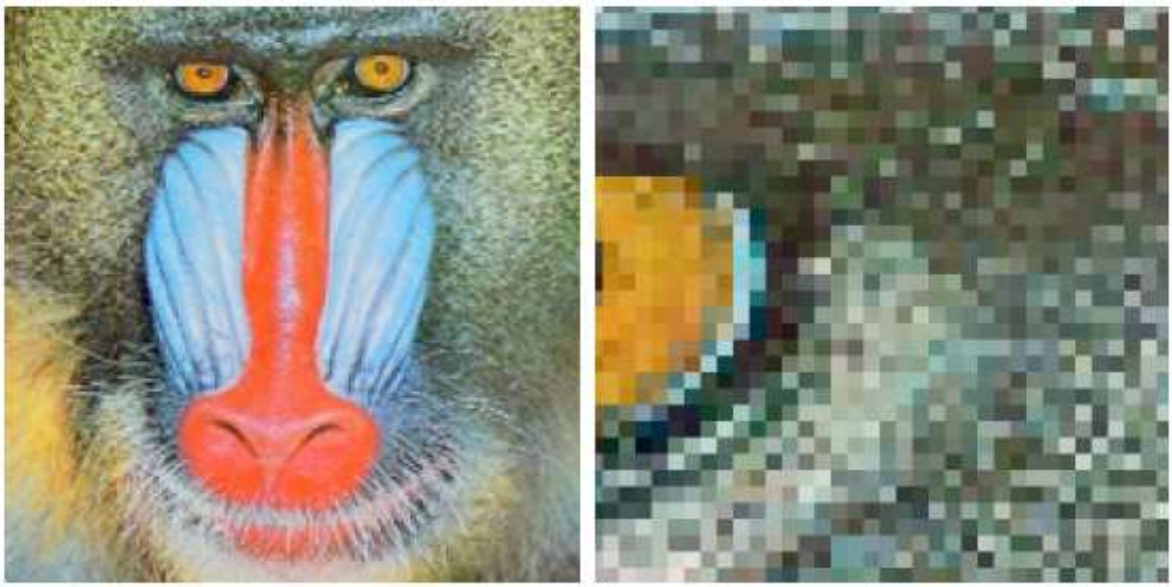


Figura 4.54: Mandril y un detalle.

Figura 4.55: Artefactos bloque en JPEG.

Figura 4.56: Pimientos y un detalle.

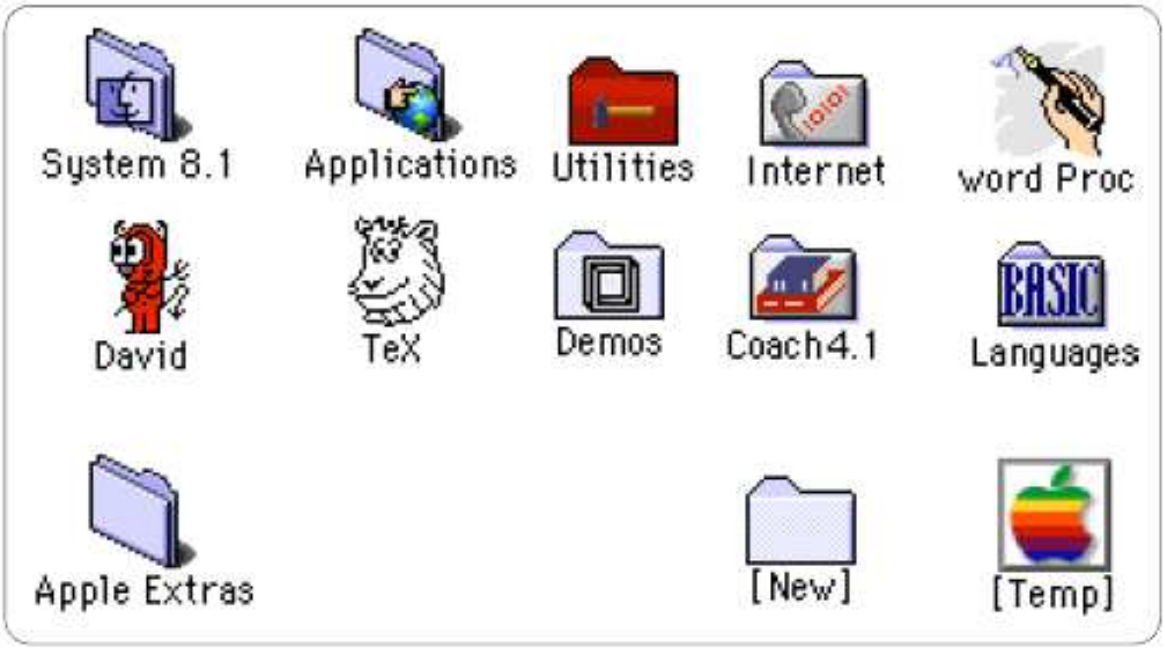


Figura 4.57: Una imagen de tonos discretos.

Figura 4.58: Una imagen de tonos discretos (Detalle).

Lena, Illinois, es una comunidad de aproximadamente 2900 personas. Lena se considera que es una comunidad limpia y segura ubicada en el centro de las ciudades más grandes que ofrecen otros intereses cuando son necesarios. Lena está a 2-1/2 millas del parque estatal del lago Le-Aqua-Na. El parque ofrece rutas de senderismo, pesca, playas para nadar, botes, esquí de fondo, senderos a caballo, así como áreas de picnic y de camping. Es un parque hermoso y bien cuidado, gratis para el público. ¡Un gran lugar para pasear en trineo y patinar sobre el hielo en el invierno! (Desde `http://www.villageoflena.com/`)

4.8. JPEG

JPEG es un método de compresión sofisticado —con (*lossy*) o sin (*lossless*) pérdidas— para imágenes estáticas (vídeo no), en color o en escala de grises. No maneja muy bien imágenes binivel (en blanco y negro). Además, funciona mejor con imágenes de tonos continuos, donde los píxeles adyacentes tienen colores similares. Una característica importante de JPEG es el uso de muchos parámetros, permitiendo al usuario ajustar la cantidad de datos perdidos (y en consecuencia, también la razón de compresión) en un rango muy amplio. A menudo, el ojo no puede ver ninguna degradación de la imagen incluso a factores de compresión de 10 ó 20. Hay dos modos de funcionamiento: con pérdidas (también llamado *baseline* o base) y sin pérdidas (que típicamente produce razones de compresión cercanas a 0,5). La mayoría de las implementaciones soportan sólo el modo con pérdidas. Este modo incluye la codificación progresiva y jerárquica. Algunas de las muchas las referencias a JPEG son: [Pennebaker y Mitchell 92], [Wallace 91], y [Zhang 90].

JPEG es un método de compresión, no un estándar completo para la representación de imágenes. Es por ésto que no especifica características de la imagen tales como la relación de aspecto de los píxeles, el espacio de color (*color space*), o el entrelazado de las filas del mapa de bits.

JPEG ha sido diseñado como un método de compresión para imágenes de tonos continuos. Los principales objetivos de la compresión JPEG son los siguientes:

1. Unas razones de compresión elevadas, especialmente en los casos donde se juzga la calidad de la imagen de muy buena a excelente.

2. El uso de muchos parámetros, permitiendo a los usuarios con conocimientos experimentar y lograr la compensación deseada en la relación compresión/calidad.

3. La obtención de buenos resultados con cualquier tipo de imagen de tonos continuos, independientemente de las dimensiones de la imagen, espacios de color, relaciones de aspecto de los píxeles u otras características de la imagen.

4. Un método de compresión sofisticado, pero no demasiado complejo, permitiendo implementaciones en software y en hardware para muchas plataformas.

5. Varios modos de funcionamiento: (a) Un modo secuencial, donde cada componente de la imagen (color) se comprime en un solo escaneo de izquierda a derecha y de arriba a abajo; (b) Un modo progresivo en el que se comprime la imagen en bloques múltiples (conocidos como "scans", "escaneos" o "exploraciones") para poder visualizarla desde el detalle grueso (imagen tosca) a fino (con mayor definición); (c) Un modo sin pérdidas, que es importante en los casos donde el usuario decide que no debe perder píxeles (la disyuntiva es una baja razón de compresión en comparación con los modos con pérdidas), y (d) un modo jerárquico donde se comprime la imagen con múltiples resoluciones, permitiendo que los bloques de más baja resolución puedan visualizarse sin tener que descomprimir los siguientes bloques con una mayor resolución.

El nombre JPEG es un acrónimo de Joint Photographic Experts Group (Grupo de expertos en ensamble fotografico). Éste fue un esfuerzo conjunto por el CCITT y la ISO (International Standards

Organization u organización de estándares internacional) que comenzó en junio de 1987 y produjo la primera propuesta de proyecto sobre JPEG en 1991. El estándar JPEG ha tenido éxito y ha sido ampliamente utilizado en la compresión de imágenes, especialmente en las páginas Web.

Estos son los principales pasos de compresión de JPEG, cada uno de los cuales se describirá posteriormente en detalle:

1. Las imágenes en color se transforman de RGB en un espacio de color de luminancia/crominancia (Sección 4.8.1; este paso se omite para imágenes en escala de grises). El ojo es sensible a pequeños cambios en la luminancia, pero no en la crominancia, por lo que la parte de crominancia puede posteriormente perder muchos datos y, por lo tanto, ser altamente comprimida, sin perjudicar visualmente la calidad global de la imagen en demasía. Este paso es opcional, pero importante, porque el resto del algoritmo trabaja con cada componente de color por separado. Sin transformar el espacio de color, ninguno de los tres componentes de color tolerará muchas pérdidas, dando lugar a una peor compresión.

2. Las imágenes en color se muestrean disminuyendo su resolución mediante la creación de píxeles en baja resolución de los originales (este paso se utiliza sólo cuando se selecciona la compresión jerárquica; siempre se omite para la escala de grises). La disminución de la resolución no se realiza para el componente de luminancia. La disminución de resolución se lleva a cabo, ya sea en una proporción de 2:1 tanto horizontal como verticalmente (el llamado `2h2v` ó muestreo $4:1:1$), ya sea en proporciones de 2:1 horizontalmente y 1:1 verticalmente (`2h1v` ó muestreo $4:2:2$). Puesto que ésto se efectúa en dos de los tres componentes de color, `2h2v` reduce la imagen a $1/3+(2/3)\times(1/4)=1/2$ de su tamaño original, mientras que `2h1v` la reduce a $1/3+(2/3)\times(1/2)=2/3$ de su tamaño original. Puesto que el componente de luminancia no se modifica, no hay ninguna pérdida apreciable en la calidad de la imagen. Las imágenes en escala de grises no pasan por este proceso.

3. Los píxeles de cada componente de color se organizan en grupos de 8×8 píxeles llamados *unidades de datos*, y cada unidad de datos es comprimida por separado. Si el número de filas o columnas de la imagen no es un múltiplo de 8, la fila inferior y la columna del extremo derecho se duplican tantas veces como sea necesario. En el modo no entrelazado, el codificador controla todas las unidades de datos del primer componente de la imagen, luego las unidades de datos del segundo componente, y finalmente las del tercer componente. En el modo entrelazado el codificador procesa las tres unidades de datos de la parte superior izquierda de los tres componentes de la imagen; luego, las tres unidades de datos a su derecha, y así sucesivamente. El hecho de que cada unidad de datos se comprima por separado es una de las desventajas del formato JPEG. Si el usuario pide la máxima compresión, la descompresión de la imagen puede exhibir artefactos bloque debidos a las diferencias entre los bloques. La Figura 4.55 es un ejemplo extremo de este efecto.

4. Se aplica entonces la transformada discreta del coseno (DCT, Sección 4.6) a cada unidad de datos para crear un mapa de 8×8 de los componentes de frecuencia (Sección 4.8.2). Ellos representan el valor promedio de los píxeles y los cambios sucesivos de alta frecuencia dentro del grupo. Éste prepara los datos de la imagen para el paso crucial de pérdida de información. Puesto que la DCT involucra la transcendental función del coseno, debe implicar una cierta pérdida de información debido a la limitada precisión de la aritmética del ordenador. Ésto significa que incluso sin el paso principal de pérdida de datos (paso 5 más adelante), habrá una cierta pérdida en la calidad de la imagen, pero normalmente es pequeña.

5. Cada uno de los 64 componentes de frecuencia en una unidad de datos se divide por un número separado llamado su *coeficiente de cuantificación* (*quantization coefficient* o QC), y luego lo redondea a un entero (Sección 4.8.3). Aquí es donde la información se pierde irreversiblemente.

Grandes QC causan más pérdidas, por lo que los componentes de alta frecuencia suelen tener típicamente grandes QCs. Cada uno de los 64 QCs es un parámetro JPEG y puede, en principio, ser especificado por el usuario. En la práctica, la mayoría de las implementaciones de JPEG utilizan las tablas QC recomendadas por el estándar JPEG para los componentes de luminancia y de crominancia de la imagen (Tabla 4.61).

6. Los 64 coeficientes de frecuencia cuantificados (que ahora son enteros) de cada unidad de datos son codificados utilizando una combinación de codificación RLE y de Huffman (Sección 4.8.4). Opcionalmente, puede usarse una variante de codificación aritmética conocida como codificador QM (Sección 2.16) en lugar de la codificación de Huffman.

7. El último paso añade encabezados y todos los parámetros de JPEG requeridos, y guarda el resultado. El archivo comprimido puede estar en uno de los tres formatos: (1) el formato de *intercambio* (*interchange*), en el que el archivo contiene la imagen comprimida y todas las tablas necesarias para el decodificador (principalmente las tablas de cuantificación y las de los códigos de Huffman), (2) el formato *abreviado* (*abbreviated*) para los datos de la imagen comprimida, donde el archivo contiene la imagen comprimida y no puede contener tablas (o sólo unas pocas tablas), y (3) el formato *abreviado* (*abbreviated*) para la tabla de datos de especificaciones, donde el archivo contiene sólo las tablas, y ninguna imagen comprimida. El segundo formato tiene sentido en los casos donde se utiliza la misma pareja codificador/decodificador, y tienen construidas internamente las mismas tablas. El tercer formato se utiliza en los casos en que muchas imágenes han sido comprimidas por el mismo codificador, usando las mismas tablas. Cuando esas imágenes necesitan ser descomprimidas, se envían a un decodificador precedidas por un archivo con la tabla de datos de especificaciones.

El decodificador JPEG revierte los pasos. (Por eso, JPEG es un método de compresión simétrico.)

El modo progresivo es una opción de JPEG. En este modo, los coeficientes de mayor frecuencia de la DCT se escriben en la secuencia de datos comprimidos en bloques llamados "scans". Cada scan que es leído y procesado por el decodificador mejora la nitidez de la imagen. La idea es utilizar las primeras exploraciones para crear rápidamente una vista previa borrosa de la imagen —de baja calidad— y posteriormente, o bien proceder con los scans restantes, o bien detener el proceso y rechazar la imagen. Como contrapartida, el codificador tiene que salvar todos los coeficientes de todas las unidades de datos en un buffer en la memoria antes de enviarlos en forma de scans, y también realizar todos los pasos para cada scan, lo que frena el modo progresivo.

La Figura 4.59a muestra un ejemplo de una imagen con resolución 1024×512. La imagen se dividide en $128 \times 64 = 8192$ unidades de datos, y cada una se transforma mediante la DCT, convirtiéndose en un conjunto de 64 números de 8 bits. La Figura 4.59b es un bloque cuya profundidad es de 8 192 unidades de datos, cuya altura corresponde a los 64 coeficientes de la DCT (el coeficiente DC es el de la parte superior, numerado con 0), y cuya anchura corresponde a los ocho bits de cada coeficiente.

Después de preparar todas las unidades de datos en un buffer en la memoria, el codificador las escribe en la secuencia de datos comprimidos usando uno de los dos métodos: *selección espectral* o *aproximaciones sucesivas* (Figura 4.59c,d). La primera exploración en cualquiera de los métodos es el conjunto de coeficientes DC. Si se utiliza la selección espectral, cada scan sucesivo consta de varios (una *banda* de) coeficientes AC consecutivos. Si se utilizan las aproximaciones sucesivas, el segundo scan se compone de los cuatro bits más significativos de todos los coeficientes AC, y en cada uno de los cuatro scans siguientes —numeradas del 3 a 6— añade un bit más significativo (bits 3 a 0, respectivamente).

En el modo jerárquico, el codificador almacena la imagen varias veces en el *stream* de salida, en varias resoluciones. Sin embargo, cada parte de alta resolución utiliza la información de las partes de baja resolución de dicho *stream* de salida, por lo que la cantidad total de información es menor que la

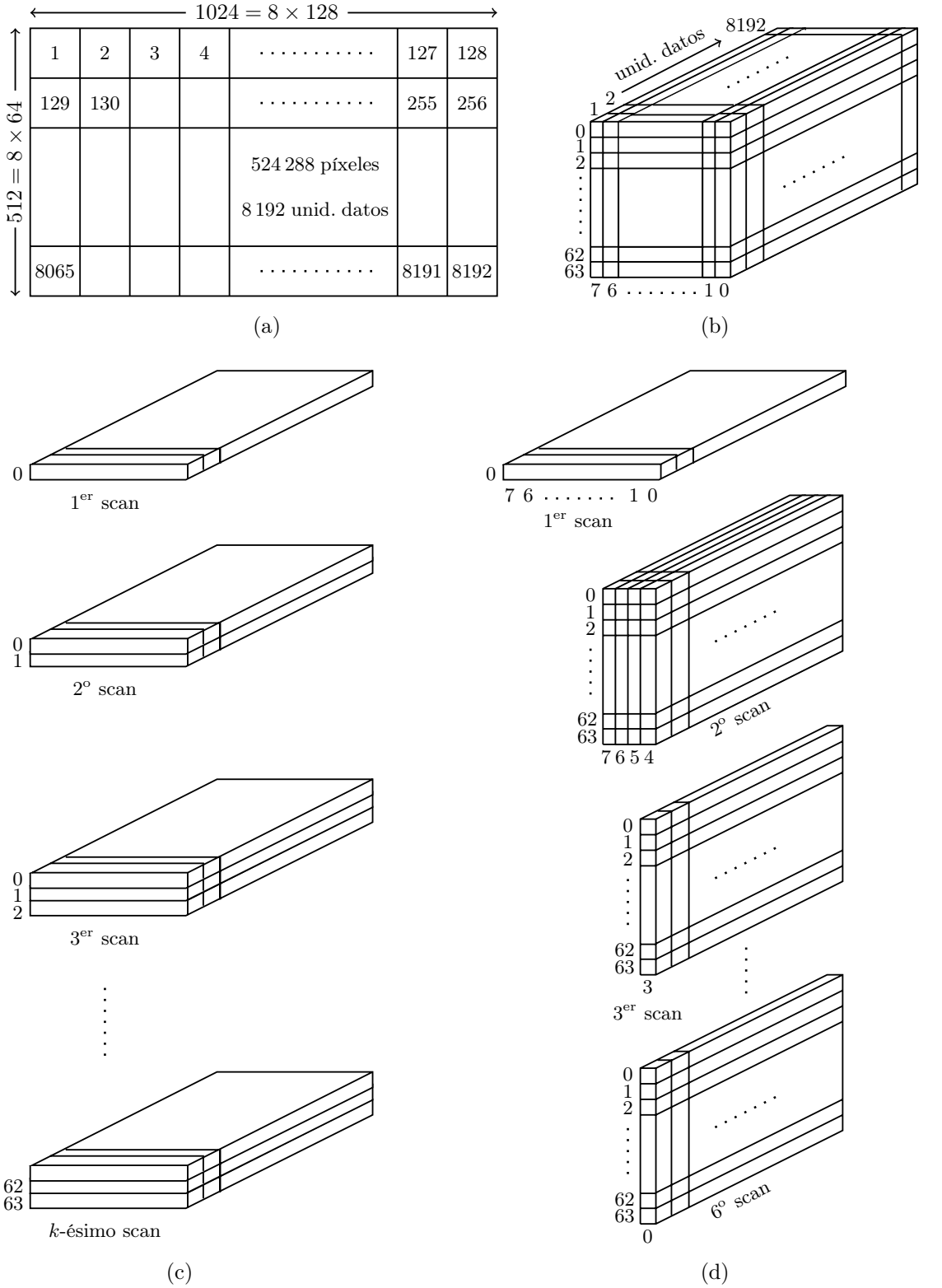


Figura 4.59: Scans en el modo progresivo de JPEG.

requerida para almacenar las diferentes resoluciones por separado. Cada parte jerárquica puede usar el modo progresivo.

El modo jerárquico es útil en aquellos casos en que una imagen de alta resolución necesita ser sacada en baja resolución. Las viejas impresoras matriciales pueden ser un buen ejemplo de dispositivo de salida de baja resolución que todavía está en uso.

El modo sin pérdidas de la imagen JPEG (Sección 4.8.5) calcula un valor "de predicción" para cada píxel, generando la diferencia entre el píxel y su valor predicho (véase en la Sección 1.3.1 la codificación relativa), y codifica la diferencia utilizando el mismo método (i.e., Huffman o codificación aritmética) empleado en el paso 5. El valor de predicción se calcula utilizando los valores de los píxeles ubicados por encima y a la izquierda del píxel actual (los píxeles que ya han sido introducidos y codificados). En las secciones siguientes se describen los pasos con más detalle:

4.8.1. Luminancia

La principal organización internacional dedicada a la luz y el color es el Comité Internacional de Iluminación (Commission Internationale de l'Éclairage), abreviado CIE. El cual es responsable de la elaboración de las normas y las definiciones en este área. Uno de los primeros logros de la CIE fue su *diagrama de cromaticidad* (*chromaticity diagram*) [Salomon 99], desarrollado en 1931. Éste demuestra que no se requieren más que tres parámetros para definir el color. La expresión de un color determinado por el triplete (x, y, z) es similar a la que denota un punto en un espacio tridimensional, de ahí el término *espacio de color*. El espacio de color más común es RGB, donde los tres parámetros son las intensidades de color rojo (Red), verde (Green) y azul (Blue) en un color. Cuando se utiliza en ordenadores, estos parámetros están normalmente en el rango $0 - 255$ (8 bits).

La CIE define el color como el resultado de la percepción de la luz en la región visible del espectro —que tiene longitudes de onda comprendidas entre 400 nm y 700 nm— incidiendo sobre la retina (un nanómetro —nm— equivale a 10^{-9} metros). La potencia física (o radiación) se expresa mediante una distribución espectral de potencia (*spectral power distribution* o SPD), a menudo con 31 componentes cada uno representando una banda de 10 nm.

El CIE define el brillo como el atributo de una sensación visual acorde con la mayor o menor emisión de luz que parece emitir un área determinada. La percepción por el cerebro del brillo es imposible de determinar, por lo que la CIE consideró una cantidad más práctica llamada *luminancia*. Ésta se define como la energía radiante ponderada por una función de sensibilidad espectral que es característica de la visión (el ojo es muy sensible al verde, un poco menos sensible al rojo, y mucho menos sensible al azul). La eficacia luminosa del observador estándar se define por el CIE como una función positiva de la longitud de onda, que tiene un máximo de aproximadamente 555 nm. Cuando una distribución de potencia espectral se integra mediante esta función como una función de ponderación, el resultado es la luminancia CIE, que se denota por Y. La luminancia es una cantidad importante en los campos del procesamiento y la compresión de imágenes digitales.

La luminancia es proporcional a la potencia de la fuente de luz. Es similar a la intensidad, pero la composición espectral de la luminancia está relacionada con la sensibilidad al brillo de la visión humana.

El ojo es muy sensible a pequeños cambios en la luminosidad, por lo que es útil tener espacios de color que utilizan Y como un de sus tres parámetros. Una sencilla manera de realizar ésto es restar Y de los componentes azul y rojo de RGB, y utilizar los tres componentes Y, $B - Y$, y $R - Y$ como un nuevo espacio de color. Los dos últimos componentes se denominan crominancia (chroma).

Representan el color en términos de la presencia o ausencia de color azul (Cb) y rojo (Cr) para una intensidad de luminancia dada.

La visión humana

Vemos la luz que entra en el ojo y llega a la retina, donde hay dos tipos de células fotosensibles. Éstas contienen pigmentos que absorben la luz visible y, por lo tanto, nos proporcionan la sensación de visión. Un tipo son los *bastones*, que son numerosos, están esparcidos por toda la retina, y sólo responden a la luz y a la oscuridad. Son muy sensibles y pueden responder a un solo fotón de luz. Hay alrededor de 110 000 000 a 125 000 000 bastones en el ojo [Osterberg 35]. El otro tipo son los *conos*, situados en una pequeña área de la retina (la fóvea). Su número es de unos 6 400 000, son sensibles al color, pero requieren una luz más intensa, del orden de cientos de fotones. Incidentalmente, los conos son muy sensibles al rojo, verde, y azul (Figura 4.60), que es una razón por la que CRT utiliza estos colores como primarios.

Cada uno de los sensores de luz en el ojo —bastones y conos— envía una sensación de luz al cerebro que es esencialmente un píxel, y el cerebro combina estos píxeles como una imagen continua. El ojo humano es, por lo tanto, similar a una cámara digital. Una vez que nos damos cuenta de ésto, es natural que queramos comparar la resolución del ojo a la de una cámara digital moderna. Las actuales cámaras digitales tienen de 300 000 sensores (para una cámara barata) a aproximadamente seis millones de sensores (para una de alta calidad).

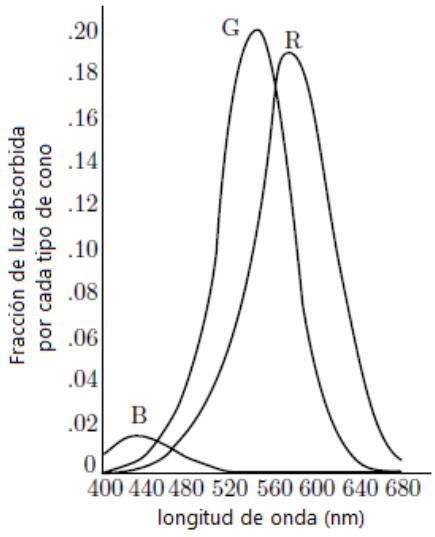


Figura 4.60: Sensibilidad de los conos.

Por consiguiente, el ojo dispone de una resolución mucho más alta, pero su resolución eficaz es aún mayor si consideramos que el ojo se puede mover y reorientar a sí mismo alrededor de tres a cuatro veces por segundo. Ésto significa que en un solo segundo, el ojo puede detectar y enviar al cerebro alrededor de la mitad de mil millones de píxeles. Suponiendo que nuestra cámara toma una instantánea una vez por segundo, la razón entre las resoluciones es aproximadamente 100.

Ciertos colores —como rojo, naranja, y amarillo— son psicológicamente asociados con el calor. Por eso se consideran *calientes* y hacen que una imagen parezca más grande y cercana de lo que realmente es. Otros colores —como azul, violeta y verde— se asocian con cosas interesantes (aire, cielo, agua, hielo) y, por lo tanto, se llaman colores *fríos*. Hacen que una imagen parezca más pequeña y lejana.

Se utilizan varios rangos numéricos para B − Y, y R − Y para diferentes aplicaciones. Los rangos de YPbPr están optimizados para los componentes del vídeo analógico. Los rangos de YCbCr son apropiados para los componentes del vídeo digital, como un video de estudio, JPEG, JPEG 2000, y MPEG.

El espacio de color YCbCr se desarrolló como parte de la Recomendación ITU-R BT.601 (antes CCIR 601) durante el desarrollo de un componente mundial de vídeo digital estándar. Y está definido dentro de un rango, de 16 a 235; Cb y Cr están definidos dentro de un rango, de 16 a 240, siendo 128

igual a cero. Hay varios formatos de muestreo YCbCr, como 4:4:4, 4:2:2, 4:1:1 y 4:2:0, que también se describen en la recomendación.

Las conversiones entre RGB con un rango $16-235$ y YCbCr son lineales y, por lo tanto, sencillas. La transformación de RGB a YCbCr se realiza con (observe que el pequeño peso del color azul):

$$\begin{aligned}\mathrm{Y} &= (77/256)\,\mathrm{R} + (150/256)\,\mathrm{G} + (29/256)\,\mathrm{B},\\ \mathrm{C}b &= -(44/256)\,\mathrm{R} + (87/256)\,\mathrm{G} + (131/256)\,\mathrm{B} + 128,\\ \mathrm{C}r &= (131/256)\,\mathrm{R} + (110/256)\,\mathrm{G} + (21/256)\,\mathrm{B} + 128;\end{aligned}$$

mientras que la transformación opuesta es:

$$\begin{aligned}\mathrm{R} &= \mathrm{Y} + 1{,}371\,(\mathrm{C}r - 128),\\ \mathrm{G} &= \mathrm{Y} - 0{,}698\,(\mathrm{C}r - 128) - 0{,}336\,(\mathrm{C}b - 128),\\ \mathrm{B} &= \mathrm{Y} + 1{,}732\,(\mathrm{C}b - 128).\end{aligned}$$

Al efectuar la conversión YCbCr a RGB, los valores RGB resultantes tienen un alcance nominal de 16 a 235, con posibles valores ocasionales en los rangos $0-15$ y $236-255$.

4.8.2. La DCT

El concepto general de transformación se discute en la Sección 4.4. La transformada discreta del coseno se discute en detalle en la Sección 4.6. Otros ejemplos de transformaciones importantes son: la transformada de Fourier (Sección 5.1) y la transformada wavelet (Capítulo 5). Ambas tienen aplicaciones en muchas áreas y también tienen versiones discretas (DFT y DWT).

El comité JPEG decidió utilizar la DCT, debido a su buena eficiencia, ya que no asume nada acerca de la estructura de los datos (la DFT, por ejemplo, supone que los datos a ser transformados son periódicos), y porque hay formas de hacerla más rápida (Sección 4.6.5).

El estándar JPEG demanda la aplicación de la DCT, no a toda la imagen, sino a las unidades de datos (bloques) de 8×8 píxeles. Las razones para ésto son: (1) La aplicación de la DCT a grandes bloques implica muchas operaciones aritméticas y es, por tanto, lento. La aplicación de la DCT a pequeñas unidades de datos es más rápido. (2) La experiencia demuestra que, en una imagen de tonos continuos, las correlaciones entre los píxeles son de corto alcance. Un píxel en tal imagen tiene un valor (componente de color o tono de gris) que está cercano al de sus vecinos más próximos, pero no tiene nada que ver con los valores de los vecinos lejanos. La DCT JPEG se puede calcular, por lo tanto, mediante la Ecuación (4.15), duplicada aquí para $n=8$.

$$G_{ij} = \frac{1}{4}C_iC_j\sum_{x=0}^{7}\sum_{y=0}^{7}p_{xy}\cos\left(\frac{(2x+1)\,i\pi}{16}\right)\cos\left(\frac{(2y+1)\,j\pi}{16}\right), \qquad (4.15)$$

$$\text{donde } C_f = \begin{cases}\frac{1}{\sqrt{2}}, & f=0,\\ 1, & f>0,\end{cases} \quad \text{y } 0\le i,j\le 7.$$

La DCT es la clave de JPEG para la compresión con pérdida. La información de la imagen sin importancia, se reduce o se elimina cuantificando los 64 coeficientes de la DCT; especialmente los ubicados hacia la parte inferior derecha. Si los píxeles de la imagen están correlacionados, la cuantización no degrada mucho la calidad de imagen. Para obtener los mejores resultados, cada uno de los 64 coeficientes se cuantifica dividiéndolo por un coeficiente de cuantificación diferente (QC). Los 64 QCs son parámetros que pueden ser controlados, en principio, por el usuario (Sección 4.8.3).

El decodificador JPEG funciona mediante el cálculo de la DCT inversa (IDCT), la Ecuación (4.16), duplicada aquí para $n = 8$.

$$p_{xy} = \frac{1}{4}\sum_{i=0}^{7}\sum_{j=0}^{7} C_i C_j G_{ij} \cos\left(\frac{(2x+1)\,i\pi}{16}\right)\cos\left(\frac{(2y+1)\,j\pi}{16}\right), \qquad (4.16)$$

$$\text{donde } C_f = \begin{cases} \frac{1}{\sqrt{2}}, & f = 0; \\ 1, & f > 0. \end{cases}$$

Ésta toma los 64 coeficientes cuantificados de la DCT y calcula los 64 píxeles p_{xy}. Si los QCs son los correctos, los 64 nuevos píxeles serán muy similares a los originales. Matemáticamente, la DCT es una aplicación uno a uno de los 64 vectores punto desde el dominio de la imagen al dominio de la frecuencia. La IDCT es la asignación inversa. Si la DCT y la IDCT pudieran ser calculadas con precisión infinita y si los coeficientes de la DCT no se cuantifican, los 64 píxeles originales serían reconstruidos con exactitud.

4.8.3. Cuantificación

Después de calcular cada unidad de datos de 8×8 de coeficientes G_{ij} de la DCT, se efectúa la cuantificación. Éste es el paso donde se pierde la información (a excepción de una pérdida inevitable, debida a la finitud de la precisión de los cálculos en los otros pasos). Cada número en la matriz de coeficientes de la DCT, se divide por el número correspondiente de la "tabla de cuantificación" particular utilizada, y el resultado se redondea al entero más cercano. Como ya se ha mencionado, se necesitan tres de tales tablas, para los tres componentes de color. El estándar JPEG permite hasta cuatro tablas, y el usuario puede seleccionar cualquiera de las cuatro para cuantificar cada componente de color. Los 64 números que constituyen cada tabla de cuantificación son todos los parámetros de JPEG. En principio, todos ellos pueden ser especificados y ajustados por el usuario para obtener la máxima compresión. En la práctica, pocos usuarios tienen la paciencia y la experiencia para experimentar con tantos parámetros, por lo que el software JPEG utiliza normalmente los dos métodos siguientes:

1. Tablas de cuantificación por defecto. Dos de dichas tablas, para la luminancia (escala de grises) y los componentes de crominancia, son el resultado de muchos experimentos realizados por el comité de JPEG. Se incluyen en el estándar JPEG y se reproducen aquí en la Tabla 4.61. Es fácil ver cómo los QCs en la tabla general, crecen a medida que nos movemos desde la esquina superior izquierda hasta la esquina inferior derecha. Así es como JPEG reduce los coeficientes de la DCT con frecuencias espaciales altas.

2. Se calcula una sencilla tabla de cuantificación Q, basada en un parámetro R especificado por el usuario. Una expresión sencilla como $Q_{ij} = 1 + (i + j) \times R$ garantiza que los QCs pequeños comiencen en la esquina superior izquierda y se hagan más grandes hacia la esquina inferior derecha. La Tabla 4.62 muestra un ejemplo de una tabla con $R = 2$.

Si la cuantificación se hace correctamente, quedarán muy pocos números distintos de cero en la matriz de coeficientes de la DCT, y típicamente estarán concentrados en la región superior izquierda. Estos números son la salida de JPEG, pero se comprimen aún más antes de ser escritos en la secuencia de datos de salida. En la literatura sobre JPEG esta compresión se llama "codificación de entropía", y la Sección 4.8.4 muestra en detalle cómo se hace. La codificación de entropía utiliza tres técnicas para comprimir la matriz de enteros de 8×8:

1. Los 64 números se recogen escaneando la matriz en zigzags (Figura 1.8b). Ésto produce una cadena de 64 números que comienza con algunos elementos distintos de cero y típicamente termina

16	11	10	16	24	40	51	61
12	12	14	19	25	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Luminancia

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

Crominancia

Tabla 4.61: Tablas de cuantificación recomendadas.

1	3	5	7	9	11	13	15
3	5	7	9	11	13	15	17
5	7	9	11	13	15	17	19
7	9	11	13	15	17	19	21
9	11	13	15	17	19	21	23
11	13	15	17	19	21	23	25
13	15	17	19	21	23	25	27
15	17	19	21	23	25	27	29

Tabla 4.62: Tabla de cuantificación $1 + (i + j) \times 2$.

con muchos ceros consecutivos. Sólo se sacan los números no nulos (después de comprimirlos aún más), y a continuación un código especial de fin de bloque (end-of block o EOB). De esta manera no hay necesidad de sacar los ceros posteriores (se puede decir que EOB es la codificación *run-length* de todos los ceros de la derecha). El lector interesado debería consultar también la Sección 8.5 para ver otros métodos sobre compresión de cadenas binarias con muchos ceros consecutivos.

◇ **Ejercicio 4.16 (sol. en pág. 1081):** Propóngase una forma práctica de escribir un bucle que recorra una matriz de 8×8 en zigzag.

2. Los números no nulos se comprimen utilizando la codificación de Huffman (Sección 4.8.4).
3. El primero de esos números (el coeficiente DC, página 294) es tratado de manera diferente a los otros (los coeficientes AC).

> Ella sólo había logrado una curva hacia abajo en un gracioso zigzag, e iba a zambullirse entre las hojas, que descubrió no eran más que las copas de los árboles bajo los que había estado vagando, cuando un silbido agudo la hizo retroceder rápidamente.
>
> —Lewis Carroll, *Alicia en el país de las Maravillas* (1865)

4.8.4. Codificación

En primer lugar, analizamos el anterior punto 3. Cada matriz 8×8 de coeficientes de la DCT cuantificados contiene un coeficiente DC [en la posición $(0, 0)$, la esquina superior izquierda] y 63

0:	0										0
1:	−1	1									10
2:	−3	−2	2	3							110
3:	−7	−6	−5	−4	4	5	6	7			1110
4:	−15	−14	...	−9	−8	8	9	10	...	15	11110
5:	−31	−30	−29	...	−17	−16	16	17	...	31	111110
6:	−63	−62	−61	...	−33	−32	32	33	...	63	1111110
7:	−127	−126	−125	...	−65	−64	64	65	...	127	11111110
⋮				⋮							
14:	−16383	−16382	−16381	...	−8193	−8192	8192	8193	...	16383	$\overbrace{1\ldots1}^{14}0$
15:	−32767	−32766	−32765	...	−16385	−16384	16384	16385	...	32767	$\overbrace{1\ldots1}^{15}0$
16:	32768										$\overbrace{1\ldots1}^{15}1$

Tabla 4.63: Codificación de diferencias de coeficientes DC.

coeficientes AC. El coeficiente DC es una medida del valor promedio de los 64 píxeles originales, que constituyen la unidad de datos. La experiencia demuestra que en una imagen de tonos continuos, las unidades de datos de los píxeles adyacentes normalmente están correlacionados en el sentido de que los valores medios de los píxeles en unidades de datos adyacentes están cercanos. Ya sabemos que el coeficiente DC de una unidad de datos es un múltiplo del promedio de los 64 píxeles que constituyen la unidad. Ésto implica que los coeficientes DC de las unidades de datos adyacentes no difieren en mucho. JPEG saca el primero (codificado), seguido por las *diferencias* (también codificadas) de los coeficientes DC de las unidades de datos consecutivas. El concepto de diferenciación se discute en la Sección 1.3.1.

Ejemplo: Si las tres primeras unidades de datos de 8×8 de una imagen tienen como coeficientes DC cuantificados 1118, 1114, y 1119, entonces la salida JPEG para la primera unidad de datos es 1118 (codificación de Huffman, véase más adelante), seguido por los 63 coeficientes AC (codificados) de ésa unidad de datos. La salida para segunda unidad de datos será $1114 - 1118 = -4$ (también codificado con Huffman), seguido por los 63 coeficientes AC (codificados) de ésa unidad de datos, y la salida para la tercera unidad de datos será $1119 - 1114 = 5$ (también codificado con Huffman), seguido de nuevo por los 63 coeficientes AC (codificados) de ésa unidad de datos. Esta forma de manejar los coeficientes DC vale la pena a pesar de las dificultades adicionales, porque las diferencias son pequeñas.

La codificación de las diferencias DC se lleva a cabo usando la Tabla 4.63, por lo que primero decimos aquí algunas palabras acerca de esta tabla. Cada fila tiene un número de fila (a la izquierda), el código unitario de la fila (a la derecha), y varias columnas en el medio. Cada fila contiene números mayores (y también más números) que su predecesora, pero no los números que figuran en las filas anteriores. La fila i contiene el rango de enteros $\left[-\left(2^i-1\right),+\left(2^i-1\right)\right]$, pero le falta el rango del medio $\left[-\left(2^{i-1}-1\right),+\left(2^{i-1}-1\right)\right]$. Por consiguiente, las filas se hacen muy largas, lo que significa que un sencillo array bidimensional no es una buena estructura de datos para esta tabla. De hecho, no es necesario almacenar estos números enteros en una estructura de datos, ya que el programa se puede averiguar dónde se encuentra en la tabla cualquier entero dado x analizando los bits de x.

El primer coeficiente DC a ser codificado en nuestro ejemplo es 1118. Reside en la fila 11, columna 930 de la tabla (la numeración de columna empieza en cero), por lo que se codifica como 111111111110 | 01110100010 (el código unario para la fila 11, seguido por el binario de 11 bits de 930). La segunda diferencia DC es −4. Está ubicada en la fila 3, columna 3 de la Tabla 4.63, por lo que se codifica como

1110 | 011 (el código unario para la fila 3, seguido por el valor binario de 3 bits de 3).

◇ **Ejercicio 4.17 (sol. en pág. 1082):** ¿Cómo es la tercera diferencia DC —5— codificada?

El punto 2 anterior está relacionado con la forma precisa como se comprimen los 63 coeficientes AC de una unidad de datos. Se utiliza una combinación de RLE, y o bien codificación Huffman, o bien codificación aritmética. La idea es que la secuencia de coeficientes AC normalmente contienen algunos números distintos de cero, con *runs* de ceros entre ellos, y con una larga secuencia de ceros al final. Para cada número x distinto de cero, el codificador: (1) encuentra el número de ceros consecutivos Z que preceden a x; (2) localiza x en la Tabla 4.63 y prepara su número de fila y de columna (R y C); (3) el par (R,Z) [que es (R,Z), no (R,C)] se utiliza como número de fila y de columna en la Tabla 4.66; y (4) el código de Huffman encontrado en ésa posición de la tabla se concatena con C (donde C se escribe como un número de R bits) y el resultado es (finalmente) el código emitido por el codificador JPEG para el coeficiente AC x y todos los ceros consecutivos que lo preceden.

Los códigos de Huffman de la Tabla 4.66 no son los recomendados por el estándar JPEG. Éste recomienda el uso de las Tablas 4.64 y 4.65, y dice que un codificador JPEG puede utilizar hasta cuatro tablas de códigos de Huffman, excepto que el modo base puede utilizar sólo dos de tales tablas. Los códigos reales en la Tabla 4.66 son, por tanto, arbitrarios.

El lector debe fijarse en el código EOB —en la posición (0,0)— y el código de ZRL —en la posición (0,15)—. El primero indica el final del bloque, y el último es el código emitido para 15 ceros consecutivos cuando el número de ceros consecutivos exceda de 15. Estos códigos son los recomendados para los coeficientes AC de luminancia de la Tabla 4.64. Los códigos EOB y ZRL recomendados para los coeficientes AC de crominancia de la Tabla 4.65 son 00 y 1111111010, respectivamente.

Como ejemplo, consideremos la secuencia:

$$1118, 2, 0, -2, \underbrace{0, \ldots 0}_{13}, -1, 0, \ldots$$

El primer coeficiente AC —2— no tiene ceros que le precedan, por lo que Z = 0. Está ubicado en la Tabla 4.63 en la fila 2, columna 2, por lo que R = 2 y C = 2. El código de Huffman en la posición (R, Z) = (2, 0) de la Tabla 4.66 es 01, por lo que el código final emitido para 2 es 01|10. El siguiente coeficiente distinto de cero, −2, tiene un cero que le precede, por lo que Z = 1. Se encuentra en la Tabla 4.63 en la fila 2, columna 1, por lo que R = 2 y C = 1. El código de Huffman en la posición (R, Z) = (2, 1) de la Tabla 4.66 es 11011, por lo que el código final emitido para 2 es 11011|01.

◇ **Ejercicio 4.18 (sol. en pág. 1082):** ¿Qué código se emite para el último coeficiente AC distinto de cero, −1?

Finalmente, la secuencia de ceros final se codifica como 1010 (EOB), por lo que la salida para la secuencia anterior de coeficientes AC es 011011011101110101010101010. Ya hemos visto que el coeficiente DC se codifica como 111111111110|1110100010, por lo que el resultado final para la unidad de datos completa de 64 píxeles es el número 46 bits:

$$1111111111100111010001001101101110111010101010.$$

Estos 46 bits codifican un componente de color de una unidad de datos de 64 píxeles. Suponiendo que los otros dos componentes de color también están codificados en números de 46 bits. Si cada píxel originalmente consta de 24 bits, entonces ésto corresponde a un factor de compresión de $64\times24/(46\times3) = 11{,}13$; ¡muy impresionante!

(Observe que el coeficiente DC de 1118 ha contribuido con 23 de los 46 bits. Posteriores unidades de datos codifican las diferencias de su coeficiente DC, lo que se podrán reducir a menos de 10 bits en vez de a 23. Se pueden obtener factores de compresión mucho más elevados como consecuencia de ello.)

Z	R				
	1 6	2 7	3 8	4 9	5 A
0	00 1111000	01 11111000	100 1111110110	1011 1111111110000010	11010 1111111110000011
1	1100 1111111110000100	11011 1111111110000101	11110001 1111111110000110	111110110 1111111110000111	11111110110 1111111110001000
2	11100 111111110001010	11111001 111111110001011	1111110111 111111110001100	111111110100 111111110001101	111111110001001 111111110001110
3	111010 1111111110010001	111110111 1111111110010010	111111110101 1111111110010011	1111111110001111 1111111110010100	1111111110010000 1111111110010101
4	111011 1111111110011001	1111111000 1111111110011010	1111111110010110 1111111110011011	1111111110010111 1111111110011100	1111111110011000 1111111110011101
5	1111010 1111111110100001	11111110111 1111111110100010	1111111110011110 1111111110100011	1111111110011111 1111111110100100	1111111110100000 1111111110100101
6	1111011 1111111110101001	111111110110 1111111110101010	1111111110100110 1111111110101011	1111111110100111 1111111110101100	1111111110101000 1111111110101101
7	11111010 1111111110110001	111111110111 1111111110110010	1111111110101110 1111111110110011	1111111110101111 1111111110110100	1111111110110000 1111111110110101
8	111111000 1111111110111001	111111111000000 1111111110111010	1111111110110110 1111111110111011	1111111110110111 1111111110111100	1111111110111000 1111111110111101
9	111111001 1111111111000010	1111111110111110 1111111111000011	1111111110111111 1111111111000100	1111111111000000 1111111111000101	1111111111000001 1111111111000110
A	111111010 1111111111001011	1111111111000111 1111111111001100	1111111111001000 1111111111001101	1111111111001001 1111111111001110	1111111111001010 1111111111001111
B	1111111001 1111111111010100	1111111111010000 1111111111010101	1111111111010001 1111111111010110	1111111111010010 1111111111010111	1111111111010011 1111111111011000
C	1111111010 1111111111011101	1111111111011001 1111111111011110	1111111111011010 1111111111011111	1111111111011011 1111111111100000	1111111111011100 1111111111100001
D	11111111000 1111111111100110	1111111111100010 1111111111100111	1111111111100011 1111111111101000	1111111111100100 1111111111101001	1111111111100101 1111111111101010
E	1111111111101011 1111111111110000	1111111111101100 1111111111110001	1111111111101101 1111111111110010	1111111111101110 1111111111110011	1111111111101111 1111111111110100
F	11111111001 1111111111111001	1111111111110101 1111111111111010	1111111111110110 1111111111111011	1111111111110111 1111111111111101	1111111111111000 1111111111001111

Tabla 4.64: Códigos de Huffman recomendados para los coeficientes AC de luminancia.

Z	R				
	1 6	2 7	3 8	4 9	5 A
0	01 111000	100 1111000	1010 111110100	11000 1111110110	11001 111111110100
1	1011 111111110101	111001 111111110001000	11110110 111111110001001	111110101 111111110001010	11111110110 111111110001011
2	11010 1111111110001100	11110111 1111111110001101	1111110111 1111111110001110	111111110110 1111111110001111	111111111000010 1111111110010000
3	11011 1111111110010010	11111000 1111111110010011	1111111000 1111111110010100	111111110111 1111111110010101	1111111110010001 1111111110010110
4	111010 1111111110011010	111110110 1111111110011011	1111111110010111 1111111110011100	1111111110011000 1111111110011101	1111111110011001 1111111110011110
5	111011 1111111110100010	1111111001 1111111110100011	1111111110011111 1111111110100100	1111111110100000 1111111110100101	1111111110100001 1111111110100110
6	1111001 1111111110101010	11111110111 1111111110101011	1111111110100111 1111111110101100	1111111110101000 1111111110101101	1111111110101001 1111111110101110
7	1111010 1111111110110010	11111111000 1111111110110011	1111111110101111 1111111110110100	1111111110110000 1111111110110101	1111111110110001 1111111110110110
8	11111001 1111111110111011	1111111110110111 1111111110111100	1111111110111000 1111111110111101	1111111110111001 1111111110111110	1111111110111010 1111111110111111
9	111110111 1111111111000100	1111111111000000 1111111111000101	1111111111000001 1111111111000110	1111111111000010 1111111111000111	1111111111000011 1111111111001000
A	111111000 1111111111001101	1111111111001001 1111111111001110	1111111111001010 1111111111001111	1111111111001011 1111111111010000	1111111111001100 1111111111010001
B	111111001 1111111111010110	1111111111010010 1111111111010111	1111111111010011 1111111111011000	1111111111010100 1111111111011001	1111111111010101 1111111111011010
C	111111010 1111111111011111	1111111111011011 1111111111100000	1111111111011100 1111111111100001	1111111111011101 1111111111100010	1111111111011110 1111111111100011
D	11111111001 1111111111101000	1111111111100100 1111111111101001	1111111111100101 1111111111101010	1111111111100110 1111111111101011	1111111111100111 1111111111101100
E	11111111100000 1111111111110001	1111111111101101 1111111111110010	1111111111101110 1111111111110011	1111111111101111 1111111111110100	1111111111110000 1111111111110101
F	111111111000011 1111111111111010	111111111010110 1111111111111011	1111111111110111 1111111111111100	1111111111111000 1111111111111101	1111111111111001 1111111111111110

Tabla 4.65: Códigos de Huffman recomendados para los coeficientes AC de crominancia.

R	Z:	0	1	...	15
0:		1010			11111111001(ZRL)
1:		00	1100	...	1111111111110101
2:		01	11011	...	1111111111110110
3:		100	1111001	...	1111111111110111
4:		1011	111110110	...	1111111111111000
5:		11010	11111110110	...	1111111111111001
⋮	⋮				

Tabla 4.66: Codificación de los coeficientes AC.

Las mismas tablas (Tablas 4.63 y 4.66) utilizadas por el codificador deben, por supuesto, ser usadas por el decodificador. Las tablas pueden estar predefinidas y ser utilizadas por defecto, por un codificador JPEG; o pueden calcularse especificamente para una imagen determinada en una pasada especial que precede a la compresión real. El estándar JPEG no especifica tabla de código alguna, por lo que cualquier codificador JPEG debe utilizar una propia.

Los lectores que consideren que este esquema de codificación es complejo deberían echar un vistazo al mucho más complejo método de codificación CAVLC que es empleado por H.264 (Sección 6.8) para codificar una secuencia similar de coeficientes de la transformada DCT de 8×8.

Algunas variantes JPEG utilizan una versión particular de la codificación aritmética, llamada codificador QM (Sección 2.16), que se especifica en el estándar JPEG. Esta versión de codificación aritmética es adaptativa, por lo que no necesita las Tablas 4.63 y 4.66. Adapta su comportamiento a la estadísticas de la imagen a medida que avanza. Utilizando la codificación aritmética puede producir un 5–10 % mejor compresión que Huffman para una imagen típica de tonos continuos. Sin embargo, es más complejo de implementar que la codificación de Huffman, por lo que en la práctica es raro encontrar un codificador JPEG que lo utilice.

4.8.5. Modo sin pérdidas (lossless)

El modo sin pérdidas de JPEG utiliza la diferenciación (Sección 1.3.1) para reducir los valores de los píxeles antes de comprirlos. Esta forma particular de diferenciación se llama *predicción*. Los valores de algunos de los vecinos más próximos de un píxel se restan del píxel para obtener un número pequeño, que luego se comprime aún más utilizando la codificación de Huffman o la aritmética. La Figura 4.67a muestra un píxel X y tres píxeles vecinos A, B, y C. La Figura 4.67b muestra ocho maneras posibles (predicciones) de combinar los valores de los tres vecinos. En el modo sin pérdidas, el usuario puede seleccionar una de estas predicciones, y luego el codificador la utiliza para combinar los tres píxeles vecinos y restar la combinación del valor de X. El resultado es normalmente un número pequeño, el cual es entonces codificado con un código de entropía de forma muy similar a la descrita en la Sección 4.8.4 para el coeficiente DC.

El predictor 0 se usa sólo en el modo jerárquico de JPEG. Los predictores 1, 2 y 3 se llaman unidimensionales. Los predictores 4, 5, 6 y 7 son bidimensionales.

Cabe señalar que el modo sin pérdidas de JPEG nunca ha tenido mucho éxito. Produce factores típicos de compresión de 2, y por tanto es inferior a otros métodos de compresión de imágenes sin pérdidas. Debido a ésto, muchas implementaciones de JPEG ni siquiera implementan este modo. Incluso el modo con pérdidas (base) de JPEG produce resultados de compresión (comparativamente) pobres cuando se le pide a limitar la cantidad de pérdidas a un mínimo. Como resultado, algunas implementaciones de JPEG no permiten ajustes de parámetros que den como resultado una pérdida mínima. La fuerza de JPEG está en su capacidad de generar imágenes de alta compresión que, cuando

	C	B		
	A	X		

(a)

Valor seleccionado	Predicción
0	sin predicción
1	A
2	B
3	C
4	$A + B - C$
5	$A + ((B - C)/2)$
6	$B + ((A - C)/2)$
7	$(A + B)/2$

(b)

Figura 4.67: Predicción de un píxel en el modo sin pérdidas.

son descomprimidas, son indistinguibles de la original. Reconociendo ésto, la ISO ha decidido proponer otro estándar para la compresión sin pérdidas de imágenes de tonos continuos. Esta norma ahora se conoce comúnmente como JPEG-LS y se describe en la Sección 4.9.

4.8.6. El archivo comprimido

Un codificador JPEG genera un archivo comprimido que incluye parámetros, marcadores, y las unidades de datos comprimidos. Los parámetros son: de cuatro bits (éstos siempre vienen en pares), de un byte, o de dos bytes de longitud. Los marcadores sirven para identificar las distintas partes del archivo. Cada uno es de dos bytes de longitud, donde el primer byte es `X'FF'` y el segundo no es ni 0 ni `X'FF'`. Un marcador puede ir precedido por un número de bytes con `X'FF'`. La Tabla 4.69 muestra todos los marcadores de JPEG (los cuatro primeros grupos son marcadores de comienzo de cuadro[10]). Las unidades de datos comprimidas se combinan en MCUs (unidades mínimas codificadas), donde una MCU es, o bien un sola unidad de datos (en el modo no entrelazado), o bien tres unidades de datos de los tres componentes de la imagen (en el modo entrelazado).

La Figura 4.68 muestra las partes principales del archivo JPEG comprimido (aquéllas entre corchetes son opcionales). El archivo comienza con el marcador SOI y termina con el marcador EOI. En medio de estos marcadores, la imagen comprimida se organiza en cuadros (también llamados marcos o más comúnmente *frames*). En el modo jerárquico, hay varios frames, y en los demás modos sólo hay un frame. En cada frame, la información de la imagen está contenida en uno o más scans, pero el frame también contiene una cabecera y unas tablas opcionales (que, a su vez, pueden incluir marcadores). El primer *scan* (escaneo) puede ir seguido por un segmento DNL opcional (definición del número de líneas), que comienza con el marcador DNL y contiene el número de líneas en la imagen que están representadas por el frame. Un scan comienza con tablas opcionales, seguidas por la cabecera del scan, seguida por varios segmentos de codificación de entropía (ECS), que están separados (opcionalmente) por marcadores de reinicio (RST). Cada ECS contiene una o más MCUs, donde una MCU es, como se ha explicado anteriormente, o bien una unidad de datos individual, o bien tres de tales unidades.

4.8.7. JFIF

Se ha mencionado anteriormente que JPEG es un método de compresión, no un formato de archivo de gráficos, por lo que no especifica características de la imagen tales como la relación de aspecto de los píxeles, el espacio de color, o el entrelazado de las filas de mapas de bits. Aquí es donde entra en juego JFIF.

[10]Start-of-frame markers.

Valor	Nombre	Descripción
	No diferencial; codificación de Huffman	
FFC0	SOF_0	DCT base
FFC1	SOF_1	DCT secuencial extendido
FFC2	SOF_2	DCT progresivo
FFC3	SOF_3	Sin pérdidas (secuencial)
	Diferencial; codificación de Huffman	
FFC5	SOF_5	DCT diferencial secuencial
FFC6	SOF_6	DCT diferencial progresivo
FFC7	SOF_7	Diferencial sin pérdidas (secuencial)
	No diferencial; codificación aritmética	
FFC8	JPG	Reservado para extensiones
FFC9	SOF_9	DCT secuencial extendido
FFCA	SOF_{10}	DCT progresivo
FFCB	SOF_{11}	Sin pérdidas (secuencial)
	Diferencial; codificación aritmética	
FFCD	SOF_{13}	DCT diferencial secuencial
FFCE	SOF_{14}	DCT diferencial progresivo
FFCF	SOF_{15}	Diferencial sin pérdidas (secuencial)
	Tabla de especificaciones de Huffman	
FFC4	DHT	Definir la tabla de Huffman
	Especificaciones de codicionamiento de la codificación aritmética	
FFCC	DAC	Definir condición(es) de la codificación aritmética
	Reiniciar la finalización del intervalo	
FFD0 – FFD7	RST_m	Reiniciar contador m usando módulo 8
	Otros marcadores	
FFD8	SOI	Comienzo de la imagen (*Start of image*)
FFD9	EOI	Fin de la imagen (*End of image*)
FFDA	SOS	Comienzo del scan (*Start of scan*)
FFDB	DQT	Definir tabla(s) de cuantificación (*Define quantization table(s)*)
FFDC	DNL	Definir número de líneas (*Define number of lines*)
FFDD	DRI	Definir intervalo de reinicio (*Define restart interval*)
FFDE	DHP	Definir progresión jerárquica (*Define hierarchical progression*)
FFDF	EXP	Expandir componente(s) de referencia
FFE0 – FFEF	APP_n	Reservado para petición de segmentos (*R. for application s.*)
FFF0 – FFFD	JPG_n	Reservado para extensiones JPEG
FFFE	COM	Comentario
	Marcadores reservados	
FF01	TEM	Para uso privado temporal
FF02 – FFBF	RES	Reservado

Tabla 4.69: Marcadores de JPEG.

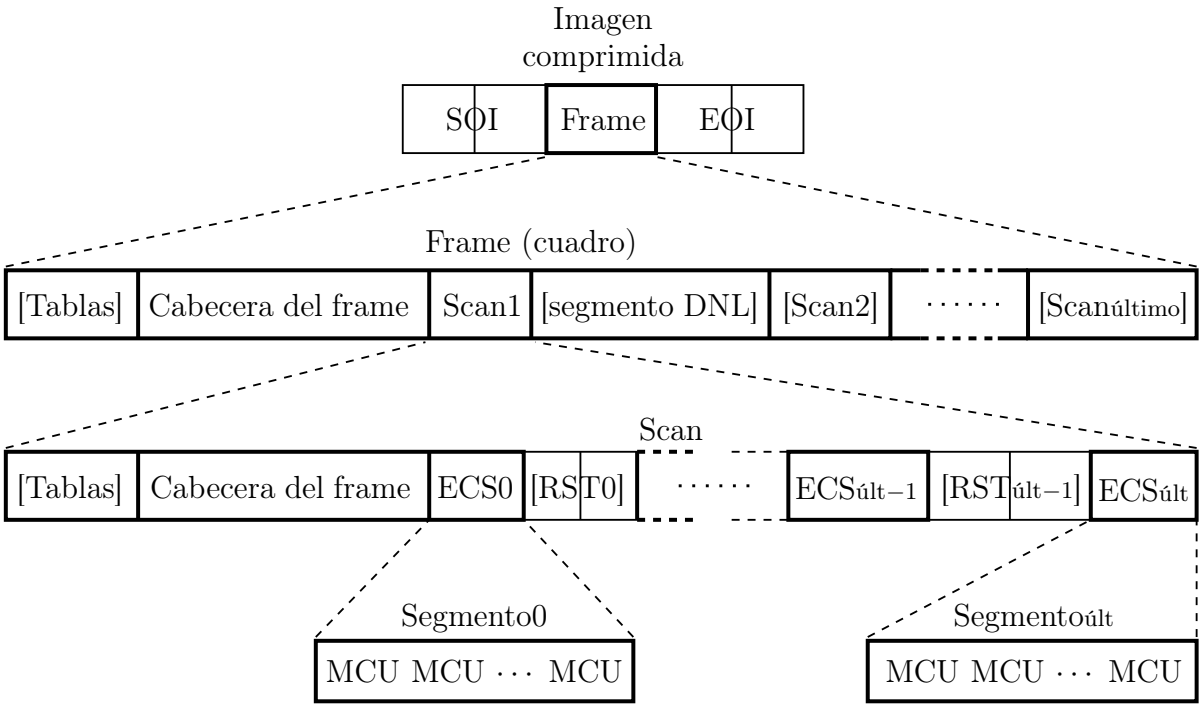


Figura 4.68: Formato del archivo JPEG.

JFIF (*JPEG File Interchange Format* —formato de intercambio de archivos de JPEG—) es un formato de archivo de gráficos que hace posible el intercambio de imágenes JPEG comprimidas entre ordenadores. Las principales características de JFIF son: el uso del espacio de color de componente triple YCbCr para imágenes en color (sólo un componente para escala de grises) y el uso de un *marcador* para especificar las características que faltan en JPEG, tales como resolución de la imagen, la relación de aspecto, y las características que son específicas de la aplicación.

El marcador de JFIF (llamado marcador APP0) comienza con la cadena `JFIF` terminada en cero. Después de ésto, hay información sobre los píxeles y otras especificaciones (véase más adelante). Después de ésto, puede haber segmentos adicionales que especifican extensiones de JFIF. Una extensión de JFIF contiene más información específica de la plataforma acerca de la imagen.

Cada extensión se comienza con la cadena `JFXX` terminada en cero, seguido de un código de 1 byte que identifica la extensión. Una extensión puede contener la información específica de la aplicación, en cuyo caso comienza con una cadena diferente —ni `JFIF`, ni `JFXX`— pero sí algo que identifica la aplicación específica o su fabricante.

El formato del primer segmento de un marcador APP0 es así:

1. El marcador APP0 (4 bytes): `FFD8FFE0`.

2. Longitud (2 bytes): La longitud total del marcador, incluyendo los 2 bytes del campo "longitud" pero excluyendo el propio marcador APP0 (campo 1).

3. Identificador (5 bytes): $\texttt{4A46494600}_{16}$. Ésta es la cadena `JFIF` que identifica el marcador APP0.

4. Versión (2 bytes): Ejemplo: $\texttt{0102}_{16}$ especifica la versión 1.02.

5. Unidades (1 byte): Unidades para las densidades X e Y. 0 significa que no hay unidades; los campos Xdensity e Ydensity especifican la relación de aspecto de los píxeles. 1 significa que Xdensity e Ydensity son puntos por pulgada; 2, que son puntos por cm.
6. Xdensity (2 bytes), Ydensity (2 bytes): La densidad de píxeles horizontales y verticales (ambas deben ser distintas de cero).
7. Xthumbnail (1 byte), Ythumbnail (1 byte): Número de píxeles horizontales y verticales del *thumbnail* (imagen en miniatura).
8. (RGB)n ($3n$ bytes): Valores RGB empacados (24-bit) de los píxeles del thumbnail. n = Xthumbnail× Ythumbnail.

La sintaxis de la extensión JFIF del segmento del marcador APP0 es así:

1. Marcador APP0.
2. Longitud (2 bytes): Longitud total del marcador, incluyendo los 2 bytes del campo "longitud", pero excluyendo el propio marcador APP0 (campo 1).
3. Identificador (5 bytes): $\texttt{4A46585800}_{16}$. Ésta es la cadena `JFXX` que identifica una extensión.
4. Código de extensión (1 byte): $\texttt{10}_{16}$ = Thumbnail codificado con JPEG. $\texttt{11}_{16}$ = Thumbnail codificado con 1 byte/píxel (monocromático). $\texttt{13}_{16}$ = Thumbnail codificado usando 3 bytes/píxel (ocho colores).
5. Los datos de extensión (variable): Este campo depende de la extensión en particular.

> JFIF es el nombre técnico del formato de imagen más conocido (pero incorrectamente) como JPEG. Este término se utiliza sólo cuando la diferencia entre el formato de la imagen, y la compresión de imagen es fundamental. En sentido estricto, sin embargo, JPEG no define un formato de imagen y, por lo tanto, en la mayoría de los casos, sería más preciso hablar de JFIF en lugar de JPEG. Otro formato de imagen para JPEG es SPIFF definido por el propio estándar JPEG, pero JFIF está mucho más extendido que SPIFF.
>
> —Erik Wilde, *WWW Online Glossary*

4.9. JPEG-LS

Como se ha mencionado en la Sección 4.8.5, el modo sin pérdidas de JPEG es ineficiente y con frecuencia ni siquiera está implementado. Como resultado, la ISO, en cooperación con la IEC, ha decidido desarrollar un nuevo estándar para la compresión sin pérdidas (o casi sin pérdidas) de imágenes de tonos continuos. El resultado es la recomendación ISO/IEC CD 14495, conocida popularmente como JPEG-LS. Los principios de este método se describen aquí, pero cabe señalar que no es simplemente una ampliación o una modificación de JPEG. Éste es un método nuevo, diseñado para ser sencillo y rápido. No utiliza la DCT, no emplea codificación aritmética, y utiliza la cuantificación de una manera restringida, y sólo en su opción near-lossless (casi sin pérdidas). JPEG-LS se basa en las ideas desarrolladas en [Weinberger et al. 96 y 00] y en su método de compresión LOCO-I. JPEG-LS examina algunos de los vecinos previamente vistos del píxel actual, los usa como *contexto* del píxel, utiliza el contexto para predecir el píxel y para seleccionar una distribución de probabilidad entre varias distribuciones de este tipo, y utiliza ésa distribución para codificar el error de predicción con

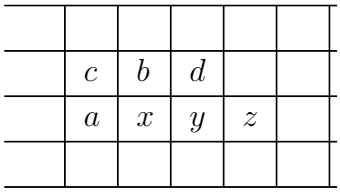


Figura 4.70: Contexto para la predicción de x.

un código de Golomb especial. También hay un modo run, donde se codifica la longitud de una racha (*run*) de píxeles idénticos.

El contexto utilizado para predecir el píxel actual x se muestra en la Figura 4.70. El codificador examina el contexto de los píxeles y decide si codificar el píxel actual x en el *modo run* o en el *modo normal*. Si el contexto sugiere que es probable que los píxeles $y, z, \ldots$ posteriores al píxel actual sean idénticos, el codificador selecciona el modo run. De lo contrario, selecciona el modo normal. En el modo casi sin pérdidas la decisión es ligeramente diferente. Si el contexto sugiere que es probable que los píxeles posteriores al píxel actual sean casi idénticos (con el parámetro de tolerancia NEAR —cercano—), el codificador selecciona el modo run. De lo contrario, selecciona el modo normal. El resto del proceso de codificación depende del modo seleccionado.

En el modo normal, el codificador utiliza los valores de píxeles de contexto a, b, y c para predecir el píxel x, y resta la predicción de x para obtener el *error de predicción*, denotado por *Errval*. Este error se *corrige* luego por un término que depende del contexto (esta corrección se hace para compensar los sesgos sistemáticos en la predicción), y es codificado con un código de Golomb. La codificación de Golomb depende de los cuatro píxels del contexto y también de los errores de predicción que fueron codificados previamente para el mismo contexto (esta información se almacena en los arrays A y N, mencionados en la Sección 4.9.1). Si se utiliza la compresión casi sin pérdidas, el error se cuantifica antes de ser codificado.

En el modo run, el codificador comienza en el píxel actual x y encuentra la racha de píxeles más larga (*run*) que son idénticos al píxel de contexto a. El codificador no extiende este *run* más allá del extremo de la fila actual de la imagen. Puesto que todos los píxeles en el *run* son idénticos a a (y a ya es conocida por el decodificador) sólo necesita codificar la longitud del mismo, y ésto se hace con un array de 32 entradas denotado por J (Sección 4.9.1). Si se utiliza la compresión casi sin pérdidas, el codificador selecciona un *run* de píxeles que esté próximo a a de acuerdo con del parámetro de tolerancia NEAR.

El decodificador no es sustancialmente distinto al codificador, por lo que JPEG-LS es un método de compresión casi simétrico. La cadena comprimida contiene segmentos de datos (con los códigos de Golomb y la longitud de cada *run* codificada), los segmentos de marcadores (con información que necesita el decodificador), y los marcadores (se utilizan algunos de los marcadores reservados de JPEG). Un marcador es un byte con todo a unos, seguido por un código especial, que indica el comienzo de un nuevo segmento. Si un marcador es seguido por un byte cuyo bit más significativo es 0, este byte es el comienzo de un segmento marcador. En caso contrario, dicho byte indica el inicio de un segmento de datos.

4.9.1. El codificador

JPEG-LS se utiliza normalmente como un método de compresión sin pérdidas. En este caso, el valor reconstruido de un píxel es idéntico a su valor original. En el modo casi sin pérdidas (*near lossless*), los valores originales y los reconstruidos pueden ser diferentes. En todos los casos se denota

la reconstrucción del valor del píxel p con Rp.

Cuando la fila superior de una imagen es codificada, los píxeles de contexto b, c, y d no están disponibles y, por lo tanto, deben considerarse ceros. Si el píxel actual está ubicado en el inicio o en el final de una fila de la imagen, ya sea a y c, ya sea d, no están disponibles. En tal caso, el codificador utiliza para a o d el valor reconstruido Rb de b (o cero si se trata de la fila superior); y para c, el valor reconstruido que fue asignado a a cuando el primer píxel de la línea anterior fue codificado. Ésto significa que el codificador tiene que hacer parte del trabajo del decodificador y tiene que averiguar los valores reconstruidos de que ciertos píxeles.

El primer paso para determinar el contexto es calcular los tres valores de *gradientes*:

$$D1 = Rd - Rb, \quad D2 = Rb - Rc, \quad D3 = Rc - Ra.$$

Si los tres valores son cero (o, para *near-lossless*, si sus valores absolutos son menores o iguales que el parámetro de tolerancia NEAR), el codificador selecciona el modo *run*, donde busca la racha más larga de píxeles idénticos a Ra. El paso 2 compara los tres gradientes Di para ciertos parámetros y calcula tres números de la región de Qi de acuerdo con ciertas reglas (no mostradas aquí). Cada número de la región de Qi puede tomar uno de los nueve valores enteros en el intervalo $[-4, +4]$, por lo que hay $9 \times 9 \times 9 = 729$ números de región diferentes. El tercer paso mapea los números de la región de Qi a un entero Q en el intervalo $[0, 364]$. La tupla $(0, 0, 0)$ se asigna a 0 y las 728 tuplas restantes se asignan a $[1, {}^{728}/_{2} = 364]$, de modo que (a, b, c) y $(-a, -b, -c)$ se mapean al mismo valor. Los detalles de este cálculo no son especificados por el estándar JPEG-LS, y el codificador puede hacerlo de cualquier forma que desee. El entero Q se convierte en el contexto del píxel actual x. Se utiliza para el índice de los arrays A y N en la Figura 4.74.

Después de determinar el contexto Q, el codificador predice el píxel x en dos pasos. El primer paso calcula la predicción Px basada en reglas de arista (*edge rules*), como se muestra en la Figura 4.71. El segundo paso corrige la predicción, como se muestra en la Figura 4.72, basada en la cantidad SIGN (determinada por los signos de las tres regiones Qi), los valores de corrección $C[Q]$ (derivados de las tendencias y no se discute aquí), y el parámetro MAXVAL.

```
if(Rc>=max(Ra,Rb)) Px=min(Ra,Rb);
  else
    if(Rc<=min(Ra,Rb)) Px=max(Ra,Rb)
      else Px=Ra+Rb-Rc;
    endif;
endif;
```

Figura 4.71: Detección de bordes.

```
if(SIGN=+1) Px=Px+C[Q]
       else Px=Px-C[Q]
endif;
if(Px>MAXVAL) Px=MAXVAL
  else if(Px<0) Px=0 endif;
endif;
```

Figura 4.72: Predicción de correcciones.

Para comprender las reglas de arista, vamos a considerar el caso en que $b \leq a$. En este caso las reglas de arista seleccionan b como la predicción de x en muchos casos donde existe un borde vertical en la imagen justo a la izquierda del píxel actual x. Similarmente, se selecciona a como predicción en muchos casos donde existe un borde horizontal en la imagen justo por encima de x. Si no se detecta ningún borde, las reglas de arista calculan como predicción $a + b - c$, y ésto tiene una sencilla interpretación geométrica. Si interpretamos cada píxel como un punto en el espacio tridimensional, con la intensidad de píxel considerada como su altura, entonces el valor $a + b - c$ sitúa la predicción Px en el mismo plano que los píxeles a, b, y c.

Una vez que se conoce la predicción Px, el codificador calcula el error de predicción *Errval* como la diferencia $x - Px$, pero invierte su signo si la cantidad SIGN es negativa.

En el modo casi sin pérdidas el error está cuantificado, y el codificador lo utiliza para calcular el valor reconstruido Rx del píxel x de la misma forma en que el decodificador lo hará en el futuro.

El codificador necesita este valor reconstruido para codificar los píxeles en el futuro. El paso de cuantificación básico paso es:

$$Errval \leftarrow \frac{Errval + \text{NEAR}}{2 \times \text{NEAR} + 1}.$$

Utiliza el parámetro NEAR, pero involucra más detalles que no se muestran aquí. El paso de reconstrucción básico es:

$$Rx \leftarrow Px + \text{SIGN} \times Errval \times (2 \times \text{NEAR} + 1).$$

El error de predicción (después de, posiblemente, ser cuantificado) pasa ahora por una reducción de rango (cuyos detalles se omiten aquí) y se encuentra finalmente listo para el importante paso de la codificación.

El código de Golomb fue introducido en la Sección 2.5, donde su parámetro principal fue denotado por b. JPEG-LS denota este parámetro m. Una vez que m ha sido seleccionado, el código de Golomb del entero no negativo n se compone de dos partes: el código unario de la parte entera de n/m, y la representación binaria de $n \bmod m$. Estos códigos son ideales para los enteros n que están distribuidos geométricamente (i.e., cuando la probabilidad de n es $(1-r)\,r^n$, donde $0 < r < 1$). Para cualquier distribución geométrica de este tipo existe un valor m tal que el código de Golomb basado en él obtiene la longitud media de código más corto posible. El caso especial donde m es una potencia de 2 ($m = 2^k$) conduce a sencillas operaciones de codificación/decodificación. El código para n está formado, en tal caso, por los k bits menos significativos de n, precedidos por el código unario de los restantes bits más significativos de n. Este código de Golomb en particular se denota por $G(k)$.

Como ejemplo, podemos calcular el código $G(2)$ de $n = 19 = 10011_2$. Como $k = 2$, m es 4. Empezamos con los dos bits menos significativos —11— de n. Equivalen al entero 3, que es también $n \bmod m$ ($3 = 19 \bmod 4$). Los restantes bits más significativos —100— son también el entero 4, que es la parte entera del cociente n/m ($19/4 = 4{,}75$). El código unario de 4 es 00001, por lo que el código $G(2)$ de $n = 19$ es 00001|11.

En la práctica, siempre tenemos un conjunto finito de números enteros no negativos, donde el mayor entero en el conjunto se denota por I. La longitud máxima de $G(0)$ es $I + 1$, y puesto que puede ser grande, es deseable limitar el tamaño del código de Golomb. Esto se hace mediante el código de Golomb especial $LG(k, glimit)$ que depende de los dos parámetros k y $glimit$. En primer lugar, construimos un número q con los bits más significativos de n. Si $q < glimit - [\log I] - 1$, el código $LG(k, glimit)$ es simplemente $G(k)$. De lo contrario, se prepara el código unario de $glimit - [\log I] - 1$ (i.e., $glimit - [\log I] - 1$ ceros seguidos por un único 1). Esto actúa como un código de escape y es seguido por la representación binaria de $n - 1$ en $[I]$ bits.

Nuestra predicción de errores no es necesariamente positiva. Está formada por diferencias, por lo que puede ser también cero o negativa; pero los distintos códigos Golomb fueron diseñados para enteros no negativos. Ésta es la razón por la que los errores de predicción deben mapearse a valores no negativos antes de poder ser codificados. Esto se realiza usando:

$$M\,Errval = \begin{cases} 2Errval, & Errval \geq 0, \\ 2|Errval| - 1, & Errval < 0. \end{cases} \tag{4.24}$$

Esta asignación intercala valores negativos y positivos en la secuencia:

$$0, -1, +1, -2, +2, -3, \ldots$$

La Tabla 4.73 muestra algunos errores de predicción, sus valores asignados (mapeo), y sus códigos $LG(2, 32)$ asumiendo un alfabeto de tamaño 256 (i.e., $I = 255$ y $[\log I] = 8$).

El siguiente punto a tratar es cómo determinar el valor del parámetro k del código de Golomb. Ésto se hace de forma adaptativa. El parámetro k depende del contexto, y el valor de k para un contexto

Error de predicción	Valor mapeado	Código
0	0	1 00
-1	1	1 01
1	2	1 10
-2	3	1 11
2	4	01 00
-3	5	01 01
3	6	01 10
-4	7	01 11
4	8	001 00
-5	9	001 01
5	10	001 10
-6	11	001 11
6	12	0001 00
-7	13	0001 01
7	14	0001 10
-8	15	0001 11
8	16	00001 00
-9	17	00001 01
9	18	00001 10
-10	19	00001 11
10	20	000001 00
-11	21	000001 01
11	22	000001 10
-12	23	000001 11
12	24	0000001 00
...		
50	100	000000000000
		000000000001
		01100011

Tabla 4.73: Predicción de errores; sus mapeos, y códigos $LG\,(2,\,32)$.

```
B[Q]=B[Q]+Errval*(2*NEAR+1);
A[Q]=A[Q]+abs(Errval);
if(N[Q]=RESET) then
  A[Q]=A[Q]>>1; B[Q]=B[Q]>>1; N[Q]=N[Q]>>1
endif;
N[Q]=N[Q]+1;
```

Figura 4.74: Actualización de los arrays A, B y N.

```
RUNval=Ra;
RUNcnt=0;
while(abs(Ix-RUNval)<=NEAR)
  RUNcnt=RUNcnt+1;
  Rx=RUNval;
  if(EOLine=1) break
    else GetNextSample()
  endif;
endwhile;
```

Figura 4.75: Escaneo del *run*.

se actualiza cada vez que se encuentra un píxel con ese contexto. El cálculo de k puede ser expresado por la sentencia única en lenguaje C:

$$\texttt{for}\,(\texttt{k}=0;\ (\texttt{N}\,[\texttt{Q}]<<\texttt{k})<\texttt{A}\,[\texttt{Q}]\,;\ \texttt{k}++)\,;$$

donde A y N son arrays indexados desde 0 hasta 364. Esta instrucción utiliza el contexto Q como un índice para los dos arreglos. Inicializa k a 0 y entra en un bucle. En cada iteración se desplaza el elemento del array $N\,[Q]$, k posiciones a la izquierda y lo compara con el elemento $A\,[Q]$. Si el valor desplazado de $N\,[Q]$ es mayor o igual que $A\,[Q]$, se elige el valor actual de k. De lo contrario, k se incrementa en 1 y se procede con la siguiente iteración del bucle.

Después de determinar k, el error de predicción *Errval* es mapeado, por medio de la Ecuación (4.24), a $M\ Errval$, el cual se codifica utilizando el código $LG\,(k, LIMIT)$. La cantidad LIMIT es un parámetro. Los arreglos A y N (junto con el array auxiliar B) se actualizan después, como muestra la Figura 4.74 (RESET es un parámetro controlado por el usuario).

La codificación en el modo *run* se realiza de manera diferente. El codificador selecciona este modo cuando encuentra x píxeles consecutivos cuyos valores Ix son idénticos e iguales al valor reconstruido Ra del píxel de contexto a. Para la compresión casi sin pérdidas, los píxeles del *run*[11] deben tener valores Ix que satisfagan:

$$|Ix - Ra| \leq \text{NEAR}.$$

Un *run* no puede continuar más allá del final de la fila de la imagen actual. La longitud del *run* está codificado (no hay necesidad de codificar el valor de los píxeles del *run*, ya que es igual a Ra), y si el *run* termina antes del final de la fila actual, su longitud codificada es seguida por la codificación del píxel inmediatamente posterior (el píxel *de interrupción* del *run*). Las dos tareas principales del codificador en este modo son: (1) escaneo del *run* y codificación *run-length* y (2) código de interrupción del *run*. La exploración del *run* se muestra en la Figura 4.75. La codificación *run-length* se muestra en las Figuras 4.76 (para segmentos *run* de longitud rm) y 4.77 (para los segmentos de longitud menor que rm). Estos son algunos de los detalles.

```
while(RUNcnt>=(1<<J[RUNindex]))
  AppendToBitStream(1,1);
  RUNcnt=RUNcnt-(1<<J[RUNindex]);
  if(RUNindex<31)
    RUNindex=RUNindex+1;
endwhile;
```

Figura 4.76: Codificación run-length I.

El codificador utiliza una tabla J de 32 de entrada que contiene valores denotados por rk. J es inicializada a los 32 valores:

$$0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 9, 10, 11, 12, 13, 14, 15.$$

[11] Recuérdese que estamos llamando *run* a una secuencia de ítems idénticos.

Para cada valor de rk, se utiliza la notación $rm = 2^{rk}$. Las 32 cantidades rm se llaman *código de orden*. Los cuatro primeros rms tienen como valor $2^0 = 1$. Los cuatro siguientes tienen valor $2^1 = 2$. Los cuatro siguientes, $2^2 = 4$, y así hasta el último rm, cuyo valor es $2^{15} = 32768$. El codificador ejecuta el procedimiento de la Figura 4.75 para determinar la longitud del *run*, que almacena en la variable `RUNlen`. Esta variable se codifica dividiendolo en fragmentos cuyos tamaños son los valores de consecutivos rms. Por ejemplo, si `RUNlen` es 6, se puede expresar en términos de rms como $1 + 1 + 1 + 1 + 2$, por lo que es equivalente a los cinco primeros rms. Se codifica escribiendo cinco bits a 1 en el *stream* comprimido. Cada uno de ésos bits se escribe mediante la sentencia `AppendToBitStream(1,1)` de la Figura 4.76. Cada vez que se escribe un 1, el valor del correspondiente rm se resta de `RUNlen`. Si `RUNlen` es originalmente 6, baja a 5, 4, 3, 2, y 0.

```
if(EOLine=0) then
  AppendToBitStream(0,1);
  AppendToBitStream(RUNcnt,J[RUNindex]);
  if(RUNindex>0)
    RUNindex=RUNindex-1;
  endif;
else if(RUNcnt>0)
  AppendToBitStream(1,1);
```

Figura 4.77: Codificación run-length II.

Puede ocurrir, por supuesto, que la longitud `RUNlen` de un *run* no sea igual a un número entero de rms. Un ejemplo es un `RUNlen` de 7. Esto se codifica escribiendo cinco bits a 1, seguidos por un bit *prefijo*, seguido por el resto hasta `RUNlen` (en nuestro ejemplo, un 1), escrito en el *stream* comprimido como un número de rk bits (el rk actual, en nuestro ejemplo es 1). Esta última operación se lleva a cabo con la llamada al procedimiento `Append-ToBitStream(RUNcnt,J[RUNindex])` de la Figura 4.77. El bit prefijo es 0, si el *run* es interrumpido por un píxel diferente. Es 1, si el *run* termina con el final de una fila de la imagen.

La segunda tarea principal del codificador —la codificación del píxel de interrupción—, es similar a la codificación del píxel actual y no se discute aquí.

4.10. Compresión progresiva de la imagen

Los métodos de compresión de imágenes más modernos, o bien son progresivos, o bien tienen la opción de serlo. La compresión progresiva es una opción atractiva cuando las imágenes comprimidas se transmiten través de una línea de comunicación y se descomprimen y ven en tiempo real. Cuando se reciben y se descomprimen tales imágenes, el decodificador puede mostrar muy rápidamente la imagen entera en un formato de baja calidad, y mejorar la calidad en la pantalla a medida que se recibe y descomprime más y más información de la imagen. Un usuario que observa la imagen en desarrollo en la pantalla puede reconocer normalmente la mayor parte de las características de la imagen después de haberse descomprimido sólo un 5−10 % de la misma.

Ésto debe compararse con la compresión de imágenes *raster-scan* (de barrido recurrente). Cuando una imagen es escaneada y comprimida, un usuario normal no puede decir mucho acerca de la imagen cuando sólo ha descomprimido y mostrado el 5−10 % de la misma. Se supone que las imágenes van a ser vistas por seres humanos, por lo que la compresión progresiva tiene sentido incluso en aquellos casos en que sea más lenta o menos eficiente que la no progresiva.

Quizás una buena manera de pensar en la compresión progresiva de imágenes es imaginar que el codificador comprime la información más importante de la imagen en primer lugar, y luego comprime la información menos importante y la añade al flujo de datos comprimidos; y así sucesivamente. Ésto explica por qué todos los métodos de compresión progresiva de imágenes tienen una opción intuitiva de compresión con pérdida; simplemente deja de comprimir en un punto determinado. El usuario puede controlar la cantidad de pérdidas por medio de un parámetro que indique al codificador cuándo detener el proceso de codificación progresiva. Cuanto antes se detenga la codificación, mejor será la razón de compresión y mayor será la pérdida de datos.

Otra ventaja de la compresión progresiva se hace evidente cuando el archivo comprimido debe ser descomprimido en varias ocasiones y se muestra con diferentes resoluciones. El decodificador puede, en cada caso, detener la descompresión cuando la imagen ha alcanzado el resolución del dispositivo de salida particular utilizado.

La compresión progresiva de imágenes ya se ha mencionado, relacionada con JPEG (página 4.8). JPEG utiliza la DCT para desglosar la imagen en sus componentes de frecuencia espacial, y comprime los componentes de baja frecuencia en primer lugar. El decodificador puede, por lo tanto, mostrar estas partes rápidamente, y son dichas partes de baja frecuencia las que contienen la información principal de la imagen. Las partes de alta frecuencia, contienen los detalles de la imagen. Por consiguiente, JPEG codifica los datos de frecuencia espacial progresivamente.

Es útil pensar en la decodificación progresiva como el proceso de mejora de las características de la imagen con el paso del tiempo, y esto se puede lograr de tres maneras:

1. Codificando progresivamente los datos de frecuencias espaciales. Un espectador que observa como está siendo decodificada una imagen ve la imagen cambiando de borrosa a nítida. Los métodos que funcionan de esta manera se caracterizan típicamente por una velocidad de codificación media y una lenta decodificación. Este tipo de compresión progresiva a veces se llama *SNR progresiva o la calidad progresiva*.

2. Comenzando con una imagen gris y añadir colores o tonos de gris a la misma. Un espectador que observa como está siendo decodificada una imagen que podrá ver todos los detalles de la imagen desde el principio, y la verá mejorar a medida que se añaden más y más colores a la misma. Los métodos de cuantificación vectorial (Sección 4.14) utilizan este tipo de compresión progresiva. Dicho método se caracteriza normalmente por una codificación lenta y una rápida decodificación.

3. Codificando la imagen en capas, donde las primeras capas están formadas por unos pocos píxeles de baja resolución, grandes; seguidos por capas posteriores de píxeles de mayor resolución, más pequeños. Una persona que observa tales imágenes según se están decodificando verá más detalles añadidos a la imagen con el tiempo. Tal método por lo tanto, incrementa el detalle (o resolución) de la imagen a medida que se descomprime. Esta forma de codificación progresiva de una imagen se denomina *codificación piramidal* o *codificación jerárquica*. La mayor parte de los métodos progresivos utilizan este principio, por lo que esta sección trata las ideas generales para la implementación de la codificación piramidal. La Figura 4.79 ilustra los tres métodos progresivos aquí mencionados. Debería contrastarse con la Figura 4.78, que ilustra una decodificación secuencial.

Suponiendo que el tamaño de la imagen es de $2^n \times 2^n = 4^n$ píxeles, el método más sencillo que viene a la mente, cuando se intenta aplicar la compresión progresiva, es el cálculo de cada píxel de la capa $i-1$ como el promedio de un grupo de 2×2 píxeles de la capa i. Así, la capa n es la imagen completa, la capa $n-1$ contiene $2^{n-1} \times 2^{n-1} = 4^{n-1}$ grandes píxeles de tamaño 2×2, y así sucesivamente, hasta bajar a la capa 1, formada por $4^{n-n} = 1$ píxel grande, que representa la imagen completa. Si la imagen no es demasiado grande, todas las capas se pueden guardar en la memoria. Los píxeles se escriben en el *stream* comprimido en orden inverso, comenzando por la capa 1. El único píxel de la capa 1 es el "padre" de los cuatro píxeles de la capa 2; cada uno de ellos, es el padre de cuatro píxeles de la capa 3, y así sucesivamente. El número total de píxeles en la pirámide es ¡un 33 % más que el número original!

$$4^0 + 4^1 + \cdots + 4^{n-1} + 4^n = (4^{n+1}-1)/3 \approx 4^n\,(4/3) \approx 1{,}33 \times 4^n = 1{,}33\,(2^n \times 2^n)\,.$$

Una forma sencilla para que el número total de píxeles bajando por la pirámide sea de 4^n es incluir sólo tres de los cuatro píxeles de un grupo en la capa i, y para calcular el valor del cuarto píxel con los padres del grupo (a partir de la capa anterior, $i-1$) y sus tres hermanos.

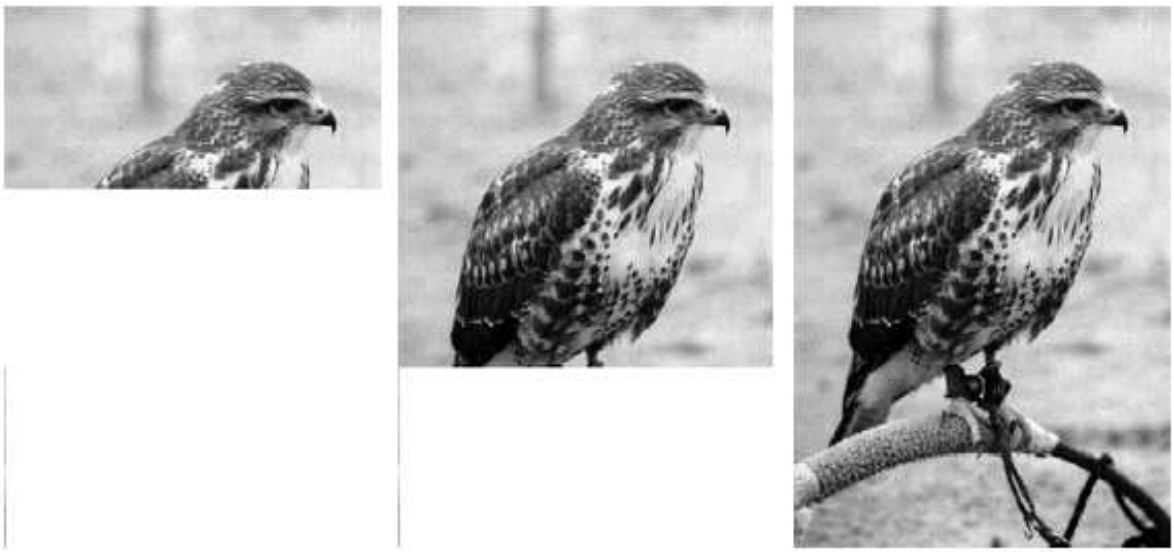


Figura 4.78: Decodificación secuencial.

Ejemplo: La Figura 4.80c muestra una imagen 4×4 que se convierte en la tercera capa en su compresión progresiva. La capa dos se muestra en la Figura 4.80b, donde, por ejemplo, el píxel 81,25 es la media de los cuatro píxeles de la capa 3: 90, 72, 140, y 23. El píxel de una capa se muestra en la Figura 4.80a. El archivo comprimido debe contener sólo los números:

$$54{,}125,\ 32{,}5,\ 41{,}5,\ 61{,}25,\ 72,\ 23,\ 140,\ 33,\ 18,\ 21,\ 18,\ 32,\ 44,\ 70,\ 59,\ 16$$

(debidamente codificados, por supuesto), de los que todos los valores de los píxeles que faltan pueden ser determinados fácilmente. El píxel ausente 81,25, por ejemplo, se puede calcular a partir de $(x + 32, 5 + 41, 5 + 61, 25)/4 = 54{,}125$.

Una pequeña complicación con este método es que los promedios de los números enteros pueden no ser enteros. Si queremos que los valores de nuestros píxeles permanezcan enteros, o bien tienen que perder precisión, o bien usar números enteros cada vez más largos. Suponiendo que los píxeles están representados por ocho bits, la adición de cuatro enteros de 8 bits produce un número entero de 10 bits. La división por cuatro, para crear la media, reduce la suma de nuevo a un número entero de 8 bits, pero con cierta pérdida de precisión. Si no queremos perder precisión, debemos representar nuestros píxeles de la segunda capa con números de 10 bits y el de nuestra primera capa (único) como un número de 12 bits. La Figura 4.80d,e,f muestra los resultados del redondeo de los valores de nuestros píxeles y la consiguiente pérdida de alguna información de la imagen. El contenido del archivo comprimido en este caso debe ser:

$$54, 33, 42, 61, 72, 23, 140, 33, 18, 21, 18, 32, 44, 70, 59, 16.$$

El primer píxel perdido de la capa tres, 81, se puede determinar a partir de la ecuación $^{(x+33+42+61)}/_4 = 54$, de la que se obtiene el valor (algo erróneo) de 80.

◇ **Ejercicio 4.19 (sol. en pág. 1082):** Demuéstrese que la suma de cuatro números de n bits es un número de $(n + 2)$ bits.

Un método mejor es dejar que el padre de un grupo ayude en el cálculo de los valores de su cuatro hijos. Ésto puede hacerse mediante el cálculo de las diferencias entre el padre y sus hijos, y escribiendo

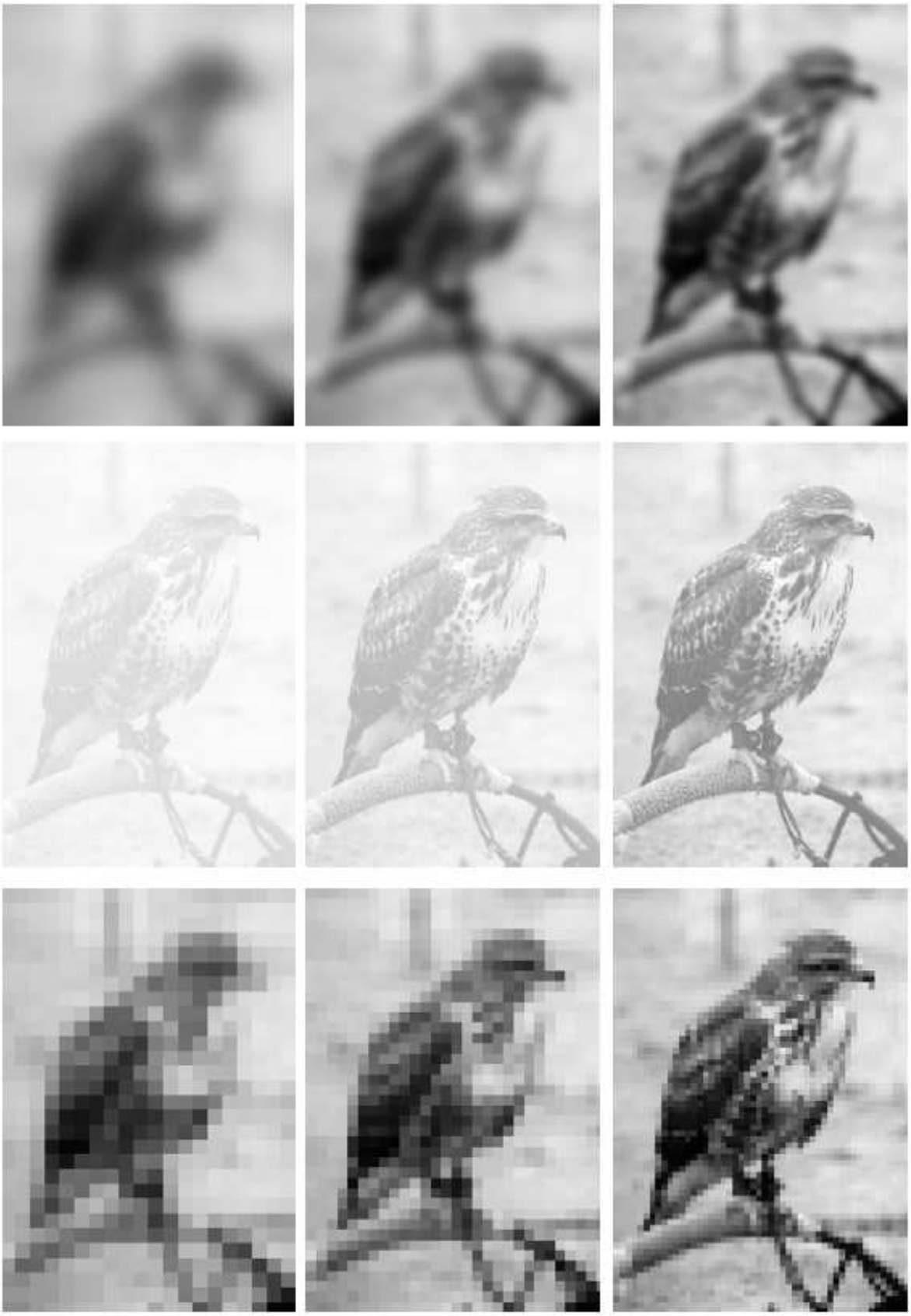


Figura 4.79: Decodificación progresiva.

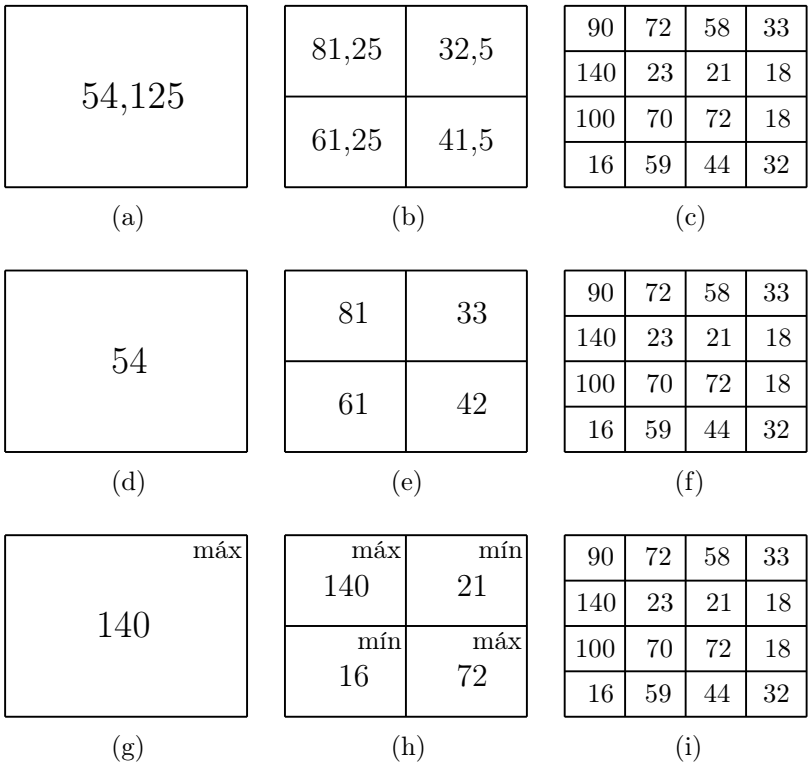


Figura 4.80: Compresión progresiva de imágenes.

las diferencias (convenientemente codificadas) en la capa i del *stream* comprimido. El decodificador decodifica las diferencias; después, utiliza el padre de la capa $i-1$ para calcular los valores de los cuatro pixeles. Para codificar las diferencias, se puede utilizar tanto la codificación de Huffman como la aritmética. Si se calculan todas las capas y se guardan en la memoria, entonces puede crearse la distribución de los valores de diferencias y utilizarse para lograr la mejor compresión estadística.

Si no hay espacio en la memoria para todas las capas, se puede implementar un sencillo modelo adaptativo. Comienza con la asignación de un contador a 1 para cada valor de la diferencia (para evitar el problema de la probabilidad cero). Cuando se calcula una diferencia en particular, se le asigna una probabilidad y se codifica de acuerdo con su contador, actualizando éste a continuación. Es una buena idea actualizar los contadores incrementandolos en un valor mayor que 1, ya de esta manera los valores originales de los contadores se hacen insignificantes, muy rápidamente.

Se puede conseguir alguna mejora si se emplean los padres para ayudar a calcular los valores de los tres píxeles hijo; y luego se utilizan estos tres y el padre, para calcular el valor del cuarto píxel del grupo. Si los cuatro píxeles de un grupo son a, b, c, y d, entonces su media es $v = (a+b+c+d)/4$. El promedio se convierte en parte de la capa $i-1$, y la capa i necesita contener sólo las tres diferencias $k = a-b$, $l = b-c$, y $m = c-d$. Una vez que el decodificador ha leído y decodificado las tres diferencias, puede utilizar sus valores —junto con el valor de v de la capa anterior—, para calcular los valores de los cuatro píxeles del grupo. El cálculo de v mediante una división por 4 aún causa la pérdida de dos bits, pero esta cantidad de 2 bits puede ser aislada antes de la división, y retenida codificándola por separado, a continuación de las tres diferencias.

Las mejoras mencionadas anteriormente se basan en el hecho bien conocido de que los enteros

pequeños son fáciles de comprimir (página 41).

El píxel padre de un grupo no tiene por qué ser su promedio. Una alternativa es seleccionar como padre el valor máximo (o el mínimo) de píxel de un grupo. Ésto tiene la ventaja de que el padre es idéntico a uno de los píxeles del grupo. El codificador tiene que codificar sólo tres píxeles en cada grupo, y el decodificador decodifica tres píxeles (o diferencias) y utiliza el padre como cuarto píxel, para completar el grupo. Cuando se codifican grupos consecutivos en una capa, el codificador debe alternar, en la elección como padres, entre el máximo y el mínimo, ya que seleccionar siempre el mismo crea una acumulación de capas que las hace demasiado oscuras o demasiado brillantes. La Figura 4.80g,h,i muestra las tres capas en este caso.

El archivo comprimido debe contener los números:

$$140, (0), 21, 72, 16, (3), 90, 72, 23, (3), 58, 33, 18, (0), 18, 32, 44, (3), 100, 70, 59,$$

donde los números entre paréntesis son, cada uno, de dos bits. Éstos le dicen dónde (en qué cuadrante) debe ir el padre de la capa anterior. Observe que la numeración del cuadrante es $\binom{0\,1}{3\,2}$.

La selección de la mediana de un grupo es un poco más lenta que la selección del máximo o el mínimo, pero mejora la apariencia de las capas durante la descompresión progresiva. En general, la mediana de una secuencia $(a_1, a_2, \ldots, a_n)$ es un elemento a_i tal que la mitad de los elementos (o muy cerca de la mitad) son más pequeños que a_i, y la otra mitad son mayores. Si los cuatro píxeles de un grupo satisfacen $a < b < c < d$, entonces b o c puede considerarse el píxel mediana del grupo. La principal ventaja de la selección de la mediana como el padre del grupo es que tiende a suavizar las diferencias grandes en los valores de píxel que pueden producirse debido a un píxel extremo. En el grupo 1, 2, 3, 100, por ejemplo, la elección de 2 ó 3 como padre es mucho más representativa que la selección del promedio. Encontrar la mediana de cuatro píxeles requiere algunas comparaciones, pero el cálculo del promedio requiere una división por 4 (o, alternativamente, un desplazamiento a la derecha).

Una vez que la mediana ha sido seleccionada y codificada como parte de la capa $i-1$, los tres píxeles restantes pueden ser codificados en la capa i mediante la codificación de sus (tres) diferencias, precedidas por un código de 2 bits que indica cuál de los cuatro es el padre. Otra pequeña ventaja de utilizar la mediana es que una vez que el decodificador lee el código de 2 bits, sabe cuántos de los tres píxeles son más pequeños y cuántos son más grandes que la media. Si el código dice, por ejemplo, que un píxel es menor, y los otros dos son mayores que la media, y el decodificador lee un píxel que es más pequeño que la media, se sabe que los próximos dos píxeles decodificados serán más grandes que la media. Este conocimiento cambia la distribución de la diferencias, y puede ser aprovechado mediante el uso de tablas de contadores para estimar probabilidades cuando las diferencias son codificadas. Una tabla se utiliza cuando un píxel se codifica para que el decodificador sepa que es mayor que la mediana. Otra tabla se utiliza para codificar los píxeles que el decodificador considerará que son más pequeños que la mediana, y la tercera tabla se utiliza para los píxeles cuya relación con la mediana, es desconocida de antemano por el decodificador. Ésto mejora la compresión en un pequeño tanto por ciento y es otro ejemplo de cómo la adición de más características a un método de compresión disminuye el rendimiento.

Algunos de los métodos de compresión progresiva de imágenes más importantes utilizados en la práctica son descritos en el resto de este capítulo.

4.10.1. Codificación de la geometría de crecimiento

La idea de la codificación de la geometría de crecimiento merece su propia sección, ya que combina la compresión de imágenes y la transmisión progresiva de imágenes de una manera original e inusual. Esta idea es debida a Amalie J. Frank [Frank et al. 80]. El método está diseñado para la compresión progresiva sin pérdidas de imágenes binivel. La idea es empezar con algunos píxeles *semilla* y aplicar

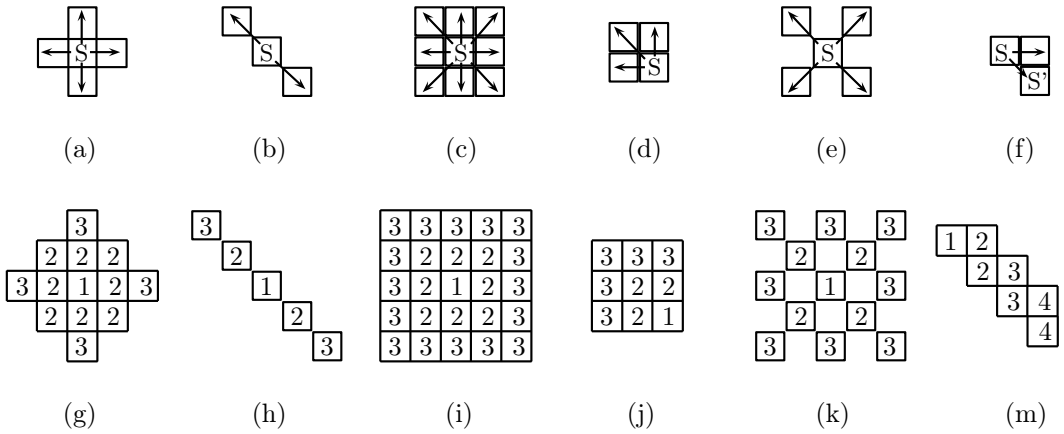


Figura 4.81: Seis reglas y patrones de crecimiento.

las reglas geométricas de crecimiento a cada píxel semilla en un patrón de píxeles. El codificador carga con la parte más difícil del trabajo. Tiene que seleccionar los píxeles semilla, la regla de crecimiento para cada semilla, y el número de veces que la regla debe aplicarse (el número de generaciones). En el *stream* comprimido, sólo se escriben estos datos. A menudo, un grupo de semillas comparte la misma regla y el mismo número de generaciones. En tal caso, la regla y el número de generaciones se escriben una vez, seguidos de las coordenadas de los píxeles semilla del grupo. El trabajo del decodificador es sencillo. Lee el primer grupo de semillas, aplica la regla una vez, lee el siguiente grupo y lo agrega al patrón actual, aplica la regla de nuevo, y así sucesivamente. La compresión se logra si el número de semillas es pequeño en comparación con el tamaño total de la imagen, i.e., si cada píxel semilla se utiliza para generar, en promedio, muchos píxeles de la imagen.

La Figura 4.81a–f muestra seis sencillas reglas de crecimiento. Cada una añade algunos píxeles vecinos adyacentes a la semilla S, y aplica la misma regla a estos vecinos en la siguiente generación (se convierten en píxeles semilla *secundarios*). Puesto que un píxel puede tener hasta ocho vecinos inmediatos, puede haber 256 de tales reglas. La Figura 4.81g–m muestra los resultados de la aplicación estas reglas dos veces (i.e., durante dos generaciones) a un solo píxel semilla. Los píxeles se numeran, uno más que el número de la generación. Las normas de crecimiento pueden ser tan complejas como sea necesario. Por ejemplo, la Figura 4.81m asume que sólo el píxel S' se convierte en una semilla secundaria. El patrón de píxeles resultante puede ser sólido o puede tener "huecos" en él, como en la Figura 4.81k.

La Figura 4.82 muestra un ejemplo de cómo se utiliza una sencilla regla de crecimiento (indicada en la Figura 4.82a) para producir el patrón inconexo de la Figura 4.82b. Los píxeles semilla se muestran en la Figura 4.82c. Están numerados 1−4 (uno más que el número de generaciones). El decodificador primero introduce los seis píxeles marcados como 4 (Figura 4.82d). Se les aplica la regla de crecimiento, produciendo el patrón de la Figura 4.82e. Luego se introduce el píxel denotado 3, y se aplica de nuevo la regla —a los píxeles 3 y 4—, para formar el patrón de Figura 4.82f. Se introducen los cuatro píxeles marcados como 2, y se aplica la regla una vez más, para formar el patrón de la Figura 4.82g. Finalmente, se leen los diez píxeles denotados 1, para completar la imagen. El número de generaciones para estos píxeles es cero; no se utilizan para el crecimiento de otros píxeles y, por lo tanto, no contribuyen a la compresión de la imagen. Observe que los píxeles marcados como 4 pasan por tres generaciones.

En este ejemplo, el codificador escribió los píxeles semilla en la secuencia de datos comprimidos con el fin de disminuir el número de generaciones. En principio, es posible complicarlo como tanto como se desee. Se pueden utilizar distintas reglas de crecimiento para grupos de píxeles diferentes (en

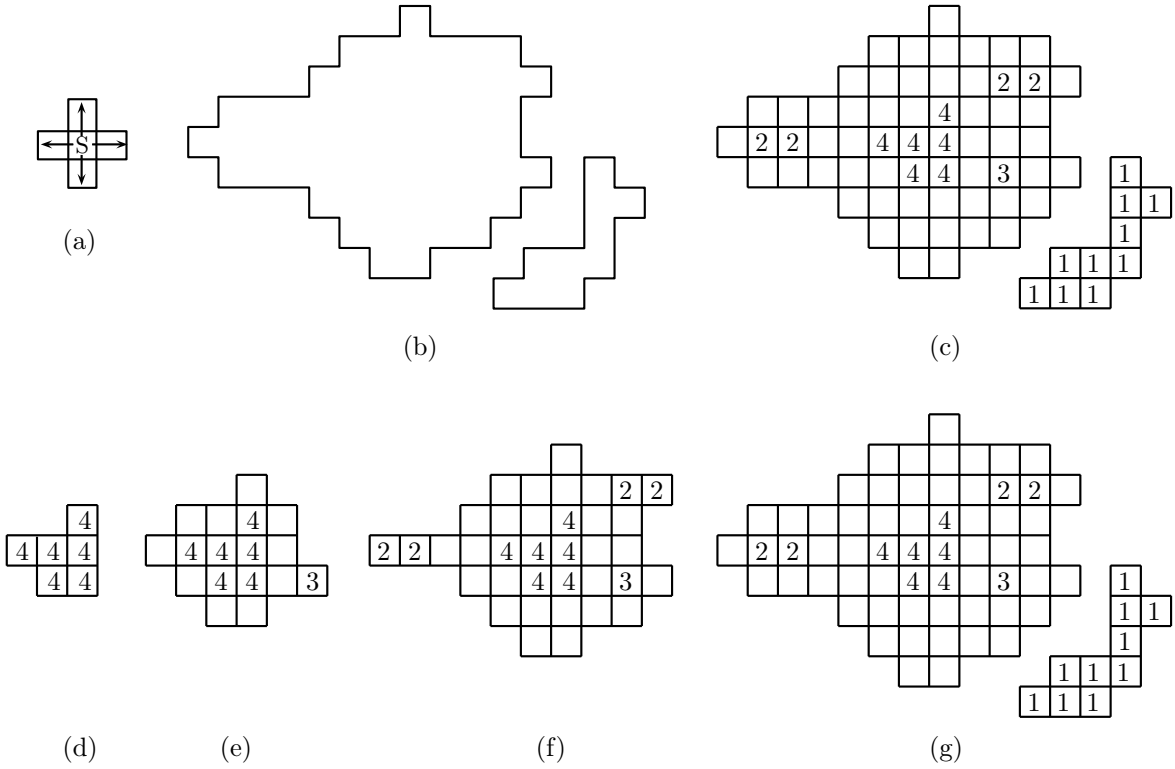
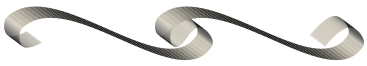


Figura 4.82: El crecimiento de un patrón.

tal caso, las reglas de crecimiento deben informar —de forma inequívoca— quién es el padre de un píxel P, de manera que el decodificador sepa qué regla de crecimiento usar para P), para detener el crecimiento cuando un píxel se adentra a una distancia especificada de otros píxeles, para cambiar la regla de crecimiento cuando un nuevo píxel choca con otro píxel, para invertir la dirección en tal caso e iniciar el borrado de píxeles, o para utilizar cualquier otro algoritmo. En la práctica, sin embargo, las reglas complejas requieren más bits para codificarlas, por lo que las reglas sencillas pueden llevar a una mejor compresión.



◇ **Ejercicio 4.20 (sol. en pág. 1082):** Las reglas de crecimiento sencillas tienen otra ventaja. ¿Cuál es?

La Figura 4.83 ilustra un sencilla propuesta para el diseño de un codificador recursivo que identifica los píxeles semilla en capas. Asumimos que la regla de crecimiento utilizada es la que se muestra en la Figura 4.83a. Todos los píxeles negros (en primer plano) están marcados inicialmente con un 1, y los píxeles (no mostrados en la figura) blancos (fondo) están marcados con un 0. El codificador escanea la imagen y marca con un 2 cada uno de los píxeles 1 que pueden crecer. Para que un píxel 1 pueda crecer debe estar rodeado por píxeles 1 por los cuatro lados. La siguiente capa consta de los píxeles 2 resultantes, que se muestran en la Figura 4.83c. Seguidamente, el codificador busca los píxeles 1 que no tienen al menos un píxel 2 flanqueándolos. Estos son píxels semilla. Hay 10 de ellos, denotados en fondo gris en la Figura 4.83c.

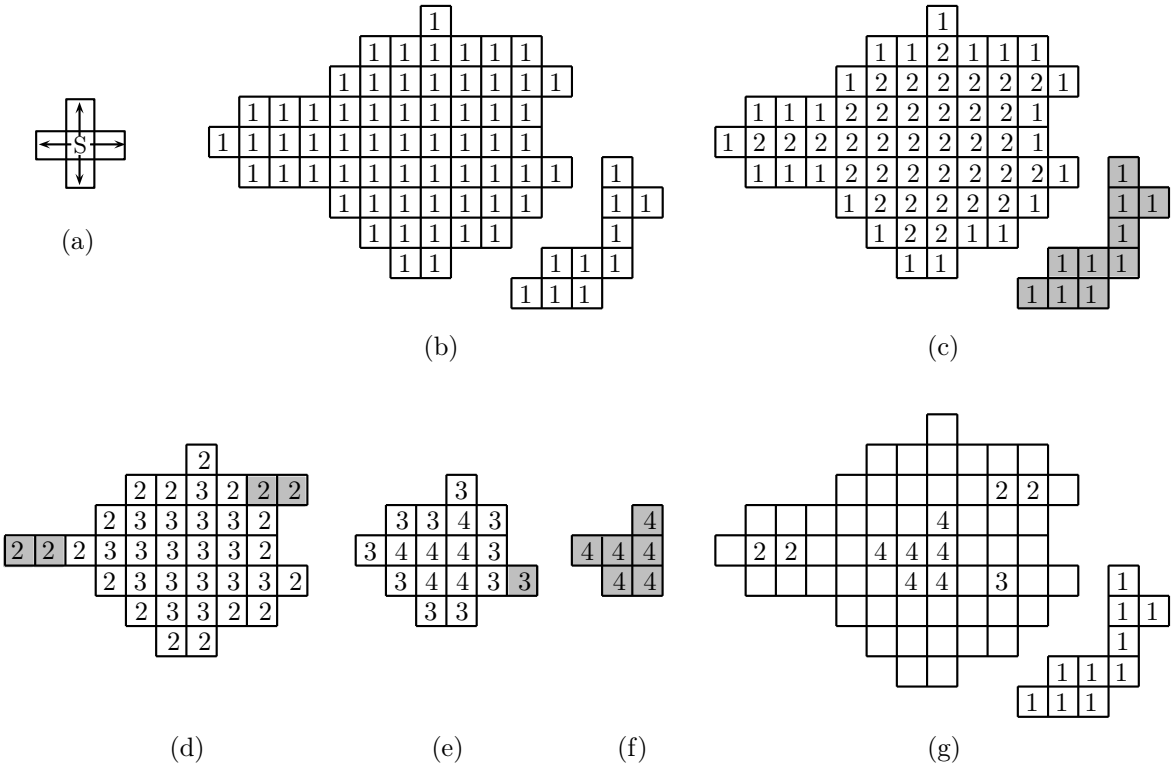


Figura 4.83: Codificación recursiva de un patrón.

El mismo proceso se aplica a la capa de píxeles marcados con 2. El codificador escanea la imagen y marca con 3 cada uno de los píxeles 2 que pueden crecer. La siguiente (tercera) capa está formada por los píxeles 3 resultantes, que se muestran en la Figura 4.83d. Seguidamente, el codificador busca los píxeles 2 que no tienen al menos un píxel 3 flanqueándolos. Estos son píxeles semilla. Hay cuatro de ellos, denotados en fondo gris en la Figura 4.83d.

La Figura 4.83e muestra la cuarta capa. Consta de seis píxeles 4 y un píxel semilla 3 (denotada en gris). La Figura 4.83f muestra las seis semillas de píxeles 4, y los píxeles semilla finales se muestran en la Figura 4.83g.

Este algoritmo base puede ser extendido de diversas maneras, entre ellas cabe señalar el uso varias normas de crecimiento para cada capa, en lugar de sólo una.

4.11. JBIG

Ningún método de compresión puede comprimir de manera eficiente todo tipo de datos. Ésta es la razón por la que continuamente se están desarrollando nuevos métodos de propósito especial. JBIG [JBIG] es un ejemplo de un método de propósito especial. Fue desarrollado específicamente para la compresión progresiva de imágenes binivel. Estas imágenes, llamadas también monocromáticas o en blanco y negro, son comunes en aplicaciones donde los dibujos (técnicos o artísticos), con o sin texto, necesitan guardarse en una base de datos y ser recuperados. Es habitual utilizar los términos "foreground (primer plano)" y "background (fondo)" en lugar de negro y blanco, respectivamente.

El término "compresión progresiva" significa que la imagen se guarda en varias "capas" en la secuencia de datos comprimidos, a medida que las resoluciones son más y más altas. Cuando se descomprime

y se visualiza una de tales imágenes, el espectador ve primero una imagen ruda e imprecisa (la primera capa), seguida por versiones mejoradas de la misma (capas posteriores). De esta manera, si la imagen es la equivocada, puede ser rechazada en una etapa temprana, sin tener que recuperarla y descomprimirla completamente. La Sección 4.10 muestra cómo cada capa de alta resolución utiliza la información de la capa precedente de inferior resolución, por lo que no hay duplicación de datos. Esta característica está soportada por JBIG, donde la define una opción llamada *predicción determinista*.

Aunque JBIG fue diseñado para imágenes binivel, donde cada píxel es un bit, es posible aplicarlo a imágenes en escala de grises separando los planos de bits y comprimiendo cada uno de ellos individualmente, como si se tratara de una imagen binivel. Debe utilizarse el RGC (código de Gray reflejado), en lugar del código binario estándar, como se discute en la Sección 4.2.1.

El nombre JBIG proviene de *Joint Bi-Level Image Processing Group* (grupo conjunto de procesamiento de imágenes binivel). Éste es un grupo de expertos de varias organizaciones internacionales, formados en 1988 para recomendar tales estándares. El nombre oficial del método JBIG es la recomendación T.82 de la ITU-T. La ITU es la Unión Internacional de Telecomunicaciones (parte de las Naciones Unidas). La ITU-T es el sector de estandarización de las telecomunicaciones de la ITU. JBIG utiliza múltiples codificaciones aritméticas para comprimir la imagen, y esta parte de JBIG, que se discute a continuación, está separada de la parte de compresión progresiva que se discute en la Sección 4.11.1.

Una característica importante de la definición de JBIG es que el funcionamiento del codificador no se define en detalle. El estándar JBIG discute los detalles del decodificador y el formato del archivo comprimido. Da por hecho que cualquier codificador que genera un archivo JBIG es un codificador JBIG válido. El método JBIG2 de la Sección 4.12 adopta el mismo enfoque, porque ésto permite a los implementadores llegar con codificadores sofisticados que analizan la imagen original de formas que no se podrían haber imaginado en el momento en que se desarrolló el estándar.

Una característica de la codificación la aritmética es que es fácil separar el modelo estadístico (la tabla con las frecuencias y las probabilidades) de las operaciones de codificación y decodificación. Es fácil codificar, por ejemplo, la primera mitad de una cadena de datos utilizando un modelo y la segunda mitad con otro modelo. Ésto se llama *codificación aritmética múltiple*, y es especialmente útil en la codificación de las imágenes, ya que se aprovecha de las estructuras locales y las interrelaciones que puedan existir en la imagen. JBIG utiliza la codificación aritmética múltiple con muchos modelos, cada uno con una tabla de dos entradas que proporciona las probabilidades de aparición de un píxel blanco y uno negro. Hay entre 1024 y 4096 de estos modelos, dependiendo de la resolución de la imagen que está siendo comprimida.

Una imagen binivel se compone de puntos en primer plano (negro) y de fondo (blanco) llamados *píxeles*. La forma más sencilla para comprimir tal imagen mediante codificación aritmética es contar la frecuencia de los píxeles negros y blancos, y calcular sus probabilidades de aparición. En la práctica, sin embargo, las probabilidades varían de una región a otra de la imagen, y este hecho, puede ser utilizado para producir una mejor compresión. Ciertas regiones, como los márgenes de una página, pueden ser completamente blancas, mientras que otras regiones, tales como el centro de un diagrama complejo, o una línea (recta) larga y gruesa, pueden ser predominante o completamente negras. Por consiguiente, las distribuciones de píxeles en las regiones adyacentes de una imagen pueden variar considerablemente.

Considere una imagen donde el 25 % de los píxeles son de color negro. La entropía de esta imagen es $-0{,}25\log_2 0{,}25 - 0{,}75\log_2 0{,}75 \approx 0{,}8113$. Lo mejor que podemos esperar es utilizar para cada píxel 0,81 bits en lugar de la representación original de 1 bit: una razón de compresión del 81 % (ó 0,81 bpp). Supongamos ahora que descubrimos que el 80 % de la imagen es predominantemente blanca, con sólo un 10 % de píxeles negros, y el 20 % restante tiene un 85 % de píxeles negros. Las entropías de estas zonas son $-0{,}1\log_2 0{,}1 - 0{,}9\log_2 0{,}9 \approx 0{,}47$ y $-0{,}85\log_2 0{,}85 - 0{,}15\log_2 0{,}15 \approx 0{,}61$, por lo que si codificamos cada parte por separado, podemos obtener 0,47 bpp el 80 % del tiempo y 0,61 bpp el 20 % restante. En promedio, ésto se traduce en $0{,}47 \times 0{,}80 + 0{,}61 \times 0{,}20 = 0{,}498$ bpp, o una razón

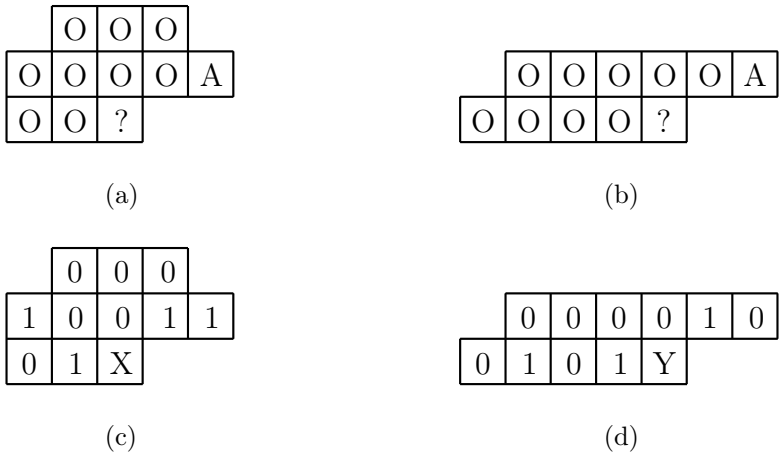


Figura 4.84: Plantillas para las capas de más baja resolución.

de compresión de aproximadamente el 50 %, ¡mucho mejor que el 81 %!

Asumimos que un píxel blanco está representado por un 0, y uno negro por un 1. En la práctica, no conocemos de antemano cuántos píxeles blancos y negros existen en cada zona de la imagen, por lo que el codificador JBIG se detiene en cada píxel y examina una *plantilla* (*template*) construida a partir de los 10 píxeles vecinos —denotados (en las Figuras 4.84 y 4.85) "X" y "A"— ubicados por encima de él y a su izquierda (aquellos que ya han sido introducidos; los de abajo y los de la derecha siguen siendo desconocidos). Interpreta los valores de estos 10 píxeles como un entero de 10 bits que posteriormente es utilizado como un puntero a un modelo estadístico, que, a su vez, se usa para codificar el píxel actual (denotado como "?"). Hay $2^{10} = 1024$ enteros de 10 bits, por lo que deben existir 1024 modelos. Cada modelo es una pequeña tabla formada por las probabilidades de aparición de los píxeles blancos y negros (sólo es necesario guardar una probabilidad en la tabla, ya que la suma de todas las probabilidades es 1).

La Figura 4.84a,b muestra las dos plantillas utilizadas para la capa de menor resolución. El codificador decide si va a utilizar la plantilla de tres líneas o la de dos líneas, y establece el parámetro LRLTWO en el archivo comprimido a 0 ó 1, respectivamente, para informar de esta elección al decodificador. (La plantilla de dos líneas produce una ejecución algo más rápida, mientras que la de tres líneas deriva en una compresión ligeramente mejor.) La Figura 4.84c muestra la plantilla de 10 bits 0001001101, que se convierte en el puntero 77. El puntero mostrado en la Figura 4.84 para el píxel Y es 0000100101 ó 37. Siempre que cualquiera de los píxeles utilizados por las plantillas se encuentre fuera la imagen, debe utilizarse la convención de bordes de JBIG que se mencionó con anterioridad.

La Figura 4.85 muestra las cuatro plantillas utilizadas para todas las otras capas. Estas plantillas reflejan el hecho de que cuando cualquier capa, excepto la primera, es decodificada los píxeles de baja resolución de la capa anterior son conocidas por el decodificador. Cada una de las cuatro plantillas se utiliza para codificar (y luego decodificar) uno de las cuatro píxeles de alta resolución de un grupo. El contexto (puntero) generado en estos casos se compone de 12 bits: 10 tomados de la plantilla de píxeles, y dos generados para indicar cuál de los cuatro píxeles se está procesando en el grupo. El número de modelos estadísticos, por tanto, es $2^{12} = 4096$. Los dos bits que indican la posición de un píxel de alta resolución en su grupo son: 00 para el píxel superior izquierdo (Figura 4.85a), 01 para el superior derecho (Figura 4.85b), 10 para el inferior izquierdo (Figura 4.85c), y 11 para el inferior derecho (Figura 4.85d).

El uso de estas plantillas implica que el modelo de probabilidad de JBIG es un modelo de Markov de orden 10 ó 12.

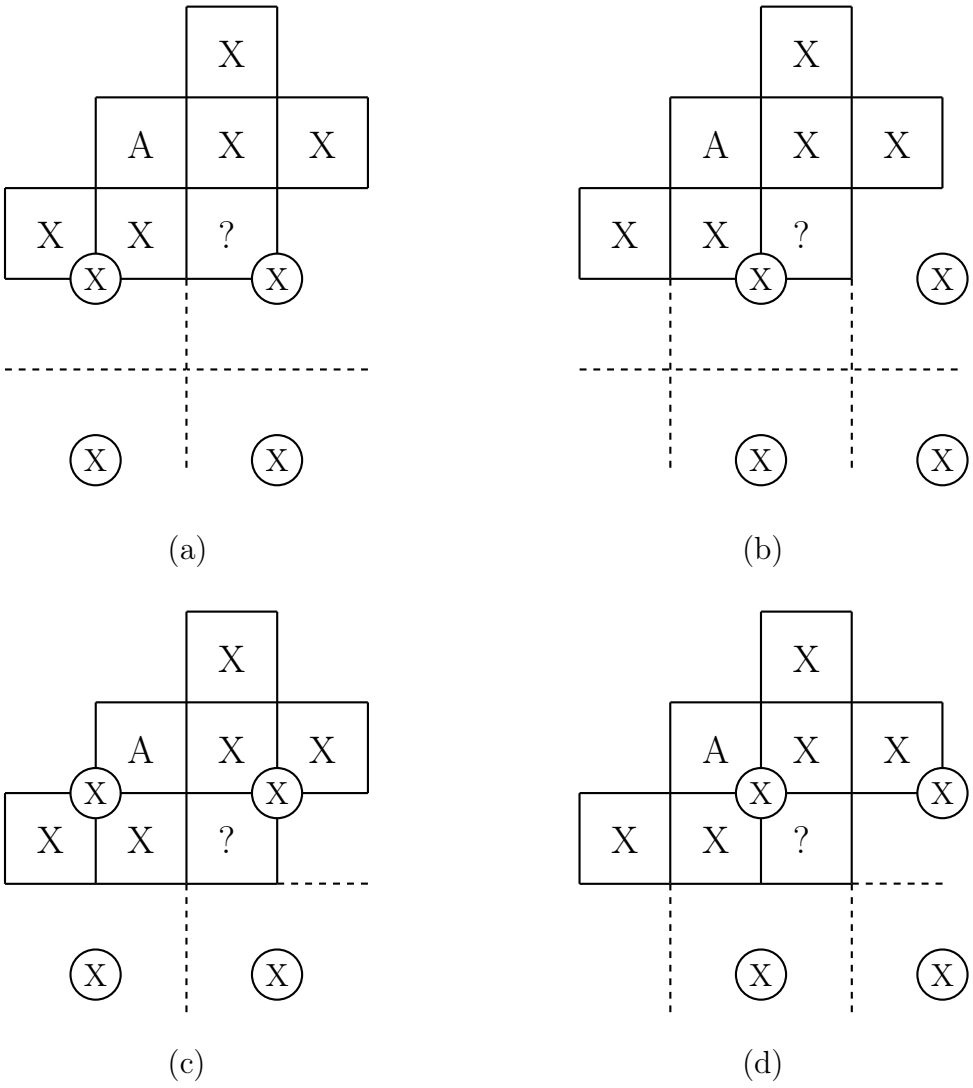


Figura 4.85: Plantillas para las otras capas.

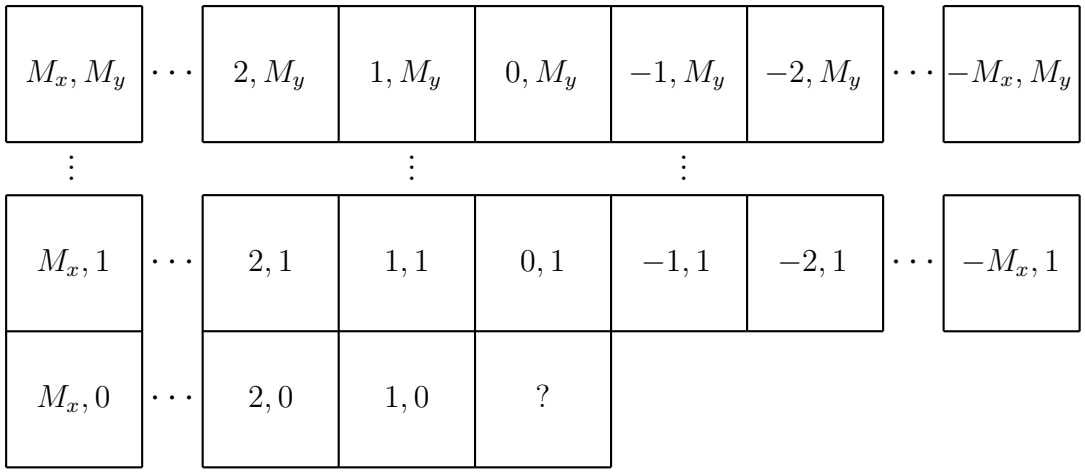


Figura 4.86: Coordenadas y posiciones permitidas de los píxeles AT.

Los píxeles de la plantilla etiquetados "A" se denominan *píxeles adaptativos* (AT). El codificador está autorizado a usar como AT cualquier otro píxel que no esté especificado en la plantilla, y utiliza dos parámetros T_x y T_y (de un byte cada uno) en cada capa, para indicar al decodificador la ubicación real del AT en ésa capa. (no se permite superponer el AT a uno de los píxeles "X" de la plantilla.) Un codificador JBIG sofisticado puede apreciar, por ejemplo, que la imagen utiliza patrones de semitonos donde un píxel negro normalmente tiene otro píxel negro tres filas por encima de él. En tal caso, el codificador puede decidir utilizar el píxel ubicado tres filas más arriba de "?" como AT. La Figura 4.86 muestra el área de la imagen donde el AT puede residir. El parámetro M_x puede variar en el rango $[0, 127]$, mientras que M_y puede tomar los valores, de 0 a 255.

4.11.1. Compresión progresiva

Una ventaja del método JBIG es su capacidad para generar versiones de baja resolución (capas) de la imagen en la secuencia de datos comprimidos. El decodificador descomprime estas capas progresivamente, de menor a mayor resolución.

En primer lugar, mira el orden de las capas. El codificador proporciona la imagen completa (la capa de más alta resolución), por lo que es natural que la construcción de las capas se realice de mayor a menor resolución y se escriban en el archivo comprimido en este orden. El decodificador, por otro lado, tiene que empezar la descompresión y la visualización de la capa de más baja resolución, por lo que es más fácil para él leer esta primera capa. Como resultado, o bien el codificador, o bien el decodificador debe usar un buffer para invertir el orden de las capas. Si la decodificación rápida es importante, el codificador debe utilizar búferes para acumular todas las capas y luego escribirlas en el archivo comprimido de bajas a altas resoluciones. El decodificador lee entonces las capas en el orden correcto. En los casos en que la codificación rápida es importante (por ejemplo, un archivo que está siendo actualizado con frecuencia, pero rara vez es descomprimido y utilizado), el codificador debe escribir las capas en el orden en que se generan (de mayor a menor) y el decodificador debe utilizar búferes. El estándar JBIG permite cualquier método, y el codificador tiene que poner un bit denotado `HITOLO` en el archivo comprimido a cero (capas en orden, de bajo a alto) o a uno (el caso contrario). Se supone que el codificador decide cuál es la resolución que puede tener la capa más baja (que no puede ser de un solo pixel). Esta decisión puede estar basada en datos proporcionados por usuario o en la información construida en el codificador sobre las necesidades específicas de un entorno en particular.

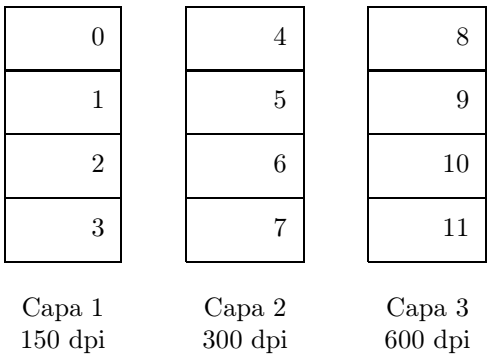


Figura 4.87: Cuatro *stripes* y tres capas en una imagen JBIG.

HITOLO	SEQ	Orden
0	0	0,1,2,3,4,5,6,7,8,9,10,11
0	1	0,4,8,1,5,9,2,6,10,3,7,11
1	0	8,9,10,11,4,5,6,7,0,1,2,3
1	1	8,4,0,9,5,1,10,6,2,11,7,3

Tabla 4.88: Los cuatro órdenes posibles de las capas.

La compresión progresiva en JBIG también implica el concepto de *stripes* (bandas). Un *stripe* es una estrecha banda horizontal, formada por L líneas de escaneo de la imagen, donde L es un parámetro de JBIG controlado por el usuario. Como ejemplo, si L se elige de manera que la altura de una banda sea de 8 mm (aproximadamente 0,3 pulgadas), entonces habrá alrededor de 36 *stripes* en 11,5 pulgadas de la imagen. El codificador escribe los *stripes* en el archivo comprimido en una de dos maneras, estableciendo el parámetro de ajuste `SEQ` para indicar la seleccionada al decodificador: O bien escribe consecutivamente todos los *stripes* de una capa en el fichero comprimido, seguido por los *stripes* de la capa siguiente, y así sucesivamente (`SEQ` = `0`), o bien escribe primero los *stripes* superiores de todas las capas, seguido por los segundos *stripes* de todas las capas, y así sucesivamente (`SEQ` = `1`). Si el codificador establece `SEQ` = `0`, entonces el decodificador descomprimirá la imagen progresivamente, capa por capa, y en cada capa, *stripe* por *stripe*. La Figura 4.87 ilustra el caso de una imagen dividida en cuatro *stripes* y codificada en tres capas (desde la resolución más baja —de 150 dpi— a la más alta —de 600 ppp—). La Tabla 4.88 muestra el orden en que son sacados los *stripes* para los cuatro valores posibles de los parámetros `HITOLO` y `SEQ`.

La idea básica de la compresión progresiva es agrupar cuatro píxeles de alta resolución en uno de baja resolución, un proceso llamado *reducción de resolución* (*downsampling*). El único problema es determinar el valor (blanco o negro) de ése pixel. Si los cuatro píxeles originales tienen el mismo valor, o incluso si tres son idénticos, la solución es obvia (seguir a la mayoría). Cuando dos píxeles son de color negro y los otros dos son de color blanco, se puede intentar una de las siguientes soluciones:

1. Crear un píxel de baja resolución que es siempre de color negro (o siempre blanco). Esta es una mala solución, ya que puede eliminar detalles importantes de la imagen, haciendo impracticable o incluso imposible para un observador evaluar la imagen mediante la visualización de la capa de baja resolución.

2. Asignar un valor aleatorio al nuevo píxel de baja resolución. Esta solución también es mala, ya que puede añadir mucho ruido a la imagen.

3. Dar al píxel de baja resolución el color de la parte superior izquierda de los cuatro píxeles de alta

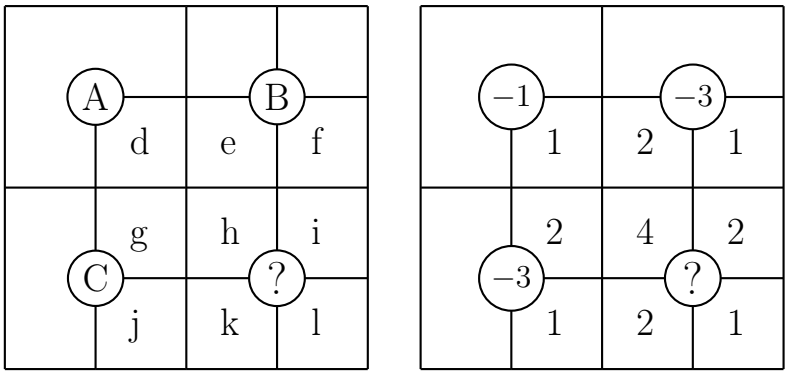


Figura 4.89: Píxeles de alta y baja resolución.

resolución. Ésta prefiere la fila superior y la columna de la izquierda, y tiene el inconveniente de que las líneas delgadas a veces pueden desaparecer completamente en la capa de baja resolución. Además, si la capa de alta resolución usa semitonos para simular escalas de grises, los patrones de medios tonos pueden dañarse.

4. Asignar un valor a los píxeles de baja resolución que dependa de los cuatro píxeles de alta resolución **y** de algunos de sus vecinos más cercanos. Esta solución es la empleada por JBIG. Si la mayoría de los vecinos más cercanos son de color blanco, se crea un píxel blanco de baja resolución; de lo contrario, se crea un píxel negro de baja resolución. La Figura 4.89a muestra los 12 píxeles vecinos de alta resolución y los tres de baja resolución que se utilizan en la toma de esta decisión. A, B, y C son tres píxeles de baja resolución cuyos valores ya han sido determinados. Los píxeles d, e, f, g, y j, están en la parte superior o a la izquierda del grupo actual de cuatro píxeles. El píxel "?" es el píxel de baja resolución, cuyo valor debe ser determinado.

La Figura 4.89b muestra los pesos asignados a los píxeles implicados en la determinación del píxel "?". La suma ponderada de los píxeles también se puede escribir como:

$$\begin{aligned} &4h + 2\,(e + g + i + k) + (d + f + j + l) - 3\,(B + C) - A \\ =&4h + 2\,(i + k) + l + (d - A) + 2\,(g - C) + (j - C) + 2\,(e - B) + (f - B)\,. \end{aligned} \tag{4.25}$$

La segunda línea de la Ecuación (4.25) muestra cómo el valor del píxel "?" depende de diferencias tales como $d - A$ (una diferencia de un píxel de alta resolución y el píxel adyacente de baja resolución). Suponiendo las mismas probabilidades para píxeles blancos y negros, la primera línea de la Ecuación (4.25) puede tener valores comprendidos entre cero (cuando todos los 12 píxeles son de color blanco) y 9 (cuando todos son de color negro), por lo que la norma es asignar al píxel "?" el valor 1 (negro) si la expresión (4.25) es mayor que 4,5 (si es 5 ó más), y el valor 0 (blanco) en caso contrario (si es 4 ó menos). En consecuencia, esta expresión actúa como un filtro que preserva la densidad de los píxeles de alta resolución en la imagen de baja resolución.

◇ **Ejercicio 4.21 (sol. en pág. 1082):** Normalmente representamos un píxel negro (o primer plano) con un 1 binario, y uno blanco (o fondo) con un 0 binario. ¿Cómo afectará al método de reducción de resolución anterior si utilizamos 0 para representar el negro y 1 para representar el blanco?

Puesto que la imagen se divide en grupos de 4×4 píxeles, debe tener un número par de filas y columnas. La *convención de bordes* de JBIG dice que una imagen se puede ampliar siempre y cuando

Los 10 patrones de excepción para preservar las pautas periódicas son (en hex):

`5c7 36d d55 b55 caa aaa c92 692 a38 638`.

Los 12 patrones de excepción para preservar el tramado son:

`fef fd7 f7d f7a 145 142 ebd eba 085 082 028 010`.

Los 132 patrones de excepción para preservar los bordes son:

```
a0f 60f 40f 20f e07 c07 a07 807 607 407 207 007 a27 627 427 227 e1f 61f e17 c17 a17 617
e0f c0f 847 647 447 247 e37 637 e2f c2f a2f 62f e27 c27 24b e49 c49 a49 849 649 449 249
049 e47 c47 a47 e4d c4d a4d 84d 64d 44d 24d e4b c4b a4b 64b 44b e69 c69 a69 669 469 269
e5b 65b e59 c59 a59 659 ac9 6c9 4c9 2c9 ab6 8b6 e87 c87 a87 687 487 287 507 307 cf8 8f8
ed9 6d9 ecb ccb acb 6cb ec9 cc9 949 749 549 349 b36 336 334 f07 d07 b07 907 707 3b6 bb4
3b4 bb2 3a6 b96 396 d78 578 f49 d49 b49 ff8 df8 5f8 df0 5f0 5e8 dd8 5d8 5d0 db8 fb6 bb6
```

donde los 12 bits de cada patrón están numerados como en el diagrama.

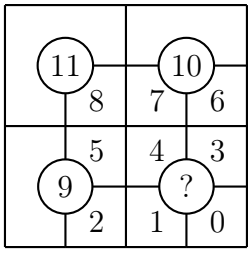


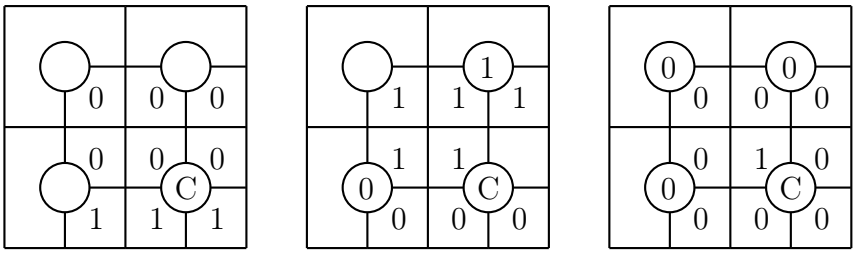
Tabla 4.90: Algunos patrones de excepción de JBIG.

sea indispensable, añadiendo columnas de 0 píxeles a la izquierda y la derecha, filas de 0 píxeles en la parte superior, y replicando la fila inferior tantas veces como sea necesario.

El método de reducción de resolución de JBIG ha sido cuidadosamente diseñado y extensamente probado. Se sabe que produce excelentes resultados para el texto, dibujos, semitonos de escalas de grises, así como para otros tipos de imágenes.

JBIG también incluye excepciones a la regla anterior con el fin de preservar los bordes (132 excepciones), las líneas verticales y horizontales (420 excepciones), los patrones periódicos en la imagen (10 excepciones, para un mejor rendimiento en las regiones de transición hacia y desde los patrones periódicos), y los patrones de tramado de la imagen (12 excepciones que ayudan a preservar los tramados de muy baja o muy alta densidad, i.e., píxeles aislados del fondo o del primer plano). Los dos últimos grupos se utilizan para conservar ciertos patrones de sombreado y también patrones generados por semitonos. Cada excepción es un número de 12 bits. La Tabla 4.90 muestra algunos de los patrones de excepción.

La Figura 4.91 muestra tres patrones típicos de excepción. Un patrón de seis ceros y tres unos, como en la Figura 4.91a, es un ejemplo de excepción, introducida para preservar las líneas horizontales. Ésto significa que el píxel de baja resolución denotado "C" debe ser complementado (asignando el opuesto de su valor normal). El valor normal de este píxel depende, por supuesto, de los tres píxeles de baja resolución por encima de él y hacia la izquierda. Puesto que estos tres píxeles pueden tener ocho valores diferentes, este patrón abarca ocho excepciones. Reflejando 4.91a sobre la diagonal principal se produce un patrón (en realidad, ocho patrones) que se utiliza normalmente como excepción para preservar las líneas verticales de la imagen original. Así, la Figura 4.91a corresponde a 16 de los 420 excepciones de preservación de líneas.



(a) (b) (c)

Figura 4.91: Algunos patrones de excepción de JBIG.

El patrón de la Figura 4.91b se desarrolló con el fin de preservar las líneas gruesas horizontales, o casi horizontales. Una línea horizontal de dos píxeles de ancho en alta resolución, e.g., dará lugar a una línea de baja resolución, cuya anchura alterna entre uno y dos píxeles de baja resolución. Cuando la fila superior de la línea de alta resolución es escaneada, producirá una fila de píxeles de baja resolución, debido a la Figura 4.91a. Cuando la siguiente fila de alta resolución es escaneada, las dos filas de la línea gruesa generarán píxeles de baja resolución alternativos debido a que el patrón 4.91b requiere un cero en el píxel de baja resolución de la parte inferior izquierda con el fin de complementar el píxel de baja resolución inferior derecha.

El patrón 4.91b deja dos píxeles sin especificar, por lo que cuenta para los cuatro patrones. Su reflexión también se utiliza, para preservar las líneas verticales gruesas, por lo que cuenta para ocho de las 420 excepciones de preservación de líneas.

La Figura 4.91c complementa un píxel de baja resolución en los casos donde hay un píxel negro de alta resolución entre los 11 blancos. Ésto es común cuando una imagen en escala de grises se convierte en semitonos blancos y negros. Este patrón es una de las 12 excepciones de preservación de tramados.

El método de JBIG para reducir la resolución parece complicado —especialmente porque incluye excepciones—, pero en realidad es fácil de implementar y también se ejecuta muy rápido La decisión de pintar el píxel actual de baja resolución en blanco o negro depende de 12 de sus píxeles vecinos, nueve píxeles de alta resolución, y tres píxeles de baja resolución. Sus valores se combinan en un número de 12 bits, que se utiliza como un puntero a una tabla de 4096 entradas. Cada entrada en la tabla es un bit, que convierte el valor del píxel actual de baja resolución. De esta manera, todas las excepciones ya están incluidas en la tabla y no requieren ningún tratamiento especial por el programa.

Se ha mencionado en la Sección 4.10 que la compresión mejora si el codificador escribe en el *stream* comprimido sólo tres de los cuatro píxeles de un grupo de la capa i. Ésto es suficiente porque el decodificador puede calcular el valor del cuarto píxel a partir de su tres hermanos y del padre del grupo (el padre procede de la capa anterior, $i-1$). La Ecuación (4.25) muestra que en JBIG, un píxel de baja resolución no se calcula como un sencillo promedio de sus cuatro "hijos" de alta resolución, por lo que puede no ser posible para el decodificador JBIG calcular el valor de, digamos, k píxeles de alta resolución a partir de los valores de sus tres hermanos h, i, y l, y de sus padres. Las muchas excepciones utilizadas por JBIG también complican este cálculo. Como resultado, JBIG utiliza tablas especiales para informar tanto al codificador como al decodificador qué píxeles de alta resolución se pueden deducir de sus hermanos y padres. Dichos píxeles no son comprimidos por el codificador, pero todos los otros píxeles de alta resolución de la capa, sí. Este método se conoce como *predicción determinista* (DP), y es una opción de JBIG. Si el codificador decide utilizarla, establece el parámetro `DPON` a 1.

La Figura 4.92 muestra la numeración de los píxeles utilizada para la predicción determinista.

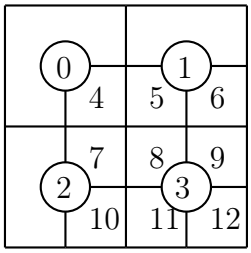


Figura 4.92: Numeración de los píxeles para la predicción determinista.

Puntero	Valor
0−63	02222222 22222222 22222222 22222222 02222222 22222222 22222222 22222222
64−127	02222222 22222222 22222222 22222222 00222222 22222222 22222222 22222222
128−191	02222222 22222222 00222222 22222222 02020222 22222222 02022222 22222222
192−255	00222222 22222222 22222222 22222221 02020022 22222222 22222222 22222222

Tabla 4.93: Predicción del píxel 8.

Puntero	Valor
0−63	22222222 22222222 22222222 22000000 02222222 22222222 00222222 22111111
64−127	22222222 22222222 22222222 21111111 02222222 22111111 22222222 22112221
128−191	02222222 22222222 02222222 22222222 00222222 22222200 20222222 22222222
192−255	02222222 22111111 22222222 22222102 11222222 22222212 22220022 22222212
256−319	20222222 22222222 00220222 22222222 20000222 22222222 00000022 22222221
320−383	20222222 22222222 11222222 22222221 22222222 22222221 22221122 22222221
384−447	20020022 22222222 22000022 22222222 20202002 22222222 20220002 22222222
448−511	22000022 22222222 00220022 22222221 21212202 22222222 22220002 22222222

Tabla 4.94: Predicción del píxel 9.

Antes de comprimir los 8 píxeles alta resolución, el codificador combina los valores de los píxeles 0−7 para formar un puntero de 8 bits para la primera tabla DP. Si la entrada de la tabla es 2, entonces los 8 píxeles deberían ser comprimidos, ya que el decodificador no puede inferirlo de sus vecinos. Si la entrada de la tabla es 0 ó 1, el codificador ignora el píxel 8. El decodificador prepara el mismo puntero (ya que los píxeles 0−7 son conocidos cuando el píxel 8 debe ser decodificado). Si la entrada de la tabla es 0 ó 1, el decodificador asigna este valor al píxel 8; de lo contrario (si la entrada de la tabla es 2), el siguiente valor de píxel se descodifica desde el archivo comprimido y se le asigna al píxel 8. Observe que de las 256 entradas de la Tabla 4.93 sólo 20 no son 2. El Píxel 9 se maneja de forma similar, excepto que el puntero utilizado se compone de los píxeles 0−8, y apunta a una entrada en la segunda tabla DP. Esta tabla, que se muestra en 4.94, consta de 512 entradas, de las cuales 108 no son 2. La norma JBIG también especifica la tercera tabla DP, para el píxel 10 (1024 entradas, de las cuales 526 no son 2), y la cuarta tabla, para el píxel 11 (2048 entradas, de las cuales 1044 no son 2). El tamaño total de las cuatro tablas es de 3840 entradas, de las cuales 1698 (o aproximadamente el 44 %) realmente predicen los valores de los píxeles.

4.12. JBIG2

¡Lo hizo de nuevo! El grupo conjunto de procesamiento de imágenes Binivel (Joint Bi-Level Image Processing Group o JBIG) ha producido otro estándar para la compresión de imágenes binivel. La nueva norma se llama JBIG2, lo que implica que la antigua norma debe ser llamada JBIG1. JBIG2 se desarrolló a finales de la década de 1990 y fue aprobado por la ISO y la ITU-T en 1999. La información actual sobre JBIG2 se puede encontrar en [JBIG2 03] y [JBIG2 06]. El estándar JBIG2 ofrece:

1. Grandes aumentos en el rendimiento de la compresión (típicamente 3−5 veces mejor que el Grupo 4/MMR, y 2−4 veces mejor que JBIG1).
2. Métodos de compresión especiales para texto, medios tonos, y otras partes de la imagen binivel.
3. Compresión con o sin pérdidas.
4. Dos modos de compresión progresiva. El Modo 1 es la compresión progresiva de calidad, donde la imagen decodificada progresa desde baja hasta alta calidad. El Modo 2 es la codificación progresiva de contenidos, donde partes importantes de la imagen (como texto) se decodifican en primer lugar, seguidas por partes menos importantes (tales como patrones de medios tonos).
5. Compresión de documentos multipágina.
6. Formato flexible, diseñado para ser embebido fácilmente en otros formatos de archivo de imágenes.
7. Descompresión rápida: En algunos modos de codificación, las imágenes pueden ser descomprimidas por encima de los 250 millones de píxeles por segundo en software.

La norma JBIG2 describe los principios de la compresión y el formato del archivo comprimido. No entra en el funcionamiento del codificador. Cualquier codificador que produce un archivo comprimido JBIG2 es un codificador JBIG2 válido. Se espera con ésto animar a los desarrolladores de software para implementar sofisticados codificadores JBIG2. El decodificador JBIG2 lee el archivo comprimido, que contiene diccionarios y la información de la imagen, y lo descomprime en el *buffer de página*. La imagen se visualiza o se imprime a partir de el buffer de la página. El decodificador puede utilizar también varios búferes auxiliares.

Un documento a ser comprimido por JBIG2 puede estar formado por más de una página. La característica principal de JBIG2 es que distingue entre texto, imágenes de medios tonos, y cualquier

otra cosa en la página. El codificador JBIG2 debe escanear de alguna manera la página antes de hacer cualquier codificación, e identificar regiones de tres tipos:

1. Regiones de texto. Éstas contienen texto; normalmente se disponen en filas. El texto no tiene que estar en un idioma específico, y puede estar formado por símbolos desconocidos, dingbats[12], notaciones musicales, o jeroglíficos. El codificador trata cada símbolo como un mapa de bit (bitmap) rectangular, lo que lo sitúa en un diccionario. El diccionario es comprimido mediante cualquier codificación aritmética (específicamente, una versión de la codificación aritmética llamada codificador MQ, [Pennebaker y Mitchell 88a] y [Pennebaker y Mitchell 88b] o MMR (Sección 2.13.2) y escrito, como un segmento, en el archivo comprimido. Similarmente, la región de texto en sí se codifica y escribe en el fichero comprimido como otro segmento. Para codificar una región de texto, el codificador prepara las coordenadas relativas de cada símbolo (las coordenadas relativas al símbolo anterior), y un puntero al símbolo del mapa de bits en el diccionario. (También es posible tener punteros a varios bitmaps y especificar el símbolo como una agregación de ésos mapas de bits, con los operadores lógicos AND, OR o XOR). Estos datos también se codifican antes de ser escritos en el archivo comprimido. La compresión se logra si el mismo símbolo aparece varias veces. Observe que el símbolo puede encontrarse en diferentes zonas de texto e incluso en páginas diferentes. El codificador debe asegurarse de que el diccionario que contiene este símbolo del mapa de bits sea retenido por el decodificador durante tanto tiempo como sea necesario. La compresión con pérdida se logra si se utiliza el mismo bitmap para símbolos que son ligeramente diferentes.

2. Regiones de semitonos. Una imagen binivel puede contener una imagen en escala de grises realizada en semitonos [Salomon 99]. El codificador analiza dicha zona celda por celda (las celdas de semitonos, también llamadas patrones, son típicamente de 3×3 ó 4×4 píxeles). Se crea un diccionario de semitonos recolectando todas las celdas diferentes y convirtiendo cada una de ellas a un número entero (típicamente de 9 ó 16 bits). El diccionario se comprime y se escribe en el archivo de salida. Cada celda de la región de semitonos se sustituye entonces por un puntero al diccionario. El mismo diccionario puede ser usado para decodificar varios diccionarios de semitonos, quizás ubicados en diferentes páginas del documento. El codificador debe etiquetar cada diccionario, por lo que el decodificador sabe cuánto tiempo debe retenerlo.

3. Regiones genéricas. Ésta es cualquier región no identificada por el codificador como texto o semitonos. Esta región puede contener un gran carácter, arte lineal, matemáticas, o incluso ruido (motas y suciedad). Una región genérica se comprime usando, o bien codificación aritmética, o bien MMR. Cuando se utiliza la primera, la probabilidad de cada píxel está determinada por su contexto (i.e., por varios píxeles situados por encima de él y a su izquierda que ya han sido codificados).

Una página puede contener cualquier número de regiones, y pueden solaparse. Las regiones son determinadas por el codificador, cuyo funcionamiento no está especificado por el estándar JBIG2. Como consecuencia, un codificador JBIG2 simple puede codificar cualquier página como una región genérica grande, mientras que un codificador sofisticado puede tardar cierto tiempo en analizar el contenido de una página e identificar las distintas regiones, lo que lleva a una mejor compresión. La Figura 4.95 es un ejemplo de una página con cuatro regiones de texto (parte de los puntos 1 y 2 anteriores, "Ejemplo de shearing o extrusión", y "Lena en semitonos (2 por 2 pulgadas)"), una región de semitonos (imagen de Lena), y dos regiones genéricas (recortes de una cara y una huella dactilar).

El hecho de que el estándar JBIG2 no especifique el codificador también se utiliza para generación de compresión con pérdida. Un codificador sofisticado puede decidir que la omisión de ciertos píxeles no deteriora la apariencia de la imagen de manera significativa. El codificador, por lo tanto, elimina

[12]Símbolos tipográficos ornamentales. Cada carácter es un dibujo.

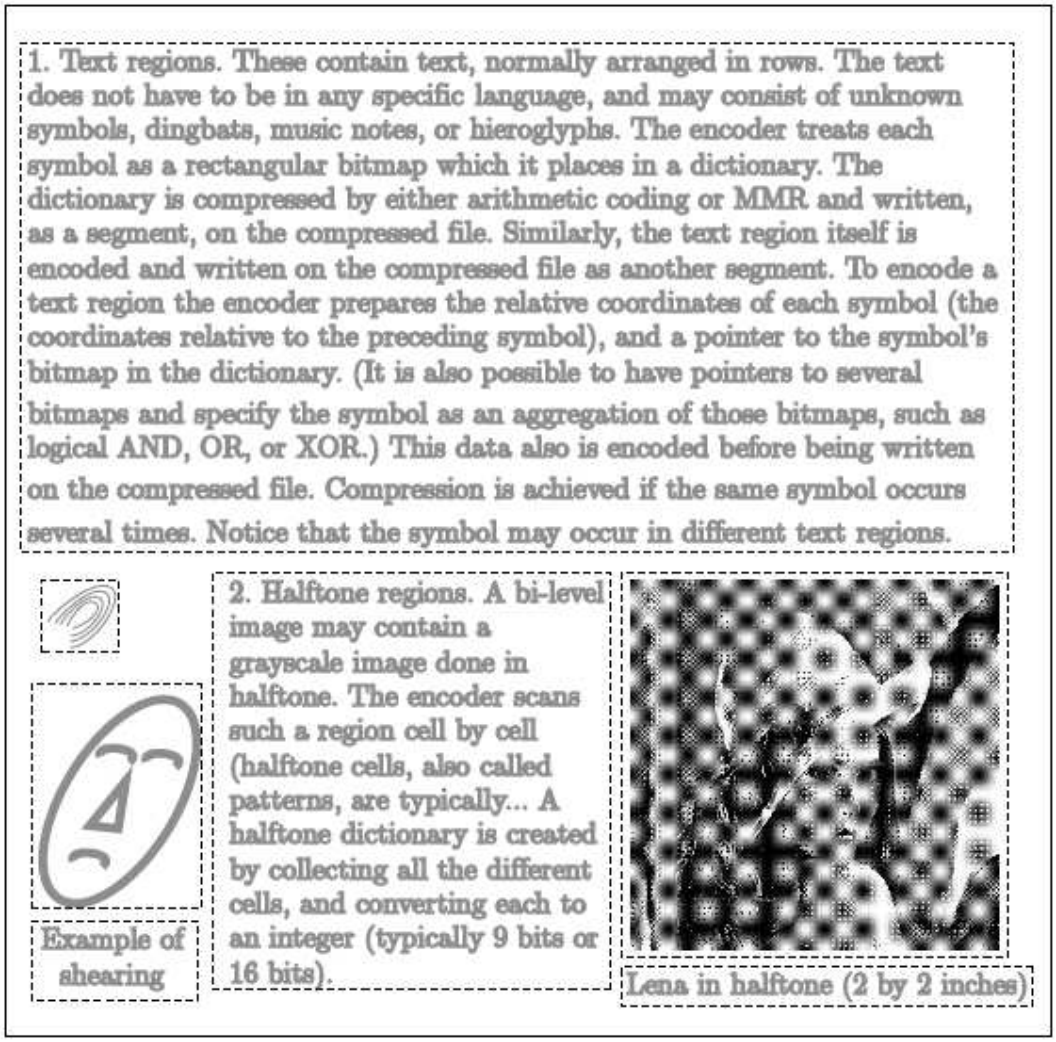


Figura 4.95: Regiones típicas de JBIG2.

ésos bits y crea un pequeño archivo comprimido. El decodificador opera como de costumbre y no sabe si la imagen que está siendo decodificada es con pérdidas o no. Otro enfoque para la compresión con pérdida es que el codificador identifique los símbolos que son muy similares y los sustituiya por punteros al mismo mapa de bits en el diccionario de símbolos (este método es arriesgado, ya que los símbolos que difieren en sólo unos pocos píxeles pueden ser completamente diferentes para un lector humano; piense en "e" y "c" o en "i" y "i"). Un codificador también puede ser capaz de identificar manchas y suciedad en el documento y, opcionalmente, ignorarlas.

JBIG2 introduce el concepto de *refinamiento* de una región. El archivo comprimido puede incluir instrucciones dirigidas al decodificador para decodificar una región A del archivo comprimido en un buffer auxiliar. Este buffer puede ser utilizado más tarde para refinar la decodificación de otra región B. Cuando B se encuentra en el archivo comprimido y se decodifica en el buffer de página, cada píxel escrito en el buffer de la página se determina mediante píxeles decodificados del archivo comprimido y píxeles ubicados en el buffer auxiliar. Un ejemplo de refinamiento de región es un documento donde

ciertas regiones de texto contienen las palabras "Top Secret" en letras grandes, grises, centradas como fondo. Un codificador JBIG2 sencillo puede considerar cada uno de los píxeles que constituyen "Top Secret", parte de algunos símbolos. Un codificador sofisticado puede identificar el fondo común, tratarlo como una región genérica, y comprimirla como un refinamiento de una región, para ser aplicada a ciertas regiones de texto por el decodificador. Otro ejemplo de refinamiento de región es un documento donde ciertas regiones de semitonos tienen un fondo oscuro. Un codificador sofisticado puede identificar el fondo oscuro en ésas regiones, tratarlas como una región genérica, y colocar las instrucciones en el archivo comprimido diciendo al decodificador que utilice esta región para perfeccionar ciertas zonas de semitonos. Por lo tanto, un codificador JBIG2 sencillo puede no usar nunca la región de refinamiento, mientras que uno sofisticado puede utilizar este concepto para añadir efectos especiales a algunas regiones.

El decodificador comienza inicializando el buffer de página a un cierto valor —0 ó 1—, de acuerdo con un código leído desde el archivo comprimido. A continuación, introduce el resto del archivo, segmento por segmento, y ejecuta para cada uno de ellos un procedimiento diferente. Hay siete procedimientos principales:

1. El procedimiento para decodificar las cabeceras de los segmentos. Cada segmento comienza con una cabecera que incluye, entre otros datos y parámetros, el tipo de segmento, el destino de la salida decodificada del segmento, y qué otros segmentos tienen que ser utilizados en la decodificación este segmento.

2. Un procedimiento para decodificar una región genérica. Se invoca cuando el decodificador encuentra un segmento que describe tal región. El segmento está comprimido —ya sea con codificación aritmética, codificación, ya sea con MMR—, y el procedimiento lo descomprime (píxel por píxel en el primer caso y rachas de píxeles en el segundo). En el caso de la codificación aritmética, los píxeles previamente decodificados se utilizan para formar un contexto de predicción. Una vez que un píxel es decodificado, el procedimiento no sólo tiene que almacenarlo en el buffer de página, sino que lo combina con el píxel que ya está en el buffer de página de acuerdo con una operación lógica (AND, OR, XOR, o XNOR) especificada en el segmento.

3. Un procedimiento para decodificar una región refinamiento genérico. Es similar al anterior, excepto que modifica un buffer auxiliar en lugar del buffer de página.

4. Un procedimiento para decodificar un diccionario de símbolos. Se invoca cuando el decodificador encuentra un segmento que contiene tal diccionario. El diccionario se descomprime y se almacena como una lista de símbolos. Cada símbolo es un mapa de bits que, o bien está expresamente especificado en el diccionario, o bien se especifica como un refinamiento (i.e., una modificación) de un símbolo conocido (un símbolo anterior procedente de este diccionario o de otro diccionario existente), o bien se especifica como un agregado (una combinación lógica) de varios símbolos conocidos).

5. Un procedimiento para decodificar una región de símbolos. Se invoca cuando el decodificador encuentra un segmento que describe una región. El segmento se descomprime y produce tripletes. El triplete de un símbolo contiene las coordenadas del símbolo relativas al símbolo anterior y un puntero (índice) al símbolo en el diccionario de símbolos. Puesto que el decodificador puede mantener varios diccionarios de símbolos en cualquier momento, el segmento debe indicar qué diccionario se va a utilizar. El mapa de bits de símbolos se guarda en el diccionario, y los píxeles se combinan con los del buffer de página de acuerdo con la operación lógica especificada en el segmento.

6. Un procedimiento para decodificar un diccionario de semitonos. Se invoca cuando el decodificador encuentra un segmento que contiene tal diccionario. El diccionario se descomprime y se almacena como una lista de patrones de semitonos (mapas de bits de tamaño fijo).

Nombre	Tipo	Tamaño	¿Con signo?	Descripción
MMR	I	1	N	Uso de MMR o codificación aritmética
GBW	I	32	N	Ancho de la región
GBH	I	32	N	Alto de la región
GBTEMPLATE	I	2	N	Número de plantilla
TPON	I	1	N	¿Uso de la predicción típica?
USESKIP	I	1	N	¿Saltar algunos píxeles?
SKIP	B			Mapa de bits (*bitmap*) para saltar
$\mathbf{GBATX}_1$	I	8	S	Coordenada X relativa de A_1
$\mathbf{GBATX}_1$	I	8	S	Coordenada Y relativa de A_1
$\mathbf{GBATX}_2$	I	8	S	Coordenada X relativa de A_2
$\mathbf{GBATY}_2$	I	8	S	Coordenada Y relativa de A_2
$\mathbf{GBATX}_3$	I	8	S	Coordenada X relativa de A_3
$\mathbf{GBATY}_3$	I	8	S	Coordenada Y relativa de A_3
$\mathbf{GBATX}_4$	I	8	S	Coordenada X relativa de A_4
$\mathbf{GBATY}_4$	I	8	S	Coordenada Y relativa de A_4

Tabla 4.96: Parámetros para la decodificación de una región genérica.

7. Un procedimiento para decodificar una región de semitonos. Se invoca cuando el decodificador encuentra un segmento que describe tal región. El segmento se descomprime en un conjunto de punteros (índices) a los patrones en el diccionario de semitonos.

Algunos de estos procedimientos se describen aquí con más detalle.

4.12.1. Decodificación de una región genérica

Este procedimiento lee varios parámetros desde el archivo comprimido (Tabla 4.96, donde "I" denota un entero y "B" denota un mapa de bits), el primero de los cuales, **MMR**, especifica el método de compresión utilizado en el segmento a punto de ser decodificado. Puede utilizarse como método, ya sea codificación aritmética, ya sea MMR. El primero es similar a la codificación aritmética en JBIG1 y se describe posteriormente. El último se utiliza como en la compresión de faxes (Sección 2.13).

Si se utiliza la codificación aritmética, los píxeles se decodifican uno por uno, y se colocan fila por fila en la región genérica que está siendo decodificada (parte del buffer de página), cuya anchura y altura están dadas por los parámetros **GBW** y **GBH**, respectivamente. No se almacenan simplemente allí, sino que se combina lógicamente con los píxeles de fondo existentes. Recordemos que el proceso de descompresión en la codificación aritmética requiere el conocimiento de las probabilidades de los ítems que están siendo descomprimidos (decodificados). En nuestro caso, estas probabilidades se obtienen generando una plantilla para cada píxel que está siendo decodificado, usando la plantilla para generar un número entero (el contexto del píxel), y utilizando ése entero como un puntero a una tabla de probabilidades. El parámetro **GBTEMPLATE** especifica cuál de los cuatro tipos de plantillas debe ser utilizado. La Figura 4.97a−d muestra las cuatro plantillas —valores de 0−3, respectivamente— correspondientes a **GBTEMPLATE** (nótese que las dos plantillas de 10 bits son idénticas a las plantillas utilizadas por JBIG1, la Figura 4.84b,d). El píxel denotado "O" es el único que está siendo decodificado, y el "X" y el "A_i" son píxeles conocidos. Si "O" está cercano a un borde de la región, se asume que sus vecinos ausentes son cero.

Los valores de los píxeles "X" y los "A_i" (un bit por píxel) se combinan para formar un contexto (un entero) de entre 10 y 16 bits. Se espera que los bits sean recogidos de arriba abajo y de izquierda a derecha, pero el estándar no especifica éso. Las dos plantillas de la Figura 4.98a,b, e.g., deben producir los contextos 1100011100010111 y 1000011011001, respectivamente. Una vez que se ha calculado un

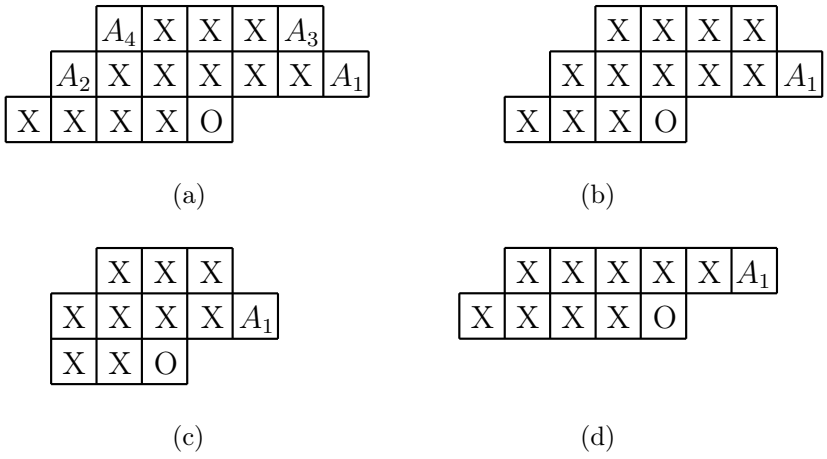


Figura 4.97: Cuatro plantillas para la decodificación de una región genérica.

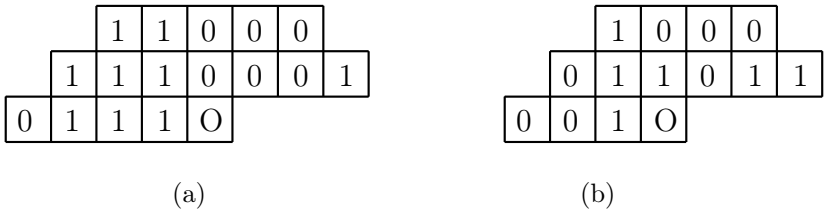


Figura 4.98: Dos plantillas.

contexto, se utiliza como un puntero a una tabla de probabilidad, y la probabilidad encontrada se envía al decodificador aritmético para decodificar el píxel "O".

El codificador especifica cuál de los cuatro tipos de plantilla se debe utilizar. Un sencilla regla general es utilizar plantillas de gran tamaño para las regiones grandes.

Una característica interesante de las plantillas es el uso de los píxeles A_i. Se les llama *adaptativos* o píxeles AT y pueden estar situados en otras posiciones distintas a las mostradas. La Figura 4.97 muestra sus posiciones normales, pero un codificador sofisticado puede descubrir, por ejemplo, que el píxel ubicado tres filas por encima del actual "O" es siempre idéntico a "O". El codificador puede, en tal caso, decir al decodificador (por medio de los parámetros $\mathbf{GBATX}_i$ y $\mathbf{GBATY}_i$) que localice A_1 en la dirección $(0, -3)$ relativa a "O". La Figura 4.99 muestra las posiciones admisibles de los píxeles AT relativas al píxel actual, y sus coordenadas. Observe que un píxel AT también puede estar situado en una de las posiciones "X" de la plantilla.

◇ **Ejercicio 4.22 (sol. en pág. 1082):** ¿Cuáles son las coordenadas relativas de los píxeles AT en los cuatro tipos de plantillas mostrados en la Figura 4.97?

El parámetro **TPON** controla la característica denominada *predicción típica*. Un fila típica se define como una fila que es idéntica a su predecesora. Si el codificador percibe que ciertas filas de la región genérica que están siendo codificadas son típicas, establece **TPON** a 1 y a continuación, escribe un código en el archivo comprimido antes de cada fila, indicando si es típico o no. Si el decodificador encuentra **TPON** a 1, tiene que decodificar y comprobar un código especial que precede a cada fila de píxeles. Cuando encuentra el código de una fila típica, el decodificador simplemente la genera copiando su predecesora.

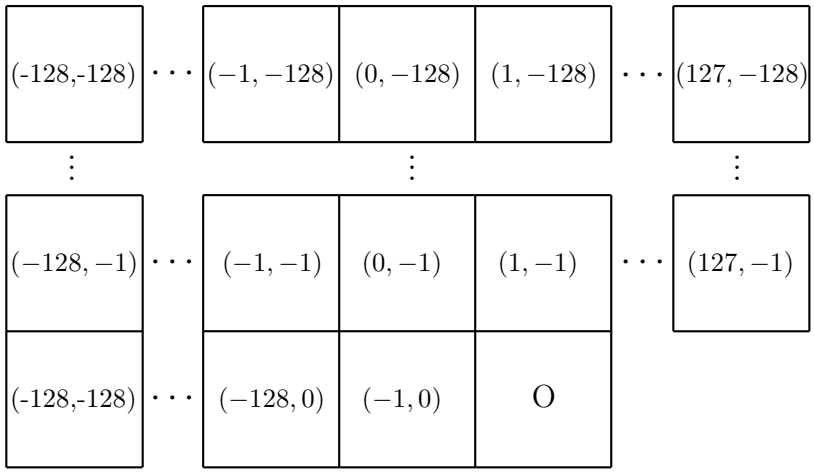


Figura 4.99: Coordenadas y posiciones permitidas de los píxeles AT.

Los dos parámetros **USESKIP** y **SKIP** controlan la función de *salto*. Si el codificador descubre que la región genérica que está codificando es dispersa (es decir, que la mayoría de sus píxeles son iguales a cero), establece **USESKIP** a 1 y **SKIP** a un mapa de bits de tamaño **GBW** × **GBH**, donde cada bit 1 indica un bit cero en la región genérica. El codificador entonces comprime sólo los bits 1 de la región genérica.

4.12.2. Decodificación de una región de símbolos

Este procedimiento se invoca cuando el decodificador comienza a leer un nuevo segmento del archivo comprimido y lo identifica como un segmento de región de símbolos. El procedimiento comienza leyendo muchos parámetros del segmento. A continuación, introduce la información codificada para cada símbolo y la decodifica (utilizando codificación aritmética, o MMR). Esta información contiene las coordenadas del símbolo relativas a su fila y el símbolo que la precede, un puntero al símbolo en el diccionario de símbolos, y posiblemente también información de refinamiento. Las coordenadas de un símbolo se denotan por S y T. Normalmente, S es la coordenada x y T es la coordenada y de un símbolo. Sin embargo, si el parámetro **TRANSPOSED** tiene el valor 1, el significado de S y T se invierte. En general, T se considera la altura de una fila de texto y S es la coordenada de un símbolo de texto dentro de su fila.

◇ **Ejercicio 4.23 (sol. en pág. 1082):** ¿Cuál es la ventaja de la transposición de S y T?

Los símbolos se codifican y escriben en el fichero comprimido por bandas (*strips*). Una banda es normalmente una fila de símbolos, pero también puede ser una columna. El codificador toma ésa decisión y establece el parámetro **TRANSPOSED** a 0 ó 1 acorde con la elección. El decodificador, por lo tanto, comienza decodificando el número de bandas, y a continuación, las propias bandas. Para cada banda, el archivo comprimido contiene la coordenada T de la misma relativa a la banda anterior, seguida por información codificada para los símbolos que constituyen la banda. Para cada símbolo, esta información consta de la coordenada S del símbolo (la separación entre él y el símbolo precedente), su coordenada T (relativa a la coordenada T de la banda), su ID (un puntero al diccionario de símbolos), e información de refinamiento (opcionalmente). En el caso especial en que todos los símbolos se alineen verticalmente en la banda, sus coordenadas T serán cero.

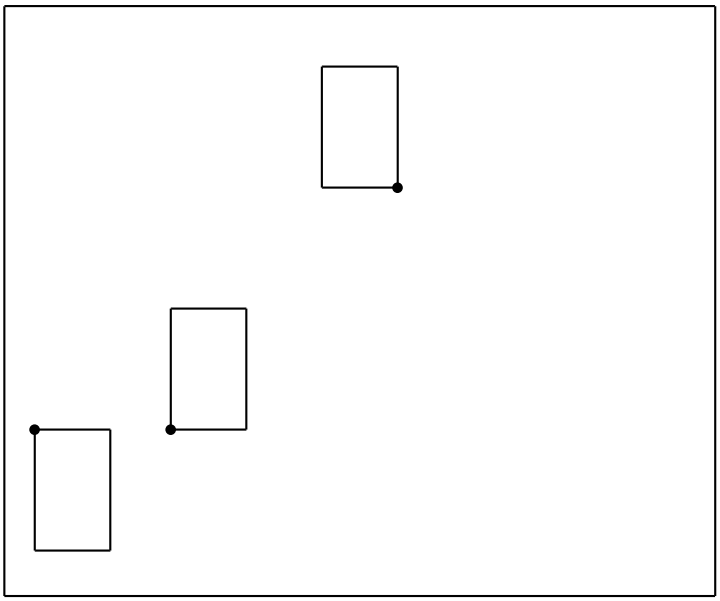


Figura 4.100: Tres mapas de bits de símbolos alineados en diferentes esquinas.

Una vez que las coordenadas absolutas (x, y) del símbolo en la región de símbolos han sido calculadas a partir de las coordenadas relativas, se recupera el *bitmap* del símbolo del diccionario de símbolos y se combina con el buffer de página. Sin embargo, el parámetro **REFCORNER** indica cuál de las cuatro esquinas del bitmap debe ser colocado en la posición (x, y). La Figura 4.100 muestra ejemplos de mapas de bits (bitmaps) de símbolos alineados de diferentes maneras.

Si el codificador decide utilizar MMR para codificar una región, selecciona una de las 15 tablas de códigos de Huffman definidas por el estándar JBIG2 y establece un parámetro para indicar al decodificador qué tabla se utiliza. Las propias tablas están incorporadas tanto en el codificador como en el decodificador. La Tabla 4.101 muestra dos de las 15 tablas. El valor de OOB (fuera de límite) se utiliza para terminar una lista en los casos en que la longitud de la lista no se conozca de antemano.

4.12.3. Decodificación de una región de semitonos

Este procedimiento se invoca cuando el decodificador comienza a leer un nuevo segmento del archivo comprimido y lo identifica como un segmento de región de semitonos. El procedimiento comienza con la lectura de varios parámetros del segmento y el establecimiento de todos los píxeles en la región de semitonos al valor del parámetro relativo al fondo **HDEFPIXEL**. A continuación, introduce la información codificada para cada patrón de semitonos y la decodifica utilizando codificación aritmética o MMR. Ésta información se compone de un puntero a un patrón de semitonos en el diccionario de semitonos. Se recupera el patrón y se combina lógicamente con los píxeles de fondo que ya están en la región de semitonos. El parámetro **HCOMBOP** especifica la combinación lógica. Puede tener uno de los valores: REPLACE, OR, AND, XOR y XNOR.

Nótese que la información de la región leída del archivo comprimido por el decodificador no incluye las posiciones de los patrones de semitonos. El decodificador agrega los patrones a la región en puntos de la retícula consecutivos de la *rejilla de semitonos* (*halftone grid*), que se define a sí misma mediante cuatro parámetros. Los parámetros **HGX** y **HGY** especifican el origen de la cuadrícula relativa al origen de la región de semitonos. Los parámetros **HRX** y **HRY** especifican la orientación de la rejilla

Valor	Código
$0-15$	0+Valor codificado como 4 bits
$16-271$	$10+$(Valor -16) codificado como 8 bits
$272-65807$	$110+$(Valor -272) codificado como 16 bits
$65808-\infty$	$111+$(Valor -65808) codificado como 32 bits

Valor	Código
0	0
1	10
2	110
$3-10$	$1110+$(Valor -3) codificado como 3 bits
$11-74$	$11110+$(Valor -11) codificado como 6 bits
$75-\infty$	$111110+$(Valor -75) codificado como 32 bits
OOB	11111

Figura 4.101: Dos tablas de códigos de Huffman para la decodificación JBIG2.

por medio de un ángulo θ. Los dos últimos parámetros se pueden interpretar como el componente horizontal y vertical de un vector $\mathbf{v}$. En ésta interpretación, θ es el ángulo entre $\mathbf{v}$ y el eje x. Los parámetros también se pueden interpretar como el coseno y el seno de θ, respectivamente. (Estrictamente hablando, son múltiplos del seno y del coseno, puesto que $\sin^2\theta+\cos^2\theta$ es igual a la unidad, pero la suma $\mathbf{HRX}^2+\mathbf{HRY}^2$ puede tener cualquier valor.) Cabe señalar que estos parámetros pueden ser también negativos y no están limitados a valores enteros. Se escriben en el archivo comprimido como enteros cuyos valores son 256 veces su valor real. Así, por ejemplo, si **HRX** debe ser $-0{,}15$, se escribe como el entero -38 porque $-0{,}15\times 256=-38{,}4$. La Figura 4.102a muestra las relaciones típicas entre una página, una región de semitonos, y una rejilla de semitonos. También se muestran las coordenadas de tres puntos sobre la rejilla.

El decodificador realiza un doble bucle en el que varía m_g desde cero hasta $\mathbf{HGH}-1$, y para cada valor de m_g, varía n_g desde cero hasta $\mathbf{HGW}-1$ (los parámetros **HGH** y **HGW** son el número de puntos de la rejilla horizontales y verticales, respectivamente). En cada iteración el par (n_g, m_g) constituye las coordenadas de un punto de la rejilla. Este punto es mapeado a un punto (x, y) en la región de semitonos usando la relación:

$$\begin{aligned} x &= \mathbf{HGX}+m_g\times\mathbf{HRY}+n_g\times\mathbf{HRX},\\ x &= \mathbf{HGY}+m_g\times\mathbf{HRX}-n_g\times\mathbf{HRY}. \end{aligned} \tag{4.26}$$

Para comprender esta relación, el lector debe compararla con una rotación de un punto (n_g, m_g) hasta un ángulo θ alrededor del origen. Tal rotación se expresa mediante:

$$(x,y)=(n_g,m_g)\begin{pmatrix}\cos\theta & -\sin\theta\\ \sin\theta & \cos\theta\end{pmatrix}=(n_g\cos\theta+m_g\sin\theta, m_g\cos\theta-n_g\sin\theta)\,. \tag{4.27}$$

Una comparación de las Ecuaciones (4.26) y (4.27) muestra que son idénticas si asociamos **HRX** con $\cos\theta$ y **HRY** con $\sin\theta$ (La Ecuación (4.26) también añade el origen de la rejilla relativa a la región, por lo que produce un punto (x, y) cuyas coordenadas son relativas al origen de la región).

Normalmente, se espera que la rejilla sea idéntica a la región, i.e., su origen debe ser $(0, 0)$ y su ángulo de rotación, cero. Ésto se logra poniendo **HGX**, **HGY**, y **HRY** a cero y estableciendo **HRX** en la anchura de un patrón de semitonos. Sin embargo, se pueden utilizar los cuatro parámetros, y un codificador JBIG2 sofisticado puede mejorar el calidad de la compresión global, probando diferentes combinaciones.

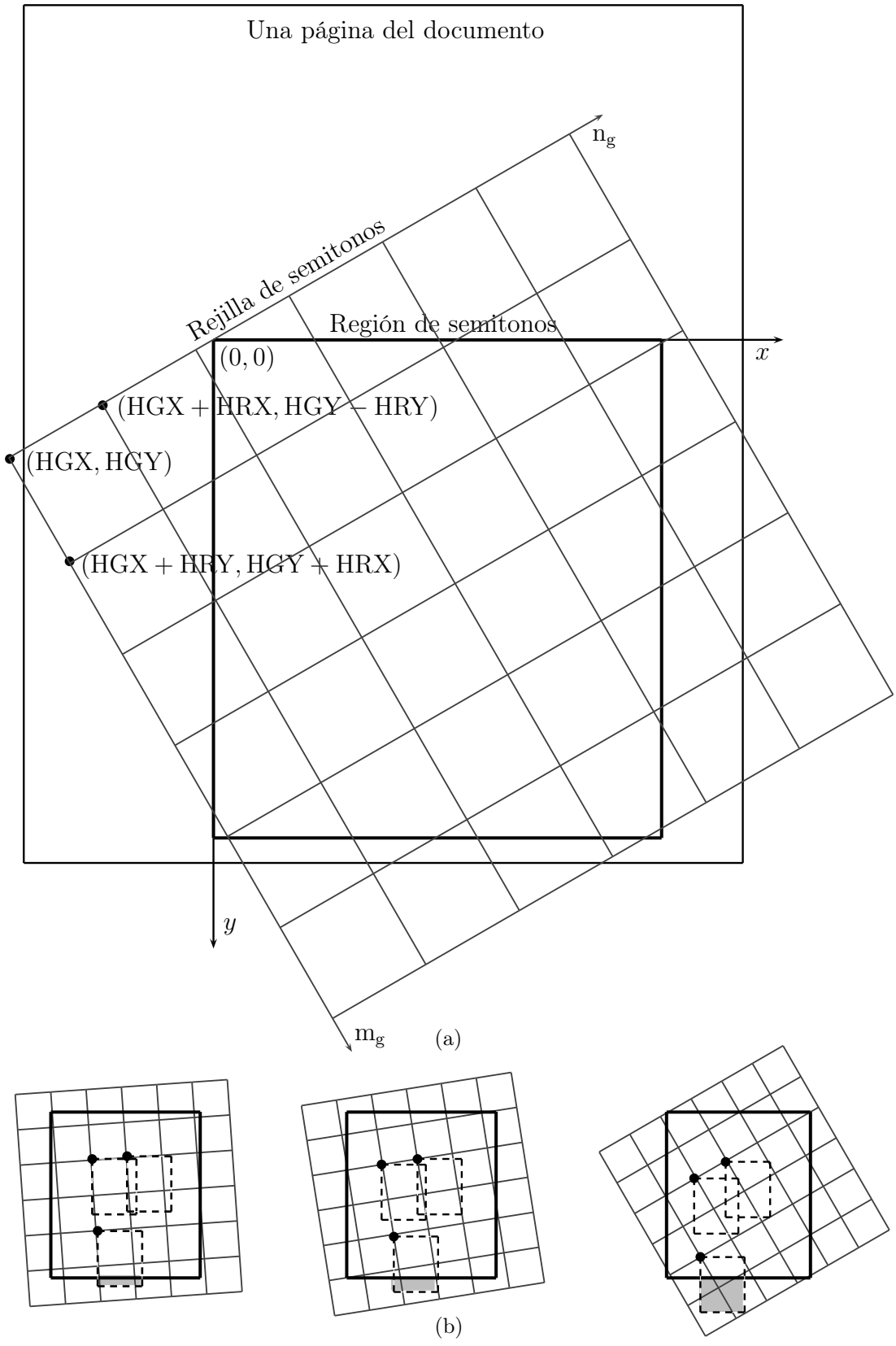


Figura 4.102: Rejillas de semitonos y regiones.

Una vez calculado el punto (x, y), el patrón de semitonos se combina con la región de semitonos de forma que su esquina superior izquierda esté en la posición (x, y) de la región. Dado que partes de la rejilla pueden estar fuera de la región, las zonas del patrón que se encuentran fuera de la región son ignoradas (véanse las zonas grises de la Figura 4.102b). Nótese que los propios patrones se orientan con la región, no con la rejilla. Cada patrón es un rectángulo con lados horizontales y verticales. La orientación de la cuadrícula sólo afecta a la posición de la esquina superior izquierda de cada patrón. Como resultado, los patrones añadidos a la región de semitonos generalmente se superponen en una cantidad que depende en gran medida de la orientación y la posición de la rejilla. Ésto se refleja también en los tres ejemplos de la Figura 4.102b.

4.12.4. El proceso de decodificación completo

El decodificador comienza leyendo del archivo comprimido información general para la página 1 del documento. Ésta información indica al decodificador qué valor de fondo (0 ó 1) poner en el buffer de página inicialmente, y cuál de los cuatro operadores de combinación —OR, AND, XOR y XNOR— utilizar cuando se combinen los píxeles con el buffer de página. El decodificador entonces lee los segmentos del archivo comprimido hasta que encuentra la información relativa a la página 2 o el final del archivo. Un segmento puede especificar su propio operador de combinación, que prevalece sobre el de la página entera. Un segmento puede ser un segmento de diccionario o un segmento de imagen. El último puede ser de uno de estos cuatro tipos:

1. Un segmento de imagen directa inmediato. El decodificador utiliza este tipo para decodificar una región directamente en el buffer de página.

2. Un segmento de imagen directa intermedio. El decodificador utiliza este tipo para decodificar una región en un buffer auxiliar.

3. Un refinamiento de un segmento de imagen inmediato. El decodificador utiliza este tipo para decodificar una imagen y combinarla con una región existente con el fin de perfeccionar esa región. La región que está siendo refinada se encuentra en el buffer de página, y el proceso de refinamiento puede utilizar un buffer auxiliar (que luego es eliminado).

4. Un refinamiento de un segmento de imagen intermedio. El decodificador utiliza este tipo para decodificar una imagen y combinarla con una región existente con el fin de perfeccionar esa región. La región que está siendo refinada se encuentra en un buffer auxiliar.

Concluimos, por lo tanto, que un segmento inmediato se decodifica en el buffer de página y un segmento intermedio involucra a un buffer auxiliar.

4.13. Imágenes simples: EIDAC

Los métodos de compresión de imágenes basadas en transformaciones ofrecen un mejor rendimiento con las imágenes de tonos continuos. Hay, sin embargo, una clase importante de imágenes en las que los métodos basados en transformaciones, tales como JPEG (Sección 4.8) y los distintos métodos de wavelets (Capítulo 5), producen una compresión mediocre. Ésta es la clase denominada *imágenes simples*. Una imagen simple es aquella que utiliza una pequeña fracción de todas las posibles escalas de grises o colores de que dispone. Un ejemplo común es una imagen binivel donde se representa cada píxel con ocho bits. Tal imagen utiliza sólo dos colores de una paleta de 256 colores posibles. Otro ejemplo es una imagen en escala de grises escaneada desde una imagen binivel. La mayoría de los píxeles serán negros o blancos, pero algunos píxeles pueden tener otras tonalidades de gris. Un *cartoon* (caricatura, dibujo animado o no, y similares) es también un ejemplo de imagen simple (sobre todo un

cartoon barato, donde sólo se utilizan unos pocos colores). Un *cartoon* típico está formado por áreas uniformes, por lo que puede utilizar un número pequeño de colores de una paleta potencialmente grande.

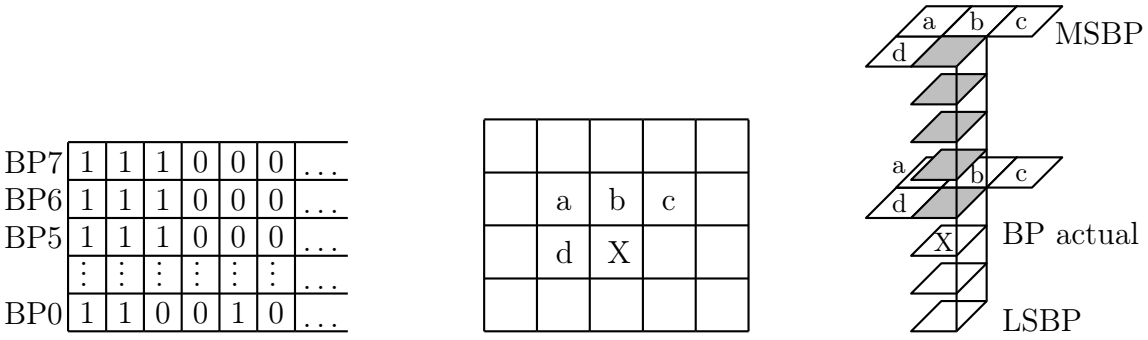
EIDAC es un acrónimo que procede de *embedded image-domain adaptive compression* (compresión adaptativa de imágenes embebidas) [Yoo et al. 98]. Este método está especialmente diseñado para la compresión imágenes simples y las combina factores de alta compresión con un rendimiento sin pérdidas (aunque la compresión con pérdida es una opción). El método es también progresivo. Comprime cada plano de bits por separado, al pasar del *bitplane* más significativo (MSBP) al menos significativo (LSBP). El decodificador lee los datos del primer MSBP, y genera inmediatamente una versión de la imagen tosca en "blanco y negro". Esta versión se mejora cada vez que se lee y decodifica un plano de bits. La compresión con pérdida se logra si algunos de los LSBPs no son procesados por el codificador.

El codificador escanea cada *bitplane* de arriba abajo y de izquierda a derecha (orden *raster*) y utiliza varios de los vecinos más cercanos al píxel actual X como contexto de X, para determinar una probabilidad para X. El píxel X y su probabilidad se envían entonces a un codificador aritmético adaptativo que realiza la codificación real. Por consiguiente, EIDAC se asemeja a JBIG y CALIC, pero hay una importante diferencia. EIDAC utiliza un *contexto bipartito* para cada píxel. La primera parte es un contexto *intra*, que está formado por varios vecinos del píxel en el mismo *bitplane*. La segunda parte es el contexto *inter*, cuyos píxeles se seleccionan de los planos de bits ya codificados (recuérdese que la codificación se realiza desde el MSBP al LSBP).

Para ver por qué el contexto inter tiene sentido, consideramos una imagen en escala de grises escaneada desde una imagen binivel. La mayoría de los valores de los píxeles serán 255 ($1111\,1111_2$) ó 0 ($0000\,0000_2$), pero algunos píxeles pueden tener valores cercanos a ellos, como 254 ($1111\,1110_2$) ó 1 ($0000\,0001_2$). La Figura 4.103a muestra (en binario) los ocho planos de bits de los seis píxeles 255 255 254 0 1 0. Es evidente que hay correlaciones espaciales entre los planos de bits. Una posición de bit que tiene un cero en un *bitplane* tiende a tener ceros en los otros *bitplanes*. Otro ejemplo es un píxel P con valor "0101xxxx" en una imagen en escala de grises. En principio, los cuatro bits restantes pueden tomar cualquiera de los 16 valores diferentes posibles. En una imagen simple, sin embargo, los cuatro bits restantes tienen sólo unos pocos valores. Por consiguiente, existe una buena posibilidad de que los píxeles adyacentes a P tengan valores similares a P y, por lo tanto, sean buenos predictores de P en todos los planos de bits.

El contexto intra realmente utilizado por EIDAC se muestra en la Figura 4.103b. Está formador por cuatro de los vecinos más cercanos ya vistos del píxel actual X. El contexto inter se muestra en la Figura 4.103c. Consta de (1) los mismos cuatro vecinos del *bitplane* inmediatamente por encima del *bitplane* actual, (2) los mismos cuatro vecinos en el MSBP, y (3) los píxeles en la misma posición que X en todos los planos de bits por encima de él (los cinco píxeles sombreados en la figura). Por lo tanto, un píxel en el MSBP no tiene un contexto inter, mientras que el contexto inter de un píxel en el LSBP consta de $4 + 4 + 7 = 15$ píxeles. Se pueden seleccionar otros contextos, lo que significa que una implementación general de EIDAC debe incluir tres ítems de información de bordes en el *stream* comprimido. El primer ítem contiene las dimensiones de la imagen, el segundo ítem determina la forma en que los contextos son seleccionados, y el tercer ítem es un flag que indica si se utiliza la *compactación del histograma* (si ésta se usa, el flag debe ir seguido por los nuevos códigos). Ésta útil característica se describe aquí.

Una imagen simple es aquella que tiene un pequeño número de colores o escalas de grises de una gran paleta de colores disponibles. Imagine una imagen simple con ocho bits por píxel. Si la imagen tiene tan sólo 27 colores, a cada uno se le puede asignar un código de 5 bits en lugar de los 8 bits originales. Ésta característica se conoce como compactación del histograma. Cuando se utiliza la compactación del histograma, los nuevos códigos para los colores tienen que ser incluidos en el *stream* comprimido, donde constituyen una sobrecarga. Generalmente, la compactación del histograma mejora la compresión, pero en casos raros, la sobrecarga puede ser mayor que el ahorro debido a la compactación del histograma.



(a) Planos de bits (en binario) (b) Contexto intra (c) Contexto inter

Figura 4.103: Contextos de píxeles utilizados por EIDAC.

4.14. Cuantificación vectorial

La cuantificación vectorial es una generalización del método de cuantificación escalar (Sección 1.6). Se utiliza tanto para compresión de imágenes como para la compresión de audio. En la práctica, la cuantificación vectorial es comúnmente utilizada para comprimir datos que han sido digitalizados a partir de una fuente analógica, como muestras de audio e imágenes escaneadas. Éstos datos se denominan *datos analógicos muestreados digitalmente* (*digitally sampled analog data* o DSAD). La cuantificación vectorial se basa en dos hechos:

1. Sabemos (véase la Sección 3.1) que los métodos de compresión que comprimen cadenas, en lugar de símbolos individuales, pueden, en principio, producir mejores resultados.

2. Los ítems adyacentes en una imagen y en el sonido digitalizado están correlacionados. Existe una buena posibilidad de que los vecinos más próximos de un píxel P tengan los mismos valores que P o valores muy similares. Además, las muestras de audio consecutivas raramente difieren en mucho.

Comenzamos con un sencillo e intuitivo método de cuantificación vectorial para la compresión de imágenes. Dada una imagen, la dividimos en pequeños bloques de píxeles, típicamente de 2×2 ó 4×4. Cada bloque se considera un vector. El codificador mantiene una lista (llamada *codebook* o *libro de códigos*) de vectores y comprime cada bloque escribiendo en el *stream* comprimido un puntero al bloque en el *codebook*. El decodificador tiene la sencilla tarea de leer los punteros, siguiendo cada puntero hasta un bloque en el libro de códigos, y añadir el bloque a la imagen generada hasta ese momento. En consecuencia, la cuantificación vectorial es un método de compresión asimétrico.

En el caso de bloques de 2×2, cada bloque (vector) se consta de cuatro píxeles. Si cada píxel es un bit, entonces un bloque es de cuatro bits de longitud y sólo hay $2^4 = 16$ bloques diferentes. Es fácil almacenar este pequeño libro de códigos de forma permanente tanto en el codificador como en el decodificador. Sin embargo, un puntero a un bloque en tal *codebook* es, por supuesto, de cuatro bits de longitud, por lo que no hay ganancia en compresión reemplazando bloques con punteros. Si cada píxel es de k bits, entonces cada bloque es de $4k$ bits de longitud, y hay 2^{4k} bloques diferentes. El libro de códigos crece muy rápido con k (para $k = 8$, por ejemplo, tiene $256^4 = 2^{32} = 4$ Giga entradas), pero la cuestión es que volvemos a sustituir un bloque de bits $4k$ por un puntero de $4k$ bits, lo que no produce ninguna ganancia en compresión. Esto es cierto para bloques de cualquier tamaño.

Una vez que se hace evidente que este sencillo método no funciona, el siguiente paso que viene a la mente es que cualquier imagen dada no puede contener todos los bloques posibles. Dados píxeles de

8 bits, el número de bloques de 2×2 es $2^{2\cdot 2\cdot 8} = 2^{32} \approx 4{,}3$ mil millones, pero cualquier la imagen en particular puede contener sólo unos pocos millones de píxeles y unos pocos miles de bloques diferentes. Por consiguiente, nuestra próxima versión de cuantificación vectorial comienza con un libro de códigos vacío y escanea la imagen bloque a bloque. Para cada bloque se consulta el libro de códigos . Si el bloque ya está en el libro de códigos, el codificador saca un puntero al bloque en el (cada vez más grande) libro de códigos. Si el bloque no está en el libro de códigos, se agrega al mismo y produce como salida un puntero.

El problema con este sencillo método es que cada bloque añadido al *codebook* tiene que ser escrito en la secuencia de datos comprimidos. Ésto reduce considerablemente la eficiencia del método y puede conducir a una baja compresión e incluso a la expansión. También existe la pequeña complicación añadida de que el *codebook* crece durante la compresión, lo que provoca que los punteros sean cada vez de mayor longitud, pero ésto no es difícil de manejar por el decodificador.

Estos problemas son la razón por la que la cuantificación vectorial de una imagen es con pérdidas. Si aceptamos la compresión con pérdida, entonces el tamaño del libro de códigos se puede reducir considerablemente. A continuación, mostramos un método intuitivo con pérdidas para la compresión de imágenes mediante cuantificación vectorial: Se analiza un gran número de imágenes diferentes de "entrenamiento" y se localizan los bloques B más comunes. Se construye un libro de códigos con estos bloques B y se incorporan tanto en el codificador como en el decodificador. Cada entrada del libro de códigos es un bloque. Para comprimir una imagen, se escanea bloque por bloque, y para cada uno de ellos se encuentra la entrada en el libro de códigos que mejor concuerde, y se proporciona como salida un puntero a esa entrada. El tamaño del puntero es, por supuesto, $[\log_2 B]$, por lo que la relación de compresión (que se conoce de antemano) es:

$$\frac{[\log_2 B]}{\text{tamaño del bloque}}.$$

Un problema con este enfoque es cómo hacer coincidir los bloques de la imagen con las entradas del *codebook*. Aquí tenemos algunas medidas comunes. Sea $B = (b_1, b_2, \ldots, b_n)$ y $C = (c_1, c_2, \ldots, c_n)$ un bloque de píxeles de la imagen y una entrada del libro de códigos, respectivamente (cada uno es un vector de n bits). Denotamos la "distancia" entre ellos por $d(B, C)$ y la medimos de tres formas diferentes como sigue:

$$\begin{aligned} d_1(B,C) &= \sum_{i=0}^{n} |b_i - c_i|, \\ d_2(B,C) &= \sum_{i=0}^{n} (b_i - c_i)^2, \\ d_3(B,C) &= \mathrm{MAX}_{i=0}^{n} |b_i - c_i|. \end{aligned} \tag{4.28}$$

La tercera medida $d_3(B, C)$ es fácil de interpretar. Encuentra el componente donde B y C difieren más, y devuelve la diferencia. Las dos primeras medidas son fáciles de visualizar en el caso $n = 3$. La medida $d_1(B, C)$ se convierte en la distancia entre los dos vectores tridimensionales B y C cuando nos movemos a lo largo de los ejes de coordenadas. La medida $d_2(B, C)$ se convierte en la distancia Euclídea (línea recta) entre los dos vectores. Las cantidades $d_i(B, C)$ también pueden considerarse medidas de distorsión.

Otro problema con este enfoque es la calidad del libro de códigos. En el caso de bloques de 2×2 de 8 bits por píxel, el número total de bloques es de $2^{32} \approx 4{,}3$ mil millones. Si decidimos limitar el tamaño de nuestro libro de códigos a, digamos, un millón de entradas, contendrá sólo el 0,023 % del número total de bloques (y todavía son 32 millones de bits, o alrededor de 4 Mb de longitud). La utilización de este libro de códigos para comprimir imágenes "atípicas" puede generar

una gran distorsión independientemente de la medida de distorsión utilizada. Cuando se descomprime la imagen comprimida, puede parecer tan diferente de la original como para que nuestro método sea inútil. Una forma natural de resolver este problema es modificar las entradas originales del *codebook* con el fin de adaptarlas a la imagen en particular que está siendo comprimida. El libro de códigos final tendrá que ser incluido en el flujo comprimido, pero ya que se ha adaptado a la imagen, puede ser lo suficientemente pequeño para producir una buena razón de compresión, pero lo suficientemente cercana a los bloques de imágenes para producir una imagen descomprimida de aspecto aceptable.

Este algoritmo ha sido desarrollado por Linde, Buzo y Gray [Linde, Buzo, y Gray 80]. Se conoce como el algoritmo LBG y que es la base de muchos métodos de cuantificación vectorial para la compresión de imágenes y de sonido (véase, por ejemplo, la Sección 4.33). Sus pasos principales son los siguientes:

- *Paso 0*: Seleccionar un valor umbral ϵ y establecer $k = 0$ y $D^{(-1)} = \infty$. Comenzar con un libro de códigos inicial de $C_i^{(k)}$ entradas (donde k es actualmente cero, pero se irá incrementando en cada iteración). Denotar los bloques de la imagen B_i (estos bloques también se llaman *vectores de entrenamiento*, ya que el algoritmo los utiliza para encontrar las mejores entradas del *codebook*).

- *Paso 1*: Tomar una entrada $C_i^{(k)}$ del libro de códigos. Buscar todos los bloques B_m de la imagen que están más cercanos a C_i que a cualquier otro C_j. Dicho de forma más precisa, encontrar el conjunto de todos los B_m que satisfacen:

$$d\left(B_m, C_i\right) < d\left(B_m, C_j\right) \text{ para todo } j \neq i.$$

  Este conjunto (o *partición*) se denota $P_i^{(k)}$. Repetir el procedimiento para todos los valores de i. Puede suceder que algunas particiones estén vacías; nos ocupamos de este problema más abajo.

- *Paso 2*: Seleccionar una i y calcular la distorsión $D_i^{(k)}$ entre la entrada del libro de códigos $C_i^{(k)}$ y el conjunto de vectores de entrenamiento (partición) $P_i^{(k)}$ encontrados para esa entrada en el paso 1. Repetir este procedimiento para todo i; a continuación, calcular el promedio $D^{(k)}$ de todas las $D_i^{(k)}$. Una distorsión $D_i^{(k)}$ para un cierto i se obtiene calculando las distancias $d\left(C_i^{(k)}, B_m\right)$ para todos los bloques B_m en la partición $P_i^{(k)}$, y después calculando la distancia media. Como alternativa, $D_i^{(k)}$ puede ser ajustado a la mínima de las distancias $d\left(C_i^{(k)}, B_m\right)$.

- *Paso 3*: Si $\left(D^{(k-1)} - D^{(k)}\right)/D^{(k)} \leq \epsilon$, detener el proceso. La salida del algoritmo es el último conjunto de entradas $C_i^{(k)}$ del libro de códigos. Este conjunto puede utilizarse ahora para la compresión (con pérdidas) de la imagen mediante cuantificación vectorial. En la primera iteración k es cero, por lo que $D^{(k-1)} = D^{(-1)} = \infty > \epsilon$. Ésto garantiza que el algoritmo no se detendrá en la primera iteración.

- *Paso 4*: Incrementar k en 1 y calcular nuevas entradas $C_i^{(k)}$ para el libro de códigos; cada una es igual al promedio de los bloques de la imagen (vectores de entrenamiento) en la partición $P_i^{(k-1)}$ que se calculó en el Paso 1. (Así es como se adaptan las entradas del *codebook* a la imagen en particular.) Ir al Paso 1.

Una comprensión completa de este tipo de algoritmo demanda un ejemplo detallado, parte que debe ser resuelta por el lector en forma de ejercicios. Con el fin de facilitar la visualización del ejemplo, suponemos que la imagen a ser comprimida consta de píxeles de 8 bits, y la dividimos en pequeños bloques de dos píxeles. Normalmente, un bloque debe ser cuadrado, pero la ventaja de usar nuestras de bloques de dos píxeles es que pueden ser trazadas en el papel como puntos bidimensionales, haciendo

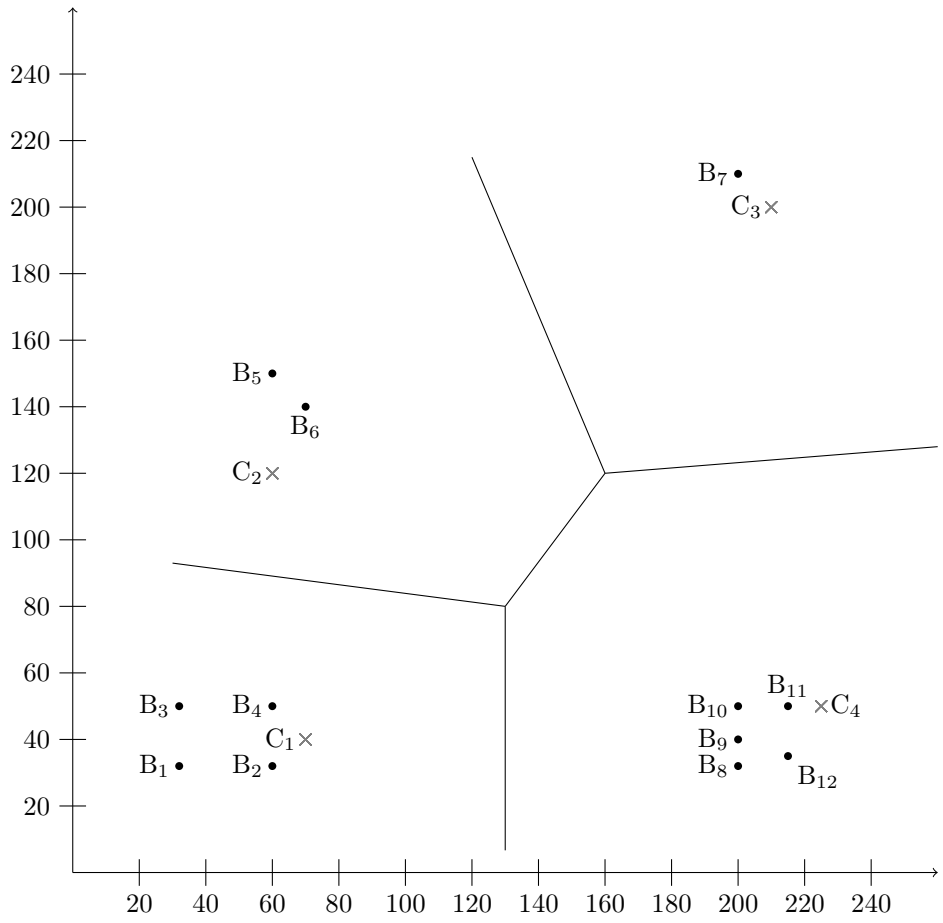


Figura 4.104: Doce puntos y cuatro entradas $C_i^{(0)}$ del libro de códigos.

$$
\begin{array}{rll}
\text{I:} & (70-32)^2+(40-32)^2=1508, & (70-60)^2+(40-32)^2=164,\\
 & (70-32)^2+(40-50)^2=1544, & (70-60)^2+(40-50)^2=200,\\
\text{II:} & (60-60)^2+(120-150)^2=\ 900, & (60-70)^2+(120-140)^2=500,\\
\text{III:} & (210-200)^2+(200-210)^2=\ 200, & \\
\text{IV:} & (225-200)^2+(50-32)^2=\ 949, & (225-200)^2+(50-40)^2=725,\\
 & (225-200)^2+(50-50)^2=\ 625, & (225-215)^2+(50-50)^2=100,\\
 & (225-215)^2+(50-35)^2=\ 325. &
\end{array}
$$

Tabla 4.105: Doce distorsiones para $k=0$.

así los datos (así como el ejemplo completo) más visibles. Ejemplos de bloques son: $(35, 168)$ y $(250, 37)$; interpretamos los dos píxeles de un bloque como las coordenadas (x, y) de un punto.

Nuestro ejemplo asume que la imagen está formada por 24 píxeles, organizados en los 12 bloques: $B_1 = (32, 32)$, $B_2 = (60, 32)$, $B_3 = (32, 50)$, $B_4 = (60, 50)$, $B_5 = (60, 150)$, $B_6 = (70, 140)$, $B_7 = (200, 210)$, $B_8 = (200, 32)$, $B_9 = (200, 40)$, $B_{10} = (200, 50)$, $B_{11} = (215, 50)$, y $B_{12} = (215, 35)$ (Figura 4.104). Es evidente que los 12 puntos se concentran en cuatro regiones. Seleccionamos un libro de códigos inicial con cuatro entradas $C_1^{(0)} = (70, 40)$, $C_2^{(0)} = (60, 120)$, $C_3^{(0)} = (210, 200)$, y $C_4^{(0)} = (225, 50)$ (mostradas como `x` en el diagrama). Estas entradas se han seleccionado más o menos al azar, sin embargo, más adelante mostramos cómo el algoritmo LBG las selecciona metódicamente, una por una. Debido a la naturaleza gráfica de los datos, es fácil determinar las cuatro particiones iniciales. Éstas son: $P_1^{(0)} = (B_1, B_2, B_3, B_4)$, $P_2^{(0)} = (B_5, B_6)$, $P_3^{(0)} = (B_7)$, y $P_4^{(0)} = (B_8, B_9, B_{10}, B_{11}, B_{12})$. La Tabla 4.105 muestra cómo se calcula el promedio de la distorsión $D^{(0)}$ para la primera iteración (usamos la función de distancia euclidiana). El resultado es:

$$D^{(0)} = \frac{(1508 + 164 + 1544 + 200 + 900 + 500 + 200 + 949 + 725 + 625 + 100 + 325)}{12} = 645$$

El Paso 3 indica que no hay convergencia, ya que $D^{(-1)} = \infty$, por lo que incrementamos k en 1 y calculamos las cuatro nuevas entradas $C_i^{(1)}$ del *codebook* (redondeando al entero más cercano para una mayor simplicidad):

$$\begin{aligned} C_1^{(1)} &= (B_1 + B_2 + B_3 + B_4)/4 = (46, 41), \\ C_2^{(1)} &= (B_5 + B_6)/2 = (65, 145), \\ C_3^{(1)} &= B_7 = (200, 210), \\ C_4^{(1)} &= (B_8 + B_9 + B_{10} + B_{11} + B_{12})/5 = (206, 41). \end{aligned} \tag{4.29}$$

Éstas se muestran en la Figura 4.106.

◇ **Ejercicio 4.24 (sol. en pág. 1082):** Realícese la siguiente iteración.

◇ **Ejercicio 4.25 (sol. en pág. 1083):** Utilícense las entradas del libro de códigos de la Ecuación (4.29) para efectuar la siguiente iteración del algoritmo LBG.

A pesar de que el algoritmo reduce la distorsión promedio de una iteración a la siguiente, no garantiza que las entradas del libro de códigos convergan a un conjunto óptimo. Pueden converger a un conjunto casi óptimo, y este aspecto del algoritmo depende en gran medida de la selección inicial de las entradas del libro de códigos (i.e., de los valores $C_i^{(0)}$). Por lo tanto, discutimos a continuación dicho aspecto del algoritmo LBG. El algoritmo LBG original propone una *técnica de división* (*splitting tech.*) computacionalmente intensiva donde las entradas iniciales $C_i^{(0)}$ del libro de códigos son seleccionadas en varios pasos como sigue:

- *Paso 0*: Se establece $k = 1$ y se selecciona un libro de códigos con k entradas (i.e., una entrada) $C_1^{(0)}$ que es un promedio de todos los bloques B_m de la imagen.

- *Paso 1*: En una iteración general habrá k entradas $C_i^{(0)}$, $i = 1, 2, \ldots, k$ en el libro de códigos. Se divide cada entrada (que es un vector) en las dos entradas similares $C_i^{(0)} \pm e$, donde e es un *vector de perturbación* fijo. Se actualiza $k \leftarrow 2k$. (Como alternativa, podemos utilizar los dos vectores $C_i^{(0)}$ y $C_i^{(0)} + e$; una elección que lleva a la distorsión total más pequeña.)

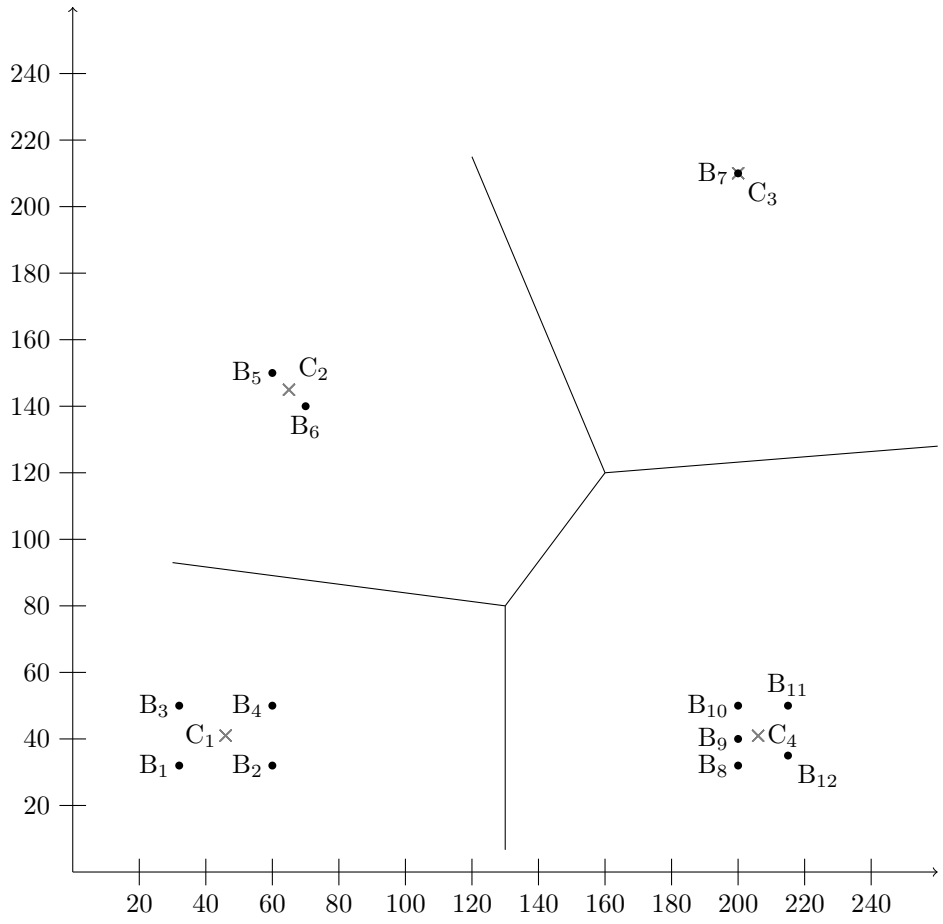


Figura 4.106: Doce puntos y cuatro entradas $C_i^{(1)}$ del libro de códigos.

- *Paso 2*: Si hay suficientes entradas en el libro de códigos, se detiene el proceso de división. El conjunto actual de k entradas del *codebook* puede servir ahora como conjunto inicial $C_i^{(0)}$ para el algoritmo LBG anteriormente expuesto. Si se necesitan más entradas, se ejecuta el algoritmo LBG para el conjunto actual de k entradas, para hacerlas converger a un mejor conjunto; a continuación, se vuelve al paso 1.

Este proceso comienza con una entrada $C_1^{(0)}$ al libro de códigos, que se utiliza para crear dos entradas, luego cuatro, ocho, y así sucesivamente. El número de entradas es siempre una potencia de 2. Si es necesario un número diferente, entonces se ejecuta por última vez el Paso 1; éste puede dividir sólo algunas entradas. Por ejemplo, si se necesitan 11 entradas en el libro de códigos, entonces el proceso de división se repite hasta haber calculado ocho entradas. Se invoca, a continuación, el Paso 1 para dividir sólo tres de las ocho entradas, y terminar con 11 entradas.

Es necesario aclarar una característica más del algoritmo LBG. El Paso 1 del algoritmo dice: "Localizar todos los bloques B_m de la imagen que están más cercanos a C_i que a cualquier otro C_j. Ellos se convierten en la partición $P_i^{(k)}$." Puede suceder que ningún bloque de la imagen esté cercano a una entrada determinada C_i del libro de códigos, lo que crea una partición vacía $P_i^{(k)}$. Ésto es un problema, porque la media de los bloques de imagen incluidos en la partición $P_i^{(k)}$ se utilizan para

$$
\begin{array}{rll}
\text{I:} & (46-32)^2+(41-32)^2=277, & (46-60)^2+(41-32)^2=277,\\
 & (46-32)^2+(41-50)^2=277, & (46-60)^2+(41-50)^2=277,\\
\text{II:} & (65-60)^2+(145-150)^2=\ 50, & (65-70)^2+(145-140)^2=\ 50,\\
\text{III:} & (210-200)^2+(200-210)^2=200, & \\
\text{IV:} & (206-200)^2+(41-32)^2=117, & (206-200)^2+(41-40)^2=\ 37,\\
 & (206-200)^2+(41-50)^2=117, & (206-215)^2+(41-50)^2=162,\\
 & (206-215)^2+(41-35)^2=117. &
\end{array}
$$

Tabla 4.107: Doce distorsiones para $k=1$.

calcular una mejor entrada para el libro de códigos en la siguiente iteración. Una buena solución es eliminar la entrada C_i del *codebook* y reemplazarla con una nueva entrada elegida al azar de uno de los bloques de la imagen incluidos en la mayor partición $P_j^{(k)}$.

Si un bloque de imagen es un vector n-dimensional, entonces el proceso de construcción de las particiones en el Paso 1 del algoritmo LBG divide el espacio n-dimensional en regiones de Voronoi [Salomon 99] con una entrada C_i al libro de códigos en el centro de cada región. Las Figuras 4.105 y 4.106 muestran las regiones de Voronoi en las dos primeras iteraciones de nuestro ejemplo. En cada iteración, las entradas del libro de códigos se mueven, cambiando así las regiones de Voronoi.

VQ con estructura de árbol: El algoritmo LBG es computacionalmente intensivo, porque el cálculo de las particiones en el Paso 1 requiere muchas comparaciones. Cada bloque B_m de la imagen tiene que ser comparado con todas las entradas C_j del libro de códigos a fin de encontrar la entrada C_i más cercana. Por consiguiente, un algoritmo sencillo requiere n pasos por cada bloque de la imagen, donde n es el número de entradas del *codebook*. La estructura de árbol aquí descrita reduce esta complejidad a $\log_2 n$ pasos por cada bloque de la imagen. Esta variante se denomina *cuantificación vectorial con estructura de árbol* (*tree-structured vector quantization* o TSVQ). La idea es utilizar los bits individuales de los píxeles que constituyen un bloque de la imagen para localizar el bloque en el espacio. En nuestro ejemplo, cada bloque consta de dos píxeles de 8 bits que se considera de forma natural las coordenadas (x,y) del bloque en el espacio bidimensional. En general, un bloque está formado por k píxeles, y puede verse como un punto en el espacio k-dimensional.

Dividimos el plano en cuatro cuadrantes numerados 00, 01, 10 y 11 como muestra la Figura 4.108a. Los bits más significativos de las coordenadas x e y de un punto determinan su cuadrante. Por lo tanto, todos los bloques de la imagen con valores $(0\ldots,0\ldots)$ se encuentran en el cuadrante 00, todos los bloques de la imagen de la forma $(1\ldots,0\ldots)$ se encuentra en el cuadrante 10, y así sucesivamente. Ahora dividimos cada cuadrante en cuatro subcuadrantes y no es difícil ver cómo los segundos bits más significativos de las coordenadas de un punto determinan su subcuadrante. La Figura 4.108b muestra los cuatro subcuadrantes del cuadrante 00 y los dos primeros bits más significativos de los puntos ubicados en ellos. De forma similar, la Figura 4.108c muestra los cuatro subsubcuadrantes del subcuadrante 10 y cómo los tres bits más significativos de un punto determinan su subsubcuadrante.

Puesto que nuestros puntos (bloques de la imagen) son coordenadas de 8 bits, podemos señalar la localización de cada punto bajando hasta un subcuadrante de orden 8. Ahora construimos un *quadtree* (Sección 4.30) de profundidad ocho, donde cada hoja es asociada con una entrada del libro de códigos. Dado un bloque $B_m=(x,y)$ de la imagen, utilizamos pares de bits consecutivos a partir de x e y para bajar por el árbol. Cuando llegamos a una hoja, colocamos B_m en la partición de la entrada del libro de códigos encontrada en esa hoja. Como ejemplo, consideremos el bloque B_4 de la imagen vista anteriormente. Sus coordenadas son $(60,50)=(00111100,00110010)$. Los ocho pares de bits tomados, de izquierda a derecha, a partir de las dos coordenadas son: $(0,0)$, $(0,0)$, $(1,1)$, $(1,1)$, $(1,0)$, $(1,0)$, $(0,1)$, y $(0,0)$. Ellos determinan cuál de las cuatro ramas del quadtree debe ser tomada en cada uno de los ocho pasos según bajamos por el árbol. Cuando llegamos a la parte inferior, nos encontramos

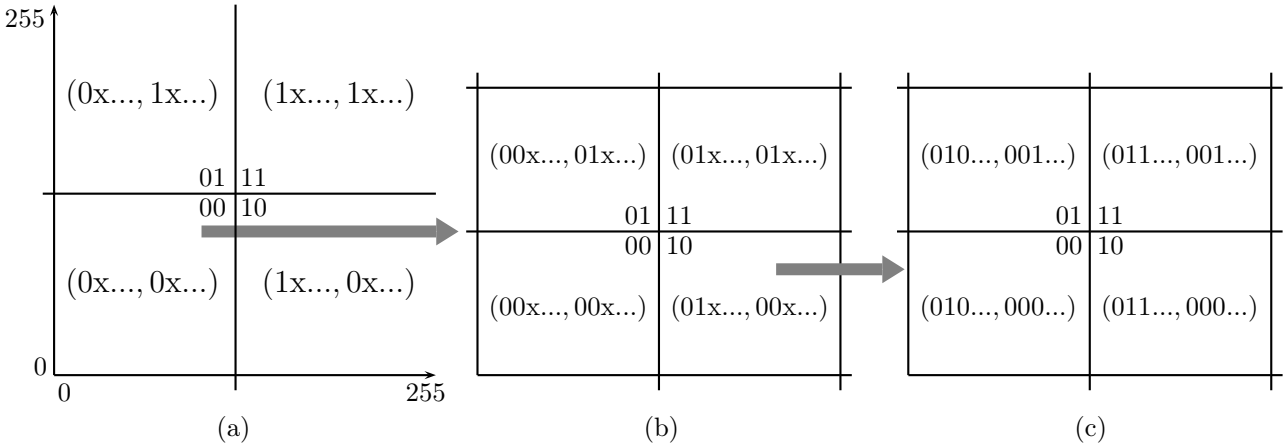


Figura 4.108: Numeración de cuadrantes.

con la entrada C_1 del libro de códigos, por lo que incluimos el bloque B_4 de la imagen en la partición P_1.

◇ **Ejercicio 4.26 (sol. en pág. 1083):** ¿Cuál es el área de un subcuadrante de orden 8?

4.15. Cuantificación vectorial adaptativa

El método de cuantificación vectorial básico de la Sección 4.14 utiliza, o bien un libro de códigos fijo, o bien un algoritmo LBG (o similar) para construir el libro de códigos a medida que avanza. En todos estos casos el libro de códigos está formado por entradas de tamaño fijo, idénticas en tamaño al bloque de imagen. El método adaptativo que se describe aquí, debido a Constantinescu y Storer [Constantinescu y Storer 94a,b], utiliza bloques de imagen y entradas del libro de códigos de tamaño variable. El libro de códigos se llama *diccionario*, ya que este método tiene un ligero parecido con los distintos métodos LZ. El método combina una pasada única y un diccionario adaptativo utilizado por diversos algoritmos basados en diccionarios, con las medidas de distancia utilizadas por los diferentes métodos de cuantificación vectorial, para obtener buenas aproximaciones de los bloques de datos.

En cada paso del proceso de codificación, el codificador selecciona un bloque de imagen (un rectángulo de cualquier tamaño), que coincida con una de las entradas del diccionario, emite un puntero a dicha entrada y actualiza el diccionario mediante la adición de una o más entradas al mismo. Las nuevas entradas están basadas en el bloque de imagen actual. El caso de un diccionario completo tiene que ser tratado, y se discute a continuación.

Algunos autores se refieren a tal método como *substitución textual*, ya que sustituye los punteros de los datos originales.

Una amplia experimentación por los desarrolladores ha producido razones de compresión que igualan o incluso superan a los de JPEG. Al mismo tiempo, el método es rápido (ya que realiza una sola pasada), sencillo (comienza con un diccionario vacío, por lo que no requiere entrenamiento), y fiable (la cantidad de pérdidas se controla fácilmente y con precisión por un único parámetro de tolerancia). El funcionamiento del decodificador es similar al de un decodificador de cuantificación vectorial, por lo que es rápido y sencillo.

El codificador tiene que seleccionar los bloques de imagen cuidadosamente, asegurándose de que cubran toda la imagen, un cubrimiento que el decodificador pueda imitar, y en el que no haya un

excesivo solapamiento entre los distintos bloques. Con el fin de seleccionar los bloques, el codificador elige uno o más *puntos de crecimiento* en cada iteración, y utiliza uno de ellos en la siguiente iteración como esquina de un nuevo bloque de imagen. El bloque comienza siendo pequeño y se incrementa en tamaño (se "hace crecer" desde la esquina) por el codificador, mientras que pueda ser emparejado con un entrada del diccionario. Un punto de crecimiento se denota G y es un triplete (x, y, q), donde x e y son las coordenadas del punto y q indica la esquina del bloque de imagen donde el punto está localizado. Suponemos que los valores de q: 0, 1, 2, y 3 identifican las esquinas superior izquierda, superior derecha, inferior izquierda e inferior derecha, respectivamente (por consiguiente, q es un número de 2 bits). Un bloque de imagen B anclada en un punto de crecimiento G se denota $N = (G, w, h)$, donde w y h son la anchura y la altura del bloque, respectivamente.

Enumeramos los principales pasos del codificador y del decodificador, y luego tratamos cada uno en detalle. Los principales pasos del codificador son los siguientes:

- *Paso 1*: Inicializa el diccionario de D a todos los posibles valores de los píxeles de la imagen. inicializa el conjunto de puntos de crecimiento (GPP) a uno o más puntos de crecimiento utilizando uno de los algoritmos que se discuten a continuación.

- *Paso 2*: Repite los pasos 3−7 hasta que el GPP esté vacío.

- *Paso 3*: Utiliza un algoritmo de crecimiento para seleccionar un punto de crecimiento G del GPP.

- *Paso 4*: Hace crecer un bloque de imagen B usando G como su esquina. Utiliza un algoritmo de comparación para emparejar a B, a medida que va creciendo, con una entrada de un diccionario de fidelidad controlado por el usuario.

- *Paso 5*: Una vez que B ha alcanzado el tamaño máximo, donde todavía puede emparejarse con una entrada del diccionario d, emite un puntero a d. El tamaño del puntero depende del tamaño (número de entradas) de D.

- *Paso 6*: Elimina G del GPP y utiliza un algoritmo para decidir qué nuevos puntos de crecimiento (si los hubiera) agrega al GPP.

- *Paso 7*: Si D está lleno, utiliza un algoritmo para eliminar una o más entradas. Usa un algoritmo para actualizar el diccionario basado en B.

El funcionamiento del codificador depende, por lo tanto, de varios algoritmos. Cada algoritmo debe ser desarrollado e implementado, y su rendimiento, probado con muchas imágenes de ensayo. El decodificador es mucho más simple y rápido. Sus principales pasos son los siguientes:

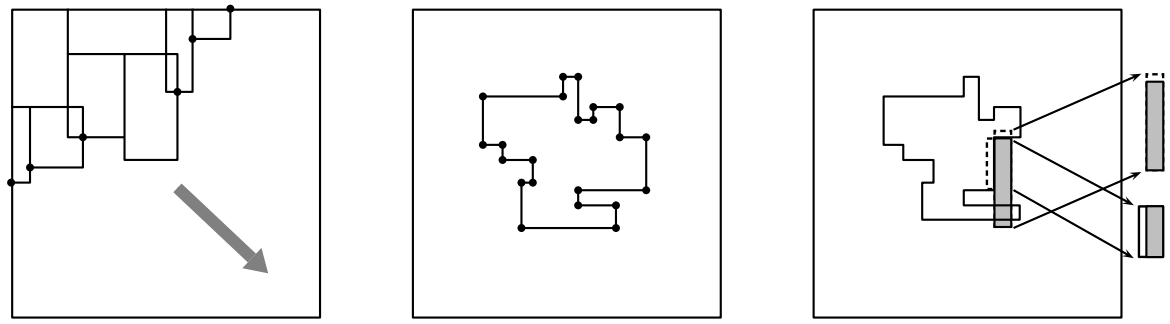
- *Paso 1*: Inicializa el diccionario D y el GPP como en el Paso 1 del codificador.

- *Paso 2*: Repite los pasos 3−5 hasta que el GPP está vacío.

- *Paso 3*: Utiliza el algoritmo de crecimiento del codificador para seleccionar un punto de crecimiento G del GPP.

- *Paso 4*: Introduce un puntero desde el *stream* comprimido, y lo usa para obtener una entrada d del diccionario; coloca d en la ubicación y la posición especificada por G.

- *Paso 5*: Actualiza D y el GPP como en los Pasos 6−7 del codificador.



(a) (b) (c)

Figura 4.109: (a) Cobertura en onda. (b) Puntos de crecimiento. (c) Actualización del diccionario.

El resto de esta sección describe los distintos algoritmos necesarios, y muestra un ejemplo.

Algoritmo: Selección de un punto de crecimiento G del GPP. Los métodos más sencillos son LIFO (que probablemente no produce buenos resultados) y FIFO (que se utiliza en el ejemplo de más abajo); pero los métodos de cobertura descritos aquí pueden ser mejores, ya que determinan cómo será cubierta la imagen con bloques.

Cobertura en onda: Este algoritmo selecciona de entre todos los puntos de crecimiento en el GPP, el punto $G = (x_s, y_s, 0)$ que satisface:

$$x_s + y_s \leq x + y, \quad \text{para cualquier } G\,(x, y, 0) \text{ en el } GPP$$

(observe que el punto de crecimiento seleccionado tiene $q = 0$, por lo que se encuentra en la esquina superior izquierda de un bloque de imagen). Cuando se utiliza este algoritmo, tiene sentido inicializar el GPP con un solo punto, la esquina superior izquierda de la imagen. Si se hace ésto, entonces el algoritmo de cobertura en onda selecciona bloques de imágenes que cubren la imagen en una onda que se mueve desde la esquina superior izquierda a la esquina inferior derecha de la imagen (Figura 4.109a).

Cobertura circular: Selecciona aquel punto de crecimiento G cuya distancia al centro de la imagen es mínima entre todos los puntos de crecimiento G en el GPP. Ésto cubre la imagen con bloques ubicados en una región circular de crecimiento alrededor del centro (Figura 4.109b).

Cobertura Diagonal: Este algoritmo selecciona de entre todos los puntos de crecimiento en el GPP, el punto $G = (x_s, y_s, q)$ que satisface:

$$|x_s - y_s| \leq |x - y|, \quad \text{para cualquier } G\,(x, y, p) \text{ en el } GPP$$

(observe que el punto de crecimiento seleccionado puede tener cualquier q). El resultado será que los bloques comienzan alrededor de la diagonal principal y se alejan de ella en dos ondas que se ejecutan en paralelo a la misma.

Algoritmo: Correspondencia de un bloque de imagen B (con un punto de crecimiento G en su esquina), con una entrada del diccionario. Aparentemente el mejor enfoque es comenzar con el bloque más pequeño B (de sólo un píxel) y probar el emparejamiento con bloques cada vez más grandes con las entradas del diccionario, hasta que con el siguiente incremento de B no se encuentre ninguna entrada en el diccionario que coincida con B, según la tolerancia especificada por el usuario. Los siguientes parámetros controlan el emparejamiento:

1. La medida de la distancia. Se puede utilizar cualquiera de las medidas propuestas en la Ecuación (4.28).

2. La tolerancia. Un parámetro real t definido por el usuario, que es la distancia máxima permitida entre un bloque de imagen B y una entrada del diccionario.

3. El tipo de cobertura (recubrimiento). Puesto que los bloques de imágenes tienen tamaños diferentes y crecen desde los puntos de crecimiento en distintas direcciones, pueden superponerse (Figura 4.109a). Imagine que un bloque de imagen B ha sido emparejado, pero que cubre en parte algunos de los otros bloques. Podemos calcular la distancia usando sólo las partes de B que no se superpongan a los otros bloques. Esto significa que el primer bloque que cubre un área determinada de la imagen determinará la calidad de la imagen descomprimida en esa zona. Ésto puede llamarse *primera cobertura*. Lo opuesto es la *última cobertura*, donde la distancia entre un bloque de imagen B y una entrada del diccionario se calcula utilizando todo B (excepto las partes que se encuentran fuera de la imagen), independientemente de cualesquiera superposiciones. También es posible tener una *cobertura media*, donde la distancia se calcula para todo B, pero en una región de solapamiento, cualquier valor de píxel utilizado para calcular la distancia será el promedio de los valores de los píxeles de todos los bloques de imagen de la región.

4. El tamaño del subbloque elemental l. Éste es una "normalización" del parámetro de compensación de los distintos tamaños de bloque. Sabemos que los bloques de imagen pueden tener diferentes tamaños. Puede suceder que un gran bloque coincida con una entrada de diccionario cumpliendo la tolerancia, mientras que algunas regiones dentro del bloque tengan una escasa coincidencia (y otras regiones tengan una muy buena correspondencia). Un bloque de imagen, por tanto, debería dividirse en subbloques de tamaño $l \times l$ (donde el valor de l depende del tamaño de la imagen, pero es típicamente 4 ó 5), y cada entrada del diccionario debería dividirse de forma similar. El algoritmo de coincidencia debe calcular la distancia entre cada subbloque de imagen y la entrada del subbloque correspondiente del diccionario. Debería seleccionarse el máximo de estas distancias como la distancia entre el bloque de imagen y la entrada del diccionario.

Algoritmo: Actualización del GPP. Una buena política es elegir como nuevos puntos de crecimiento aquellos puntos ubicados en o cerca de los bordes de la imagen parcialmente codificada (Figura 4.109b). Ésto hace que los nuevos bloques de imagen estén adyacentes a los antiguos. Esta política tiene sentido, porque un bloque antiguo, ha contribuido a formar las nuevas entradas del diccionario, y esas entradas son los candidatos perfectos para el emparejamiento con un nuevo bloque adyacente; porque los bloques adyacentes, en general, no difieren mucho. Inicialmente, cuando la imagen parcialmente codificada es pequeña, está formada principalmente por bloques de un único píxel; esta táctica agrega dos puntos de crecimiento al GPP para cada pequeño bloque añadido a la imagen codificada. Ésos son los puntos situados por debajo y a la derecha del bloque de imagen (véase el ejemplo de más abajo). Otra buena localización de los nuevos puntos de crecimiento es cada esquina cóncava nueva que se haya generado mediante la adición del bloque de imagen reciente a la imagen parcialmente codificada (Figura 4.109a,b). A continuación mostramos que cuando un nuevo bloque de imagen crece desde un punto de esquina, contribuye con dos o tres nuevas entradas en el diccionario.

Algoritmo: Actualización del diccionario. Ésta se basa en dos principios: (1) Cada bloque emparejado añadido a la imagen actual debe ser usado para añadir una o más entradas al diccionario; (2) las nuevas entradas agregadas deben contener píxeles de la imagen actual codificada (por lo que el decodificador puede actualizar su diccionario de la misma manera). Denotando el bloque emparejado actual B, una táctica simple consiste en añadir al diccionario los dos bloques obtenidos mediante la adición de una columna de píxeles a la izquierda de B y una fila de píxeles por encima de él, respectivamente. A veces, como muestra la Figura 4.109c, la fila o columna se puede añadir a sólo una parte de B. Experimentos posteriores con el método, agregan un tercer bloque al diccionario: el bloque obtenido mediante la adición de la columna a la izquierda de B *y* de la fila por encima de B. Puesto

que los bloques de imagen B son inicialmente pequeños y cuadrados, esta tercera entrada de diccionario es también cuadrada. La adición de este tercer bloque mejora el rendimiento de la compresión significativamente.

Algoritmo: Supresión del diccionario. El diccionario sigue creciendo, pero sólo puede crecer hasta cierto punto. Cuando alcanza su tamaño máximo, podemos seleccionar alguna de las siguientes ideas: (1) Eliminar la entrada menos utilizada recientemente (LRU) (excepto cuanto a las entradas originales, todos los posibles valores de píxel, no deben ser eliminados), (2) congelar el diccionario (i.e., no añadir nuevas entradas), y (3) eliminar el diccionario completo (excepto las entradas originales) y comenzar de nuevo.

◇ **Ejercicio 4.27 (sol. en pág. 1084):** Sugiérase otro enfoque para la eliminación del diccionario. (Sugerencia: Véase la Sección 3.18).

Los experimentos realizados por los desarrolladores del método sugieren que el algoritmo básico es robusto y no depende tanto de la elección particular de los algoritmos anteriores, a excepción de dos opciones que parecen mejorar el rendimiento de la compresión significativamente:

1. La cobertura en onda parece ofrecer un mejor rendimiento que la cobertura circular o en diagonal.
2. La inspección visual de muchas imágenes descomprimidas muestra que la pérdida de datos es más visible para el ojo en las regiones de la imagen que son lisas (i.e., uniformes o casi uniformes). El método, por lo tanto, se puede mejorar utilizando un parámetro de tolerancia más pequeño para tales regiones. Una sencilla medida A de la lisura es la relación V/H, donde V es la varianza de los píxeles en la región (véase la página 432 para la definición de varianza) y M es su media. Cuanto mayor es A, más activa (i.e., menos lisa) es la región. La tolerancia t es determinada por usuario, y los desarrolladores del método sugieren el uso de valores de tolerancia más pequeños para las regiones lisas, como los siguientes:

$$\text{si}\begin{cases} A \leq 0{,}05, & \text{utilice } 0{,}4t, \\ 0{,}05 < A \leq 0{,}1, & \text{utilice } 0{,}6t, \\ A > 0{,}1, & \text{utilice } t. \end{cases}$$

Ejemplo: Seleccionamos una imagen formada por píxeles de 4 bits. Los nueve píxeles de la esquina superior izquierda de esta imagen se muestran en la Figura 4.110a. Tienen coordenadas de imagen que van desde $(0,0)$ a $(2,2)$. Las primeras 16 entradas del diccionario D (ubicaciones de 0 a 15) se inicializan con los 16 posibles valores de los píxeles. El GPP se inicializa a la esquina superior izquierda de la imagen, la posición $(0,0)$. Asumimos que el GPP se utiliza como una cola (FIFO). Aquí están los detalles de los primeros pasos:

- *Paso 1*: El punto $(0,0)$ es sacado del GPP. El valor del píxel en esta posición es 8. Puesto que el diccionario contiene sólo los píxeles individuales, no los bloques, éste punto puede ser sólo comparado con la entrada del diccionario que contiene 8. Esta entrada pasa a estar ubicada en la localización 8 el diccionario, por lo que el codificador envía el puntero 8. Este emparejamiento no tiene ninguna esquina cóncava, por consiguiente empujamos el punto hacia la derecha del bloque emparejado, $(0,1)$, y el punto por debajo de él, $(1,0)$, en el GPP.
- *Paso 2*: El punto $(1,0)$ es sacado del GPP. El valor del píxel en esta posición es 4. El diccionario todavía contiene sólo píxeles individuales, ningún bloque más grande, por lo que este punto sólo puede ser comparado con la entrada del diccionario que contiene 4. Esta entrada se encuentra en la localización 4 del diccionario, por tanto, el codificador envía el puntero 4. El emparejamiento no tiene esquinas cóncavas, por consiguiente empujamos el punto hacia la derecha del bloque

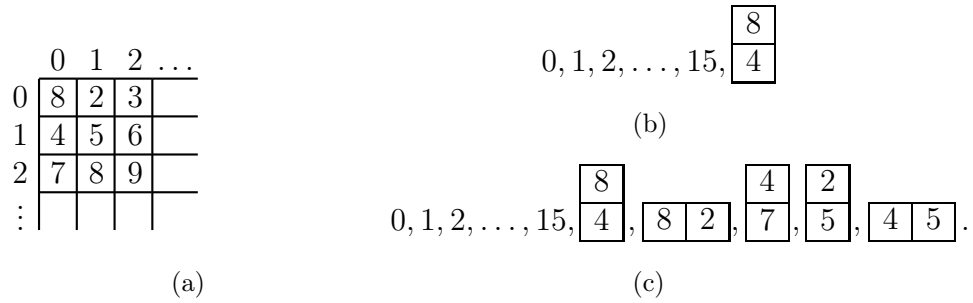


Figura 4.110: Una imagen y un diccionario.

emparejado, $(1,1)$, y el punto por debajo de él, $(2,0)$, en el GPP. El GPP contiene ahora (del más, al menos reciente) los puntos: $(1,1)$, $(2,0)$ y $(0,1)$. El diccionario se actualiza mediante la adición al mismo (en la localización 16), del bloque $\begin{array}{|c|}\hline 8\\\hline 4\\\hline\end{array}$, como se muestra en la Figura 4.110b.

- *Paso 3*: El punto $(0,1)$ es sacado del GPP. El valor del píxel en esta posición es 2. El diccionario contiene los 16 valores de píxel individuales y un bloque más grande. La mejor correspondencia para este punto es con la entrada del diccionario que contiene 2, que está en la localización 2 del diccionario, por lo que el codificador envía el puntero 2. El emparejamiento no tiene ninguna esquinas cóncava, por consiguiente empujamos el punto hacia la derecha del bloque emparejado, $(0,2)$, y el punto por debajo de él, $(1,1)$, en el GPP. El GPP contiene ahora (del más reciente al más antiguo) los puntos: $(0,2)$, $(1,1)$, y $(2,0)$. El diccionario se actualiza añadiendo al mismo (en la localización 17) el bloque $\begin{array}{|c|c|}\hline 8&2\\\hline\end{array}$ (Figura 4.110c).

◇ **Ejercicio 4.28 (sol. en pág. 1084):** Realízense los dos pasos siguientes.

4.16. Emparejamiento de bloques

El método LZ77 de ventana deslizante para la compresión de texto (Sección 3.3) se puede aplicar también a imágenes. En esta sección se describe como una aplicación para la compresión sin pérdidas de imágenes. Sigue las ideas expuestas en [Storer y Helfgott 97]. La Figura 4.111a muestra cómo se pueden hacer deslizar en orden *raster*[13] un buffer de búsqueda y un buffer de preanálisis a lo largo de una imagen. El principio de la compresión de imágenes (Sección 4.1) dice que los vecinos de un píxel P tienden a tener el mismo valor que P o valores muy similares. Así, si P es el píxel del extremo izquierdo en el buffer de preanálisis, hay una buena posibilidad de que algunos de sus vecinos a su izquierda (es decir, en el buffer de búsqueda) tengan el mismo valor que P. También hay una posibilidad de que una cadena de píxeles vecinos en el buffer de búsqueda coincida con P y la cadena de píxeles que le siguen en el buffer de preanálisis. La experiencia también sugiere que en cualquier imagen no aleatoria hay una buena posibilidad de tener cadenas idénticas de píxeles en diferentes ubicaciones en la imagen.

[13] De arriba abajo y de izquierda a derecha.

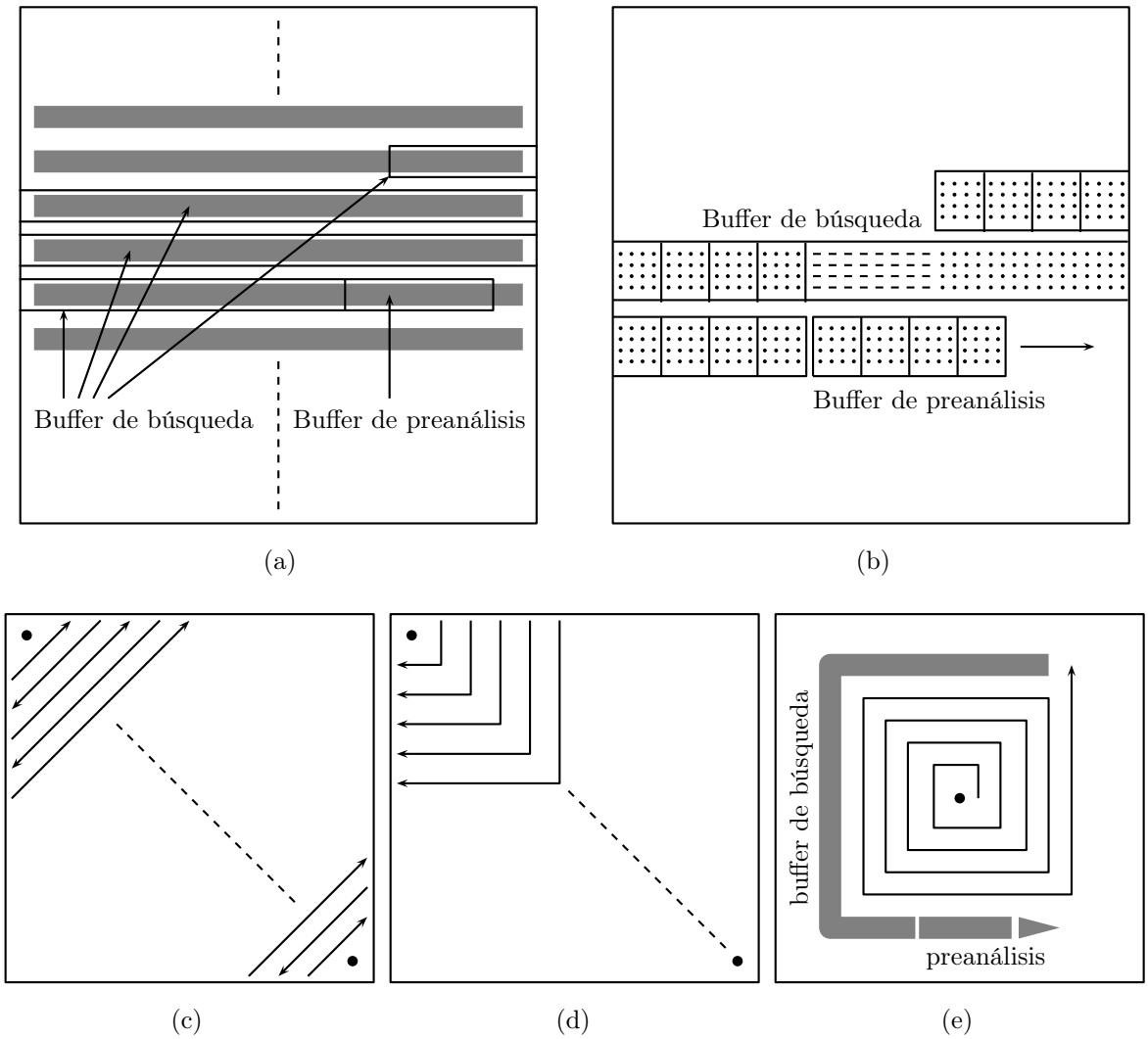


Figura 4.111: Emparejamiento de bloques (LZ77 para imágenes).

Puesto que los vecinos cercanos de un píxel se encuentran también por encima y por debajo del mismo y no sólo a su izquierda y a su derecha, tiene sentido utilizar búsquedas "de mayor ámbito" y búferes de preanálisis, y comparar los bloques de, digamos, 4×4 píxeles en lugar de píxeles individuales (Figura 4.111b). Ésta es la razón del nombre *emparejamiento de bloques*. Cuando se utiliza este método, el número de filas y columnas de la imagen debe ser divisible por 4. Si la imagen no satisface ésto, deben añadirse hasta tres filas y/o columnas artificiales. Dos *convenciones sobre bordes* razonables son: (1) agregar las columnas de píxeles cero a la izquierda y las filas de píxeles cero a la parte superior de la imagen, (2) duplicar la columna del extremo derecho y la fila inferior tantas veces como sea necesario.

La Figura 4.111c,d,e muestra tres otras formas de deslizamiento de ventana en la imagen. Las transversales en la diagonal de $-45^{\rm o}$ visitan los píxeles en el orden: $(1,1)$, $(1,2)$, $(2,1)$, $(1,3)$, $(2,2)$, $(3,1)$,.... En las transversales rectilíneas los píxeles se recorren en el orden: $(1,1)$, $(1,2)$, $(2,2)$, $(2,1)$, $(1,3)$, $(2,3)$, $(3,3)$, $(3,2)$, $(3,1)$,.... La transversal circular visita los píxeles en orden creciente de

distancia desde el centro.

El codificador procede como en el algoritmo LZ77. Encuentra todos los bloques en el buffer de búsqueda que coinciden con el bloque actual de 4×4 en el buffer de preanálisis y selecciona el emparejamiento más largo. Emite un token (i.e., lo escribe en el *stream* comprimido), y avanza en los dos búferes. En el LZ77 original el token está formado por un desplazamiento (distancia u *offset*), la longitud del emparejamiento, y el símbolo siguiente en el buffer de preanálisis (y los búferes avanzan en una más la longitud del emparejamiento). El tercer ítem garantiza que se hagan progresos, incluso en los casos en que no encuentra ninguna correspondencia. Cuando LZ77 se extiende a imágenes, el tercer ítem de un token es el bloque de píxeles siguiente en el buffer de preanálisis. El método, por lo tanto, debe modificarse para emitir los tokens de salida sin éste tercer ítem, y aquí se describen tres maneras de hacer ésto.

◇ **Ejercicio 4.29 (sol. en pág. 1084):** ¿Qué problema hay en la inclusión del bloque de píxeles siguiente en el token?

1. El *stream* de salida está formado por tokens de dos campos (desplazamiento, longitud). Cuando no se encuentra ninguna coincidencia, se emite un token de la forma $(0, 0)$, seguido por el bloque de píxeles no emparejado sin modificar (ésto es común en el inicio del proceso de compresión, cuando el buffer de búsqueda está vacío o casi). El decodificador puede imitar fácilmente ésto.

2. El *stream* de salida está formado por referencias (etiquetas o *tags*) de tamaño variable, algunas de las cuales van seguidas por un token de 2 campos o por un bloque sin modificar de 16 píxeles. Este método es especialmente adecuado para las imágenes binivel, donde un píxel se representa mediante un único bit. Sabemos por experiencia que en una imagen binivel "típica" hay más áreas uniformes blancas que negras (el lector dubitativo debería verificar el octavo de los documentos de adiestramiento de fax de la UIT-T, que se listan en la Tabla 2.40, como ejemplo de imagen atípica). A la etiqueta, por lo tanto, se le puede asignar uno de los siguientes cuatro valores: 0 si el bloque actual en el buffer de preanálisis es completamente blanco, 10 si el bloque actual es completamente negro, 110 si se encontró una coincidencia (en cuyo caso la etiqueta va seguida por un desplazamiento y una longitud, codificados ya sea en tamaño fijo, ya sea en tamaño variable), y 111 si no se encontró ninguna coincidencia (en este caso la etiqueta va seguida por el bloque de píxeles de 16 bits sin modificar).

3. Ésta es similar a la 2 de arriba, pero al descubrir que el bloque actual es completamente blanco o completamente negro, el codificador busca una racha (*run*) en tales bloques. Si encuentra n bloques consecutivos completamente blancos, genera una etiqueta 0 seguida por el valor codificado de n. Similarmente, una etiqueta 10 precede a una cuenta codificada de un *run* de bloques de píxeles completamente negros. Dado que los valores grandes de n son poco frecuentes, tiene sentido utilizar un código de tamaño variable, tal como alguno de los códigos unarios generales de la Sección 2.3.1. Un codificador sencillo puede utilizar siempre el mismo código unario, por ejemplo, el código $(2, 1, 10)$, que tiene 2044 valores. Un codificador sofisticado puede utilizar la resolución de la imagen para estimar el tamaño máximo de un *run* de bloques idénticos, seleccionar un valor adecuado para k basado en éste, y utilizar el código $(2, 1, k)$. El valor de k que está siendo utilizado debería ser incluido en la cabecera del archivo comprimido, para uso del decodificador. La Tabla 4.112 muestra el número de códigos de cada uno de los códigos unarios generales $(2, 1, k)$ para $k = 2, 3, \ldots, 11$. Ésta tabla fue calculada mediante la sentencia `Table[2^(k+1)-4,{k,2,11}]` en *Mathematica*.

El desplazamiento puede ser un solo número: la distancia de la cadena coincidente de bloques de píxeles desde borde del buffer de búsqueda. Ésto tiene la ventaja de que el valor máximo de la distancia depende del tamaño del buffer de búsqueda. Sin embargo, ya que el buffer de búsqueda

k :	2	3	4	5	6	7	8	9	10	11
$(2,1,k)$:	4	12	28	60	124	252	508	1020	2044	4092

Tabla 4.112: Número de códigos unarios generales $(2,1,k)$ para $k = 2, 3, \ldots, 11$.

puede serpentear por la imagen de una manera compleja, el decodificador puede tener que trabajar duro para determinar las coordenadas de la imagen a partir de la distancia. Una alternativa es utilizar como desplazamiento las coordenadas de la imagen (fila y columna) de la cadena emparejada en lugar de su distancia. Ésto facilita al decodificador encontrar la cadena coincidente de bloques de píxeles, pero disminuye el rendimiento de la compresión, ya que la fila y la columna pueden ser números grandes y el desplazamiento no depende de la distancia real (los desplazamientos para una distancia corta y para una larga podrían requerir el mismo número de bits). Un ejemplo puede arrojar luz sobre este punto. Imagine una imagen con una resolución de $1\text{K} \times 1\text{K}$. El número de filas y de columnas son de 10 bits cada uno, por consiguiente, la codificación de un desplazamiento como el par (fila, columna) requiere 20 bits. Un codificador sofisticado, sin embargo, podría decidir, basándose en el tamaño de la imagen, seleccionar un buffer de búsqueda con $256\text{K} = 2^{18}$ posiciones (el 25 % del tamaño de la imagen). La codificación de los desplazamientos, así como de las distancias en el buffer de búsqueda requiere 18 bits, una pequeña ganancia.

4.16.1. Detalles de la implementación

El método aquí descrito para encontrar un emparejamiento es sólo uno de muchos posibles métodos. Es sencillo y rápido, pero no garantiza encontrar la mejor coincidencia. Es similar al método utilizado por LZP (Sección 3.16) para encontrar coincidencias. Observe que la búsqueda de emparejamientos es una tarea que realiza el codificador. La tarea del decodificador es utilizar el token para localizar una cadena de bloques de píxeles, y ésto es sencillo.

El codificador comienza con el bloque de píxeles actual B en el buffer de preanálisis. Realiza una operación hash con los valores de los píxeles de B para crear un puntero p a una tabla de índices T. En $T[p]$ el codificador normalmente encuentra un puntero q apuntando al buffer de búsqueda S. Compara el bloque B con $S[q]$. Si son idénticos, el codificador encuentra el emparejamiento más largo. Si B y $S[q]$ son diferentes, o si $T[p]$ es inválido (no contiene un puntero al buffer de búsqueda), no hay ninguna coincidencia. En cualquiera de estos casos, almacena un puntero a B en $T[p]$, reemplazando su contenido anterior. Una vez que el codificador encuentra una coincidencia, o decide que no hay ninguna coincidencia, codifica sus descubrimientos utilizando uno de los métodos descritos anteriormente.

La tabla de índices T se inicializa a todas las entradas no válidas, por lo que inicialmente no existen coincidencias. A medida que se almacenan en T más y más punteros a los bloques de píxeles B, la tabla de índices se vuelve más útil, ya que sus entradas contienen más punteros significativos.

Otro detalle de la implementación tiene que ver con el encabezado del archivo comprimido. Éste debe incluir información adicional que depende de la imagen o del método particular utilizado por el codificador. Dicha información es obviamente necesaria para el decodificador. Los principales elementos incluidos en la cabecera son: la resolución de la imagen, el número de planos de bits, el tamaño de bloque (4 en los ejemplos anteriores), y el método utilizado para el desplazamiento de los buffers. Otros ítems que pueden incluirse son: las dimensiones de los búferes de búsqueda y de preanálisis (que determinan los tamaños de los campos desplazamiento y longitud de un token), la convención de bordes utilizada, el formato del fichero comprimido (uno de los tres métodos descritos anteriormente), y si el desplazamiento es una distancia o un par de coordenadas de la imagen.

◇ **Ejercicio 4.30 (sol. en pág. 1084):** Indíquese un ítem más que un codificador sofisticado tenga que incluir en la cabecera.

4.17. Codificación por truncamiento de bloques

La cuantificación es una técnica importante para la compresión de datos en general y para la compresión de imágenes en particular. Cualquier método de cuantificación debe basarse en un principio que determina qué ítems de datos cuantificar y por cuánto. El principio utilizado por el método de *codificación por truncamiento de bloques* (BTC) y sus variantes es cuantificar los píxeles en una imagen preservando al mismo tiempo los dos o tres primeros *momentos estadísticos*. La principal referencia es [Dasarathy 95], que incluye una introducción y copias de los documentos principales en este área. En el método BTC básico, la imagen se divide en bloques (normalmente de 4×4 u 8×8 píxeles cada uno). Suponiendo que un bloque contiene n píxeles con intensidades, de p_1 a p_n, los dos primeros momentos son la media y la varianza, definidas como:

$$\overline{p}=\frac{1}{n}\sum_{i=1}^{n}p_i, \tag{4.30}$$

y

$$\overline{p^2}=\frac{1}{n}\sum_{i=1}^{n}p_i^2, \tag{4.31}$$

respectivamente. La desviación estándar del bloque es:

$$\sigma=\sqrt{\overline{p^2}-\overline{p}}. \tag{4.32}$$

Al principio de la cuantificación se seleccionan tres valores: un umbral (*threshold*) p_{thr}, un valor superior p^+, y un valor inferior p^-. Cada píxel es reemplazado, o bien por p^+, o bien por p^-, de manera que los dos primeros momentos de los nuevos píxeles (i.e., su media y varianza) sean idénticos a los momentos iniciales del bloque de píxeles. La regla de cuantificación es que un píxel p_i es cuantificado a p^+ si es mayor que el umbral, y se cuantifica a p^-es menor que el umbral (si p_i es igual al umbral, puede ser cuantificado a cualquiera de los dos valores). En consecuencia,

$$p_i\leftarrow\begin{cases}p^+, & \text{si } p_i\ge p_{thr},\\ p^-, & \text{si } p_i<p_{thr}.\end{cases}$$

Intuitivamente, es evidente que la media $\overline{p}$ es una buena elección para el umbral. Los valores superior e inferior se pueden determinar escribiendo ecuaciones que preserven los dos primeros momentos, y resolviéndolas. Denotamos n^+ al número de píxeles en el bloque actual que son mayores o iguales que el umbral. Similarmente, n^- denota el número de píxeles que son menores que el umbral. La suma n^++n^- es igual, por supuesto, al número de píxeles n en el bloque. Una vez que la media $\overline{p}$ ha sido calculada, tanto n^+ como n^- son fáciles de calcular. La preservación de los dos primeros momentos se expresa mediante las dos ecuaciones:

$$n\overline{p}=n^-p^--n^+p^+,\quad n\overline{p^2}=n^-\left(p^-\right)^2-n^+\left(p^+\right)^2. \tag{4.33}$$

Éstas son fáciles de resolver a pesar de que no son lineales, y las soluciones son:

$$p^-=\overline{p}-\sigma\sqrt{\frac{n^+}{n^-}},\quad p^+=\overline{p}+\sigma\sqrt{\frac{n^-}{n^+}}. \tag{4.34}$$

Estas soluciones son generalmente números reales, pero tienen que ser redondeados al entero más próximo, lo que implica que la media y la varianza del bloque cuantificado pueden ser algo distintas de las del bloque original. Observe que las soluciones se encuentran en los dos lados de la media $\overline{p}$ a distancias que son proporcionales a la desviación estándar σ del bloque de píxeles.

Ejemplo: Seleccionamos el bloque de 4×4 píxeles de 8 bits:

$$\begin{pmatrix} 121 & 114 & 56 & 47 \\ 37 & 200 & 247 & 255 \\ 16 & 0 & 12 & 169 \\ 43 & 5 & 7 & 251 \end{pmatrix}.$$

La media es $\overline{p} = 98{,}75$, por lo que contamos $n^+ = 7$ píxeles mayores que la media y $n^- = 16 - 7 = 9$ píxeles menores que la media. La desviación estándar es $\sigma = 92{,}95$, por lo que los valores superior e inferior son:

$$p^+ = 98{,}75 + 92{,}95\sqrt{\frac{9}{7}} = 204{,}14, \quad p^- = 98{,}75 - 92{,}95\sqrt{\frac{7}{9}} = 16{,}78.$$

los cuales se redondean a 204 y 17, respectivamente. El bloque resultante es:

$$\begin{pmatrix} 204 & 204 & 17 & 17 \\ 17 & 204 & 204 & 204 \\ 17 & 17 & 17 & 204 \\ 17 & 17 & 17 & 204 \end{pmatrix},$$

y es evidente que el bloque original de 4×4 puede ser comprimido a los 16 bits:

$$\begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

además de los dos valores de 8 bits: 204 y 17; un total de $16 + 2 \times 8$ bits, en comparación con los 16×8 bits del bloque original. El factor de compresión es:

$$\frac{16 \times 8}{16 + 2 \times 8} = 4.$$

◇ **Ejercicio 4.31 (sol. en pág. 1084):** Hágase lo mismo para el bloque de píxeles de 8 bits de 4×4:

$$\begin{pmatrix} 136 & 27 & 144 & 216 \\ 172 & 83 & 43 & 219 \\ 200 & 254 & 1 & 128 \\ 64 & 32 & 96 & 25 \end{pmatrix}.$$

Es evidente que el factor de compresión de este método se conoce de antemano, porque depende del tamaño n de los bloques y del número de bits b por píxel. En general, el factor de compresión es:

$$\frac{bn}{n + 2b}.$$

La Figura 4.113 muestra los factores de compresión para los valores de b: 2, 4, 8, 12 y 16 bits por píxel; y para tamaños de bloque n entre 2 y 16.

El método básico de la BTC es sencillo y rápido. Su principal inconveniente es la forma en que pierde información de la imagen, que se basa en las intensidades de los píxeles en cada bloque, y no en las propiedades del sistema visual humano. Debido a ésto, las imágenes comprimidas mediante la BTC tienden a tener un carácter de bloque cuando son descomprimidas y reconstruidas. Ésto llevó a muchos investigadores para desarrollar versiones mejoradas y ampliadas de la BTC básica, algunos de los cuales se mencionan brevemente aquí:

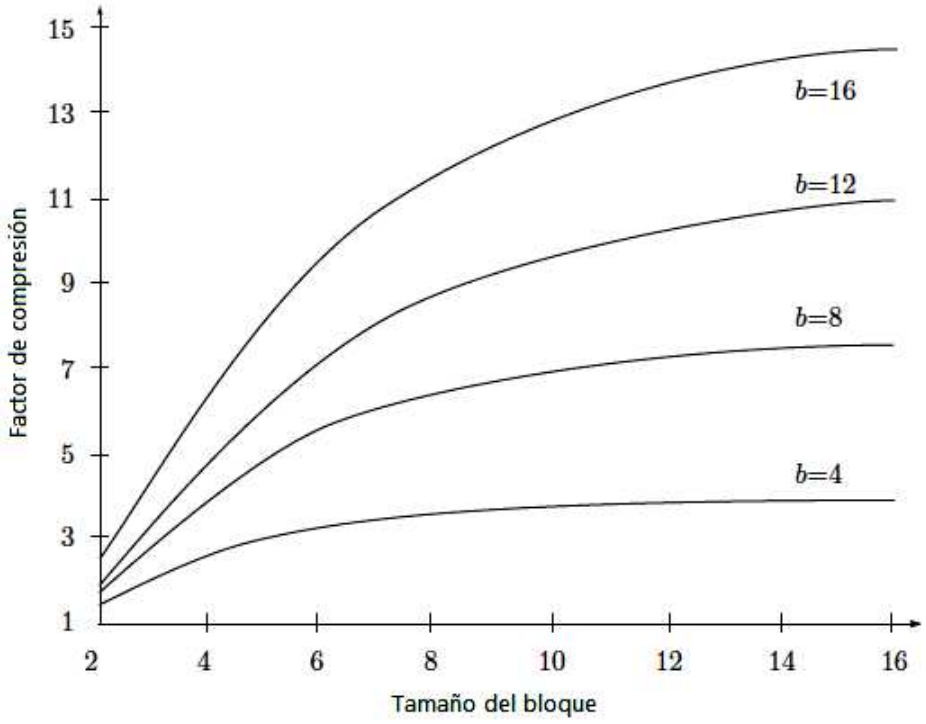


Figura 4.113: Factores de compresión de la BTC para la preservación de dos momentos.

1. Se codifican los resultados antes de escribirlos en el *stream* comprimido. En la BTC base, cada bloque original de n píxeles se convierte en un bloque de n bits más dos números. Es posible acumular B bloques de n bits cada uno, y codificar los Bn bits utilizando la codificación run-length (RLE). Ésto debe ir seguido por $2B$ valores, y ésos pueden ser codificados con códigos prefijo.

2. Se ordenan los n píxeles p_i de un bloque incrementando la intensidad de los valores, y se denotan los píxeles ordenados s_i. Los primeros n^- píxeles s_i tienen valores que son menores que el umbral, y los n^+ píxeles s_i restantes tienen valores mayores. Los valores superior e inferior —p^+ y p^-—, son ahora los valores que minimizan la expresión del error cuadrático:

$$\sum_{i=1}^{n^- -1} \left(s_i - p^-\right)^2 + \sum_{i=n^-}^{n^+} \left(s_i - p^+\right)^2 . \tag{4.35}$$

   Estos valores son:

$$p^- = \frac{1}{n^-} \sum_{i=1}^{n^- -1} s_i \quad \text{y} \quad p^+ = \frac{1}{n^+} \sum_{i=n^-}^{n^+} s_i .$$

   El umbral en este caso se determina buscando todos los $n-1$ posibles valores umbral y escogiendo el que produce el valor más bajo en la expresión del error cuadrático de la Ecuación (4.35).

3. Se ordenan los píxeles como en el punto 2, y se cambia la Ecuación (4.35) a una similar con

valores absolutos en lugar de cuadrados:

$$\sum_{i=1}^{n^- -1} \mid s_i - p^- \mid + \sum_{i=n^-}^{n^+} \mid s_i - p^+ \mid . \tag{4.36}$$

Los valores superior e inferior —p^+y p^-— deberían ser, en este caso, las medianas de los conjuntos de píxeles de baja y alta intensidad, respectivamente. Por consiguiente, p^+ debe ser la mediana del conjunto $\{s_i \mid i = 1, 2, \ldots, n^- - 1\}$, y p^+ debe ser la mediana del conjunto $\{s_i \mid i = n^-, \ldots, n^+\}$. El valor umbral se determina como en el punto 2.

4. Esta es una extensión del método básico de la BTC, donde se conservan *tres* momentos, en lugar de dos. Las tres ecuaciones resultantes permiten el cálculo del valor del umbral, además de aquellos de los parámetros superior (p^+) e inferior (p^-). El tercer momento es:

$$\overline{p^3} = \frac{1}{n}\sum_{i=1}^{n} p_i^3, \tag{4.37}$$

y su preservación produce la ecuación:

$$n\overline{p^3} = n^- \left(p^-\right)^3 - n^+ \left(p^+\right)^3. \tag{4.38}$$

Se resuelven las tres Ecuaciones (4.33) y (4.38) con p^-, p^+, y n^+ como rendimientos desconocidos:

$$n^+ = \frac{n}{2}\left(1 + \frac{\alpha}{\sqrt{\alpha^2 + 4}}\right),$$

donde

$$\alpha = \frac{3\overline{p}\left(\overline{p^2}\right) - \overline{p^3} - 2\left(\overline{p}\right)^3}{\sigma^3}.$$

Y el umbral p_{thr} se selecciona como el píxel p_{n^+}.

5. BTC es un método de compresión con pérdida donde los datos que se pierden se basan en la media y la varianza de los píxeles en cada bloque. Esta pérdida no tiene nada que ver con el sistema visual humano [Salomon 99], o con cualquiera de las características generales de la imagen que se está compriendo, por lo que puede provocar artefactos no deseados y regiones irreconocibles en la imagen descomprimida. Una característica especialmente molesta de la BTC básica es que los bordes rectos en la imagen original se vuelven irregulares en la imagen reconstruida.

   El algoritmo de seguimiento de bordes [Ronson y Dewitte 82] es una extensión de la BTC básica, donde esta tendencia se reduce al mínimo mediante la identificación de los píxeles que se encuentran en un borde, y utilizando un nivel de cuantificación adicional para los bloques que contienen tales píxeles. Cada píxel en dichos bloques se cuantifica en uno de *tres* valores diferentes, en lugar de los dos habituales. Ésto reduce el factor de compresión, pero mejora la apariencia visual de la imagen reconstruida.

   Otra extensión de la BTC básica es un proceso de tres etapas, propuesto en [Dewitte y Ronson 83], donde la primera etapa es la BTC básica, produciendo una matriz de binarios y dos números. La segunda etapa clasifica la matriz binaria en una de tres categorías como sigue:

   a) El bloque no tiene bordes (lo que indica una región uniforme o casi uniforme).

   b) Hay un solo borde transversal en el bloque.

c) Hay un par de bordes transversales en el bloque.

La clasificación se efectúa contando el número de transiciones de 0 a 1 y de 1 a 0 a lo largo de los cuatro bordes de la matriz binaria. Los bloques con cero transiciones o con más de cuatro transiciones se consideran de categoría a (sin bordes). Los bloques con dos transiciones se consideran de categoría b (un solo borde), y los bloques con cuatro transiciones se clasifican como categoría c (dos bordes).

La tercera etapa compara los bloques clasificados de píxeles con un modelo particular que se elige en función de su clasificación. Experimentos con este método sugieren que aproximadamente el 80 % de los bloques pertenecen a la categoría a, el 12 % son de categoría b, y el 8 % pertenecen a la categoría c. El hecho de que los bloques de categoría a sean tan comunes implica que la BTC básica puede ser utilizada en la mayoría de los casos, y los modelos particulares utilizados para las categorías b y c mejoran la apariencia de la imagen reconstruida sin degradar significativamente el tiempo de compresión y el factor de compresión.

6. La BTC original se basa en el principio de preservar los dos primeros momentos estadísticos de un bloque de píxeles. La variante aquí descrita cambia este principio para preservar los dos primeros *momentos absolutos*. El primer momento absoluto viene dado por la ecuación:

$$\overline{p_a} = \frac{1}{n}\sum_{i=1}^{n} \mid p_i - \overline{p} \mid,$$

y se puede demostrar que ésto implica:

$$p^+ = \overline{p} + \frac{n\overline{p_a}}{2n^+} \quad \text{y} \quad p^- = \overline{p} + \frac{n\overline{p_a}}{2n^-},$$

que muestra que los valores superior e inferior son más sencillos que los usados por la BTC básica. Se obtienen a partir de la media sumando y restando cantidades que son directamente proporcionales al primer momento absoluto $\overline{p_a}$ e inversamente proporcionales al número de píxeles n^+ y n^- en ambas partes de esta media. Además, estos valores superior e inferior también se pueden escribir como:

$$p^+ = \frac{1}{n^+}\sum_{p_i \geq \overline{p}} p_i \quad \text{y} \quad p^- = \frac{1}{n^-}\sum_{p_i < \overline{p}} p_i,$$

simplificando su cálculo aún más.

Además de estas mejoras y extensiones hay variantes de la BTC que se desvían significativamente de la idea base. La *BTC de tres niveles* es un buen ejemplo de ésto. Cuantifica cada píxel p_i en uno de tres valores, en lugar de los dos originales. Ésto, por supuesto, requiere dos bits por píxel, lo que reduce el factor de compresión. El método comienza escaneando todos los píxeles en el bloque y encontrando el rango Δ de intensidades:

$$\Delta = \text{máx}\,(p_i) - \text{mín}\,(p_i)\,, \text{ para } i = 1, 2, \ldots, n.$$

Define dos valores medios —superior e inferior, respectivamente— así:

$$\overline{p_h} = \text{máx}\,(p_i) - \Delta/3 \quad \text{y} \quad \overline{p_l} = \text{mín}\,(p_i) + \Delta/3.$$

Los tres niveles de cuantificación —alto, bajo y medio—, vienen dados por:

$$p^+ = \frac{1}{2}\left[\overline{p_h} + \text{máx}\,(p_i)\right], \quad p^- = \frac{1}{2}\left[\overline{p_l} + \text{mín}\,(p_i)\right], \quad p^m = \frac{1}{2}\left[\overline{p_l} + \overline{p_h}\right].$$

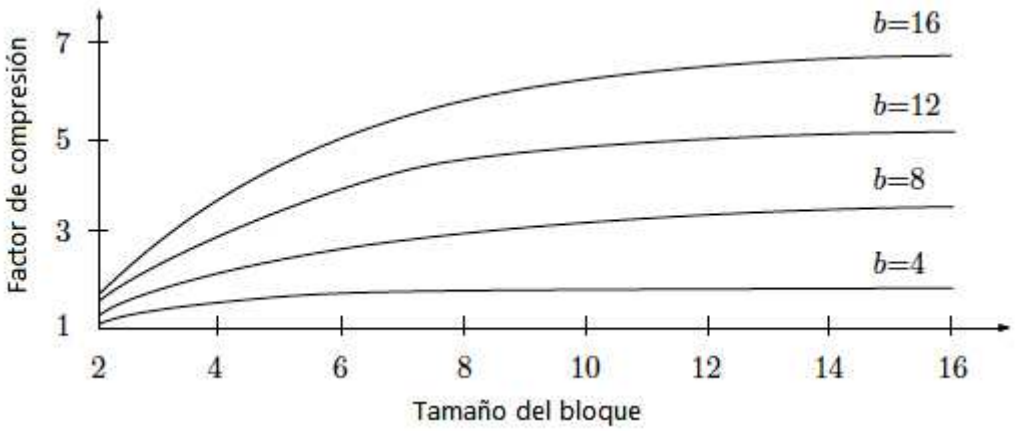


Figura 4.114: Factores de compresión de la BTC para tres niveles de cuantificación.

Compara un píxel p_i con p^+, p^-, y p^m y lo cuantifica al más cercano de estos valores. Se necesitan dos bits para expresar el resultado de la cuantificación de p_i, por lo que el bloque original de n píxeles de b bits se convierte en un bloque de $2n$ bits. Además, los valores de máx (p_i) y mín (p_i) tienen que ser escritos en el flujo de datos comprimidos como dos números de b bits. El factor de compresión resultante es:

$$\frac{bn}{2n+2b},$$

menor que en la BTC básica. La Figura 4.114 muestra los factores de compresión para $b = 4, 8, 12$, y 16 y para n valores en el intervalo $[2, 16]$.

A pesar de empeorar la compresión, la BTC de tres niveles tiene un rendimiento general mejor que la BTC básica, ya que requiere cálculos más sencillos (en particular, el segundo momento no es necesario) y porque sus imágenes reconstruidas son de mayor calidad y no tienen el típico aspecto de bloques de las imágenes comprimidas y descomprimidas bajo la BTC básica.

4.18. Métodos basados en el contexto

La mayoría de los métodos de compresión de imágenes se basan en la observación de que para cualquier píxel seleccionado al azar en la imagen, sus vecinos más próximos tienden a tener el mismo valor que el píxel (o valores similares). Un método de compresión de imágenes basado en el contexto generaliza esta observación. Se apoya en la idea de que el contexto de un píxel se puede utilizar para predecir (estimar la probabilidad de) el píxel.

Dicho método comprime una imagen escaneando píxel por píxel, examinando el contexto de cada píxel, y asignándole una probabilidad que depende del número de veces que se ha visto el mismo contexto en el pasado. El píxel y su probabilidad asignada se envían a un codificador aritmético que efectúa la codificación real. Los métodos aquí descritos se deben a Langdon [Langdon y Rissanen 81] y Moffat [Moffat 91], y se aplican a imágenes monocromáticas (binivel). El contexto de un píxel está formado por algunos de sus píxeles vecinos que ya han sido procesados. El diagrama de abajo muestra un posible contexto de siete píxeles (los píxeles denotados P) formado por cinco píxeles por encima y dos en el lado izquierdo del píxel actual X. Los píxeles denotados "?" no han sido introducidos aún, por tanto no pueden incluirse en el contexto.

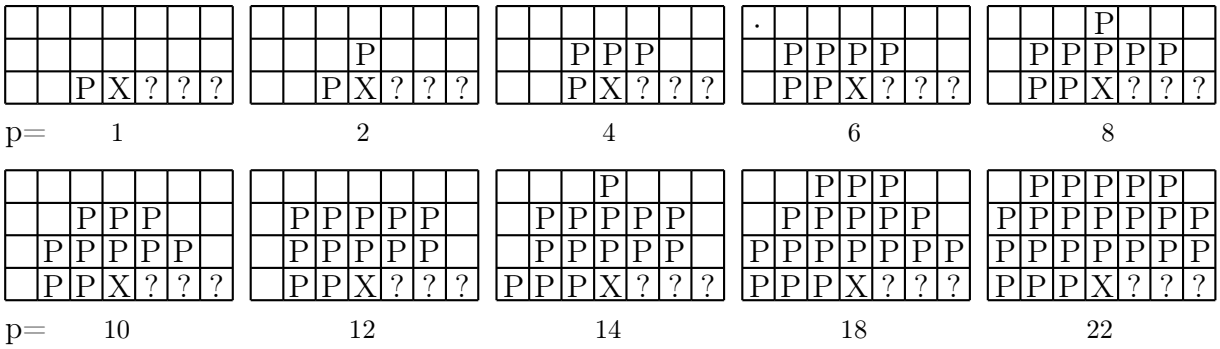


Figura 4.115: Patrones del contexto de un píxel.

.	.	P	P	P	P	P	.	.
.	.	P	P	X	?	?	?	?

La idea principal consiste en utilizar los valores de los siete píxeles de contexto como un índice de 7 bits a una tabla de frecuencias, y encontrar el número de veces que se ha visto un píxel 0 y un píxel 1 en el pasado con el mismo contexto. Aquí hay un ejemplo:

.	.	1	0	0	1	1	.	.
.	.	0	1	X	?	?	?	?

Puesto que $1001101_2 = 77$, se examina la ubicación 77 de la tabla de frecuencias. Suponemos que contiene un recuento de 15 para un píxel 0 y un recuento de 11 para un píxel 1. Por consiguiente, al píxel actual se le asigna una probabilidad de $^{15}/_{(15+11)} \approx 0{,}58$ si es 0 y de $^{11}/_{26} = 0{,}42$ si es 1.

	77	
$\cdots$	15	$\cdots$
$\cdots$	11	$\cdots$

Se incrementa entonces uno de los recuentos en la localización 77, dependiendo de lo que sea el píxel actual. La Figura 4.115 muestra diez maneras posibles de seleccionar los contextos; desde un contexto de 1 bit, a un contexto de 22 bits. En este último caso, puede haber $2^{22} \approx 4{,}2$ millones de contextos, la mayoría de los cuales raramente (o nunca) se producen en la imagen. En lugar de mantener un arreglo enorme y casi vacío, la tabla de frecuencias puede ser implementada como un árbol de búsqueda binaria o una tabla *hash* de dispersión. Para contextos cortos de 7 bits, la tabla de frecuencias puede ser un array, lo que resulta en una búsqueda rápida.

> "Bendígame, bendígame, Sept," contestó la anciana, "¡usted no ve el contexto! Devuélvamelo, querido."
>
> —Charles Dickens, *El misterio de Edwin Drood* (1870)

La experiencia demuestra que los contextos más largos producen una mejor compresión, hasta aproximadamente 12−14 bits. Los contextos más largos que este resultado empeoran la compresión,

Número	Píxel	Contexto	Recuentos	Probabilidad	Nuevos recuentos
1	1	$00 = 0$	1, 1	$1/(1+1) = 1/2$	1, 2
2	0	$01 = 1$	1, 1	$1/2$	2, 1
3	1	$00 = 0$	1, 2	$2/3$	1, 3
4	1	$01 = 1$	2, 1	$1/3$	2, 2

Tabla 4.116: Recuentos y probabilidades para los primeros cuatro píxeles.

lo que indica que un píxel de una imagen típica no "está relacionado con" la distancia entre píxeles (las correlaciones entre los píxeles están limitadas típicamente a una distancia de $1-2$).

Como es usual para un método de compresión basado en el contexto, debe tenerse cuidado para evitar las probabilidades cero. La tabla de frecuencias, por lo tanto, debe ser inicializada a valores distintos de cero, y la experiencia muestra que la forma exacta de hacer ésto no afecta al rendimiento de la compresión de manera significativa. Parece que lo mejor es inicializar cada elemento de la tabla a 1.

Cuando comienza el proceso, los primeros píxeles que deben ser analizados no tienen vecinos por encima de ellos o por la izquierda. Una sencilla manera de manejar ésto es asumir que los vecinos que no existen son ceros. Es como si la imagen se ampliara mediante la adición de tantas filas de píxeles a cero en la parte superior y, tantas columnas a cero por la izquierda, como sea necesario.

Ahora se usa el contexto de 2 bits de la Figura 4.115, como ejemplo, para codificar la primera fila de la imagen de 4×4:

	0 0 0 0
0	1 0 1 1
0	0 1 0 1
0	1 0 0 1
0	0 0 1 0

Los resultados se resumen en la Tabla 4.116.

◇ **Ejercicio 4.32 (sol. en pág. 1084):** Continúese la Tabla 4.116 para la siguiente fila de cuatro píxeles 0101.

Los contextos de la Figura 4.115 no son simétricos con respecto al píxel actual, ya que deben utilizar los píxeles que ya han sido procesados ("histórico" de píxeles). Si el algoritmo escanea la imagen por filas, aquellos serán los píxeles por encima del píxel actual o a su izquierda. En la práctica es imposible incluir los píxeles "futuros" en el contexto (porque el decodificador no tendría sus valores al decodificar el píxel actual), pero en experimentos, donde no hay necesidad de decodificar realmente la imagen comprimida, es posible almacenar toda la imagen en la memoria, por lo que el codificador puede examinar cualquier píxel en cualquier momento. Los experimentos realizados con contextos simétricos han demostrado que el rendimiento de la compresión puede mejorar, como mucho, en un 30 %. (El método MLP, Sección 4.21, ofrece un interesante giro al problema del contexto simétrico.)

◇ **Ejercicio 4.33 (sol. en pág. 1085):** (Fácil.) ¿Por qué es posible utilizar píxeles "futuros" en un experimento, pero no en la práctica? ¿Podría ocurrir que la imagen, o parte de ella, fuera almacenada en la memoria y el codificador pudiera utilizar cualquier píxel como parte de un contexto?

Una desventaja de un gran contexto es que para "aprender" de él es preciso un algoritmo más largo. Un contexto de 20 bits, por ejemplo, permite aproximadamente un millón de contextos diferentes. Se necesitan muchos millones de píxeles para recoger los recuentos suficientes para todos los contextos, lo cual es una de las razones por la que los contextos grandes no producen una mejor compresión. Una forma de mejorar nuestro método consiste en implementar un algoritmo *binivel* que utilice un contexto largo sólo si este contexto ya se ha visto Q veces o más (donde Q es un parámetro, típicamente

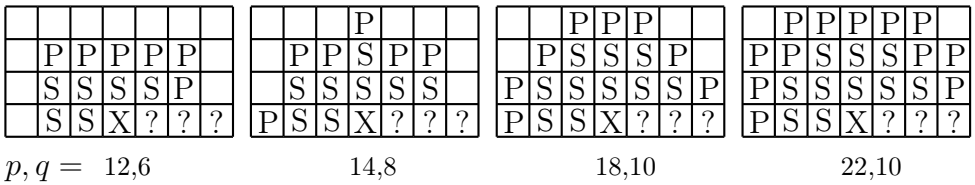


Figura 4.117: Patrones del contexto binivel de un píxel.

establecido a un valor tan pequeño como 2 ó 3). Si un contexto ha aparecido menos de Q veces, se considera poco fiable, y sólo se utiliza un pequeño subconjunto del mismo para predecir el píxel actual. La Figura 4.117 muestra cuatro de tales contextos, donde los píxeles del subconjunto están etiquetados con S. La notación p, q significa: contexto binivel de p bits con un subconjunto de q bits.

La experiencia demuestra que los contextos 18, 10 y 22, 10 producen una mejor, aunque no revolucionaria, compresión.

4.19. FELICS

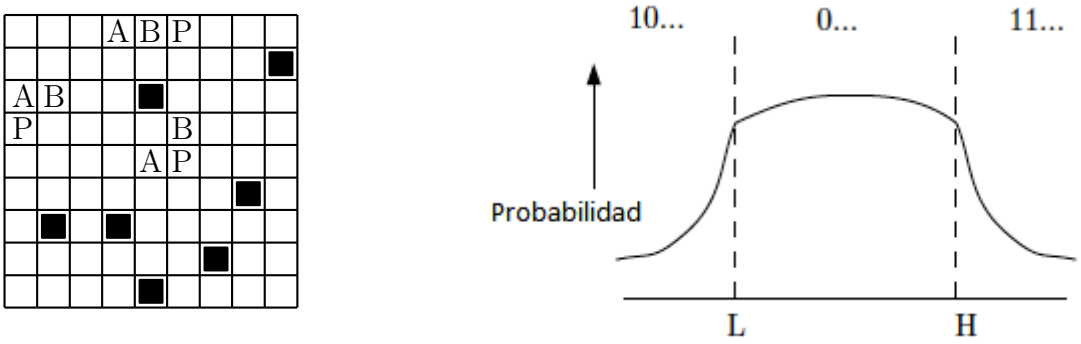
FELICS es un acrónimo de Fast, Efficient, Lossless Image Compression System (sistema de compresión de imágenes sin pérdidas, rápido y eficiente). Es un método de compresión de propósito especial [Howard y Vitter 93] diseñado para imágenes en escala de grises y compite con el modo sin pérdidas de JPEG (Sección 4.8.5). Es rápido y generalmente produce una buena compresión. Sin embargo, no se puede comprimir una imagen por debajo de un bit por píxel, por lo que no es una buena elección para imágenes binivel o altamente redundantes.

El principio de FELICS es codificar cada píxel con un código de tamaño variable basado en los valores de dos de sus píxeles vecinos vistos anteriormente. La Figura 4.118a muestra los dos vecinos conocidos A y B de algunos píxeles P. Para un píxel en general, son los vecinos situados por encima de él y a su izquierda. Para un píxel ubicado en la fila superior, estos son sus dos vecinos izquierdos (excepto para los dos primeros píxeles de la imagen). Para un píxel ubicado en la columna del extremo izquierdo, estos son los primeros dos píxeles de la línea por encima de él. Observe que los dos primeros píxeles de la imagen no tienen ningún vecino visto previamente, pero ya que hay sólo dos de ellos, pueden ser emitidos sin ningún tipo de codificación, causando sólo una ligera degradación en el proceso de compresión total.

◇ **Ejercicio 4.34 (sol. en pág. 1085):** ¿Qué modelo es utilizado por FELICS para predecir el píxel actual?

Considere los dos vecinos A y B de un píxel P. Usamos A, B y P para identificar tanto a los tres píxeles como a sus intensidades (valores de escala de grises). Denotamos L y H a los vecinos con la intensidad más pequeña y más grande, respectivamente. Al píxel P debe asignársele un código de tamaño variable dependiendo de dónde se encuentre la intensidad de P con respecto a L y H. Hay tres casos:

1. La intensidad del píxel P está entre L y H (se encuentra en la región central de la Figura 4.118b). Se sabe experimentalmente que ésto ocurre en aproximadamente la mitad de los píxeles, y a P se le asigna, en este caso, un código que comienza con 0. (Ocurre un caso especial cuando L = H. En tal caso, el intervalo [L, H] consta de un único valor, y la probabilidad de que P tenga ése valor es pequeño.) La probabilidad de que P esté en esta región central es casi, pero no completamente, plana, por consiguiente, a P debe asignársele un código binario que tenga aproximadamente el mismo tamaño en toda la región pero ligeramente más corto en el centro de la región.



(a) (b)

Figura 4.118: (a) Los dos vecinos. (b) Las tres regiones.

2. La intensidad de P es menor que L (P está en la región izquierda). El código asignado a P, en este caso, comienza con 10.

3. La intensidad de los P's es mayor que H. A P se le asigna un código que comienza con 11.

Cuando el píxel P está en una de las regiones exteriores, la probabilidad de que su intensidad sea distinta de L o H es mucho más pequeña, por lo que a P se le puede asignar un código largo en estos casos.

El código asignado a P, por tanto, depende en gran medida de si P está en la región central o en una de las regiones exteriores. Así es como el código es asignado cuando P está en la región central. Necesitamos $\mathrm{H}-\mathrm{L}+1$ códigos de tamaño variable que no difieran mucho en tamaño y, por supuesto, satisfagan la propiedad prefija. Fijamos $k=[\log_2(\mathrm{H}-\mathrm{L}+1)]$ y calculamos los enteros a y b mediante:

$$a=2^{k+1}-(\mathrm{H}-\mathrm{L}+1)\,,\quad b=2\left(\mathrm{H}-\mathrm{L}+1-2^k\right).$$

Ejemplo: Si $\mathrm{H}-\mathrm{L}=9$, entonces $k=3$, $a=2^{3+1}-(9+1)=6$, y $b=2\left(9+1-2^3\right)=4$. Ahora seleccionamos los a códigos 2^k-1, 2^k-2,... expresados como números de k bits, y los b códigos 0, 1, 2, ... expresados como números de $k+1$ bits. En el ejemplo anterior, los a códigos son: $8-1=111$, $8-2=110$, y así hasta $8-6=010$; y los b códigos: 0000, 0001, 0010, y 0011. Los a códigos cortos se asignan a los valores de P ubicados en la mitad de la región central, y los b códigos largos se asignan a los valores de P más cercanos a L o H. Obseve que b es par, por lo que los b códigos siempre se pueden dividir en dos conjuntos iguales. La Tabla 4.119 muestra cómo pueden ser asignados diez de dichos códigos en el caso $\mathrm{L}=15$, $\mathrm{H}=24$.

Cuando P está en una de las regiones exteriores, digamos la superior, el valor $\mathrm{P}-\mathrm{H}$ debe asignarse a un código de tamaño variable cuya longitud pueda crecer rápidamente a medida que $\mathrm{P}-\mathrm{H}$ se haga más grande. Una forma de hacer ésto es seleccionar un entero pequeño no negativo m (típicamente 0, 1, 2, ó 3) y asignar el entero n a un código de dos partes. La segunda parte está formada por los m bits menos significativos de n, y la primera parte es el código unario de [n sin sus m bits menos significativos] (véase el Ejercicio 2.5 como recordatorio del código unario). Ejemplo: Si $m=2$, entonces $n=1101_2$ se asigna al código 110|01, ya que 110 es el código unitario de 11. Este código es un caso especial del código de Golomb (Sección 2.5), donde el parámetro b es una potencia de 2 (2^m). La Tabla 4.120 muestra algunos ejemplos de este código para $m=0,1,2,3$ y $n=1,2,\ldots,9$. El valor de m utilizado en cualquier trabajo de compresión en particular puede ser seleccionado, como un parámetro, por el usuario.

Píxel P	Código de la región	Código del píxel
L =15	0	0000
16	0	0010
17	0	010
18	0	011
19	0	100
20	0	101
21	0	110
22	0	111
23	0	0001
H =24	0	0011

Tabla 4.119: Los códigos para la región central.

Píxel		Código de	$m =$			
P	P − H	la región	0	1	2	3
H + 1 =25	1	11	0	00	000	0000
26	2	11	10	01	001	0001
27	3	11	110	100	010	0010
28	4	11	1110	101	011	0011
29	5	11	11110	1100	1000	0100
30	6	11	111110	1101	1001	0101
31	7	11	1111110	11100	1010	0110
32	8	11	11111110	11101	1011	0111
33	9	11	111111110	111100	11000	10000
...	...	...	...		...	
...	...	...	...		...	

Tabla 4.120: Los códigos para otra región.

2	5	7	12
3	0	11	10
2	1	8	15
4	13	11	9

Tabla 4.121: Un mapa de bits de 4×4.

◇ **Ejercicio 4.35 (sol. en pág. 1085):** Dado el mapa de bits de 4×4 de la Figura 4.121, calcúlense los códigos FELICS para los tres píxeles con valores de 8, 7, y 0.

4.20. FELICS progresivo

El método FELICS original puede ser fácilmente extendido a la compresión progresiva de imágenes debido a su principio fundamental. FELICS escanea la imagen en orden *raster* (fila por fila) y codifica un píxel basándose en los valores de dos de sus vecinos (vistos y codificados previamente). El FELICS progresivo [Howard y Vitter 94a] funciona de manera similar, pero explora los píxeles por niveles. Cada nivel utiliza los k píxeles codificados en todos los niveles anteriores para codificar k píxeles más, por lo que el número de píxeles codificados se duplica tras cada nivel. Suponiendo que la imagen se compone de $n \times n$ píxeles, y el primer nivel comienza con exactamente cuatro píxeles, los niveles consecutivos dan lugar a:

$$4, 8, \ldots, \frac{n^2}{8}, \frac{n^2}{4}, \frac{n^2}{2}, n^2$$

píxeles. Así, el número de niveles es el número de términos, $2\log_2 n - 1$, en esta secuencia. (Esta propiedad es fácil de demostrar. El primer término se puede escribir[14]:

$$4 = \frac{n^2}{2^{-2}n^2} = \frac{n^2}{2^{2log\, n-2}}.$$

Por consiguiente, los términos de la secuencia contienen las potencias de 2 con exponentes, de 0 a $2\log_2 n - 2$, mostrando que hay $2\log_2 n - 1$ términos.)

La Figura 4.122 muestra los píxeles codificados en la mayoría de los niveles de una imagen de 16×16 píxeles. La Figura 4.123 muestra cómo son seleccionados los píxeles de cada nivel. En la Figura 4.123a hay $8 \times 8 = 64$ píxeles, una cuarta parte el número final, dispuestos en una cuadrícula. Cada grupo de cuatro píxeles se utiliza para codificar un nuevo píxel, por lo que la Figura 4.123b tiene 128 píxeles, la mitad del número final. La imagen de la Figura 4.123b se rota luego 45º y es escalada en un factor de $\sqrt{2} \approx 1{,}414$ en ambas direcciones, para producir la Figura 4.123c, la cual es una cuadrícula que se ve exactamente igual que la Figura 4.123a. El siguiente paso (no mostrado en la figura) es utilizar todos los grupos de 4×4 píxeles de la Figura 4.123c para codificar un píxel, codificando así los 128 píxeles restantes. En la práctica, no hay necesidad de efectuar una rotación real, ni de escalar la imagen; el programa simplemente alterna entre las coordenadas: xy y diagonal.

Cada grupo de cuatro píxeles se utiliza para codificar el píxel ubicado en su centro. Observe que en los primeros niveles, los cuatro píxeles de un grupo están lejos el uno del otro y no están correlacionados, por lo que la compresión es pobre. Sin embargo, los dos últimos niveles codifican 3/4 del número total de píxeles, y estos niveles contienen grupos compactos. Dos de los cuatro píxeles de un grupo son seleccionados para codificar el píxel central, y se designan L y H. La experiencia muestra que la mejor elección para L y H son los dos píxeles mediana (página 371), los que tienen valores medios (i.e., no los píxeles máximos o mínimos del grupo). Los vínculos se pueden resolver de cualquier forma, pero deben ser consistentes. Si las dos medianas de un grupo son iguales, entonces se pueden elegir los píxeles mediana y mínimo (o mediana y máximo). Los dos píxels seleccionados, L y H, se utilizan para codificar el píxel central de la misma manera que FELICS utiliza dos vecinos para codificar un píxel. La única diferencia es que se usa un código prefijo nuevo (Sección 4.20.1), en lugar del código de Golomb.

◇ **Ejercicio 4.36 (sol. en pág. 1086):** ¿Por qué es importante resolver los vínculos de una manera coherente?

[14] Recuérdese que: $a = 2^{\log_2 a}$; $\log a^b = b \cdot \log a$; y $2^a \cdot 2^b = 2^{a+b}$.

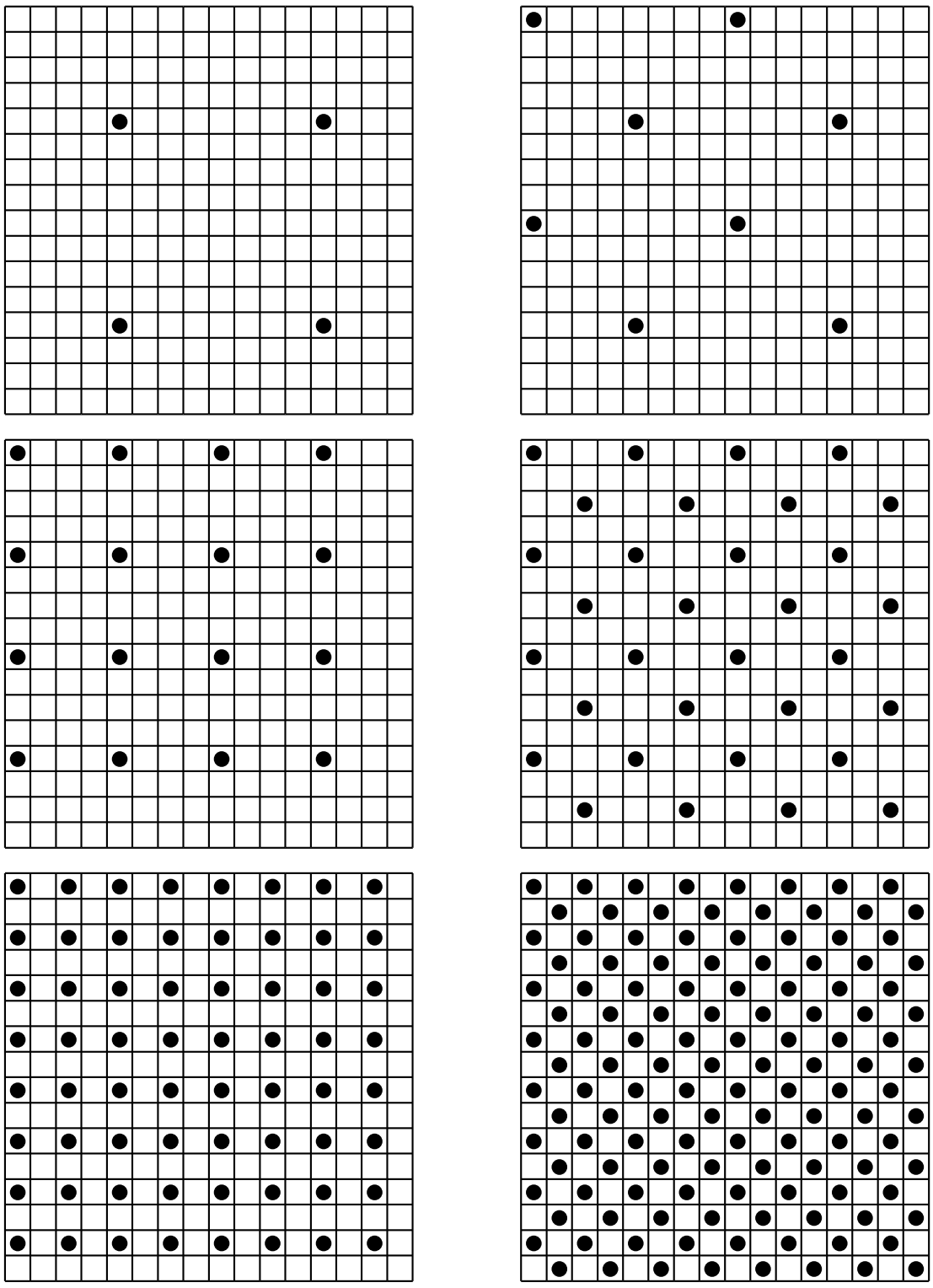


Figura 4.122: Algunos niveles de una imagen de 16×16.

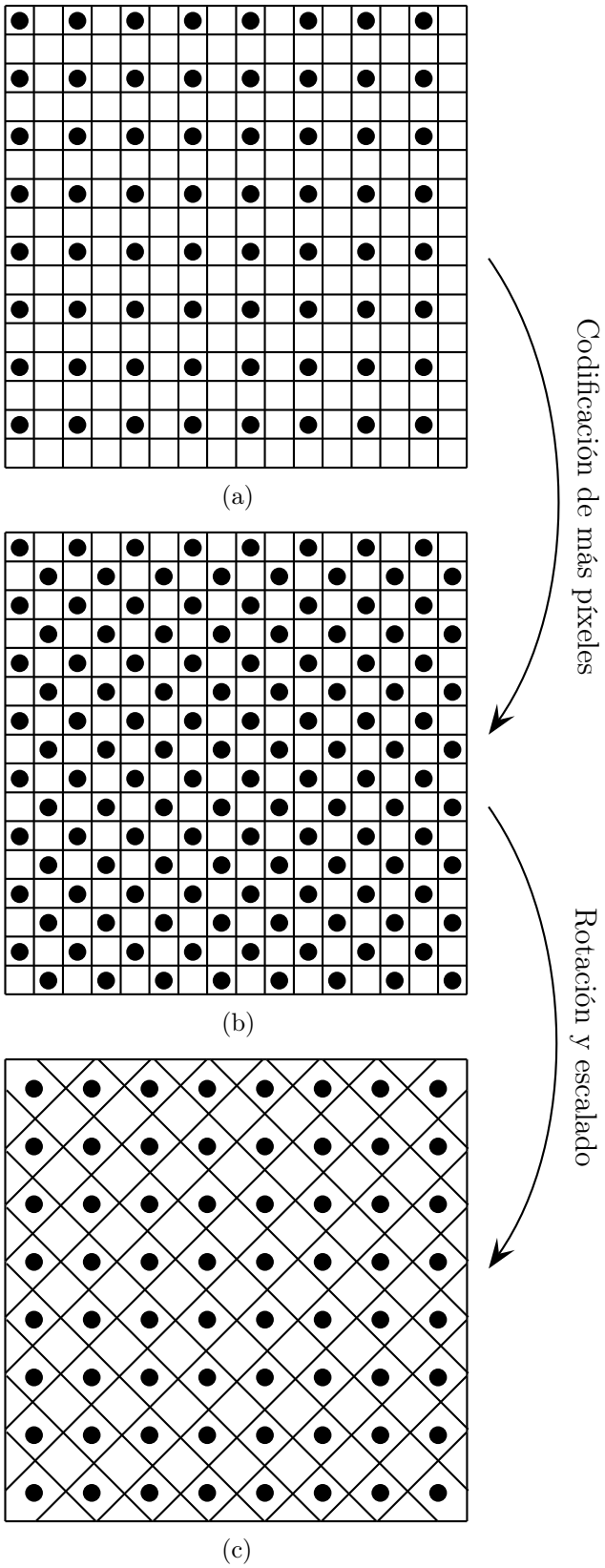


Figura 4.123: Rotación y escalado.

4.20.1. Código subexponential

En los primeros niveles, los cuatro píxeles utilizados para codificar un píxel están distantes el uno del otro. A medida que se van codificando progresivamente más niveles los grupos se vuelven más compactos, por lo que sus píxeles están más próximos. El codificador utiliza la diferencia absoluta entre los píxeles L y H en un grupo (el *contexto* del grupo) para codificar el píxel ubicado en el centro del grupo, pero una diferencia absoluta dada representa más para los niveles finales que para los primeros, porque los grupos de los últimos niveles son más pequeños, por lo que sus píxeles están más correlacionados. Por consiguiente, el codificador debe aumentar la diferencia mediante un parámetro de ponderación s, que aumenta de valor con cada nivel. El valor específico de s no es crítico, y los experimentos recomiendan el valor 12.

El código prefijo utilizado por el FELICS progresivo recibe el nombre de *subexponential*. Está relacionado con los códigos de Rice de la Sección 7.9. Al igual que el código de Golomb (Sección 2.5), este nuevo código depende de un parámetro $k \geq 0$. La principal característica del código subexponential es su longitud. Para los enteros $n < 2^{2k+1}$ la longitud del código se incrementa linealmente con n, pero para valores mayores de n, el incremento es logarítmico. El código subexponential de enteros no negativos n se calcula en dos pasos. En el primer paso, los valores b y u se calculan mediante:

$$b = \begin{cases} k & \text{si } n < 2^k, \\ [\log_2 n] & \text{si } n \geq 2^k, \end{cases} \quad \text{y} \quad u = \begin{cases} 0 & \text{si } n < 2^k, \\ b - k + 1 & \text{si } n \geq 2^k. \end{cases}$$

En el segundo paso, el código unario de u (de $u + 1$ bits), seguido por los b bits menos significativos de n, se convierten en el código subexponential de n. Por consiguiente, el tamaño total del código es:

$$u + 1 + b = \begin{cases} k + 1 & \text{si } n < 2^k, \\ 2\,[\log_2 n] - k + 2 & \text{si } n \geq 2^k. \end{cases}$$

La Tabla 4.124 muestra ejemplos del código subexponential para varios valores de n y k. Se puede demostrar que para un n dado, las longitudes de código para valores consecutivos de k difieren, a lo sumo, en 1.

Si el valor del píxel a codificar se encuentra entre aquellos de L y H, el píxel se codifica como en FELICS. Si se encuentra fuera del rango [L, H], el píxel se codifica usando el código subexponential, donde el valor de k se selecciona mediante la regla siguiente: Suponemos que el píxel actual P a ser codificado tiene un contexto C. El codificador mantiene un total acumulado, para algunos valores razonables de k, de la longitud del código que el codificador tendría si hubiera utilizado el valor de k para codificar todos los píxeles encontrados hasta ahora en el contexto C. A continuación, el codificador utiliza el valor de k con la longitud acumulada más pequeña de código para codificar el píxel P.

4.21. MLP

Nota: El método de compresión de imágenes MLP descrito en esta sección es diferente y sin relación con el método de compresión de audio MLP (meridian lossless packing —empaquetamiento sin pérdidas de meridian—) de la Sección 7.7. Los acrónimos son idénticos, una desafortunada coincidencia.

Los métodos de compresión de texto pueden utilizar el contexto para predecir (i.e., para estimar la probabilidad de) el siguiente carácter de texto. El contexto también puede utilizarse para predecir la intensidad del siguiente píxel en la compresión de imágenes, pero esto es más complejo por dos razones: (1) Una imagen es bidimensional, lo que permite muchos contextos posibles, y (2) es una imagen digital es, a menudo, el resultado de la digitalización de una imagen analógica. La intensidad de cualquier píxel individual está determinado por los detalles del escaneo y puede diferir de la intensidad "ideal".

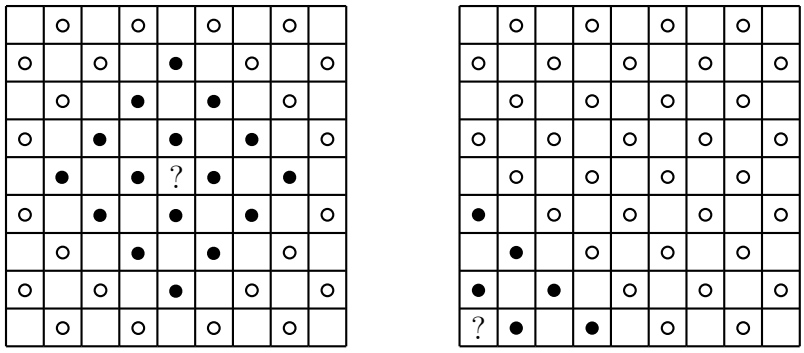
n	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$
0	0|	0|0	0|00	0|000	0|0000	0|00000
1	10|	0|1	0|01	0|001	0|0001	0|00001
2	110|0	10|0	0|10	0|010	0|0010	0|00010
3	110|1	10|1	0|11	0|011	0|0011	0|00011
4	1110|00	110|00	10|00	0|100	0|0100	0|00100
5	1110|01	110|01	10|01	0|101	0|0101	0|00101
6	1110|10	110|10	10|10	0|110	0|0110	0|00110
7	1110|11	110|11	10|11	0|111	0|0111	0|00111
8	11110|000	1110|000	110|000	10|000	0|1000	0|01000
9	11110|001	1110|001	110|001	10|001	0|1001	0|01001
10	11110|010	1110|010	110|010	10|010	0|1010	0|01010
11	11110|011	1110|011	110|011	10|011	0|1011	0|01011
12	11110|100	1110|100	110|100	10|100	0|1100	0|01100
13	11110|101	1110|101	110|101	10|101	0|1101	0|01101
14	11110|110	1110|110	110|110	10|110	0|1110	0|01110
15	11110|111	1110|111	110|111	10|111	0|1111	0|01111
16	111110|0000	11110|0000	1110|0000	110|0000	10|0000	0|10000

Tabla 4.124: Algunos códigos subexponenciales.

El método progresivo multinivel (MLP) que se describe aquí [Howard y Vitter 92a], es un método sin pérdidas y computacionalmente intensivo para la compresión de imágenes en escala de grises. Utiliza el contexto para predecir las intensidades de los píxeles, luego usa la codificación aritmética para codificar la diferencia entre la predicción y el valor real de un píxel (el error). Emplea la distribución de Laplace para estimar la probabilidad del error. El método combina cuatro etapas separadas: (1) secuenciación de píxeles, (2) predicción (modelado de la imagen), (3) modelado de los errores (por medio de la distribución de Laplace), y (4) codificación aritmética de los errores.

MLP también es progresivo, codificando la imagen en niveles, donde los píxeles de cada nivel se seleccionan como en el FELICS progresivo. Cuando la imagen es decodificada, cada nivel añade detalles a la imagen completa, no sólo a ciertas partes, por lo que el usuario puede ver la imagen a medida que se decodifica y decidir en tiempo real la posibilidad de aceptarla o rechazarla. Esta característica es útil cuando la imagen tiene que ser seleccionada de un gran archivo de imágenes comprimidas. El usuario puede navegar a través de las imágenes muy rápido, sin tener que esperar a que una imagen sea completamente decodificada y mostrada. Otra ventaja de la compresión progresiva es que proporciona una opción intuitiva de compresión con pérdida . El codificador puede indicar que se deje de codificar antes de que alcance el último nivel (codificando, por lo tanto, sólo la mitad del número total de píxeles) o antes de que llegue al penúltimo nivel (codificando sólo una cuarta parte del número total de píxeles). Tal opción ofrece una compresión excelente, pero produce una pérdida de datos de la imagen. El decodificador puede indicar que se utilice la interpolación para determinar las intensidades de los píxeles que falten.

Al igual que cualquier método de compresión para imágenes en escala de grises, MLP puede utilizarse para comprimir imágenes en color. El color de la imagen original debe ser dividido en tres componentes de color, y cada componente comprime de forma individual como una imagen en escala de grises. A continuación se describen en detalle los pasos individuales de la codificación MLP.



(a) (b)

Figura 4.125: (a) Dieciséis vecinos. (b) Seis vecinos.

> Lo que sabemos no es mucho. Lo que no sabemos es inmenso.
>
> —(Presuntamente, últimas palabras de Laplace.)

4.21.1. Secuenciación de píxeles

Los píxeles se seleccionan por niveles, como en el FELICS progresivo, donde en cada nivel se codifica el mismo número de píxeles que en todos los niveles precedentes combinados, lo que duplica el número de píxeles codificados. Ésto significa que el último nivel codifica la mitad del número de píxeles, el nivel que le precede codifica una cuarta parte del número total, y así sucesivamente. El primer nivel debe comenzar con al menos cuatro píxeles, pero también puede comenzar con 8, 16, o cualquier potencia deseada de 2.

> El hombre únicamente sigue a fantasmas.
>
> —(Presuntamente, últimas palabras de Laplace.)

4.21.2. Predicción

Un píxel se predice mediante el cálculo de un promedio ponderado de 16 de sus vecinos conocidos. Tenga en cuenta que los píxeles no son escaneados en orden *raster* (de arriba abajo y de izquierda a derecha), sino que se codifican (y por tanto, también se decodifican) por niveles. Cuando se decodifican los píxeles del nivel L, el decodificador MLP ya ha decodificado todos los píxeles de todos los niveles precedentes, y puede utilizar sus valores (escalas de grises) para predecir los valores de los píxeles de L. La Figura 4.125a muestra la situación cuando el codificador MLP procesa el último nivel. La mitad de los píxeles ya han sido codificados en los niveles anteriores, por consiguiente, el decodificador conoce cuándo se ha decodificado el último nivel. El codificador, por lo tanto, puede utilizar un grupo en forma de diamante de 4×4 píxeles (mostrado en negro) de los niveles anteriores para predecir el píxel ubicado en el centro del grupo. Este grupo se convierte en el *contexto* del píxel. Los métodos de compresión que escanean la imagen en orden *raster* puede utilizar sólo los píxeles situados arriba y a la izquierda del píxel P, para predecir P. Debido a la naturaleza progresiva del MLP, se puede usar

				1								0,0039								
			−9		−9						−0,0351		−0,0351							
		−9		81		−9				−0,0351		0,3163			−0,0351					
	1		81		81		1		0,0039		0,3163		0,3163				0,0039			
		−9		81		−9				−0,0351		0,3163			−0,0351					
			−9		−9						−0,0351		−0,0351							
				1								0,0039								

Tabla 4.126: 16 pesos. (a) Enteros. (b) Normalizados.

un contexto simétrico, que genera predicciones más precisas. Por otro lado, los píxeles del contexto no son vecinos cercanos e incluso pueden ser (en los niveles iniciales) lejanos al píxel predicho.

La Tabla 4.126 muestra los 16 pesos utilizados para un grupo. Se calculan mediante interpolación polinómica bicúbica (Sección 4.21.4) y se normalizan de manera que sumen 1. (Observe que en la Tabla 4.126a los pesos no están normalizados; suman 256. Cuando se utilizan éstos pesos, la suma ponderada debe dividirse por 256.) Para predecir un píxel cercano a un borde, donde faltan algunos de los 16 vecinos (como en la Figura 4.125b), sólo se utilizan aquellos vecinos que existen, y sus pesos son renormalizados, para que su suma sea 1.

◇ **Ejercicio 4.37 (sol. en pág. 1086):** ¿Por qué los pesos tienen que sumar 1?

◇ **Ejercicio 4.38 (sol. en pág. 1086):** Muéstrese cómo renormalizar los seis pesos necesarios para predecir el píxel ubicado en la esquina inferior izquierda de la Figura 4.125b.

El codificador predice todos los píxeles de un nivel utilizando el grupo en forma de diamante de 4×4 (o inferior) de píxeles "más viejos" alrededor de cada píxel del nivel. Este es el *modelo de imagen* utilizado por el MLP.

> Es difícil de predecir, especialmente el futuro.
>
> —Niels Bohr.

4.21.3. Modelado de los errores

Supongamos que la suma ponderada de los 16 vecinos cercanos al píxel P es igual a R. En consecuencia, R es el valor predicho para P. El error de predicción, E, es simplemente la diferencia R − P. Asumiendo una imagen con 16 niveles de gris (cuatro bits por píxel), el mayor valor de E es 15 (cuando R = 15 y P = 0) y el más pequeño es −15. Dependiendo de la imagen, podemos esperar que la mayoría de los errores sean un entero pequeño, cero o casi cero. Pocos errores serán ±15 o cercanos a éso. Los experimentos con un gran número de imágenes (véase, por ejemplo, [Netravali y Limb 80]) han producido la distribución de errores mostrada en la Figura 4.128a. Ésta es una curva simétrica, estrecha, y con un pico agudo, lo que indica que la mayor parte de los errores son pequeños y están, por lo tanto, concentrados en la parte superior. Esta curva tiene la forma de la bien conocida distribución de Laplace (Sección 7.2.1) con media 0. Esta distribución estadística es similar a la distribución normal (gaussiana), pero es más estrecha. Se ajusta a la curva de errores mejor que la distribución normal, porque la mayoría de los valores erróneos son pequeños.

La Tabla 4.127 muestra algunos valores para las distribuciones de Laplace con $m = 0$ y $V = 3, 4, 5$ y 1000. La Figura 4.128b muestra las gráficas de los tres primeros de ellos. Es evidente, que a medida que crece V, el gráfico se hace mas bajo y más ancho, con un pico menos pronunciado.

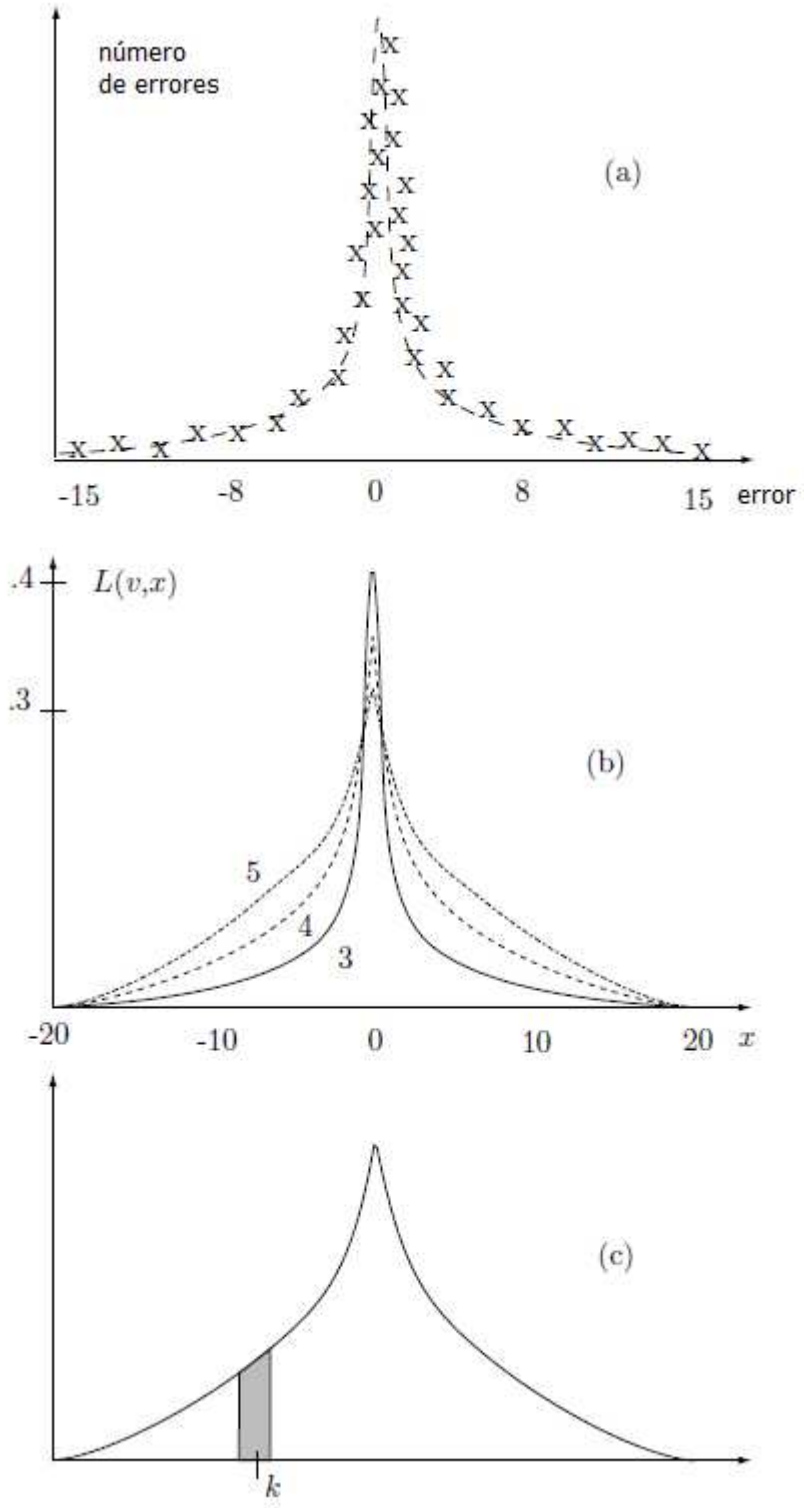


Figura 4.128: (a) Distribución de errores. (b) Distribuciones de Laplace. (c) Probabilidad de k.

	x					
V	0	2	4	6	8	10
3:	0,408248	0,0797489	0,015578	0,00304316	0,00059446	0,000116125
4:	0,353553	0,0859547	0,020897	0,00508042	0,00123513	0,000300282
5:	0,316228	0,0892598	0,025194	0,00711162	0,00200736	0,000566605
1000:	0,022360	0,0204475	0,018698	0,0170982	0,0156353	0,0142976

Tabla 4.127: Algunos valores de la distribución de Laplace con $V = 3, 4, 5$ y 1 000.

El factor $1/\sqrt{2V}$ está incluido en la definición de la distribución de Laplace con el fin de escalar el área bajo la curva de la distribución a 1. Debido a ésto, es fácil de utilizar la curva de la distribución para calcular la probabilidad de cualquier valor de error. La Figura 4.128c muestra una banda gris, de 1 unidad de ancho, bajo la curva de la distribución, centrada en un valor de error de k. El área de esta banda es igual a la probabilidad de cualquier error E que tenga el valor k. Matemáticamente, el área es la integral:

$$P_V(k) = \int_{k-0,5}^{k+0,5} \frac{1}{\sqrt{2V}} \exp\left(-\sqrt{\frac{2}{V}} \mid x \mid\right) dx, \tag{4.39}$$

y ésta es la clave para codificar los errores. Con pixeles de 4 bits, los valores de error están en el intervalo $[-15, +15]$. Cuando se obtiene un error k, el codificador MLP lo codifica aritméticamente con una probabilidad calculada mediante la Ecuación (4.39). En la práctica, tanto el codificador como el decodificador debe tener una tabla con todas las probabilidades precalculadas posibles.

El único punto pendiente de discutir es qué valor de varianza V debe utilizarse en la Ecuación (4.39). Ambos, codificador y decodificador, necesitan conocer este valor. Es evidente, a partir de la Figura 4.128b, que el uso de una varianza grande (que corresponde a una distribución baja y plana) da como resultado una estimación de probabilidad demasiado baja para valores pequeños de error k. El codificador aritmético produciría un código innecesariamente largo en tal caso. Por otro lado, utilizando una varianza pequeña (que corresponde a una distribución alta y estrecha) asignaría probabilidades demasiado bajas a valores grandes de error k. La elección de la varianza, por tanto, es importante. Un método ideal para estimar la varianza debe asignar el mejor valor de varianza de cada error y no debe acarrear ninguna sobrecarga (i.e., no debe escribirse ningún dato adicional en el flujo comprimido para ayudar al decodificador a estimar las varianzas). Éstos son algunos planteamientos para la selección de la varianza:

1. La distribución de Laplace fue adoptada como la distribución de errores de MLP, después de efecuar muchos experimentos con imágenes reales. La distribución obtenida por todas esas imágenes tiene un cierto valor de V, y este valor podría ser utilizado por MLP. Éste es un método sencillo, que puede ser utilizado en las versiones rápidas de MLP. Sin embargo, no es adaptativo (ya que siempre utiliza la misma varianza), y por tanto no ofrece el mejor rendimiento en compresión para todas las imágenes.

2. (Un trabajo de compresión de dos pasos.) Cada compresión debe comenzar con una pasada en la que se calculan todos los valores de error y su varianza (véase más abajo). Esta variación debe utilizarse, tras la primera pasada, para calcular una tabla de probabilidades a partir de la Ecuación (4.39). La tabla debe ser utilizada en la segunda pasada, donde los valores de error son codificados y escritos en la secuencia de datos comprimidos. El rendimiento de la compresión será excelente, pero cualquier trabajo de dos pasos es lento. Nótese que la tabla entera puede ser escrita al principio del *stream* comprimido, lo que simplifica enormemente la tarea del decodificador.

◇ **Ejercicio 4.39 (sol. en pág. 1086):** Muéstrese un ejemplo en este enfoque es práctico (i.e., cuando la codificación lenta no es importante, pero un decodificador rápido y una excelente compresión son importantes).

3. Cada vez que se obtiene un error y está a punto de ser codificado aritméticamente, utilizar algún método para estimar la varianza asociada con ese error. Cuantificar la estimación y utilizar un número de tablas de probabilidad precalculadas, una para cada valor de varianza cuantificada, para calcular la probabilidad del error. Ésto es computacionalmente intensivo, pero puede ser un buen compromiso entre los enfoques 1 y 2 anteriores.

Ahora necesitamos discutir cómo puede ser estimada la varianza de un error, y comenzamos con una explicación del concepto de varianza. La varianza es un concepto estadístico definido para una secuencia de valores $a_1, a_2, \ldots, a_n$. Mide cómo varían los elementos a_i calculando las diferencias entre ellos y el promedio A de la secuencia, que viene dado, como siempre, por $A = (1/n)\sum a_i$. Ésta es la razón por la cual las curvas de la Figura 4.128b que corresponden a varianzas pequeñas son más estrechas; sus valores se concentran cercanos a la media, que en éste caso es cero. La secuencia $(5, 5, 5)$, por ejemplo, tiene un promedio de 5 y una varianza 0, porque cada elemento de la secuencia es igual a la media. La secuencia $(0, 5, 10)$ también tiene promedio 5 pero debe tener una varianza distinta de cero, porque dos de sus elementos difieren de la media. En general, la varianza de la secuencia a_i se define como la cantidad no negativa:

$$V = \sigma^2 = E\left(a_i - A\right)^2 = \frac{1}{n}\sum_1^n \left(a_i - A\right)^2,$$

por lo que la varianza de $(0, 5, 10)$ es $\left[(0-5)^2+(5-5)^2+(10-5)^2\right]/3 = 50/3$. Los estadísticos también utilizan una cantidad llamada *desviación estándar* (denotada por σ) que se define como la raíz cuadrada de la varianza.

Ahora discutiremos varias maneras de estimar la varianza de una predicción de error E.

3.1. La Ecuación (4.39) da la probabilidad de un error E con un valor k, pero esta probabilidad depende de V. Podemos considerar una función $P_V(k)$ de las dos variables, V y k, y encontrar el valor óptimo de V resolviendo la ecuación: $\partial P_V(k)/\partial V = 0$. La solución es $V = 2k^2$, pero este método no es práctico, porque el decodificador no conoce k (está tratando de decodificar k por lo que sabe encontrar el valor del píxel P), y por lo tanto, no puede reflejar las operaciones del codificador. Es posible escribir los valores de todas las varianzas en el *stream* comprimido, pero ésto reduciría significativamente la relación de compresión. Este método puede ser utilizado para codificar una imagen en particular con el fin de encontrar la mejor relación de compresión de la imagen y compararla con lo que se consigue en la práctica.

3.2. Mientras se están codificando los píxeles de un nivel, se examinan sus errores E formando una secuencia de números, y se encuentra su varianza V. Se utiliza V para codificar los píxeles del siguiente nivel. El número de niveles no es muy grande, por lo que todos los valores de varianza se pueden escribir (codificados aritméticamente) en el *stream* comprimido, lo que deriva en una rápida decodificación. La varianza se usa para codificar el primer nivel, y debería ser un parámetro controlado por el usuario, cuyo valor no es crítico, porque ese nivel contiene sólo unos pocos píxeles. Puesto que MLP cuantifica una varianza a uno de 37 valores (véase más adelante), cada varianza que se escribe en el flujo comprimido es codificada en sólo $\log_2 37 \approx 5{,}21$ bits, una sobrecarga insignificante. La desventaja obvia de este método es que no tiene en cuenta las concentraciones locales de píxeles idénticos o muy similares en el mismo nivel.

3.3. (Similar a 3.2.) Mientras los píxeles de un nivel están siendo codificados, se recogen los errores de predicción de cada bloque de $b \times b$ píxeles y se usan para calcular una varianza que será utilizada

para codificar los píxeles internos a este bloque, en el siguiente nivel. Los valores de varianza para un nivel también se pueden escribir en el *stream* comprimido después de todos los errores de codificación para ése nivel, por lo que el decodificador podría usarlos sin tener que hacer cálculos. El parámetro b debe ser ajustado mediante experimentos, y los autores recomiendan el valor $b = 16$. Este método acarrea gastos generales significativos y, por consiguiente, puede degradar el rendimiento de la compresión.

3.4. (Esta es una adición posterior a MLP, véase [Howard y Vitter 92b].) Tanto el codificador como el decodificador calculan un *índice de variabilidad* para cada píxel. Este índice depende de la cantidad en que el píxel difiere de sus vecinos cercanos. Los índices de variabilidad de todos los píxeles en un nivel se utilizan adaptativamente para estimar las varianzas de los píxeles, basadas en la asunción de que píxeles con índices de variabilidad similares deben usar distribuciones de Laplace con varianzas similares. El método procede con los siguientes pasos:

1. Los índices de variabilidad se calculan para todos los píxeles del nivel actual, basándose en los valores de los píxeles en los niveles precedentes. Ésto es realizado por el codificador y posteriormente reflejado por el decodificador. Después de varios intentos, los desarrolladores de MLP han establecido una forma sencilla de calcular el índice de variabilidad. Éste se calcula como la varianza de los cuatro vecinos más próximos del píxel actual (los vecinos proceden de los niveles precedentes, por lo que el decodificador puede reflejar esta operación).
2. La estimación de la varianza V se establece en un valor inicial. La elección de este valor no es crítico, ya que V va a ser actualizado a menudo posteriormente. El decodificador elige dicho valor de la misma manera.
3. Los píxeles del nivel actual se ordenan en orden según el índice de variabilidad. El decodificador puede reflejar ésto a pesar de que aún no tiene los valores de estos píxeles (el decodificador ya ha calculado los valores del índice de variabilidad en el paso 1, ya que dependen de los píxeles de los niveles precedentes).
4. El codificador revisa los píxeles ordenados en orden decreciente (desde los índices de variabilidad más grandes, a los más pequeños). Para cada píxel:
   4.1. El codificador calcula el error E del píxel y envía E y V al codificador aritmético. El decodificador refleja este paso. Conoce V, por lo que puede decodificar E.
   4.2. El codificador acualiza V mediante:

$$V \leftarrow f \times V + (1 - f)\, E^2,$$

donde f es un parámetro de suavizado (la experiencia aconseja un valor grande, tal como 0,99, para f). Así es como se adapta V, píxel a píxel, utilizando los errores E. Debido al gran valor de f, V disminuye en pasos pequeños. Ésto significa que los últimos píxeles (los que tienen índices de variabilidad pequeños) tendrán asignadas varianzas pequeñas. La idea es que la compresión de los píxeles con índices de variabilidad grandes es menos sensible a valores precisos de V.

A medida que el ciclo avanza, a V se le asignan valores más precisos, y éstos se utilizan para comprimir los píxeles con índices de variabilidad pequeños, que son más sensibles a los valores de la varianza. Observe que el decodificador puede reflejar este paso, puesto que ya ha decodificado E en el paso 4.1. Note también que el codificador aritmético escribe los valores de error codificados en el *stream* comprimido en orden decreciente del índice de variabilidad, no fila por fila. El decodificador puede reflejar ésto también, puesto que ya ha ordenado los píxeles en este orden en el paso 3.

```
para cada nivel L hacer
  para cada píxel P en el nivel L hacer
    Calcular una predicción R para P usando un grupo de nivel L−1;
    Calcular E=R−P;
    Estimar la varianza V para ser utilizada en la codificación de E;
    Cuantificar V y usarlo como un índice para seleccionar una tabla de Laplace LV;
    Utilizar E como un índice de la tabla LV y recuperar LV[E];
    Utilizar LV[E] como la probabilidad para codificar aritméticamente E;
  finpara;
Determinar los píxeles del siguiente nivel (rotar & escalar);
finpara;
```

Tabla 4.129: Codificación MLP.

Rango de varianza	Var. usada	Rango de varianza	Var. usada	Rango de varianza	Var. usada
0,005 – 0,023	0,016	2,882 – 4,053	3,422	165,814 – 232,441	195,569
0,023 – 0,043	0,033	4,053 – 5,693	4,809	232,441 – 326,578	273,929
0,043 – 0,070	0,056	5,693 – 7,973	6,747	326,578 – 459,143	384,722
0,070 – 0,108	0,088	7,973 – 11,170	9,443	459,143 – 645,989	540,225
0,108 – 0,162	0,133	11,170 – 15,627	13,219	645,989 – 910,442	759,147
0,162 – 0,239	0,198	15,627 – 21,874	18,488	910,442 – 1285,348	1068,752
0,239 – 0,348	0,290	21,874 – 30,635	25,875	1285,348 – 1816,634	1506,524
0,348 – 0,502	0,419	30,635 – 42,911	36,235	1816,634 – 2574,021	2125,419
0,502 – 0,718	0,602	42,911 – 60,123	50,715	2574,021 – 3663,589	3007,133
0,718 – 1,023	0,859	60,123 – 84,237	71,021	3663,589 – 5224,801	4267,734
1,023 – 1,450	1,221	84,237 – 118,157	99,506	5224,801 – 7247,452	6070,918
1,450 – 2,046	1,726	118,157 – 165,814	139,489	7247,452 – 10195,990	8550,934
2,046 – 2,882	2,433				

Tabla 4.130: Treinta y Siete valores de varianza cuantificados.

Este método da excelentes resultados, pero es aún más intensivo computacionalmente que el MLP original (fin del método 3.4).

Para estimar la varianza V, se utiliza uno de los cuatro métodos anteriores. Antes de usar V para codificar el error E, se cuantifica V a uno de 37 valores, como se muestra en la Tabla 4.130. Por ejemplo, si el valor estimado de la varianza es 0,31, se cuantifica a 7. El valor cuantificado se usa entonces para seleccionar una de las 37 tablas de probabilidad precalculadas (en nuestro ejemplo, se selecciona la Tabla 7, precalculada para un valor de varianza de 0,290) preparadas utilizando la Ecuación (4.39), y se utiliza el valor del error E para indexar esa tabla. (Las tablas de probabilidad no se muestran aquí.) El valor recuperado de la tabla es la probabilidad que se envía al codificador aritmético, junto con el valor del error, para codificar aritméticamente el error E.

MLP es, por lo tanto, uno de los muchos métodos de compresión que implementan un modelo para estimar las probabilidades y usar la codificación aritmética para efectuar la compresión real.

La Tabla 4.129 presenta un resumen de pseudocódigo de la codificación MLP.

4.21.4. Polinomios de interpolación

En esta sección se muestra cómo predecir el valor de un píxel a partir de 16 de sus vecinos más cercanos por medio de un polinomio de interpolación bidimensional. Los resultados se utilizan en la Tabla 4.126.

Comenzamos con una discusión intuitiva del término *interpolación*. Dados dos números a y b, su promedio $(a+b)/2$ se encuentra siempre entre ellos, en la mitad, por lo que podemos utilizar el promedio para interpolarlos. Sin embargo, dados cuatro números a, b, c, y d, su promedio $(a+b+c+d)/4$ no es una buena interpolación, porque no se encuentra "a medio camino" de los cuatro. Un ejemplo sencillo es el de los cuatro números: 1, 1, 1, y 100. Su promedio está cercano a 25, por lo que no está en absoluto "a medio camino" de los cuatro números. La interpolación de los cuatro números se efectúa por tanto: (1) convirtiendo los números a puntos bidimensionales, (2) calculando una curva suave que pase por los puntos, y (3) encontrando el punto medio de la curva.

Cualesquier números a, b, c, y d pueden convertirse en los puntos: $(1,a)$, $(2,b)$, $(3,c)$, y $(4,d)$. Es intuitivamente claro que el punto medio (x,y) de una curva suave que pasa a través de estos puntos es un buen candidato para el título "*interpolación de cuatro puntos*". La coordenada y se convierte en la interpolación de los cuatro números, y la coordenada x es ignorada.

Este método se denomina interpolación unidimensional. Puede extenderse a más de cuatro números, y también a píxeles, donde se convierte en una interpolación bidimensional. Como se mencionó antes, queremos utilizar un grupo de 16 píxeles vecinos para predecir el valor de un píxel en el centro del grupo. La idea principal es considerar los 16 píxeles vecinos un conjunto de 4×4 puntos igualmente espaciados sobre una superficie (donde el valor de un píxel se interpreta como la altura de la superficie) y derivar una función polinomio $\mathbf{P}(u,w)$ que pase a través de todos los 16 puntos. Gráficamente, $\mathbf{P}(u,w)$ puede ser considerado como una superficie. El valor del píxel en el centro del grupo de 4×4 puede entonces predecirse mediante el cálculo de la altura del punto central $\mathbf{P}(0{,}5, 0{,}5)$ de la superficie. Matemáticamente, esta superficie es la interpolación polinómica bidimensional de los 16 puntos.

4.21.5. Interpolación unidimensional

Una superficie puede ser vista como una extensión de una curva, por lo que comenzamos derivando un polinomio unidimensional (una curva) que interpola cuatro puntos, y luego se expande a un polinomio bidimensional (una superficie) que interpola una cuadrícula de 4×4 puntos.

Dados cuatro puntos $\mathbf{P}_1$, $\mathbf{P}_2$, $\mathbf{P}_3$, y $\mathbf{P}_4$ buscamos un polinomio que pase por ellos. En general, un polinomio de grado n en x se define (Sección 3.28) como la función:

$$P_n(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n, \tag{4.40}$$

donde los a_i son los $n+1$ coeficientes del polinomio y el parámetro x es un número real. El polinomio de interpolación unidimensional que nos interesa es especial, y difiere de la definición anterior en dos aspectos:

1. Este polinomio va desde el punto $\mathbf{P}_1$ al $\mathbf{P}_4$. Su longitud es finita, por lo que es mejor describirlo como la función:

$$P_n(t) = \sum_{i=0}^{n} a_i t^i = a_0 + a_1 t + a_2 t^2 + \cdots + a_n t^n;\ \text{donde } 0 \le t \le 1.$$

   Ésta es la *representación paramétrica* de un polinomio. Queremos que este polinomio vaya desde $\mathbf{P}_1$ a $\mathbf{P}_4$ cuando el parámetro t varíe de 0 a 1.

2. Los únicos datos dados son los cuatro puntos y tenemos que utilizarlos para calcular todos los $n+1$ coeficientes del polinomio. Ésto sugiere el valor $n=3$ (un polinomio de grado 3, un polinomio cúbico; uno que tiene cuatro coeficientes). La idea es plantear y resolver cuatro ecuaciones, con los cuatro coeficientes como incógnitas, y con los cuatro puntos como cantidades conocidas. En consecuencia, utilizamos la notación (T indica transposición):

$$\mathbf{P}(t)=\mathbf{a}t^3+\mathbf{b}t^2+\mathbf{c}t+\mathbf{d}=\left(t^3,t^2,t,1\right)(\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d})^T=\mathbf{T}(t)\cdot\mathbf{A}. \tag{4.41}$$

Los cuatro coeficientes **a**, **b**, **c**, **d** se muestran en negrita porque no son números. Tenga en cuenta que el polinomio tiene que pasar por los puntos dados, por lo que el valor de $\mathbf{P}(t)$ para cualquier t debe ser las tres coordenadas de un punto. Cada coeficiente, por lo tanto, debe ser un triplete. $\mathbf{T}(t)$ es el vector fila $\left(t^3,t^2,t,1\right)$, y $\mathbf{A}$ es el vector columna $(\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d})^T$. Por consiguiente, el cálculo de la curva implica encontrar los valores de las cuatro incógnitas: **a**, **b**, **c**, y **d**. $\mathbf{P}(t)$ se denomina polinomio *cúbico paramétrico* (o PC).

Resulta que el grado 3 es el más pequeño que sigue siendo útil para un polinomio de interpolación. Un polinomio de grado 1 tiene la forma $\mathbf{P}_1(t)=\mathbf{c}t+\mathbf{d}$ y es, por tanto, una línea recta, por lo que se puede utilizar solamente en casos especiales. Un polinomio de grado 2 (cuadrático) tiene la forma $\mathbf{P}_2(t)=\mathbf{b}t^2+\mathbf{c}t+\mathbf{d}$ y es una sección cónica, por lo que sólo puede tomar unas pocas formas diferentes. Un polinomio de grado 3 (cúbico) es, por lo tanto, la más simple que puede tomar formas complejas, y también puede ser una curva en el espacio.

◇ **Ejercicio 4.40 (sol. en pág. 1086):** Demuéstrese que un polinomio cuadrático debe ser una curva plana.

Nuestro problema fundamental es interpolar los píxeles. Los píxeles están siempre espaciados uniformemente, por lo que asumimos que los dos puntos interiores $\mathbf{P}_2$ y $\mathbf{P}_3$ están igualmente espaciados entre $\mathbf{P}_1$ y $\mathbf{P}_4$. El primer punto $\mathbf{P}_1$ es el punto de inicio $\mathbf{P}(0)$ del polinomio; el último punto, $\mathbf{P}_4$ es el punto final $\mathbf{P}(1)$, y los dos puntos interiores $\mathbf{P}_2$ y $\mathbf{P}_3$ son los dos puntos interiores igualmente espaciados $\mathbf{P}(1/3)$ y $\mathbf{P}(2/3)$ del polinomio.

Por consiguiente, escribimos $\mathbf{P}(0)=\mathbf{P}_1$, $\mathbf{P}(1/3)=\mathbf{P}_2$, $\mathbf{P}(2/3)=\mathbf{P}_3$, $\mathbf{P}(1)=\mathbf{P}_4$, o

$$\begin{aligned}
\mathbf{a}(0)^3+\mathbf{b}(0)^2+\mathbf{c}(0)+\mathbf{d}&=\mathbf{P}_1,\\
\mathbf{a}(1/3)^3+\mathbf{b}(1/3)^2+\mathbf{c}(1/3)+\mathbf{d}&=\mathbf{P}_2,\\
\mathbf{a}(2/3)^3+\mathbf{b}(2/3)^2+\mathbf{c}(2/3)+\mathbf{d}&=\mathbf{P}_3,\\
\mathbf{a}(1)^3+\mathbf{b}(1)^2+\mathbf{c}(1)+\mathbf{d}&=\mathbf{P}_4.
\end{aligned}$$

Éstas ecuaciones son fáciles de resolver, y las soluciones son:

$$\begin{aligned}
\mathbf{a}&=-9/2\mathbf{P}_1+27/2\mathbf{P}_2-27/2\mathbf{P}_3+9/2\mathbf{P}_4,\\
\mathbf{b}&=9\mathbf{P}_1-45/2\mathbf{P}_2+18\mathbf{P}_3-9/2\mathbf{P}_4,\\
\mathbf{c}&=-11/2\mathbf{P}_1+9\mathbf{P}_2-9/2\mathbf{P}_3+\mathbf{P}_4,\\
\mathbf{d}&=\mathbf{P}_1.
\end{aligned}$$

Sustituyendo en la Ecuación (4.41) obtenemos:

$$\begin{aligned}
\mathbf{P}(t)=&\left(-9/2\mathbf{P}_1+27/2\mathbf{P}_2-27/2\mathbf{P}_3+9/2\mathbf{P}_4\right)t^3\\
&+\left(9\mathbf{P}_1-45/2\mathbf{P}_2+18\mathbf{P}_3-9/2\mathbf{P}_4\right)t^2\\
&+\left(-11/2\mathbf{P}_1+9\mathbf{P}_2-9/2\mathbf{P}_3+\mathbf{P}_4\right)t+\mathbf{P}_1,
\end{aligned}$$

que, tras reordenarla, se convierte en:

$$\begin{aligned}\mathbf{P}(t) =& \left(-4{,}5t^3 + 9t^2 - 5{,}5t + 1\right)\mathbf{P}_1 + \left(13{,}5t^3 - 22{,}5t^2 + 9t\right)\mathbf{P}_2 \\ &+ \left(-13{,}5t^3 + 18t^2 - 4{,}5t\right)\mathbf{P}_3 + \left(4{,}5t^3 - 4{,}5t^2 + t\right)\mathbf{P}_4 \\ =& G_1(t)\,\mathbf{P}_1 + G_2(t)\,\mathbf{P}_2 + G_3(t)\,\mathbf{P}_3 + G_4\mathbf{P}_4 \\ =& \mathbf{G}(t)\cdot\mathbf{P}, \end{aligned} \tag{4.42}$$

donde:

$$\begin{aligned} G_1(t) &= \left(-4{,}5t^3 + 9t^2 - 5{,}5t + 1\right), & G_2(t) &= \left(13{,}5t^3 - 22{,}5t^2 + 9t\right), \\ G_3(t) &= \left(-13{,}5t^3 + 18t^2 - 4{,}5t\right), & G_4(t) &= \left(4{,}5t^3 - 4{,}5t^2 + t\right); \end{aligned} \tag{4.43}$$

$\mathbf{P}$ es la columna $(\mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3, \mathbf{P}_4)^T$, y $\mathbf{G}(t)$ es el vector fila:

$$(G_1(t), G_2(t), G_3(t), G_4(t)).$$

Las funciones $G_i(t)$ se denominan *funciones de blending (de mezcla o de fusión)*, porque crean cualquier punto de la curva como una mixtura de los cuatro puntos propuestos. Observe que su suma es 1, para cualquier valor de t. Esta propiedad debe ser satisfecha por cualquier conjunto de funciones de mezcla, y tales funciones se denominan *baricéntricas*. También podemos escribir:

$$G_1(t) = \left(t^3, t^2, t, 1\right)(-4{,}5,\ 9, -5{,}5,\ 1)^T$$

y, similarmente, para $G_2(t)$, $G_3(t)$, y $G_4(t)$. En notación matricial, ésto se expresa así:

$$\mathbf{G}(t) = \left(t^3, t^2, t, 1\right)\begin{pmatrix} -4{,}5 & 13{,}5 & -13{,}5 & 4{,}5 \\ 9{,}0 & -22{,}5 & 18 & -4{,}5 \\ -5{,}5 & 9{,}0 & -4{,}5 & 1{,}0 \\ 1{,}0 & 0 & 0 & 0 \end{pmatrix} = \mathbf{T}(t)\cdot\mathbf{N}. \tag{4.44}$$

La curva puede escribirse $\mathbf{P}(t) = \mathbf{G}(t)\cdot\mathbf{P} = \mathbf{T}(t)\cdot\mathbf{N}\cdot\mathbf{P}$. $\mathbf{N}$ se llama matriz de la base, y $\mathbf{P}$ es el vector geométrico. De la Ecuación (4.41) sabemos que $\mathbf{P}(t) = \mathbf{T}(t)\cdot\mathbf{A}$, por lo que podemos escribir $\mathbf{A} = \mathbf{N}\cdot\mathbf{P}$. La Ecuación (5.17) muestra una aplicación de este polinomio de interpolación para la compresión de imágenes.

> La palabra *baricéntrico* se deriva de *baricentro*, que significa "centro de gravedad", porque tales pesos se utilizan para calcular el centro de gravedad de un objeto. Los pesos baricéntricos tienen muchos usos en la geometría en general, y en curvas y diseño de superficies en particular.

Dados cuatro puntos, el polinomio de interpolación se puede calcular en dos pasos:

1. Establecer la ecuación $\mathbf{A} = \mathbf{N}\cdot\mathbf{P}$ y resolverla para $\mathbf{A} = (\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d})^T$.
2. El polinomio es: $\mathbf{P}(t) = \mathbf{T}(t)\cdot\mathbf{A}$.

4.21.6. Ejemplo

(Este ejemplo es bidimensional, cada uno de los cuatro puntos $\mathbf{P}_i$ y cada uno de los cuatro coeficientes $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, y $\mathbf{d}$ es un par. Para curvas tridimensionales, el método es el mismo, excepto que

deben utilizarse tripletes, en lugar de pares.) Dados los cuatro puntos en dos dimensiones $\mathbf{P}_1 = (0,0)$, $\mathbf{P}_2 = (1,0)$, $\mathbf{P}_3 = (1,1)$, y $\mathbf{P}_4 = (0,1)$, creamos la ecuación:

$$\begin{pmatrix} \mathbf{a} \\ \mathbf{b} \\ \mathbf{c} \\ \mathbf{d} \end{pmatrix} = \mathbf{A} = \mathbf{N} \cdot \mathbf{P} = \begin{pmatrix} -4{,}5 & 13{,}5 & -13{,}5 & 4{,}5 \\ 9{,}0 & -22{,}5 & 18 & -4{,}5 \\ -5{,}5 & 9{,}0 & -4{,}5 & 1{,}0 \\ 1{,}0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} (0,0) \\ (1,0) \\ (1,1) \\ (0,1) \end{pmatrix},$$

que es fácil de resolver:

$$\begin{aligned} \mathbf{a} &= -4{,}5\,(0,0) + 13{,}5\,(1,0) - 13{,}5\,(1,1) + 4{,}5\,(0,1) = (0,-9)\,, \\ \mathbf{b} &= 9\,(0,0) - 22{,}5\,(1,0) + 18\,(1,1) - 4{,}5\,(0,1) = (-4{,}5, 13{,}5)\,, \\ \mathbf{c} &= -5{,}5\,(0,0) + 9\,(1,0) - 4{,}5\,(1,1) + 1\,(0,1) = (4{,}5, -3{,}5)\,, \\ \mathbf{d} &= 1\,(0,0) - 0\,(1,0) + 0\,(1,1) - 0\,(0,1) = (0,0)\,. \end{aligned}$$

Por consiguiente:

$$\mathbf{P}(t) = \mathbf{T} \cdot \mathbf{A} = (0,-9)\,t^3 + (-4{,}5, 13{,}5)\,t^2 + (4{,}5, -3{,}5)\,t.$$

Ahora es fácil calcular y verificar que:

$$\begin{gathered} \mathbf{P}(0) = (0,0) = \mathbf{P}_1, \\ \mathbf{P}(1/3) = (0,-9)\,1/27 + (-4{,}5, 13{,}5)\,1/9 + (4{,}5, -3{,}5)\,1/3 = (1,0) = \mathbf{P}_2, \text{ y} \\ \mathbf{P}(1) = (0,-9)\,1^3 + (-4{,}5, 13{,}5)\,1^2 + (4{,}5, -3{,}5)\,1 = (0,1) = \mathbf{P}_4. \end{gathered}$$

◇ **Ejercicio 4.41 (sol. en pág. 1086):** Calcúlese $\mathbf{P}(2/3)$ y verifíquese que es igual a $\mathbf{P}_3$.

◇ **Ejercicio 4.42 (sol. en pág. 1086):** Imagínese el arco circular de radio uno en el primer cuadrante (un cuarto de círculo). Escríbanse las coordenadas de los cuatro puntos que están igualmente espaciados en este arco. Utilícense las coordenadas para calcular un polinomio PC de interpolación para aproximar dicho arco. Calcúlese el punto $\mathbf{P}(1/2)$. ¿Qué distancia le separa del centro real del cuarto de círculo?

La principal ventaja de este método es su simplicidad. Dados cuatro puntos, es fácil calcular el polinomio PC que pasa a través de ellos.

◇ **Ejercicio 4.43 (sol. en pág. 1086):** Este método tiene sentido si los cuatro puntos están (al menos aproximadamente) igualmente espaciados a lo largo de la curva. Si no están igualmente espaciados, puede hacerse lo siguiente: En lugar de utilizar $1/3$ y $2/3$ como valores intermedios, el usuario puede especificar valores α, β tales que $\mathbf{P}_2 = \mathbf{P}(\alpha)$ y $\mathbf{P}_3 = \mathbf{P}(\beta)$. Generalícese la Ecuación (4.44) para que dependa de α y β.

4.21.7. Interpolación bidimensional

El polinomio PC, Ecuación (4.41), puede extenderse fácilmente a dos dimensiones por medio de una técnica llamada *producto Cartesiano*. El polinomio se generaliza a partir de una curva cúbica, a una superficie *bicúbica*.

Un polinomio PC unidimensional tiene la forma $\mathbf{P}(t) = \sum_{i=0}^{3} \mathbf{a}_i t^i$. Dos de tales curvas, $\mathbf{P}(u)$ y

$\mathbf{P}(\omega)$, pueden combinarse mediante esta técnica para formar la superficie:

$$\begin{aligned}\mathbf{P}(u,\omega) &= \sum_{i=0}^{3}\sum_{j=0}^{3}\mathbf{a}_{ij}u^i\omega^j \\ &= \mathbf{a}_{33}u^3\omega^3+\mathbf{a}_{32}u^3\omega^2+\mathbf{a}_{31}u^3\omega+\mathbf{a}_{30}\omega^3+\mathbf{a}_{23}u^2\omega^3+\mathbf{a}_{22}u^2\omega^2+\mathbf{a}_{21}u^2\omega+\mathbf{a}_{20}u^2 \\ &+ \mathbf{a}_{13}u\omega^3+\mathbf{a}_{12}u\omega^2+\mathbf{a}_{11}u\omega+\mathbf{a}_{10}u+\mathbf{a}_{03}\omega^3+\mathbf{a}_{02}u\omega^2+\mathbf{a}_{01}u\omega+\mathbf{a}_{00} \\ &= \left(u^3,u^2,u,1\right)\begin{pmatrix}\mathbf{a}_{33}&\mathbf{a}_{32}&\mathbf{a}_{31}&\mathbf{a}_{30}\\ \mathbf{a}_{23}&\mathbf{a}_{22}&\mathbf{a}_{21}&\mathbf{a}_{20}\\ \mathbf{a}_{13}&\mathbf{a}_{12}&\mathbf{a}_{11}&\mathbf{a}_{10}\\ \mathbf{a}_{03}&\mathbf{a}_{02}&\mathbf{a}_{01}&\mathbf{a}_{00}\end{pmatrix}\begin{pmatrix}\omega^3\\ \omega^2\\ \omega\\ 1\end{pmatrix}, \quad \text{donde } 0\le u,\omega\le 1. \end{aligned} \tag{4.45}$$

Éste es un polinomio cúbico doble (de ahí el nombre de *bicúbico*) con 16 términos, donde cada uno de los 16 coeficientes $\mathbf{a}_{ij}$ es un triplete. Observe que la superficie depende de todos los 16 coeficientes. Cualquier cambio en cualquiera de ellos produce una superficie diferente. La Ecuación (4.45) es la *representación algebraica* de una superficie bicúbica. Con el fin de utilizarlo en la práctica, los 16 coeficientes desconocidos tienen que ser expresados en términos de 16 puntos (igualmente espaciados) conocidos. Denotamos estos puntos:

$$\begin{array}{cccc}\mathbf{P}_{03}&\mathbf{P}_{13}&\mathbf{P}_{23}&\mathbf{P}_{33}\\ \mathbf{P}_{02}&\mathbf{P}_{12}&\mathbf{P}_{22}&\mathbf{P}_{32}\\ \mathbf{P}_{01}&\mathbf{P}_{11}&\mathbf{P}_{21}&\mathbf{P}_{31}\\ \mathbf{P}_{00}&\mathbf{P}_{10}&\mathbf{P}_{20}&\mathbf{P}_{30}.\end{array}$$

Para determinar los 16 coeficientes desconocidos, podemos escribir las 16 ecuaciones, cada una basada en uno de los puntos dados:

$$\begin{array}{cccc}\mathbf{P}(0,0)=\mathbf{P}_{00}&\mathbf{P}(0,1/3)=\mathbf{P}_{01}&\mathbf{P}(0,2/3)=\mathbf{P}_{02}&\mathbf{P}(0,1)=\mathbf{P}_{03}\\ \mathbf{P}(1/3,0)=\mathbf{P}_{10}&\mathbf{P}(1/3,1/3)=\mathbf{P}_{11}&\mathbf{P}(1/3,2/3)=\mathbf{P}_{12}&\mathbf{P}(1/3,1)=\mathbf{P}_{13}\\ \mathbf{P}(2/3,0)=\mathbf{P}_{20}&\mathbf{P}(2/3,1/3)=\mathbf{P}_{21}&\mathbf{P}(2/3,2/3)=\mathbf{P}_{22}&\mathbf{P}(2/3,1)=\mathbf{P}_{23}\\ \mathbf{P}(1,0)=\mathbf{P}_{30}&\mathbf{P}(1,1/3)=\mathbf{P}_{31}&\mathbf{P}(1,2/3)=\mathbf{P}_{32}&\mathbf{P}(1,1)=\mathbf{P}_{33}.\end{array}$$

Resolviendo, sustituyendo las soluciones en la Ecuación (4.45), y simplificando, aparece la *representación geométrica* de la superficie bicúbica:

$$\mathbf{P}(u,\omega)=\left(u^3,u^2,u,1\right)\mathbf{N}\begin{pmatrix}\mathbf{P}_{33}&\mathbf{P}_{32}&\mathbf{P}_{31}&\mathbf{P}_{30}\\ \mathbf{P}_{23}&\mathbf{P}_{22}&\mathbf{P}_{21}&\mathbf{P}_{20}\\ \mathbf{P}_{13}&\mathbf{P}_{12}&\mathbf{P}_{11}&\mathbf{P}_{10}\\ \mathbf{P}_{03}&\mathbf{P}_{02}&\mathbf{P}_{01}&\mathbf{P}_{00}\end{pmatrix}\mathbf{N}^T\begin{pmatrix}\omega^3\\ \omega^2\\ \omega\\ 1\end{pmatrix}, \tag{4.46}$$

donde $\mathbf{N}$ es a matriz de Hermite de la Ecuación (4.44).

La superficie de la Ecuación (4.46) ahora se puede utilizar para predecir el valor de un píxel como un polinomio de interpolación de 16 de sus vecinos más próximos. Todo lo que se necesita es sustituir $u=0{,}5$ y $\omega=0{,}5$. El siguiente código en *Mathematica*:

```
Clear[Nh,P,U,W];
Nh={{-4.5,13.5,-13.5,4.5},{9,-22.5,18,-4.5},
    {-5.5,9,-4.5,1},{1,0,0,0}};
P={{p33,p32,p31,p30},{p23,p22,p21,p20},
    {p13,p12,p11,p10},{p03,p02,p01,p00}};
U={u^3,u^2,u,1};
W={w^3,w^2,w,1};
u:=0.5;
w:=0.5;
Expand[U.Nh.P.Transpose[Nh].Transpose[W]]
```

efectúa ésto y produce:

$$\begin{aligned}\mathbf{P}&(0{,}5, 0{,}5)\\ &= 0{,}00390625\mathbf{P}_{00} - 0{,}0351563\mathbf{P}_{01} - 0{,}0351563\mathbf{P}_{02} + 0{,}00390625\mathbf{P}_{03}\\ &- 0{,}0351563\mathbf{P}_{10} - 0{,}316406\mathbf{P}_{11} - 0{,}316406\mathbf{P}_{12} + 0{,}0351563\mathbf{P}_{13}\\ &- 0{,}0351563\mathbf{P}_{20} - 0{,}316406\mathbf{P}_{21} - 0{,}316406\mathbf{P}_{22} + 0{,}0351563\mathbf{P}_{23}\\ &+ 0{,}00390625\mathbf{P}_{30} - 0{,}0351563\mathbf{P}_{31} - 0{,}0351563\mathbf{P}_{32} + 0{,}00390625\mathbf{P}_{33},\end{aligned}$$

donde los 16 coeficientes son los usados en la Tabla 4.126.

◇ **Ejercicio 4.44 (sol. en pág. 1087):** ¿Cómo puede utilizarse este método en aquellos casos en que no se conocen todos los 16 puntos?

◇ **Ejercicio 4.45 (sol. en pág. 1087):** El punto central de la superficie se calcula como una suma ponderada de los 16 puntos de datos igualmente espaciados. Tiene sentido asignar pesos pequeños a los puntos ubicados fuera del centro, pero nuestro resultado asigna ponderaciones *negativas* a ocho de los 16 puntos. Explíquese el significado de los pesos negativos y muéstrese cuál es el papel que desempeñan en la interpolación del centro de la superficie.

Los lectores que tengan dificultades para seguir los detalles de arriba deberían comparar la forma en que se presenta aquí la interpolación polinómica bidimensional, con la forma que se discute en [Press et al. 88]. La siguiente cita es de la página 125: "... Las fórmulas que obtengan las c's de la función y valores derivados son precisamente una transformación lineal compleja, con coeficientes que, una vez que han sido determinados, en la bruma de la historia numérica, pueden tabularse y ser olvidados."

4.22. Golomb adaptativo

El principio de codificación *run-length* es sencillo. Dada una imagen binivel, se escanea fila por fila y se calculan los run lengths de píxeles negros y blancos alternos. Cada run length está en el intervalo $[1, r]$, donde r es el tamaño de la fila. En principio, los run lengths constituyen la imagen comprimida, pero en la práctica necesitamos escribirlos en el *stream* comprimido de tal manera que el decodificador sea capaz de leerlos sin ambigüedades. Ésto se hace reemplazando cada run length por un código prefijo y escribiendo los códigos en la secuencia de datos comprimidos sin ningún separador. En esta sección se muestra que los códigos de Golomb (Sección 2.5) son ideales para esta aplicación. Los códigos de Golomb dependen de un parámetro m, por lo que también propone un sencillo algoritmo adaptativo para estimar m para cada run length con el fin de obtener la mejor compresión en un algoritmo de una pasada.

La Figura 4.131 es una sencilla máquina de estados finitos (véase la sección 8.8 para más información) que resume el proceso de lectura de píxeles negros y blancos de una imagen. En cualquier

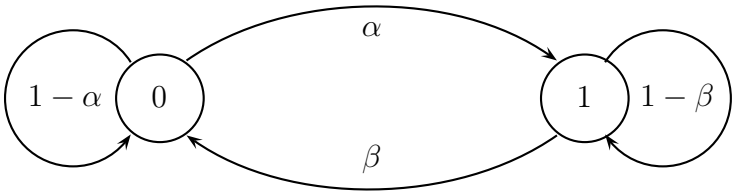


Figura 4.131: Una máquina de estados finitos para la lectura de píxeles binivel.

momento, la máquina puede estar en uno de dos estados: un 0 (blanco) o un 1 (negro). Si el estado actual es 0 (i.e., el último píxel introducido era blanco), entonces la probabilidad de que el siguiente estado sea 1 (el siguiente píxel introducido sea negro) se denota α, lo que implica que la probabilidad de que el siguiente estado sea también 0 es $1-\alpha$.

Supongamos que estamos en el estado 0. La probabilidad $p_\omega(1)$ de una secuencia de exactamente un píxel blanco es la probabilidad α de una transición al estado 1. Similarmente, la probabilidad $p_\omega(2)$ de una secuencia de exactamente dos píxeles blancos es la probabilidad de retornar una vez al estado 0, y luego conmutar al estado 1. Puesto que estos eventos son independientes, la probabilidad es el producto $(1-\alpha)\alpha$. En general, la probabilidad $p_\omega(n)$ de un run length de n píxeles blancos es $(1-\alpha)^{n-1}\alpha$ (ésto se puede demostrar por inducción). Por consiguiente, concluimos que $p_\omega(n)$ y, por simetría, también $p_b(n)=(1-\beta)^{n-1}\beta$ se distribuyen geométricamente[15]. Este análisis es conocido como el modelo de Capon Markov-1 de los run lengths de una imagen binivel [Capon 59] y éste justifica (como se explica en la Sección 2.5) el uso de códigos de Golomb para comprimir los run lengths.

Una vez comprendido ésto, es fácil de obtener un proceso sencillo y adaptativo que estime el mejor parámetro m de Golomb para cada run length, calcule el código de Golomb para el run length, y actualice una tabla que se utiliza para obtener mejores estimaciones para el resto de la entrada.

Cada run length es un entero en el intervalo $[1,r]$, donde r es el tamaño de la fila. El primer paso es decrementar el run length en 1, ya que los códigos de Golomb son para enteros no negativos (en oposición a los positivos). Por lo tanto, denotamos el run length modificado de l píxeles de color c por $(c,1)$, donde c es b o ω y l se encuentra en el intervalo $[0,r-1]$. Una tabla $S[b:\omega,\,0:r-1]$ de dos filas indexadas mediante b y ω, y r columnas indexadas desde 0 a $r-1$, se mantiene con recuentos de run lengths pasados y se utiliza para estimar la probabilidad p de cada run length. Asumiendo que la longitud de ejecución actual es (c,l), se calcula entonces p como la entrada $S[c,l]$ dividida por la suma de todas las entradas de S. Luego se utiliza p para calcular el mejor parámetro m de Golomb como el entero más cercano a $-1/\log_2(1-p)$. El run length (c,l) es comprimido calculando el código de Golomb para el entero l con el parámetro m, y se actualiza la tabla S incrementando el elemento $S[c,l]$ en 1. Este proceso puede ser revertido por el decodificador mientras mantenga la misma tabla.

Existe el problema de cómo inicializar S. En principio, todas sus entradas deberían ser inicializadas a cero, pero ésto produciría probabilidades cero, que no pueden ser utilizadas para calcular los valores de m. Este es el problema de la probabilidad cero, introducido en la Sección 2.18. Una solución razonable consiste en inicializar cada entrada de tabla a un valor predeterminado, tal como 1 y permitir al usuario cambiar este valor de acuerdo con la naturaleza de la imagen.

Ésto tiene sentido porque aunque el tipo de imagen más simple es la de dos niveles, existe una amplia variedad de tales imágenes. Una página de texto impreso, por ejemplo, normalmente produce, una vez digitalizada, un 5−6 % de píxeles negros y rachas muy cortas de ésos píxeles (estos valores son utilizados por los fabricantes de impresoras para estimar la vida útil de los cartuchos de impresión). Los valores de α y β para dicha página son aproximadamente 0,02 − 0,03 y 0,34 − 0,35, respectivamente. En el otro extremo, encontramos los grabados tradicionales. Cuando se imprime un grabado y se digitaliza en blanco y negro, a menudo produce una gran mayoría (y con largas rachas) de píxeles negros.

[15]La ω de $p_\omega(n)$ y la b de $p_b(n)$ proceden de *white* (blanco) y *black* (negro), respectivamente.

Un grabado es el arte de tallar en (e imprimir directamente desde) una pieza plana de madera. Las piezas que van a ser impresas permanecen al mismo nivel de la superficie de la madera y las partes restantes deben ser retiradas por el artista. Un rodillo se impregna con tinta y se hace rodar por la madera por lo que cubre la superficie, pero no las partes talladas, con tinta. A continuación se coloca una hoja de papel sobre la madera y se presiona, ya sea a mano o mediante un rodillo, para transferir la tinta de la madera. La impresión resultante sobre el papel es la imagen especular de la residente en la madera, lo que hace que sea especialmente difícil incluir texto en un grabado.

Cuando se imprime en papel, un grabado tiende a tener grandes zonas negras debido a que las áreas blancas (no imprimibles) tienen que ser retiradas de la madera, pero las áreas negras son aquellas partes de la superficie de la madera que no se ven alteradas por el artista.

Es posible hacer grabados en color. El artista comienza con varias piezas planas de madera, una para cada color. En la pieza para el rojo, por ejemplo, las partes rojas de la imagen están intactas y el resto se elimina. La misma hoja de papel es prensada para cada pieza de madera por turnos y se deja secar entre un prensado y otro.

Debido a la rapidez de la vida moderna, muchos artistas en grabados eliminan las áreas no imprimibles de la madera con un chorro de arena en lugar de efectuar la lenta talla a mano (después de cubrir las áreas a imprimir con un metal o un escudo de plástico).

El arte de tallar un grabado en la madera se llama xilografía.

4.23. PPPM

El lector debe revisar el método PPM, Sección 2.18, antes de leer esta sección. El método PPPM utiliza las ideas de MLP (Sección 4.21). También está (remotamente) relacionado con el método de compresión de imágenes basado en el contexto de la Sección 4.18.

PPM codifica un símbolo mediante la comparación de su contexto actual con otros contextos similares y selecciona la coincidencia más larga. El contexto seleccionado se utiliza entonces para estimar la probabilidad del símbolo en el contexto actual. Esta forma de emparejamiento de contexto funciona bien para el texto, donde podemos esperar repeticiones exactas de cadenas de símbolos, pero no funciona bien para las imágenes, debido a que una imagen digital es a menudo el resultado de la digitalización de una imagen analógica. Supongamos que el píxel actual tiene una intensidad 118 y su contexto son los dos píxeles vecinos con valores 118 y 120. Es posible que 118 nunca haya sido visto en el pasado con el contexto 118, 120, pero sí se haya visto con los contextos 119, 120 y 118, 121. Claramente, estos contextos son lo suficientemente cercanos al actual para justificar el uso de uno de ellos. Una vez que se ha encontrado un contexto estrechamente coincidente, éste se utiliza para estimar la varianza (no la probabilidad) de la predicción de error actual. Esta idea sirve como principio del método: predicción mediante emparejamiento de precisión parcial (Prediction by Partial Precision

Matching o PPPM) [Howard y Vitter 92a]. El otro principio es el uso de la distribución de Laplace para estimar la probabilidad de la predicción de error, como se hace en MLP.

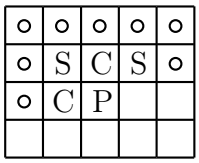


Figura 4.132: Contextos de predicción y estimación de varianza para PPPM.

La Figura 4.132 muestra cómo se efectúa la predicción en PPPM. Los píxeles son escaneados en orden *raster*, fila por fila. Los dos píxeles denotados C se usan para predecir aquél denotado P. La predicción R es simplemente el promedio redondeado de los dos píxeles C. Los píxeles de los bordes superior o izquierdo se predicen mediante un único vecino. El píxel superior izquierdo de la imagen se codifica sin predicción. Después de predecir el valor de P, el codificador calcula el error $E = R - P$ y utiliza la distribución de Laplace para estimar la probabilidad del error, como en MLP.

El único punto a discutir es cómo estima PPPM la varianza de la distribución de Laplace particular, que debe ser utilizada para obtener la probabilidad de E. PPPM utiliza los cuatro vecinos denotados C y S en la Figura 4.132. Estos píxeles ya han sido codificados, de modo que sus valores son conocidos. Se utilizan como el contexto de la estimación de varianza de P. Supongamos que los cuatro valores son 3, 0, 7 y 5, expresados como números de 4 bits. Éstos se combinan para formar la clave de 16 bits 0011|0000|0111|0101, y el codificador utiliza una tabla hash para encontrar todas las apariciones anteriores de este contexto. Si dicho contexto se produjo suficientes veces en el pasado (más que el valor de un parámetro de umbral), las estadísticas de estas ocurrencias se utilizan para obtener una media m y varianza una V. Si el contexto no se ha producido suficientes veces en el pasado, el bit menos significativo de cada uno de los cuatro valores es desechado para obtener la clave de 12 bits 001|000|011|010, y el codificador realiza una operación hash con este valor. Por consiguiente, el codificador itera en un bucle hasta que encuentra m y V. (Resulta que el uso de los errores de los píxeles C y S como una clave, en lugar de sus valores, produce una compresión algo mejor, así que ésto es lo que hace realmente PPPM .)

Una vez que se han obtenido m y V, el codificador cuantifica V y lo usa para seleccionar una de las 37 tablas de probabilidad de Laplace, como en MLP. El codificador entonces suma $E + m$, y envía este valor para ser codificado aritméticamente con la probabilidad obtenida a partir de la tabla de Laplace. Para actualizar las estadísticas, el codificador PPPM emplea un enfoque perezoso. Actualiza las estadísticas del contexto que se usan realmente, y también actualiza, si procede, el contexto, con un bit de precisión adicional.

Un punto crítico es el número de veces que un contexto tuvo que ser visto en el pasado para ser considerado significativo y no aleatorio. El método PPMB, Sección 2.18.4, "confía" en un contexto si se ha visto dos veces. Para una imagen, un umbral de $10-15$ es más razonable.

4.24. CALIC

Las Secciones 4.18 a 4.23 describen los métodos de compresión de imágenes basados en el contexto que tienen una característica en común: Determinan el contexto de un píxel a partir de algunos de sus píxeles vecinos que ya han sido vistos y procesados. Normalmente, éstos son algunos de los píxeles situados por encima y a la izquierda del píxel actual, lo que conduce a un contexto asimétrico. Parece intuitivo que un contexto simétrico, que predice el píxel actual basándose en los píxeles ubicados a su

0,0 0,1	**0,2** 0,3	**0,4** 0,5	**0,6** 0,7
1,0 **1,1**	1,2 **1,3**	1,4 **1,5**	1,6 **1,7**
2,0 2,1	**2,2** 2,3	**2,4** 2,5	**2,6** 2,7
3,0 **3,1**	3,2 **3,3**	3,4 **3,5**	3,6 **3,7**
4,0 4,1	**4,2** 4,3	**4,4** 4,5	**4,6** 4,7
5,0 **5,1**	5,2 **5,3**	5,4 **5,5**	5,6 **5,7**
6,0 6,1	**6,2** 6,3	**6,4** 6,5	**6,6** 6,7
7,0 **7,1**	7,2 **7,3**	7,4 **7,5**	7,6 **7,7**

Tabla 4.133: Los 4×4 valores $\mu[i,j]$ para una imagen de 8×8 píxeles.

alrededor, podría producir una mejor compresión, por lo que se han hecho intentos para desarrollar métodos de compresión de imágenes que emplean tales contextos.

El método MLP, Sección 4.21, ofrece un interesante giro al problema del contexto simétrico. El método CALIC de esta sección tiene un enfoque diferente. El nombre CALIC ([Wu 95] y [Wu 96]) es un acrónimo de Context-based, Adaptive, Lossless Image Compression (compresión de imágenes sin pérdidas, adaptativo y basado en el contexto). Se efectúan tres pasadas para crear un contexto simétrico alrededor del píxel actual, y se usa la cuantificación para reducir el número de posibles contextos a algo manejable. El método ha sido desarrollado para la compresión de imágenes en escala de grises (donde cada píxel es un número de c bits que representa un tono de gris), pero como cualquier otro método para escala de grises, puede manejar una imagen en color, separando los tres componentes de color y tratando cada componente como una imagen en escala de grises.

4.24.1. Tres pasadas

Comenzamos con una imagen $I[i,j]$ que está formada por H filas y W columnas de píxeles. Ambos, codificador y decodificador, realizan tres pasadas sobre la imagen. En la primera, se calculan los promedios de pares de píxeles. Explora sólo los píxeles $I[i,j]$, donde i y j tienen la misma paridad (i.e., ambos son pares o ambos son impares). En la segunda pasada utiliza estos promedios para codificar realmente los mismos píxeles. En la tercera pasada usa los mismos promedios más los píxeles de la segunda pasada para codificar todos los píxeles $I[i,j]$, donde i y j tengan paridades diferentes (uno es impar y el otro es par).

La primera pasada calcula los ${}^{W}/_{2} \times {}^{H}/_{2}$ valores $\mu[i,j]$ definidos por:

$$\mu[i,j] = \frac{I[2i,2j] + I[2i+1,2j+1]}{2}, \text{ para } 0 \le i < {}^{H}/_{2},\ 0 \le j < {}^{W}/_{2}. \tag{4.47}$$

(En los documentos originales sobre CALIC, i y j denotan columnas y filas, respectivamente. Utilizamos la notación estándar donde el primer índice denota las filas.) Cada $\mu[i,j]$ es, por lo tanto, el promedio de dos píxeles adyacentes diagonalmente. La Tabla 4.133 muestra los píxeles (en negrita) involucrados en este cálculo para una imagen de 8×8 píxeles. Cada par que se usa para calcular un valor $\mu[i,j]$ está conectado con una flecha. Observe que los dos píxeles originales no pueden ser reconstruidos completamente a partir de la media debido a que se puede perder un bit en la división por 2 de la Ecuación (4.47).

Los valores recién calculados, $\mu[i,j]$, se consideran ahora los píxeles de una nueva y pequeña imagen de ${}^{W}/_{2} \times {}^{H}/_{2}$ píxeles (un cuarto del tamaño de la imagen original). Esta imagen es escaneada en orden *raster*, y cada uno de sus píxeles es predicho con cuatro de sus vecinos, tres centrados encima de él y

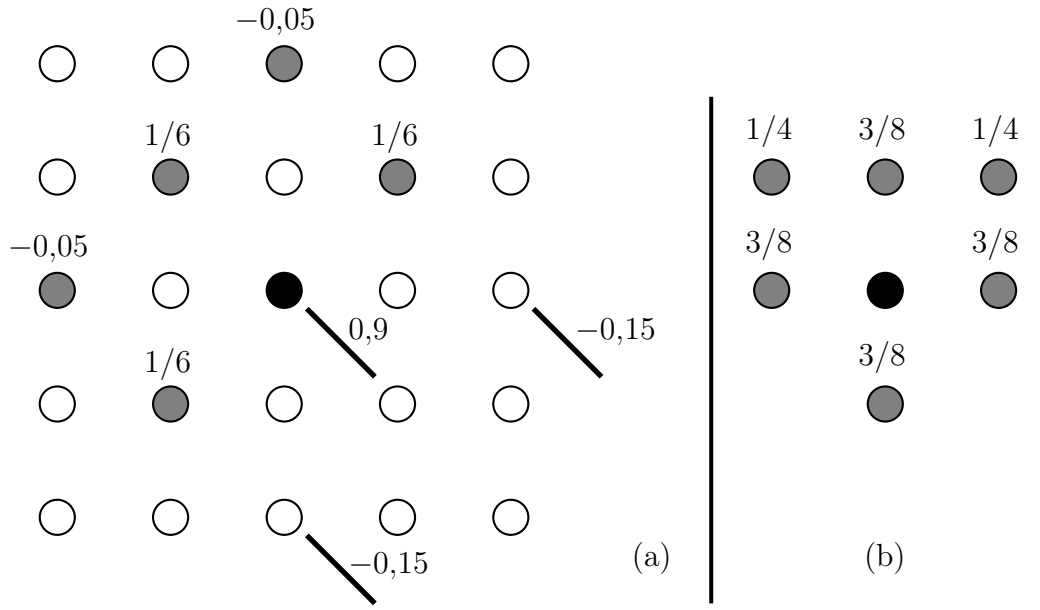


Figura 4.134: Sumas ponderadas para contextos de 360°.

uno a su izquierda. Si $x = \mu[i,j]$ es el píxel actual, se prevé mediante la cantidad:

$$\hat{x} = \frac{1}{2}\mu[i,j-1] - \frac{1}{4}\mu[i-1,j-1] + \frac{1}{2}\mu[i-1,j] + \frac{1}{4}\mu[i-1,j+1]. \tag{4.48}$$

(Los coeficientes $1/2$, $1/4$, y $-1/4$, así como los coeficientes utilizados en las otras pasadas, se han determinado mediante una regresión lineal, utilizando un conjunto de "entrenamiento" de imágenes. La idea es encontrar el conjunto de coeficientes a_k que proporcione la mejor compresión para ésas imágenes, luego redondearlos a potencias enteras de 2, y construir con ellos el algoritmo.) Después se codifica el valor del error $x - \hat{x}$.

La segunda pasada involucra a los mismos píxeles de la primera pasada (la mitad de los píxeles de la imagen original), pero esta vez, cada uno de ellos se prevé individualmente. Son escaneados en orden *raster*, y asumiendo que $x = [2i, 2j]$ denota el píxel actual, éste se predice utilizando cinco píxeles vecinos conocidos por encima de él y a su izquierda, y tres promedios μ, conocidos de la primera pasada, por debajo de él y a su derecha:

$$\begin{aligned}\hat{x} =& 0{,}9\mu[i,j] + \frac{1}{6}\left(I[2i+1,2j-1] + I[2i-1,2j-1] + I[2i-1,2j+1]\right)\\ &- 0{,}05\left(I[2i,2j-2] + I[2i-2,2j]\right) - 0{,}15\left(\mu[i,j+1] + \mu[i+1,j]\right).\end{aligned} \tag{4.49}$$

La Figura 4.134a muestra los cinco píxeles (puntos grises) y tres medias (líneas inclinadas) involucrados. La tarea del codificador es codificar de nuevo el valor del error $x - \hat{x}$ para cada píxel x.

◇ **Ejercicio 4.46 (sol. en pág. 1088):** El píxel $I[2i-1, 2j+1]$ se encuentra por debajo de $x = I[2i, 2j]$, así que ¿cómo hace el decodificador para conocer su valor cuando x es decodificado?

La tercera pasada involucra a los píxeles restantes:

$$I[2i, 2j+1] \text{ e } I[2i+1, 2j], \quad \text{para } 0 \le i < H/2,\ 0 \le j < W/2. \tag{4.50}$$

Cada uno se predice con ayuda de un contexto casi simétrico de seis píxeles (Figura 4.134b) consistente en la totalidad de sus vecinos de cuatro conexiones y dos de sus vecinos de ocho conexiones. Si $x = I[i, j]$ es el píxel actual, éste se prevé mediante:

$$\begin{aligned}\hat{x} =& \frac{3}{8}\left(I[i, j-1] + I[i-1, j] + I[i, j+1] + I[i+1, j]\right) \\ & - \frac{1}{4}\left(I[i-1, j-1] + I[i-1, j+1]\right).\end{aligned} \tag{4.51}$$

El decodificador puede imitar esta operación, ya que los píxeles por debajo y a la derecha de x son conocidos por él desde la segunda pasada.

Observe que cada una de las dos últimas pasadas del decodificador crea la mitad de los píxeles de la imagen. Por consiguiente, CALIC puede considerarse un método progresivo, donde los dos pasos progresivos incrementan la resolución de la imagen.

4.24.2. Cuantificación del contexto

En cada una de las tres pasadas, los valores de error $x - \hat{x}$ están codificados aritméticamente, lo que significa que tienen que tener asignadas probabilidades. Supongamos que el píxel x es predicho por los n píxeles vecinos $x_1, x_2, \ldots, x_n$. Los valores de n para las tres pasadas son: 4, 8, y 6, respectivamente. Con el fin de asignar una probabilidad a x, el codificador tiene que contar el número de veces que encuentra x en el pasado con cada posible contexto de n píxeles. Si un píxel se almacena como un número de bits c (que representa un tono de gris), entonces el número de contextos posibles para un píxel es $2^{n \cdot c}$. Incluso para el caso $c = 4$ (exactamente 16 tonos de gris), este número es $2^{8 \cdot 4} \approx 4{,}3$ mil millones, demasiado grande para una aplicación práctica. CALIC reduce el número de contextos en varios pasos. En primer lugar, crea el único número de n bits $t = t_n \ldots t_1$ mediante:

$$t_k = \begin{cases} 0, & \text{si } x_k \geq \hat{x}, \\ 1, & \text{si } x_k < \hat{x}. \end{cases}$$

A continuación, calcula la cantidad:

$$\Delta = \sum_{k=1}^{n} \omega_k \left| x_k - \hat{x} \right|,$$

donde los coeficientes ω_k fueron preparados con antelación, utilizando el mismo conjunto de imágenes de entrenamiento, y están integrados en el algoritmo. La cantidad Δ se llama *discriminante de resistencia del error* y se cuantifica a uno de los L valores d enteros, donde L se establece típicamente a 8. Una vez que se conocen $\hat{x}$ y los n vecinos $x_1, x_2, \ldots, x_n$, tanto t como el valor cuantificado d de Δ puede ser determinado, y ellos se convierten en los índices del contexto. Ésto reduce el número de contextos a $L \cdot 2^n$, que es, como mucho, $8 \cdot 2^8 = 2048$. El codificador mantiene un arreglo S de d filas y t columnas, donde se conservan los recuentos del contexto. Lo siguiente es un resumen de los pasos efectuados por el codificador CALIC.[16]

[16] Mantengo las palabras *for*, *do*, *end*, *endfor*, *if*, *then*, y *else*, comunes en muchos lenguajes de programación para realizar bucles y comprobaciones.

```
Para todas las pasadas
INICIALIZACIÓN: N(d,t):=1; S(d,t):=0; d=0,1,...,L, t=0,1,...,2^n;
PARÁMETROS: a_k y ω_k son asignados a sus valores;
for todos los píxeles x en la pasada actual do
  0: x̂ = Σ_{k=1}^{n} a_k · x_k;
  1: Δ = Σ_{k=1}^{n} ω_k (x_k − x̂);
  2: d = Cuantificar(Δ);
  3: Calcular t = t_n ... t_2 t_1;
  4: ε̄ = S(d,t) + ε;  N(d,t) := N(d,t) + 1;
  5: ẋ = x̂ + ε̄;
  6: ε = x − ẋ;
  7: S(d,t) := S(d,t) + ε;  N(d,t) := N(d,t) + 1;
  8: if N(d,t) ≥ 128 then
        S(d,t) := S(d,t)/2;  N(d,t) := N(d,t)/2;
  9: if S(d,t) < 0 codificar(−ε, d) else codificar(ε, d);
endfor;
fin.
```

4.25. Compresión diferencial sin pérdida

Siempre hay un equilibrio entre velocidad y rendimiento, por lo que siempre hay una demanda de métodos de compresión rápidos, así como métodos que son lentos pero muy eficaces. El método diferencial de esta sección, debido a Sayood y Anderson [Sayood y Anderson 92], pertenece a la primera clase. Es rápido y fácil de implementar, al tiempo que ofrece un buen, aunque no espectacular, rendimiento.

El principio consiste en comparar cada píxel p a un *píxel de referencia*, que es uno de sus vecinos inmediatos previamente codificados, y codificar p en dos partes: un prefijo, que es el número de bits más significativos de p que son idénticos a aquellos del píxel de referencia; y un sufijo, formado por los restantes bits menos significativos de p. Por ejemplo, si el píxel de referencia es 10110010 y p es 10110100, entonces el prefijo es 5, porque los cinco bits más significativos de p son idénticos a los del píxel de referencia, y el sufijo es 00. Observe que los tres bits menos significativos restantes son 100, pero el sufijo no tiene que incluir el 1, ya que el descodificador puede deducir fácilmente su valor.

◇ **Ejercicio 4.47 (sol. en pág. 1088):** ¿Cómo puede el decodificador de hacer ésto?

El prefijo en nuestro ejemplo es 5, y en general es un número entero en el rango $[0, 8]$, y la compresión se puede mejorar codificando además el prefijo. La codificación de Huffman es una buena elección para este propósito, ya sea con un conjunto fijo de nueve códigos de Huffman, ya sea con códigos adaptativos. El sufijo puede ser cualquier número de entre cero y ocho bits, por lo que hay 256 sufijos posibles. Dado que este número es relativamente grande, y puesto que esperamos que la mayoría de los sufijos sean pequeños, tiene sentido escribir el sufijo en la secuencia de salida sin codificar.

Este método codifica cada píxel con un número distinto de bits. El codificador genera bits hasta que tenga 8 ó más de ellos, después emite un byte. El decodificador puede imitar ésto fácilmente. Todo lo que tiene que saber es la ubicación del píxel de referencia y los códigos de Huffman. En el ejemplo anterior, si el código de Huffman de 6 es, digamos, 010, el código de p serán los cinco bits: 010|00.

El único punto que queda por discutir es la selección del píxel de referencia. Éste debe estar cercano al píxel actual p, y debe ser conocido por el decodificador cuando p es decodificado. Las normas adoptadas por los desarrolladores de este método para seleccionar el píxel de referencia son, por tanto, sencillas. El primer píxel de una imagen se escribe en la secuencia de salida sin codificar. Para cada píxel en la primera línea (superior) de escaneo, el píxel de referencia se selecciona como su

vecino izquierdo inmediato. Para el primer píxel (del extremo izquierdo) en las subsiguientes líneas escaneadas, el píxel de referencia es uno por encima de él. Para cada cualquier otro píxel, es posible seleccionar el píxel de referencia de una de tres maneras: (1) el píxel inmediatamente a su izquierda; (2) el píxel por encima de él; y (3) el píxel de la izquierda, excepto si el prefijo resultante es menor que un umbral predeterminado, en cuyo case se selecciona el píxel por encima de él.

Un ejemplo del caso 3 es un valor umbral de 3. Inicialmente, el píxel de referencia elegido para p es su vecino izquierdo, pero si ésto resulta en un valor de prefijo 0, 1, ó 2, el píxel de referencia se cambia a uno encima de p, independientemente del valor de prefijo que es entonces producido.

Este método asume un byte por píxel (256 colores o valores en escala de grises). Si un píxel se expresa mediante tres bytes, la imagen debe ser separada en tres partes, y el método se aplica a cada parte por separado.

◇ **Ejercicio 4.48 (sol. en pág. 1088):** ¿Puede utilizarse este método para imágenes con 16 valores de escala de grises (donde cada píxel es de cuatro bits, y un byte contiene dos píxeles)?

> "El impulso nervioso, incluyendo las ondas escépticas, tendrá que saltar el pequeño espacio de la sinapsis y, al hacerlo, los pensamientos dominantes serán menos atenuados que los otros. En resumen, si además saltamos la sinapsis, vamos a llegar a una región en la que podemos, por un tiempo al menos, detectar lo que queremos oír con menos interferencia de ruido trivial."
> "¿De veras?", Preguntó con malicia Morrison. "Esta noción de la atenuación diferencial es nueva para mí."
>
> —Isaac Asimov, *Viaje fantástico II: Destino, el cerebro*

4.26. DPCM

El método de compresión DPCM es un miembro de la familia de los métodos de compresión por codificación diferencial, que en sí mismos son una generalización del sencillo concepto de codificación relativa (Sección 1.3.1). Se basa en el hecho bien conocido de que los píxeles vecinos en una imagen (y también las muestras adyacentes en el sonido digitalizado, Sección 7.2) están correlacionados. Los valores correlacionados son generalmente similares, por lo que sus diferencias son pequeñas, dando lugar a la compresión. La Tabla 4.135 muestra 25 valores consecutivos del $\sin\theta_i$, calculados para valores de θ_i a 360° en pasos de 15°. Por consiguiente, toma valores desde -1 a $+1$; pero las 24 diferencias $\sin\theta_{i+1} - \sin\theta_i$ (también aparecen en la tabla) están en el rango $[-0{,}259, 0{,}259]$. El promedio de los 25 valores es cero, así como el promedio de las 24 diferencias. Sin embargo, la varianza de las diferencias es pequeña, ya que todos ellos están próximos a su promedio.

La Figura 4.136a muestra un histograma de una imagen hipotética formada por píxeles de 8 bits. Para cada valor de píxel entre 0 y 255 hay un número diferente de píxeles. La Figura 4.136b muestra un histograma de las diferencias de píxeles consecutivos. Es fácil ver que la mayor parte de las diferencias (que, en principio, pueden pertenecer al rango $[0, 255]$) son pequeñas; y sólo unas pocas están fuera del rango $[-50, +50]$.

Los métodos de codificación diferencial calculan las diferencias $d_i = a_i - a_{i-1}$ entre ítems de datos a_i consecutivos, y codifican los d_i's. El primer ítem de datos, a_0, o bien se codifica por separado, o bien se escribe en el *stream* comprimido en formato *raw* (puro, sin codificar). En cualquier caso, el decodificador puede decodificar y generar a_0 exactamente. En principio, cualquier método adecuado, con o sin pérdidas, puede ser utilizado para codificar las diferencias. En la práctica, la cuantificación se utiliza a menudo, dando como resultado una compresión con pérdida. La cantidad codificada no es la diferencia d_i sino un número similar cuantificado que denotamos por $\hat{d}_i$. La diferencia entre d_i y $\hat{d}_i$ es el *error de cuantificación* q_i. Por consiguiente, $\hat{d}_i = d_i + q_i$.

$\sin(t)$:	0	0,259	0,500	0,707	0,866	0,966	1,000	0,966	
diff :	–	0,259	0,241	0,207	0,207	0,100	0,034	−0,034	
$\sin(t)$:	0,866	0,707	0,500	0,259	0	−0,259	−0,500	−0,707	
diff :	−0,100	−0,159	−0,207	−0,241	−0,259	−0,259	−0,241	−0,207	
$\sin(t)$:	−0,866	−0,966	−1,000	−0,966	−0,866	−0,707	−0,500	−0,259	0
diff :	−0,159	−0,100	−0,034	0,034	0,100	0,159	0,207	0,241	0,259

Tabla 4.135: 25 valores del seno y 24 diferencias.

```
S=Table[N[Sin[t Degree]], {t,0,360,15}]
Table[S[[i+1]]-S[[i]], {i,1,24}]
```

Código para la Tabla 4.135.

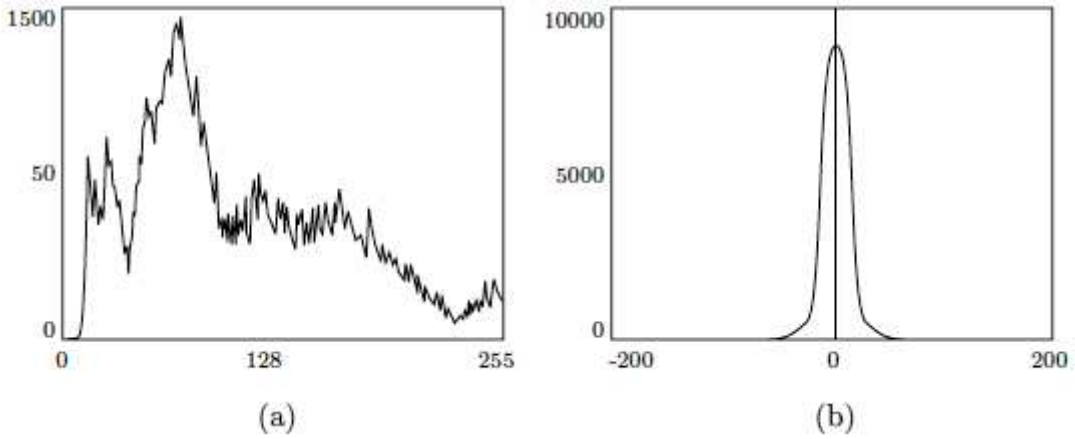


Figura 4.136: Un histograma de una imagen y sus diferencias.

Resulta que la compresión con pérdida de las diferencias introduce un nuevo problema, a saber, la acumulación de errores. Ésto es fácil de ver cuando consideramos el funcionamiento del decodificador. El decodificador introduce valores codificados $\hat{d}_i$, los decodifica, y los utiliza para generar valores "reconstruidos" $\hat{a}_i$ (donde $\hat{a}_i = \hat{a}_{i-1} + \hat{d}_i$) en lugar de los valores de los datos originales a_i. El decodificador comienza leyendo y decodificando a_0. Después, introduce $\hat{d}_1 = d_1 + q_1$ y calcula $\hat{a}_1 = a_0 + \hat{d}_1 = a_0 + d_1 + q_1 = a_1 + q_1$. El siguiente paso es la introducción de $\hat{d}_2 = d_2 + q_2$ y el cálculo de $\hat{a}_2 = \hat{a}_1 + \hat{d}_2 = a_1 + q_1 + d_2 + q_2 = a_2 + q_1 + q_2$. El valor decodificado $\hat{a}_2$ contiene la suma de dos errores de cuantificación. En general, el valor decodificado $\hat{a}_n$ es igual a:

$$\hat{a}_n = a_n + \sum_{i=1}^{n} q_i,$$

e incluye la suma de n errores de cuantificación. A veces, los errores q_i son con signo y tienden a anularse entre sí a largo plazo. En general, sin embargo, ésto es un problema.

La solución es fácil de comprender una vez nos damos cuenta de que el codificador y el decodificador operan sobre muestras de datos diferentes. El codificador genera las diferencias exactas d_i a partir de los ítems de datos originales a_i, mientras que el decodificador genera los $\hat{a}_i$ reconstruidos utilizando sólo las diferencias cuantificadas $\hat{d}_i$. La solución es, por lo tanto, modificar el codificador para calcular

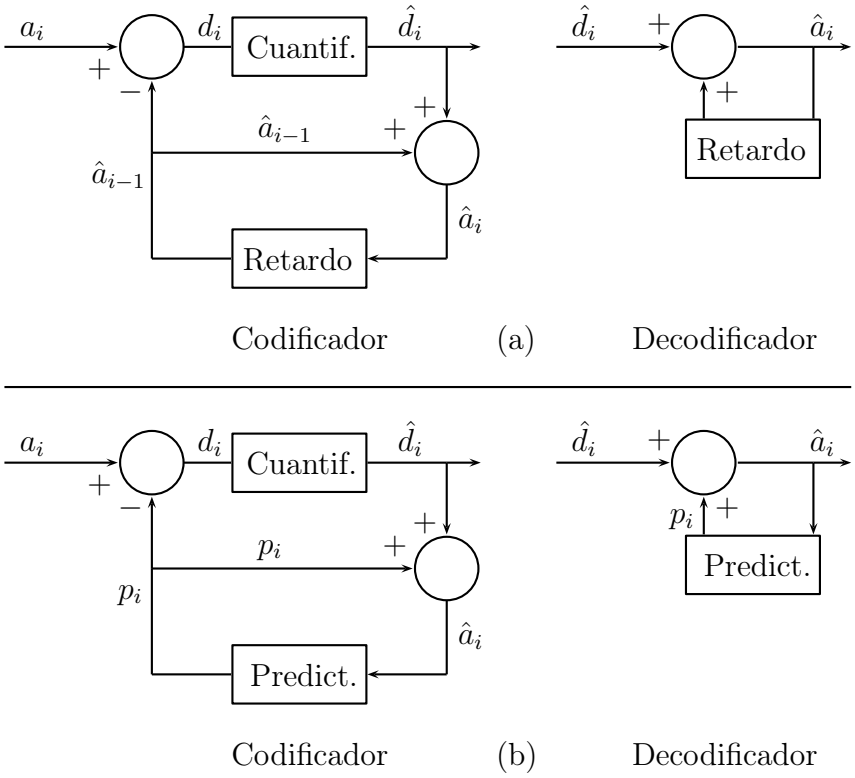


Figura 4.137: (a) Un codificador diferencial. (b) DPCM.

las diferencias de la forma $d_i = a_i - \hat{a}_{i-1}$. Por tanto, para calcular una diferencia general d_i, se resta el valor reconstruido más reciente $\hat{a}_{i-1}$ (que ambos, codificador y decodificador, ya tienen) del ítem original actual a_i.

Ahora, el decodificador empieza a leer y decodificar a_0. A continuación, introduce $\hat{d}_1 = d_1 + q_1$ y calcula $\hat{a}_1 = a_0 + \hat{d}_1 = a_0 + d_1 + q_1 = a_1 + q_1$. El siguiente paso, es la introducción de $\hat{d}_2 = d_2 + q_2$ y el cálculo de $\hat{a}_2 = \hat{a}_1 + \hat{d}_2 = \hat{a}_1 + d_2 + q_2 = a_2 + q_2$. El valor decodificado $\hat{a}_2$ contiene sólo el error de cuantificación q_2 y, en general, el valor decodificado de $\hat{a}_i$ es igual a $a_i + q_i$, por lo que únicamente contiene el error de cuantificación q_i. Decimos que el *ruido de cuantificación* en la decodificación de $\hat{a}_i$ es igual al ruido generado cuando a_i fue cuantificado.

La Figura 4.137a resume las operaciones tanto de codificador como del decodificador. Muestra cómo el ítem de datos actual a_i es guardado en una unidad de almacenamiento (un retardo), que se utiliza para codificar el siguiente ítem a_{i+1}.

El siguiente paso en el desarrollo de un método de codificación diferencial general es aprovechar el hecho de que los ítems de datos que están siendo comprimidos están correlacionados. Esto significa que, en general, un ítem a_i depende de *varios* de sus vecinos más próximos, no sólo del ítem precedente a_{i-1}. Por lo tanto, puede obtenerse una mejor predicción (y, como resultado, diferencias más pequeñas) utilizando N de los vecinos previamente vistos para codificar ítem actual a_i (donde N es un parámetro). Por consiguiente, quisiéramos tener una función $p_i = f(\hat{a}_{i-1}, \hat{a}_{i-2}, \ldots, \hat{a}_{i-N})$ para predecir a_i (Figura 4.137b). Observe que f tiene que ser una función de $\hat{a}_{i-j}$, no de a_{i-j}, ya que el decodificador tiene que calcular la misma f. Cualquier método que utilice un predictor se llama *modulación por codificación de pulsos diferencial*, o DPCM. En la práctica, los métodos de DPCM se utilizan sobre todo para la compresión de audio, pero aquí se muestran en el dominio de la compresión de imágenes.

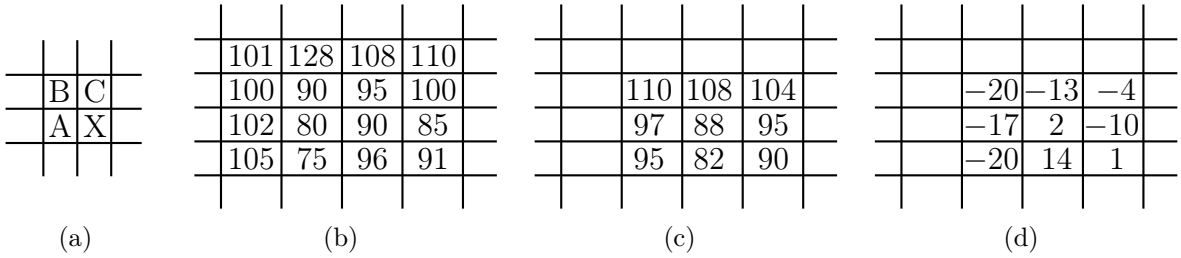


Figura 4.138: Predicción de píxeles y cálculo de diferencias.

La forma más simple de predictor es lineal. En dicho predictor el valor del píxel actual a_i es predicho mediante una suma ponderada de N de sus vecinos vistos previamente (en el caso de una imagen ésos son los píxeles ubicados por encima de él o a su izquierda):

$$p_i = \sum_{j=1}^{N} \omega_j a_{i-j},$$

donde los ω_j son los pesos, que aún necesitan ser determinados.

La Figura 4.138a muestra un sencillo ejemplo para el caso $N = 3$. Vamos a asumir que un píxel X se predice mediante tres de sus vecinos A, B y C de acuerdo con la sencilla suma ponderada:

$$X = 0{,}35A + 0{,}3B + 0{,}35C. \tag{4.52}$$

La Figura 4.138b muestra 16 píxeles de 8 bits, parte de una imagen más grande. Utilizamos la Ecuación (4.52) para predecir los nueve píxeles en la parte inferior derecha. Las predicciones se muestran en la Figura 4.138c. La Figura 4.138d muestra las diferencias entre los valores del píxel a_i y sus predicciones p_i.

Las ponderaciones utilizadas en la Ecuación (4.52) se han seleccionado de forma más o menos arbitraria y son sólo para fines ilustrativos. Sin embargo, tienen sentido, porque suman la unidad.

◇ **Ejercicio 4.49 (sol. en pág. 1088):** ¿Por qué la suma de los pesos es uno? (Éste es un ejercicio fácil, pero importante, porque los pesos que suman la unidad son muy comunes. Tales pesos reciben el nombre de baricéntricos.)

Con el fin de determinar los mejores pesos, denotamos e_i al error de predicción para el píxel a_i:

$$e_i = a_i - p_i = a_i - \sum_{j=1}^{N} \omega_j a_{i-j}, \quad i = 1, 2, \ldots, n,$$

donde n es el número de píxeles a ser comprimido (en nuestro ejemplo, nueve de los 16 píxeles), y hallamos el conjunto de pesos ω_j que minimiza la suma:

$$E = \sum_{i=1}^{n} e^2 = \sum_{i=1}^{n} \left[a_i - \sum_{j=1}^{N} \omega_j a_{i-j} \right]^2.$$

Denotamos $\mathbf{a} = (90, 95, 100, 80, 90, 85, 75, 96, 91)$ al vector de los nueve píxeles a ser comprimidos. Denotamos $\mathbf{b}^{(k)}$ al vector formado por los k-ésimos vecinos de los seis píxeles. Por consiguiente:

$$\mathbf{b}^{(1)} = (100, 90, 95, 102, 80, 90, 105, 75, 96),$$
$$\mathbf{b}^{(2)} = (101, 128, 108, 100, 90, 95, 102, 80, 90),$$
$$\mathbf{b}^{(3)} = (128, 108, 110, 90, 95, 100, 80, 90, 85).$$

```
a={90.,95,100,80,90,85,75,96,91};
b1={100,90,95,102,80,90,105,75,96};
b2={101,128,108,100,90,95,102,80,90};
b3={128,108,110,90,95,100,80,90,85};
Solve[{b1.(a-w1 b1-w2 b2-w3 b3)==0,
b2.(a-w1 b1-w2 b2-w3 b3)==0,
b3.(a-w1 b1-w2 b2-w3 b3)==0},{w1,w2,w3}]
```

Figura 4.139: Resolución para tres pesos.

El error de predicción cuadrático total es:

$$E = \left| a - \sum_{j=1}^{3} \omega_j b^{(j)} \right|^2 ,$$

donde las barras verticales indican el valor absoluto del vector ubicado entre ellos. Para minimizar este error, necesitamos encontrar la combinación lineal de los vectores $\mathbf{b}^{(j)}$ que más se aproxime a $\mathbf{a}$. Los lectores familiarizados con el concepto algebraico de espacios vectoriales saben que ésto se consigue encontrando la proyección ortogonal de $\mathbf{a}$ en el espacio vectorial generado por los $\mathbf{b}^{(j)}$'s, o, equivalentemente, calculando el vector diferencia:

$$a - \sum_{j=1}^{3} \omega_j \mathbf{b}^{(j)}$$

que es ortogonal a todos los $\mathbf{b}^{(j)}$'s. Dos vectores son ortogonales si su producto escalar es cero, lo que produce las M (en nuestro caso, 3) ecuaciones:

$$\mathbf{b}^{(k)} \cdot \left(\mathbf{a} - \sum_{j=1}^{M} \omega_j \mathbf{b}^{(j)} \right) = 0, \quad \text{para } 1 \leq k \leq M,$$

o, equivalentemente:

$$\sum_{j=1}^{M} \omega_j \left(\mathbf{b}^{(k)} \cdot \mathbf{b}^{(j)} \right) = \left(\mathbf{b}^{(k)} \cdot \mathbf{a} \right), \quad \text{para } 1 \leq k \leq M.$$

El código en *Mathematica* de la Figura 4.139 produce, para nuestro ejemplo, las soluciones: $\omega_1 = 0{,}1691$, $\omega_2 = 0{,}1988$, y $\omega_3 = 0{,}5382$. Observe que suman 0,9061, no 1, y ésto se discute en el siguiente ejercicio.

◇ **Ejercicio 4.50 (sol. en pág. 1088):** Repítase este cálculo para los seis píxeles: 90, 95, 100, 80, 90, y 85. Coméntense sus resultados.

DPCM adaptativo: Esta variante de DPCM se utiliza comúnmente para la compresión de audio. En el ADPCM el tamaño del paso de cuantificación se adapta al cambio de la frecuencia del sonido que se está comprimiendo. El predictor también tiene que adaptarse y recalcular los pesos de acuerdo con los cambios en la entrada. Existen varias versiones de ADPCM. Una versión popular es el estándar IMA ADPCM (Sección 7.6), que especifica la compresión de PCM, desde 16 hasta cuatro bits por muestra. ADPCM es rápido, pero introduce un ruido de cuantificación notable y alcanza los factores de compresión más mediocres de los cuatro.

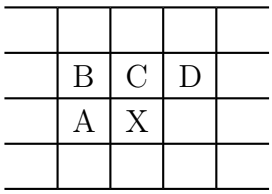


Figura 4.140: Predicción de píxel.

4.27. Ponderación del árbol de contexto

El método de ponderación del árbol de contexto de la Sección 2.19 se puede aplicar imágenes. El método aquí descrito [Ekstrand 96] se desarrolla en cinco pasos de la siguiente manera:

- *Paso 1: predicción.* Este paso utiliza diferencias para predecir los píxeles y es similar al modo sin pérdidas de JPEG (Sección 4.8.5). Se utilizan los cuatro vecinos inmediatos A, B, C, y D del píxel actual X (Figura 4.140) para calcular una predicción lineal P de X de acuerdo con:

  $$L = aA + bB + cC + dD, \quad P = X - [L + 1/2], \quad \text{para } a + b + c + d = 1.$$

  Los cuatro pesos deben sumar 1 y se seleccionan de tal manera que a y c tienen asignados valores ligeramente mayores que b y d. Una posible elección es $a = c = 0{,}3$ y $b = d = 0{,}2$, pero la experiencia parece sugerir que la precisión de los valores no es crítica.

  Los valores de P son, en su mayoría, cercanos a cero y siguen, a menudo, la distribución de Laplace (Figura 4.128).

- *Paso 2: codificación de Gray.* Nuestra intención es aplicar el método CTW a la compresión de imágenes en escala de grises o en color. En el caso de una imagen en escala de grises tenemos que separar los planos de bits y comprimir cada uno individualmente, como si se tratara de una imagen binivel. Una imagen en color tiene que ser dividida en sus tres componentes, y cada componente debe ser comprimido por separado, como en una imagen en escala de grises. La Sección 4.2.1 discute los *códigos de Gray reflejados* (*reflected Gray codes* o RGC) y muestra cómo su uso preserva las correlaciones entre píxeles después de que los distintos planos de bits estén separados.

- *Paso 3: serialización.* La imagen se convierte en una cadena de bits que será codificada aritméticamente en el paso 5. Tal vez la mejor forma de hacer ésto es escanear la imagen fila por fila para cada plano de bits. La cadena de bits, por lo tanto, comienza con todos los bits del primer (menos significativo) plano de bits, continúa con los bits del segundo plano de bits menos significativo, y así sucesivamente. Cada bit de la cadena de bits resultante tiene como contexto (sus vecinos de la izquierda) los bits que originalmente estaban por encima de él o a su izquierda en el plano de bits bidimensional. Una alternativa es comenzar con las primeras filas de todos los planos de bits, continuar con la segunda fila, y así sucesivamente.

- *Paso 4: la estimación.* La cadena de bits que se lee bit a bit, y se construye y actualiza un árbol de contexto. La probabilidad ponderada en la raíz del árbol se utiliza para predecir el bit actual, y tanto la probabilidad como el bit se envían al codificador aritmético (Paso 5). El árbol debe ser profundo, ya que la predicción efectuada en el paso 1 reduce la correlación entre los píxeles. Por otro lado, un árbol CTW profundo ralentiza el codificador, ya que deben ser actualizados más nodos para cada nuevo bit introducido.

- *Paso 5: codificación.* Esto se realiza mediante un codificador aritmético adaptativo estándar.

4.28. Descomposición de bloques

Los lectores de este capítulo pueden haber notado que la mayoría de los métodos de compresión de imágenes han sido diseñados para, y funcionar mejor, con imágenes de tonos continuos, donde los píxeles adyacentes normalmente tienen intensidades o colores similares. El método aquí descrito está destinado a la compresión sin pérdidas de imágenes de tonos discretos, ya sean binivel, en escala de grises o de color. Estas imágenes son (con pocas excepciones) artificiales, siendo obtenidas mediante el escaneo de un documento, o la captura de una pantalla de ordenador. En tal imagen, los colores de los píxeles no varían de forma continua o suavemente, pero posee un pequeño conjunto de valores, tales que los píxeles adyacentes pueden diferir mucho en intensidad o color.

El método trabaja buscando y localizando bloques idénticos de píxeles. Una copia B de un bloque A se comprime preparando la altura, anchura, y ubicación (coordenadas de imagen) de A, y comprimiendo esos cuatro números mediante códigos de Huffman. El método se llama *descomposición de bloques automática flexible* (*Flexible Automatic Block Decomposition* o FABD) [Gilbert y Brodersen 98]. La localización de bloques idénticos de píxeles es un método natural para la compresión de imágenes, porque una imagen es una estructura bidimensional. El formato de archivo de gráficos GIF (Sección 3.19), por ejemplo, escanea la imagen fila por fila para comprimirla. Por consiguiente, es un método unidimensional, por lo que su eficacia como un compresor de imágenes no es muy alta. El método JBIG de la Sección 4.11 considera los píxeles de forma individual, y examina sólo los vecinos próximos de un píxel. No intenta descubrir correlaciones en partes distantes de la imagen (al menos, no explícitamente).

FABD, por otro lado, asume que las partes idénticas (bloques) de píxeles pueden aparecer varias veces en la imagen. En otras palabras, supone que las imágenes tienen una *redundancia bidimensional global*. También supone existen bloques de píxeles grandes y uniformes en la imagen. Por lo tanto, FABD se desenvuelve bien con imágenes que satisfacen estos supuestos, tales como imágenes de tonos discretos. El método analiza la imagen en orden *raster*, fila por fila, y la divide en conjuntos de bloques (posiblemente solapados). Hay tres tipos de bloques, a saber, bloques de copia, bloques de relleno denso, y *punts*.

> Básicamente, "punting" se utiliza a menudo como jerga (al menos en Massachusetts, de donde procedo) en el sentido de "renunciar" o hacer algo subóptimo —en mi caso *punting* es una especie de cajón desastre para asegurar que los píxeles que no pueden participar eficientemente en un bloque de relleno o de copia sean, a pesar de todo, codificados—.
>
> —Jeffrey M. Gilbert

Un bloque de copia B es una porción rectangular de la imagen que ha sido vista antes (ubicada por encima, o en la misma línea —pero a la izquierda— del píxel actual). Puede tener cualquier tamaño. Un bloque de relleno denso es una región uniforme rectangular de la imagen. Un *punt* es una zona de la imagen que no es ni un bloque de copia, ni uno de relleno denso. Cada uno de estos tres tipos se comprime preparando de un conjunto de parámetros que describen completamente el bloque, y escribiendo sus códigos de Huffman en el *stream* comprimido. Aquí se ofrece una descripción general de las operaciones del codificador.

El codificador procede píxel a píxel. Mira en las proximidades del píxel actual P para ver si sus futuros vecinos (aquellos a la derecha y por debajo de P) tienen el mismo color. En caso afirmativo, el método localiza el mayor bloque uniforme del cual P es la esquina superior izquierda. Una vez que dicho bloque ha sido identificado, el codificador conoce la anchura y la altura del bloque. Escribe los códigos de Huffman de estas dos cantidades en el *stream* comprimido, seguido por el color del bloque, y precedido por un código que especifica un bloque de relleno denso. Los cuatro valores:

código del bloque de relleno, anchura, altura, valor del píxel,

se envían, codificados, a la salida. La Figura 4.141a muestra un bloque de relleno con dimensiones 4×3 para el píxel B.

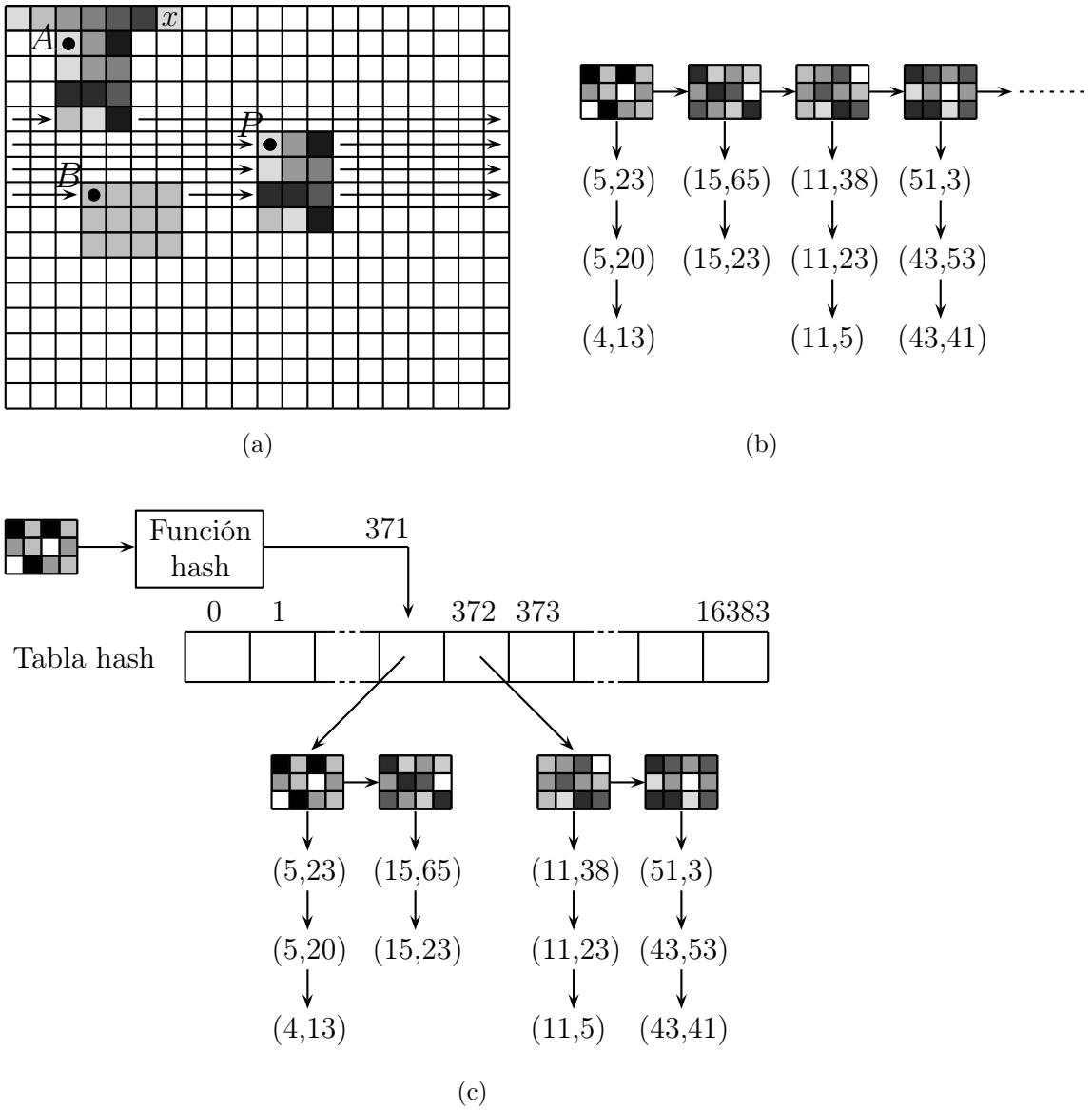


Figura 4.141: Codificación FABD de una imagen.

Si los vecinos más próximos de P tienen valores diferentes, el codificador comienza a buscar un bloque idéntico entre los píxeles del pasado. Considera P, la esquina superior izquierda de un bloque de dimensiones no especificadas, y busca los píxeles vistos en el pasado (aquellos ubicados arriba o a la izquierda de P) para encontrar el mayor bloque A, que coincida con P. Si encuentra un bloque, entonces P se designa bloque copia de A. Puesto que A ya ha sido comprimido, su bloque copia P puede ser plenamente identificado, preparando sus dimensiones (anchura y altura) y la ubicación de

origen (las coordenadas de A). Las cinco cantidades:

$$\text{código del bloque copia, anchura, altura}, A_x,\ A_y,$$

se envían, adecuadamente codificadas, a la salida. Observe que no hay necesidad de escribir las coordenadas de P en la salida, porque ambos, codificador y decodificador, proceden píxel por píxel, en orden *raster*. La Figura 4.141a muestra un bloque copia con dimensiones de 3×4 píxeles en P. Éste es una copia del bloque A, cuyas coordenadas en la imagen son $(2, 2)$, por lo que las cantidades:

$$\text{código del bloque copia}, 3, 4, 2, 2,$$

deben ser escritas, codificadas, en la salida.

Si no se localiza ningún bloque idéntico a A (proponemos que los bloques deben tener un tamaño mínimo determinado, tal como 4×4), el codificador marca P y continúa con el siguiente píxel. Por consiguiente, P se convierte en un píxel *punt*. Supongamos que P y los cuatro píxeles que le siguen son *punts*, pero el siguiente es el inicio de un bloque de copia o de relleno. El codificador prepara las cantidades:

$$\text{código del bloque punt}, 5, P_1, P_2, P_3, P_4, P_5,$$

donde los P_i son píxeles *punt*, y son escritos, codificados, en la salida. La Figura 4.141a muestra cómo los primeros seis píxeles de la imagen son todos diferentes y, por lo tanto, forman un bloque *punt*. El séptimo píxel, denotado x, es idéntico al primer píxel, y por lo tanto tiene una oportunidad de iniciar un bloque de relleno o de copia.

En cada uno de estos casos, el codificador marca los píxeles del bloque actual "codificado", y los salta en su barrido recurrente de la imagen. La Figura 4.141a muestra cómo el orden de escaneo se ve afectado cuando se identifica un bloque.

El decodificador escanea la imagen en orden *raster*. Para cada píxel P introduce el siguiente código de bloque desde el *stream* comprimido. Éste puede ser el código de un bloque de copia, un bloque de relleno, o un bloque *punt*. Si éste es el código de un bloque de copia, el decodificador sabe que irá seguido por un par de dimensiones y un par de coordenadas de la imagen. Una vez que el decodificador introduce aquellos, conoce las dimensiones del bloque y desde dónde copiarlo. El bloque es copiado y anclado a P como su esquina superior izquierda. El decodificador continúa su barrido recurrente, pero salta todos los píxeles del bloque recién construido (y cualesquier otros bloques construidos con anterioridad). Si el siguiente código de bloque es el de un bloque de relleno o de un bloque *punt*, el decodificador genera el bloque de píxeles, y continúa con el escaneo en orden *raster*, saltando los píxeles ya generados. La tarea del decodificador, por tanto, es muy sencilla, haciendo este método ideal para su uso por los exploradores web, donde una decodificación rápida es una necesidad.

Es evidente que FABD es un método altamente asimétrico. El trabajo del decodificador consiste sobre todo en seguir punteros y copiar píxeles; pero el codificador tiene que identificar los bloques de relleno y de copia. En principio, el codificador tiene que escanear la imagen completa para cada píxel P, en busca de bloques cada vez más grandes idénticos a los que se inician en P. Si la imagen se compone de n píxeles, ésto implica n^2 búsquedas, donde cada búsqueda tiene que examinar varios, incluso posiblemente muchos, píxeles. Para $n = 10^6$, el número de búsquedas es 10^{12} y el número de píxeles examinados pueden ser $2-3$ órdenes de magnitud más grandes. Este tipo de búsqueda exhaustiva puede tardar horas en los equipos actuales y tiene que ser mejorado drásticamente antes de que el método pueda ser considerado práctico. La discusión de aquí explora la manera de acelerar la búsqueda del codificador. Las dos características siguientes son obvias:

1. La búsqueda examinan sólo los píxeles pasados, no los futuros. La búsqueda en toda la imagen tiene que ser efectuada sólo para los últimos píxeles. No hay ninguna búsqueda en absoluto para el primer píxel. Esto reduce el número total de búsquedas de n^2 a $^{n^2}/_2$ en promedio. De hecho, la aplicación real de FABD intenta iniciar bloques sólo en las ubicaciones de píxeles no codificados —lo que reduce el número de búsquedas de n^2 a $n^2/\left(\text{tamaño de bloque medio}\right)^2$—.

2. No hay ninguna búsqueda para los píxeles de un bloque de relleno denso. El proceso de identificación de tales bloques es sencillo y rápido.

Todavía queda un gran número de búsquedas, y se aceleran con la siguiente técnica. El tamaño mínimo del bloque puede ser limitado a 4×4 píxeles sin reducir la magnitud de la compresión, ya que los bloques más pequeños que 4×4 no son mucho más grandes que un solo píxel. Ésto sugiere la construcción de una lista con todos los patrones de píxel de 4×4 encontrados hasta el momento considerado en la imagen (en orden inverso, de modo que los patrones recientemente encontrados se colocan en el inicio de la lista). Si el píxel actual es P, el codificador construye el bloque B de 4×4 comenzando en P, y busca en la lista de ocurencias de B. La lista puede ser sumamente larga. Para una imagen binivel, donde cada píxel es un bit, el número total de patrones de 16 bits es $2^{16} = 65\,536$. Para una imagen con píxeles de 8 bits, el número total de tales patrones es $2^{8\cdot16} = 2^{128} = 3{,}4 \cdot 10^{38}$. No deben aparecer todos ellos en una determinada imagen, pero el número de patrones que se producen aún puede ser grande, y son necesarias más mejoras. Aquí hay tres mejoras:

3. La lista de patrones debe incluir cada patrón encontrado hasta el momento actual *sólo una vez*. Nuestra lista se llama ahora lista principal, y se convierte en una lista de listas. Cada elemento de la lista principal comienza con un patrón único de 4×4, seguido por una lista de coincidencias con las localizaciones de la imagen donde se ha encontrado dicho patrón hasta ahora. La Figura 4.141b muestra un ejemplo. Observe cómo cada patrón apunta a una lista de coincidencias formada por coordenadas que comienzan con los píxeles encontrados recientemente. Siempre se añade un nuevo ítem al inicio de la lista de coincidencias, por lo que esas listas se mantienen, de forma natural, ordenadas. Los elementos de una lista de coincidencias están ordenados (descendentemente) por filas; y cada fila, por columnas.

4. Se puede utilizar el *hashing* para localizar un patrón de 4×4 en la lista principal. A los bits de un patrón se les aplica un algoritmo hash, produciendo un número de 14 bits que se emplea como un puntero (para los píxeles de 1 bit, un patrón consta de 16 bits. La aplicación real de FABD asume píxeles de 8 bits, dando lugar a patrones de 128 bits). Éste apunta a un ítem de un arreglo de punteros (la tabla hash). El seguimiento de un puntero de la tabla hash lleva al codificador al comienzo de una lista de coincidencias. Debido a las colisiones de hash, puede ser seleccionado más de un patrón de 16 bits con el mismo número de 14 bits producido en la operación hash, por lo que cada puntero en la tabla hash apunta, en realidad, a una lista (normalmente corta) cuyos elementos son listas coincidencias. La Figura 4.141c muestra un ejemplo.

5. Si la imagen tiene muchos píxeles y una gran redundancia, algunas listas de coincidencias pueden ser extensas. El tiempo total de compresión depende en gran medida de una búsqueda rápida en las listas de coincidencias, así que tiene sentido para limitar esas búsquedas. Una implementación práctica puede tener un parámetro k que limite la profundidad de la búsqueda en una lista de coincidencias. Por tanto, el establecimiento de $k = 1000$ limita la búsqueda en una lista de coincidencias a sus 1000 mejores elementos. Ésto reduce el tiempo de búsqueda, ejerciendo sólo tienen un efecto perjudicial mínimo en la razón de compresión final. La experiencia demuestra que incluso valores tan bajos como $k = 50$ pueden ser útiles. Este valor puede aumentar la relación de compresión en un pequeño tanto por ciento, si bien reduce el tiempo de compresión de una imagen típica de $1\text{K} \times 1\text{K}$ a tan sólo unos segundos. Otro efecto beneficioso de limitar la profundidad de búsqueda tiene que ver con las regiones difíciles de comprimir de la imagen. Estas regiones pueden aparecer raramente, pero sin embargo, tienden a aumentar el tiempo total de compresión de manera significativa.

Denotando el píxel actual por P, la tarea del codificador es: (1) construir el bloque B de 4×4 del cual P es la esquina superior izquierda, (2) efectuar la operación hash sobre los 16 valores de píxeles de B, produciendo un puntero de 14 bits, (3) seguir el puntero a la tabla hash y, desde allí, a una corta

lista principal, (4) buscar en esta lista principal linealmente, para encontrar una lista de coincidencias que comience con B, y (5) buscar los primeros k ítems en esta lista de coincidencias. Cada ítem es el lugar del comienzo de un bloque que empareja con con P en por lo menos 4×4 píxeles. La mayor coincidencia es seleccionada.

La última característica interesante de FABD tiene que ver con la transformación de las cantidades que deben ser codificadas, antes de la codificación real. Esto se basa en la *localidad espacial* de las imágenes de tonos discretos. Esta propiedad implica que un bloque normalmente se copiará de un bloque fuente cercano. Además, una región determinada tiende a tener sólo unos pocos colores, incluso si la imagen en general, puede tener muchos colores.

Por consiguiente, la localidad espacial sugiere el uso de *coordenadas de imagen relativas*. Si el píxel actual está situado en, digamos, $(81, 112)$ y es una copia de un bloque ubicado en $(41, 10)$, entonces la localización del bloque origen se expresa medicante el par $(81 - 41, 112 - 10)$ de coordenadas relativas. Las coordenadas relativas son números pequeños y también se distribuyen uniformemente. Por lo tanto, se comprimen bien con la codificación de Huffman.

La localidad espacial también sugiere el uso de la *edad del color*. Ésta es una manera sencilla de asignar códigos relativos a los colores. La edad de un color C es el número de colores únicos situados entre la instancia actual de C y su instancia previa. Como ejemplo, dada la secuencia de colores:

$$\text{verde}, \text{amarillo}, \text{rojo}, \text{rojo}, \text{rojo}, \text{verde}, \text{rojo},$$

sus edades de color son:

$$?, ?, ?, 0, 0, 2, 1.$$

Es evidente que las edades de color en una imagen con la localidad espacial son números pequeños. La primera vez que se ve un color no tiene una edad, por lo que no se codifica.

Los experimentos con FABD en la compresión de imágenes de tonos discretos obtienen un rendimiento entre 0,04 bpp (para imágenes binivel) y 0,65 bpp (para imágenes de 8 bits).

El autor está en deuda con Jeff Gilbert por la revisión de esta sección.

4.29. Codificación predictiva mediante árboles binarios

La codificación predictiva mediante árboles binarios (*Binary tree predictive coding* o BTPC) está diseñada para la compresión —sin y con pérdidas— de todo tipo de imágenes. El método se basa en el concepto de imagen piramidal, y su modo con pérdidas también utiliza la cuantificación. BTPC fue diseñado para satisfacer los criterios siguientes:

1. Debería ser capaz de comprimir imágenes de tonos continuos (fotográficas), de tonos discretos (gráficas), y mixtas, así como los métodos estándar para cada uno.
2. La opción con pérdidas no debe exigir un cambio fundamental en el algoritmo básico.
3. Cuando se comprime la misma imagen en varias ocasiones, perdiendo cada vez más y más datos, el resultado visual debe aparecer como una imagen gradualmente borrosa de las imágenes descomprimidas, no como cambios repentinos en la calidad de la imagen.
4. Una implementación en software debe ser eficiente en sus requisitos de tiempo y memoria. Los requisitos de memoria del decodificador deben ser un poco mayores que el tamaño de la imagen. El tiempo de decodificación (número de pasos) debe ser un múltiplo pequeño del tamaño de la imagen.
5. Una implementación en hardware de ambos, codificador y decodificador, debe permitir un paralelismo de grano fino. En el caso ideal, el hardware dispone de suficientes procesadores para que cada procesador sea capaz de procesar un bit de la imagen.

Hay dos versiones, BTPC1 y BTPC2 [Robinson 97]. Aquí se describen los detalles de ambos, y se utiliza el nombre BTPC para las características que son comunes para ambos.

La principal novedad de BTPC es el uso de una *imagen piramidal binaria*. La técnica descompone repetidamente la imagen en dos componentes, una de banda baja L, que se descompone de forma recursiva, y una de banda alta H. La banda baja es una zona de baja resolución de la imagen. La banda alta contiene diferencias entre L y la imagen original. Estas diferencias se utilizan posteriormente por el decodificador para reconstruir la imagen a partir de L. Si la imagen original está altamente correlacionada, L contendrá valores correlacionados (i.e., muy redundantes), pero aún contribuirá a la compresión global, puesto que es pequeño. Por otro lado, las diferencias contenidas en H están descorrelacionadas (i.e., con poca o ninguna redundancia izquierda), será un número pequeño, y tendrá un histograma que los picos de alrededor de cero. Por lo tanto, su entropía será pequeña, lo que hace posible comprimirlos eficientemente con un codificador de entropía. Puesto que los valores diferencia en H son pequeños, es natural obtener una compresión con pérdida cuantificandolos, un proceso que genera muchos ceros.

La razón de compresión final depende de cuán pequeña sea L y el grado de descorrelación de los valores de H. La idea principal de pirámide binaria utilizada por BTPC es descomponer la imagen original en dos bandas L_1 y H_1, descomponer L_1 en bandas L_2 y H_2, y continuar la descomposición de las bandas bajas hasta obtener las bandas L_8 y H_8. Las ocho bandas altas y la última banda baja L_8 son escritas en el *stream* comprimido después de haber sido codificadas entrópicamente. Ellos constituyen la pirámide de la imagen binaria, y son utilizadas por el decodificador BTPC para reconstruir la imagen original.

Para el codificador resulta natural escribirlos en el orden: L_1, H_1, H_2, ..., H_8. El decodificador, sin embargo, los necesita en el orden inverso, así que tiene sentido para el codificador albergar estas matrices en la memoria y escribirlas en orden inverso. El decodificador necesita sólo un buffer de memoria, el tamaño de la imagen original, y algo más de memoria para introducir una fila de H_i desde el *stream* comprimido. Los elementos de la fila se utilizan para procesar píxeles en el buffer, y la fila siguiente es entonces introducida.

Si la imagen original tiene $2^n \times 2^n = N$ píxeles, entonces la banda H_1 contiene $N/2$ elementos, la banda H_2 tiene $N/4$ elementos, y las bandas L_8 y H_8 tienen $N/2^8$ elementos cada una. El número total de valores a codificar, por lo tanto, es:

$$(N/2 + N/2^2 + N/2^3 + \cdots + N/2^8) + N/2^8 = N\left(1 - 2^{-8}\right) + N/2^8 = N.$$

Si el tamaño de la imagen original tiene $2^{10} \times 2^{10} = 2^{20}$ píxeles, entonces las bandas L_8 y H_8 tienen $2^{12} = 4096$ elementos cada una. Cuando se trata de imágenes más grandes, puede ser mejor continuar más allá de L_8 y H_8 bajando a matrices de tamaño 2^{10} a 2^{12}.

La Figura 4.142 ilustra los detalles de la descomposición BTPC. La Figura 4.142a muestra una imagen en escala de grises de 8×8 altamente correlacionada (observe que los valores de píxel crecen de 1 a 64). La primera banda baja L_1 se obtiene eliminando cada píxel en posición par de cada fila impar y cada píxel en posición impar de cada fila par. El resultado (mostrado en la Figura 4.142b) es una retícula rectangular con ocho filas y cuatro píxeles por columna. Contiene la mitad del número de píxeles de la imagen original. Dado que sus elementos son píxeles, la llamamos *banda de submuestreo* (*subsampled band*). Las posiciones de los píxeles eliminados se denominan H_1 y sus valores se muestran en letra pequeña en la Figura 4.142c. Cada H_1 se calcula restando el valor del píxel original en ese lugar del promedio de los píxeles de L_1 ubicados directamente por encima y por debajo de él. Por ejemplo, el valor 3 de H_1 en la fila superior, columna 2, se obtiene restando el píxel original 2 del promedio $(10+0)/2 = 5$ (usamos una *regla de borde* simple, donde los píxeles ausentes a lo largo de los bordes de la imagen se consideran cero, con el fin de predecir los píxeles). El valor -33 de H_1 en la esquina inferior izquierda se obtiene restando 57 del promedio $(0+49)/2 = 24$. Para simplificar, en estos ejemplos usamos sólo números enteros. Una implementación práctica, sin embargo, debe enfrentarse

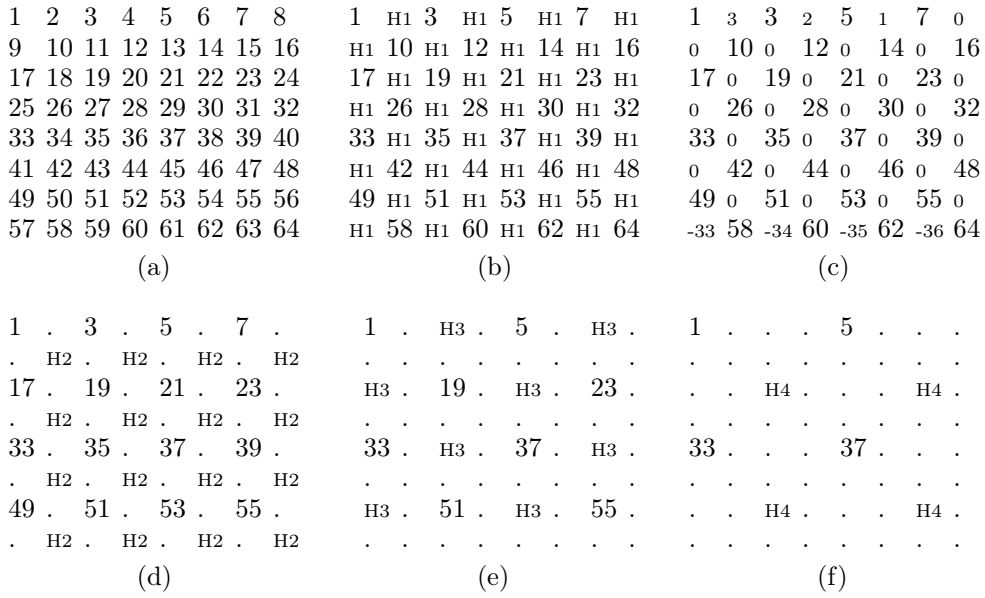


Figura 4.142: Una imagen de 8×8 y sus tres primeras bandas L.

con valores reales. Además, los métodos de predicción utilizados en la realidad por BTPC1 y BTPC2 son más sofisticados que el sencillo método que aquí se muestra. Ellos se discuten a continuación. Las diversas bandas H_i se llaman *bandas de diferencia*.

Entre las bandas de diferencia, H_1 es la más fina (ya que contiene la mayoría de los valores), y H_8 la más gruesa. Similarmente, L_8 (que es la única banda de submuestreo escrita en el *stream* comprimido) es la banda de submuestreo más gruesa.

La Figura 4.142d muestra cómo la segunda banda baja L_2 se obtiene a partir de L_1 eliminando los píxeles de las filas pares. El resultado es un patrón cuadrado que contiene la mitad del número de píxeles en L_1. También se muestran las posiciones de los valores de H_2. Observe que ellos no tienen vecinos encima o por debajo, por lo que usamos dos vecinos diagonales para la predicción (de nuevo, la predicción real utilizada por BTPC1 y BTPC2 es diferente). Por ejemplo, el valor -5 de H_2 en la Figura 4.143a, fue obtenido restando el píxel original 16 de la predicción $^{(23+0)}/_2 = 11$. La siguiente banda baja, L_3 (Figura 4.142e), se obtiene a partir de L_2 de la misma manera que L_1 se obtiene de la imagen original. La banda L_4 (Figura 4.142f) se obtiene a partir de L_3 de la misma manera que L_2 se obtiene a partir L_1. Cada banda contiene la mitad el número de píxeles de su predecesor. También es evidente que los cuatro vecinos más próximos de un valor H_i están ubicados en sus cuatro esquinas para valores pares de i y en sus cuatro lados para valores impares de i.

◇ **Ejercicio 4.51 (sol. en pág. 1088):** Calcúlense los valores de L_i y H_i, para $i = 2, 3, 4$. Para valores pares de i, úsese el promedio de los vecinos de la parte inferior izquierda y superior derecha para la predicción. Para valores impares de i hágase el promedio de los vecinos de arriba y abajo.

◇ **Ejercicio 4.52 (sol. en pág. 1090):** Utilícese la Figura 4.143a para calcular la entropía de la banda H_2.

El siguiente paso en la codificación BTPC es convertir la pirámide binaria en un árbol binario, similar al *bintree* de la Sección 4.30.1. Ésto es útil, porque muchos valores de H_i tienden a ser cero, especialmente cuando se utiliza la compresión con pérdida. Si un nodo v en este árbol es igual a cero y todos sus hijos son ceros, los hijos no serán escritos en el *stream* comprimido y v contendrá un código

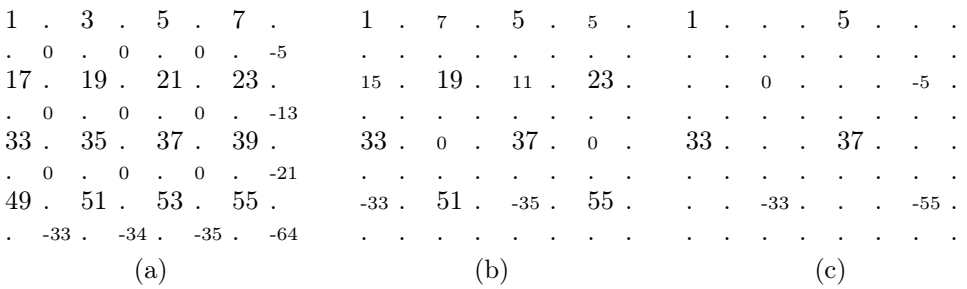


Figura 4.143: (a) Bandas L_2 y H_2. (b) Bandas L_3 y H_3. (c) Bandas L_4 y H_4.

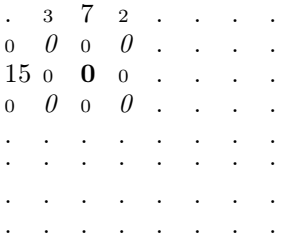


Figura 4.144: Los catorce descendientes de los ceros de H_4.

de terminación especial para comunicarle al decodificador que debe sustituir los hijos por ceros. La regla utilizada por BTPC para construir el árbol le dice cómo asociar un valor diferencia en H_i con sus dos hijos en H_{i-1}:

1. Si i es par, un valor en H_i tiene un hijo de $i/2$ filas por encima de él y otro hijo de $i/2$ columnas a su izquierda en H_{i-1}. Por consiguiente, el 0 en H_4 de la Figura 4.143c tiene los dos hijos 7 y 15 en H_3, en la Figura 4.143b. Además, el -33 en H_4 tiene los dos hijos 0 y -33 en H_3.

2. Para i impar, si el padre (en H_{i-1}) de un valor v en H_i se encuentra por debajo de v, entonces los dos hijos de v están ubicados en H_{i-1}, $(i-1)/2$ filas por debajo de él e $(i-1)/2$ columnas a su izquierda y derecha. Por ejemplo, el 5 en H_3 de la Figura 4.143b tiene su padre (el -5 de H_4) por debajo de él, por lo que sus dos hijos 0 y -5 se encuentran una fila por debajo y una columna a su izquierda y derecha en la banda H_2 de la Figura 4.143a.

3. Para i impar, si el padre (en H_{i-1}) de un valor v en H_i está a la derecha de v, entonces los dos hijos de v se encuentran en H_{i-1}, $(i-1)/2$ filas por debajo de él. Uno está $(i-1)/2$ columnas a su derecha, y el otro a $3(i-1)/2$ columnas a su derecha. Por ejemplo, el -33 de H_3 tiene su padre (el -33 de H_4) a su derecha, por lo que sus hijos son -33 y -34 de H_2. Éstos están ubicados una fila por debajo de él, y una y tres columnas a su derecha.

Estas normas parecen arbitrarias, pero tienen la ventaja de que los descendientes de un valor de diferencia forman, o bien un cuadrado, o bien un rectángulo a su alrededor. Por ejemplo, los descendientes del valor 0 de H_4 (denotados en la Figura 4.144 en negrita) son (1) de sus dos hijos, el 7 y el 15 de H_3, (2) de sus cuatro hijos, los cuatro ceros la parte superior izquierda de H_2 (denotados en cursiva), y (3) de los ocho nietos de H_1 (denotados en letra pequeña). Estos 14 descendientes se muestran en la Figura 4.144.

No todos estos descendientes son ceros, pero si lo fueran, el valor cero de H_4 tendría un código de terminación de cero especial asociado a él. Este código indica al decodificador que este valor y

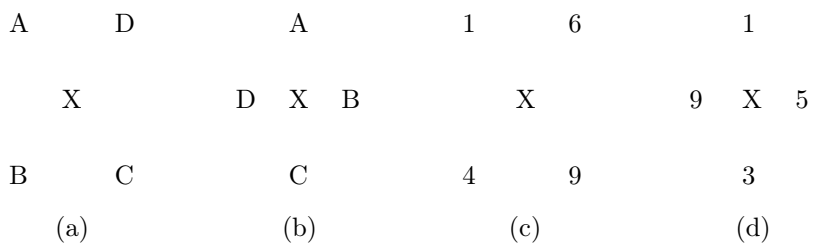


Figura 4.145: Los cuatro vecinos utilizados para predecir un valor X en H_i.

todos sus descendientes son ceros y no están escritos en el *stream* comprimido. BTPC2 añade otra peculiaridad al árbol. Utiliza la palabra de código (*codeword*) de la hoja para sus hermanos de la izquierda en H_1 para indicar que ambos hermanos son cero. Ésto mejora ligeramente la codificación de H_1.

Los experimentos muestran que rotando la pirámide binaria en un árbol binario se incrementa el factor de compresión significativamente.

Ahora debería quedar claro por qué BTPC es un candidato natural para la implementación en un computador paralelo. Cada valor de diferencia introducido por el decodificador desde una banda de diferencias H_i es utilizada para calcular un píxel en la banda de submuestreo L_i, y estos cálculos pueden hacerse en paralelo, porque son independientes (excepto por la competencia debida a los accesos a memoria).

Pasamos ahora a la predicción de los píxeles usados por BTPC. El objetivo es conseguir una predicción sencilla, con sólo dos o cuatro vecinos de un píxel, si bien minimizando el error de predicción. La Figura 4.142 muestra cómo cada valor de diferencia H_i está ubicado en el centro de un grupo de cuatro píxeles que son conocidos por el decodificador desde la decodificación de las bandas H_{i-1}, H_{i-2}, etc. Por consiguiente, es una buena idea que el codificador utilice este grupo para predecir el valor H_i. La Figura 4.145a,b muestra que existen tres métodos para estimar un valor H_i en el centro X de un grupo, donde A y C son dos píxeles opuestos, y B y D son los otros dos. Dos estimaciones son las predicciones lineales $(A+C)/2$ y $(B+D)/2$, y la tercera es la bilineal $(A+B+C+D)/4$. Basándose en los resultados de los experimentos, los desarrolladores de BTPC1 decidieron usar la siguiente regla de predicción (observe que el decodificador puede imitar esta regla):

> **Regla:** Si los dos píxeles extremos (i.e., los más grandes y más pequeños) de A, B, C y D son opuestos entre sí en el cuadrado de predicción, se utiliza la media de los otros dos píxeles opuestos (i.e., los dos centrales de los cuatro valores). De lo contrario, se utiliza el promedio de los dos píxeles opuestos más próximos en valor entre ellos.

Los dos valores extremos en la Figura 4.145c son 1 y 9. Éstos son opuestos entre sí, por lo que la predicción es el promedio 5 de los otros dos píxeles opuestos: 4 y 6. En la Figura 4.145d, por otra parte, los dos valores extremos, que son los mismos 1 y 9, no son opuestos entre sí, por lo que la predicción es el promedio 2, de 1 y 3, ya que son los píxeles opuestos más próximos en valor.

◇ **Ejercicio 4.53 (sol. en pág. 1090):** Parece que la mejor predicción se obtiene cuando el codificador prueba las tres formas de estimar X y selecciona la mejor. ¿Por qué no se usa esta predicción?.

Cuando BTPC2 fue desarrollado, se llevaron a cabo extensos experimentos, teniendo como efecto una predicción más sofisticada. BTPC2 selecciona una de las 13 predicciones diferentes, dependiendo de los valores y posiciones de los cuatro píxeles que forman el cuadrado de predicción. A estos píxeles se les asigna nombres a, b, c, y d, de tal manera que se satisfagan $a < b < c < d$. Los 13 casos se resumen en la Tabla 4.147.

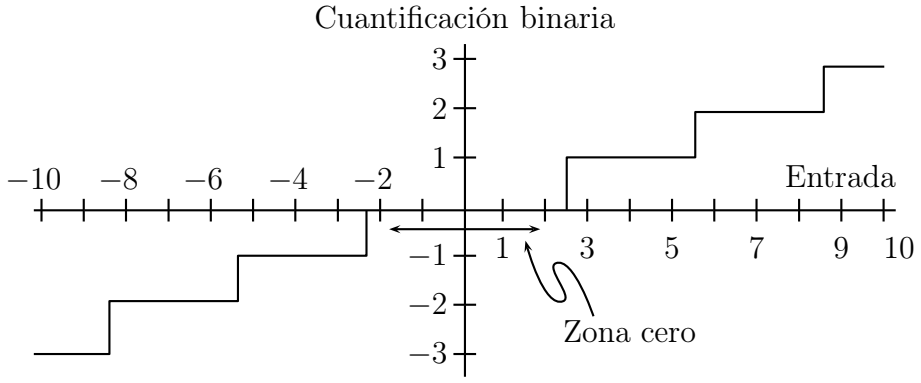


Figura 4.146: Cuantificación en BTPC.

Como se ha mencionado, BTPC tiene una opción natural con pérdidas que cuantifica las predicciones de diferencias (los valores H_i). El objetivo principal de la cuantificación es aumentar el número de diferencias cero, por lo que utiliza una zona cero de tamaño doble, ilustrada en la Figura 4.146. El tamaño del paso cuantificador en esta figura es 3, por lo que los valores 3, 4, y 5 se cuantifican a 4 (cuantificación bin 1). Los valores -6, -7, y -8 se cuantifican a -7 (bin -2), pero los valores 0, 1, y 2 y se cuantifican a 0 (bin 0), así como los valores 0, -1, y -2.

La cantidad de cuantificación varía de un nivel a otro en la pirámide. La primera banda de diferencias H_1 (la más fina) es la más rudamente cuantificada, ya que tiene la mitad del tamaño de la imagen y sus valores de diferencia afectan a píxeles individuales. Cualquier error de cuantificación en este nivel no se propaga a las diferencias de las bandas siguientes. La principal decisión en el proceso de cuantificación es cómo variar el tamaño del paso de cuantificación s de un nivel a otro. El principio es establecer el tamaño de paso s_i del nivel actual H_i, de manera que cualquier valor que se cuantifique a 0 con el paso s_{i-1} del nivel precedente utilizando una predicción exacta (sin cuantificación), se cuantifique a 0 con la predicción inexacta obtenida realmente con el tamaño de paso s_i. Si podemos calcular el rango de predicción de errores causado por los errores de cuantificación más recientes, entonces el valor adecuado para s_i es el tamaño de paso previo s_{i-1} más el error máximo en la predicción. Puesto que este tipo de cálculo consume mucho tiempo, BTPC determina el tamaño de paso s_i para el nivel i mediante el escalado hacia abajo del paso anterior s_{i-1} con un factor constante a. Por consiguiente, $s_i = as_{i-1}$, donde la constante a satisface $a < 1$. Su valor fue determinado mediante un experimento para pertenecer al rango 0,75 a 0,8.

El último paso en la compresión BTPC es la codificación de entropía de L_8 y las ocho bandas de diferencia H_i. BTPC1 no incluye un codificador de entropía. Su salida puede ser enviada a cualquier codificador adaptativo sin pérdidas, tal como Huffman, aritmético o basado en diccionarios. BTPC2 incluye un codificador de Huffman adaptativo integrado. Este codificador es reiniciado con cada banda de diferencia, ya que cada banda es cuantificada de forma diferente, por lo que sus estadísticas son diferentes. Si el codificador no se reiniciara al comienzo de una banda H_i, podría producir una mala compresión mientras se efectúa la adaptación al modelo estadístico de H_i.

La estructura de árbol binario introduce otra característica que debe tenerse en cuenta. Cuando se encuentra una hoja con un código de terminación cero, ésto significa que el valor de la diferencia actual y todos sus descendientes son cero. Sin embargo, si encuentra un nodo interior con un valor cero en el árbol binario, y si su hijo izquierdo es una hoja, entonces su hijo derecho no puede ser una hoja (pues de lo contrario, el padre tendría dos hijos cero y él mismo sería una hoja). Como resultado, un hijo derecho cuyo padre o madre es cero y cuyo hermano izquierdo es una hoja, es especial en algún sentido. Debido a esta propiedad, BTPC2 utiliza tres codificadores de Huffman adaptativos para cada

banda de diferencias: uno para los hijos izquierdos, otro para los hijos derechos "normales", y el tercero para los hijos derechos "especiales".

Núm	Nombre	Píxeles	Predicción (BTPC1) Par opuesto más cercano	Predicción (BTPC2) Adaptativo
0	Flat (*plano*)	a a a a	a	a
1	High point (*punto más alto*)	a a a b	a	a
2	Line (*línea*)	a b b a	a ó b	a ó b (nota 1)
3	Aligned edge (*borde alineado*)	a a b b	$(a+b)/2$	a ó b (nota 2)
4	Low (*bajo*)	a b b b	b	b
5	Twisted edge (*borde con giro*)	a a b c	$(a+b)/2$	$(a+b)/2$
6	Valley (*valle*)	a b c a	a	a ó $(a+b)/2$ (nota 1)
7	Edge (*borde*)	a b b c	b	b
8	Double Twisted edge (*doble borde con giro*)	a b c b	$(a+b)/2$ ó $(b+c)/2$	b
9	Twisted edge (*borde con giro*)	a b c c	$(b+c)/2$	$(b+c)/2$
10	Ridge (*arista*)	a c c b	c	c ó $(a+b)/2$ (nota 1)
11	Edge (*borde*)	a b c d	$(b+c)/2$	$(b+c)/2$
12	Double Twisted edge (*doble borde con giro*)	a b d c	$(a+c)/2$ ó $(b+d)/2$	$(b+c)/2$
13	Line (*línea*)	a c d b	$(a+b)/2$ ó $(c+d)/2$	$(a+b)/2$ ó $(c+d)/2$ (nota 1)

Nota 1: El Flag es necesario para indicar la elección.
Nota 2: Dependiendo de otros valores circundantes.

Tabla 4.147: Las trece predicciones utilizadas por BTPC2.

4.30. Quadtrees

Una compresión quadtree de una imagen binivel se basa en el principio de la compresión de imágenes (Sección 1.4), que establece: Si seleccionamos un píxel al azar en la imagen, hay una buena posibilidad de que sus vecinos inmediatos tengan el mismo color o uno similar. El método quadtree escanea el mapa de bits, zona por zona, en busca de áreas compuestas por píxeles idénticos (áreas uniformes). Ésto debe compararse con la compresión de imágenes RLE (Sección 1.4), donde sólo se verifican los vecinos de la fila escaneada, a pesar de que los vecinos ubicados en la misma columna también pueden ser idénticos o muy similares.

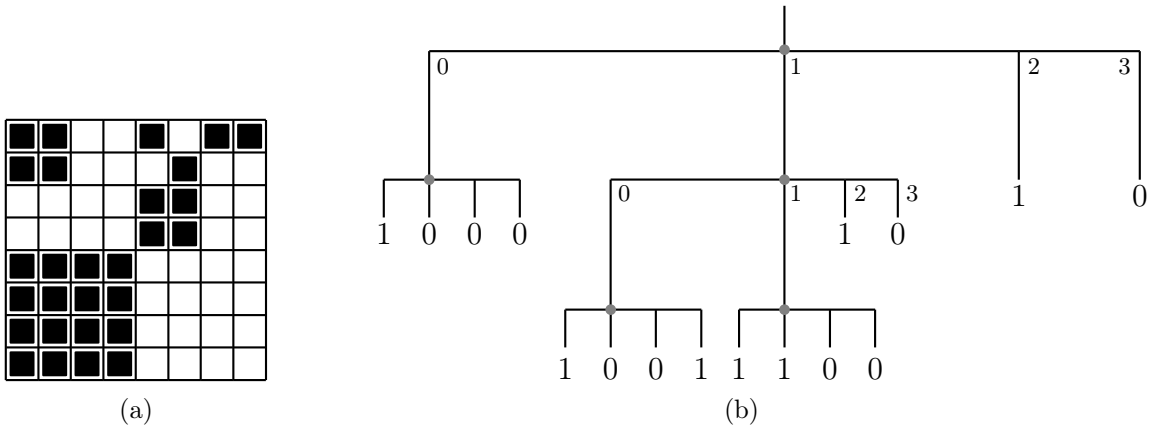


Figura 4.148: Un quadtree (árbol cuaternario).

La entrada está formada por píxeles de un mapa de bits (*bitmap*), y la salida es un árbol (un árbol de cuatro ramas, cuaternario, o *quadtree*, donde cada nodo es una hoja o tiene exactamente cuatro hijos). El tamaño del quadtree depende de la complejidad de la imagen. Para imágenes complejas, el árbol puede ser más grande que el mapa de bits original, lo que produce expansión. El método comienza con la construcción de un solo nodo, la raíz del quadtree final. El mapa de bits se divide en cuatro cuadrantes, cada uno se convierte en un hijo de la raíz. Un cuadrante uniforme (uno donde todos los píxeles tienen el mismo color) se guarda como un hijo hoja de la raíz. Un cuadrante no uniforme, se guarda como un hijo (nodo interior) de la raíz. Todos los cuadrantes no uniformes son entonces divididos recursivamente en cuatro subcuadrantes más pequeños que se guardan como cuatro nodos hermanos del quadtree. La Figura 4.148 muestra un ejemplo sencillo.

El bitmap de 8×8 en 4.148a produce el quadtree de 21 nodos de 4.148b. Dieciséis nodos son hojas (cada uno contiene el color de un cuadrante, 0 para el blanco, 1 para el negro), y los otros cinco (los círculos grises) son nodos interiores que contienen cuatro punteros cada uno. La numeración del cuadrante utilizada es $\left(\begin{smallmatrix} 0 & 1 \\ 2 & 3 \end{smallmatrix}\right)$ (sin embargo, el Ejercicio 4.68 muestra un esquema de numeración más natural).

El tamaño de un quadtree depende de la complejidad de la imagen. Suponiendo un tamaño de bitmap de $2^N \times 2^N$, un caso extremo es cuando todos los píxeles son idénticos. El quadtree, en este caso consta de un solo nodo, la raíz. El otro caso extremo es cuando cada cuadrante, incluso el más pequeño, no es uniforme. El nivel más bajo del quadtree tiene, en tal caso, $2^N \times 2^N = 4^N$ nodos. El nivel directamente por encima de él tiene una cuarta parte de ese número (4^{N-1}), y el nivel superior a éste tiene 4^{N-2} nodos. El número total de nodos en este caso es: $4^0 + 4^1 + \cdots + 4^{N-1} + 4^N = (4^{N+1}-1)/3 \approx 4^N\,(4/3) \approx 1{,}33 \times 4^N = 1{,}33\left(2^N \times 2^N\right)$. En éste peor caso el quadtree contiene alrededor del 33 % más nodos que el número de píxeles (el tamaño del mapa de bits). Tal imagen, por lo tanto, genera una expansión considerable cuando se convierte en un quadtree.

Un enfoque no recursivo para generar un quadtree comienza con la construcción del árbol cuaternario completo asumiendo que todos los cuadrantes no son uniformes y comprobando dicha asunción. Cada vez que se comprueba un cuadrante y resulta ser uniforme, los cuatro nodos correspondientes a sus cuatro cuartos son eliminados del quadtree. Este proceso procede desde abajo (las hojas) y hacia la raíz. Los pasos principales son los siguientes:

1. Se construye un quadtree completo de altura N. Éste contiene los niveles 0, 1, ..., N, donde el nivel k tiene 4^k nodos.

2. Todos los $2^N \times 2^N$ píxeles se copian desde el mapa de bits, en las hojas (el nivel más bajo) del

quadtree.

3. El árbol es escaneado nivel por nivel, desde la parte inferior (nivel N) hasta la raíz (nivel 0). Cuando el nivel k es escaneado, sus 4^k nodos son examinados en grupos de cuatro; los cuatro nodos en cada grupo tienen un padre en común. Si los cuatro nodos en un grupo son hojas y tienen el mismo color (i.e, si representan un cuadrante uniforme), son eliminados, y sus padres trasladados desde un nodo interior a una hoja que tenga el mismo color.

Es fácil analizar la complejidad en tiempo de este enfoque. Un árbol de cuaternario completo tiene alrededor de $1{,}33 \times 4^N$ nodos, y puesto que cada uno se comprueba una vez, el número de operaciones (en el paso 3) es $1{,}33 \times 4^N$. El paso 1 requiere $1{,}33 \times 4^N$ operaciones, y el paso 2 requiere 4^N operaciones. El número total de operaciones es, por tanto, $(1{,}33 + 1{,}33 + 1) \times 4^N = 3{,}66 \times 4^N$. Comparemos frente a frente el primer método —que requiere $(1/3) \times 4^N$ pasos— con el segundo método —que necesita $3{,}66 \times 4^N$ operaciones—. Puesto que N varía normalmente en el estrecho intervalo $8-12$, la diferencia no es muy significativa. Un análisis similar de los requisitos de almacenamiento muestra que el primer método únicamente necesita el espacio de memoria requerido por el quadtree final, mientras que el segundo método utiliza todo el almacenamiento necesario para un quadtree completo.

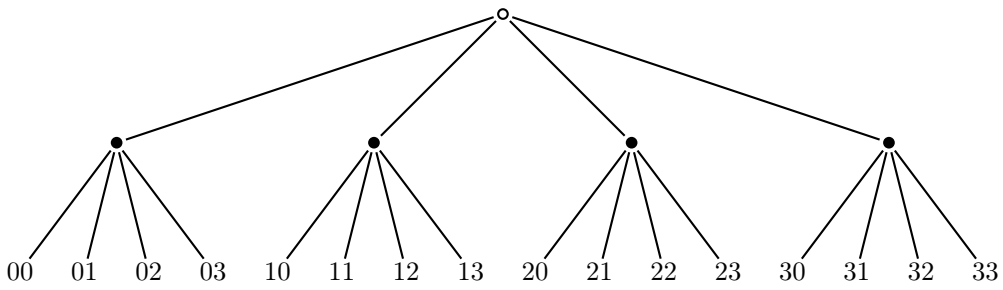
La siguiente discusión muestra la relación entre las posiciones de los píxeles en un quadtree y en la imagen. Imagine un quadtree completo. Su última fila está formada por todos los $2^N \times 2^N$ píxeles de la imagen. Supongamos que escaneamos estos píxeles de izquierda a derecha y los numeramos. Mostramos cómo se puede utilizar el número de un píxel para determinar sus coordenadas (x, y) en la imagen.

Cada cuadrante, subcuadrante, o píxel (o, en resumen, cada subcuadrado) obtenido con la partición del quadtree de una imagen, puede ser representado mediante una cadena de elementos en base 4 (sólo utiliza los cuatro dígitos 0, 1, 2 y 3). Cuanto más larga sea la cadena, menor es el que subcuadrado que representa. Denotamos tal cadena $d_i d_{i-1} \ldots d_0$, donde $0 \leq i \leq n$. Asumimos que la numeración del cuadrante de la Figura 4.171a (Sección 4.34) se extiende recursivamente para subcuadrados de todos los tamaños. La Figura 4.171b muestra cómo cada uno de los 16 subcuadrantes producidos a partir de los cuatro originales se identifica mediante un número en base 4 de 2 dígitos. Después de otra subdivisión, cada uno de los subsubcuadrantes resultantes se identifica mediante un número de 3 dígitos, y así sucesivamente. El área de color negro en la Figura 4.171c, por ejemplo, se identifica mediante el número entero en base cuatro 1032, y el área gris se identifica mediante el número entero 2011_4.

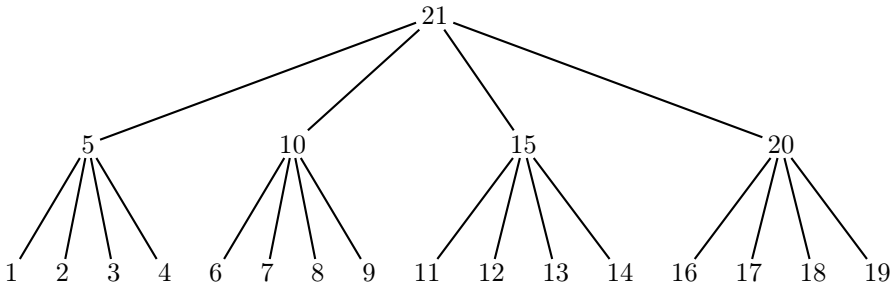
Los números en base 4 consecutivos son fáciles de generar. Simplemente tenemos que incrementar el dígito 3_4 al número de dos dígitos 10_4. Los primeros números en base 4 de n dígitos son (observe que hay $4 \times 2^{2n-2} = 2^n \times 2^n$ de ellos):

$$0, 1, 2, 3, 10, 11, 12, 13, 20, 21, 22, 23, 30, 31, 32, 33, 100, 101, 102, 103, \ldots,$$
$$\ldots, 130, 131, 132, 133, 200, \ldots, \underbrace{33\ldots3}_{n}.$$

La Figura 4.149a muestra un quadtree completo para una imagen de $2^2 \times 2^2$. Los 16 píxeles constituyen la fila inferior, y se muestran sus números cuaternarios (en base 4). Una vez que sabemos cómo localizar un píxel de la imagen mediante su número en base 4, podemos construir el quadtree accediendo de abajo arriba, y de izquierda a derecha. He aquí los detalles: Comenzamos con los cuatro píxeles con números cuaternarios 00, 01, 02, y 03. Éstos se convierten en la parte inferior izquierda del árbol y se numeran $1-4$ en la Figura 4.149b. Construimos su nodo padre (el número 5 en la Figura 4.149b). Si los cuatro píxeles son uniformes, se eliminan del quadtree. Hacemos lo mismo con el siguiente grupo de píxeles, cuyos números en base 4 son: 10, 11, 12 y 13. Se convierten en los píxeles $6-9$, con un padre numerado 10 en la Figura 4.149b. Si son uniformes, se eliminan. Ésto se repite hasta construir los cuatro padres, numerados 5, 10, 15, y 20. Se crea su padre (número 21), y se revisan



(a)



(b)

Figura 4.149: Un quadtree para una imagen de $2^2 \times 2^2$.

los cuatro nodos. Si son uniformes, que se eliminan. El proceso continúa hasta que se crea la raíz, sus cuatro nodos hijo se han revisado, y, si es necesario, eliminado. Éste es un enfoque recursivo, cuya ventaja es que no necesita memoria extra. Los nodos del quadtree innecesarios se eliminan tan pronto como ellos y sus padres se hayan creado.

Dada una imagen cuadrada con $2^n \times 2^n$ píxeles, cada píxel se identifica mediante un número en base 4 de n dígitos. Dado un número $d_{n-1}d_{n-2}\ldots d_0$, mostramos cómo localizar "su" píxel en la imagen. Asumimos que un píxel de la imagen tiene coordenadas (x, y), con el origen del sistema de coordenadas en la esquina inferior izquierda de la imagen. Comenzamos a analizar el origen y escaneamos las cifras cuaternarias de izquierda a derecha. Un dígito 1 nos dice que ascendamos para llegar a nuestro píxel objetivo. Un dígito 2 indica un movimiento hacia la derecha, y un dígito de 3 nos dirige arriba y a la derecha. Un dígito corresponde a ningún movimiento. La cantidad de movimiento reduce a la mitad de un dígito a otro. El dígito de más a la izquierda corresponde a un movimiento de 2^{n-1} filas y/o columnas, el segundo dígito desde la izquierda corresponde a un movimiento de 2^{n-2} filas y/o columnas, y así sucesivamente, hasta que el dígito de la derecha corresponde a movimientos de sólo 1 $(= 2^0)$ píxel (o ninguno, si este dígito es un 0). El pseudocódigo de la Figura 4.150 resume este proceso.

También hay que señalar que los quadtrees son un caso especial de la curva de Hilbert, discutida en la Sección 4.32.

```
x:=0; y:=0;
for k:= n − 1 step −1 to 0 do
  if digit(k)=1 or 3 then y := y + 2^k;
  if digit(k)=2 or 3 then x := x + 2^k
endfor;
```

Figura 4.150: Pseudo–código para localizar un píxel en una imagen.

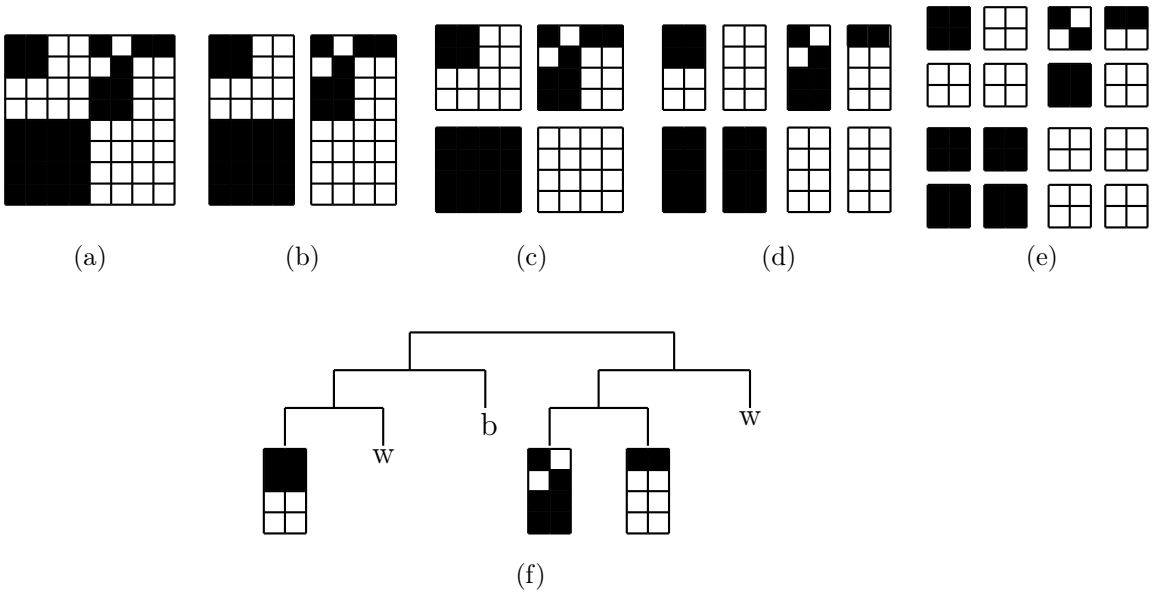


Figura 4.151: Un bintree (árbol binario) para una imagen de 8×8.

4.30.1. Bintrees (árboles binarios)

En lugar de dividir la imagen en cuadrantes, se puede dividir recursivamente en mitades. Éste es el principio del método *bintree*. La Figura 4.151a–e muestra la imagen de 8×8 de la Figura 4.148a y los cuatro primeros pasos en su partición bintree. Figura 4.151f muestra parte del bintree resultante. Es fácil ver cómo el método bintree alterna entre divisiones verticales y horizontales, y cómo las subimágenes que se generan incluyen todas aquellas producidas por un quadtree, además de algunas otras. Como método de compresión, la partición bintree es menos eficiente que el quadtree, pero puede ser útil en casos donde se necesitan muchas subimágenes. Un ejemplo de ello es el método WFA de la Sección 4.34. El método original utiliza un árbol quadtree para dividir una imagen en subcuadrados no superpuestos, y comprime la imagen, haciendo coincidir un subcuadrado con una combinación lineal de otros subcuadrados (posiblemente más grandes). Una extensión de WFA (página 511) utiliza bintrees para obtener más subimágenes y, por lo tanto, mejora la compresión.

4.30.2. Valores de composición y de diferencia

El promedio de un conjunto de números enteros es un buen representante de los enteros en el conjunto, pero no siempre es un número entero. La mediana de tal conjunto es un entero, pero no siempre es un buen representante. El método de imagen progresiva que se describe en esta sección, debido a K. Knowlton [Knowlton 80], emplea los conceptos *composite* (composición) y *differentiator*

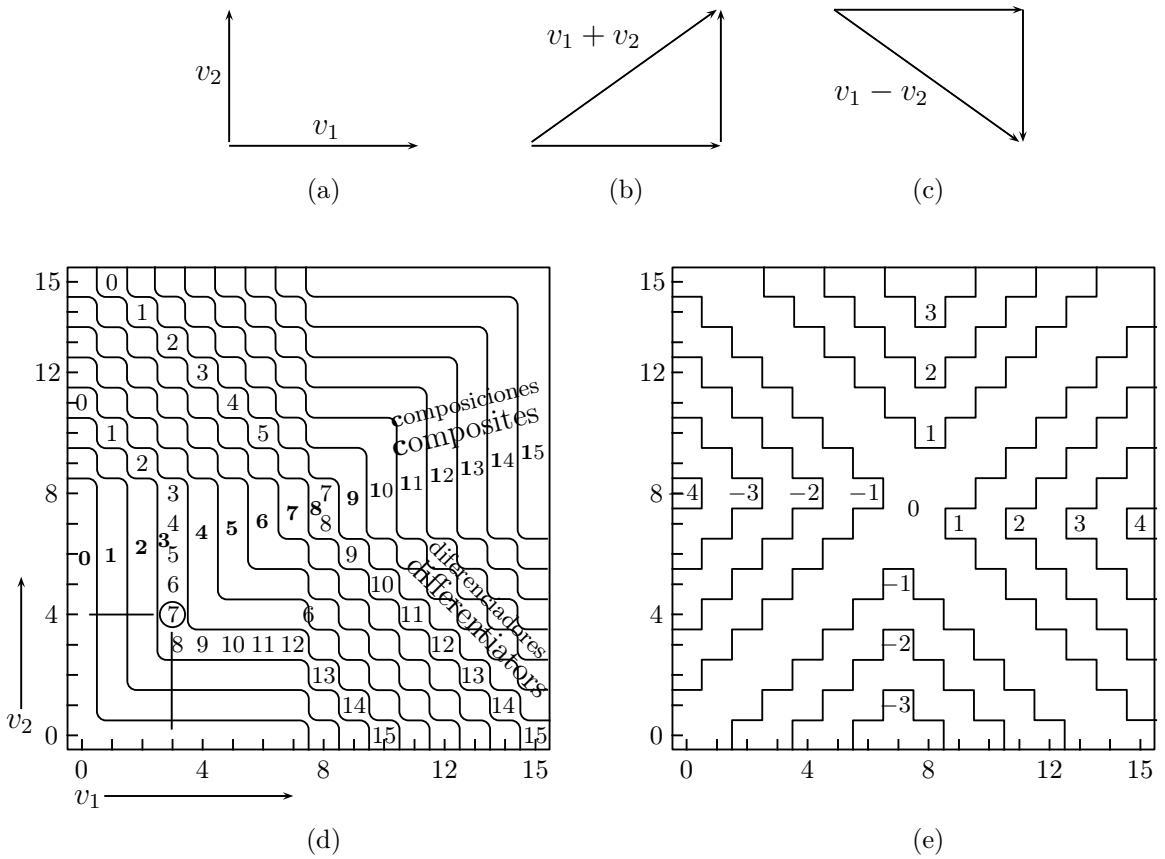


Figura 4.152: Determinación de los valores de composición y de diferencia.

(diferenciador) para para codificar una imagen de forma progresiva. La imagen es codificada en capas, donde las capas iniciales están formadas por unos pocos bloques grandes de baja resolución, seguidas por capas posteriores con pequeños bloques de alta resolución. La imagen se divide en capas utilizando el método de *bintrees* (Sección 4.30.1). La imagen completa se divide en dos mitades verticales, cada una de ellas se divide en dos cuadrantes horizontales, y así sucesivamente.

La primera capa consiste en un área única y uniforme, del tamaño de la imagen completa. Nosotros lo llamamos aproximación progresiva *de orden cero*. La siguiente capa es la *primera* aproximación. Se compone de dos rectángulos uniformes (células o celdas), uno encima del otro, cada uno de la mitad del tamaño de la imagen. Podemos considerarlos dos hijos de la única célula de aproximación de orden cero. En la *segunda* aproximación, cada una de estas células se divide en dos hijos, que son dos células más pequeñas, colocadas una al lado de la otra. Si la imagen original tiene 2^n píxeles, entonces éstos (los píxeles) son las células de la n-ésima aproximación, lo que constituye la última capa.

Si la imagen tiene 16 escalas de grises, entonces cada píxel se compone de cuatro bits, describiendo su intensidad. Los píxeles están ubicados en la parte inferior del bintree (son las hojas del árbol), por lo que cada célula de la capa inmediatamente superior a las hojas, es el padre de dos píxeles. Queremos representar dicha célula mediante dos números de 4 bits. El primer número, denominado *composite*, debe ser similar a un promedio. Debe ser un representante de los dos píxeles tomados como una unidad. El segundo número, llamado *differentiator*, debe reflejar la diferencia entre los píxeles. Utilizando los valores de composición y de diferencia (que son enteros de 4 bits) debería ser posible

reconstruir los dos píxeles.

La Figura 4.152a,b,c, muestra cómo dos números $v_1 = 30$ y $v_2 = 03$ pueden ser considerados los vectores $(3, 0)$ y $(0, 3)$, y cómo su suma $v_1 + v_2 = (3, 3)$ y su diferencia $v_1 - v_2 = (3, -3)$ corresponden a una rotación de 45° de los vectores. La suma puede ser considerada, en cierto sentido, la media de los vectores, y la diferencia puede utilizarse para reconstruirlos. Sin embargo, la suma y la diferencia de vectores binarios no son, en general, binarios. La suma y la diferencia de los vectores $(0, 0, 1, 1)$ y $(1, 0, 1, 0)$, por ejemplo, son $(1, 0, 2, 1)$ y $(-1, 0, 0, 1)$.

El método aquí propuesto para la determinación de los valores *composite* y *differentiator* está ilustrado en la Figura 4.152d. Sus 16 filas y 16 columnas corresponden a las 16 escalas de grises de nuestra hipotética imagen. El diagrama se divide en 16 bandas estrechas de 16 posiciones cada una. Las bandas están numeradas del 0 al 15, y estos números (mostrados en negrita) son los valores de composición (*composite*). Las 16 ubicaciones dentro de cada banda están también numeradas de 0 a 15 (esta numeración se muestra para las bandas 3 y 8), y son los diferenciadores (*differentiator*s). Dadas dos células adyacentes, $v_1 = 0011$ y $v_2 = 0100$ en una de las aproximaciones, utilizamos sus valores como sus coordenadas (x, y) de la tabla para determinar el valor de la composición $3 = 0011$ y del diferenciador $7 = 0111$ de sus padres.

Una vez que se tienen un par de valores *composite* y *differentiator*, se puede utilizar la Figura 4.152d para reconstruir los dos valores originales. Es obvio que dicho diagrama se puede diseñar de muchas formas diferentes, pero la característica importante de este diagrama en particular es la manera en que se determinan los valores *composite*. Siempre están cercanos la media verdadera de v_1 y v_2. Los elementos ubicados a lo largo de la diagonal principal, por ejemplo, satisfacen $v_1 = v_2$, y el diagrama está diseñado de tal manera que los valores compuestos c para estos elementos son $c = v_1 = v_2$. El diagrama se construye colocando una estrecha (de 1 unidad) banda, de longitud 16 en la esquina inferior izquierda, y añadiendo bandas similares que son simétricas respecto de la diagonal principal.

La mayor parte del tiempo, el valor *composite* que es determinado mediante este diagrama para un par v_1 y v_2 de números de 4 bits es su verdadera media, o difiere de la verdadera media en 1. Sólo en 82 de los 256 posibles pares (v_1, v_2) el valor de composición difiere de la media verdadera $(v_1+v_2)/2$ en más de 1. Esto es aproximadamente el 32 %. Las desviaciones de los valores de composición a partir de los promedios verdaderos se muestran en la Figura 4.152e. La desviación máxima es de 4, y se produce en sólo dos casos.

Puesto que los valores *composite* están tan cerca de la media, se utilizan para colorear las células de las diversas aproximaciones, lo que produce imágenes progresivas de aspecto realista. Ésta es la principal característica del método.

La Figura 4.153 muestra el árbol binario resultante de las diversas aproximaciones. La raíz representa la imagen completa, y las hojas son los píxeles individuales. Cada par de píxeles se convierte en un par (c, d) de valores *composite* y *differentiator* en el nivel superior de las hojas. Cada par de valores de composición en ese nivel se convierte en un par (c, d) en el nivel superior, y así sucesivamente, hasta que la raíz se convierte en uno de tales pares. Si la imagen contiene 2^n píxeles, entonces el nivel inferior (las hojas) contiene 2^n nodos. El nivel superior que contiene 2^{n-1} nodos, y así sucesivamente. La visualización progresiva de la imagen se realiza mediante el decodificador progresivo en etapas como sigue:

1. El par (c_0, d_0) para la raíz es introducido y la imagen se muestra como un gran bloque uniforme (0-ésima aproximación) de intensidad c_0.

2. Los valores c_0 y d_0 se utilizan para determinar, a partir de la Figura 4.152d, los dos valores de composición c_{10} y c_{11}. La primera aproximación, que consta de dos grandes células uniformes con intensidades c_{10} y c_{11}, es mostrada, reemplazando la 0-ésima aproximación. Se introducen dos valores de diferencia d_{10} y d_{11} para el siguiente nivel (segunda aproximación).

3. Los valores c_{10} y d_{10} se utilizan para determinar, a partir de la Figura 4.152d, los dos valores

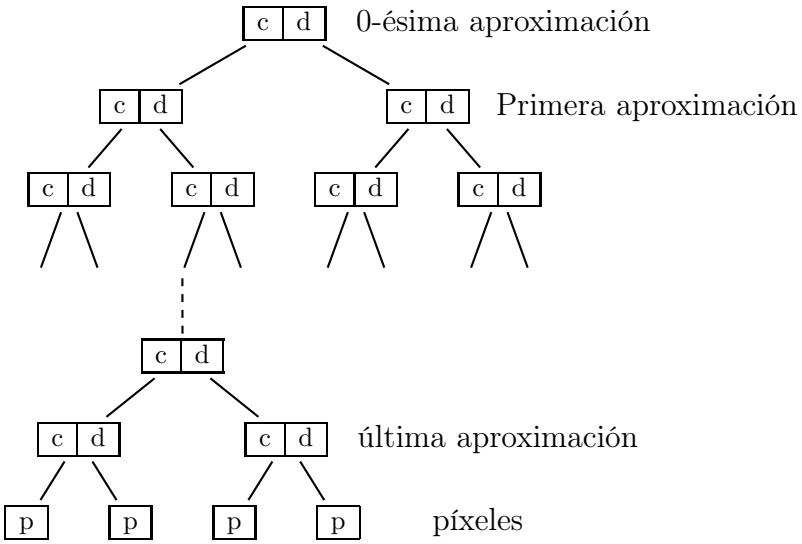


Figura 4.153: Aproximaciones sucesivas en un árbol binario.

de composición c_{20} y c_{21}. Similarmente, se usan los valores c_{11} y c_{11} para determinar los dos valores *composite* c_{22} y c_{23}. La segunda aproximación, que consta de cuatro células grandes con intensidades c_{20}, c_{21}, c_{22} y c_{23}, es mostrada, reemplazando la primera aproximación. Se introducen cuatro valores *differentiator* d_{2i} para el siguiente nivel (tercera aproximación).

4. Este proceso se repite hasta que, en el último paso, los 2^{n-1} valores de composición para la penúltima aproximación es determinada y mostrada. Los 2^{n-1} valores de diferencia para estos valores de composición son ahora introducidos, y cada uno de los 2^{n-1} pares es utilizado para calcular un par de píxeles. Se muestran los 2^n píxeles, completando la generación progresiva de la imagen.

Estos pasos muestran que el archivo de imagen debe contener el valor c_0, el valor d_0, los dos valores d_{10} y d_{11}, los cuatro valores d_{20}, d_{21}, d_{22} y d_{23}, y así sucesivamente, hasta los 2^{n-1} valores de diferencia para la penúltima aproximación. El total es un valor *composite* y $2^n - 1$ valores *differentiator*. Por consiguiente, el archivo de imagen contiene 2^n valores de 4 bits, por lo que su tamaño es igual a la de la imagen original. El método discutido hasta ahora es para la transmisión de imágenes progresiva y todavía no proporciona ningún tipo de compresión.

◇ **Ejercicio 4.54 (sol. en pág. 1090):** Dados los ocho valores de píxeles 3, 4, 5, 6, 6, 4, 5, y 8, constrúyase un árbol similar al de la Figura 4.153 y muéstrese el contenido del archivo de imagen progresiva.

El codificador empieza con los píxeles de la imagen original (las hojas del árbol binario) y se prepara el árbol de abajo arriba. El archivo de imagen, sin embargo, debe comenzar con la raíz del árbol, por lo que el codificador debe mantener todos los niveles del árbol en la memoria mientras el árbol está siendo generado. Afortunadamente, ésto no requiere ninguna memoria adicional. Los valores de los píxeles originales no son necesarios después de la primera etapa, y pueden ser desechados, dejando espacio para que los valores *composite* y *differentiator*. Los valores de composición se utilizan en el segundo paso y luego pueden ser también eliminados. Sólo los valores de diferencia tienen que ser guardados

Valor diff.	Recuento	Código de Huffman
0	27	00000000
1	67	00000001
2	110	0000001
3	204	000001
4	485	00001
5	1564	0001
6	4382	001
7	8704	01
8	10206	11
9	4569	101
10	1348	1001
11	515	10001
12	267	100001
13	165	1000001
14	96	10000001
15	58	10000000

Tabla 4.154: Posibles códigos de Huffman para valores de diferencia de 4 bits.

para todos los pasos, pero ellos pueden almacenarse en el espacio de memoria ocupado originalmente por la imagen.

Es evidente que las primeras aproximaciones son demasiado toscas para mostrar cualquier imagen reconocible. Sin embargo, implican muy pocas células y se calculan y despliegan rápidamente. Cada aproximación tiene dos veces el número de células de su predecesor, por lo que el tiempo que tarda en mostrarse una aproximación aumenta geométricamente. Puesto que los cálculos involucrados son simples, se puede estimar el tiempo necesario para generar y visualizar una aproximación considerando el tiempo que requiere introducir los valores necesarios para la siguiente aproximación; el tiempo para los cálculos puede ser ignorado. Asumiendo una velocidad de transmisión de 28 800 baudios, la aproximación de orden cero (introduce dos valores de 4 bits) consume $^{8}/_{28800} \approx 0{,}00028$ segundos, la primera aproximación (introduce otros dos valores de 4 bits) toma el mismo tiempo. La segunda aproximación (cuatro valores de 4 bits) lleva 0,000556 seg, y la décima aproximación (introduce 2^{10} números de 4 bits) consume 0,14 seg. Aproximaciones posteriores requieren 0,28, 0,56, 1,04, y 2,08 seg.

También es posible desarrollar la imagen progresivamente de una manera no uniforme. La forma más sencilla de codificar una imagen progresivamente es calcular los valores de diferencia de cada aproximación en orden *raster* y escribirlas en el archivo de imagen en este orden. Sin embargo, si estamos interesados en el centro de la imagen, podemos cambiar dicho orden, siempre y cuando lo hagamos de la misma manera para el codificador y el decodificador. El codificador puede escribir los valores de diferencia para el centro de la imagen primero, seguidos por los valores de diferencia restantes. El decodificador tiene que imitar ésto. Debe comenzar cada aproximación visualizando el centro de la imagen, seguido por el resto de la aproximación.

El archivo de imagen generado con este método puede ser comprimido mediante una codificación de entropía de los valores individuales en él. Todos los valores (excepto uno) en este archivo son de diferencias, y los experimentos indican que estos valores siguen una distribución normal. Para los píxeles de 4 bits, los valores *differentiator* están en el rango de 0 a 15, y la Tabla 4.154 muestra los recuentos típicos y los posibles códigos de Huffman para cada uno de los 16 valores. Ésto reduce los datos de cuatro bits por *differentiator* a aproximadamente 2,7 $^{bits}/_{\text{valor}}$.

Otra manera de comprimir el archivo de imagen es cuantificar los píxeles originales de 16 a 15

niveles de intensidad y utilizar el valor de píxel adicional como un código de terminación, indicando una célula uniforme. Cuando decodificador introduce este valor desde el archivo de imagen, no divide más la célula.

4.30.3. Compresión bintree progresiva

En la Sección 4.10 se discuten varias técnicas para la representación de imágenes progresivas. La Sección 4.30.2 describe una aplicación de los bintrees para la transmisión progresiva de imágenes en escala de grises. Esta sección (basada en [Knowlton 80]) describe un método similar para imágenes binivel. Ilustramos el método en una imagen con resolución 384×512. La imagen se divide en bloques de tamaño 3×2 cada uno. Hay $128 = 2^7$ filas y $256 = 2^8$ columnas de estas 6 tuplas, por lo que su número total es $2^{15} = 32\,768$.

El codificador construye un árbol binario dividiendo la imagen completa en dos mitades horizontales, cada una de ellas, en dos cuadrantes verticales, y continuando de esta manera, hasta el nivel de 6 tuplas. En la práctica, esto se realiza de abajo arriba, tras lo cual el árbol se escribe en el archivo de salida empezando por la parte superior hasta la parte inferior. La Figura 4.155 muestra el árbol final. Cada nodo está marcado como: negro uniforme (b, con el código prefijo 10), blanco uniforme ($w = 11$), o mixto ($m = 0$). Un nodo mixto también contiene otro código prefijo, tras el 0, para especificar uno de cinco tonos de gris. Un buen conjunto de estos códigos es:

$$110 \text{ para } 16{,}7\,\%, \quad 00 \text{ para } 33{,}3\,\%, \quad 01 \text{ para } 50\,\%, \quad 10 \text{ para } 66{,}7\,\%, \quad 111 \text{ para } 83{,}3\,\%.$$

Se necesitan cinco códigos, ya que un grupo de seis píxeles puede tener entre cero y seis píxeles blancos, correspondientes a siete tonos de gris. Dos tonos, digamos blanco y negro, no pueden producirse en un entorno mixto de tuplas de 6, por lo que sólo es necesario especificar los cinco tonos indicados arriba. El tamaño medio de estos códigos es de 2,4 bits, por lo que un nodo mezcla (m) del árbol tiene un promedio de 3,4 bits: el código 0 seguido por dos o tres bits.

Cuando el decodificador introduce uno de estos niveles, divide cada bloque del nivel precedente en dos bloques más pequeños y utiliza los códigos para pintarlos de negro, blanco, o uno de los cinco tonos de gris.

Los dos niveles de la parte inferior del árbol son diferentes. El penúltimo nivel contiene 2^{15} nodos, uno por cada tupla de 6. Un nodo en este nivel contiene uno de siete códigos, especificando el número de píxeles blancos en la tupla de 6. Cuando el decodificador llega este nivel, cada bloque es ya una tupla de 6, y se pinta el tono adecuado de gris, de acuerdo con el número de píxeles blancos en la tupla de 6. Cuando el decodificador llega al último nivel, ya sabe cuántos píxeles blancos contiene cada tupla de 6. Sólo necesita que le digan dónde se encuentran estos píxeles blancos en la tupla de 6. Esto también se realiza mediante códigos. Si una tupla de 6 contiene sólo un píxel blanco, ese píxel puede estar localizado en una de seis posiciones, por lo que se necesitan seis códigos de 3 bits. Una tupla de 6 con cinco píxeles blancos también requiere un código de 3 bits. Si una tupla de 6 contiene dos píxeles blancos, éstos pueden ubicarse en la tupla de 6 de 15 formas diferentes, por lo que son necesarios 15 códigos de 4 bits. Una tupla de 6 con cuatro píxeles blancos también requiere un código de 4 bits. Cuando una tupla de 6 contiene tres píxeles blancos, éstos pueden formar 20 configuraciones; por tanto, se utilizan 20 códigos prefijo (de tamaño variable), que varían en tamaño de 3 a 5 bits.

◇ **Ejercicio 4.55 (sol. en pág. 1090):** Muéstrense las 15 tuplas de 6 con dos píxeles blancos.

Ahora puede estimarse el tamaño del árbol binario de la Figura 4.155. Los niveles más altos (excluyendo los tres niveles inferiores) contienen entre 1 y 2^{14} nodos, dando un total de $2^{15} - 1 \approx 2^{15}$. El tercer nivel desde la parte inferior contiene 2^{15} nodos, por lo que el número total de nodos hasta ahora es 2×2^{15}. Cada uno de estos nodos contiene un código de 2 bits (para nodos b uniformes y nodos w) o un código de 3,4 bits (para nodos m). Suponiendo que la mitad de los nodos son mixtos, la

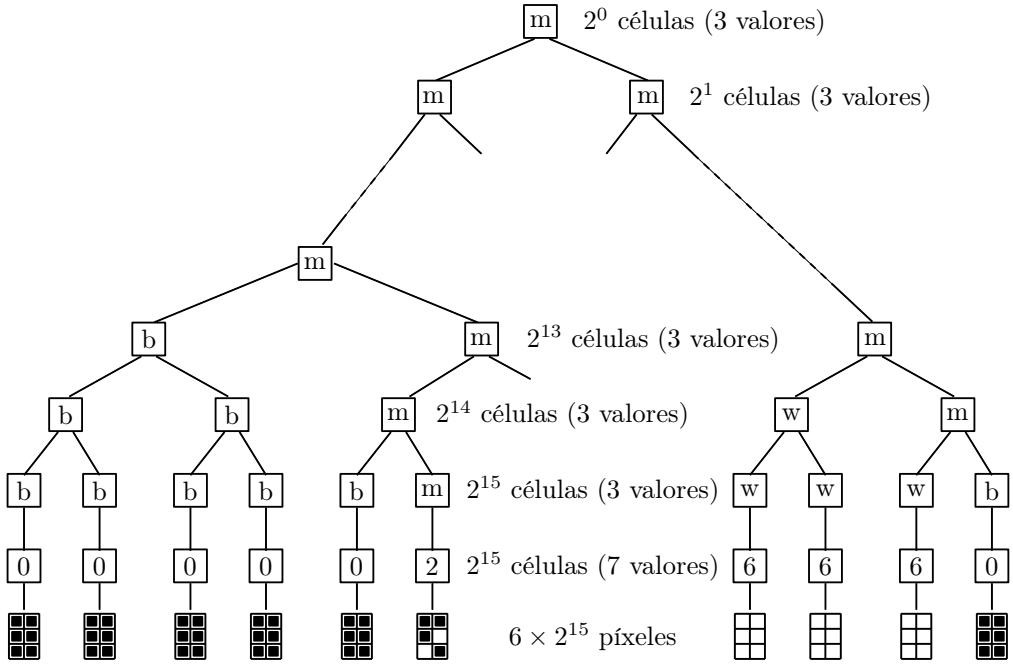


Figura 4.155: Un *bintree* (árbol binario) completo para una imagen binivel.

media es de 2,7 bits por nodo. El número total de bits hasta el momento es de $2 \times 2^{15} \times 2{,}7 = 5{,}4 \times 2^{15}$. Ésto es casi igual al tamaño original de la imagen (que es 6×2^{15}), ¡y todavía tenemos dos niveles más que recorrer!

El penúltimo nivel contiene 2^{15} nodos, con un código de 3 bits cada uno (uno de los siete tonos de gris). El nivel inferior también contiene 2^{15} nodos, con códigos de 3, 4 ó 5 bits cada uno. Asumiendo un tamaño promedio de 4 bits por código en este nivel, el tamaño total del árbol es:

$$(5{,}4 + 3 + 4) \times 2^{15} = 12{,}4 \times 2^{15},$$

¡más del doble del tamaño de la imagen! La compresión se consigue podando el árbol, como un quadtree, de tal manera que cualquier nodo b o w situado en la parte alta del árbol se convierte en una hoja. Los experimentos indican un factor de compresión típico de 6, lo que implica que el árbol se reduce en tamaño del doble a 1/6 del tamaño de la imagen: un factor de 12. [Knowlton 80] describe un esquema de codificación más compleja que produce factores típicos de compresión de 8.

Una ventaja del método aquí propuesto es el hecho de que produce bloques grises para los nodos del árbol de tipo m (mixto). Ésto significa que el método puede ser utilizado incluso con una pantalla binivel. Un bloque que consta de píxeles en blanco y negro se ve gris cuando lo observamos desde la distancia, y la cantidad de gris está determinada por la mezcla de píxeles en blanco y negro en el bloque. Los métodos que tratan de conseguir un aspecto agradable de bloques grises en una pantalla binivel se conocen como *dithering* (*interpolación*) [Salomon 99].

4.30.4. Compresión de estructuras *N*-tree

Los objetos encuentran en la vida real son normalmente tridimensionales, aunque muchos objetos están cercanos de ser bi- o incluso unidimensionales. En geometría, los objetos pueden tener cualquier número de dimensiones, aunque no podemos visualizar objetos en más de tres dimensiones. Una

estructura de datos N-tree es un árbol especial que almacena un objeto N-dimensional. El ejemplo más común es un quadtree (Sección 4.30), una popular estructura para almacenar un objeto bidimensional, tal como una imagen.

Un *octree* es la extensión obvia de un quadtree a tres dimensiones [Samet 90a,b]. Un octree no se utiliza para la compresión de datos; es una estructura de datos donde puede ser almacenado un objeto tridimensional. En un octree, un nodo, o bien es una hoja, o bien tiene exactamente ocho hijos. El objeto se divide en ocho octantes, cada octante no uniforme se divide recursivamente en ocho suboctantes, y el proceso continua hasta que los subsuboctantes alcanzan un tamaño mínimo determinado. Árboles similares (N-trees) pueden, en principio, ser construidos para objetos N-dimensionales.

La técnica aquí descrita, debida a [Buyanovsky 02], es un método sencillo y rápido para comprimir árbol N-tree. Ha sido desarrollado como parte de un paquete de software para manejar datos médicos tales como NMR (resonancia magnética nuclear), ultrasonidos, y CT (tomografía computarizada o escáner). Tales datos se componen de un conjunto de imágenes médicas tridimensionales tomadas durante un corto período de tiempo y, por tanto, tienen cuatro dimensiones, con el tiempo como cuarta dimensión. Se pueden almacenar en un *hextree*, donde cada nodo es una hoja o tiene exactamente 16 hijos. Esta sección describe el aspecto de la compresión de datos del proyecto y utiliza un quadtree como ejemplo.

Imagine un quadtree con los píxeles de una imagen. A efectos de la compresión, asumimos que cualquier nodo A de un quadtree contiene (además de punteros a los cuatro hijos) dos valores: Los valores del píxel máximo y mínimo de los nodos del subárbol cuya raíz es A. Por ejemplo, el quadtree de la Figura 4.156 tiene píxeles con valores entre 0 y 255, por lo que la raíz del árbol contiene el par 0, 255. Sin compresión, los valores de pixel en este quadtree son números de 8 bits, pero el método aquí descrito hace posible escribir muchos de los valores en el flujo comprimido codificados con menos bits. El hijo del extremo izquierdo de la raíz, el nodo (1), es en sí mismo la raíz de un subárbol cuyos valores de píxeles están en el intervalo $[15, 255]$. Obviamente, este intervalo no puede ser más extenso que el intervalo $[0, 255]$ para el quadtree completo, por lo que los valores de píxel en este subárbol pueden, en principio, ser codificados en menos de ocho bits. El número de bits necesarios para codificar aritméticamente esos valores es $\log_2(255 - 15 + 1) \approx 7{,}91$; por consiguiente, hay una pequeña ganancia para los cuatro hijos inmediatos del nodo (1). Similarmente, los valores de píxel en el subárbol definido por el nodo (2) están en el intervalo $[101, 255]$ y, por lo tanto, requieren $\log_2(255 - 101 + 1) \approx 7{,}276$ bits cada uno cuando son codificados aritméticamente. Los valores de píxel en los subárboles definidos por el nodo (3) están en el estrecho intervalo $[172, 216]$ y, por lo tanto, requieren sólo $\log_2(216 - 172 + 1) \approx 5{,}49$ bits cada uno. Ésto se logra mediante la codificación relativa a la constante 172 de estos cuatro números en el *stream* comprimido. Los números codificados en realidad son 44, 0, 9, y 25.

Los dos nodos marcados con (7) tienen idénticos valores de píxel mínimo y máximo, lo que indica que estos nodos corresponden a cuadrantes uniformes de la imagen y, por lo tanto, son hojas del quadtree.

El punto siguiente ayuda a comprender cómo se codifican los valores de píxel. Cuando el decodificador comienza la decodificación de un subárbol, ya conoce los valores de los píxeles máximo y mínimo del subárbol porque ya ha decodificado el árbol padre de ese subárbol. Por ejemplo, cuando el decodificador comienza a decodificar el subárbol cuya raíz es (2), sabe que los valores de píxel máximo y mínimo del subárbol son 101 y 255, porque ya ha decodificado los cuatro hijos del nodo (1). Este conocimiento se utiliza para comprimir aún más un subárbol, tal como el (3), cuyos cuatro hijos son píxeles.

Un grupo de cuatro píxeles está codificado mediante la primera codificación de los valores de los dos primeros píxeles (i.e., del extremo izquierdo) usando el número de bits requerido. Para los cuatro

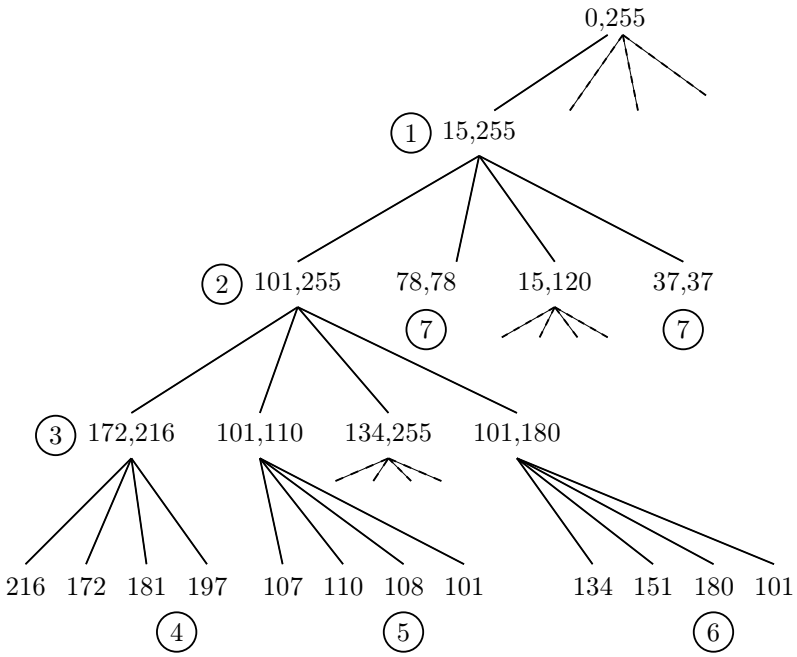


Figura 4.156: Un *quadtree* con píxeles en el intervalo [0, 255].

hijos del nodo (3), estos son 44 y 0, codificados en 5,49 bits cada uno. El resto de los otros dos valores de los píxeles pueden ser codificados con menos bits, y esto se hace distinguiendo tres casos como sigue:

- *Caso 1*: Los dos primeros valores de píxeles son el mínimo y el máximo de los cuatro. Una vez que el decodificador lee y decodifica esos dos valores, todo lo que sabe acerca de los siguientes dos valores es que están entre el mínimo y el máximo. Como resultado, los dos últimos valores tienen que ser codificados por el codificador con el número de bits requerido y no hay ganancia. En nuestro ejemplo, los dos primeros valores son 216 y 172, por lo que los siguientes dos valores, 181 y 197 (denotados (4)), tienen que ser escritos como los números 9 y 25. El codificador codifica aritméticamente los cuatro valores 44, 0, 9, y 25 en 5,49 bits cada uno. El decodificador lee y decodifica los dos primeros valores y descubre que son el mínimo y el máximo, por lo que sabe que este es el Caso 1 y quedan dos valores más por leer y decodificar.

- *Caso 2*: Uno de los valores de los dos primeros píxeles es el mínimo o el máximo. En este caso, uno de los otros dos valores es el máximo o el mínimo, y esto se indica mediante un indicador de 1 bit que es escrito por el codificador, en forma codificada, en el archivo comprimido, después del segundo píxel. Considere los cuatro valores de píxel 107, 110, 108, y 101. Ellos deben ser codificados en 3,32 bits cada uno [porque $\log_2(110 - 101 + 1) \approx 3{,}32$] relativo a 101. Los dos primeros valores codificados son 6 y 9. Después de que el decodificador lee y decodifica estos valores, sabe que el máximo (110) es uno de ellos, pero el mínimo (101) no lo es. El decodificador, por lo tanto, deduce que está en el Caso 2, y un indicador, seguido de un valor más, permanecen aún sin leer. Una vez leído el indicador, el decodificador sabe que el mínimo es el cuarto valor y, por lo tanto, el siguiente valor es el tercero de los cuatro píxeles. Ese valor (108, denotado (5)) se lee y decodifica. La compresión se incrementa debido a que el codificador no necesita codificar el píxel mínimo, 101. Sin el uso de un indicador, los cuatro valores requieren $4 \times 3{,}32 = 13{,}28$

bits. Con el indicador, el número de bits requeridos es de $2 \times 3{,}32 + 1 + 3{,}32 = 10{,}96$, un ahorro de 2,32 bits o el 17,5 % de 13,28.

- *Caso 3*: Ninguno de los valores de los dos primeros píxeles es el mínimo o el máximo. Por consiguiente, de los otros dos valores, uno es el máximo y el otro es el mínimo. El codificador escribe los dos primeros valores (codificados) en el archivo comprimido, seguido de un indicador de 1 bit que indica cuál de los dos valores es el máximo. La compresión mejora, ya que el codificador no tiene que escribir los valores reales de los píxeles mínimo y máximo. Después de la lectura y la decodificación de los dos primeros valores, el decodificador se entera de que no son el mínimo y el máximo, por lo que deduce que está en el Caso 3, y sólo permanece sin ser leído y decodificado un indicador. Los cuatro valores de píxeles 134, 151, 180, y 101 sirven como ejemplo. Los dos primeros valores se escriben (con relación a 101) en 6,32 bits cada uno. El 180 (denotado (6)) es el máximo, por lo que un indicador de 1 bit es codificado para indicar que el máximo es el tercer valor. En lugar de utilizar $4 \times 6{,}32 = 25{,}28$ bits, el codificador utiliza $2 \times 6{,}32 + 1 = 13{,}64$ bits.

El caso máx = mín +1 es especialmente interesante. En este caso $\log_2(\text{máx} - \text{mín} + 1) = 1$, por lo que cada valor de píxel se codifica, en promedio, en un bit. El algoritmo aquí descrito hace justamente éso y no trata este caso de ninguna manera especial. Sin embargo, en principio, es posible considerar este caso por separado, y codificar los cuatro valores de píxel en menos de cuatro bits (en 3,8073 bits, para ser precisos). Aquí se muestra cómo.

Denotamos min y max mediante n y x, respectivamente, y exploramos todas las posibles combinaciones de n y x.

Si uno de los dos primeros valores de píxel es n y el otro es x, entonces tanto el codificador como el decodificador sabe que este es el caso 1 anterior. No se utiliza ningún indicador. Observe que los dos píxeles restantes pueden ser uno de los cuatro pares (n, n), (n, x), (x, n), y (x, x), por lo que un indicador de 1 bit no sería suficiente para distinguirlos. Por lo tanto, el codificador codifica cada uno de los cuatro valores de píxel con un bit, haciendo un total de cuatro bits.

Si los dos primeros valores de píxel son ambos n, o ambos x, entonces este es el caso 2 anterior. Puede haber dos subcasos como sigue:

- *Caso 2.1*: Los dos primeros valores son n y n. Éstos pueden ir seguidos por uno de los tres pares (n, x), (x, n), o (x, x).
- *Caso 2.2*: Los dos primeros valores son x y x. Éstos pueden ir seguidos por uno de los tres pares, (n, x), (x, n), o (n, n).

Por consiguiente, una vez que el decodificador ha leído los dos primeros valores, tiene que identificar una de las tres alternativas. Para lograr ésto, el codificador puede, en principio, seguir los dos primeros valores (que se codifican en un bit cada uno) con un código especial que indica una de las tres alternativas. Tal código puede, en principio, ser codificado en $-\log_2 3 \approx 1{,}585$ bits. El número total de bits necesarios, en principio, para codificar el Caso 2 es, por lo tanto, $2 + 1{,}585 = 3{,}585$.

En el Caso 3, ninguno de los dos primeros valores de píxel es n o x. Ésto, por supuesto, no puede ocurrir cuando máx = mín +1, ya que en este caso cada uno de los cuatro valores es, o bien n, o bien x. El Caso 3 es, por lo tanto, imposible.

Por consiguiente, concluimos que en el caso especial máx = mín +1, los cuatro valores de píxel pueden ser codificados, ya sea en cuatro bits, ya sea en 3,585 bits. En promedio, los cuatro valores se pueden codificar en menos de cuatro bits, ¡que parece mágico! La explicación es que nunca ocurren dos de las 16 combinaciones posibles. Podemos pensar en los cuatro valores de píxel como una tupla de 4 donde los cuatro elementos son, o bien n, o bien x. En general, existen 16 tuplas de 4, pero los dos casos (n, n, n, n) y (x, x, x, x) no pueden suceder, lo que deja tan sólo 14 tuplas de 4 para ser codificadas.

Codificar una de las 14 tuplas de 4 binarias, requiere $-\log_2(14) \approx 3{,}8073$ bits. Dado que las hojas de un quadtree tienden a ocupar mucho espacio y puesto que el caso máx = mín +1 es altamente probable en las imágenes, se estima que el tratamiento especial que aquí se describe puede mejorar la compresión en un 2−3 %. El pseudocódigo listado aquí, sin embargo, no trata el caso máx = mín +1 de ninguna manera especial.

El pseudocódigo muestra cómo los valores de píxel y los indicadores se envían a un procedimiento `encode(value, min, max)` para ser codificados aritméticamente. Este procedimiento codifica su primer parámetro en $\log_2(\text{máx} - \text{mín} + 1)$ bits, en promedio. Un indicador se codifica estableciendo mín = 0, máx = 1, y un valor 0 ó 1. El bit indicador está, por tanto, codificado en $\log_2(1 - 0 + 1) = 1$ bit, en promedio. (Ésto significa que algunos de los indicadores se codifican en más de un bit, pero otros se codifican en tan poco como ¡cero bits! En general, el número de bits consumidos en codificar un indicador se distribuye normalmente alrededor de 1.)

El método se ilustra mediante el siguiente código al estilo de C:

```
/* Definiciones:
encode( value , min , max ); - función de codificación del valor
  (output bits stream),
  min<=value<=max, la longitud del código es Log2(max-min+1) bits;
Log2(N) - nivel de profundidad del píxel del quadtree.
struct knot_descriptor
{
int min,max; //min,max del subplano completo
int pix ; // valor del píxel (en caso de nivel de píxel)
int depth ; // profundidad de nudo (knot).
knot_descriptor *square[4];//subplanos hijos
} ;
Compact_quadtree (...) - procedimiento recursivo para compresión de quadtree.
*/
Compact_quadtree (knot_descriptor *knot, int min, int max)
{
encode( knot->min , min , max ) ;
encode( knot->max , min , max ) ;
if ( knot->min == knot->max ) return ;
min = knot->min ;
max= knot->max ;
if ( knot->depth < Log2(N) )
{
Compact_quadtree(knot->square[ 0 ],min,max) ;
Compact_quadtree(knot->square[ 1 ],min,max) ;
Compact_quadtree(knot->square[ 2 ],min,max) ;
Compact_quadtree(knot->square[ 3 ],min,max) ;
}
else
{ // knot->depth == Log2(N) y nivel de píxel
int slc = 0 ;
encode( (knot->square[ 0 ])->pix , min , max );
if ((knot->square[ 0 ])->pix == min) slc=1;
else if ((knot->square[ 0 ])->pix==max) slc=2;
encode( (knot->square[ 1 ])->pix , min , max );
if ((knot->square[ 1 ])->pix == min) slc |= 1;
else if ((knot->square[ 1 ])->pix==max) slc |= 2;
switch( slc )
{
case 0:
```

```
encode(((knot->square[2])->pix==max),0,1);
return ;
case 1:
if ((knot->square[2])->pix==max)
{
encode(1,0,1);
encode((knot->square[ 3 ])->pix,min,max);
}
else
{
encode(0,0,1);
encode((knot->square[ 2 ])->pix,min,max);
}
return ;
case 2:
if ((knot->square[2])->pix==min)
{
encode(1,0,1);
encode((knot->square[ 3 ])->pix,min,max);
}
else
{
encode(0,0,1);
encode((knot->square[ 2 ])->pix,min,max);
}
return ;
case 3:
encode((knot->square[ 2 ])->pix,min,max);
encode((knot->square[ 3 ])->pix,min,max);
}
}
}
```

Una mejora

La siguiente mejora para el método de esta sección me fue comunicada por su autor, Stephan Wolf, quien también escribió estos párrafos.

Me sorprendió el hecho de que se sugiriera la codificación aritmética para la función `encode()`. La codificación aritmética es aparentemente el mejor método si los símbolos individuales tienen distintas probabilidades. Pero en este caso, el documento parece suponer probabilidades iguales para todos los valores de píxeles individuales en un rango de codificación particular.

Por lo tanto, se me ocurrió un algoritmo muy sencillo que utiliza exactamente $\log_2 n$ bits para codificar un valor para un rango de valores relativos a cero determinado.

Pensé esperar hasta recibir su libro y ver si describe este algoritmo, i.e., si se trata de una técnica ya conocida.

Pero no pude encontrar ninguna pista en su libro o en cualquier otro lugar (por ejemplo, Internet) sobre el algoritmo que ideé.

Así que permítame presentarle este algoritmo para revisión. También quiero añadir que las siguientes ideas son todas mías:

Dado un rango `r` (relativo a cero) de valores `v` igualmente distribuidos, i.e., todos los valores tienen la misma probabilidad. En este documento, denoto un par rango/valor como una tupla $(\mathtt{r}, \mathtt{v})$. Tanto el codificador como el decodificador necesitan conocer ya el rango `r` en cada etapa de codifica-

ción/decodificación de un valor individual v.

Como ejemplo, utilizo los siguientes pares rango/valor: $(9, 2)$, $(199, 73)$, $(123, 89)$ y $(50, 43)$. El número de bits necesarios teórico para codificar cada valor individual es $\log_2(r+1)$ que se traduce (en valores redondeados usando tres dígitos decimales) a $\log_2(9+1) = 3{,}32$, $\log_2(199+1) = 7{,}64$, $\log_2(123+1) = 6{,}95$, y $\log_2(50+1) = 5{,}67$. En consecuencia, se requiere un máximo total de 23,6 bits para codificar todos los valores del ejemplo. Un codificador perfecto, por tanto, usaría como mucho 24 bits.

Propongo el algoritmo $\mathtt{e}[\mathtt{n}+1] = \mathtt{e}[\mathtt{n}] * (\mathtt{r}[\mathtt{n}]+1) + \mathtt{v}[\mathtt{n}]$ para codificar los valores de los pares rango/valor anteriores. Los resultados son:

$$\begin{aligned}
\mathtt{e}[1] &= 0 \times (9+1) + 2 = 2,\\
\mathtt{e}[2] &= 2 \times (199+1) + 73 = 473,\\
\mathtt{e}[3] &= 473 \times (123+1) + 89 = 58\,741,\\
\mathtt{e}[4] &= 58\,741 \times (50+1) + 43 = 2\,995\,834.
\end{aligned}$$

Este valor puede expresarse en 22 bits como 10 1101 1011 0110 0111 1010.

La decodificación es muy similar (observe que % denota la operación módulo, i.e., el resto de la división entera)

```
v[n+1] = d[n] % r
d[n+1] = d[n] / r
v[1] = 2,995,834 % (50+1) = 43
d[1] = 2,995,834 / (50+1) = 58,741
v[2] = 58,741 % (123+1) = 89
d[2] = 58,741 / (123+1) = 473
v[3] = 473 % (199+1) = 73
d[3] = 473 / (199+1) = 2
v[4] = 2 % (9+1) = 2
d[4] = 2 / (9+1) = 0
```

Observe que cualquier aplicación de este algoritmo de codificación/decodificación requiere aritmética de enteros largos, similar a la que normalmente está disponible en la mayoría de las bibliotecas de criptografía.

4.30.5. Compresión prefija (mediante prefijos)

La compresión prefija es una variante de los quadtrees, propuesta en [Anedda y Felician 88] (véase también la Sección 8.5.5). Comenzamos con una imagen de $2^n \times 2^n$. Cada cuadrante en el quadtree de esta imagen es identificado con un número de dos bits: 0, 1, 2, ó 3. Cada subcuadrante es denotado con un número de dos dígitos (i.e., 4 bits), y cada subsubcuadrante recibe un número de 3 dígitos. A medida que los cuadrantes se hacen más pequeños, sus números se hacen más largos. Cuando este esquema de numeración se lleva a píxeles individuales, el número de un píxel resulta ser de n dígitos, ó $2n$ bits, de longitud. La compresión prefija está diseñada para imágenes binivel que contienen texto o diagramas donde el número de píxeles negros es relativamente pequeño. No es adecuada para imágenes en escala de grises, en color, o para cualquier otra que contenga muchos píxeles negros, tal como una pintura. El método se explica mejor con un ejemplo. La Figura 4.157 muestra la numeración de los píxeles en una imagen de 8×8 (i.e., $n = 3$) y también una sencilla imagen de 8×8 formada por 18 píxeles negros. Cada número de píxel consta de tres dígitos, y están en el rango de 000 a 333.

000	001	010	011	100	101	110	111
002	003	012	013	102	103	112	113
020	021	030	031	120	121	130	131
022	023	032	033	122	123	132	133
200	201	210	211	300	301	310	311
202	203	212	213	302	303	312	313
220	221	230	231	320	321	330	331
222	223	232	233	322	323	332	333

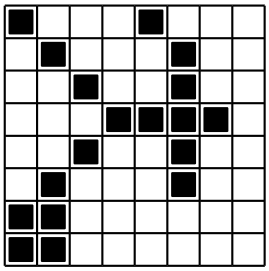


Figura 4.157: Ejemplo de compresión prefija.

El primer paso es utilizar los métodos de quadtree para calcular los números de tres dígitos, que identifican los 18 píxeles negros. Ellos son: 000 101 003 103 030 121 033 122 123 132 210 301 203 303 220 221 222 223.

El siguiente paso es seleccionar un valor P de prefijo . Seleccionamos $P = 2$, una elección que se justifica a continuación. El código de un píxel se divide ahora en P dígitos prefijo seguidos por $3 - P$ dígitos sufijo. El último paso se realiza sobre la secuencia de píxeles negros y se seleccionan todos los pixeles con el mismo prefijo. El primer prefijo es 00, por lo que todos los píxeles que comienzan por 00 son seleccionados. Ellos son: 000 y 003. Éstos se eliminan de la secuencia original y se comprimen escribiendo el token 00|1|0|3 en la secuencia de salida. La primera parte de este token es un prefijo (00), la segunda parte es un recuento (1), y el resto son los sufijos de los dos píxeles que tienen como prefijo 00. Observe que un recuento de uno implica dos píxeles. El recuento es siempre uno menos que el número de píxeles que están siendo contados. Ahora permanecen dieciséis píxeles en la secuencia original, y el primero de ellos tiene prefijo 10. Los dos píxeles con este prefijo son eliminados, y se comprimen escribiendo el token 10|1|1|3 en la secuencia de salida. Ésto continúa hasta que la secuencia original esté vacía. El resultado final es la cadena de 9 tokens:

$$00|1|0|3, 10|1|1|3, 03|1|0|3, 12|2|1|2|3, 13|0|2, 21|0|0, 30|1|1|3, 20|0|3, 22|3|0|1|2|3$$

(sin las comas). Tal cadena se puede decodificar de forma única, ya que cada segmento comienza con un prefijo de dos dígitos, seguido por un recuento c de un dígito, seguido por $c+1$ sufijos de un dígito.

En general, el prefijo tiene P dígitos de longitud, y el recuento y cada sufijo tienen $n - P$ dígitos cada uno. En consecuencia, el número máximo de sufijos en un segmento es 4^{n-P}. El tamaño máximo de un segmento consta, por tanto, de $P + (n - P) + 4^{n-P}(n - P)$ dígitos. Cada segmento corresponde a un prefijo diferente. Un prefijo está formado por P dígitos, cada uno entre 0 y 3, por lo que el número máximo de segmentos es 4^P. La cadena completa comprimida, por tanto, ocupa como máximo:

$$4^P \left[P + (n - P) + 4^{n-P}(n - P)\right] = n \cdot 4^P + 4^n (n - P)$$

dígitos. Para encontrar el valor óptimo de P derivamos[17] la expresión anterior con respecto a P,

$$\frac{d}{dP}\left[n \cdot 4^P + 4^n (n - P)\right] = n \cdot 4^P \ln 4 - 4^n,$$

y al igualar la derivada a cero. La solución es:

$$4^p = \frac{4^n}{n \cdot \ln 4}, \quad \text{o} \quad P = \log_4 \left[\frac{4^n}{n \cdot \ln 4}\right] = \frac{1}{2} \log_2 \left[\frac{4^n}{n \cdot \ln 4}\right].$$

[17]Recordemos las reglas de derivación utilizadas: $(f \pm g)' = f' \pm g'$; $(k \cdot f)' = k \cdot f'$; $(k)' = 0$; $(a^g)' = a^g g' \ln a$; siendo f y g funciones, y a y k constantes (la derivada está representada mediante $'$). Como la derivada es con respecto a P, $\left(a^P\right)' = a^P \ln a$.

Para $n = 3$ ésto produce:

$$P \approx \frac{1}{2}\log_2\left[\frac{4^3}{3 \times 1{,}386}\right] = \frac{\log_2 15{,}388}{2} = \frac{3{,}944}{2} = 1{,}97.$$

Ésta es la razón por la cual se seleccionó anteriormente $P = 2$.

Una desventaja de este método es que a algunos píxeles se les asignarán números con prefijos diferentes a pesar de que son vecinos cercanos. Ésto sucede cuando están ubicados en distintos cuadrantes. Un ejemplo son los píxeles 123 y 301 de la Figura 4.157.

Mejora: El campo de recuento, `count`, se fijó arbitrariamente a un dígito (dos bits). El recuento máximo es, por lo tanto, 3 ($= 11_2$), i.e., cuatro píxeles. Es posible tener un campo de recuento de tamaño variable que contenga un código de tamaño variable, tal como el código unario de la Sección 2.3.1. De esta forma, un único token podría comprimir cualquier número de pixeles.

4.31. Quadrisection (Cuatrisección)

Un quadtree (árbol cuaternario) explota la redundancia en una imagen examinando cuadrantes cada vez más pequeños, en busca de áreas uniformes. El método *quadrisection* [Kieffer et al. 96a,b 11] está relacionado con los quadtrees, ya que utiliza el principio de quadtree para dividir una imagen en subimágenes. Sin embargo, quadrisection no divide la imagen en cuatro partes, sino que más bien forma de nuevo la imagen en pasos aumentando el número de sus filas y disminuyendo el número de sus columnas. El método es sin pérdidas. Se comporta bien con imágenes binivel y aquí se ilustra para tales imágenes, pero también se puede aplicar a imágenes en escala de grises (y, por lo tanto, en color).

El método asume que la imagen original es una matriz cuadrada M_0 de $2^k \times 2^k$, y que forma de nuevo M_0 en una secuencia de matrices M_1, M_2, ..., M_{k+1} con cada vez menos columnas. Estas matrices tienen naturalmente cada vez más filas, y el método quadrisection logra la compresión buscando y eliminando filas duplicadas. Cuantas más filas y menos columnas tenga una matriz, mejor será la suerte de tener filas duplicadas. El resultado final es una matriz M_{k+1} con una columna y no más de 64 filas. Esta matriz se trata como una cadena corta y se escribe al final del *stream* comprimido, para ayudar en la decodificación y la recuperación de la imagen M_0 original. El *stream* comprimido debe contener también información sobre cómo reconstruir cada matriz M_{j-1} a partir de su sucesor M_j. Esta información está en forma de un *vector indicador* denotado por I_{j-1}. Por consiguiente, la salida está formada por todos los vectores I_j (cada uno codificado aritméticamente en una cadena w_j), seguido por los bits de M_{k+1}^T (la transpuesta de la última matriz, de una sola columna, sin comprimir). Esta cadena es precedida por un prefijo que indica el valor de k, los tamaños de todos los w_j's, y el tamaño de M_{k+1}^T.

Aquí está el proceso de codificación un poco más detallado. Asumimos que la imagen original es una matriz cuadrada M_0 de $2^k \times 2^k$ de 4^k píxeles, cada uno de un solo bit. El codificador utiliza una operación llamada *proyección* (discutida más adelante) para la construcción de una secuencia de matrices M_1, M_2, hasta M_{k+1}, de tal manera que M_j tiene 4^{k-j+1} columnas. Esto implica que M_1 tiene $4^{k-1+1} = 4^k$ columnas y, por lo tanto, sólo una fila. La matriz M_2 tiene 4^{k-1} columnas y, por lo tanto, cuatro filas. La matriz M_3 tiene 4^{k-2} columnas, pero puede tener menos de ocho filas, ya que las filas duplicadas son eliminadas de M_2 antes de construir M_3. El número de filas de M_3 es cuatro veces el número de filas *distintas* de M_2. El vector indicador I_2 se construye al mismo tiempo que M_3 para indicar aquellas filas de M_2 que son duplicados. El tamaño de I_2 es igual al número de filas de M_2, y los elementos de I_2 son enteros no negativos. El decodificador utiliza I_2 para reconstruir M_2 a partir de M_3.

Todos los vectores indicador I_j son necesarios para el decodificador. Dado que el tamaño de I_j es igual al número de filas de la matriz M_j, y puesto que estas matrices tienen cada vez más filas, los tamaños combinados de todos los vectores indicador I_j pueden ser grandes. Sin embargo, la mayor

parte de los elementos de un vector indicador típico I_j son ceros, por lo que estos vectores pueden ser altamente comprimidos mediante una codificación aritmética. Además, los primeros pocos vectores indicadores son normalmente todo ceros, de manera que todos pueden ser reemplazados por un número que indique cuántos de ellos hay.

◇ **Ejercicio 4.56 (sol. en pág. 1090):** A pesar de que aún no hemos descrito los detalles del método (en concreto, la operación de proyección), es posible deducir lógicamente (o adivinar) la respuesta a este ejercicio. La pregunta es: Sabemos que las imágenes con poca o ninguna correlación no se comprimen. Sin embargo, el tamaño de la última matriz, M_{k+1}, es pequeño (64 bits o menos) independientemente de la imagen que se comprime. ¿Puede quadrisection comprimir imágenes de alguna manera que otros métodos no pueden?

El decodificador comienza mediante la decodificación del prefijo, para obtener los tamaños de todas las cadenas comprimidas w_j. A continuación, descomprime cada w_j para obtener un vector indicador I_j. El decodificador sabe cuántos vectores indicadores existen, ya que j va de 1 a k. Después de decodificar todos los vectores indicadores, el descodificador lee el resto del *stream* comprimido y considera los bits de M_{k+1}. El decodificador entonces utiliza los vectores indicadores para reconstruir las matrices M_k, M_{k-1}, y así sucesivamente bajando hasta la M_0.

La operación de proyección es el corazón de quadrisection. Se describe aquí en tres pasos:

- *Paso 1*: Un vector indicador I para una matriz M con r filas contiene r componentes, uno para cada fila de M. Cada fila distinta de M tiene un componente cero en I, y cada fila duplicada tiene una componente positiva. Si, por ejemplo, la fila 8 es un duplicado de la quinta fila distinta, entonces el octavo componente de I será 5. Observe que la quinta fila distinta no es necesariamente la fila 5. Como ejemplo, el vector indicador para la matriz M de la Ecuación (4.53) es $I = (0, 0, 0, 3, 0, 4, 3, 1)$:

$$M_3 = \begin{bmatrix} 1001 \\ 1101 \\ 1011 \\ 1011 \\ 0000 \\ 0000 \\ 1011 \\ 1001 \end{bmatrix}. \tag{4.53}$$

- *Paso 2*: Dado un vector fila v de longitud m, donde m es una potencia de 2, realizamos lo siguiente: (1) Lo dividimos en $\sqrt{m}$ segmentos de longitud $\sqrt{m}$ cada uno. (2) Alojamos los segmentos en una matriz de tamaño $\sqrt{m} \times \sqrt{m}$. (3) la dividimos en cuatro cuadrantes; cada uno es una matriz de tamaño $\sqrt{m}/2 \times \sqrt{m}/2$. (4) Identificamos los cuadrantes 1, 2, 3, y 4 de acuerdo con $\left(\begin{smallmatrix} 1 & 2 \\ 3 & 4 \end{smallmatrix}\right)$. (5) Desplegamos cada uno en un vector v de longitud $\sqrt{m}$. Como ejemplo, consideremos el vector:

$$v = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15),$$

de longitud $16 = 2^4$. En primer lugar, se convierte en la matriz de 4×4:

$$M = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{pmatrix}.$$

El particionamiento de M produce las cuatro matrices de 2×2:

$$M_1 = \begin{pmatrix} 0 & 1 \\ 4 & 5 \end{pmatrix}, \quad M_2 = \begin{pmatrix} 2 & 3 \\ 6 & 7 \end{pmatrix}, \quad M_3 = \begin{pmatrix} 8 & 9 \\ 12 & 13 \end{pmatrix}, \quad \text{y } M_4 = \begin{pmatrix} 10 & 11 \\ 14 & 15 \end{pmatrix}.$$

Cada uno se despliega, para producir los cuatro vectores:

$$v_1 = (0, 1, 4, 5)\,, \quad v_2 = (2, 3, 6, 7)\,, \quad v_3 = (8, 9, 12, 13)\,, \quad \text{y } v_4 = (10, 11, 14, 15)\,.$$

Este paso ilustra la relación entre quadrisection y los quadtrees.

- *Paso 3*: Éste por fin describe la operación de proyección. Dada una matriz M de $n \times m$, donde m (el número de columnas) es una potencia de 2, construimos en primer lugar un vector indicador I para ella. El vector I tendrá n componentes, para las n filas de M. Si M tiene r filas distintas, el vector I tendrá r ceros correspondientes a estas filas. Construimos la matriz de proyección M' con $4r$ filas y $m/4$ columnas con los siguientes pasos: (3.1) Ignoramos todas las filas duplicadas de M (es decir, todas las filas correspondientes a los elementos distintos de cero de I). (3.2) Para cada uno de las r filas distintas restantes de M construimos cuatro vectores v_i como se muestra en el Paso 2 anterior, y los convertirmos en las cuatro filas siguientes de M'. Usamos la notación:

$$M \xrightarrow{I} M',$$

para indicar esta proyección.

Ejemplo: Dada la matriz M_0 de 16×16 de la Figura 4.158 concatenamos sus filas para construir una matriz M_1 con una fila y 256 columnas. Éste es siempre nuestro punto de partida, independientemente de si las filas de M_0 son distintas o no. Puesto que M_1 tiene sólo una fila, su vector indicador es $I_1 = (0)$.

La matriz M_2 se proyecta fácilmente desde M_1. Tiene cuatro filas y 64 columnas:

$$M_2 = \begin{bmatrix} 0000000000000001000000110000001100000011000011110011111101111111 \\ 1000000010000000110000000111000011100000110000001100000011000000 \\ 0111111111111111111111111111101111111001111111010111110101111101 \\ 1000000010000000000000001000000010000000100000001000000011001111 \end{bmatrix}.$$

Las cuatro filas de M_2 son distintas, por lo que su vector indicador es $I_2 = (0, 0, 0, 0)$. Observe cómo cada fila de M_2 es una submatriz de 8×8 de M_0. La fila superior, por ejemplo, es la esquina superior izquierda de 8×8 de M_0.

Para proyectar M_3 desde M_2 se efectúa el Paso 2 de arriba en cada fila de M_2, convirtiéndola de una matriz de 1×64 a una de 4×16. Por consiguiente, la matriz M_3 se compone de cuatro partes, cada una de 4×16, por lo que su tamaño es de 16×16:

$$M_3 = \left[\begin{array}{c} 0000000000000000 \\ 0000000100110011 \\ 0000000000110111 \\ 0011111111111111 \\ \hline 1000100011000111 \\ 0000000000000000 \\ 1110110011001100 \\ 0000000000000000 \\ \hline 0111111111111111 \\ 1111111111111011 \\ 1111111101110111 \\ 1001110111011101 \\ \hline 1000100000001000 \\ 0000000000000000 \\ 1000100010001100 \\ 0000000000001111 \end{array}\right].$$

Observe cómo cada fila de M_3 es una submatriz de 4×4 de M_0. La fila superior, por ejemplo, es el bloque de 4×4 de la esquina superior izquierda de M_0, y la segunda fila es el segundo bloque (hacia

la derecha) de 4×4 desde la esquina superior izquierda. Examinando las 16 filas de M_3 se obtiene el vector indicador $I_3 = (0,0,0,0,0,1,0,1,0,0,0,0,0,1,0,0)$ (i.e., las filas 6, 8, y 14 son duplicados de la fila 1). Así es cómo la siguiente proyección crea algo de compresión. La proyección de M_3 a M_4 se efectúa ignorando las filas 6, 8, y 14. M_4 [Ecuación (4.54)], por lo tanto, tiene $4 \times 13 = 52$ filas y cuatro columnas (una cuarta parte del número de M_3). Por consiguiente, consta de $52 \times 4 = 208$ elementos en vez de 256.

$$M_4 = \begin{bmatrix} 0000 \\ 0000 \\ 0000 \\ 0000 \\ 0000 \\ 0001 \\ 0000 \\ 1111 \\ 0000 \\ 0000 \\ 0001 \\ 1111 \\ 0011 \\ 1111 \\ 1111 \\ 1111 \\ \hline 1010 \\ 0000 \\ 1101 \\ 0011 \\ 1111 \\ 1000 \\ 1111 \\ 0000 \\ \hline 0111 \\ 1111 \\ 1111 \\ 1111 \\ 1111 \\ 1111 \\ 1111 \\ 1110 \\ 1111 \\ 1111 \\ 1111 \\ 0101 \\ 1111 \\ 1011 \\ 0101 \\ 1111 \\ 0101 \\ \hline 1010 \\ 0000 \\ 0010 \\ 0000 \\ 1010 \\ 0000 \\ 1011 \\ 0000 \\ 0000 \\ 0000 \\ 0011 \\ 0011 \end{bmatrix} \tag{4.54}$$

Es evidente que con filas tan cortas, M_4 debe tener muchas filas duplicadas. Un análisis indica que

0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0
0	0	0	0	0	0	1	1	0	1	1	1	0	0	0	0
0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0
0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0
0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0
0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0
0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
1	1	1	1	1	0	1	1	1	0	0	0	0	0	0	0
1	1	1	1	1	0	0	1	1	0	0	0	0	0	0	0
1	1	1	1	1	1	0	1	1	0	0	0	0	0	0	0
0	1	1	1	1	1	0	1	1	0	0	0	0	0	0	0
0	1	1	1	1	1	0	1	1	1	0	0	1	1	1	1

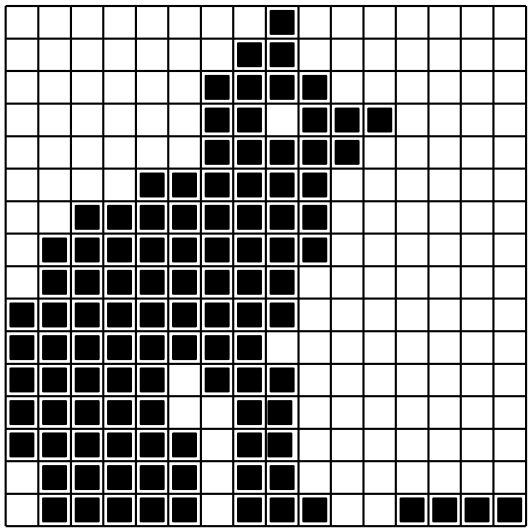


Figura 4.158: Una matriz M_0 de 16×16.

sólo 12 filas son distintas, y produce el vector I_4:

$$I_4 = (0,1,1,1,1,0,1,0,1,1,2,3,0,3,3,3 \mid 0,1,0,4,3,0,3,1 \mid \\ 0,3,3,3,3,3,0,3,3,3,0,3,0,10,3,10 \mid 5,1,0,1,5,1,11,1,1,1,4,4).$$

La proyección de M_4 para obtener M_5 se realiza como antes. La matriz M_5 tiene $4 \times 12 = 48$ filas y sólo una columna (un cuarto del número de columnas de M_4). Esto es, por lo tanto, la matriz final M_{k+1}, cuya transpuesta será la última parte del *stream* comprimido.

◇ **Ejercicio 4.57 (sol. en pág. 1091):** Calcúlese la matriz M_5.

El flujo de datos comprimido se compone, por lo tanto, de un prefijo, seguido por el valor de k ($= 4$), seguido de I_1, I_2, I_3, e I_4, codificados aritméticamente, seguidos por los 48 bits de M_5^T. No hay necesidad de codificar los últimos bits, ya que, a lo sumo, puede haber 64 de ellos (véase el ejercicio 4.58). Además, puesto que I_1 e I_2 son ceros, no hay necesidad de escribirlos en la salida. Todo lo que el codificador tiene que escribir en su lugar es el número 3 (codificado) para informar de que el primer vector indicador incluido en la salida es I_3.

El prefijo consiste en: 4 (k), 3 (índice del primer vector indicador distinto de cero), y las longitudes codificadas de los vectores indicadores I_3 e I_4. Después de esta decodificación, el decodificador puede decodificar los dos vectores indicadores, leer la secuencia de datos comprimidos restante como M_5^T, y luego utilizar todos estos datos para reconstruir M_4, M_3, M_2, M_1, y M_0.

◇ **Ejercicio 4.58 (sol. en pág. 1091):** Muéstrese por qué M_5 no puede tener más de 64 elementos.

Extensiones: Quadrisection se utiliza para comprimir datos bidimensionales y, por lo tanto, puede ser considerado el segundo método de una sucesión de tres métodos de compresión, el primero y tercero de los cuales son *bisection* (bisección) y *octasection* (octasección). A continuación, una breve descripción de estos dos métodos.

Bisection es una extensión (o más bien una reducción) de quadrisection para el caso donde los datos son unidimensionales (ejemplos típicos son: muestras de audio, una cadena binaria, o texto). Suponemos que los datos consisten en una cadena L_0 de 2^k ítems de datos, donde un ítem puede ser

un simple bit, un código ASCII, una muestra de audio, o cualquier otra cosa, pero todos los ítems tienen el mismo tamaño.

El codificador itera k veces, variando j desde 1 hasta k. En la iteración j, crea una lista L_j mediante la bisección de los elementos de la lista precedente L_{j-1}. Construye un vector indicador I_J para L_j. Los elementos duplicados de L_j son eliminados (aquí es donde obtenemos la compresión).

Cada uno de los dos elementos de la primera lista construida L_1 es, por lo tanto, un bloque de 2^{k-1} ítems de datos (la mitad del número de ítems de datos en L_0). El vector indicador I_1 también tiene dos elementos. Cada elemento de L_2 es un bloque de 2^{k-2} ítems de datos, pero el número de elementos de L_2 no es necesariamente cuatro. Puede ser más pequeño dependiendo de cuántos elementos distintos tiene L_1. En el último paso, donde $j = k$, se crea la lista L_k, donde cada ítem es un bloque de tamaño $2^{k-k} = 1$. Por consiguiente, cada elemento de L_k es uno de los ítems de datos originales de L_0. Aquél no es necesario para la construcción del vector indicador I_k (el último vector indicador generado es I_{k-1}).

El flujo de datos comprimido consiste en: k, seguido por los vectores indicadores I_1, I_2 a I_{k-1} (comprimidos), seguidos de L_k (tal cual es, sin codificar). Como los primeros pocos vectores indicadores tienden a ser todo ceros, pueden ser reemplazados por un único número que indica el índice del primer vector indicador distinto de cero.

Ejemplo: La cadena de 32 elementos L_0, donde cada elemento de datos es un único bit:

$$L_0 = (0,1,1,1,0,0,1,0,1,1,0,0,0,0,1,0,0,1,1,0,1,1,1,1,0,1,0,1,1,0,1,0).$$

Las dos mitades de L_0 son distintas, por lo que L_1 está formado por dos elementos:

$$L_1 = (0111001011000010, 0110111101011010),$$

y el primer vector indicador es $I_1 = (0,0)$. Los dos elementos de L_1 son distintos, por lo que L_2 tiene cuatro elementos:

$$L_2 = (01110010, 11000010, 01101111, 01011010),$$

y el segundo vector indicador es $I_2 = (0,0,0,0)$. Los cuatro elementos de L_2 son diferentes, por lo que L_3 tiene ocho elementos:

$$L_3 = (0111, 0010, 1100, 0010, 0110, 1111, 0101, 1010)$$

y el tercer vector indicador es $I_3 = (0,0,0,2,0,0,0,0)$. Siete elementos de L_3 son distintos, por lo que L_4 tiene 14 elementos:

$$L_4 = (01, 11, 00, 10, 11, 00, 01, 10, 11, 11, 01, 01, 10, 10),$$

y el cuarto vector indicador es $I_4 = (0,0,0,0,2,3,1,4,2,2,1,1,4,4)$. Sólo cuatro elementos de L_4 son distintos, por lo que L_5 tiene ocho elementos: $L_5 = (0,1,1,1,0,0,1,0)$.

La salida consiste, por tanto en: $k = 5$, los vectores indicadores $I_1 = (0,0)$, $I_2 = (0,0,0,0)$, $I_3 = (0,0,0,2,0,0,0,0)$, e $I_4 = (0,0,0,0,2,3,1,4,2,2,1,1,4,4)$ (codificados), seguidos por $L_5 = (0,1,1,1,0,0,1,0)$. Puesto que el primer vector indicador distinto de cero es I_3, podemos omitir tanto I_1 como I_2 y reemplazarlos con el entero 3.

◇ **Ejercicio 4.59 (sol. en pág. 1091):** Descríbanse las operaciones del decodificador para este ejemplo.

◇ **Ejercicio 4.60 (sol. en pág. 1092):** Utilícese la bisección para codificar la cadena de 32 bits:

$$L_0 = (0101010101010101\ 1010101010101010).$$

La discusión anterior muestra que, si los datos originales a ser comprimidos son una cadena de bits de longitud 2^k, entonces L_{k-1} es una lista de pares de bits. L_{k-1}, por lo tanto, puede tener un máximo de cuatro elementos distintos, por lo que L_k, cuyos elementos son simples bits, puede tener como mucho ocho elementos. Ésto, por supuesto, no significa que cualquier cadena binaria L_0 se pueda comprimir en ocho bits. El lector debe tener en cuenta que el flujo comprimido debe incluir también los vectores indicadores distintos de cero. Si los elementos de L_0 no son bits, entonces L_k podría, en principio, ser tan largo como L_0.

Ejemplo: Una cadena origen $L_0 = (a_1, a_2, \ldots, a_{32})$, donde los a_i's son ítems de datos distintos, tales como caracteres ASCII. L_1 consta de dos elementos:

$$L_1 = (a_1 a_2 \ldots a_{16}, a_{17} a_{18} \ldots a_{32}) ,$$

y el primer vector indicador es $I_1 = (0, 0)$. Los dos elementos de L_1 son distintos, por lo que L_2 está formado por cuatro elementos:

$$L_2 = (a_1 a_2 \ldots a_8, a_9 a_{10} \ldots a_{16}, a_{17} a_{18} \ldots a_{24}, a_{25} a_{26} \ldots a_{32}) ,$$

y el segundo vector indicador es $I_2 = (0, 0, 0, 0)$. Los cuatro elementos de L_2 son distintos, por tanto:

$$L_3 = (a_1 a_2 a_3 a_4, a_5 a_6 a_7 a_8, a_9 a_{10} a_{11} a_{12}, \ldots, a_{29} a_{30} a_{31} a_{32}) .$$

Continuando de esta manera, es fácil ver que todos los vectores indicadores serán cero, y L_5 tiene los mismos 32 elementos que L_0. El resultado será que no se produce compresión alguna.

Si la longitud L de los datos originales no es una potencia de dos, todavía podemos utilizar la bisección teniendo en cuenta lo siguiente: Existe algún entero k tal que $2^{k-1} < L < 2^k$. Si L está cercano a 2^k, podemos añadir $d = 2^k - L$ ceros a la cadena original L_0, comprimirla por bisección, y escribir d en el *stream* comprimido, por lo que el decodificador puede eliminar los d ceros. Si L está cercano a 2^{k-1} dividimos L_0 en una cadena L_0^1 con los primeros 2^{k-1} ítems, y la cadena L_0^2 con los ítems restantes. La primera cadena puede ser comprimida mediante bisección y la última puede comprimirse bien añadiendo ceros a la misma o dividiéndola recursivamente.

Octasection es una extensión de ambos, bisection y quadrisection, para datos tridimensionales. Ejemplos de estos datos son: un video (una secuencia de imágenes), una imagen en escala de grises, y una imagen en color. Dichas imágenes puede ser vista como una matriz tridimensional, donde la tercera dimensión está formada por los bits que constituyen cada píxel (los planos de bits). Suponemos que los datos son una caja rectangular P de dimensiones $2^{k_1} \times 2^{k_2} \times 2^{k_3}$, donde cada entrada es un ítem de datos (un solo bit o varios bits) y todas sus entradas tienen el mismo tamaño. El codificador realiza k iteraciones, donde $k = \text{mín}\,(k_1, k_2, k_3)$, variando j de 1 a k. En la iteración j, se crea una lista L_j, subdividiendo cada elemento de la lista precedente L_{j-1} en ocho cajas rectangulares más pequeñas. Se construye un vector indicador I_j para L_j. Los elementos duplicados de L_j son eliminados (aquí es donde obtenemos la compresión).

El flujo de datos comprimido está formado, como antes, por todos los vectores indicadores (codificados aritméticamente), seguidos de la última lista L_k.

4.32. Curvas de relleno de espacio (*space-filling*)

Una curva de relleno de espacio (*space-filling*) [Sagan 94] es una función paramétrica $\mathbf{P}(t)$ que pasa a través de todos los puntos en un área determinada bidimensional, normalmente el cuadrado unidad, cuando su parámetro t varía en el rango $[0, 1]$. Para cualquier valor t_0, el valor de $\mathbf{P}(t_0)$ es un punto $[x_0, y_0]$ en el cuadrado unidad. Matemáticamente, esta curva (que carece de autointersecciones) es una correspondencia biyectiva (mapeo) del intervalo $[0, 1]$ en el intervalo bidimensional $[0, 1] \times [0, 1]$. Para comprender cómo se construye una curva, lo mejor es pensar en ella como el límite de una sucesión

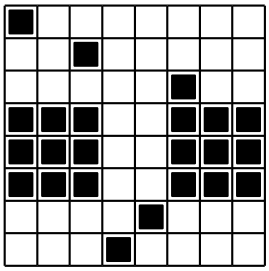


Figura 4.159: Un mapa de bits de 8×8.

infinita de curvas $\mathbf{P}_1(t), \mathbf{P}_2(t), \ldots$ que se dibujan en el interior del cuadrado unidad, donde cada curva se deriva de su predecesora mediante un proceso de *refinamiento*. Los detalles del refinamiento dependerán de la curva específica. La Sección 4.33.1 discute varias curvas de relleno de espacio, entre ellas la curva de Hilbert y la curva Sierpiński. Puesto que la secuencia de curvas es infinita, es imposible para calcular todos sus componentes. Afortunadamente, estamos interesados en una curva que pasa a través de cada píxel de un mapa de bits, no a través de cada punto matemático en la unidad cuadrada. Puesto que el número de píxeles es finito, es posible construir una curva en la práctica.

Para entender por qué tales curvas son útiles para la compresión de imágenes, el lector debe recordar el principio que se ha mencionado varias veces en el pasado, a saber, si se selecciona un píxel de una imagen al azar, hay una buena probabilidad de que sus vecinos tengan los mismos (o similares) colores. Tanto la compresión de imágenes RLE como el método quadtree están basados en este principio, pero no siempre son eficaces, como muestra la Figura 4.159. Este mapa de bits de 8×8 tiene dos concentraciones de pixeles, pero ni el método RLE ni el quadtree lo comprimen muy bien, ya que no hay rachas largas y puesto que las concentraciones de píxeles se producen al cruzar los límites del cuadrante.

La mejor compresión puede ser producida mediante un método que escanea el mapa de bits área por área en lugar de línea a línea o cuadrante por cuadrante. Ésta es la razón por la que las curvas *space-filling* proporcionan un nuevo enfoque a la compresión de imágenes. Con tal tipo de curva se visita cada punto en un área determinada, de modo que tras recorrer todos los puntos en una subárea, se desplaza a la subárea siguiente y lo atraviesa, y así sucesivamente. Usamos la curva de Hilbert (Sección 4.33.1) como ejemplo. Cada curva H_i se construye haciendo cuatro copias de la curva anterior H_{i-1}, reducciéndolas, rotándolas, y conectándolas. La nueva curva termina cubriendo la misma área que la anterior. Éste es el proceso de refinamiento de la curva de Hilbert.

El escaneo de un *bitmap* de 8×8 siguiendo una curva de Hilbert, produce la secuencia de píxeles:

$$
\begin{array}{l}
(0,0),(0,1),(1,1),(1,0),(2,0),(3,0),(3,1),(2,1),\\
(2,2),(3,2),(3,3),(2,3),(1,3),(1,2),(0,2),(0,3),\\
(0,4),(1,4),(1,5),(0,5),(0,6),(0,7),(1,7),(1,6),\\
(2,6),(2,7),(3,7),(3,6),(3,5),(2,5),(2,4),(3,4),\\
(4,4),(5,4),(5,5),(4,5),(4,6),(4,7),(5,7),(5,6),\\
(6,6),(6,7),(7,7),(7,6),(7,5),(6,5),(6,4),(7,4),\\
(7,3),(7,2),(6,2),(6,3),(5,3),(4,3),(4,2),(5,2),\\
(5,1),(4,1),(4,0),(5,0),(6,0),(6,1),(7,1),(7,0).
\end{array}
$$

La Sección 4.33.1 discute las curvas de relleno de espacio en general y muestra métodos para realizar un rápido recorrido de algunas curvas. Aquí nos gustaría señalar que los quadtrees (Sección 4.30) son un caso especial de la curva de Hilbert, un hecho ilustrado por seis figuras en dicha sección. Véase también [Prusinkiewicz y Lindenmayer 90] y .

◇ **Ejercicio 4.61 (sol. en pág. 1092):** Explórese el mapa de bits de 8×8 de la Figura 4.159 utilizando una curva de Hilbert y calcúlense las rachas (*runs*) de píxeles idénticos, comparándolas con las producidas por RLE.

En realidad, hay dos tipos de curvas de relleno de espacio de Hilbert. Las relaciones entre estos tipos, L-sistemas de Lindemayer, y códigos de Gray, pueden encontrarse en *Hilbert Curve* (curva de Hilbert) por Eric Weisstein, disponible en MathWorld — Una web de recursos de Wolfram: `http://mathworld.wolfram.com/HilbertCurve.html`

4.33. Escaneo de Hibert y VQ

La curva de relleno de espacio de Hilbert (Sección 4.33.1) es empleada por este método, disponible en [Sampath y Ansari 93] como un paso de preprocesamiento para la compresión de imágenes con pérdidas (nótese que los autores utilizan el término "Peano scan", pero en realidad utilizan la curva de Hilbert). La Curva de Hilbert se utiliza para transformar la imagen original en otra, altamente correlacionada. La compresión en sí misma se realiza mediante un algoritmo de cuantificación vectorial que aprovecha la correlación producida por el paso de preprocesamiento.

En la etapa de preprocesamiento, una imagen original de tamaño $M \times N$ es dividida en pequeños bloques de $m \times m$ píxeles cada uno (con $m = 8$, típicamente) que son escaneados siguiendo el orden de la curva de Hilbert. El resultado es una secuencia de bloques lineal (unidimensional), que luego se reorganizan en una nueva matriz bidimensional de tamaño $\frac{M}{m} \times Nm$. Este proceso produce como resultado la agrupación (o *clustering*) de los bloques que están altamente correlacionados. La "distancia" entre los bloques adyacentes es ahora menor que la distancia entre los bloques que son vecinos en un escaneo en orden raster de la imagen original.

La distancia entre dos bloques B_{ij} y C_{ij} de píxeles se mide mediante la *diferencia absoluta media*, una cantidad definida por:

$$\frac{1}{m^2}\sum_{i=1}^{m}\sum_{j=1}^{m}|B_{ij}-C_{ij}|\,.$$

Resulta que la diferencia absoluta media de bloques adyacentes en la nueva imagen reorganizada, es aproximadamente la mitad de los bloques adyacentes en una exploración en orden raster de la imagen original. La razón de ésto es que la curva de Hilbert es de relleno de espacio. Los puntos que están cercanos a la curva se encuentran próximos en la imagen. Y a la inversa, los puntos que están cerca en la imagen están normalmente próximos a la curva. En consecuencia, la curva de Hilbert actúa como una transformada de la imagen. Este hecho es la principal innovación del método aquí descrito.

Como ejemplo de exploración de Hilbert, imagine una imagen de tamaño $M \times N = 128 \times 128$. Si seleccionamos $m = 8$, obtenemos $16 \times 16 = 256$ bloques que son escaneados y convertidos en un arreglo unidimensional. Después de reorganizar, nos encontramos con una nueva imagen de tamaño 2×128 bloques ó $\frac{128}{8} \times 128 \cdot 8 = 16 \times 1024$ píxeles. La Figura 4.160 muestra cómo son escaneados los bloques siguiendo la curva de Hilbert para producir la secuencia que se muestra en la Figura 4.161. Esta secuencia es entonces reorganizada. La fila superior (primera) de la imagen reordenada contiene los bloques 1, 17, 18, 2, 3, 4, ...; y la fila inferior (última) contiene los bloques 138, 154, 153, 169, 185, 186, ...

La nueva imagen reordenada es dividida ahora en nuevos bloques de $4 \times 4 = 16$ píxeles cada uno. Los 16 píxeles de un bloque constituyen un vector. (Observe que el vector tiene 16 componentes, cada uno de los cuales puede ser de uno o más bits, dependiendo del tamaño de un píxel.) El algoritmo LBG (Sección 4.14) es utilizado para la cuantificación vectorial de los vectores. Este algoritmo requiere

un libro de códigos inicial, por lo que los implementadores eligieron cinco imágenes, escaneadas por ellos con una resolución de 256×256, y las usaron como imágenes de entrenamiento, para generar tres libros de códigos, formados por 128, 512, y 1024 vectores de código, respectivamente.

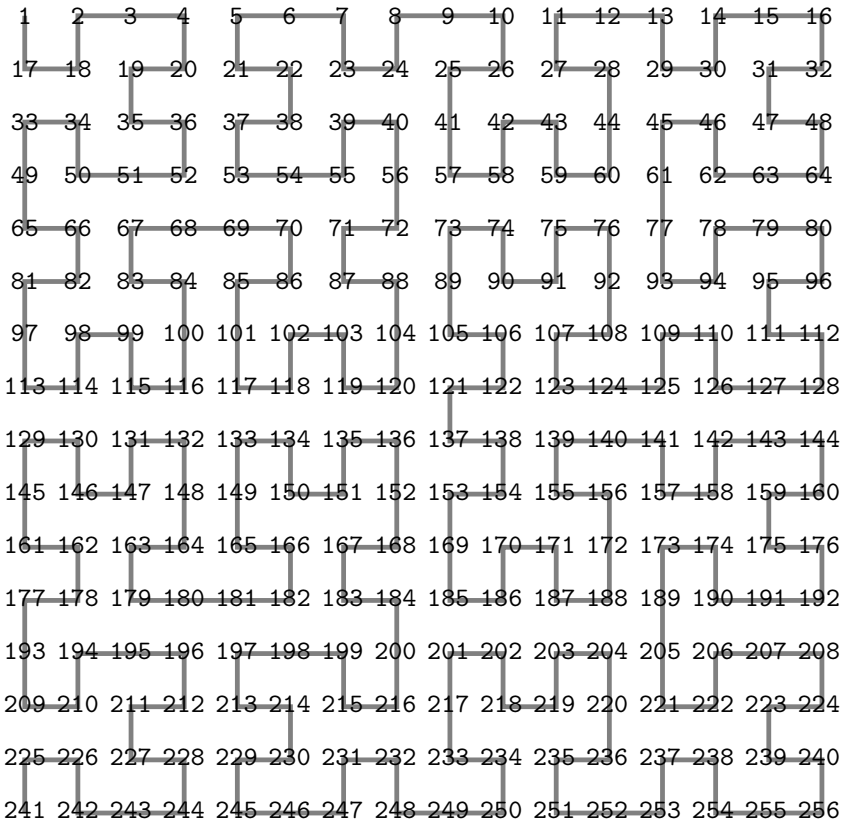


Figura 4.160: Escaneo de Hilbert de 16×16 bloques.

La principal característica del particular algoritmo de cuantificación de vectores aquí utilizado es que el escaneo de Hilbert produce bloques adyacentes que están altamente correlacionados. Como resultado, el algoritmo LBG asigna frecuentemente el mismo vector de código a una racha (*run*) de bloques, y este hecho puede ser utilizado para comprimir la imagen. Los experimentos indican que, en dicha racha, pueden participar tanto como $2-10$ bloques consecutivos (para las imágenes con detalles) y $30-70$ bloques consecutivos (para las imágenes con alta redundancia espacial). Por consiguiente, el método precede cada vector de código con un código que indica la longitud de la racha (un bloque o varios). Existen dos versiones del método, uno con un código de tamaño fijo y el otro con un código prefijo de tamaño variable.

Cuando un código de tamaño fijo es utilizado precediendo cada vector de código, la cuestión principal es el tamaño del código. Un código largo, tal como seis bits, permite rachas (*runs*) de hasta 64 bloques consecutivos con el mismo vector de código. Por otro lado, si la imagen tiene pequeños detalles, las rachas serían más cortas y algunos de los 6 bits se desperdiciarían. Un código corto, tal como dos bits, permite rachas de hasta sólo cuatro bloques, pero se pierden menos bits en el caso de una imagen con detalle alto. Una buena solución consiste en escribir el tamaño del código (que es típicamente de $2-6$ bits) al comienzo del flujo de datos comprimidos, así el decodificador sabe lo que es. El codificador puede ser muy sofisticado, tratando varios tamaños de código antes de decidirse por

1, 17, 18, 2, 3, 4, 20, 19, 35, 36, 52, 51, 50, 34, 33, 49,
65, 66, 82, 81, 97, 113, 114, 98, 99, 115, 116, 100, 84, 83, 67, 68,
69, 70, 86, 85, 101, 117, 118, 102, 103, 119, 120, 104, 88, 87, 71, 72,
56, 40, 39, 55, 54, 53, 37, 38, 22, 21, 5, 6, 7, 23, 24, 8,
9, 10, 26, 25, 41, 57, 58, 42, 43, 59, 60, 44, 28, 27, 11, 12,
13, 29, 30, 14, 15, 16, 32, 31, 47, 48, 64, 63, 62, 46, 45, 61,
77, 93, 94, 78, 79, 80, 96, 95, 111, 112, 128, 127, 126, 110, 109, 125,
124, 123, 107, 108, 92, 76, 75, 91, 90, 74, 73, 89, 105, 106, 122, 121,
137, 138, 154, 153, 169, 185, 186, 170, 171, 187, 188, 172, 156, 155, 139, 140,
141, 157, 158, 142, 143, 144, 160, 159, 175, 176, 192, 191, 190, 174, 173, 189,
205, 221, 222, 206, 207, 208, 224, 223, 239, 240, 256, 255, 254, 238, 237, 253,
252, 251, 235, 236, 220, 204, 203, 219, 218, 202, 201, 217, 233, 234, 250, 249,
248, 232, 231, 247, 246, 245, 229, 230, 214, 213, 197, 198, 199, 215, 216, 200,
184, 183, 167, 168, 152, 136, 135, 151, 150, 134, 133, 149, 165, 166, 182, 181,
180, 179, 163, 164, 148, 132, 131, 147, 146, 130, 129, 145, 161, 162, 178, 177,
193, 209, 210, 194, 195, 196, 212, 211, 227, 228, 244, 243, 242, 226, 225, 241

Figura 4.161: El resultado de 256 bloques.

uno y usarlo para comprimir la imagen.

El uso de códigos prefijo puede producir una compresión ligeramente mejor, especialmente si el codificador puede llevar a cabo un trabajo de dos pasos para determinar la frecuencia de cada racha antes de comprimir cualquier cosa. En tal caso, los mejores códigos de Huffman puede ser asignados a las rachas, resultando en una mejor compresión.

Puede lograrse una mejora adicional mediante una variante del método que utiliza el *particionamiento dinámico del libro de códigos*. Ésto se basa de nuevo en que los bloques adyacentes son muy similares. Incluso si tales bloques acabaran en vectores de código diferentes, esos vectores de código pueden ser muy similares. Esta variante selecciona los vectores de código para el primer bloque en la forma usual, utilizando el libro de códigos completo. A continuación, selecciona un conjunto de los siguientes mejores vectores de código que podrían haber sido utilizados para codificar este bloque. Este conjunto se convierte en la *parte activa* del libro de códigos. (Nótese que el decodificador puede imitar esta selección.) Luego, se compara el segundo bloque con el primer bloque y se mide la distancia entre ellos. Si esta distancia es menor que un umbral dado, el vector de código para el segundo bloque es seleccionado del conjunto activo. Puesto que el conjunto activo es mucho más pequeño que el libro de códigos completo, ésto conduce a una compresión mucho mejor. Sin embargo, cada vector de código debe ser precedido por un bit para informar al decodificador si el vector de código fue seleccionado del libro de códigos o sólo del conjunto activo.

Si la distancia es mayor que el umbral, es seleccionado un vector de código del libro de códigos completo para el segundo bloque, y se elige un nuevo conjunto activo, para ser utilizado (con suerte) con el tercer bloque.

Si el escaneo de Hilbert realmente acaba con bloques adyacentes que están altamente correlacionados, una gran fracción de los bloques son codificados a partir de los conjuntos activos, lo que mejora

considerablemente la compresión. Un tamaño típico del libro de códigos es 128−1024 entradas, mientras que el tamaño de un conjunto activo puede ser de tan sólo cuatro vectores de código. Ésto reduce el tamaño de un puntero del libro de códigos de 7−10 bits a 2 bits.

La elección del umbral para esta variante es importante. Parece que una modificación adaptativa del umbral puede funcionar mejor, pero los desarrolladores no indican cómo puede ser implementado ésto.

4.33.1. Ejemplos

Una curva de relleno de espacio llena completamente parte del espacio, pasando por cada cada punto en ese área. Lo consigue cambiando de dirección repetidamente. Sólo comentaremos curvas que llenan parte del plano bidimensional, pero el concepto de curva de relleno de espacio existe para cualquier número de dimensiones.

◇ **Ejercicio 4.62 (sol. en pág. 1092):** Muéstrese un ejemplo de una curva de relleno de espacio en una dimensión.

Se conocen varias de tales curvas y todas se definen recursivamente. Una definición típica comienza con una curva simple C_0, muestra cómo utilizarla para construir otra curva más compleja C_1, y define la curva de relleno de espacio final, como el límite de la sucesión de curvas C_0, C_1, ...

4.33.2. La curva de Hilbert

(Esta discusión se basa en el enfoque de [Wirth 76].) Quizás la más conocida de estas curvas es la curva de Hilbert, descubierta por el gran matemático David Hilbert en 1891. La curva de Hilbert [Hilbert 91] es el límite de una secuencia H_0, H_1, H_2,... de curvas, algunas de las cuales se muestran en la Figura 4.162. Se definen mediante los siguientes pasos:

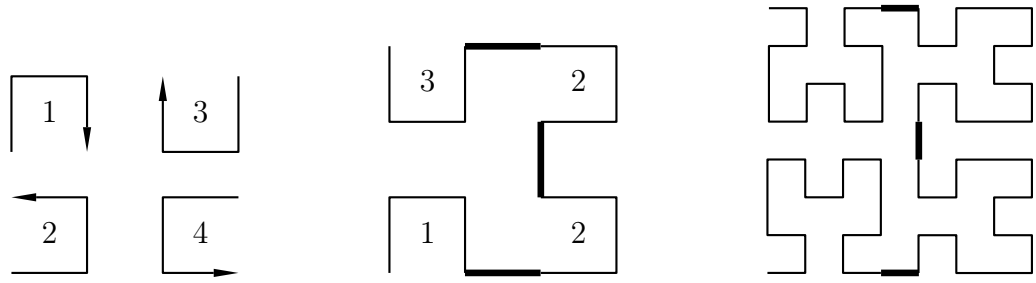


Figura 4.162: Curvas de Hilbert de órdenes 1, 2, y 3.

0. H_0 es un simple punto.

1. H_1 consta de cuatro copias (del punto) H_0, conectadas con tres segmentos consecutivos de longitud h en ángulos rectos entre sí. Cuatro orientaciones de esta curva, etiquetadas 1, 2, 3, y 4, se muestran en la Figura 4.162a.

2. La siguiente curva, H_2, en la secuencia se construye conectando cuatro copias con diferentes orientaciones de H_1 con tres segmentos rectos de longitud $h/2$ (mostrados en negrita en la Figura

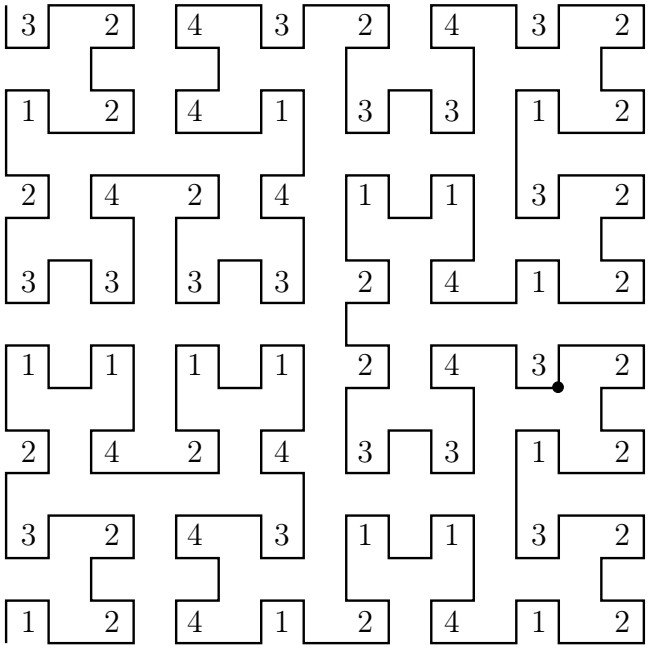


Figura 4.163: Curva de Hilbert de orden 4.

4.162b). De nuevo, hay cuatro posibles orientaciones de H_2, y la mostrada es #2. Ésta se construye con las orientaciones 1223 de H_1, conectadas por segmentos que se dirigen a la derecha, arriba y a la izquierda. La construcción de las cuatro orientaciones de H_2 se resume en la Tabla Sol.43.

La curva H_3 se muestra en la Figura 4.162c. La curva en particular mostrada es la orientación 1223 de H_2.

Las Figuras 4.163, 4.164, y 4.165 muestran las curvas de Hilbert[18] de orden 4, 5 y 6. Es fácil ver lo rápido que estas curvas se vuelven extremadamente complejas.

4.33.3. La curva de Sierpiński

Otra curva conocida curva de relleno de espacios es la curva de Sierpiński. La Figura 4.166 muestra las curvas S_1 y S_2, y Sierpiński demostró [Sierpiński 12] que el límite de la sucesión S_1, S_2,... es una curva que pasa por todos los puntos del cuadrado unidad $[0,1] \times [0,1]$.

Para construir esta curva, necesitamos encontrar cómo se construye S_2 a partir de cuatro copias de S_1. Lo primero que viene a la mente es seguir el método de construcción utilizado para la curva de Hilbert, i.e., tomar cuatro copias de S_1, eliminar un borde en cada una, y conectarlas. Ésto, por desgracia, no funciona, porque la curva de Sierpiński es muy diferente de la curva de Hilbert. Es cerrada, y tiene una única orientación. Un mejor enfoque es comenzar con cuatro partes que constituyen cuatro orientaciones de una curva abierta, y conectarlas con segmentos rectos. Los segmentos se muestran en discontinuo en la Figura 4.166. Observe cómo la Figura 4.166a se construye a partir de cuatro orientaciones de una curva base de tres partes conectadas por cuatro segmentos cortos, de trazos. La Figura 4.166b se construye de forma similar con cuatro orientaciones de una curva compleja de 15

[18] Estas figuras se han realizado en TEX mediante el sistema de Lindenmayer siguiente: $L \rightarrow +RF - LFL - FR+$, $R \rightarrow -LF + RFR + FL-$, axioma=L, ángulo de 90°, rotación de $-90°$.

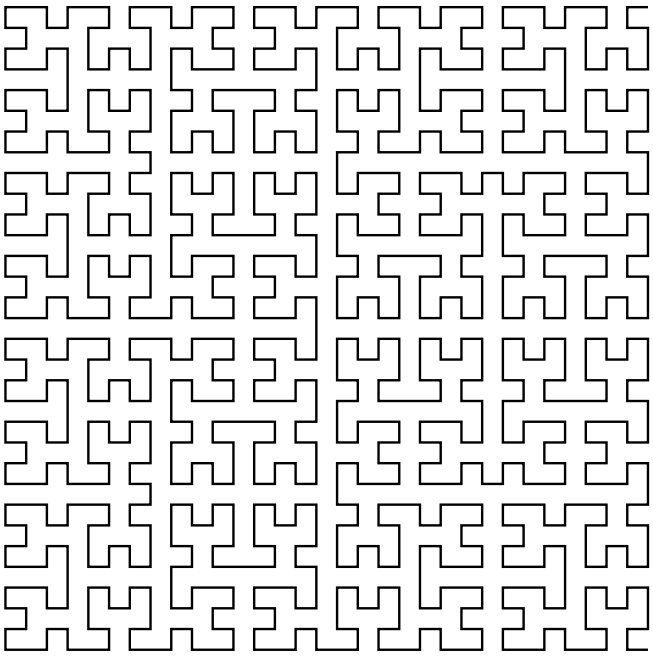


Figura 4.164: Curva de Hilbert de orden 5.

partes, conectadas por el mismo corto segmento de trazos. Si denotamos las cuatro curvas básicas A, B, C, y D, entonces el regla de construcción básica de la curva Sierpiński es: S : A ↘ B ↙ C ↖ D ↗, y las reglas de recursión son:

$$\begin{array}{ll} \text{A}: & \text{A} \searrow \text{B} \rightarrow\rightarrow \text{D} \nearrow \text{A} \\ \text{B}: & \text{B} \swarrow \text{C} \downarrow\downarrow \text{A} \searrow \text{B} \\ \text{C}: & \text{C} \nwarrow \text{D} \leftarrow\leftarrow \text{B} \swarrow \text{C} \\ \text{D}: & \text{D} \nearrow \text{A} \uparrow\uparrow \text{C} \nwarrow \text{D} \end{array} \tag{4.55}$$

La Figura 4.167 muestra las cinco curvas de Sierpiński[19] de órdenes 1−5 superpuestas una a la otra.

◇ **Ejercicio 4.63 (sol. en pág. 1092):** La Figura 4.168 muestra tres iteraciones de la curva de relleno de Peano[20], desarrollada en 1890. Utilícense las técnicas desarrolladas anteriormente para las curvas de Hilbert y Sierpiński, para describir cómo se construye la curva de Peano. (Pista: Las curvas mostradas son P_1, P_2, y P_3. La primera curva, P_0, no se muestra en esta secuencia.)

4.33.4. Recorrido de la curva de Hilbert

Las curvas de relleno de espacio se utilizan en la compresión de imágenes (Sección 4.32), por lo que es importante desarrollar métodos para un rápido recorrido de tales curvas. Aquí se ilustran dos

[19] Todas las gráficas de Sierpiński se han realizado en TEX mediante el sistema de Lindenmayer siguiente: $L \rightarrow +R-F-R+$, $R \rightarrow -L+FF+L-$, axioma=$L--F--L--F$, ángulo de 45º, rotación de 45º. Este sistema es una modificación del encontrado en el documento "Introducción a la geometría fractal III. Sistemas L" de Carlos Munuera; Universidad de Valladolid. Con este sistema, la curva S_i, es en realidad la gráfica de Sierpiński de orden $(2 \times i)+1$. Así, las cinco curvas mostradas son las gráficas de Sierpiński de orden 3, 5, 7, 9, y 11 del L-sistema que he usado; éstas se corresponden con las curvas de Sierpiński, 1, 2, 3, 4, y 5, respectivamente.

[20] Estas figuras se han realizado en TEX mediante el sistema de Lindenmayer siguiente: $L \rightarrow LFRFL-F-RFLFR+F+LFRFL$, $R \rightarrow RFLFR+F+LFRFL-F-RFLFR$, axioma=L, ángulo de 90º, rotación de 90º.

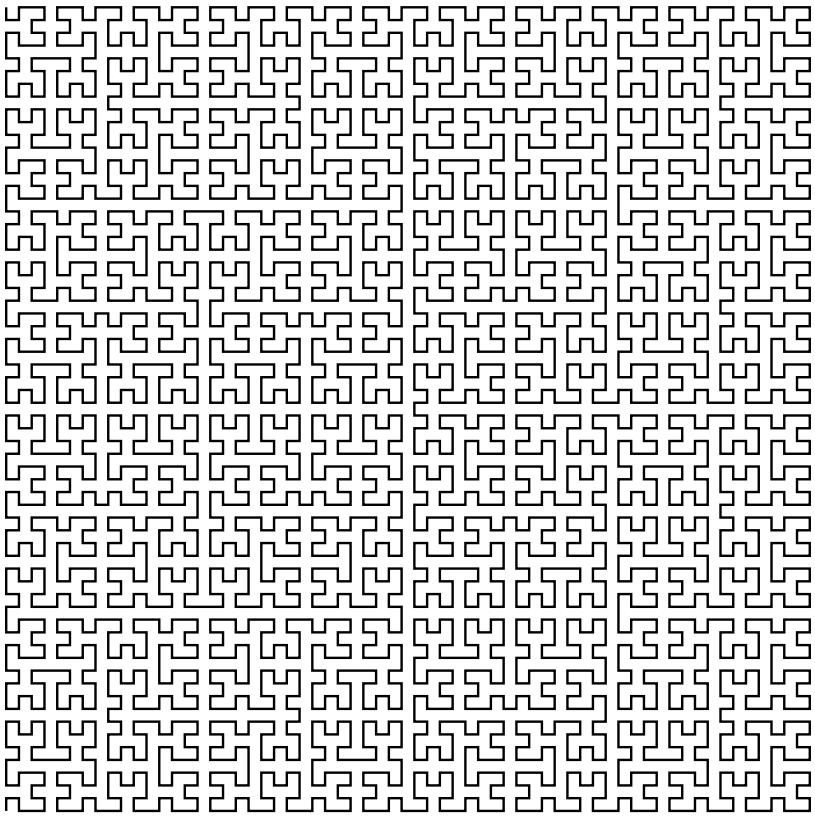
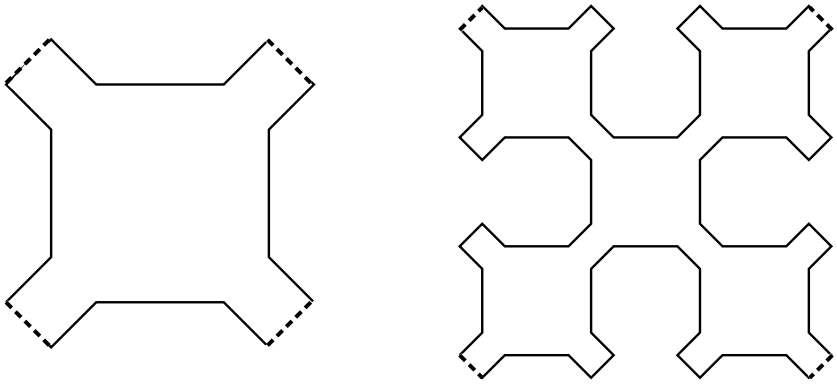


Figura 4.165: Curva de Hilbert de orden 6.



(a) (b)

Figura 4.166: Curvas de Sierpiński de órdenes 1 y 2.

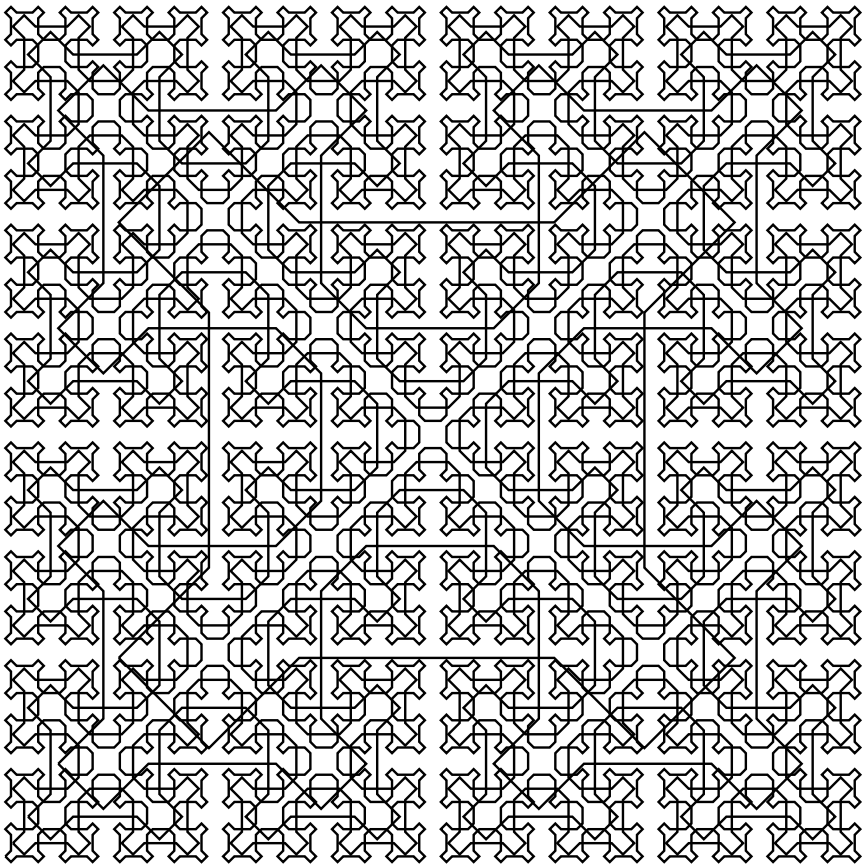
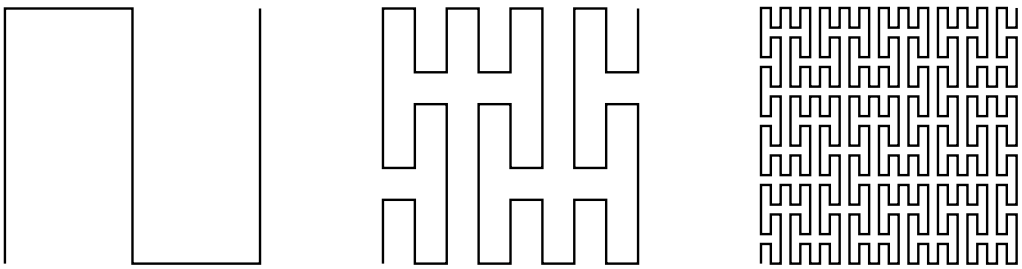


Figura 4.167: Curvas de Sierpiński de órdenes 1−5.



(a) (b) (c)

Figura 4.168: Tres iteraciones de la curva de Peano.

Par (bits)	x	y	Tabla sig.	Par (bits)	x	y	Tabla sig.	Par (bits)	x	y	Tabla sig.	Par (bits)	x	y	Tabla sig.
00	0	0	2	00	0	0	1	00	1	1	4	00	1	1	3
01	1	0	1	01	0	1	2	01	0	1	3	01	1	0	4
10	1	1	1	10	1	1	2	10	0	0	3	10	0	0	4
11	0	1	4	11	1	0	3	11	1	0	2	11	0	1	1
	(1)				(2)				(3)				(4)		

Tabla 4.169: Coordenadas de los nodos en H_i.

enfoques, ambos guiados por tablas, para recorrer la curva de Hilbert.

El primer enfoque [Cole 86] se basa en la observación de que la curva de Hilbert H_i se construye a partir de cuatro copias de su predecesora H_{i-1} colocada en diferentes orientaciones. Una mirada a las Figuras 4.162, 4.163, 4.164, 4.165 debe convencer al lector de que H_i está formada por 2^{2i} nodos conectados con segmentos rectos. Los números de nodo, por lo tanto, varían de 0 a $2^{2i}-1$ y requieren $2i$ bits cada uno. Con el fin de atravesar la curva necesitamos una función que calcule las coordenadas (x, y) de un nodo i desde el número de nodo i. Las coordenadas (x, y) de un nodo en H_i son números de i bits.

Un vistazo a la Figura 4.163 muestra cómo los nodos sucesivos se ubican inicialmente en el cuadrante inferior izquierdo, y luego se desplazan al cuadrante inferior derecho, el cuadrante superior derecho, y finalmente, el superior izquierdo. Esta figura muestra la orientación #2 de la curva, por lo que podemos decir que esta orientación de H_i atraviesa los cuadrantes 0, 1, 2, y 3, donde la numeración de los mismos es $\left(\begin{smallmatrix}3&2\\0&1\end{smallmatrix}\right)$. Ahora es claro que los dos bits del extremo izquierdo de un número de nodo determinan su cuadrante. Similarmente, el siguiente par de bits en el número de nodo determinan su subcuadrante dentro del cuadrante, pero aquí nos encontramos con el agravante de que cada subcuadrante se coloca en una orientación diferente en su cuadrante. Este enfoque, por lo tanto, utiliza la Tabla 4.169 para determinar las coordenadas de un nodo desde su número.

Como ejemplo, podemos calcular las coordenadas xy del nodo 109 (el 110-ésimo nodo) de orientación #2 de H_4. La curva H_4 tiene $2^{2\cdot 4} = 256$ nodos, por lo que los números de nodo constan de ocho bits cada uno, y $109 = 01101101_2$. Comenzamos con la Tabla 4.169(1). Los dos bits el extremo izquierdo del número de nodo son 01, y la tabla (1) nos dice que la coordenada x comienza en 1, y la coordenada y, en 0, y debemos continuar en la tabla (1). El siguiente par de bits es 10, y la tabla (1) nos dice que el siguiente bit de x es 1, el siguiente bit de y es 1, y debemos permanecer en la tabla (1). El tercer par de bits es 11, por lo que la tabla (1) nos dice que el siguiente bit de x es 0, el siguiente bit de y es 1, y hay que pasar a la tabla (4). El último par de bits es 01, y la tabla (4) nos dice que debemos agregar 1 y 0 a las coordenadas de x e y, respectivamente. Por consiguiente, las coordenadas son $x = 1101 = 13$, $y = 0110 = 6$, como puede comprobarse directamente en la Figura 4.163 (el círculo pequeño).

También es posible transformar un par de coordenadas (x, y), cada una dentro del rango $\left[0, 2^i - 1\right]$, a un número de nodo en H_i mediante la Tabla 4.170.

◇ **Ejercicio 4.64 (sol. en pág. 1092):** Utilícese la Tabla 4.170 para calcular el número de nodo, del nodo H_4, cuyas coordenadas son $(13, 6)$.

El segundo enfoque de recorrido de la curva de Hilbert utiliza la Tabla Sol.43. La orientación #2 de la curva H_2 mostrada en la Figura 4.162(b) se recorre en el orden 1223. La misma orientación de la curva H_3 de la Figura 4.162(c) es recorrida en orden 2114 1223 1223 4332, pero la Tabla Sol.43 nos dice que 2114 es el orden de recorrido para la orientación #1 de H_2, 1223 es el recorrido para la orientación #2 de H_2, y 4332 lo es para la orientación #3. El recorrido de la orientación #2 de H_3,

Par xy	Par int.	Tabla sig.	Par xy	Par int.	Tabla sig.	Par xy	Par int.	Tabla sig.	Par xy	Par int.	Tabla sig.
00	00	2	00	00	1	00	10	3	00	10	4
01	11	4	01	01	2	01	01	3	01	11	1
10	01	1	10	11	3	10	11	2	10	01	4
11	10	1	11	10	2	11	00	4	11	00	3
	(1)			(2)			(3)			(4)	

Tabla 4.170: Números de nodo en H_i.

por lo tanto, se basa también en la secuencia de 1223. Del mismo modo, la orientación #2 de H_4 se recorre (Figura 4.163) en el orden:

$$\begin{array}{c} 1223\ 2114\ 2114\ 3441\ \ 2114\ 1223\ 1223\ 4332 \\ 2114\ 1223\ 1223\ 4332\ \ 3441\ 4332\ 4332\ 1223, \end{array}$$

el cual se reduce a 2114 1223 1223 4332, que a su vez se reduce a la misma secuencia 1223.

La idea es, pues, crear el orden de recorrido para la orientación #2 de H_i comenzando con la secuencia de 1223, y desarrolarla recursivamente $i-1$ veces, usando la Tabla Sol.43.

◇ **Ejercicio 4.65 (sol. en pág. 1092):** (Fácil.) Muéstrese cómo aplicar este método para recorrer la orientación #1 de H_i.

Una función (`hilbert.m`) en MATLAB, para calcular el recorrido de la curva, está disponible en [Matlab 99]. Fue escrita por Daniel Leo Lau (dllau@engr.uky.edu). La llamada `hilbert(4)` produce la matriz de 4×4:

$$\begin{array}{cccc} 5 & 6 & 9 & 10 \\ 4 & 7 & 8 & 11 \\ 3 & 2 & 13 & 12 \\ 0 & 1 & 14 & 15 \end{array}.$$

4.33.5. Recorrido de la curva de Peano

Las curvas de Peano P_0, P_1, y P_2 de la Figura Sol.42, tienen 1, 3^2, y 3^4 nodos, respectivamente. En general, P_n consta de 3^{2n} nodos, numerados 0, 1, 2, ..., $3^{2n}-1$. Ésto sugiere que la curva de Peano [Peano 90] está de alguna manera basada en el número 3, en contraste con la curva de Hilbert, que se basa en el 2. Las coordenadas de los nodos pueden variar de $(0,0)$ a $(n-1,n-1)$. Resulta que existe una correspondencia entre los números de nodo y sus coordenadas [Cole 85], que utiliza códigos de Gray reflejados de base 3 (Sección 4.2.1).

Un código de Gray reflejado [Gray 53] es una permutación enteros de i dígitos de forma que los enteros consecutivos difieren en un solo dígito. Esta es una manera de obtener estos códigos para números binarios. Se comienza con $i=1$. Sólo hay dos dígitos de 1 bit, a saber, 0 y 1; y su diferencia es de 1 bit. Para obtener el RGC para $i=2$ se procede de la siguiente manera:

1. Se copia la secuencia $(0,1)$.

2. Se anexa (a la izquierda o a la derecha) un bit 0 a la secuencia original y un bit 1 a la copia. El resultado es $(00,01)$, $(10,11)$.

3. Se reflejar (invierte) la segunda secuencia. El resultado es $(11,10)$.

4. Se concatenan las dos secuencias para obtener $(00, 01, 11, 10)$.

Es fácil ver que números consecutivos difieren en sólo en un bit.

◇ **Ejercicio 4.66 (sol. en pág. 1093):** Síganse las reglas anteriores para obtener el RGC binario para $i = 3$.

Observe que los números primero y último en un RGC también difieren en un bit. Los RGCs pueden ser creados para cualquier sistema numérico utilizando la siguiente notación y reglas: Sea $a = a_1 a_2 \dots a_m$ un número entero no negativo en base n (i.e., $0 \leq a_i < n$). Se define la cantidad $p_j = \left(\sum_{i=1}^{j} a_i\right) \bmod 2$, y se denota el RGC en base n de a como $a' = b_1 b_2 \dots b_m$. Los dígitos b_i de a' pueden calcularse mediante:

$$b_1 = a_1; \quad b_i = \begin{cases} a_i & \text{si } p_{i-1} = 0; \\ n - 1 - a_i & \text{si } p_{i-1} = 1. \end{cases} \quad n \text{ impar} \qquad i = 2, 3, \dots, m.$$

$$\begin{cases} a_i & \text{si } a_{i-1} \text{ es par;} \\ n - 1 - a_i & \text{si } a_{i-1} \text{ es impar.} \end{cases} \quad n \text{ par}$$

[Observe que $(a')' = a$ tanto si n es par como si es impar.] Por ejemplo, el RGC de la secuencia de números en base 3 (trits) 000, 001, 002, 010, 011, 012, 020, 021, 022, 100, 101... es 000, 001, 002, 012, 011, 010, 020, 021, 022, 122, 121....

La conexión entre las curvas de Peano y los RGCs es la siguiente: Sea a un nodo en la curva P_m de Peano. Se escribe a como un número ternario (en base 3) con $2m$ trits $a = a_1 a_2 \dots a_{2m}$. Sea $a' = b_1 b_2 \dots b_{2m}$ el RGC equivalente de a. Se calculan los dos números $x' = b_2 b_4 b_6 \dots b_{2m}$ e $y' = b_1 b_3 b_5 \dots b_{2m-1}$. El número x' es el RGC de un número x, y similarmente para y'. Los dos números (x, y) son las coordenadas del nodo a en P_m.

4.34. Métodos de autómatas finitos

Los autómatas finitos son hasta cierto punto similares al método IFS de la Sección 4.35. Se basan en el hecho de que las imágenes utilizadas en la práctica tienen una cierta cantidad de autosemejanza, i.e., a menudo es posible encontrar una parte de la imagen que tiene el mismo (o casi el) aspecto que otra parte, excepto tal vez en el tamaño, el brillo o el contraste. En la Sección 4.35 se muestra que IFS utiliza transformaciones afines para el emparejamiento de partes de la imagen. Los métodos aquí descritos, por otra parte, tratan de describir una subimagen como una suma ponderada de otras subimágenes.

Se describen dos métodos en esta sección: los *autómatas finitos ponderados* (WFA o *Weighted Finite Automata*) y los *autómatas finitos generalizados* (GFA o *Generalized Finite Automata*). Ambos se deben a Karel Culik, el primero en colaboración con Jarkko Kari, y el segundo con Vladimir Valenta. El término "autómata" se utiliza porque estos métodos representan la imagen a comprimir en términos de un grafo que es muy similar a los grafos utilizados para representar los autómatas de estado finito (también llamados máquinas de estado finito, véase la Sección 8.8). Las principales referencias son: [Culik y Kari 93], [Culik y Kari 94a,b], y [Culik y Kari 95].

4.34.1. Autómata finito ponderado

El WFA comienza con una imagen para ser comprimida. Localiza las partes de la imagen que son idénticas o muy similares a la imagen completa, o a otras partes, y construye un grafo que refleja las relaciones entre estas partes y la imagen completa. Luego se comprimen los diversos componentes del grafo y se convierten en la imagen comprimida.

Las partes de la imagen utilizadas por el WFA se obtienen realizando un particionamiento quadtree de la imagen. Cualquier cuadrante, subcuadrante, o píxel de la imagen puede ser representado mediante una cadena de dígitos 0, 1, 2, y 3. Cuanto más larga sea la cadena, menor es el área de la imagen (subcuadrado) que representa. Denotamos tal secuencia por $a_1a_2\ldots a_k$.

Los quadtrees se introdujeron en la Sección 4.30. Asumimos que la numeración de cuadrantes de la Figura 4.171a se extiende recursivamente a los subcuadrantes. La Figura 4.171b muestra cómo cada uno de los 16 subcuadrantes producidos a partir de los 4 los originales se identifican mediante una cadena de dos dígitos de cifras 0, 1, 2, y 3. Después de otra subdivisión, cada uno de los subsubcuadrantes resultantes es identificado por una cadena de 3 dígitos, y así sucesivamente. El área en negro de la Figura 4.171c, por ejemplo, se identifica mediante la cadena 1032.

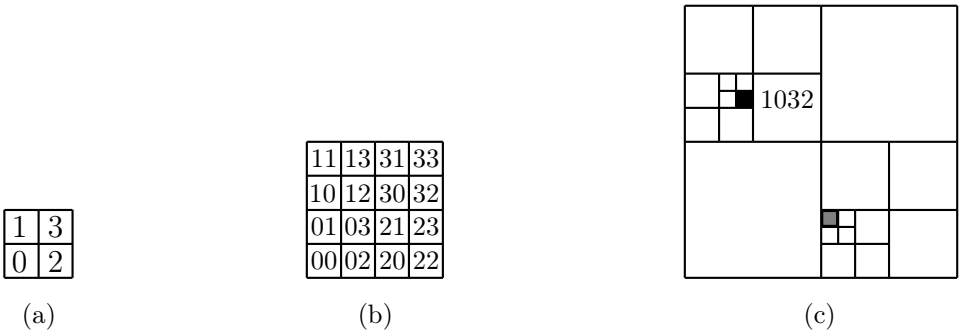


Figura 4.171: Numeración de cuadrantes.

◇ **Ejercicio 4.67 (sol. en pág. 1093):** ¿Qué cadena identifica el área gris de la Figura 4.171c.

En lugar de estar constantemente diciendo "cuadrante, subcuadrante o subsubcuadrante," usaremos el término "subcuadrado" en esta sección.

◇ **Ejercicio 4.68 (sol. en pág. 1093):** (Propuesto por Karel Culik) ¿Qué tiene de especial la particular numeración de cuadrantes de la Figura 4.171a?

Si el tamaño de la imagen es de $2^n \times 2^n$, entonces un solo píxel está representado por una cadena de n dígitos, y una cadena $a_1a_2\ldots a_k$ de k dígitos representa un subcuadrado de tamaño $2^{n-k} \times 2^{n-k}$ píxeles. Una vez que ésto se comprende, no es difícil ver cómo cualquier tipo de imagen —binivel, en escala de grises, o en color— puede ser representada mediante un grafo. Matemáticamente, un grafo es un conjunto de nodos (o vértices) conectados mediante aristas. Un nodo puede contener datos, y una arista puede tener una etiqueta. Los grafos utilizados en un WFA representan autómatas finitos, por lo que se utiliza el término *estado* en lugar de nodo o vértice.

El codificador WFA comienza generando un grafo que representa la imagen que va a ser comprimida. Un estado en este grafo representa la imagen completa, y cada uno de los otros estados representa una subimagen. Las aristas muestran cómo ciertas subimágenes pueden expresarse como combinaciones lineales de la imagen completa (escalada) o de otras subimágenes. Puesto que las subimágenes son generadas mediante un quadtree, no se superponen. La regla básica para la conexión de los estados con aristas es: Si un cuadrante a (donde a puede ser 0, 1, 2, ó 3) de la subimagen i es idéntica a la subimagen j, se construye una arista del estado i al estado j, y se etiqueta a. En una imagen arbitraria, no es muy común encontrar dos subimágenes idénticas, por lo que la regla anterior se amplía a; si el subcuadrado a de una subimagen i puede obtenerse multiplicando todos los píxeles de la subimagen j por una constante ω, se construye una arista del estado i al estado j, etiquetada a, y se le asigna un peso ω. Usamos la notación $a\,(\omega)$ o, si $\omega = 1$, sólo a.

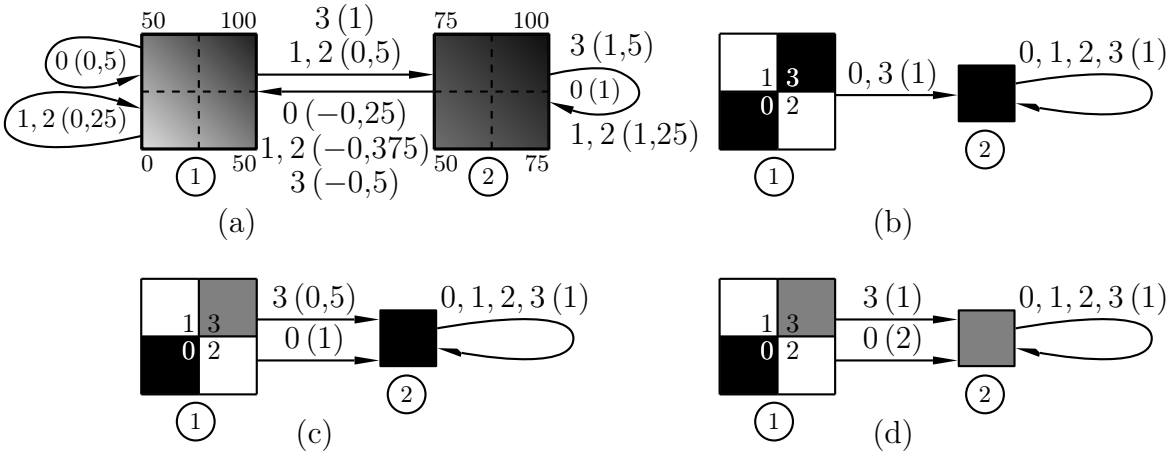


Figura 4.172: Grafos para imágenes sencillas.

Un sofisticado algoritmo de codificación puede descubrir que un subcuadrado de la subimagen i puede expresarse como la suma ponderada de las subimágenes j y k con los pesos u y ω, respectivamente. En tal caso, deben construirse dos aristas. Una de i a j, etiquetada $a\,(u)$, y otra, de i a k, etiquetada $a\,(\omega)$. En general, dicho algoritmo puede descubrir que un subcuadrado de la subimagen i puede expresarse como una suma ponderada (o una combinación lineal) de varias subimágenes, con diferentes pesos. En tal caso, debe construirse una arista para cada término de la suma.

Denotamos el conjunto de estados del grafo, Q, y el número de estados, m.

Observe que los pesos no tienen que sumar 1. Ellos pueden ser cualesquier números reales, y pueden ser mayores que 1 ó negativos. La Figura 4.172b es la gráfica de un sencillo tablero de ajedrez de 2×2. Es evidente que se pueden ignorar los subcuadrados 1 y 2, ya que son todo blancos. El grafo final incluye los estados para los subcuadrados que no son todo blancos, y tiene dos estados. El estado 1 es la imagen completa, y el estado 2 es una subimagen toda negra. Dado que los subcuadrados 0 y 3 son idénticos al estado 2, hay dos aristas (mostradas como una sola) del estado 1 al estado 2, donde la notación $0, 3\,(1)$ representa $0\,(1)$ y $3\,(1)$. Puesto que el estado 2 representa los subcuadrados 0 y 3, puede denominarse q_0 o q_3.

La Figura 4.172c muestra el grafo de un tablero de ajedrez de 2×2 donde el cuadrante 3 es un 50 % gris. Ésto es también es un grafo de dos estados. El estado 2 de esta gráfica es una imagen de color negro y es idéntico al cuadrante 0 de la imagen (también podemos llamarlo q_0). Por ello, hay una arista etiquetada $0\,(1)$. El estado 2 también puede generar el cuadrante 3, si se multiplica por 0,5, y ésto se ha expresado mediante la arista $3\,(0{,}5)$. La Figura 4.172d muestra otro grafo de dos estados representando la misma imagen. Esta vez, el estado 2 (que puede ser nombrado q_3) es una imagen en un 50 % de color gris, y los pesos son diferentes.

La Figura 4.172a muestra un ejemplo de una suma ponderada. La imagen (estado 1) varía suavemente desde el negro en la esquina superior derecha hasta el blanco en la esquina inferior izquierda. El grafo contiene un estado más, estado 2, que es idéntico al cuadrante 3 (también podemos llamarlo q_3). Varía desde el negro en la esquina superior derecha, al 50 % de gris en la esquina inferior izquierda, con un 75 % de gris en las otras dos esquinas. El cuadrante 0 de la imagen se obtiene cuando la imagen completa se multiplica por 0,5, por lo que hay una arista del estado 1 a sí mismo etiquetado $0\,(0{,}5)$. Los cuadrantes 1 y 2 son idénticos y se obtienen como una suma ponderada de la imagen completa (estado 1) multiplicada por 0,25 y del estado 2 (cuadrante 3) multiplicado por 0,5. Los cuatro cuadrantes del

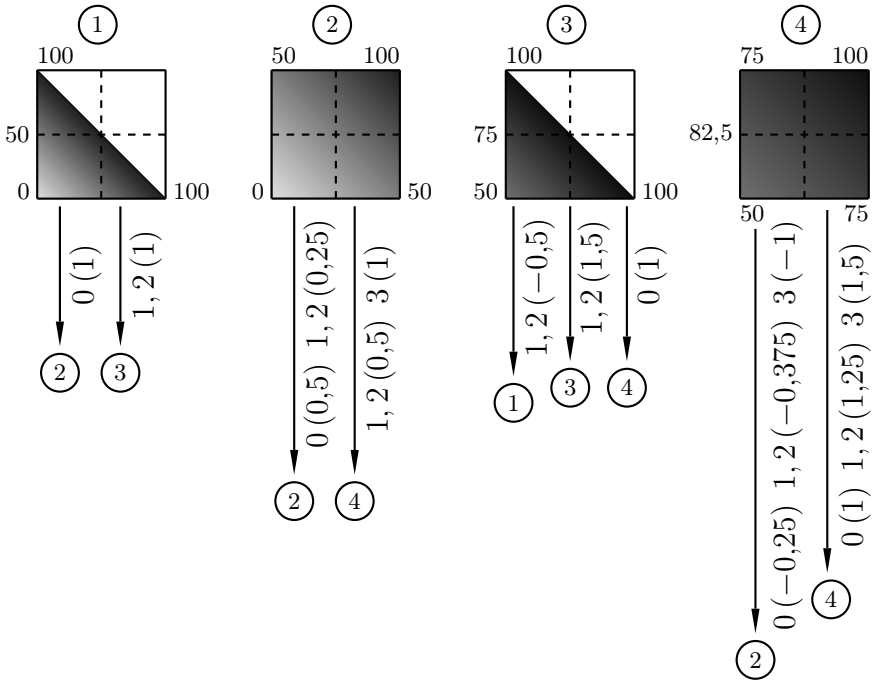


Figura 4.173: Un grafo de cuatro estados.

estado 2 se obtienen de forma similar como sumas ponderadas de los estados 1 y 2.

El gris promedio de cada estado es fácil de entender en este sencillo caso. Por ejemplo, el gris promedio de estado 1 de la Figura 4.172a es 0,5 y la del cuadrante 3 del estado 2 es $^{(1+0,75)}/_{2} = 0{,}875$. El gris promedio se utiliza porteriormente y recibe el nombre de *distribución final*.

Observe que los dos estados de la Figura 4.172a tienen el mismo tamaño. De hecho, nada se ha dicho hasta ahora sobre los tamaños y resoluciones de la imagen y las diferentes subimágenes. El proceso de construcción de la gráfica no depende de la resolución, por lo que el WFA se puede aplicar a *imágenes multirresolución*, i.e., imágenes que pueden existir en varias resoluciones diferentes.

La Figura 4.173 es otro ejemplo. La imagen original es el estado 1 del grafo, y el grafo tiene cuatro estados. El estado 2 es el cuadrante 0 de la imagen (por lo que puede ser llamado q_0). El estado 3 representa los cuadrantes (idénticos) 1 y 2 de la imagen (que pueden ser nombrados q_1 o q_2). Cabe señalar que, en principio, un estado de un grafo WFA no tiene que corresponderse con algún subcuadrado de la imagen. Sin embargo, el algoritmo de inferencia recursiva utilizado por el WFA genera un grafo donde cada estado corresponde a un subcuadrado de la imagen.

◇ **Ejercicio 4.69 (sol. en pág. 1093):** En la Figura 4.173, el estado 2 del grafo se representa en función de sí misma y del estado 4. Muéstrese cómo se representa en términos de sí mismo y de un estado completamente negro.

Una imagen multirresolución M puede, en principio, ser representada mediante una función de intensidad $f_M(x, y)$ que proporciona la intensidad de la imagen en el punto (x, y). Por simplicidad, asumimos que x e y varían de forma independiente en el intervalo $[0, 1]$, que la intensidad varía entre 0 (blanco) y 1 (negro), y que la esquina inferior izquierda de la imagen está localizada en el origen [i.e., es el punto $(0, 0)$]. Por consiguiente, la subimagen del estado 2 de la Figura 4.173 se define mediante la función:

$$f(x, y) = \frac{x + y}{2}.$$

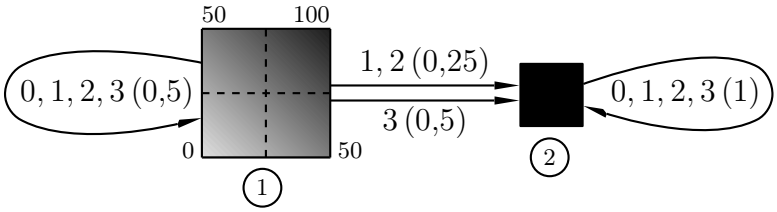


Figura 4.174: Un grafo de dos estados.

◇ **Ejercicio 4.70 (sol. en pág. 1093):** ¿Qué función describe la imagen del estado 1 de la Figura 4.173?

Dado que se trata de imágenes multirresolución, la función intensidad f debe ser capaz de generar la imagen en cualquier resolución. Si deseamos, por ejemplo, reducir la resolución a un cuarto de la original, cada nuevo píxel debe ser calculado a partir de un conjunto de cuatro originales. El valor obvio para tal píxel "grueso" es el promedio de los cuatro píxeles originales. Ésto implica que a baja resolución, la función intensidad f debe calcular la intensidad media de una región de la imagen original. En general, el valor de f para un subcuadrado ω debe satisfacer:

$$f(\omega) = \frac{1}{4}\left[f(\omega 0) + f(\omega 1) + f(\omega 2) + f(\omega 3)\right].$$

Esta función recibe el nombre de *preservación del promedio* (*average preserving* o ap). También es posible aumentar la resolución, pero los detalles creados en tal caso serían artificiales.

Dada una imagen arbitraria, sin embargo, generalmente, no conocemos su función de intensidad. Además, cualquier imagen dada tiene una resolución limitada, finita. En consecuencia, el WFA procede como sigue: Comienza con una imagen determinada con una resolución de $2^n \times 2^n$. Utiliza un algoritmo de inferencia para construir su grafo; después extrae información suficiente a partir del grafo para poder reconstruir la imagen en su resolución original o inferior. La información obtenida a partir del grafo se comprime y se convierte en la imagen comprimida.

El primer paso en la extracción de información a partir del grafo es la construcción de cuatro *matrices de transición* W_0, W_1, W_2 y W_3 de acuerdo con la siguiente regla: Si existe una arista etiquetada q del estado i al estado j, entonces el elemento (i, j) de la matriz de transición W_q se establece en el peso ω de la arista. De lo contrario, $(W_q)_{i,j}$ se establece en cero. Un ejemplo son las cuatro matrices de transición resultantes del grafo de la Figura 4.174. Ellas son:

$$W_0 = \begin{pmatrix} 0{,}5 & 0 \\ 0 & 1 \end{pmatrix},\ W_1 = \begin{pmatrix} 0{,}5 & 0{,}25 \\ 0 & 1 \end{pmatrix},\ W_2 = \begin{pmatrix} 0{,}5 & 0{,}25 \\ 0 & 1 \end{pmatrix},\ W_3 = \begin{pmatrix} 0{,}5 & 0{,}5 \\ 0 & 1 \end{pmatrix}.$$

El segundo paso es la construcción de un vector columna F de tamaño m llamado *distribución final* (éste no es el mismo que la función intensidad f). Cada componente de F es la intensidad media de la subimagen asociada con un estado del grafo. Así, para el grafo de dos estados de la Figura 4.174 obtenemos:

$$F = (0{,}5,\ 1)^T.$$

◇ **Ejercicio 4.71 (sol. en pág. 1093):** Escríbanse las cuatro matrices de transición y la distribución final para el grafo de la Figura 4.173.

En el tercer paso definimos las cantidades $\psi_i(a_1 a_2 \ldots a_k)$. Como recordatorio, si el tamaño de la imagen original es de $2^n \times 2^n$, entonces la cadena $a_1 a_2 \ldots a_k$ (donde $a_i = 0, 1, 2,$ ó 3) define un subcuadrado de tamaño $2^{n-k} \times 2^{n-k}$ píxeles. La definición es:

$$\psi_i(a_1 a_2 \ldots a_k) = (W_{a1} \cdot W_{a2} \cdots W_{ak} \cdot F)_i. \tag{4.56}$$

Por lo tanto, $\psi_i(a_1a_2\ldots a_k)$ es el i-ésimo elemento de la columna $(W_{a1}\cdot W_{a2}\cdots W_{ak}\cdot F)^T$; éste es un número.

Cada matriz de transición W_a tiene como dimensiones $m\times m$ (donde m es el número de estados del grafo) y F es una columna de tamaño m. Para cualquier cadena dada $a_1a_2\ldots a_k$ hay, por lo tanto, m números $\psi_i(a_1a_2\ldots a_k)$, siendo $i=1,2,\ldots,m$. Utilizamos el grafo biestado de Figura 4.174 para calcular algunos ψ_i's :

$$\begin{aligned}
\psi_i(0) &= (W_0\cdot F)_i = \begin{pmatrix} 0{,}5 & 0\\ 0 & 1\end{pmatrix}\begin{pmatrix} 0{,}5\\ 1\end{pmatrix}_i = \begin{pmatrix} 0{,}25\\ 1\end{pmatrix}_i,\\
\psi_i(01) &= (W_0\cdot W_1\cdot F)_i = \begin{pmatrix} 0{,}5 & 0\\ 0 & 1\end{pmatrix}\begin{pmatrix} 0{,}5 & 0{,}25\\ 0 & 1\end{pmatrix}\begin{pmatrix} 0{,}5\\ 1\end{pmatrix}_i = \begin{pmatrix} 0{,}25\\ 1\end{pmatrix}_i,\\
\psi_i(1) &= (W_1\cdot F)_i = \begin{pmatrix} 0{,}5 & 0{,}25\\ 0 & 1\end{pmatrix}\begin{pmatrix} 0{,}5\\ 1\end{pmatrix}_i = \begin{pmatrix} 0{,}5\\ 1\end{pmatrix}_i,\\
\psi_i(00) &= (W_0\cdot W_0\cdot F)_i = \begin{pmatrix} 0{,}5 & 0\\ 0 & 1\end{pmatrix}\begin{pmatrix} 0{,}5 & 0\\ 0 & 1\end{pmatrix}\begin{pmatrix} 0{,}5\\ 1\end{pmatrix}_i = \begin{pmatrix} 0{,}125\\ 1\end{pmatrix}_i,\\
\psi_i(03) &= (W_0\cdot W_3\cdot F)_i = \begin{pmatrix} 0{,}5 & 0\\ 0 & 1\end{pmatrix}\begin{pmatrix} 0{,}5 & 0{,}5\\ 0 & 1\end{pmatrix}\begin{pmatrix} 0{,}5\\ 1\end{pmatrix}_i = \begin{pmatrix} 0{,}375\\ 1\end{pmatrix}_i,\\
\psi_i(33\ldots3) &= (W_3\cdot W_3\cdots W_3\cdot F)_i\\
&= \begin{pmatrix} 0{,}5 & 0{,}5\\ 0 & 1\end{pmatrix}\begin{pmatrix} 0{,}5 & 0{,}5\\ 0 & 1\end{pmatrix}\cdots\begin{pmatrix} 0{,}5 & 0{,}5\\ 0 & 1\end{pmatrix}\begin{pmatrix} 0{,}5\\ 1\end{pmatrix}_i\\
&= \begin{pmatrix} 0 & 1\\ 0 & 1\end{pmatrix}\begin{pmatrix} 0{,}5\\ 1\end{pmatrix}_i = \begin{pmatrix} 1\\ 1\end{pmatrix}_i.
\end{aligned} \tag{4.57}$$

◇ **Ejercicio 4.72 (sol. en pág. 1094):** Calcúlese ψ_i en el centro de la imagen.

Observe que cada $\psi_i(\omega)$ se identifica mediante dos cantidades, su subíndice i (que corresponde a un estado del grafo) y una cadena $\omega=a_1a_2\ldots a_k$ que especifica un subcuadrado de la imagen. El subcuadrado puede ser tan grande como la imagen completa o tan pequeña como un único píxel.

La cantidad ψ_i (donde no se especifica ningún subcuadrado) recibe el nombre de *imagen* del estado i del grafo. El nombre *imagen* tiene sentido, ya que para cada subcuadrado ω, $\psi(\omega)$ es un número, por lo que ψ_i está formado por varios números a partir de los cuáles se pueden calcular y mostrar los píxeles de la imagen del estado i. Los algoritmos de codificación del WFA descritos en esta sección generan un grafo donde cada estado es un subcuadrado de la imagen. En principio, sin embargo, algunos estados del grafo pueden no corresponderse con algún subcuadrado de la imagen. Un ejemplo es el estado 2 del grafo de la Figura 4.174. Éste es todo negro, aunque la imagen no tiene un subcuadrado completamente negro.

La Figura 4.175 muestra la imagen de la Figura 4.174 con resoluciones de 2×2, 4×4, y 256×256.

◇ **Ejercicio 4.73 (sol. en pág. 1094):** Utilícese software matemático para calcular una matriz semejante a la de la Figura 4.175.

Ahora introducimos la *distribución inicial* $I=(I_1,I_2,\ldots,I_m)$, un vector fila con m elementos. Si establecemos $I=(1,0,0,\ldots,0)$, entonces el producto escalar $I\cdot\psi(a_1a_2\ldots a_k)$ [donde la notación $\psi(q)$ representa un vector con los m valores $\psi_1(q)$ a $\psi_m(q)$] proporciona la intensidad media $f(a_1a_2\ldots a_k)$ del subcuadrado especificado por $a_1a_2\ldots a_k$. Esta función de intensidad también se puede expresar como la matriz producto:

$$f(a_1a_2\ldots a_k) = I\cdot W_{a1}\cdot W_{a2}\cdots W_{ak}\cdot F. \tag{4.58}$$

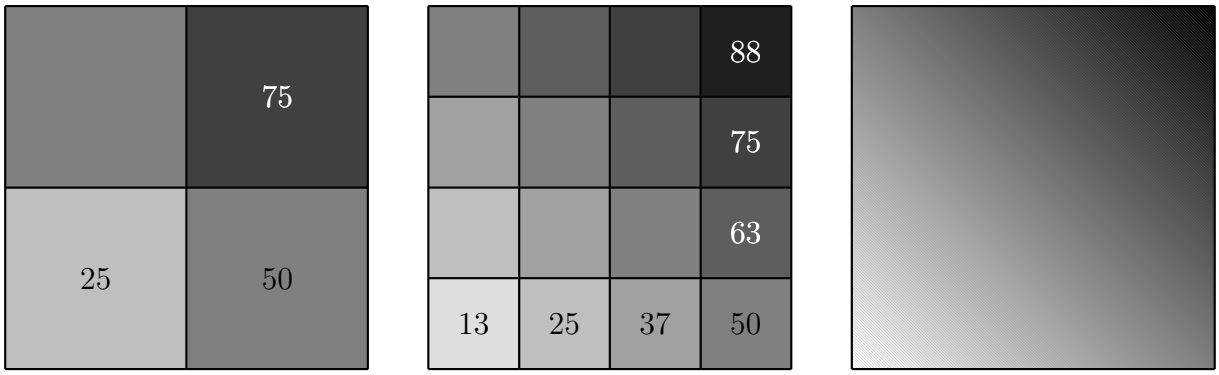


Figura 4.175: Imagen $f = (i + j)/2$ en tres resoluciones.

En general, la distribución inicial especifica la imagen definida por el WFA como una combinación lineal $I_1\psi_1 + \cdots + I_n\psi_n$ de "imágenes de estado". Si $I = (1, 0, 0, \ldots, 0)$, entonces la imagen definida es ψ_1, la imagen correspondiente al primer estado. Éste es siempre el caso para la imagen resultante del algoritmo de inferencia que se describe posteriormente en esta sección.

Dado un WFA, podemos fácilmente encontrar otro WFA para la imagen obtenida mediante el zoom del subcuadrado con la dirección $a_1a_2 \ldots a_k$. Solamente tenemos que reemplazar la distribución inicial I por $I \cdot W_{a1} \cdot W_{a2} \cdots W_{ak}$.

Para probar ésto, considere el subcuadrado con dirección $b_1b_2 \ldots b_m$ en el cuadrado ampliado. Se corresponde con el subcuadrado de la imagen completa con la dirección $a_1a_2 \ldots a_kb_1b_2 \ldots b_m$. Por consiguiente, el valor de gris calculado para que el WFA original es:

$$IW_{a1}W_{a2} \ldots W_{ak}W_{b1}W_{b2} \ldots W_{bm}F.$$

Utilizando el nuevo WFA para el subcuadrado correspondiente, obtenemos el mismo valor, a saber, $I'W_{b1}W_{b2} \ldots W_{bm}F$, donde $I' = W_{a1}W_{a2} \ldots W_{ak}$ (demostración proporcionada por Karel Culik).

Para los ψ_i's calculados en la Ecuación (4.57) los productos escalares de la forma $I\psi(a_1a_2 \ldots a_k)$ producen:

$$\begin{aligned}
f(0) &= I \cdot \psi(0) = (1, 0) \begin{pmatrix} 0{,}25 \\ 1 \end{pmatrix} = I_1\psi_1(0) + I_2\psi_2(0) = 0{,}25 \\
f(01) &= I \cdot \psi(01) = 1 \times 0{,}25 + 0 \times 1 = 0{,}25, \\
f(1) &= I \cdot \psi(1) = 0{,}5, \\
f(00) &= I \cdot \psi(00) = 0{,}125, \\
f(03) &= I \cdot \psi(03) = 0{,}375, \\
f(33 \ldots 3) &= I \cdot \psi(33 \ldots 3) = 1.
\end{aligned}$$

◇ **Ejercicio 4.74 (sol. en pág. 1094):** Calcúlense los ψ_i's y los valores f correspondientes para los subcuadrados 0, 01, 1, 00, 03, y 3 del grafo de cinco estados de la Figura 4.173.

La Ecuación (4.56) es la definición de $\psi_i(a_1a_2 \ldots a_k)$. Ella muestra que esta cantidad es el i-ésimo elemento del vector columna $(W_{a1} \cdot W_{a2} \cdots W_{ak} \cdot F)^T$. Ahora examinamos el vector columna reducido $(W_{a2} \cdots W_{ak} \cdot F)^T$. Su i-ésimo elemento es, de acuerdo con la definición de ψ_i, la cantidad $\psi_i(a_1a_2 \ldots a_k)$. Por consiguiente, concluimos que:

$$\psi_i(a_1a_2 \ldots a_k) = (W_{a1})_{i,1}\,\psi_1(a_2 \ldots a_k) + (W_{a1})_{i,2}\,\psi_2(a_2 \ldots a_k) + \cdots + (W_{a1})_{i,m}\,\psi_m(a_2 \ldots a_k),$$

o, si denotamos la cadena $a_2 \ldots a_k$ por ω,

$$\begin{aligned}\psi_i(a_1\omega) &= (W_{a1})_{i,1}\,\psi_1(\omega) + (W_{a1})_{i,2}\,\psi_2(\omega) + \cdots + (W_{a1})_{i,m}\,\psi_m(\omega)\\ &= \sum_{j=1}^{m} (W_{a1})_{i,j}\,\psi_j(\omega).\end{aligned} \tag{4.59}$$

La cantidad $\psi_i(a_1\omega)$ (que corresponde al cuadrante a_1 del subcuadrado ω) puede expresarse como una combinación lineal de las cantidades $\psi_j(\omega)$, donde $j = 1, 2, \ldots, m$. Ésto justifica llamar a ψ_i la *imagen* del estado i del grafo. Decimos que el subcuadrado a_1 de la imagen $\psi_i(\omega)$ puede expresarse como una combinación lineal de las imágenes $\psi_j(\omega)$, para $j = 1, 2, \ldots, m$. Así es cómo entra en escena la autosimilitud de la imagen original.

La Ecuación (4.59) es recursiva, ya que define una imagen más pequeña en términos de las de mayor tamaño. La imagen más grande es, por supuesto, la imagen original, para la cual ω es nulo (denotamos la cadena nul, ϵ). Aquí es donde comienza la recursión:

$$\begin{aligned}\psi_i(a) &= (W_a)_{i,1}\,\psi_1(\epsilon) + (W_a)_{i,2}\,\psi_2(\epsilon) + \cdots + (W_a)_{i,m}\,\psi_m(\epsilon)\\ &= \sum_{j=1}^{m} (W_a)_{i,j}\,\psi_j(\epsilon), \quad \text{para } a = 0, 1, 2, 3.\end{aligned} \tag{4.60}$$

Por otra parte, la Ecuación (4.56) muestra que $\psi_i(\epsilon) = F_i$, por lo que la Ecuación (4.60) se convierte en:

$$\psi_i(a) = \sum_{j=1}^{m} (W_a)_{i,j}\,F_j, \quad \text{para } a = 0, 1, 2, 3. \tag{4.61}$$

Ahora debería quedar claro que si conocemos las cuatro matrices de transición y la distribución final, podemos calcular las imágenes $\psi_i(\omega)$ para cadenas ω de cualquier longitud (i.e., subcuadrados de cualquier tamaño). Una vez que se conoce una imagen $\psi_i(\omega)$, la intensidad media $f(\omega)$ del subcuadrado ω puede calcularse mediante la Ecuación (4.58) (que requiere conocer la distribución inicial).

El problema de la representación de una imagen f en el WFA se reduce así a la búsqueda de vectores I y F y matrices W_0, W_1, W_2, y W_3 que generen f (o una imagen cercana a la misma). Alternativamente, podemos encontrar I y las imágenes $\psi_i(\epsilon)$ para $i = 1, 2, \ldots, m$.

Aquí hay un acercamiento informal a este problema. Podemos empezar con un grafo formado por un estado y seleccionar este estado como la imagen completa, $\psi_1(\epsilon)$. Ahora nos concentramos en el cuadrante 0 de la imagen y tratamos de determinar $\psi_1(0)$. Si el cuadrante 0 es idéntico, o bastante similar, a (una versión escalada de) la imagen completa, podemos escribir:

$$\psi_1(0) = \psi_1(\epsilon) = \sum_{j=1}^{m} (W_0)_{1,j}\,\psi_j(\epsilon),$$

lo cual es cierto si la primera fila de W_0 es $(1, 0, 0, \ldots, 0)$. Hemos determinado la primera fila de W_0, a pesar de que aún no conocemos su tamaño (es m, pero m no se ha determinado todavía). Si cuadrante 0 es sustancialmente diferente de la imagen completa, añadimos $\psi_1(0)$ al grafo como un nuevo estado, el estado 2 (i.e., incrementamos m en 1) y lo denominamos $\psi_2(0)$. El resultado es:

$$\psi_1(0) = \psi_2(0) = \sum_{j=1}^{m} (W_0)_{1,j}\,\psi_j(0),$$

lo cual es cierto si la primera fila de W_0 es $(0, 1, 0, \ldots, 0)$. Hemos determinado de nuevo la primera fila de W_0, a pesar de que aún no conocemos su tamaño.

A continuación, procesamos los tres cuadrantes restantes de $\psi_1(\epsilon)$ y los cuatro cuadrantes de $\psi_2(0)$. Vamos a examinar, por ejemplo, el procesamiento del cuadrante tres $[\psi_2(3)]$ de $\psi_2(0)$. Si podemos expresarlo (de forma precisa o lo suficientemente próximo) como la combinación lineal:

$$\psi_2(03) = \alpha\psi_1(3) + \beta\psi_2(3) = \sum_j (W_0)_{2,j}\,\psi_j(3),$$

entonces sabemos que la segunda fila de W_0 debe ser $(\alpha, \beta, 0, \ldots, 0)$. Si no podemos expresar $\psi_2(03)$ como una combinación lineal de los ψ's ya conocidos, entonces declaramos un nuevo estado $\psi_3(03)$, y esto implica $(W_3)_{2,3} = 1$, y también determina que la segunda fila de W_0 sea $(0, 0, 1, 0, \ldots, 0)$.

Este proceso continúa hasta que todos los cuadrantes de todos las ψ_j's se hayan procesado. Ésto normalmente genera muchas más ψ_i's, todas ellas son subcuadrados de la imagen.

Este intuitivo algoritmo se describe ahora con más precisión, usando pseudocódigo. Se construye un grafo de una imagen multirresolución f, un estado cada vez. El grafo tiene el número mínimo de estados (que se denota por m), pero un número relativamente grande de aristas. Como m es mínimo, las cuatro matrices de transición (cuyas dimensiones son $m \times m$) son pequeñas. Sin embargo, dado que el número de aristas es grande, la mayor parte de los elementos de estas matrices son distintos de cero. La imagen se comprime escribiendo las matrices de transición (en formato comprimido) en el *stream* comprimido, por lo que cuanto menos densas sean las matrices, mejor será la compresión. Por consiguiente, este algoritmo no produce una buena compresión y se describe aquí debido a su simplicidad. Denotamos i al índice del primer estado sin procesar, y γ, a una correspondencia (mapeo) de los estados a los subcuadrados. Los pasos de este algoritmo son los siguientes:

- *Paso 1*: Establecer $m = 1$, $i = 1$, $F(q_1) = f(\epsilon)$, $\gamma(q_1) = \epsilon$.
- *Paso 2*: Procesar q_i, i.e., para $\omega = \gamma(q_i)$ y $a = 0, 1, 2, 3$, hacer:
  - *Paso 2a*: Comenzar con $\psi_j = f_{\gamma(q_j)}$ para $j = 1, \ldots, m$ y tratar de encontrar los números reales $c_1, \ldots, c_m$ tales que $f_{\omega a} = c_1\psi_1 + \cdots + c_m\psi_m$. Si se localizan estos números, se convierten en los m elementos de la fila q_i de la matriz de transición W_a, es decir, $W_a(q_i, q_j) = c_j$ para $j = 1, \ldots m$.
  - *Paso 2b*: Si dichos números no se pueden encontrar, se incrementa el número de estados $m = m + 1$, y se establece $\gamma(q_m) = \omega a$, $F(q_m) = f(\omega a)$ (donde F es la distribución final), y $W_a(q_i, q_m) = 1$.
- *Paso 3*: Incrementar el índice del siguiente estado sin procesar $i = i + 1$. Si $i < m$, ir al Paso 2.
- *Paso 4*: El paso final. Construir la distribución inicial I estableciendo $I(q_1) = 1$ e $I(q_j) = 0$ para $j = 2, 3, \ldots, m$.

[Litow y Olivier 95] presentan otro algoritmo de inferencia que también obtiene un WFA de estado mínimo y utiliza sólo adiciones y productos internos.

El verdadero avance en la compresión WFA se produjo cuando se desarrolló un algoritmo de inferencia mejor. Este algoritmo es la parte más importante del método WFA. Genera un grafo que puede tener más que el número mínimo de estados, pero que tiene un pequeño número de aristas. Las cuatro matrices de transición pueden ser grandes, pero son dispersas, produciendo una mejor compresión de las matrices y, por lo tanto, una mejor compresión de la imagen. El algoritmo comienza con una imagen dada A de resolución finita, y genera su grafo emparejando los subcuadrados de la imagen con la imagen completa o con otros subcuadrados. Un punto importante es que este algoritmo puede ser con pérdidas, con la cantidad de pérdidas controlada por un parámetro G definido por el usuario. Los valores más altos de G "permiten" al algoritmo emparejar dos partes de la imagen, incluso

function $make\_\omega\, f\, a\,(i, k, max)$;
If $max < 0$, return ∞;
$cost \leftarrow 0$;
if $k = 0$, $cost \leftarrow d_0\,(f, 0)$
else hacer pasos 1−5 con $\psi = (\psi_i)_a$ para $a = 0, 1, 2, 3$;

1. Encontrar $r_1, r_2, \ldots r_m$ tales que el valor de

$$cost1 \leftarrow d_{k-1}\,(\psi, r_1\psi_1 + \cdots + r_n\psi_n) + G \cdot s$$

   es pequeño, donde s indica el incremento de tamaño del grafo causado por la adición de aristas del estado i a los estados j con pesos r_j distintos de cero y etiqueta a, y d_{k-1} denota la distancia entre dos imágenes multirresolución en el nivel $k-1$ (un buen bocado).

2. Establecer $m_0 \leftarrow m$, $m \leftarrow m+1$, $\psi_m \leftarrow \psi$ y añadir una arista con etiqueta a y peso 1 del estado i al nuevo estado m. Actualizar s indicando el aumento en tamaño del grafo causado por el nuevo estado y la nueva arista.

3. Establecer $cost2 \leftarrow G \cdot s + make\_\omega\, f\, a\,(m, k-1, min\,(max - cost, cost1) - G \cdot s)$;

4. If $cost2 \leq cost1$, establecer $cost \leftarrow cost + cost2$;

5. If $cost1 < cost2$, establecer $cost \leftarrow cost + cost1$; eliminar todas las aristas salientes de los estados $m_0 + 1, \ldots, m$ (adicionados durante la llamada recursiva), así como la arista desde el estado i añadida en el paso 2. Establecer $m \leftarrow m_0$ y agregar las aristas del estado i con etiqueta a a los estados $j = 1, 2, \ldots, m$ con pesos r_j cuando $r_j \neq 0$.

If $cost \leq max$, return($cost$), else return(∞).

Figura 4.176: El algoritmo de inferencia WFA recursivo.

si se parecen pobremente entre sí. Ésto naturalmente conduce a una mejor compresión. La métrica utilizada por el algoritmo para emparejar dos imágenes (o partes de la imagen) f y g es el cuadrado de la métrica L2, una medida común donde la distancia $d_k\,(f, g)$ se define como:

$$d_k\,(f, g) = \sum_{\omega} [f\,(\omega) - g\,(\omega)]^2\,,$$

donde la suma es sobre todos los subcuadrados ω.

El algoritmo trata de producir un grafo pequeño. Si denotamos el tamaño del grafo (tamaño de A), el algoritmo intenta mantener lo más pequeño posible el valor de:

$$d_k\,(f, f_A) + G \cdot (\text{tamaño de } A)\,.$$

La cantidad m indica el número de estados, y existe una imagen multirresolución ψ_i para cada estado $i = 1, 2, \ldots, m$. El pseudocódigo de la Figura 4.176 describe una función recursiva $make\_w\, f\, a(i, k, max)$ que trata de aproximar ψ_i al nivel k lo mejor posible añadiendo nuevas aristas y (posiblemente) nuevos estados al grafo. La función minimiza el valor de la cantidad cost:

$$cost = d_k\,(\psi_i, \psi_i') + G \cdot s,$$

donde ψ_i' es la aproximación actual a ψ_i y s es el aumento en tamaño del grafo causado por la adición de nuevas aristas y estados. Si $cost > max$, la función devuelve ∞, de lo contrario, devuelve $cost$.

Cuando el algoritmo comienza, m se establece a 1 y ψ a f, donde f es la función que necesita ser aproximada al nivel k. El algoritmo entonces ejecuta $make\_wfa(1, k, \infty)$, que se llama a sí misma. Para cada uno de los cuatro cuadrantes, la llamada recursiva $make\_wfa(i, k, max)$ intenta aproximar $(\psi_i)_a$ para $a = 0, 1, 2, 3$ de dos maneras diferentes: como una combinación lineal de las funciones de los estados existentes (paso 1), y mediante la adición de un nuevo estado y de forma recursiva llamarse a sí mismo (pasos 2 y 3). El mejor de los dos resultados es entonces seleccionado (en los pasos 4 y 5).

El algoritmo construye una distribución inicial $I = (1, 0, \ldots, 0)$ y una distribución final $F_i = \psi_i(\epsilon)$ para todos los estados i.

WFA es un acrónimo común de, e.g., "World Fellowship Activities."

A continuación, discutimos cómo se pueden comprimir los elementos del grafo. El Paso 2 crea una arista cada vez que un nuevo estado es añadido al grafo. Ellas forman un árbol, y sus pesos son siempre 1, por lo que cada arista puede ser codificada en cuatro bits. Cada uno de los cuatro bits indica cuál de las dos alternativas fue seleccionada para la etiqueta en los pasos 4−5.

El Paso 1 crea aristas que representan combinaciones lineales (i.e., los casos donde las subimágenes se expresan como combinaciones lineales de otras subimágenes). Ambos, el peso y los criterios de valoración, necesitan ser almacenados. Los experimentos indican que los pesos siguen una distribución normal, por lo que pueden ser codificados eficientemente con códigos prefijo. El almacenamiento de los puntos extremos de las aristas equivale a almacenar cuatro matrices binarias dispersas, por tanto, se puede utilizar la codificación run length.

El decodificador WFA lee los vectores I y F y las cuatro matrices de transición W_a desde el *stream* comprimido y los descomprime. Su principal tarea consiste en utilizarlos para reconstruir la imagen original A de $2^n \times 2^n$ (de forma precisa o aproximadamente), i.e., para calcular $f_A(\omega)$ para todas las cadenas $\omega = a_1 a_2 \ldots a_n$ de tamaño n. El algoritmo de decodificación WFA original es rápido, pero tiene unos requisitos de almacenamiento del orden de $m4^n$, donde m es el número de estados del grafo. El algoritmo está formado por las cuatro etapas siguientes:

- *Paso 1*: Establecer $\psi_p(\epsilon) = F(p)$ para todo $p \in Q$.
- *Paso 2*: Repetir el Paso 3 para $i = 1, 2, \ldots, n$.
- *Paso 3*: Para todo $p \in Q$, $\omega = a_1 a_2 \ldots a_{i-1}$, y $a = 0, 1, 2, 3$,calcular:

$$\psi_p(a\omega) = \sum_{q \in Q} W_a(p, q) \cdot \psi_q(\omega).$$

- *Paso 4*: Para cada $\omega = a_1 a_2 \ldots a_{i-1}$ calcular:

$$f_A(\omega) = \sum_{q \in Q} I(q) \cdot \psi_q(\omega).$$

Este algoritmo de decodificación fue mejorado por Raghavendra Udupa, Vinayaka Pandit, y Ashok Rao [Udupa et al. 99]. Ellos definen una nueva función multirresolución Φ_p para cada estado p del grafo WFA mediante:

$$\begin{aligned} \Phi_p(\epsilon) &= I(p), \\ \Phi_p(\omega a) &= \sum_{q \in Q} (W_a)_{q,p} \Phi_q(\omega), \quad \text{para } a = 0, 1, 2, 3, \end{aligned}$$

o equivalentemente:

$$\Phi_p(a_1 a_2 \ldots a_k) = (I W_{a1} \ldots W_{ak})_p,$$

donde $\Phi_p(\omega a)$ es la suma la suma de los pesos de todos los caminos con la etiqueta ωa que terminan en p. El peso de una ruta que comienza en el estado i es el producto de la distribución inicial de i y los pesos de las aristas a lo largo del camino ωa.

Un resultado de esta definición es que la función de intensidad f puede expresarse como una combinación lineal de la nuevas funciones multirresolución Φ_p,

$$f(\omega) = \sum_{q \in Q} \Phi_p(\omega) F(q),$$

¡pero hay una observación más importante! Supongamos que tanto $\Phi_p(u)$ y $\psi_p(\omega)$ son conocidos por todos los estados $p \in Q$ del grafo para todas las cadenas u y ω. Entonces la función intensidad del subcuadrado $u\omega$ se puede expresar como:

$$f(u\omega) = \sum_{p \in Q} \Phi_p(u) \psi_p(\omega).$$

Esta observación se utiliza en la mejora del paso 6 del algoritmo que viene a continuación para reducir tanto la requisitos de espacio como de tiempo. La entrada al algoritmo es un WFA A con una resolución de $2^n \times 2^n$, y la salida es una función de intensidad $f_A(\omega)$ para cada cadena ω hasta la longitud n.

- *Paso 1*: Seleccionar los enteros no negativos $n1$ y $n2$ que satisfagan $n1 + n2 = n$. Establecer $\Phi_p = I(p)$ y $\psi_p = F(p)$ para todo $p \in Q$.
- *Paso 2*: Para $i = 1, 2, \ldots, n1$ efectuar el Paso 3.
- *Paso 3*: Para todo $p \in Q$ y $\omega = a_1 a_2 \ldots a_{i-1}$ y $a = 0, 1, 2, 3$, calcular:

$$\Phi_p(a\omega) = \sum_{q \in Q} (W_a)_{q,p} \Phi_q(\omega).$$

- *Paso 4*: Para $i = 1, 2, \ldots, n2$ efectuar el Paso 5.
- *Paso 5*: Para todo $p \in Q$ y $\omega = a_1 a_2 \ldots a_{i-1}$ y $a = 0, 1, 2, 3$, calcular:

$$\psi_p(a\omega) = \sum_{q \in Q} (W_a)_{p,q} \psi_q(\omega).$$

- *Paso 6*: Para cada $u = a_1 a_2 \ldots a_{n1}$ y $\omega = b_1 b_2 \ldots b_{n2}$, calcular:

$$f_A(u\omega) = \sum_{q \in Q} \Phi_q(u) \psi_q(\omega).$$

El algoritmo de decodificación WFA original es un caso especial del algoritmo anterior para $n1 = 0$ y $n2 = n$. Para el caso en que $n = 2l$ sea un número par, se puede demostrar que la necesidad de espacio de este algoritmo es del orden de $m4^{n1} + m4^{n2}$. Esta expresión tiene un mínimo cuando $n1 = n2 = n/2 = l$, lo que implica un requisito de espacio del orden de $m4^l$.

Otra mejora del WFA original es el uso de *bintrees* (Sección 4.30.1). Esta idea se debe a Ullrich Hafner [Hafner 95]. Recordemos que WFA utiliza los métodos de quadtree para la partición de la imagen en un conjunto de subcuadrados no superpuestos. Estos pueden ser llamados (en común con la notación utilizada por IFS) *imágenes de rango*. Cada imagen de rango es entonces emparejada, de forma precisa o aproximadamente, a una combinación lineal de otras imágenes que se convierten las *imágenes de dominio*. El uso de los métodos de bintree para la división de la imagen produce un

particionamiento más fino y un aumento en el número de imágenes de dominio disponibles para el emparejamiento. Además, la compresión de las matrices de transición y de las distribuciones inicial y final se mejora. Cada nodo en un bintree tiene dos hijos en comparación con los cuatro hijos en un quadtree. Una subimagen, por lo tanto, puede ser identificada mediante una cadena de bits en lugar de una cadena compuesta por los dígitos 0−3. Ésto también significa que existen sólo dos matrices de transición W_0 y W_1.

La compresión WFA funciona particularmente bien para imágenes en color, ya que se construye un único grafo WFA para los tres componentes de color de cada píxel. Cada componente tiene su propio estado inicial, pero otros estados son compartidos. Normalmente, esto ahorra muchos estados porque los componentes de color de los píxeles en las imágenes reales no son independientes. La experiencia indica que la compresión WFA de una imagen típica puede crear, digamos, 300 estados para el componente de color Y, 40 estados para I, y sólo unos 5 estados para el componente Q. Otra característica de la compresión WFA es que es relativamente lenta para la compresión de alta calidad, ya que construye un gran autómata, pero es más rápido para baja calidad de compresión, ya que el autómata construido en tal caso es pequeño.

4.34.2. Autómata finito generalizado

Los grafos construidos y utilizados por un WFA representan autómatas de estados finitos con "entradas ponderadas". Sin pesos, los autómatas finitos especifican imágenes binivel multirresolución. En la resolución $2^k \times 2^k$, un autómata finito especifica una imagen que es negra en aquellos subcuadrados cuyas direcciones son aceptadas por el autómata. Es bien conocido que cada autómata no determinista puede convertirse en uno determinista equivalente. Ésto no es ventajoso para el WFA, donde el no determinismo da mucho más poder. Esta es la razón por la cual una imagen en escala de grises variando suavemente, tal como la representada en la Figura 4.174, puede ser representada por un sencillo autómata de dos estados. También por ello, un WFA puede comprimir eficientemente una gran variedad de imágenes en color y en escala de grises.

Para las imágenes binivel, la situación es diferente. Aquí el no determinismo podría proporcionar una descripción más concisa. Sin embargo, los experimentos muestran que un autómata no determinista no mejora la compresión de tales imágenes en mucho, y es más lento de construir que uno determinista. Ésta es la razón por la que el método de los *autómatas finitos generalizados* (GFA) fue desarrollado originalmente ([Culik y Valenta 96] y [Culik y Valenta 97a,b]). Nosotros también mostramos cómo se extiende para imágenes en color. El algoritmo para generar el grafo GFA es completamente diferente del utilizado por WFA. En particular, no es recursivo (en contraste con el algoritmo WFA de optimización de aristas de la Figura 4.176, que es recursivo). El GFA también utiliza tres tipos de transformaciones —rotaciones, saltos, y complementación—, para emparejar partes de la imagen.

Un grafo del GFA se compone de estados y aristas que los conectan. Cada estado representa un subcuadrado de la imagen, con el primer estado representando la imagen completa. Si el cuadrante q de la imagen representada por el estado i es idéntico (hasta un factor de escala) a la imagen representada por el estado j, entonces se construye una arista de i a j y se etiqueta q. De forma más general, si el cuadrante q del estado i se puede hacer idéntico al estado j mediante la aplicación de la transformación t a la imagen de j, entonces se construye una arista desde i hasta j y se etiqueta q, t o (q, t). Existen 16 transformaciones, mostradas en la Figura 4.178. La transformación 0 es la identidad, por lo que puede omitirse cuando una arista es etiquetada. Las transformaciones 1−3 son rotaciones de 90°, y las transformaciones 4−7 son rotaciones de una reflexión de la transformación 0. Las transformaciones 8−15 son los videos inversos de 0−7, respectivamente. Cada transformación t se especifica, por lo tanto mediante un número de 4 bits.

La Figura 4.179 muestra una sencilla imagen binivel y su grafo del GFA. El cuadrante 3 del estado 0, por ejemplo, tiene que ir a través de la transformación 1 para que sea idéntico al estado 1, por lo que existe una arista etiquetada $(3, 1)$ del estado 0 al estado 1.

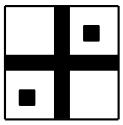


Figura 4.177: Imagen para el Ejercicio 4.75.

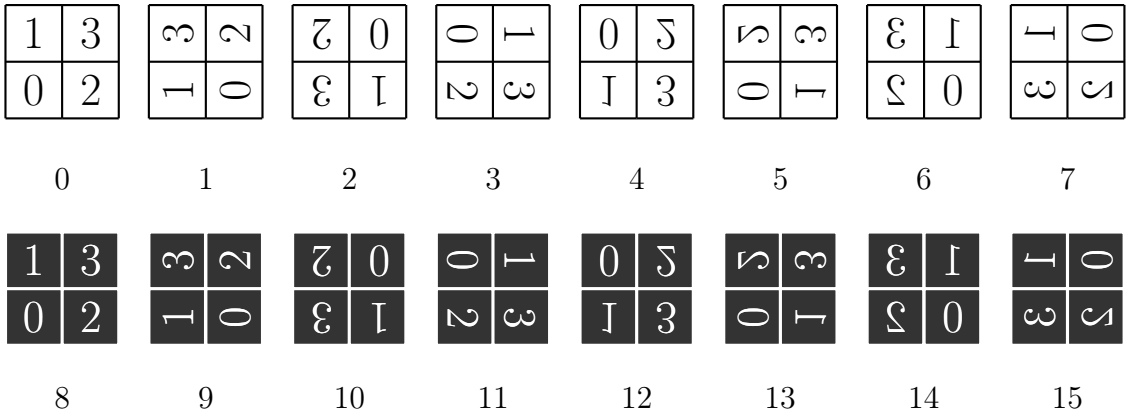


Figura 4.178: Dieciséis transformaciones de imágenes.

◇ **Ejercicio 4.75 (sol. en pág. 1095):** Constrúyase el GFA de la imagen de la Figura 4.177 usando transformaciones, y la lista de todas las aristas resultantes en una tabla.

Las imágenes utilizadas en la práctica son más complejas y menos autosimilares que los ejemplos que se muestran aquí, por lo que el algoritmo del GFA particular discutido aquí (aunque no es GFA en general) permite una cierta cantidad de pérdidas de datos en el emparejamiento de los subcuadrados de la imagen. La distancia $d_k(f,g)$ entre dos imágenes o subimágenes f y g de resolución $2^k \times 2^k$ se define como:

$$d_k(f,g) = \frac{\sum_{\omega=a1a2\ldots ak} |f(\omega) - g(\omega)|}{2^k \times 2^k}.$$

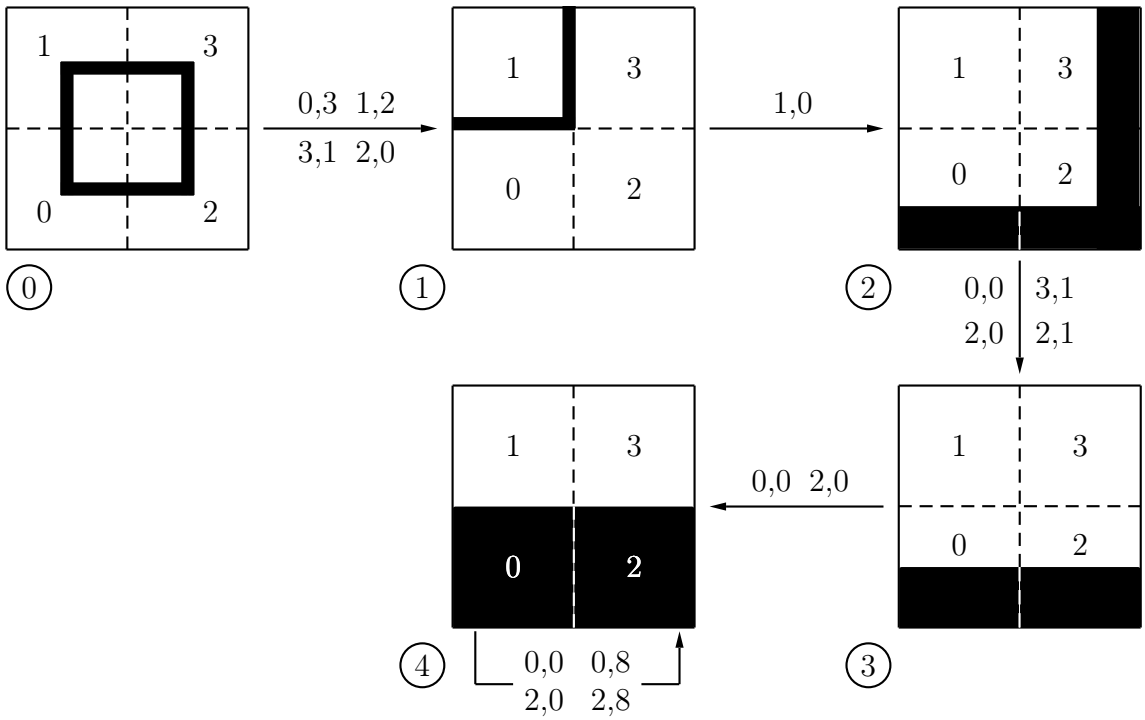
El numerador de esta expresión cuenta el número de píxeles que difieren en las dos imágenes, mientras que el denominador es el número total de píxeles en cada imagen. La distancia es, por lo tanto, el porcentaje de píxeles que difieren en las dos imágenes (suponemos que 0 y 1 representan los píxeles en blanco y negro, respectivamente). El algoritmo de cuatro pasos aquí descrito para la construcción del grafo también utiliza un parámetro de error introducido por el usuario para controlar la cantidad de pérdidas.

- *Paso 1*: Se genera el estado 0 como la imagen completa (i.e., el subcuadrado ϵ). Seleccionamos la distribución inicial como $I = (1, 0, 0, \ldots, 0)$, por lo que el estado 0 será el único que se muestra. La distribución final es un vector de todo unos.

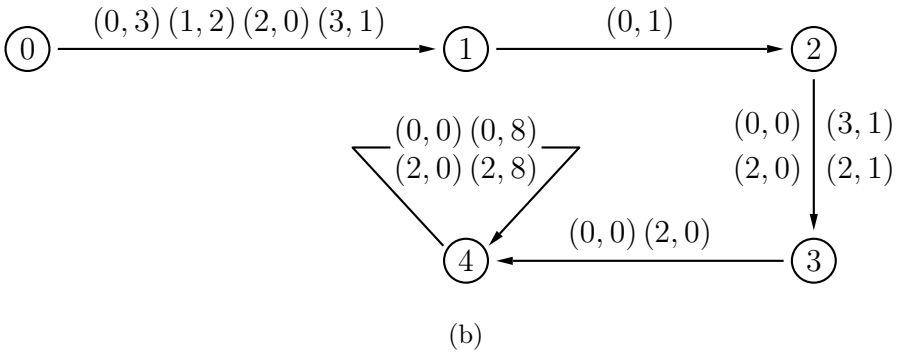
- *Paso 2*: Se procesa cada estado de la siguiente manera: permitimos que el siguiente estado sin procesar sea q, representando el subcuadrado ω. Se divide ω en sus cuatro cuadrantes $\omega 0$, $\omega 1$, $\omega 2$, y $\omega 3$, y se efectúa el Paso 3 para cada uno de los cuatro ωa's.

- *Paso 3*: Denotamos la imagen del subcuadrado ωa por ψ'. Si $\psi' = 0$, no hay ninguna arista desde el estado q con etiqueta a. De lo contrario, examinamos todos los estados generados hasta el momento, y tratamos de encontrar un estado p y una transformación t tal que $t(\psi_p)$ (la

imagen del estado p transformada mediante t) sea bastante similar a la imagen ψ'. Ésto se expresa $d_k(\psi', t(\psi_p)) \leq$ error. Si se encuentran tales p y t, se construye una arista $q(a,t)p$. En caso contrario, se añade un nuevo estado r, sin procesar, a ψ' y se construye una nueva arista $q(a,0)r$.



(a)



(b)

Figura 4.179: Un GFA de cinco estados.

- *Paso 4*: Si existe un estado cualquiera sin procesar, se va al Paso 2. De lo contrario se detiene el proceso.

El paso 3 puede implicar muchas búsquedas, ya que es posible que muchos estados deban ser exa-

minados hasta 16 veces hasta que se encuentre una coincidencia. Por lo tanto, el algoritmo utiliza dos versiones para esta búsqueda, un *primer ajuste* (*first fit*) y un *mejor ajuste* (*best fit*). Ambos enfoques avanzan de un estado a otro, aplicando cada una de las 16 transformaciones a los estados, y calculando la distancia entre cada estado transformado y ψ'. El enfoque del primer ajuste, se detiene cuando encuentra el primer estado transformado que ajuste, mientras que el mejor ajuste lleva a cabo la búsqueda completa y selecciona el estado y la transformación que producen el mejor ajuste. El primero es, por lo tanto, más rápido, mientras que el último produce una mejor compresión.

El grafo del GFA para una imagen compleja real puede tener un gran número de estados, de modo el algoritmo GFA discutido aquí tiene una opción donde se utiliza la cuantificación vectorial. Esta opción reduce el número de estados (y, por lo tanto, acelera el proceso de construcción del grafo), tratando los subcuadrados de tamaño 8×8 de manera diferente. Cuando esta opción está seleccionada, el algoritmo no intenta encontrar un estado similar a un subcuadrado pequeño, sino que codifica el subcuadrado mediante cuantificación vectorial (Sección 4.14). Un subcuadrado de 8×8 consta de 64 píxeles, y el algoritmo utiliza un libro de códigos de 256 entradas para codificarlos. Cabe señalar que, en general, el GFA no utiliza una cuantificación o algún libro de códigos específico. La opción de cuantificación vectorial es específica de la aplicación discutida aquí.

Después de construir el grafo del GFA, sus componentes son compridos y escritos en la secuencia de datos comprimidos en tres partes. La primera parte es la información de las aristas creada en el paso 3 del algoritmo. La segunda parte es la información de los estados (los índices de la estados), y la tercera parte son las 256 palabras de código utilizadas en la cuantificación vectorial de los pequeños subcuadrados. Las tres partes son codificadas aritméticamente.

La decodificación del GFA se realiza como en el WFA, excepto en que el decodificador tiene que considerar las posibles transformaciones cuando genera nuevas partes de la imagen a partir de las ya existentes.

El GFA se ha extendido a las imágenes en color. La imagen se separa en planos de bits individuales y se construye un grafo para cada uno. Sin embargo, los estados comunes en estos grafos se almacenan sólo una vez, mejorando así la compresión. Pensando un poco se ve que ésto funciona mejor para imágenes en color formadas por varias áreas con límites bien definidos (i.e., imágenes de tonos discretos tonos, o imágenes *cartoon-like*[21]). Cuando dicha imagen se separa en sus planos de bits, estos tienden a ser similares. Un subcuadrado q en el plano de bit 1, por ejemplo, puede ser idéntico al mismo subcuadrado en, digamos, el plano de bit 3. Los gráficos para estos planos de bits pueden así compartir el estado para el subcuadrado q.

El algoritmo del GFA se aplica entonces a los planos de bit, uno por uno. Cuando se trabaja en el plano de bit b, El Paso 3 del algoritmo busca a través de todos los estados de todos los grafos construidos hasta el momento para los $b-1$ planos de bits anteriores. Este proceso puede ser visto en dos maneras diferentes. El algoritmo puede construir n grafos, uno para cada plano de bits, y compartir estados entre ellos. Alternativamente, se puede construir únicamente un grafo con n estados iniciales.

Los experimentos muestran que el GFA funciona mejor para imágenes con un número reducido de colores. Dada una imagen con muchos colores, la cuantificación se utiliza para reducir el número de colores.

4.35. Sistemas de funciones iteradas

Los fractales han sido populares desde la década de 1970 y tienen muchas aplicaciones (véase [Demko et al. 85], [Feder 88], [Mandelbrot 82], [Peitgen et al. 82], [Peitgen y Saupe 85], y [Reghbati 81] como ejemplos). Una de estas aplicaciones, relativamente infrautilizada, es la compresión de datos. La aplicación de los fractales a la compresión de datos se efectúa por medio de *sistemas de funciones*

[21] Dibujos animados, caricaturas, etc.

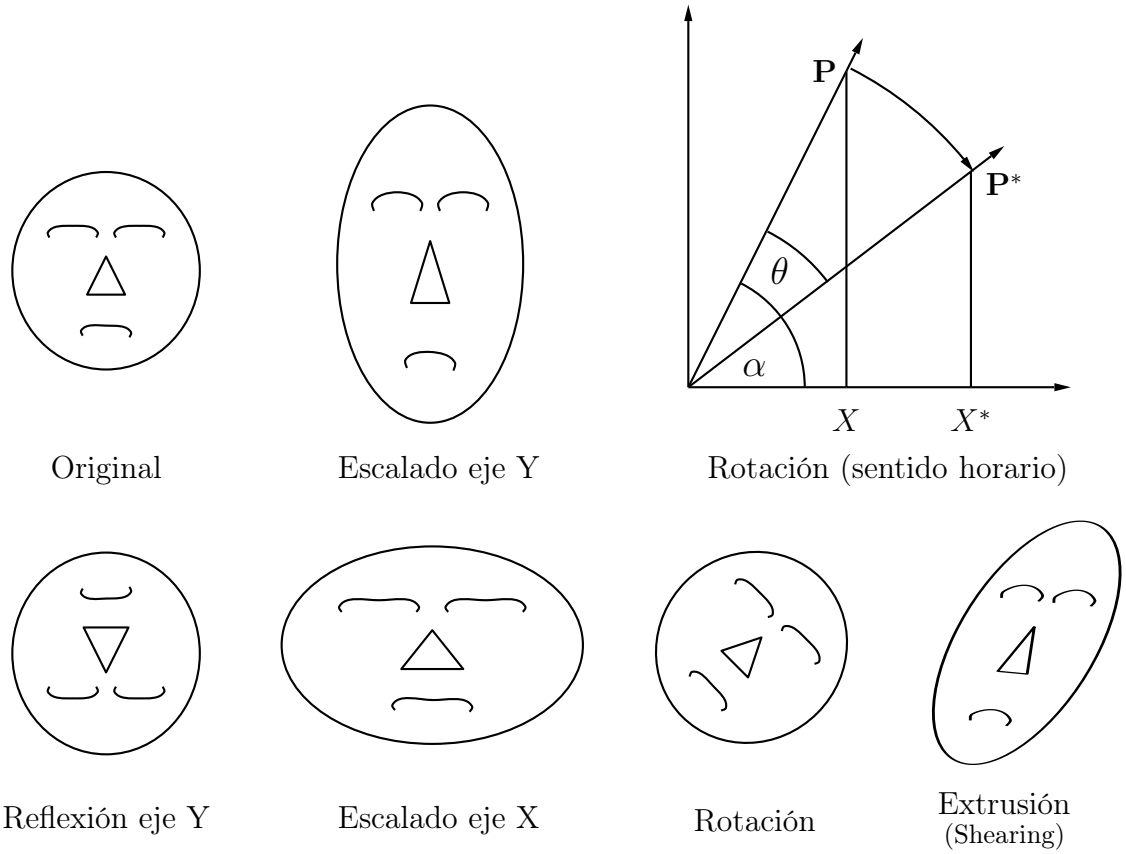


Figura 4.180: Transformaciones bidimensionales.

iteradas, o IFS. Dos referencias a IFS son: [Barnsley 88] y [Fisher 95]. La compresión IFS puede ser muy eficiente, lograr excelentes factores de compresión (32 no es raro), pero es con pérdidas y también computacionalmente intensiva. El codificador IFS divide la imagen en partes llamadas rangos; luego empareja cada rango a alguna otra parte denominada dominio, y produce una *transformación afín* del dominio al rango. Las transformaciones se escriben en el *stream* comprimido, y ellas constituyen la imagen comprimida. Empezamos con una introducción a las transformaciones afines bidimensionales.

4.35.1. Transformaciones afines

En los gráficos por ordenador, una imagen bidimensional completa se construye parte por parte y es normalmente editada antes de que se considere satisfactoria. La edición se realiza seleccionando una figura (parte del dibujo) y aplicando una transformación a la misma. Transformaciones típicas (Figura 4.180) son: movimiento o deslizamiento (traslación), reflexión o volteo (imagen especular), zoom (escalado), rotación, y extrusiones (*shearing*).

La transformación puede aplicarse a todos los píxeles de la figura. Alternativamente, se pueden aplicar a algunos puntos clave que definen completamente la figura (tales como las cuatro esquinas de un rectángulo), tras lo cual la figura se reconstruye a partir de los puntos clave transformados.

Usamos la notación $\mathbf{P} = (x, y)$ de un punto bidimensional, y $\mathbf{P}^* = (x^*, y^*)$ para el punto transformado. La transformación lineal más sencilla es $x^* = ax + cy$, $y^* = bx + dy$ en la que cada una de las nuevas coordenadas es una *combinación lineal* de las dos coordenadas originales. Esta transformación

Figura 4.181: Tijeras y shearing (extrusión o cizallamiento).

puede escribirse $\mathbf{P}^* = \mathbf{PT}$, donde $\mathbf{T}$ es la matriz de 2×2, $\left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right)$.

Para entender las funciones de los cuatro elementos de la matriz, comenzamos estableciendo $b = c = 0$. La transformación se convierte en: $x^* = ax$, $y^* = dy$. Tal transformación se denomina *escalado* (*scaling*). Si se aplica a todos los puntos de un objeto, todas las dimensiones x son escaladas en un factor a, y todas las dimensiones y son escaladas en un factor d. Observe que a y d también puede ser menor que 1, causando la reducción del objeto. Si cualquiera —a o d— es igual a -1, la transformación es una *reflexión* (*reflection*). Otros valores negativos producen una combinación de escalado y reflexión.

Observe que el escalado de un objeto con los factores a y d cambia su área en un factor $a \times d$, y que este factor es también el valor del determinante de la matriz de escalado, $\left(\begin{smallmatrix} a & 0 \\ 0 & d \end{smallmatrix}\right)$.

(El escalado, la reflexión y otras transformaciones geométricas pueden extenderse a tres dimensiones, donde se hacen mucho más complejas.)

A continuación se muestran ejemplos de matrices para el escalado y la reflexión. En a, las coordenadas y son escaladas en un factor de 2. En b, las coordenadas x se reflejan. En c, las dimensiones x se reducen a 0,001 de sus valores originales. En d, la figura se redujo a una línea vertical:

$$a = \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}, \quad b = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}, \quad c = \begin{pmatrix} 0{,}001 & 0 \\ 0 & 1 \end{pmatrix}, \quad d = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}.$$

El siguiente paso es establecer $a = 1$, $d = 1$ (sin escalado, ni reflexión), y explorar el efecto de b y c en las transformaciones. La transformación se convierte en $x^* = x + cy$, $y^* = bx + y$. En primer lugar, seleccionamos la matriz $\left(\begin{smallmatrix} 1 & 1 \\ 0 & 1 \end{smallmatrix}\right)$ y la utilizamos para transformar el rectángulo cuyas cuatro esquinas son: $(1,0)$, $(3,0)$, $(1,1)$, y $(3,1)$. Las esquinas se transforman en $(1,1)$, $(3,3)$, $(1,2)$, $(3,4)$. El rectángulo original ha sido *acortado* (*sheared*) verticalmente y se transforma en un paralelogramo. Un efecto similar se produce cuando probamos la matriz $\left(\begin{smallmatrix} 1 & 0 \\ 1 & 1 \end{smallmatrix}\right)$. Por consiguiente, las cantidades b y c son responsables de la *extrusión* (*shearing*). La Figura 4.181 muestra la conexión entre la extrusión y el uso de unas tijeras. La palabra *shearing* viene del concepto de extrusión (*shear*) en mecánica.

◇ **Ejercicio 4.76 (sol. en pág. 1095):** Aplíquese la transformación de distorsión $\left(\begin{smallmatrix} 1 & -1 \\ 0 & 1 \end{smallmatrix}\right)$ a los cuatro puntos $(1,0)$, $(3,0)$, $(1,1)$, y $(3,1)$. ¿Cuáles son los puntos transformados? ¿Qué figura geométrica representan?

La siguiente transformación importante es la *rotación* (*rotation*). La Figura 4.180 ilustra la rotación. Muestra un punto $\mathbf{P}$, rotado en el sentido de las agujas del reloj hasta un ángulo θ, para convertirse en $\mathbf{P}^*$. La simple trigonometría produce $x = R\cos\alpha$ e $y = R\sin\alpha$ ɑ. A partir de ésto, obtenemos las expresiones para x^* e y^*:

$$\begin{aligned} x^* &= R\cos(\alpha - \theta) = R\cos\alpha\cos\theta + R\sin\alpha\sin\theta = x\cos\theta + y\sin\theta, \\ y^* &= R\sin(\alpha - \theta) = -R\cos\alpha\sin\theta + R\sin\alpha\cos\theta = -x\sin\theta + y\cos\theta. \end{aligned}$$

Por lo que, la matriz de rotación en dos dimensiones es:

$$\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}, \tag{4.62}$$

que también es igual a:

$$\begin{pmatrix} \cos\theta & 0 \\ 0 & \cos\theta \end{pmatrix} \begin{pmatrix} 1 & -\tan\theta \\ \tan\theta & 1 \end{pmatrix}.$$

Esto demuestra que cualquier rotación en dos dimensiones es una combinación de escalado (y, quizás, reflexión) y extrusión, un resultado inesperado que es cierto para todos los ángulos que satisfacen: $\tan\theta \neq \infty$.

La matrix $\mathbf{T}_1$ de más abajo rota en el sentido contrario a las agujas del reloj. La matrix $\mathbf{T}_2$ refleja sobre la recta $y = x$, y la matriz $\mathbf{T}_3$ refleja sobre la recta $y = -.x$. Observe los determinantes de estas matrices. En general, un determinante de $+1$ indica una rotación pura, mientras que un determinante de -1 indica un reflejo puro. (Como recordatorio, $\det \left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right) = ad - bc$.)

$$\mathbf{T}_1 = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}, \quad \mathbf{T}_2 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad \mathbf{T}_3 = \begin{pmatrix} 0 & -1 \\ -1 & 0 \end{pmatrix}.$$

Una rotación de 90^o

En el caso de una rotación de 90^o en el sentido horario, la matriz de rotación es:

$$\begin{pmatrix} \cos(90) & -\sin(90) \\ \sin(90) & \cos(90) \end{pmatrix} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}. \tag{4.63}$$

Un punto $\mathbf{P} = (x, y)$ se transforma, por lo tanto, en el punto $(y, -x)$. Para una rotación de 90^o en el sentido antihorario, (x, y) se transforma en $(-y, x)$. Ésto se conoce como la regla de *negación e intercambio*.

Traslaciones

Por desgracia, nuestra sencilla matriz de 2×2 ¡no puede generar todas las transformaciones necesarias! Específicamente, no puede generar la *traslación*. Ésto se demuestra teniendo en cuenta que cualquier objeto que contiene el origen, después de cualesquiera transformaciones anteriores, aún contendrá el origen (el resultado de $(0, 0) \times \mathbf{T}$ es $(0, 0)$ para cualquier matriz $\mathbf{T}$).

Una forma de implementar la traslación (que puede ser expresada mediante $x^* = x+m$, $y^* = y+n$), es generalizar nuestras transformaciones a $\mathbf{P}^* = \mathbf{PT} + (m, n)$, donde $\mathbf{T}$ es la familiar matriz de transformación de 2×2, $\left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right)$. Un enfoque más elegante, sin embargo, es a permanecer con $\mathbf{P}^* = \mathbf{PT}$ y generalizar $\mathbf{T}$ a la matriz de 3×3:

$$\mathbf{T} = \begin{pmatrix} a & b & 0 \\ c & d & 0 \\ m & n & 1 \end{pmatrix}.$$

Este enfoque se denomina *coordenadas homogéneas* y se utiliza comúnmente en geometría proyectiva. Ésto hace que sea posible unificar todas las transformaciones bidimensionales dentro de una matriz. Observe que sólo seis de los nueve elementos de la matriz $\mathbf{T}$ son variables. Nuestros puntos deberían ser ahora los tripletes $P = (x, y, 1)$.

Es fácil ver que las transformaciones descritas anteriormente pueden cambiar longitudes y ángulos. El escalado cambia las longitudes de los objetos. La rotación y la extrusión cambian los ángulos. Una

propiedad que se conserva, sin embargo, es el paralelismo de las líneas. Un par de líneas paralelas permanecerán paralelas después de cualquier escalado, reflexión, rotación, extrusión, y traslación. Una transformación que preserva el paralelismo se llama *afín*.

La conclusión final de esta sección es que cualquier transformación afín bidimensional puede ser completamente especificada ¡por sólo seis números!

Las transformaciones afines pueden definirse de diferentes maneras. Una definición importante es que una transformación de puntos en el espacio es afín si preserva las *sumas baricéntricas* de los puntos. Una suma baricéntrica de puntos $\mathbf{P}_i$ tiene la forma $\sum \omega_i \mathbf{P}_i$, donde los ω_i son números y $\sum \omega_i = 1$; por consiguiente, si $\mathbf{P} = \sum \omega_i \mathbf{P}_i$ y $\sum \omega_i = 1$, entonces cualquier transformación afín $\mathbf{T}$ satisface:

$$\mathbf{TP} = \sum_{1}^{n} \omega_i \mathbf{TP}_i.$$

4.35.2. Definición de IFS

Un sencillo ejemplo de IFS es el conjunto de las tres transformaciones:

$$\mathbf{T}_1 = \begin{pmatrix} 0{,}5 & 0 & 0 \\ 0 & 0{,}5 & 0 \\ 8 & 8 & 1 \end{pmatrix}, \quad \mathbf{T}_2 = \begin{pmatrix} 0{,}5 & 0 & 0 \\ 0 & 0{,}5 & 0 \\ 96 & 16 & 1 \end{pmatrix}, \quad \mathbf{T}_3 = \begin{pmatrix} 0{,}5 & 0 & 0 \\ 0 & 0{,}5 & 0 \\ 120 & 60 & 1 \end{pmatrix}. \tag{4.64}$$

En primer lugar, tratamos el concepto de *punto fijo*. Imagine la secuencia $\mathbf{P}_1 = \mathbf{P}_0\mathbf{T}_1$, $\mathbf{P}_2 = \mathbf{P}_1\mathbf{T}_1$, ..., donde la transformación $\mathbf{T}_1$ es aplicada varias veces para crear una secuencia de puntos $\mathbf{P}_1$, $\mathbf{P}_2$, ... Comenzamos con un punto arbitrario $\mathbf{P}_0 = (x_0, y_0)$ y examinamos el límite de la secuencia $\mathbf{P}_1 = \mathbf{P}_0\mathbf{T}_1$, $\mathbf{P}_2 = \mathbf{P}_1\mathbf{T}_1$, ... Es fácil demostrar que el límite de los $\mathbf{P}_i$ para i grande es el punto fijo $(2m, 2n) = (16, 16)$ (donde m y n son los factores de la traslación de matriz $\mathbf{T}_1$, La Ecuación (4.64)) sin tener en cuenta los valores de x_0 e y_0. El punto $(16, 16)$ se denomina *punto fijo* de $\mathbf{T}_1$, y *no depende* del punto $\mathbf{P}_0$ de partida seleccionado en particular.

Demostración: $\mathbf{P}_1 = \mathbf{P}_0\mathbf{T}_1 = (0{,}5x_0 + 8,\ 0{,}5y_0 + 8)$, $\mathbf{P}_2 = \mathbf{P}_1\mathbf{T}_1 = (0{,}5\,(0{,}5x_0 + 8) + 8,\ 0{,}5\,(0{,}5y_0 + 8) + 8)$. Es fácil ver (y probar por inducción) que $x_i = 0{,}5^i x_0 + 0{,}5^{i-1}8 + 0{,}5^{i-2}8 + \cdots + 0{,}5^1 8 + 8$. En el límite, $x_i = 0{,}5^i x_0 + 8\sum_{j=0}^{\infty} 0{,}5^j = 0{,}5^i x_0 + 8 \times 2$, que se aproxima al límite $8 \times 2 = 16$ para i grande independientemente de x_0.

Ahora es fácil demostrar que para las transformaciones anteriores, con factores de escala de 0,5 y sin extrusión (shearing), cada nuevo punto se mueve en la secuencia la mitad de la distancia restante hacia el punto fijo. Dado un punto $\mathbf{P}_i = (x_i, y_i)$, el punto medio entre $\mathbf{P}_i$ y el punto fijo $(16, 16)$ es:

$$\left(\frac{x_i + 16}{2}, \frac{y_i + 16}{2}\right) = (0{,}5x_i + 8,\ 0{,}5y_i + 8) = (x_{i+1}, y_{i+1}) = \mathbf{P}_{i+1}.$$

Por consiguiente, para las transformaciones particulares anteriores no hay necesidad de utilizar la matriz de transformación. En cada paso de la iteración, el punto $\mathbf{P}_{i+1}$ se obtiene mediante $(\mathbf{P}_i + (2m, 2n))/2$. Para otras transformaciones, es necesaria la multiplicación de matrices para calcular el punto $\mathbf{P}_{i+1}$.

En general, cada transformación afín donde la escala y los factores de extrusión son menores que 1 tienen un punto fijo, pero puede que no sea fácil de encontrar.

El principio de IFS ahora es fácil de describir. Se selecciona un conjunto de transformaciones (un código del IFS). Se calcula una secuencia de puntos y se representan comenzando con un punto arbitrario $\mathbf{P}_0$, seleccionando una transformación del conjunto al azar, y aplicándolo a $\mathbf{P}_0$, transformándolo en un punto $\mathbf{P}_1$, y luego seleccionando al azar otra transformación y aplicandola a $\mathbf{P}_1$, generando así el punto $\mathbf{P}_2$, y así sucesivamente.

Cada punto se representa en la pantalla a medida que es calculado, y gradualmente, el objeto comienza a tomar forma ante los ojos del espectador. La forma del objeto se llama *atractor* del IFS, y

El triángulo Sierpiński, también conocido como el *gasket* de Sierpiński (Figura 4.182a), se define recursivamente. Se comienza con cualquier triángulo, se encuentra el punto medio de cada arista, se conectan los tres puntos medios para obtener un nuevo triángulo confinado totalmente en el original, y se extrae el nuevo triángulo. El agujero recién creado ahora divide el triángulo original en tres más pequeños. Se repite el proceso en cada uno de los triángulos más pequeños. En el límite, ya no queda área disponible en el triángulo. Se parece a un queso suizo sin nada de queso, sólo agujeros.

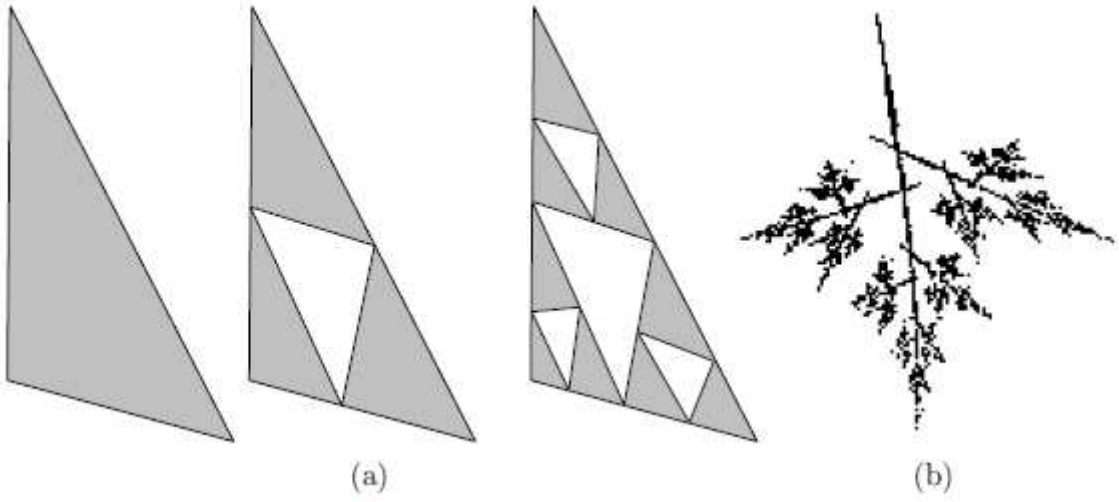


Figura 4.182: (a) Triándulo de Sierpiński. (b) Una hoja de helecho.

depende del código del IFS (las transformaciones) seleccionado. La forma también depende ligeramente de la selección particular de $\mathbf{P}_0$. Lo mejor es elegir como $\mathbf{P}_0$ uno de los puntos fijos del código IFS (si se conoce de antemano). En tal caso, todos los puntos en la secuencia se encontrarán dentro del atractor. Para cualquier otra elección de $\mathbf{P}_0$, un número finito de puntos estarán fuera del atractor, pero eventualmente pasarán al atractor y permanecerán allí.

Es sorprendente que el atractor no dependa del orden preciso de las transformaciones utilizadas. Este resultado ha sido demostrado por el matemático John Elton.

Otra característica sorprendente del IFS es que los números aleatorios utilizados no tienen que estar distribuidos uniformemente, sino que pueden ser ponderados. La transformación $\mathbf{T}_1$, por ejemplo, puede ser seleccionada al azar el 50 % del tiempo, la transformación $\mathbf{T}_2$, el 30 %, y $\mathbf{T}_3$, el 20 %. La forma que se genera no depende de las probabilidades, pero el tiempo de computación sí. Los pesos deben sumar 1 (un requisito normal para un conjunto de pesos matemáticos), y ninguno puede ser cero.

Las tres transformaciones de la Ecuación (4.64) de más arriba crean un atractor en forma de triángulo de Sierpiński (Figura 4.182a). Los factores de la traslación determinar las coordenadas de las tres esquinas del triángulo. Las seis transformaciones de la Tabla 4.183 crean un atractor en forma de helecho (Figura 4.182b). La notación utilizada en la Tabla 4.183 es $\left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right) + \left(\begin{smallmatrix} m \\ n \end{smallmatrix}\right)$.

El programa de la Figura 4.184 calcula y muestra los atractores del IFS para el conjunto de transformaciones dado. Se ejecuta en un ordenador Macintosh (debido a cambios en el sistema operativo de Macintosh, este programa ya no se puede ejecutar y se debe considerarse pseudocódigo).

	a	b	c	d	m	n
1 :	0	−28	0	29	151	92
2 :	−2	37	−31	29	85	103
3 :	−1	18	−18	−1	88	147
4 :	64	0	0	64	82	6
5 :	0	−80	−22	1	243	151
6 :	2	−48	0	50	160	80

Todos los números se muestran como números enteros, pero a, b, c, y d deben dividirse por 100, para hacerlos menores que 1. Los valores de m y n son los factores de traslación.

Tabla 4.183: Seis transformaciones.

```
PROGRAM IFS;
USES ScreenIO, Graphics, MathLib;
CONST LB = 5; Width = 490; Height = 285;
(* LB=esquina inferior izquierda de la ventana *)
VAR i,k,x0,y0,x1,y1,NumTransf: INTEGER;
Transf: ARRAY[1..6,1..10] OF INTEGER;
Params:TEXT;
filename:STRING;
BEGIN (* main *)
Write('params file='); Readln(filename);
Assign(Params,filename); Reset(Params);
Readln(Params,NumTransf);
FOR i:=1 TO NumTransf DO
   Readln(Params,Transf[1,i],Transf[2,i],Transf[3,i],
      Transf[4,i],Transf[5,i],Transf[6,i]);
   OpenGraphicWindow(LB,LB,Width,Height,'IFS shape');
   SetMode(paint);
   x0:=100; y0:=100;
   REPEAT
      k:=RandomInt(1,NumTransf+1);
      x1:=Round((x0*Transf[1,k]+y0*Transf[2,k])/100)+Transf[5,k];
      y1:=Round((x0*Transf[3,k]+y0*Transf[4,k])/100)+Transf[6,k];
      Dot(x1,y1); x0:=x1; y0:=y1;
   UNTIL Button()=TRUE;
   ScBOL; ScWriteStr('Pulse una tecla y cierre esta ventana para salir');
   ScFreeze;
END.
```

Figura 4.184: Cálculo y representación de los atractores del IFS.

4.35.3. Principios del IFS

Antes de describir cómo se utiliza un IFS para comprimir imágenes de la vida real, echemos un vistazo al IFS desde un punto de vista diferente. La Figura 4.185 muestra tres imágenes: una persona, la letra "T", y el empaque (*gasket*) de Sierpiński (o triángulo). Las dos primeras imágenes se transforman de una manera especial. Cada imagen es reducida a la mitad de su tamaño, luego copiada tres veces, y las tres copias se disponen en forma de triángulo. Cuando esta transformación se aplica algunas veces a una imagen, todavía es posible discernir las copias individuales de la imagen original. Sin embargo, cuando se aplica varias veces, el resultado es el *gasket* de Sierpiński (o algo muy cercano a él, en función del número de iteraciones y de la resolución del dispositivo de salida). La cuestión es que cada transformación reduce la imagen (las transformaciones son contractivas), por lo que el resultado final no depende de la forma de la imagen original.

La forma puede ser la de una persona, una carta, o cualquier otra cosa; el resultado final depende sólo de la transformación particular aplicada a la imagen. Una transformación diferente creará un resultado diferente, lo que de nuevo no depende de la imagen particular que está siendo transformada. La Figura 4.185d, por ejemplo, muestra los resultados de la transformación de la letra "T" reduciéndola, haciendo tres copias, organizándolas en un triángulo, y volteando la primera copia. La imagen final obtenida en el límite, después de aplicar una cierta transformación un número infinito de veces, se llama *atractor* de la transformación.

Los siguientes conjuntos de números crean patrones especialmente interesantes.

1. Una fachada

5					
0	−28	0	29	151	92
64	0	0	64	82	6
−2	37	−31	29	85	103
17	−51	−22	3	183	148
−1	18	−18	−1	88	147

2. Un litoral

4					
−17	−26	34	−12	84	53
25	−20	29	17	192	57
35	0	0	35	49	3
25	6	6	25	128	28

3. Una hoja (Figura 4,182b)

4					
2	−7	−2	48	141	83
40	0	−4	65	88	10
−2	45	−37	10	82	132
−11	−60	−34	22	237	125

4. Un triángulo de Sierpiński

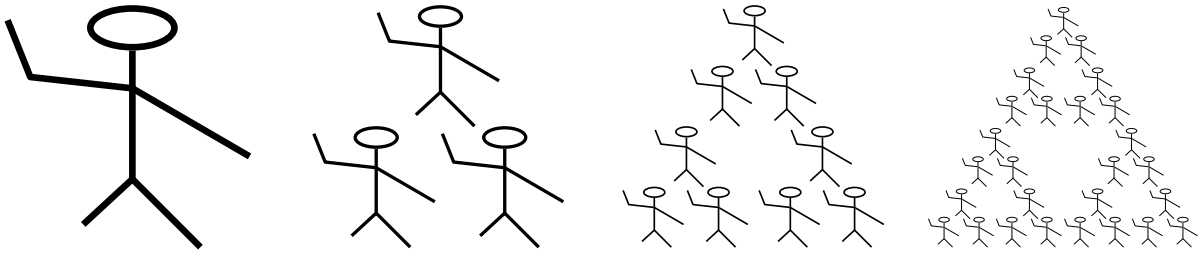
3					
50	0	0	50	0	0
50	0	0	50	127	79
50	0	0	50	127	0

◇ **Ejercicio 4.77 (sol. en pág. 1095):** Las tres transformaciones afines del ejemplo 4 anterior (el triángulo de Sierpiński) son diferentes de aquéllos de la Ecuación (4.64). ¿Cuál es la explicación?

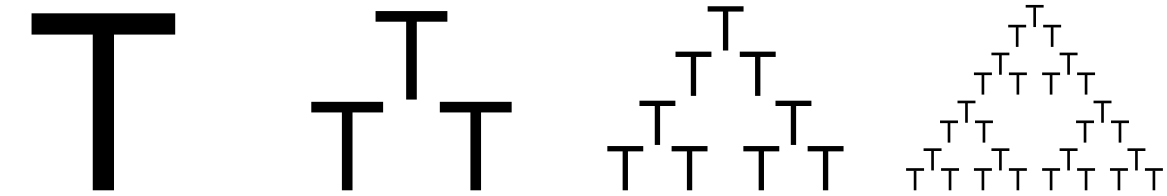
El resultado de cada transformación es una imagen que contiene todas las imágenes de todas las transformaciones anteriores. Si aplicamos la misma transformación muchas veces, es posible ampliar el resultado, para magnificarlo muchas veces, y todavía ver los detalles de las imágenes originales. En principio, si se aplica la transformación un número infinito de veces, el resultado final mostrará los detalles a *cualquier* aumento. Será un fractal.

El caso de la Figura 4.185c es especialmente interesante. Parece que la imagen original simplemente se muestra en cuatro ocasiones, sin ninguna transformación. Pensando un poco, sin embargo, se demuestra que nuestra transformación particular transforma esta imagen en sí misma. La imagen original ya es el *gasket* de Sierpiński, y se transforma a sí mismo porque es autosimilar.

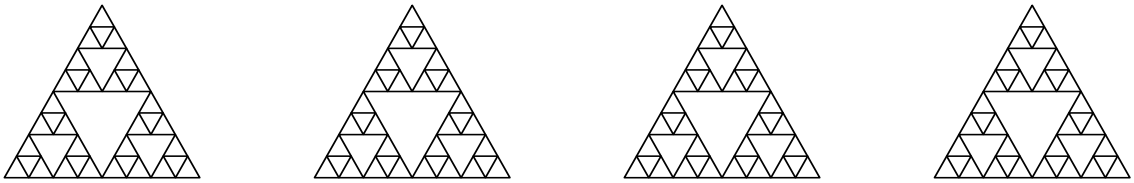
◇ **Ejercicio 4.78 (sol. en pág. 1095):** Explíquese el significado geométrico de la combinación de las



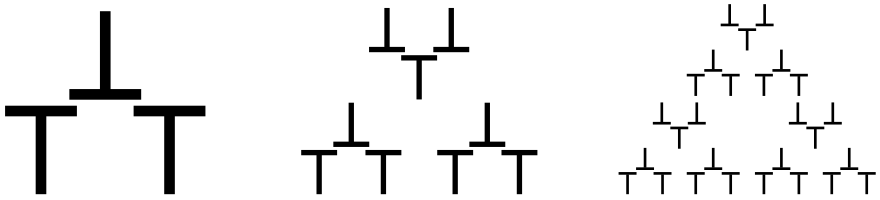
(a)



(b)



(c)



(d)

Figura 4.185: Creación del gasket de Sierpiński.

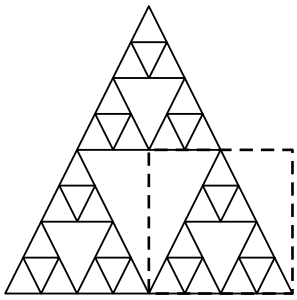


Figura 4.186: Una imagen autosimilar.

tres transformaciones afines siguientes y muéstrese el atractor al que convergen:

$$\omega_1 \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1/2 & 0 \\ 0 & 1/2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix},$$
$$\omega_2 \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1/2 & 0 \\ 0 & 1/2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0 \\ 1/2 \end{pmatrix},$$
$$\omega_3 \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1/2 & 0 \\ 0 & 1/2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 1/2 \\ 0 \end{pmatrix}.$$

El *gasket* de Sierpiński es, por tanto, fácil de comprimir, ya que es similar a sí mismo; es fácil encontrar partes del mismo que son idénticas a la imagen completa. De hecho, cada parte de él es idéntica a toda la imagen. La Figura 4.186 muestra la parte inferior derecha del gasket rodeado por líneas discontinuas. Es fácil ver la relación entre esta parte y la imagen completa. Sus formas son idénticas, hasta un factor de escala. El tamaño de esta parte es la mitad del tamaño de la imagen, y se sabe donde está ubicada en relación a la imagen completa (podemos medir el desplazamiento de su esquina inferior izquierda desde la esquina inferior izquierda de la imagen entera).

Esto apunta a una posible manera de comprimir imágenes reales. Si somos capaces de dividir una imagen en partes de tal manera que cada parte sea idéntica (o al menos muy cercana) a la imagen completa hasta a un factor de escala, entonces podemos comprimir la imagen mediante un IFS. Todo lo que necesitamos es el factor de escala (en realidad dos factores de escala, en las direcciones x e y) y el desplazamiento de cada parte en relación con la imagen completa [las distancias (x, y) entre una esquina de la parte y la misma esquina de la imagen entera]. A veces podemos encontrar una parte de la imagen que necesita ser reflejada con el fin de hacerse idéntica a toda la imagen. En tales casos también necesitamos los coeficientes de reflexión. Por consiguiente, podemos comprimir una imagen averiguando las transformaciones que transforman cada parte (llamada "rango") en la imagen completa. La transformación para cada parte se expresa mediante unos pocos números, y esos números se convierten en el *stream* comprimido.

Es fácil ver que este simple enfoque no funcionará para imágenes de la vida real. Tales imágenes son complejas, y por lo general es imposible dividir dicha imagen en partes que sean del todo idénticas (o incluso muy cercanas) a la imagen completa. Se necesita un enfoque diferente para hacer el IFS práctico. El enfoque utilizado por cualquier algoritmo de IFS práctico es la partición de la imagen en partes que no se superpongan llamadas *rangos*. Pueden ser de cualquier tamaño y forma, pero en la práctica es más fácil trabajar con cuadrados, rectángulos o triángulos. Para cada rango R_i, el codificador tiene que encontrar un *dominio* D_i que es muy similar, o incluso idéntico en forma, al rango pero es más grande. Una vez que se encuentra dicho dominio, es fácil imaginarse la transformación ω_i que va a transformar el dominio en el rango $R_i = \omega_i(D_i)$. Tienen que ser determinados dos factores

de escala (el escalado es de reducción, ya que el dominio es más grande que el rango), así como el desplazamiento del dominio relativo al rango [las distancias (x, y) entre una esquina del dominio y la misma esquina del rango]. A veces, el dominio tiene que ser rotado y/o reflejado para hacerlo idéntico al rango, y la transformación debe, por supuesto, incluir también estos factores. Este enfoque para la compresión de imágenes de IFS se llama PIFS (por IFS particionados).

4.35.4. Decodificación del IFS

Antes de examinar los detalles de la codificación del PIFS, vamos a tratar de comprender cómo funciona un decodificador PIFS. Todo lo que el decodificador tiene es el conjunto de transformaciones, una por cada rango. No conoce las formas de ningún rango o dominio. A pesar de ésto, la decodificación es muy simple. Se basa en el hecho, antes mencionado, de que una transformación contractiva crea un resultado que no depende de la forma de la imagen inicial utilizada. Podemos, por lo tanto, crear cualquier rango R_i aplicando la transformación contractiva ω_i muchas veces para *cualquier* forma grande D_i (excepto una forma totalmente blanca).

El decodificador comienza estableciendo todos los dominios a formas arbitrarias (e.g, puede establecer inicialmente la imagen completa a negro). A continuación, entra en un bucle donde en cada iteración aplica cada transformación ω_i una vez. La primera iteración aplica las transformaciones a los dominios D_i que son enteramente negros. Ésto crea rangos R_i que ya pueden, después de ésta sola iteración, parecerse ligeramente a los rangos originales. Esta iteración cambia la imagen desde la inicial completamente negra, a algo parecido a la imagen original. En la segunda iteración, el decodificador aplica de nuevo todas las transformaciones ω_i, pero esta vez son aplicadas a dominios que ya no son negros por completo. Los dominios ya se parecen algo a los originales, por lo que los resultados de la segunda iteración produce rangos de mejor aspecto, y por lo tanto, una mejor imagen. La experiencia muestra que normalmente sólo se necesitan $8-10$ iteraciones para obtener un resultado muy parecido a la imagen original.

Es importante darse cuenta de que este proceso de decodificación ¡es independiente de la resolución! Normalmente, el decodificador comienza con una imagen inicial, cuyo tamaño es idéntico al de la imagen original. Puede, sin embargo, comenzar con una imagen en negro de cualquier tamaño. Las transformaciones afines utilizadas para codificar la imagen original no dependen de la resolución de la imagen o de las resoluciones de los rangos. La decodificación de una imagen en, digamos, el doble de su tamaño original creará una imagen grande y suave, con detalles que no se ven en el original y sin pixelización (sin bordes dentados o píxeles "gruesos"). Los detalles adicionales serán, por supuesto, artificiales. Puede no ser lo que uno ve cuando mira la imagen original a través de una lupa, pero la cuestión es que la decodificación del PIFS es independiente de la resolución; crea una imagen de aspecto natural en cualquier tamaño, y no implica pixelización.

La natural independencia de la resolución de la decodificación del PIFS también significa que tenemos que ser cuidadosos al medir el rendimiento de la compresión. Tras la compresión de una imagen de 64 Kb en, digamos, 2 Kb, el factor de compresión es de 32. La decodificación del archivo comprimido de 2 Kb en uno grande, de 2 Mb, no significa que hemos cambiado el factor de compresión de la imagen (con un montón de detalles artificiales) a $^{2M}/_{2K} = 1024$. El factor de compresión es todavía el mismo: 32.

4.35.5. Codificación del IFS

La decodificación del PIFS es, por tanto, fácil, aunque algo mágica, pero aún tenemos que ver los detalles de la codificación del PIFS. El primer punto a considerar es la forma de seleccionar los rangos y encontrar los dominios. La siguiente es una manera sencilla de hacer ésto.

Supongamos que la imagen original tiene una resolución de 512×512. Podemos seleccionar como rangos los grupos no superpuestos de 16×16 píxeles. Hay $32 \times 32 = 1024$ de estos grupos. Los dominios

deben ser más grandes que los rangos, por lo que podemos seleccionar como dominios los 32×32 grupos de píxeles en la imagen (que pueden, por supuesto, superponerse). Hay $(512-31)\times(512-31)=231\,361$ de tales grupos. El codificador debe comparar cada rango con todos los 231 361 dominios. Cada comparación consta de ocho pasos, porque un rango puede ser idéntico a una rotación o una reflexión de un dominio (Figura 4.178). El número total de pasos en la comparación de rangos y dominios es, por lo tanto, $1024\times231361\times8=1\,895\,309\,312$. Si cada uno paso toma un microsegundo, el tiempo total requerido es de 1 895 segundos, o aproximadamente 31 minutos.

◇ **Ejercicio 4.79 (sol. en pág. 1095):** Repítase el cálculo anterior para una imagen de 256×256 con rangos de tamaño 8×8 y dominios de tamaño 16×16.

Si el codificador está buscando un dominio para que coincida con el rango R_i y tiene la suerte de encontrar uno que es idéntico a R_i, puede proceder con el siguiente rango. En la práctica, sin embargo, los dominios idénticos a un determinado rango son muy raros, por lo que el codificador tiene que comparar todos los $231\,361\times8$ dominios con cada rango R_i y seleccionar el que está más cercano a R_i (PIFS es, en general, un método de compresión con pérdida). Por consiguiente, tiene que responder a dos preguntas: ¿Cuándo es un dominio idéntico a un rango? (recuerde: tienen tamaños diferentes) y ¿cómo hacemos medir la "distancia" entre un dominio y un rango?

Para comparar un dominio de 32×32 píxeles con un rango de 16×16 píxeles, podemos, o bien elegir un píxel de cada uno de los 2×2 cuadrados de píxeles en el dominio (ésto se denomina submuestreo), o bien hacer un promedio de cada cuadradado de 2×2 píxeles en el dominio y compararlo con un píxel del rango (promedio).

Para decidir cuán cerca está un rango R_i de un dominio D_j, tenemos que utilizar una de varias *métricas*. Una métrica es una función que mide la "distancia" entre, o la "cercanía" de, dos magnitudes matemáticas. La experiencia recomienda el uso de la métrica *rms* (root mean square o raíz cuadrática media):

$$M_{\text{rms}}\left(R_i,D_i\right)=\sqrt{\sum_{x,y}\left[R_i\left(x,y\right)-D_j\left(x,y\right)\right]^2}. \tag{4.65}$$

Ésto implica un cálculo de la raíz cuadrada, por lo que una métrica más simple puede ser:

$$M_{\max}\left(R_i,D_i\right)=\max\left|R_i\left(x,y\right)-D_j\left(x,y\right)\right|$$

(la diferencia más grande entre un píxel de R_i y un píxel de D_j). Cualquiera que sea la métrica que se utilice, una comparación de un rango y un dominio implica un submuestreo (o promedio), seguido por un cálculo de la métrica.

Después de comparar el rango R_i con todos los dominios (rotados y reflejados), el codificador selecciona el dominio con la menor métrica y determina la transformación que llevará el dominio al rango. Este proceso se repite para todos los rangos.

Incluso esta sencilla manera de hacer coincidir las partes de la imagen produce excelentes resultados. Los factores de compresión están típicamente en el rango de $15-32$ y la pérdida de datos es mínima.

◇ **Ejercicio 4.80 (sol. en pág. 1095):** ¿Cuál es una forma razonable de estimar la cantidad de información de la imagen perdida en la compresión PIFS?

La principal desventaja de este método de determinación de rangos y dominios es el tamaño fijo de los rangos. Un método donde los rangos pueden tener tamaños diferentes puede producir una mejor compresión y menos pérdidas. Imagine una imagen de una mano con un anillo en un dedo. Si el anillo pasa a estar dentro de un rango R_i amplio , puede ser imposible encontrar un dominio que incluso se acerque a R_i. En tal caso se puede perder una gran cantidad de datos. Por otro lado, si una parte de una imagen es bastante uniforme, puede convertirse en un rango extenso, ya que existe una mejor oportunidad de que empareje con un dominio. Evidentemente, son preferibles los rangos grandes, ya

que el *stream* comprimido contiene una transformación por cada rango. Por lo tanto, los quadtrees ofrecen una buena solución.

Quadtrees: Comenzamos con un número reducido de grandes rangos, cada uno un subcuadrante. Si un rango no empareja bien con algún dominio (la métrica entre él y el dominio es mayor que un parámetro de tolerancia controlado por el usuario), se divide en cuatro subrangos, y cada uno es emparejado por separado. Como ejemplo, considere una imagen de 256×256 píxeles. Podemos optar por los dominios de todos los cuadrados de la imagen de tamaño 8, 12, 16, 24, 32, 48, y 64 píxeles. Comenzamos con rangos que sean subcuadrantes de tamaño 32×32. Cada rango se compara con los dominios que sean más grandes que sí mismos (48 ó 64) píxeles. Si un rango no empareja bien, se divide en cuatro cuadrantes de tamaño 16×16 cada uno, y cada uno se compara, como un nuevo rango, con todos los dominios de tamaños 24, 32, 48 y 64 píxeles. Este proceso continúa hasta que todos los rangos se hayan emparejado a los dominios. Los rangos grandes producen una mejor compresión, pero los rangos pequeños son más fáciles de emparejar, ya que contienen pocos píxeles adyacentes, y sabemos por experiencia que los píxeles adyacentes tienden a estar altamente correlacionados en las imágenes de la vida real.

4.35.6. IFS para imágenes en escala de grises

Hasta ahora hemos asumido que nuestras transformaciones tienen que ser afines. La verdad es que puede utilizarse cualquier transformación contractiva, incluso las no lineales, para el IFS. Las transformaciones afines se utilizan simplemente porque son lineales y, por lo tanto, computacionalmente sencillas. Además, hasta ahora hemos supuesto una imagen monocromática, donde el único problema es determinar qué píxeles deben ser negros. El IFS puede ser fácilmente ampliado para comprimir imágenes en escala de grises (y, por lo tanto, también imágenes en color; véase más adelante). El problema aquí es determinar qué píxeles dibujar, y qué nivel de gris utilizar en cada uno.

La comparación entre un dominio y un rango ahora involucra las intensidades de los píxeles en ambos. Cualquiera que sea la métrica empleada, debe usar sólo aquellas intensidades para determinar la "cercanía" entre el dominio y el rango. Supongamos que un cierto dominio D contiene n píxeles con niveles de gris $a_1, \ldots, a_n$, y el codificador del IFS intenta emparejar D con un rango R que contiene n píxeles con niveles de gris $b_1, \ldots, b_n$. La métrica rms, mencionada anteriormente, funciona localizando dos números, r y g (llamados controles de contraste y brillo), que minimizan la expresión:

$$Q = \sum_{1}^{n} \left((r \cdot a_i + g) - b_i \right)^2 . \tag{4.66}$$

Esto se efectúa resolviendo las dos ecuaciones $\partial Q/\partial r = 0$ y $\partial Q/\partial g = 0$ para r y g desconocidas (vea los detalles más abajo). Minimizar Q minimiza la diferencia en contraste y brillo entre el dominio y el rango. El valor de la métrica rms es $\sqrt{Q}$ [compare con la Ecuación (4.65)].

Cuando el codificador del IFS decide finalmente qué dominio asocia con el rango actual, tiene que encontrar la transformación ω entre ellos. La cuestión es que r y g deben ser incluidos en la transformación, por lo que el decodificador sabrá de qué nivel de gris deberá pintar los píxeles cuando el dominio se recrea en las sucesivas iteraciones de decodificación. Es común el uso de transformaciones de la forma:

$$\omega \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & r \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} l \\ m \\ g \end{pmatrix} . \tag{4.67}$$

Un píxel (x, y) en el dominio D, viene dado ahora con una tercera coordenada z (su nivel de gris) y se transforma en un píxel (x^*, y^*, z^*) en el rango R, donde $z^* = z \cdot r + g$. La transformación (4.67) tiene otra propiedad. Es contractiva si $r < 1$, independientemente de los factores de escala.

Cualquier método de compresión para imágenes en escala de grises puede extenderse a las imágenes en color. Sólo es necesario separar la imagen en tres componentes de color (preferiblemente YIQ[22]) y comprimir cada uno individualmente como una imagen en escala de grises. Así es cómo puede ser aplicado un IFS a la compresión de imágenes en color.

El siguiente punto que merece consideración es cómo escribir los coeficientes de una transformación ω en el flujo de datos comprimidos. Hay tres grupos de coeficientes, los factores de escala a y d, los factores de reflexión/rotación a, b, c, y d, los desplazamientos l y m, y los controles de contraste/brillo r y g. Si un dominio es el doble de grande que un rango, entonces los factores de escala siempre son 0,5 y, por consiguiente, no tienen que ser escritos en el flujo comprimido. Si los dominios y los rangos pueden tener varios tamaños, entonces sólo son posibles algunos factores de escala, y ellos pueden ser codificados, ya sea aritméticamente, ya sea con algún código de prefijos. La rotación o reflexión en particular del dominio relativo al rango, puede ser codificada con tres bits, ya que sólo hay ocho rotaciones/reflexiones posibles. El desplazamiento puede ser codificado mediante la codificación de las posiciones y los tamaños del dominio y del rango.

Las cantidades r y g no están distribuidas de cualquier manera uniforme, y son también números reales (en coma flotante) que pueden tener muchos valores diferentes. Por tanto, deben ser cuantificadas, i.e., convertidas en un entero comprendido en un cierto rango. La experiencia muestra que el contraste r puede ser cuantificado en un entero de 4 ó 5 bits (i.e., en la práctica son suficientes 16 ó 32 valores de contraste), mientras que el brillo g debe ser un número entero de 6 ó 7 bits (resultando en 64 ó 128 valores de brillo).

Aquí están los detalles del cálculo de r y g que minimizan Q y posteriormente el cálculo de Q (minimizado) y la métrica rms.

De la Ecuación (4.66), obtenemos:

$$\begin{aligned}\frac{\partial Q}{\partial g} =& 0 \to \sum 2\left(r \cdot a_i + g - b_i\right) = 0 \to ng + \sum\left(r \cdot a_i - b_i\right) = 0,\\ g =& \frac{1}{n}\left[\sum b_i - r\sum a_i\right],\end{aligned} \tag{4.68}$$

y

$$\begin{aligned}\frac{\partial Q}{\partial g} = 0 \to \sum 2\left(r \cdot a_i + g - b_i\right) a_i = 0 \to \sum\left(r \cdot a_i^2 + g \cdot a_i - a_i b_i\right) = 0,\\ r\sum a_i^2 + \frac{1}{n}\left[\sum b_i - r\sum a_i\right]\sum a_i - \sum a_i b_i = 0,\\ r\left[\sum a_i^2 - \frac{1}{n}\left(\sum a_i\right)^2\right] = \sum a_i b_i - \frac{1}{n}\sum a_i \sum b_i,\\ r = \frac{\sum a_i b_i - \frac{1}{n}\sum a_i \sum b_i}{\sum a_i^2 - \frac{1}{n}\left(\sum a_i\right)^2} = \frac{n\sum a_i b_i - \sum a_i \sum b_i}{n\sum a_i^2 - \left(\sum a_i\right)^2}.\end{aligned} \tag{4.69}$$

De la misma Ecuación (4.66), también obtenemos el Q minimizado:

$$\begin{aligned}Q =& \sum_1^n \left(r \cdot a_i + g - b_i\right)^2 = \sum\left(r^2 a_i^2 + g^2 + b_i^2 + 2rga_i - 2ra_i b_i - 2gb_i\right)\\ =& r^2\sum a_i^2 + ng^2 + \sum b_i^2 + 2rg\sum a_i - 2r\sum a_i b_i - 2g\sum b_i.\end{aligned} \tag{4.70}$$

Los siguientes pasos son necesarios para calcular la métrica rms:

[22] YIQ es un espacio de color: Y es la luminancia (energía que percibe el observador); I procede de *in-phase* (en fase); Q es la inicial de *quadrature* (cuadratura).

```
t:=algún valor por defecto; [t es la tolerancia]
push(imagen completa); [la pila contiene los rangos a emparejar]
repeat
   R:=pop();
   comparar todos los dominios con R, encontrar aquel (D) que sea más cercano a R,
   pop(R);
if metric(R,D)<t then
   calcular la transformación w de D a R y emitirla;
else particionar R en rangos más pequeños e introducirlos en la pila;
endif;
until la pila esté vacía;
```

Figura 4.187: Codificación de un IFS. Versión I.

1. Calcular las sumas $\sum a_i$ y $\sum a_i^2$ para todos los dominios.
2. Calcular las sumas $\sum b_i$ y $\sum b_i^2$ para todos los rangos.
3. Cada vez que un rango R y un dominio D sean comparados, calcular:
   - 3.1 La suma $\sum a_i b_i$
   - 3.2 Las cantidades r y g, a partir de las Ecuaciones (4.68) y (4.69) utilizando las cinco sumas anteriores. Cuantificar r y g.
   - 3.3 Calcular Q a partir de la Ecuación (4.70) utilizando los r, g cuantificados y las cinco sumas. El valor de la métrica rms para estas R y D en particular, es $\sqrt{Q}$.

Finalmente, las Figuras 4.187 y (4.188) son algoritmos en pseudocódigo que describen dos enfoques para la codificación de un IFS. El primero es más intuitivo. Para cada intervalo R selecciona el dominio más cercano a R. El último intenta reducir la pérdida de datos sacrificando la relación de compresión. Esto se hace dejando al usuario especificar el número mínimo T de transformaciones a generar. (Cada transformación ω que sea escrita en el *stream* comprimido utilizanda aproximadamente el mismo número de bits, por lo que el tamaño del *stream* es proporcional al número de transformaciones.) Si cada rango ha sido emparejado, y el número de transformaciones todavía es menor que T, el algoritmo continúa tomando rangos que ya han sido emparejados y los particiona en unos más pequeños. Ésto aumenta el número de transformaciones, pero reduce la pérdida de datos, ya que los rangos pequeños son más fáciles de combinar con los dominios.

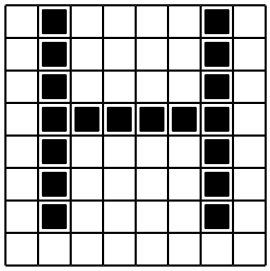
> Se reunieron y luego golpeó con el puño sobre la mesa. "Al diablo con el arte, dije yo."
>
> "Usted no lo dice solamente, sino que lo hace repetidamente tedioso", dijo Clutton severamente.
>
> —W. Somerset Maugham, de *Human Bondage (servidumbre humana)*

4.36. Codificación de las celdas o células

Imagínese una imagen almacenada en un mapa de bits y que se muestra en una pantalla. Empecemos con el caso donde la imagen se compone sólo de texto, con cada carácter ocupando la misma área,

```
El usuario introduce T;
push(imagen completa); [la pila contiene los rangos a emparejar]
repeat
   para cada R en la pila sin emparejar encontrar el mejor dominio D que empareje,
      calcular la transformación w, e introducir D y w en la pila;
   if el número de rangos en la pila es <T then
      encontrar el rango R con la métrica más larga (peor emparejamiento)
      pop R, D y w (sacarlos de la pila)
      particionar R en rangos más pequeños e introducirlos, como no emparejados,
         en la pila;
   endif
until todos los rangos en la pila estén emparejados;
sacar todas las transformaciones w de la pila;
```

Figura 4.188: Codificación de un IFS. Versión II.



0 1 0 0 0 0 1 0
0 1 0 0 0 0 1 0
0 1 0 0 0 0 1 0
0 1 1 1 1 1 1 0
0 1 0 0 0 0 1 0
0 1 0 0 0 0 1 0
0 1 0 0 0 0 1 0
0 0 0 0 0 0 0 0

Figura 4.189: Una letra H de 8×8.

decir 8×8 píxeles (lo suficientemente grande para un carácter de 5×7 ó 6×7 y algunos espacios). Asumiendo un conjunto de 256 caracteres, cada celda se puede codificar como un puntero de 8 bits apuntando a una tabla de 256 entradas donde cada entrada contiene la descripción de un carácter de 8×8 como una cadena de 64 bits. El factor de compresión es, por lo tanto, de $^{64}/_{8}$, u ocho a uno. La Figura 4.189 muestra la letra **H** tanto como un mapa de bits, como una cadena de 64 bits.

La codificación de las celdas no es muy útil para el texto (que siempre puede ser representado con ocho bits por carácter), pero este método también se puede extender a una imagen formada por líneas rectas solamente. El mapa de bits completo se divide en celdas de, digamos, 8×8 píxeles, y es escaneado celda por celda. La primera celda se almacena en la entrada 0 de una tabla y se codifica (i.e., se escribe en el archivo comprimido) como el puntero 0. Cada celda posterior es buscada en la tabla. Si se encuentra, su índice en la tabla se convierte en su código y se escribe en el archivo comprimido. En caso contrario, se añade a la tabla. Con celdas de 8×8, cada uno de los 64 píxeles puede ser negro o blanco, por lo que el número total de celdas diferentes es $2^{64} \approx 1{,}8 \times 10^{19}$, un número inmenso. Sin embargo, algunos patrones no aparecen nunca, ya que no representan ninguna combinación posible de segmentos de línea. Además, muchas celdas son trasladadas o reflejadas a partir de otras celdas (Figura 4.190). Todo esto hace que el número total de celdas distintas sea sólo 108 [Jordan y Barrett 74]. Estas 108 celdas o células pueden almacenarse en una memoria ROM y utilizarse frecuentemente para comprimir imágenes.

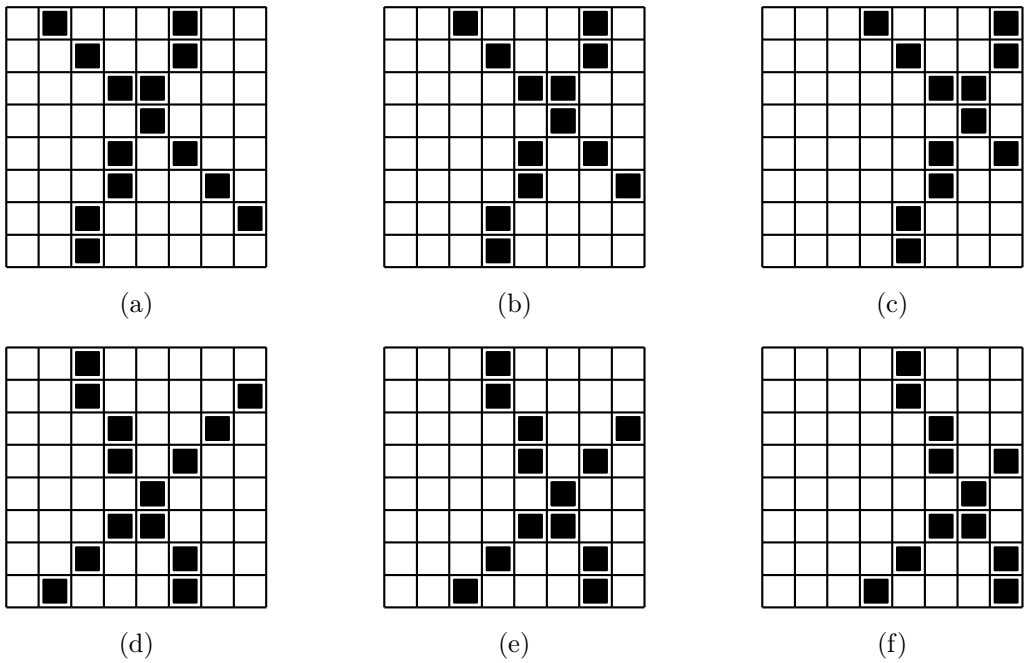


Figura 4.190: Seis *bitmaps* de 8×8 trasladados (a−c) y reflejados (d−f).

Una onza (28,3495231 gramos) de imagen vale 1 libra (453,59237 gramos) de rendimiento.

—Anónimo

Capítulo 5

Métodos wavelet

A principios de 1800, el matemático francés Joseph Fourier descubrió que casi cualquier función periódica se puede expresar como una suma (posiblemente infinita) de senos y cosenos. Este hecho sorprendente, hoy se conoce como la expansión de Fourier y tiene muchas aplicaciones en ingeniería, principalmente en el análisis de señales. Se pueden aislar las diferentes frecuencias que subyacen en una señal y con ello permitir al usuario estudiar la señal y también editarla mediante la supresión o adición de determinadas frecuencias. El lado negativo de la expansión de Fourier es que no nos dice cuándo (en qué punto o puntos en el tiempo) está activa cada frecuencia en una señal dada. Por lo tanto se dice que la expansión de Fourier ofrece una resolución en frecuencia, pero no una resolución en tiempo.

El análisis wavelet (o la transformada wavelet) es un enfoque exitoso para el problema del análisis de una señal tanto en tiempo como en frecuencia. Dada una señal que varía con el tiempo, seleccionamos un intervalo de tiempo, y utilizamos el enfoque wavelet para identificar y aislar las frecuencias que constituyen la señal en ese intervalo. El intervalo puede ser ancho, en cuyo caso decimos que la señal se analiza a gran escala. A medida que el intervalo de tiempo se hace más estrecho, la escala de análisis se hace cada vez más pequeña. Un análisis de gran escala ilustra el comportamiento global de la señal, mientras que cada análisis a pequeña escala, ilumina el camino del comportamiento de la señal en un intervalo de tiempo corto; es como hacer un zoom en la señal en el tiempo, en lugar de en el espacio. Por lo tanto, la idea fundamental subyacente en las wavelets es analizar una función o una señal de acuerdo con la escala.

Matemáticamente, las wavelets son funciones que satisfacen ciertos requisitos. Entre ellos está el requisito de que una wavelet se integra como cero. Ésto implica que para cada área de la función wavelet por encima del eje x, debe haber un área igual por debajo de dicho eje. Así, la función wavelet tiene que ondear por encima y por debajo del eje x, lo que justifica el nombre "*wave* (onda)". Los requisitos de otro resultado en las funciones que se localizan en el espacio, sugieren el uso del diminutivo "*wavelet* (ondita)" en lugar de "wave".

Este capítulo comienza con un análisis de la transformada de Fourier y los conceptos de los dominios de tiempo y frecuencia, lo que conduce naturalmente a un principio de incertidumbre. A continuación se ofrece una discusión de la transformada wavelet continua. La parte importante del capítulo introduce la transformada wavelet discreta mediante el uso de la transformada de Haar como un ejemplo. La transformada simple de Haar se extiende entonces a los más generales *filter banks* (bancos de filtros). Con el fin de ilustrar el poder del enfoque wavelet, el capítulo describe varios métodos de compresión importantes, tales como la pirámide Laplaciana, SPIHT, WSQ, y JPEG 2000.

5.1. La transformada de Fourier

El concepto de transformada es familiar para los matemáticos. Es una herramienta estándar matemática que se utiliza para resolver problemas en muchas áreas. La idea es cambiar una cantidad matemática (un número, un vector, una función o cualquier otra cosa) a otra forma, donde puede parecer desconocido, pero también tener propiedades útiles. La cantidad transformada se utiliza para resolver un problema o para realizar un cálculo, y el resultado se transforma de nuevo a la forma original.

Un sencillo e ilustrativo ejemplo son los números romanos. Los antiguos romanos, presumiblemente sabían cómo operar con dichos números, pero cuando tenemos que, digamos, multiplicar dos números romanos, podemos encontrar más conveniente transformarlos en notación moderna (árabe), multiplicarlos, y luego transformar el resultado en un número romano. Aquí tenemos un sencillo ejemplo:

$$\text{XCVI} \times \text{XII} \rightarrow 96 \times 12 = 1152 \rightarrow \text{MCLII}.$$

Las funciones que se usan en la ciencia y la ingeniería a menudo utilizan el *tiempo* como parámetro. Nosotros, por lo tanto, decimos que una función $g(t)$ se representa en el *dominio del tiempo*. Una función típica varía con el tiempo, por lo que podemos pensar en ello como algo similar a una onda, y podemos tratar de representarla como una onda (o como una combinación de ondas). Cuando se hace ésto, denotamos la función resultante por $G(f)$, donde f representa la frecuencia de la onda, y decimos que la función está representada en el *dominio de la frecuencia*. El concepto de dos dominios resulta ser útil, ya que muchas operaciones sobre las funciones son fáciles de llevar a cabo en el dominio de la frecuencia. La transformación de una función entre dominio del tiempo y el de la frecuencia es fácil cuando la función es *periódica*, pero también se puede realizar para ciertas funciones no periódicas.

Definición: Una función $g(t)$ es periódica si existe una constante P distinta de cero tal que $g(t+P) = g(t)$ para todos los valores de t. El P más pequeño se llama el *período* de la función.

La Figura 5.1 muestra tres funciones periódicas. La función de la Figura 5.1a es un pulso cuadrado, la de la Figura 5.1b es una onda sinusoidal, y la de la Figura 5.1c es más compleja.

Una función periódica tiene cuatro características importantes: su amplitud, su período, su frecuencia, y su fase. La amplitud de la función es el valor máximo que tiene en cualquier período. La frecuencia f es la inversa del período ($f = 1/P$); se expresa en ciclos por segundo, o hercios (Hz). La fase es el atributo de los cuatro más difícil de comprender; mide la posición de la función dentro de un período, y es fácil de visualizar cuando una función se compara con su propia copia. Considere las dos sinusoides de la Figura 5.1b: son idénticas, pero desfasadas. Una sigue a la otra en un intervalo fijo llamado *diferencia de fase*. Podemos expresarlas como $g_1(t) = A\sin(2\pi ft)$ y $g_2(t) = A\sin(2\pi ft + \theta)$. La diferencia de fase entre ellas es θ; pero también podemos decir que la primera no tiene ninguna fase, mientras que la segunda tiene una fase de θ. (Éste ejemplo muestra también que el coseno es una función seno con una fase de $\theta = \pi/2$.)

5.2. El dominio de la frecuencia

Para entender el concepto de dominio de la frecuencia, vamos a ver dos ejemplos sencillos. La función $g(t) = \sin(2\pi ft) + (1/3)\sin(2\pi(3f)t)$ es una combinación de dos ondas sinusoidales con amplitudes 1 y 1/3, y frecuencias f y $3f$, respectivamente. Se muestran en Figura 5.2a,b. Su suma (Figura 5.22c) también es periódica, con frecuencia f (la más pequeña de las dos frecuencias f y $3f$). El dominio de la frecuencia de $g(t)$ es una función que consta de tan sólo los dos puntos $(f, 1)$ y $(3f, 1/3)$ (Figura 5.2h). Ésto indica que la función original (en el dominio del tiempo) se compone de la frecuencia f con amplitud 1, y frecuencia $3f$ con amplitud 1/3.

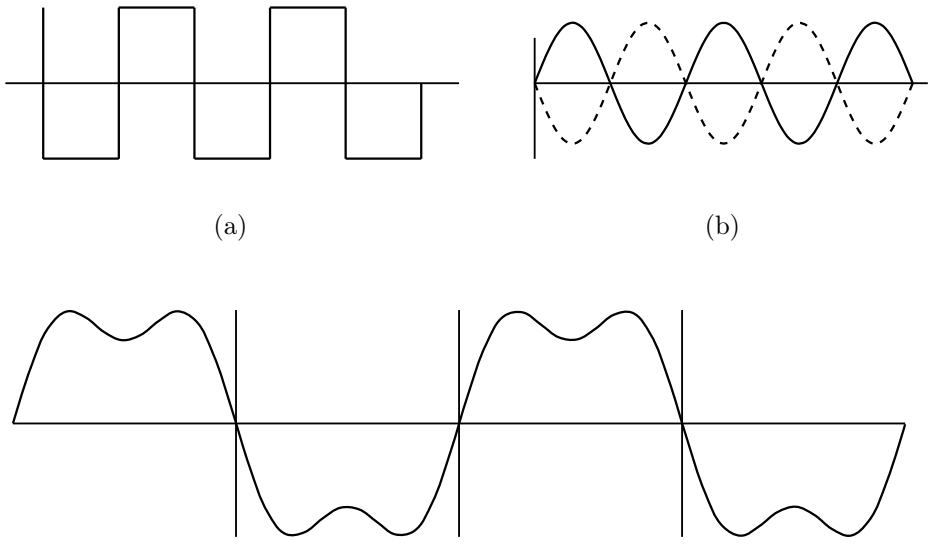


Figura 5.1: Funciones periódicas.

Éste ejemplo es muy sencillo, ya que involucra sólo a dos frecuencias. Cuando una función consta de varias frecuencias que son múltiplos enteros de alguna frecuencia más baja, ésta última se denomina *frecuencia fundamental* de la función.

No todas las funciones tienen una representación sencilla en el dominio de la frecuencia. Considere el pulso cuadrado de la Figura 5.2d. Su dominio del tiempo es:

$$g(t) = \begin{cases} 1, & -a/2 \leq t \leq a/2, \\ 0, & \text{en cualquier otro caso,} \end{cases}$$

pero su dominio de la frecuencia es la Figura 5.2e. Ésta consta de todas las frecuencias de 0 a ∞, con amplitudes que caen de forma continua. Ésto significa que la representación en el dominio del tiempo, aunque sencilla, está formada por todas las frecuencias posibles, con las frecuencias más bajas contribuyendo más, y las más altas, contribuyendo cada vez menos.

En general, una función periódica puede representarse en el dominio de la frecuencia como la suma de ondas sinusoidales (desfasadas) con frecuencias que son múltiplos enteros (armónicos) de alguna frecuencia fundamental. Sin embargo, el pulso cuadrado de la Figura 5.2d no es periódico. Resulta que los conceptos de dominio de la frecuencia se pueden aplicar a una función no periódica, pero sólo si es distinta de cero dentro de un intervalo finito (como el pulso cuadrado). Tal función se representa como la suma de ondas sinusoidales (desfasadas) con todo tipo de frecuencias, no sólo armónicos.

El *espectro* en el dominio de la frecuencia (a veces llamado también *contenido de frecuencias* de la función) es el rango de frecuencias que contiene. Para la función de la Figura 5.2h, el espectro consta de las dos frecuencias f y $3f$. Para una de la Figura 5.2e, es el rango de enteros $[0, \infty]$. El *ancho de banda* en el dominio de la frecuencia es la anchura del espectro. Ésta es $2f$ en el primer ejemplo, e infinito en el segundo ejemplo.

Otro concepto importante a definir es la *componente dc* de la función. El dominio del tiempo de una función puede incluir un componente de frecuencia cero. Los ingenieros llaman a ésto componente de *corriente continua* (*direct current*), por lo que el resto de nosotros hemos adoptado el término

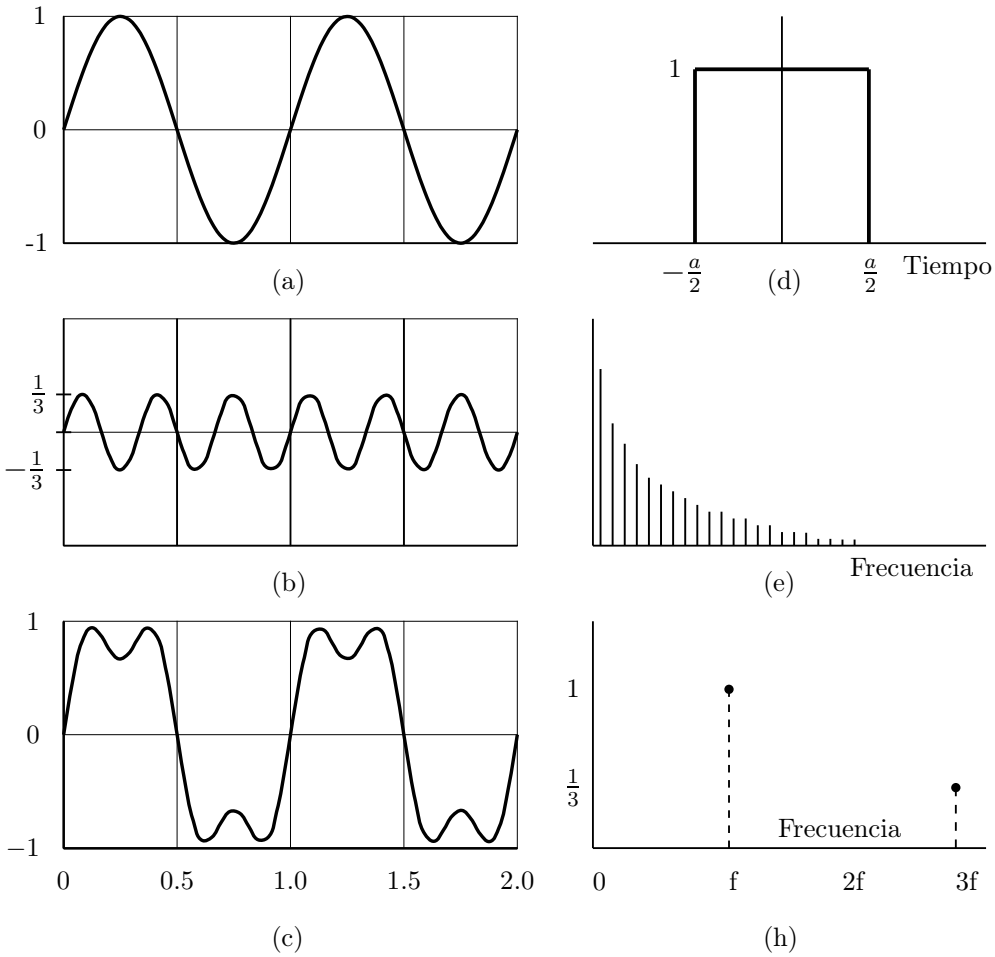


Figura 5.2: Dominios del tiempo y de la frecuencia.

"componente dc". La Figura 5.3a es idéntica a la Figura 5.2c, excepto que va desde 0 a 2, en lugar de desde -1 a $+1$. El dominio de la frecuencia (Figura 5.3b) ahora tiene un punto añadido en $(0, 1)$, que representa la componente dc.

Todo el concepto de los dos dominios se debe al matemático francés Joseph Fourier. Él probó un teorema fundamental que dice que toda función periódica —real o compleja— puede ser representada como la suma de funciones seno y coseno. También demostró cómo transformar una función entre el dominio del tiempo y el de la frecuencia. Si la forma de la función está lejana a la de una onda periódica, su desarrollo de Fourier incluirá un número infinito de frecuencias. Para una función continua $g(t)$, la transformada de Fourier y su inversa están dadas por:

$$G(f) = \int_{-\infty}^{\infty} g(t) \left[\cos(2\pi f t) - i \sin(2\pi f t)\right] dt,$$
$$g(t) = \int_{-\infty}^{\infty} G(f) \left[\cos(2\pi f t) + i \sin(2\pi f t)\right] df.$$

En las aplicaciones informáticas, normalmente tenemos funciones discretas, que toman sólo n valores

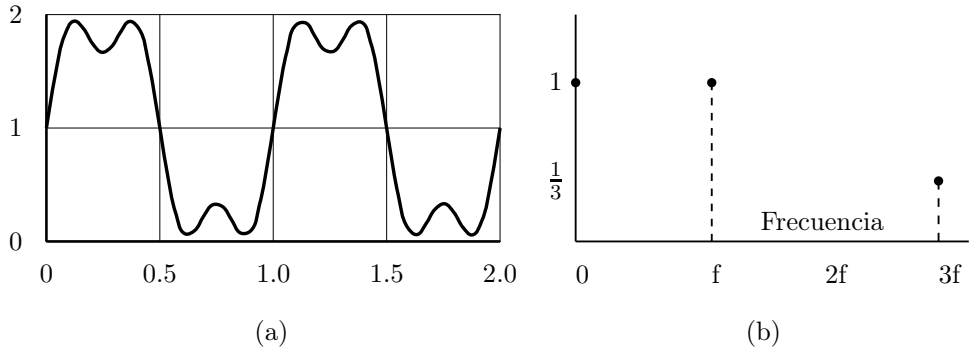


Figura 5.3: Dominios del tiempo y de la frecuencia con una componente dc.

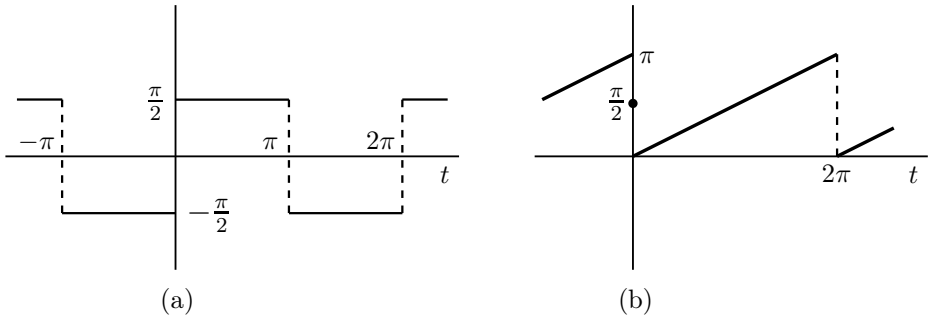


Figura 5.4: Dos funciones $g(t)$ para ser transformadas.

(igualmente espaciados). En tal caso, la transformada discreta de Fourier es:

$$\begin{aligned} G(f) &= \sum_{t=0}^{n-1} g(t) \left[\cos\left(\frac{2\pi ft}{n}\right) - i \sin\left(\frac{2\pi ft}{n}\right)\right] \\ &= \sum_{t=0}^{n-1} g(t)\, e^{-ift}, \quad 0 \le f \le n-1, \end{aligned} \tag{5.1}$$

y su inversa:

$$\begin{aligned} g(t) &= \frac{1}{n} \sum_{f=0}^{n-1} G(f) \left[\cos\left(\frac{2\pi ft}{n}\right) + i \sin\left(\frac{2\pi ft}{n}\right)\right] \\ &= \sum_{t=0}^{n-1} G(f)\, e^{ift}, \quad 0 \le t \le n-1. \end{aligned} \tag{5.2}$$

Nótese que $G(f)$ es compleja, por lo que se puede escribir como $G(f) = \mathcal{R}(f) + i\mathcal{I}(f)$. Para cualquier valor de f, la amplitud (o magnitud) de G viene dada por $|G(f)| = \sqrt{\mathcal{R}^2(f) + \mathcal{I}^2(f)}$.

Unas palabras sobre la terminología. En general, $G(f)$ se denomina la transformada de Fourier de $g(t)$. Sin embargo, si $g(t)$ es periódica, entonces $G(f)$ es su serie de Fourier.

Una función $f(t)$ es una *función de paso de banda (paso-banda o bandpass)* si su transformada de Fourier $F(\omega)$ está limitada a un intervalo de frecuencias $\omega_1 < |\omega| < \omega_2$, donde $\omega_1 > 0$ y ω_2 es finito.

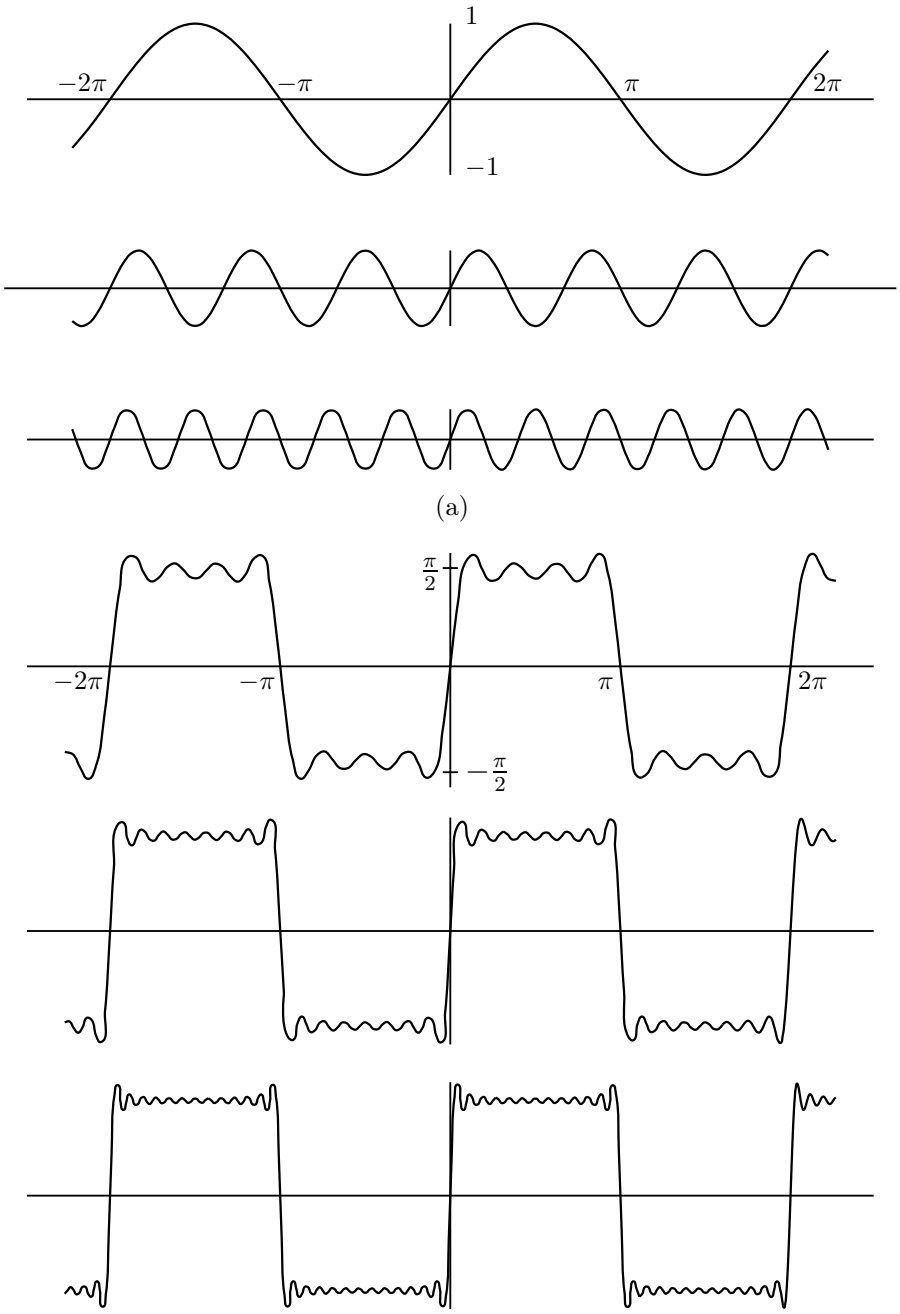


Figura 5.5: Los primeros tres términos y tres sumas parciales.

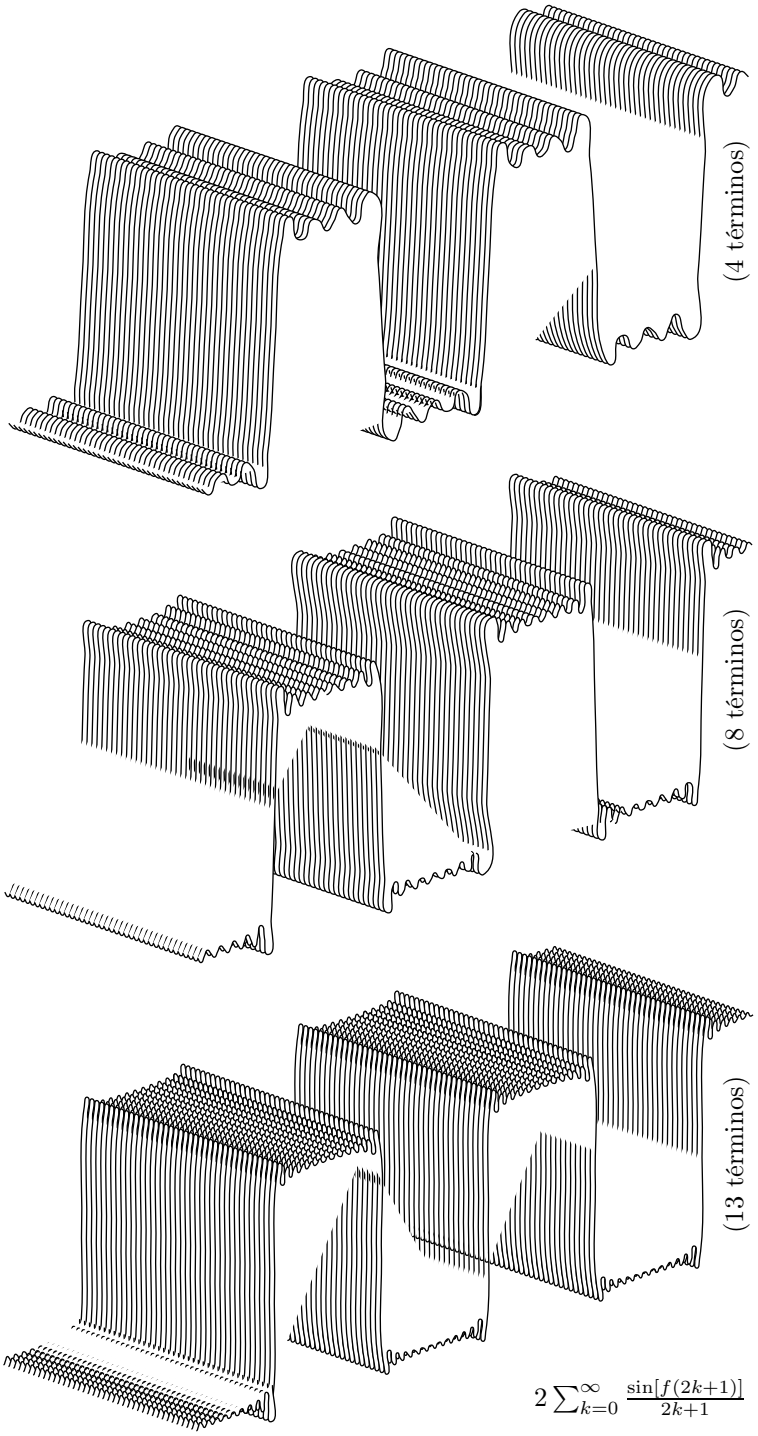
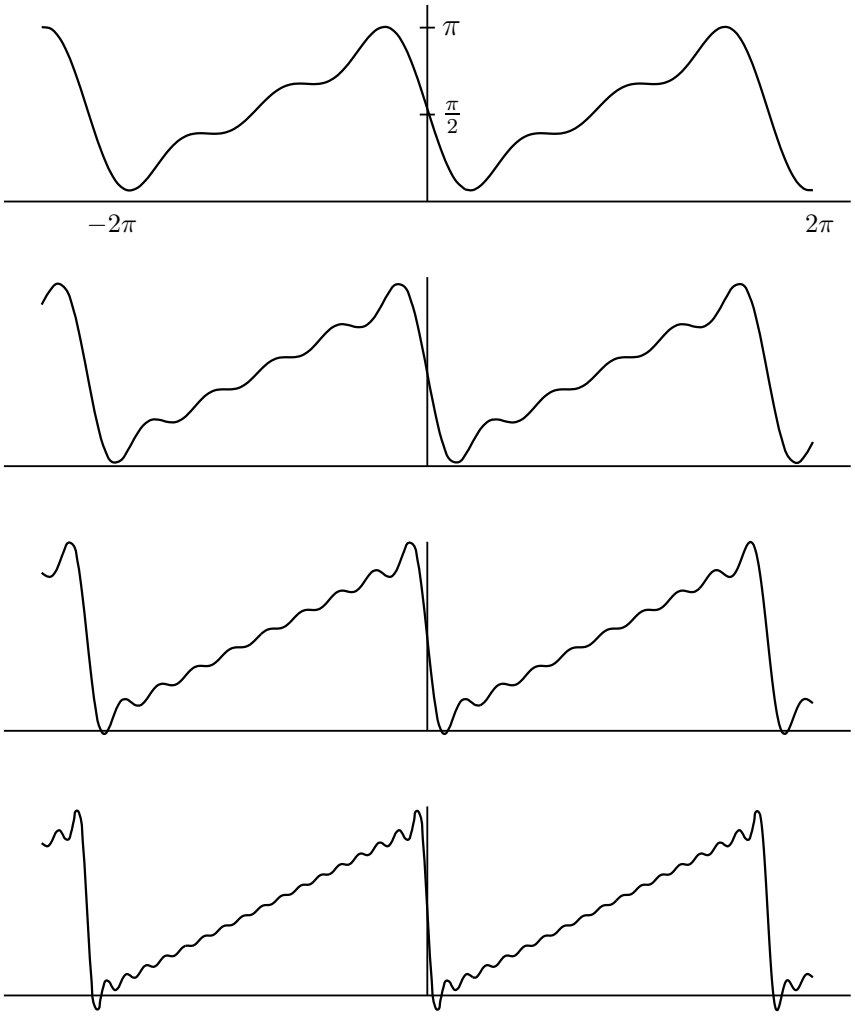


Figura 5.5: (c) Tres sumas parciales en 3D.



(a)

Figura 5.6: Cuatro sumas parciales.

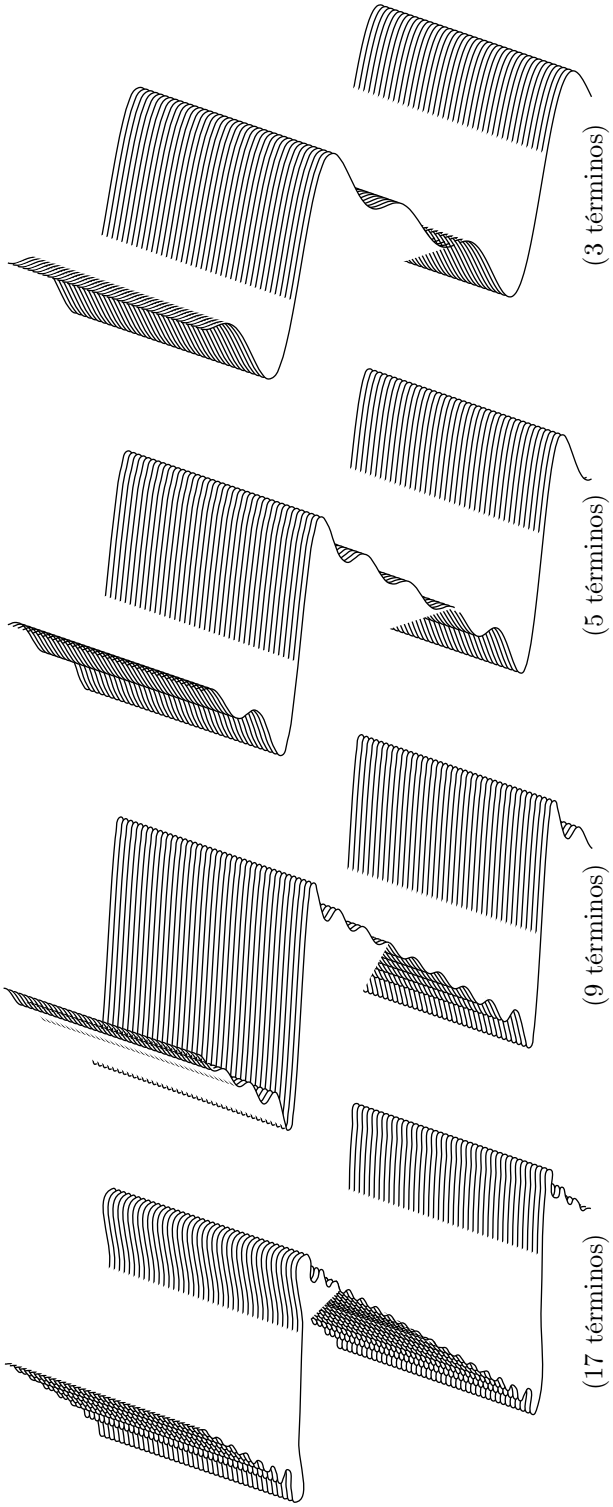


Figura 5.6: (b) Cuatro sumas parciales de $\frac{\pi}{2} - \sum_{k=1}^{\infty} \frac{\sin(k\,f)}{k}$ en 3D.

Observe que la función de la Figura 5.2c, obtenida sumando las sencillas funciones de las Figuras 5.2a,b, comienza a asemejarse a un pulso cuadrado. Resulta que podemos acercarla aún más a un pulso cuadrado (como el de la Figura 5.1a) mediante la adición de $(1/5)\sin(2\pi(5f)t)$, $(1/7)\sin(2\pi(7f)t)$, y así sucesivamente. Decimos que la serie de Fourier de una onda cuadrada con una amplitud A y frecuencia f es la suma infinita:

$$A\sum_{k=1,3,5,\ldots}^{\infty}\frac{1}{k}\sin(2\pi kft),$$

donde los términos sucesivos tienen cada vez amplitudes más pequeñas.

Aquí hay dos ejemplos que muestran la relación entre una función $g(t)$ y su expansión de Fourier $G(f)$. El primer ejemplo es la función paso o escalón de la Figura 5.4a, definida por:

$$g(t)=\begin{cases}\pi/2, & \text{para } 2k\pi\le t<(2k+1)\pi,\\ -\pi/2, & \text{para } (2k+1)\pi\le t<(2k+2)\pi,\end{cases}$$

donde k es un entero (positivo o negativo). La expansión de Fourier de $g(t)$ es:

$$G(f)=2\sum_{k=0}^{\infty}\frac{\sin[f(2k+1)]}{2k+1}=2\sin f+\frac{2\sin 3f}{3}+\frac{2\sin 5f}{5}+\cdots.$$

La Figura 5.5a muestra los primeros tres términos de esta serie, y la Figura 5.5b muestra tres sumas parciales, de los primeros cuatro, ocho y 13 términos (en la Figura 5.5c se pueden ver en 3D). Es evidente que estas sumas parciales rápidamente se aproximan a la función original.

El segundo ejemplo es la función de diente de sierra de la Figura 5.4b, definida por $g(t)=t/2$ para cada intervalo $[2k\pi,(2k+1)\pi$. Su expansión de Fourier es:

$$G(f)=\frac{\pi}{2}-\sum_{k=1}^{\infty}\frac{\sin(kf)}{k}.$$

La Figura 5.6a muestra cuatro sumas parciales, de los primeros tres, cinco, nueve y 17 términos, de esta expansión (en la Figura 5.6b se pueden ver en 3D).

5.3. El principio de incertidumbre

Estos ejemplos ilustran una importante relación entre el tiempo y el dominio de la frecuencia; es a saber, son *complementarios*. Cada uno de ellos complementa al otro en el sentido de que cuando uno de ellos está localizado el otro debe ser global. Consideremos, por ejemplo, una onda sinusoidal pura. Tiene una frecuencia, por lo que está bien localizada en el dominio de la frecuencia. Sin embargo, es infinitamente larga, por lo que en el dominio del tiempo es global. Por otro lado, una función puede estar localizada en el dominio del tiempo, tal como la espiga única (*single spike*) de la Figura 5.38a (pág. 591), pero siempre será invariablemente global en el dominio de la frecuencia; su expansión de Fourier contendrá muchas frecuencias (posiblemente incluso un número infinito). Esta relación entre los dominios del tiempo y de la frecuencia los hace complementarios, y popularmente se describe con el término *relación de incertidumbre* o *principio de incertidumbre*.

El principio de incertidumbre de Heisenberg, reconocido por primera vez en 1927 por Werner Heisenberg, es un principio físico muy importante. Dice que la posición y el momento son complementarios. Cuanto mejor conozcamos la posición de una partícula de materia, menos seguros estaremos de su momento. La razón de esta relación es la manera de medir las posiciones. Para encontrar una partícula en el espacio, tenemos que verla (con los ojos o con un instrumento). Ésto requiere luz brillante sobre

la partícula (ya sea la luz visible o de alguna otra longitud de onda), y es esta luz que perturba la partícula, la que la mueve desde su posición y, por lo tanto, cambia su momento. Pensamos en la luz como un conjunto de pequeñas unidades —fotones— que no tienen ninguna masa, pero tienen un momento. Cuando un fotón golpea una partícula, da a la partícula una "patada", que la aleja. Cuanto más grande sea la partícula, más pequeños son los efectos de la la patada, pero en principio, la relación complementaria entre la posición y el momento existe incluso para los objetos macroscópicos.

Es importante darse cuenta de que el principio de incertidumbre forma parte de la naturaleza; no es sólo el resultado de nuestro conocimiento imperfecto o nuestros instrumentos primitivos. Se expresa mediante la relación

$$\Delta x \Delta p \geq \frac{\hbar}{2} = \frac{h}{4\pi},$$

donde Δx y Δp son las incertidumbres en la posición y el momento de la partícula, y h es la constante de Planck ($6{,}626176 \times 10^{-27}\text{erg} \cdot \text{seg}$). La cuestión es que h es muy pequeña, tanto como:

$$\hbar \stackrel{\text{def}}{=} \frac{h}{2\pi} = 1{,}05 \times 10^{-34} = 0{,}00000000000000000000000000000000105\,\text{julios} \cdot \text{seg}.$$

(Un erg y un julio son unidades de energía, cuyos valores precisos son irrelevantes para esta discusión.) Es por éso que no apreciamos la incertidumbre posición/fuerza en la vida cotidiana. Un sencillo ejemplo debería dejar ésto claro. Supongamos que la masa de una bola móvil es de 0,15 Kg y medimos su velocidad con una precisión de 0,05 m/seg (que es aproximadamente 1 mph). El momento es el producto de la masa y de la velocidad, por lo que si la incertidumbre en la velocidad de la bola es de 0,05 m/seg, la incertidumbre en su momento es $\Delta p = 0{,}15 \times 0{,}05 = 0{,}0075$ Kg $\cdot$ m/seg. El principio de incertidumbre dice que $\Delta x \times 0{,}0075 \geq 1{,}05 \times 10^{-34}/2$, lo que implica que la incertidumbre en la posición de la bola es del orden de 70×10^{-34}m ó 7×10^{-30}mm, ¡demasiado pequeño para medirlo!

El principio de incertidumbre es un caso especial del principio de la complementariedad. La posición y el momento son cantidades complementarias, pero hay otros pares de cantidades físicas que son complementarias, tales como el tiempo y la energía.

Aunque estamos hablando de la física, aquí hay otra similitud entre los dominios del tiempo y de la frecuencia, y la realidad física. Los objetos reales (es decir, los objetos con masa) tienen un propiedad llamada "dualidad onda-partícula". Ésto es cierto para todos los objetos; pero es perceptible y puede ser medida, solamente en las partículas microscópicas. Esta propiedad significa que cada objeto tiene una onda asociada con ella. La amplitud de esta onda (en realidad, el cuadrado del valor absoluto de la amplitud) en una región dada describe la probabilidad de encontrar el objeto en esa región. Cuando observamos el objeto, lo habitual es que se encuentre en la región donde su función de onda tiene la mayor amplitud.

La onda asociada con un objeto generalmente no se transmite a lo largo de todo el universo, sino que se limita a una pequeña región en el espacio. Esta onda "localizada" se llama *paquete de ondas*. Tal paquete no tiene una longitud de onda particular, sino que está formada por ondas de muchas longitudes de onda. El momento de un objeto depende de su longitud de onda, por lo que un objeto que está localizado en el espacio no tiene un momento específico y bien definido; tiene muchos momentos. En otras palabras, hay una incertidumbre en el momento de la partícula.

La incertidumbre de Nuggets

Puede encontrarse una discusión completa del principio de incertidumbre de Heisenberg en el Apéndice. Por otra parte, puede que no.

La incertidumbre es la esencia del romanticismo. Lo que no sabemos es lo que nos intriga.

5.4. Compresión de imágenes de Fourier

Aplicamos ahora los conceptos de dominio en tiempo y en frecuencia a las imágenes digitales. Imagine una fotografía en escala de grises escaneada línea por línea. Para fines prácticos, podemos asumir que la fotografía tiene una resolución infinita (sus tonos de gris pueden variar de forma continua). Un escaneo ideal, por tanto, produce una secuencia infinita de números que pueden ser considerados los valores de una función $I(t)$ de intensidad (continua). En la práctica, podemos almacenar sólo una cantidad limitada de datos en la memoria, por lo que tenemos que seleccionar un número finito de valores, de $I(1)$ a $I(n)$. Este proceso se conoce como *muestreo* (*sampling*).

Intuitivamente, el muestreo se asemeja a un equilibrio entre calidad y precio. Cuanto mayor es la muestra, mejor será la calidad de la imagen final, pero requiere más hardware (más memoria y mayor resolución de pantalla), generando mayores costos. Esta conclusión intuitiva, sin embargo, no es del todo cierto. La teoría de muestreo nos dice que podemos muestrear una imagen y reconstruirla posteriormente en la memoria sin pérdida de calidad si logramos hacer lo siguiente:

1. Transformar la función de intensidad del dominio del tiempo $I(t)$ en el dominio de la frecuencia $G(f)$.

2. Encontrar la frecuencia máxima f_m.

3. Muestrear $I(t)$ a una velocidad ligeramente superior a $2f_m$ (e.g., si $f_m = 22\,000$ Hz, generar muestras a razón de 44 100 Hz).

4. Almacenar los valores muestreados en el mapa de bits. La imagen resultante debería ser de igual calidad que la original en la fotografía.

Hay dos puntos a tener en cuenta. El primero es que f_m puede ser infinito. En este caso, debe seleccionarse un valor de f_m tal que las frecuencias que son mayores que f_m no contribuyan mucho (tengan amplitudes bajas). Hay una cierta pérdida de calidad de imagen en dicho caso. El segundo punto es que el mapa de bits (y, en consecuencia, la resolución) puede ser demasiado pequeño para la muestra generada en el paso 3. En tal caso, debe tomarse una muestra más pequeña, resultando de nuevo en una pérdida de calidad de imagen.

El resultado anterior fue demostrado por Harry Nyquist [Nyquist 28], y la cantidad $2f_m$ se denomina la *tasa o frecuencia de Nyquist* (*Nyquist rate*). Se utiliza en muchas situaciones prácticas. El rango de audición humano, por ejemplo, está entre 16 Hz y 22 000 Hz. Cuando el sonido es digitalizado en alta calidad (como la música grabada en un CD), se muestrea a una velocidad de 44 100 Hz. Cualquier valor menor que ése provoca distorsiones.

¿Por qué es, que el muestreo a una tasa de Nyquist es suficiente para restaurar la señal original? Parece que el muestreo ignora el comportamiento de la señal analógica entre las muestras, y por lo tanto, se puede perder información importante. ¿Qué garantiza que la señal no irá hacia arriba o hacia abajo considerablemente entre muestras consecutivas? En principio, tal comportamiento puede ocurrir, pero en la práctica, todas las señales analógicas tienen un límite de respuesta en frecuencia porque son creados por fuentes (tales como, el micrófono, el altavoz, o la boca), cuya velocidad de respuesta es limitada. Por consiguiente, la velocidad a la que una señal real puede cambiar es limitada, lo que hace posible predecir la forma en que puede variar de muestra a muestra. Decimos que el *ancho de banda finito* de las señales reales es lo que hace posible su digitalización.

> Fourier es un poema matemático.
>
> —William Thomson (Lord Kelvin)

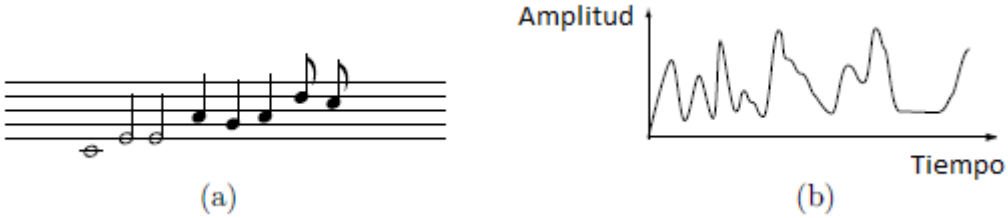


Figura 5.7: Dos representaciones de la música.

La transformada de Fourier es útil y popular, aplicándose en muchas áreas. Tiene, sin embargo, un inconveniente: muestra el contenido en frecuencia de una función $f(t)$, pero no especifica dónde (i.e., para qué valores de t) tiene la función frecuencias bajas o altas. La razón de ésto es que las funciones base (seno y coseno) utilizadas por esta transformada son infinitamente largas. Recogen las diferentes frecuencias de $f(t)$, independientemente de donde que se encuentren.

Una transformada mejor debería especificar el contenido en frecuencia de $f(t)$ como una función de t. En lugar de producir una función unidimensional (en el caso continuo) o una matriz unidimensional de números (en el caso discreto), debería producir una función bidimensional o una matriz de números $W(a,b)$ que describa el contenido en frecuencia de $f(t)$ para distintos valores de t. Una columna $W(a_i,b)$ de esta matriz (donde $i=1,2,\ldots,n$) indica el espectro de frecuencia de $f(t)$ para un cierto valor (o intervalo de valores) de t. Una fila $W(a,b_i)$ contiene números que describen cuánto de una cierta frecuencia (o rango de frecuencias) $f(t)$ tiene para cualquier t dado.

La *transformada wavelet* es un método que cumple éso. Ha sido desarrollado, investigado y aplicado a muchas áreas de la ciencia y de la ingeniería desde principios de 1980, aunque sus raíces son mucho más tempranas. La idea principal es seleccionar una onda pequeña (*wavelet*) madre, una función que es distinta de cero en un cierto intervalo pequeño, y utilizarla para explorar las propiedades de $f(t)$ en ese intervalo. La wavelet madre es entonces trasladada a otro intervalo de t y utilizada de nuevo de la misma manera. El parámetro b especifica la traslación. Diferentes resoluciones de frecuencia de $f(t)$ son exploradas escalando la wavelet madre con un factor de escala a.

Antes de entrar en detalles, ilustramos la relación entre la representación "normal" (dominio del tiempo) de una función y su transformación bidimensional observando la notación musical estándar. Esta notación, utilizada en occidente desde hace cientos de años, es bidimensional. Las notas se escriben en un pentagrama bidimensional, donde el eje horizontal denota el tiempo (de izquierda a derecha) y el eje vertical denota el tono. Cuanto más alta está la nota en el pentagrama, mayor es el tono del sonido reproducido. Además, la forma de una nota indica su duración. La Figura 5.7a muestra, de izquierda a derecha, una nota redonda (llamada "C" en los EE.UU. y "do" en Europa), dos blancas, tres negras, y dos corcheas. Además del pentagrama y las notas, la notación musical incluye muchos otros símbolos e indicaciones del compositor. Sin embargo, la cuestión es que la misma música también puede ser representada por una función unidimensional que simplemente describa la amplitud del sonido como una función del tiempo (Figura 5.7b). Las dos representaciones son matemáticamente equivalentes, pero se utilizan de forma diferente en la práctica. La representación bidimensional es utilizada por los músicos para realizar música en la realidad. La representación unidimensional se utiliza para reproducir la música que ya ha sido realizada y registrada.

◇ **Ejercicio 5.1 (sol. en pág. 1095):** Propóngase un ejemplo de una notación común que tenga una representación bidimensional familiar utilizada por los seres humanos y una representación unidimensional desconocida utilizada por las máquinas.

El principio de análisis de una función en el tiempo y en la frecuencia se puede aplicar a compresión de imágenes porque las imágenes contienen áreas que presentan "tendencias" y áreas con "anomalías".

Una tendencia (*trend*) es una característica de la imagen que involucra sólo a unas pocas frecuencias (está localizada en frecuencia) pero que se extiende espacialmente. Un ejemplo típico es un área de la imagen donde el brillo varía gradualmente. Una anomalía es una imagen característica que involucra a varias frecuencias, pero está localizada espacialmente (se concentra en un área pequeña de la imagen). Un ejemplo es una arista.

Comenzamos buscando funciones que puedan servir como wavelet, definiendo la transformada wavelet continua (CWT) y su inversa, e ilustrando la forma en que trabaja la CWT. A continuación, mostramos en detalle cómo puede aplicarse la wavelet de Haar a la compresión de imágenes. Ésto conduce naturalmente a los conceptos de bancos de filtros y a la *transformada wavelet discreta* (Sección 5.8). El *esquema de elevación* (*lifting scheme*) para el cálculo de la transformada de wavelet y su inversa se describen en la Sección 5.11. A ésto sigue una descripción de varios métodos de compresión que, o bien emplean la transformada wavelet, o bien comprimen los coeficientes que resultan de tal transformación.

5.5. La CWT y su inversa

La transformada wavelet continua (CWT, [Lewalle 95] y [Rao y Bopardikar 98]) de una función $f(t)$ involucra a una wavelet madre $\psi(t)$. La wavelet madre puede ser cualquier función continua, real o compleja que satisfaga las siguientes propiedades:

1. El área total bajo la curva de la función es cero, i.e.:

$$\int_{-\infty}^{\infty} \psi(t)\, dt = 0.$$

2. El área total de $|\psi(t)|^2$ es finita, i.e.:

$$\int_{-\infty}^{\infty} |\psi(t)|^2\, dt < \infty.$$

   Esta condición implica que la integral del cuadrado de la wavelet tiene que existir. Podemos decir también que una wavelet tiene que ser *cuadrado integrable*, o que pertenece al conjunto $\mathcal{L}^2(R)$ de todas las funciones cuadrado integrables.

3. La condición de admisibilidad, discutida más adelante.

La energía de una función

Una función $y = f(x)$ relaciona cada valor de la variable independiente x con un valor de y. El trazado de estos pares de valores produce una representación de la función como una curva en dos dimensiones. La *energía* de una función se define en términos de la zona delimitada por esta curva y el eje x. Tiene sentido decir que una curva que permanece cerca del eje tiene poca energía, mientras que una curva que gasta "tiempo" lejos del eje x tiene más energía. Los valores negativos de y empujan la curva bajo el eje x, donde su área se considera negativa, por lo que la energía de $f(t)$ se define como el área bajo la curva de la función no negativa $f(x)^2$. Si la función es compleja, su valor absoluto se usa en los cálculos del área, por lo que el energía de $f(x)$ se define como:

$$\int_{-\infty}^{\infty} |f(x)|^2\, dx.$$

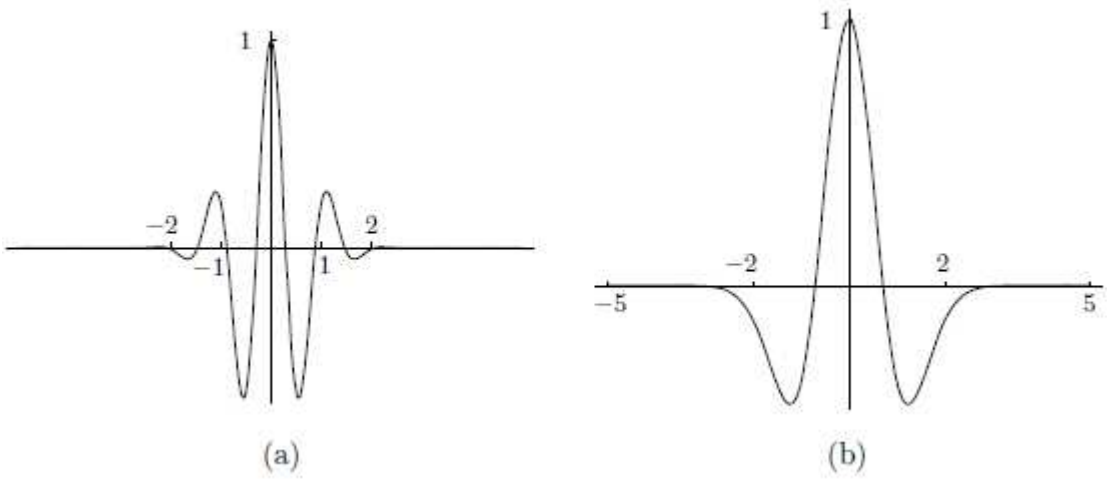


Figura 5.8: Las wavelets (a) de Morlet (b) sombrero Mexicano.

La propiedad 1 sugiere una función que oscila por encima y por debajo del eje t. Tal función tiende a tener una apariencia ondulada. La propiedad 2 implica que la *energía* de la función es finita, lo que sugiere que la función está localizada en algún intervalo finito y es cero, o casi cero, fuera de este intervalo. Estas propiedades justifican el nombre "wavelet (onda pequeña)". Un número infinito de funciones satisfacen estas condiciones, y algunas de ellas han sido investigadas y se utilizan comúnmente para las transformadas wavelet. La Figura 5.8a muestra la wavelet de Morlet, definida por:

$$\psi(t) = e^{-t^2} \cos\left(\pi t \sqrt{\frac{2}{\ln 2}}\right) \approx e^{-t^2} \cos(2{,}885\pi t).$$

Ésta es una curva de coseno cuyas oscilaciones se amortiguan por el factor exponencial (o Gaussiano). Más del 99 % de su energía se concentra en el intervalo $-2{,}5 \leq t \leq 2{,}5$. La Figura 5.8b muestra la wavelet llamada sombrero Mexicano, que se define como:

$$\psi(t) = \left(1 - 2t^2\right) e^{-t^2}.$$

Ésta es la segunda derivada de la función (negativa) gaussiana $-0{,}5e^{-t^2}$.

Una vez elegida una wavelet $\psi(t)$, la CWT de una función integrable cuadrada $f(t)$ se define como:

$$W(a,b) = \int_{-\infty}^{\infty} f(t) \frac{1}{\sqrt{|a|}} \psi^* \left(\frac{t-b}{a}\right) dt. \tag{5.3}$$

La transformada es una función de los dos parámetros reales a y b. El * denota el complejo conjugado. Si definimos una función $\psi_{a,b}(t)$ como:

$$\psi_{a,b}(t) = \frac{1}{\sqrt{|a|}} \psi \left(\frac{t-b}{a}\right),$$

podemos escribir la Ecuación (5.3) en la forma:

$$W(a,b) = \int_{-\infty}^{\infty} f(t) \psi_{a,b}(t) dt. \tag{5.4}$$

Matemáticamente, la transformada es el *producto interno* de las dos funciones $f(t)$ y $\psi_{a,b}(t)$. La cantidad $1/\sqrt{|a|}$ es un factor de normalización que asegura que la energía de $\psi(t)$ permanezca independiente de a y b, i.e.,

$$\int_{-\infty}^{\infty} |\psi_{a,b}(t)|^2 \, dt = \int_{-\infty}^{\infty} |\psi(t)|^2 \, dt.$$

Para cualquier a, $\psi_{a,b}(t)$ es una copia de $\psi_{a,0}$ desplazada b unidades a lo largo del eje de tiempo. Por consiguiente, b es un *parámetro de traslación*. Establecer $b = 0$ muestra que

$$\psi_{a,0}(t) = \frac{1}{\sqrt{|a|}} \psi\left(\frac{t}{a}\right),$$

lo que implica que a es un parámetro de escala (o una dilatación). Los valores $a > 1$ ensanchan la wavelet, mientras que los valores $0 < a < 1$ la estrechan.

Puesto que las wavelets se utilizan para transformar una función, es necesaria la transformada inversa. Denotamos por $\Psi(\omega)$ a la transformada de Fourier de $\psi(t)$:

$$\Psi(\omega) = \int_{-\infty}^{\infty} \psi(t) \, e^{-i\omega t} dt.$$

Si $W(a,b)$ es la CWT de una función $f(t)$ con una wavelet $\psi(t)$, entonces la CWT inversa está definida por:

$$f(t) = \frac{1}{C} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \frac{1}{|a|^2} W(a,b) \, \psi_{a,b}(t) \, da \, db,$$

donde la cantidad C se define como:

$$C = \int_{-\infty}^{\infty} \frac{|\Psi(\omega)|^2}{|\omega|} d\omega.$$

La CWT inversa existe si C es positivo y finito. Dado que C se define mediante Ψ, que a su vez se define mediante la wavelet $\psi(t)$, el requisito de que C sea positivo y finito impone otra restricción, llamada *condición de admisibilidad*, sobre la elección de la wavelet.

Es mejor pensar en la CWT como una matriz de números que son productos internos de $f(t)$ y $\psi_{a,b}(t)$. Las filas de la matriz corresponden a los valores de a, y las columnas son indexadas mediante b. El *producto interno* de dos funciones $f(t)$ y $g(t)$ se define como:

$$\langle f(t), g(t) \rangle = \int_{-\infty}^{\infty} f(t) \, g^*(t) \, dt,$$

por lo que la CWT es el producto interno:

$$\langle f(t), \psi_{a,b}(t) \rangle = \int_{-\infty}^{\infty} f(t) \, \psi_{a,b}(t) \, dt,$$

Después de esta introducción, ahora estamos en condiciones de explicar el significado intuitivo de la CWT. Empezamos con un ejemplo sencillo: la CWT de una onda sinusoidal, donde el sombrero Mexicano es elegido como wavelet. La Figura 5.9a muestra una onda sinusoidal con dos copias de la wavelet. La copia 1 se posiciona en un punto donde la onda sinusoidal tiene un máximo. En este punto hay una buena correlación entre la función que se analiza (el seno) y la wavelet. La wavelet *replica* las características de la onda sinusoidal. Como resultado, el producto interno del seno y la wavelet es un número positivo grande. Por el contrario, la copia 2 se ubica donde la onda sinusoidal tiene un mínimo. En ese punto, la onda y la wavelet son casi imágenes especulares una de la otra. Cuando la

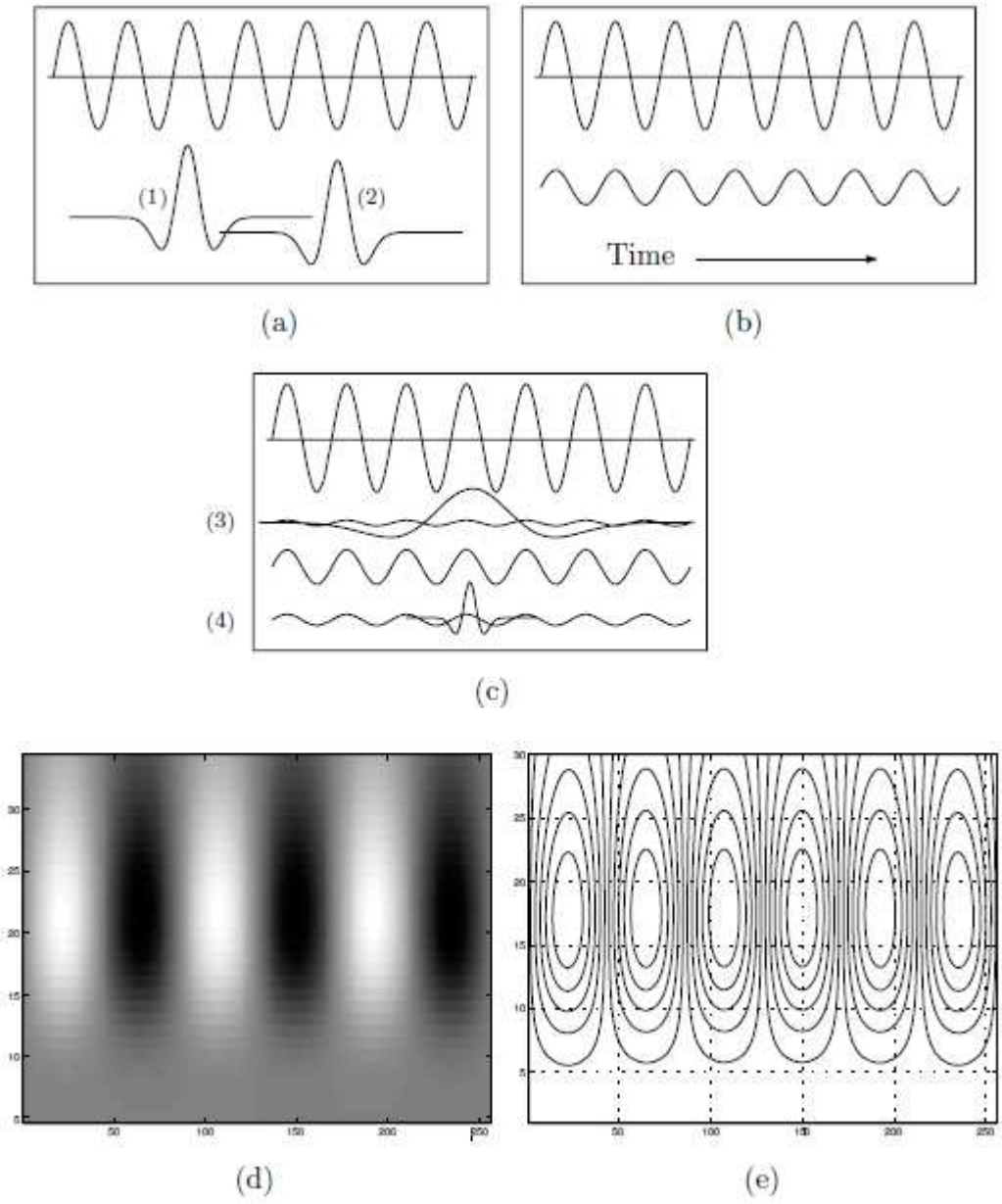


Figura 5.9: La transformada wavelet continua de una onda sinusoidal pura.

```
t=linspace(0,6*pi,256);
sinwav=sin(t);
plot(t,sinwav)
cwt=CWT(sinwav,10,'Sombrero');
axis('ij'); colormap(gray);
imagesc(cwt')
x=1:256; y=1:30;
[X,Y]=meshgrid(x,y);
contour(X,Y,cwt',10)
```

```
t=linspace(-10,10,256);
sombr=(1-2*t.^2).*exp(-t.^2);
plot(t,sombr)
```

Código para la Figura 5.9.

onda sinusoidal es positiva, la wavelet es negativa y viceversa. El producto de la onda y la wavelet en este punto es negativo, y la CWT (la integral o el producto interno) se convierte en un gran número negativo. Entre los puntos 1 y 2, la CWT cambia de positivo, a cero, a negativo.

Como la wavelet es trasladada a lo largo de la onda seno, de izquierda a derecha, la CWT alterna entre positivo y negativo y produce la pequeña onda mostrada en la Figura 5.9b. Esta forma es el resultado de la estrecha relación entre la función que está siendo analizada (la onda seno) y el análisis de la wavelet. Ambas tienen formas similares y también frecuencias similares.

Ahora extendemos el análisis a diferentes frecuencias. Ésto se efectúa escalando la wavelet. La Figura 5.9c muestra (en la parte 3) lo que ocurre cuando la onda es estirada de forma que cubra varios períodos de la onda sinusoidal. La traslación de la wavelet de izquierda a derecha no afecta, por tanto, a la correspondencia entre ella y la onda sinusoidal, resultando que la CWT varía sólo un poco. Cuanto más ancha sea la onda, más cercana a una costante está la CWT. Nótese cómo la amplitud de la onda se ha reducido, lo que reduce su área y produce una constante pequeña. Algo similar ocurre en la parte 4 de la figura, donde la wavelet se ha reducido y es mucho más estrecha que un ciclo de la onda sinusoidal. Puesto que la wavelet es tan "fina", el producto interno de ésta y la onda seno es siempre un número pequeño (positivo o negativo), independientemente de la posición precisa de la wavelet relativa a la onda sinusoidal. Vemos que la traslación de la wavelet no afecta mucho a su correspondencia con la onda seno, resultando una CWT que está cercana a una constante.

Los resultados de la traslación de la wavelet, escalandola, y trasladándola una y otra vez se resumen en la Figura 5.9d. Ésta es una gráfica de densidad de la función de transformación $W(a,b)$ donde el eje horizontal corresponde a los valores de b (traslación) y el eje vertical corresponde a los valores de a (escalado). La Figura 5.9e es un gráfico de contorno de la misma $W(a,b)$. Estos diagramas muestran que existe una buena correlación entre la función y la wavelet a una determinada frecuencia (la frecuencia de la onda seno). A otras frecuencias la correspondencia se deteriora, produciendo una transformación que se acerca cada vez más a una constante.

Así es cómo la CWT proporciona un análisis tiempo–frecuencia de una función $f(t)$. El resultado es una función $W(a,b)$ de dos variables que muestra la analogía entre $f(t)$ y la wavelet a diferentes frecuencias de la wavelet y en diferentes instantes. Es obvio que la calidad de la correspondencia depende de la elección de wavelet. Si la wavelet es muy diferente de $f(t)$ a cualquier frecuencia y cualquier tiempo, todos los valores del resultado $W(a,b)$ van a ser pequeños y no presentarán mucha variación. Como consecuencia, cuando se utilizan wavelets para comprimir imágenes, deben seleccionarse distintas wavelets para tipos de imágenes diferentes (binivel, de tonos continuos y de tonos discretos), pero la elección precisa de la wavelet es aún objeto de mucha investigación.

Puesto que tanto nuestra función como nuestra wavelet son funciones sencillas, en este caso es posible calcular la integral de la CWT como una integral indefinida, y se conoce con una fórmula cerrada para $W(a,b)$. En el caso general, sin embargo, esto es imposible —ya sea en la práctica, ya sea al principio— y los cálculos deben realizarse numéricamente.

El siguiente ejemplo es ligeramente más complejo y conduce a una mejor comprensión de la CWT. La función que está siendo analizada esta vez es una onda seno con una frecuencia acelerada: la llamada *chirp* (*chirrido*). Viene dada por $f(t) = \sin\left(t^2\right)$, y se muestra en la Figura 5.10a. La wavelet es el mismo sombrero Mexicano. Los lectores que hayan repasado el ejemplo anterior cuidadosamente no tendrán problemas en la interpretación de la CWT de este ejemplo. Viene dado por Figura 5.10b,c y muestra cómo la frecuencia de $f(t)$ aumenta con el tiempo.

Estos dos ejemplos ilustran cómo se utiliza la CWT para analizar la frecuencia en tiempo de una función $f(t)$. Es evidente que el usuario necesita experiencia tanto con el fin de seleccionar la onda correcta para el trabajo como para interpretar los resultados. Sin embargo, con la experiencia adecuada, la CWT puede ser una poderosa herramienta analítica en manos de científicos y ingenieros.

◇ **Ejercicio 5.2 (sol. en pág. 1095):** Experiméntese con la CWT tratando de resolver el siguiente análisis. Selecciónese la función $f(t) = 1 + \sin\left(\pi t + t^2\right)/8$ como candidata para el análisis. Ésta es

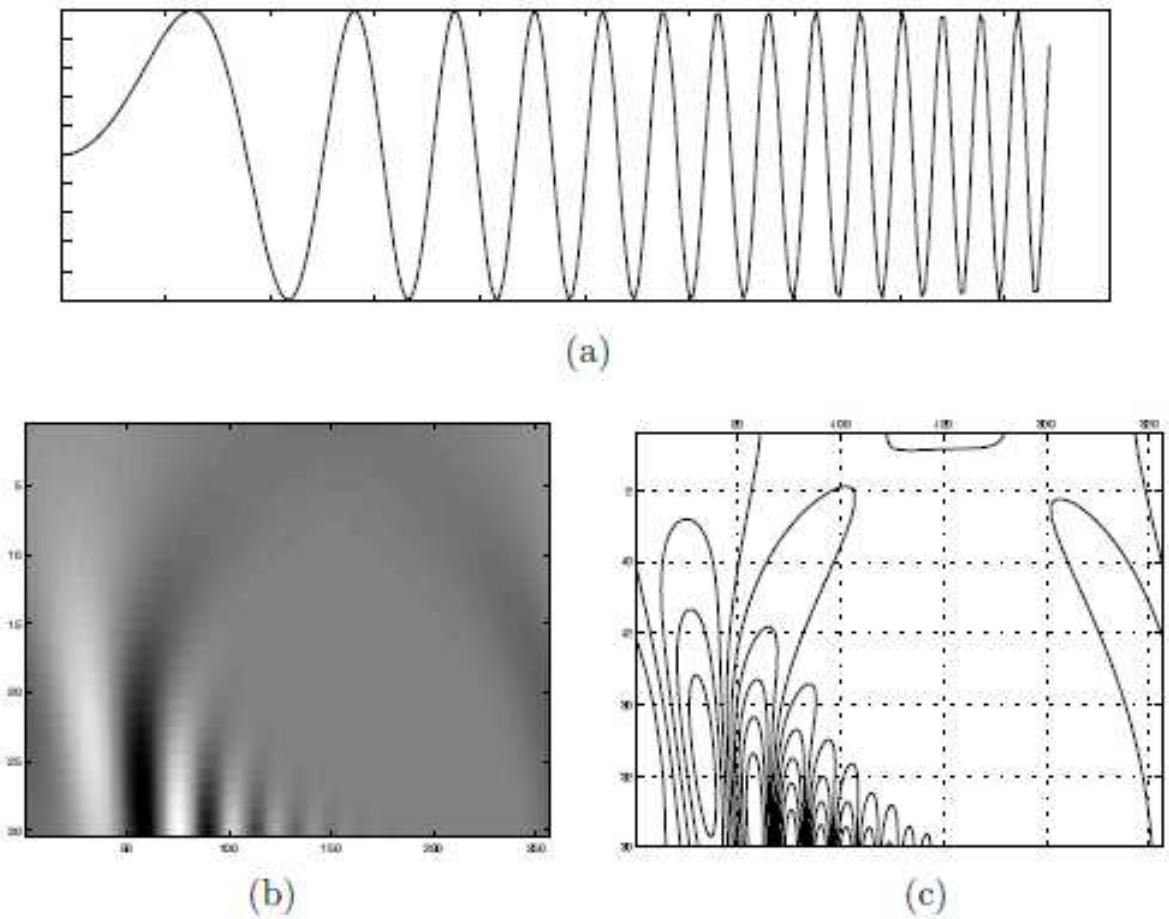


Figura 5.10: La transformada wavelet continua de un chirrido (chirp).

```
t=linspace(0,6*pi,256);
sinwav=sin(t);
plot(t,sinwav)
cwt=CWT(sinwav,10,'Sombrero');
axis('ij'); colormap(gray);
imagesc(cwt')
x=1:256; y=1:30;
[X,Y]=meshgrid(x,y);
contour(X,Y,cwt',10)
```

```
t=linspace(-10,10,256);
sombr=(1-2*t.^2).*exp(-t.^2);
plot(t,sombr)
```

Código para la Figura 5.10.

una de onda sinusoidal que oscila entre $y = 1 - 1/8$ e $y = 1 + 1/8$ con una frecuencia $2\pi + t$, que aumenta con t. Como wavelet, selecciónese el sombrero Mexicano. Represéntense varias copias trasladadas de la wavelet frente a $f(t)$, luego utilícese software apropiado, tal como *Mathematica* o Matlab, para calcular y mostrar la CWT.

Las wavelets también están sujetas al principio de incertidumbre (Sección 5.3). La wavelet de Haar está muy bien localizada en el dominio del tiempo, pero se ensancha en el dominio de la frecuencia debido a la aparición de bandas laterales en el espectro de Fourier. En contraste, la wavelet del sombrero Mexicano y, especialmente, la wavelet de Morlet tienen frecuencias más concentradas, pero se distribuyen en el tiempo. Un resultado importante del principio de incertidumbre es que es imposible lograr un mapeo completo simultáneo en tiempo y frecuencia. Las wavelets proporcionan un compromiso, o una solución óptima cercana, a este problema, y es ésta una característica que las hace superiores a los análisis de Fourier.

Podemos comparar el análisis de la wavelet para mirar un objeto complejo primero desde la distancia, luego más cerca, luego a través de una lupa, y finalmente a través de un microscopio. Cuando se mira desde una distancia, vemos la forma general del objeto, pero no cualquier pequeño detalle. Al mirar a través de un microscopio, vemos pequeños detalles, pero en general no la forma. Por esta razón, es importante el análisis a diferentes escalas. Cuando cambiamos la escala de la wavelet, obtenemos nueva información acerca de la función que se analiza.

5.6. La transformada de Haar

La información que está siendo transformada mediante una wavelet en aplicaciones prácticas, tales como digitalización de sonidos e imágenes, es discreta, formada por números individuales. Ésta es la razón por la que, en la práctica, se utiliza la transformada wavelet discreta, y no la continua. La transformada wavelet discreta se describe de forma general en la Sección 5.8, pero precedemos esta discusión presentando un ejemplo sencillo de este tipo de transformación, a saber, la transformada wavelet de Haar.

El uso de la transformada de Haar para la compresión de imágenes se describe aquí desde un punto de vista práctico. En primer lugar, mostramos cómo se aplica esta transformación a la compresión de imágenes en escala de grises; a continuación, mostramos cómo puede este método ser extendido a imágenes en color. La transformada de Haar [Stollnitz et al. 96] fue introducida en la Sección 4.5.3.

La transformada de Haar utiliza una función de escala $\phi(t)$ y una wavelet $\psi(t)$, ambas mostradas en la Figura 5.11a, para representar un gran número de funciones $f(t)$. La representación es la suma infinita:

$$f(t) = \sum_{k=-\infty}^{\infty} c_k \phi(t-k) + \sum_{k=-\infty}^{\infty} \sum_{j=0}^{\infty} d_{j,k} \psi\left(2^j t - k\right),$$

donde c_k y $d_{j,k}$ son coeficientes para ser calculados.

La función de escala base $\phi(t)$ es el pulso unidad:

$$\phi(t) = \begin{cases} 1, & 0 \le t < 1, \\ 0, & \text{en otro caso.} \end{cases}$$

La función $\phi(t-k)$ es una copia de $\phi(t)$, desplazada k unidades a la derecha. Similarmente, $\phi(2t-k)$ es una copia de $\phi(t-k)$ escalada a la mitad del ancho de $\phi(t-k)$. Las copias desplazadas se utilizan para aproximar $f(t)$ en instantes t diferentes. Las copias a escala se utilizan para aproximar $f(t)$ en altas resoluciones. La Figura 5.11b muestra las funciones $\phi\left(2^j t - k\right)$ para $j = 0, 1, 2$, y 3, y para $k = 0, 1, \ldots, 7$.

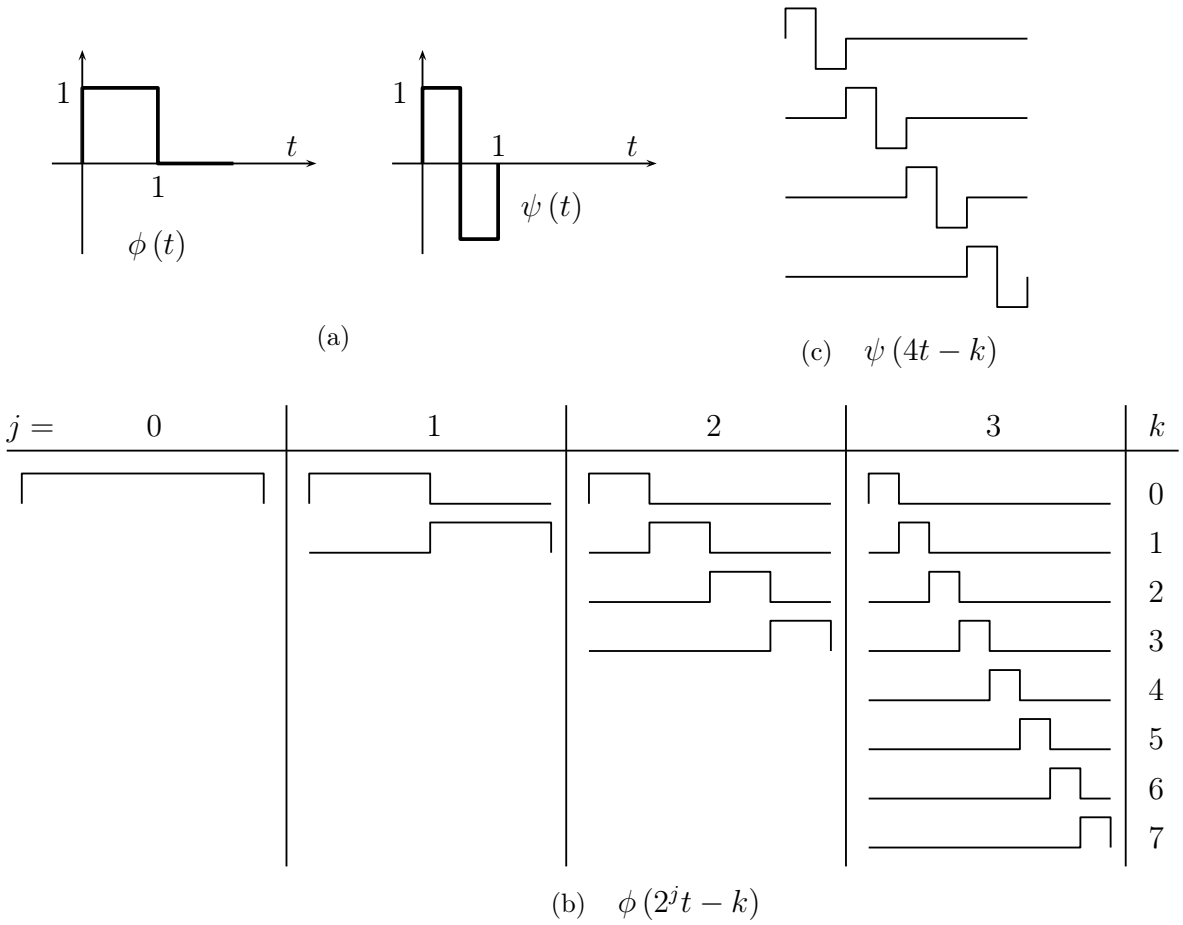


Figura 5.11: Las funciones de Haar de escala base y wavelet.

La wavelet de Haar base es la función escalón:

$$\psi(t)=\begin{cases}1, & 0\le t<0{,}5,\\ -1, & 0{,}5\le t<1.\end{cases}$$

A partir de ésto podemos ver que la wavelet de Haar general $\psi\left(2^j t-k\right)$ es una copia de $\psi(t)$ desplazada k unidades hacia la derecha y escalada de forma que su anchura total es $1/2^j$.

◇ **Ejercicio 5.3 (sol. en pág. 1095):** Represéntense las cuatro wavelets de Haar $\psi\left(2^2 t-k\right)$ para $k=0,1,2$,y 3.

Tanto $\phi\left(2^j t-k\right)$ como $\psi\left(2^j t-k\right)$ son distintas de cero en un intervalo de $1/2^j$ de ancho. Este intervalo es su *soporte*. Dado que este intervalo suele ser corto, se dice que estas funciones tienen *soporte compacto*.

Ilustramos la transformación base sobre la sencilla función de paso:

$$f(t)=\begin{cases}5, & 0\le t<0{,}5,\\ 3, & 0{,}5\le t<1.\end{cases}$$

Es fácil ver que $f(t) = 4\phi(t) + \psi(t)$. Decimos que los pasos originales $(5, 3)$ han sido transformados a la (baja resolución) media 4 y a la (alta resolución) de detalle 1. Utilizando notación matricial, ésto puede ser expresado (hasta un factor de $\sqrt{2}$) como $(5, 3)\,\mathbf{A}_2 = (4, 1)$, donde $\mathbf{A}_2$ es la matriz de la tansformada de Haar de orden 2 de la Ecuación (4.11).

Una imagen es una matriz bidimensional de valores de píxel. Para ilustrar cómo se utiliza la transformada de Haar para comprimir una imagen, comenzamos con una sola fila de valores de píxel, i.e., una matriz unidimensional de n valores. Para simplificar, asumimos que n es una potencia de 2. (Utilizamos esta suposición en este capítulo, pero no hay ninguna pérdida de generalidad. Si n tiene un valor diferente, los datos pueden ampliarse añadiendo ceros. Después de la descompresión, los ceros adicionales son eliminados.) Considérese el conjunto de ocho valores $(1, 2, 3, 4, 5, 6, 7, 8,)$. En primer lugar, calculamos los cuatro promedios: $^{(1+2)}/_2 = {}^3/_2$, $^{(3+4)}/_2 = {}^7/_2$, $^{(5+6)}/_2 = {}^{11}/_2$, y $^{(7+8)}/_2 = {}^{15}/_2$. Es imposible reconstruir los ocho valores originales a partir de estos cuatro promedios, por lo que también calculamos las cuatro diferencias $^{(1-2)}/_2 = {}^{-1}/_2$, $^{(3-4)}/_2 = {}^{-1}/_2$, $^{(5-6)}/_2 = {}^{-1}/_2$, y $^{(7-8)}/_2 = {}^{-1}/_2$. Estas diferencias son llamadas *coeficientes de detalle*, y en este apartado se utilizan indistintamente los términos "diferencia" y "detalle". Podemos pensar en los promedios como una representación a baja resolución de la imagen original, y en los detalles como los datos necesarios para reconstruir la imagen original de esta tosca resolución. Si los píxeles de la imagen están correlacionados, la representación a baja resolución se parecerá a los píxeles originales, mientras que los detalles serán pequeños. Ésto explica por qué la compresión wavelet de Haar de imágenes usa promedios y detalles.

> Los periodos prolongados y lúgubres de domingo por la tarde en una ciudad universitaria podrían ser mitigados por la asistencia a las fiestas de té de Sillery, a las que nadie podría llegar después de las tres y media. La acción de alguna ley de los promedios, siempre regula el número de asistentes a estas reuniones a entre cuatro y ocho personas, en su mayoría estudiantes universitarios, aunque en ocasiones, un profesor desconocido.
>
> –Anthony Powell, *Una cuestión de Crianza*

Es fácil ver que la matriz $({}^3/_2, {}^7/_2, {}^{11}/_2, {}^{15}/_2, {}^{-1}/_2, {}^{-1}/_2, {}^{-1}/_2, {}^{-1}/_2)$, formada por los cuatro promedios y las cuatro diferencias, se puede utilizar para reconstruir los ocho valores originales. Esta matriz tiene ocho valores, pero sus últimas cuatro componentes —las diferencias— tienden a ser números pequeños, lo que ayuda en la compresión. Animados por ésto, repetimos el proceso en los cuatro promedios, los grandes componentes de nuestra matriz. Ellos se transforman en dos promedios y dos diferencias, produciendo $({}^{10}/_4, {}^{26}/_4, {}^{-4}/_4, {}^{-4}/_4, {}^{-1}/_2, {}^{-1}/_2, {}^{-1}/_2, {}^{-1}/_2)$. La siguiente, y última, iteración de este proceso transforma los dos primeros componentes de la nueva matriz en un promedio (la media de los ocho componentes de la matriz original) y una diferencia $({}^{36}/_8, {}^{-16}/_8, {}^{-4}/_4, {}^{-4}/_4, {}^{-1}/_2, {}^{-1}/_2, {}^{-1}/_2, {}^{-1}/_2)$. La última matriz es la *transformada wavelet de Haar* de los ítems de datos originales.

Debido a las diferencias, la transformada wavelet tiende a tener números más pequeños que los valores de los píxeles originales, por lo que es más fácil de comprimir utilizando RLE, quizás en combinación con la codificación mover–al–frente y Huffman. La compresión con pérdida puede conseguirse si algunas de las diferencias más pequeñas son cuantificadas o incluso completamente eliminadas (cambias a cero).

Antes de continuar, es interesante (y también útil) calcular la *complejidad* de esta transformación, i.e., el número de operaciones aritméticas necesarias en función del tamaño de los datos. En nuestro ejemplo teníamos $8+4+2 = 14$ operaciones (sumas y restas), un número que también puede expresarse como $14 = 2(8-1)$. En el caso general, supongamos que comenzamos con $N = 2^n$ ítems de datos. En la primera iteración necesitamos 2^n operaciones, en la segunda necesitamos 2^{n-1} operaciones, y así sucesivamente, hasta la última iteración, donde se necesitan $2^{n-(n-1)} = 2^1$ operaciones. Por

```
procedure NWTcalc(a:array of real, n:int);
  comment n es el tamaño del array (una potencia de 2)
  a:=a/√n comment divide el array entero
  j:=n;
  while j≥2 do
    NWTstep(a, j);
    j:=j/2;
  endwhile;
end;

procedure NWTstep(a:array of real, j:int);
  for i=1 to j/2 do
    b[i]:=(a[2i-1]+a[2i])/√2;
    b[j/2+i]:=(a[2i-1]-a[2i])/√2;
  endfor;
  a:=b; comment mueve el array completo
end;
```

Figura 5.12: Cálculo de la transformada wavelet normalizada.

consiguiente, el número total de operaciones es:

$$\sum_{i=1}^{n} 2^i = \sum_{i=0}^{n} 2^i - 1 = \frac{1-2^{n+1}}{1-2} - 1 = 2^{n+1} - 2 = 2\left(2^n - 1\right) = 2\left(N-1\right).$$

La transformada wavelet de Haar de N ítems de datos, por lo tanto, puede efectuarse con $2\left(N-1\right)$ operaciones, por lo que su complejidad es $\mathcal{O}\left(N\right)$, un resultado excelente.

Es útil asociar con cada iteración una cantidad llamada *resolución*, que se define como el número de promedios que quedan al final de la iteración. Las resoluciones después de cada una de las tres iteraciones anteriores son: $4\left(=2^2\right)$, $2\left(=2^1\right)$, y $1\left(=2^0\right)$. La Sección 5.6.3 muestra que cada componente de la transformada wavelet fue normalizado dividiéndolo por la raíz cuadrada de la resolución. (Ésta es la *transformada de Haar ortonormal*, también discutida en la Sección 4.5.3.) Por consiguiente, nuestro ejemplo de la transformada wavelet se convierte en:

$$\left(\frac{36/8}{\sqrt{2^0}}, \frac{-16/8}{\sqrt{2^0}}, \frac{-4/4}{\sqrt{2^1}}, \frac{-4/4}{\sqrt{2^1}}, \frac{-1/2}{\sqrt{2^2}}, \frac{-1/2}{\sqrt{2^2}}, \frac{-1/2}{\sqrt{2^2}}, \frac{-1/2}{\sqrt{2^2}}\right).$$

Si se utiliza la transformada wavelet normalizada, puede demostrarse formalmente que ignorar las diferencias menores es la mejor opción para la compresión wavelet con pérdidas, ya que ésto produce la pérdida de información más pequeña de la imagen.

Los dos procedimientos[1] de la Figura 5.12 ilustran cómo puede calcularse la transformada wavelet normalizada de una matriz de n componentes (donde n es una potencia de 2). La reconstrucción de la matriz original a partir de la transformada wavelet normalizada se ilustra mediante el par de procedimientos de la Figura 5.13.

Estos procedimientos parecen al principio ser diferentes de las medias y las diferencias discutidas anteriormente. No calculan promedios, porque dividen por $\sqrt{2}$ en lugar de por 2; el primer procedimiento se inicia dividiendo toda la matriz por $\sqrt{n}$, y el segundo termina revirtiendo el proceso. El resultado final, sin embargo, es el mismo que el mostrado anteriormente. Comenzando con el arreglo

[1] Lo que hay tras cada `comment` es un comentario.

```
procedure NWTreconst(a:array of real, n:int);
  j:=2;
  while j≤n do
    NWTRstep(a, j);
    j:=2j;
  endwhile
  a:=a√n; comment multiplicar el array completo
end;

procedure NWTRstep(a:array of real, j:int);
  for i=1 to j/2 do
    b[2i-1]:=(a[i]+a[j/2+i])/√2;
    b[2i]:=(a[i]-a[j/2+i])/√2;
  endfor;
  a:=b; comment mover el array completo
end;
```

Figura 5.13: Restauración desde una transformación wavelet normalizada.

$(1, 2, 3, 4, 5, 6, 7, 8)$, las tres iteraciones del procedimiento `NWTcalc` producen como resutado:

$$\left(\frac{3}{\sqrt{2^4}}, \frac{7}{\sqrt{2^4}}, \frac{11}{\sqrt{2^4}}, \frac{15}{\sqrt{2^4}}, \frac{-1}{\sqrt{2^4}}, \frac{-1}{\sqrt{2^4}}, \frac{-1}{\sqrt{2^4}}, \frac{-1}{\sqrt{2^4}}\right),$$
$$\left(\frac{10}{\sqrt{2^5}}, \frac{26}{\sqrt{2^5}}, \frac{-4}{\sqrt{2^5}}, \frac{-4}{\sqrt{2^5}}, \frac{-1}{\sqrt{2^4}}, \frac{-1}{\sqrt{2^4}}, \frac{-1}{\sqrt{2^4}}, \frac{-1}{\sqrt{2^4}}\right),$$
$$\left(\frac{36}{\sqrt{2^6}}, \frac{-16}{\sqrt{2^6}}, \frac{-4}{\sqrt{2^5}}, \frac{-4}{\sqrt{2^5}}, \frac{-1}{\sqrt{2^4}}, \frac{-1}{\sqrt{2^4}}, \frac{-1}{\sqrt{2^4}}, \frac{-1}{\sqrt{2^4}}\right),$$
$$\left(\frac{36/8}{\sqrt{2^0}}, \frac{-16/8}{\sqrt{2^0}}, \frac{-4/4}{\sqrt{2^1}}, \frac{-4/4}{\sqrt{2^1}}, \frac{-1/2}{\sqrt{2^2}}, \frac{-1/2}{\sqrt{2^2}}, \frac{-1/2}{\sqrt{2^2}}, \frac{-1/2}{\sqrt{2^2}}\right).$$

We spake no word, Tho' each I ween did hear the other's soul.
Not a wavelet stirred,
And yet we heard
The loneliest music of the weariest waves
That ever roll.[a]

—Abram J. Ryan, *Poems*

[a] No pronunciamos ninguna palabra, aunque ambos nos oíamos el alma; Ninguna pequeña ola se agitó; Y sin embargo, escuchamos; La solitaria música de las más debiles olas; Que siempre vuelven.

5.6.1. La aplicación de la transformada de Haar

Una vez aprendido el concepto de transformada wavelet, es fácil generalizarlo a una imagen bidimensional completa. Esto se puede realizar de varias maneras que se discuten en la Sección 5.10. Aquí se muestran dos enfoques, llamados *descomposición estándar* y *descomposición en pirámide*.

12	12	12	12	14	12	12	12
12	12	12	12	14	12	12	12
12	12	12	12	14	12	12	12
12	12	12	12	14	12	12	12
12	12	12	12	14	12	12	12
16	16	16	16	14	16	16	16
12	12	12	12	14	12	12	12
12	12	12	12	14	12	12	12

(a)

12	12	13	12	0	0	2	0
12	12	13	12	0	0	2	0
12	12	13	12	0	0	2	0
12	12	13	12	0	0	2	0
12	12	13	12	0	0	2	0
16	16	15	16	0	0	$\underline{2}$	0
12	12	13	12	0	0	2	0
12	12	13	12	0	0	2	0

(b)

12	12	13	12	0	0	2	0
12	12	13	12	0	0	2	0
14	14	14	14	0	0	0	0
12	12	13	12	0	0	2	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
$\underline{4}$	$\underline{4}$	$\underline{2}$	$\underline{4}$	0	0	4	0
0	0	0	0	0	0	0	0

(c)

Figura 5.14: Una imagen de 8×8 y su descomposición en subbandas.

El primero (Figura 5.15) comienza calculando la transformada wavelet de cada fila de la imagen. Ésto produce una imagen transformada, donde la primera columna contiene los promedios y todas las demás columnas contienen las diferencias. El algoritmo estándar calcula después la transformada wavelet de cada columna. Ésto resulta en un valor medio en la esquina superior izquierda, con el resto de la línea superior conteniendo los promedios de las diferencias, y con todos los demás valores de píxel transformados en diferencias.

El último método calcula la transformada wavelet de la imagen alternando entre filas y columnas. El primer paso es el cálculo de los promedios y las diferencias para todas las filas (una sola iteración, no toda la transformada wavelet). Ésto crea promedios en la mitad izquierda de la imagen y diferencias en la mitad derecha. El segundo paso es calcular promedios y diferencias (sólo una iteración) para todas las columnas, lo que genera promedios en el cuadrante superior izquierdo de la imagen y diferencias en otro lugar. Los pasos 3 y 4 operan en las filas y columnas de ese cuadrante, lo que produce promedios concentrados en el subcuadrante superior izquierdo. Pares de pasos son ejecutados repetidamente en subcuadrados cada vez más pequeños, hasta que sólo queda un promedio, en la esquina superior izquierda de la imagen, y todos los otros valores de píxel se han reducido a diferencias. Este proceso se resume en la Figura 5.16.

Las transformaciones que se describen en la Sección 4.4 son ortogonales. Ellas transforman los píxeles originales en unos pocos números grandes y muchos números pequeños. En contraste, la transformada wavelet , así como la transformada de Haar, son *transformaciones de subbanda*. Ellas particionan la imagen en regiones tales que cada una contiene un gran número (promedios en el caso de la transformada de Haar) y las demás regiones contienen números pequeños (diferencias). Sin embargo, estas regiones, que se llaman subbandas, son más que conjuntos de números grandes y pequeños. Son el reflejo de diferentes artefactos geométricos de la imagen. Para ilustrar esta importante característica, examinamos una pequeña imagen, en su mayor parte uniforme con una línea vertical y una línea horizontal. La Figura 5.14a muestra una imagen de 8×8 con valores de píxel de 12, excepto por una línea vertical con valores de píxel de 14 y una línea horizontal con valores de píxel de 16.

La Figura 5.14b muestra los resultados de la aplicación de la transformada de Haar una vez a las filas de la imagen. La mitad derecha de esta figura (las diferencias) es mayormente de ceros, lo que refleja la naturaleza uniforme de la imagen. Sin embargo, las trazas de la línea vertical pueden verse fácilmente (la notación $\underline{2}$ indica una diferencia negativa). La Figura 5.14c muestra los resultados de aplicar la transformada de Haar una vez a las columnas de la Figura 5.14b. La subbanda superior derecha ahora presenta trazas de la línea vertical, mientras que la subbanda inferior izquierda muestra trazas de la línea horizontal. Estas subbandas se han denotado HL y LH, respectivamente (Figuras 5.16 y 5.55, aunque hay inconsistencia en el uso de esta notación por varios autores). La subbanda inferior derecha, denotada HH, refleja los artefactos de la diagonal de la imagen (de los que carece nuestra imagen de ejemplo). Lo más interesante es la subbanda superior izquierda, denotada LL, que está formada en su totalidad por promedios. Esta subbanda es una versión de un cuarto de la imagen

```
procedure StdCalc(a:array of real, n:int);
  comment el tamaño del array es nxn (n = potencia de 2)
  for r=1 to n do NWTcalc(fila r de a, n);
  endfor;
  for c=n to 1 do comment bucle hacia atrás
    NWTcalc(col c de a, n);
  endfor;
end;
procedure StdReconst(a:array of real, n:int);
  for c=n to 1 do comment bucle hacia atrás
    NWTreconst(col c de a, n);
  endfor;
  for r=1 to n do
    NWTreconst(fila r de a, n);
  endfor;
end;
```

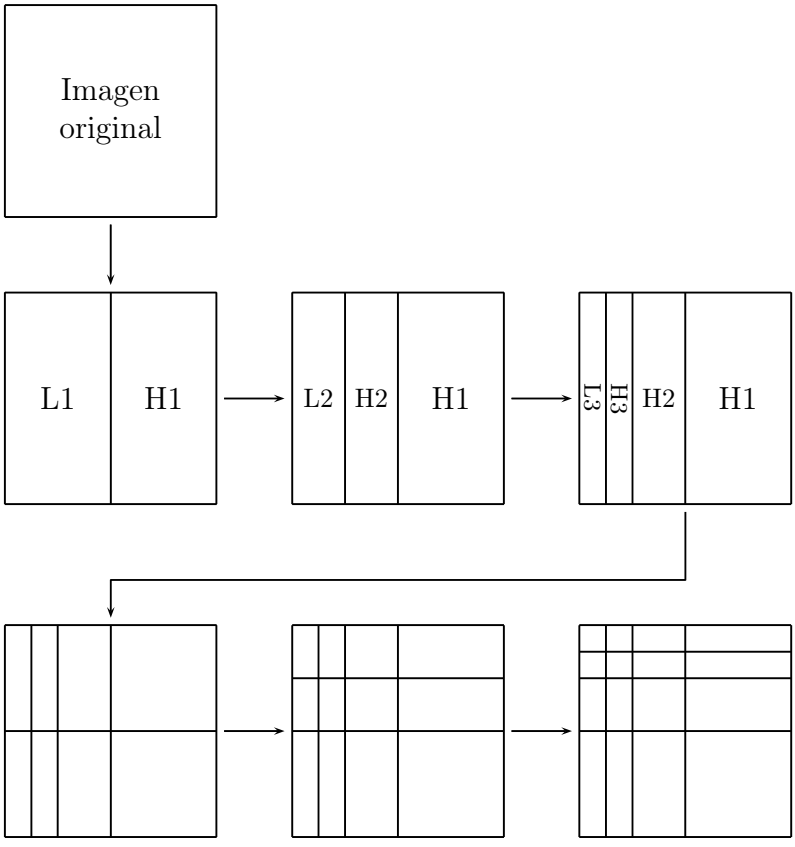


Figura 5.15: Transformada wavelet estándar de imágenes y descomposición.

```
procedure NStdCalc(a:array of real, n:int);
  a:=a/√n comment dividir el array entero
  j:=n;
  while j≥2 do
    for r=1 to j do NWTstep(fila r de a, j);
    endfor;
    for c=j to 1 do comment bucle hacia atrás
      NWTstep(col c de a, j);
    endfor;
    j:=j/2;
  endwhile;
end;
procedure NStdReconst(a:array of real, n:int);
  j:=2;
  while j≤n do
    for c=j to 1 do comment bucle hacia atrás
      NWTRstep(col c de a, j);
    endfor;
    for r=1 to j do
      NWTRstep(fila r de a, j);
    endfor;
    j:=2j;
  endwhile
  a:=a√n; comment multiplicar el array entero
end;
```

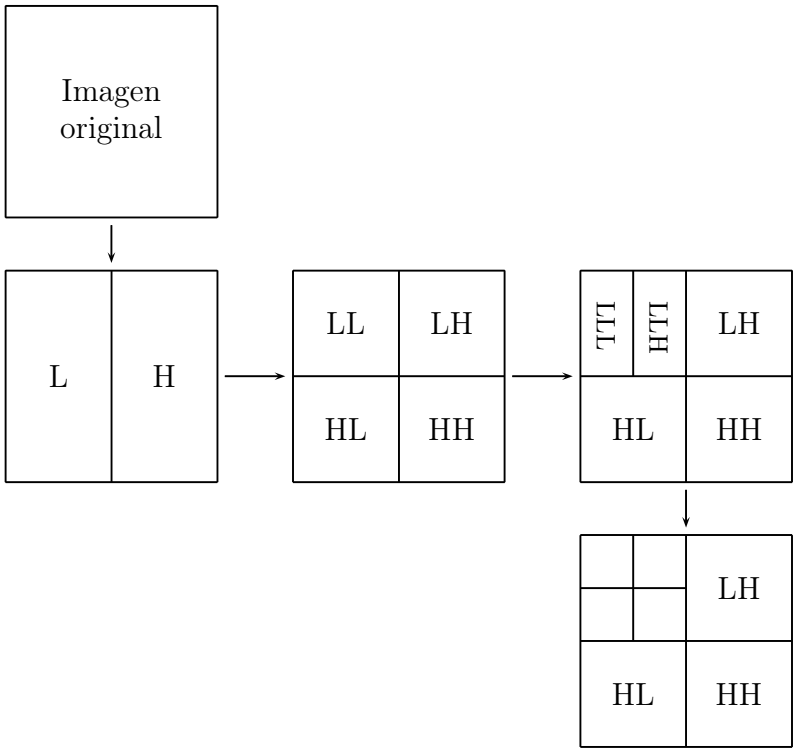


Figura 5.16: Transformada wavelet de imágenes de pirámide.

completa, que contiene trazas tanto de las líneas verticales como de las horizontales.

◇ **Ejercicio 5.4 (sol. en pág. 1095):** Constrúyase un diagrama similar al de la Figura 5.14 para mostrar cómo la subbanda HH refleja los artefactos de la diagonal de la imagen.

(Artefacto: Una características no presentes de forma natural, introducida durante la preparación o la investigación.)

La Figura 5.55 muestra cuatro niveles de subbandas, donde el nivel 1 contiene las características detalladas de la imagen (también referidos como coeficientes wavelet de alta frecuencia o de resolución fina) y el nivel superior, nivel 4, contiene las características más toscas de la imagen (los coeficientes de baja frecuencia o coeficientes de resolución gruesa). Es evidente que los niveles más bajos pueden ser cuantificados toscamente sin mucha pérdida de información importante de la imagen, mientras que los niveles más altos deben ser cuantificados finamente. La estructura de subbanda es la base de todos los métodos de compresión de imágenes que utilizan la transformada wavelet.

La Figura 5.17 muestra los resultados típicos de la transformada wavelet en pirámide. La imagen original se muestra en la Figura 5.17a, y la Figura 5.17c es una descomposición en pirámide general. Con el fin de ilustrar cómo funciona la transformada en pirámide, esta imagen se compone sólo de líneas horizontales, verticales e inclinadas. Los cuatro cuadrantes de la Figura 5.17b muestran versiones menores de la imagen. La subbanda superior izquierda, que contiene los promedios, es similar a la imagen completa, mientras que cada una de las otras tres subbandas muestra los detalles de la imagen. Debido a la forma en que se construye la transformación en pirámide, la subbanda superior derecha contiene los detalles verticales, la subbanda inferior izquierda contiene los detalles horizontales, y la parte inferior derecha contiene los detalles de las líneas inclinadas. La Figura 5.17c muestra los resultados de aplicar repetidamente esta transformación. La imagen se convierte en subbandas de detalles verticales, horizontales, y diagonales, mientras que el subcuadrado superior izquierdo, que contiene los promedios, se reduce hasta un único píxel.

La Sección 4.4 analiza las transformadas de imágenes. Debe mencionarse que hay dos tipos principales de transformadas: *ortogonales*, y de *subbanda*. Una transformada ortogonal lineal se realiza calculando el *producto interno* de los datos (valores de píxeles o muestras de sonido) con un conjunto de *funciones base*. El resultado es un conjunto de coeficientes de transformación que más tarde pueden ser cuantificados o comprimidos mediante RLE, la codificación de Huffman, u otros métodos. Varios ejemplos de transformaciones ortogonales importantes, tales como la DCT, la WHT, y la KLT, se describen en detalle en la Sección 4.4. La transformada de Fourier también pertenece a este categoría. Ésta se discute en la Sección 5.1.

El otro tipo principal de transformación es la *transformación de subbanda*. Ésta se lleva a cabo calculando una convolución de los datos (Sección 5.7) con un conjunto de filtros de *paso de banda*. Cada subbanda resultante codifica una porción particular del contenido en frecuencia de los datos.

Como recordatorio, el producto interno discreto de los dos vectores f_i y g_i se define por:

$$\langle f, g \rangle = \sum_i f_i g_i.$$

La convolución discreta h se define mediante la Ecuación (5.5):

$$h_i = f \star g = \sum_j f_j g_{i-j}. \tag{5.5}$$

(Cada elemento h_i de la convolución discreta h es la suma de los productos. Y depende de i en la especial forma mostrada.)

Cualquiera de los métodos, estándar o uniforme, produce una transformada, aunque todavía no se haya comprimido, de la imagen que tiene un promedio en la esquina superior izquierda y pequeños números, diferencias, o promedios de las diferencias en cualquier otra parte. Estos números se pueden

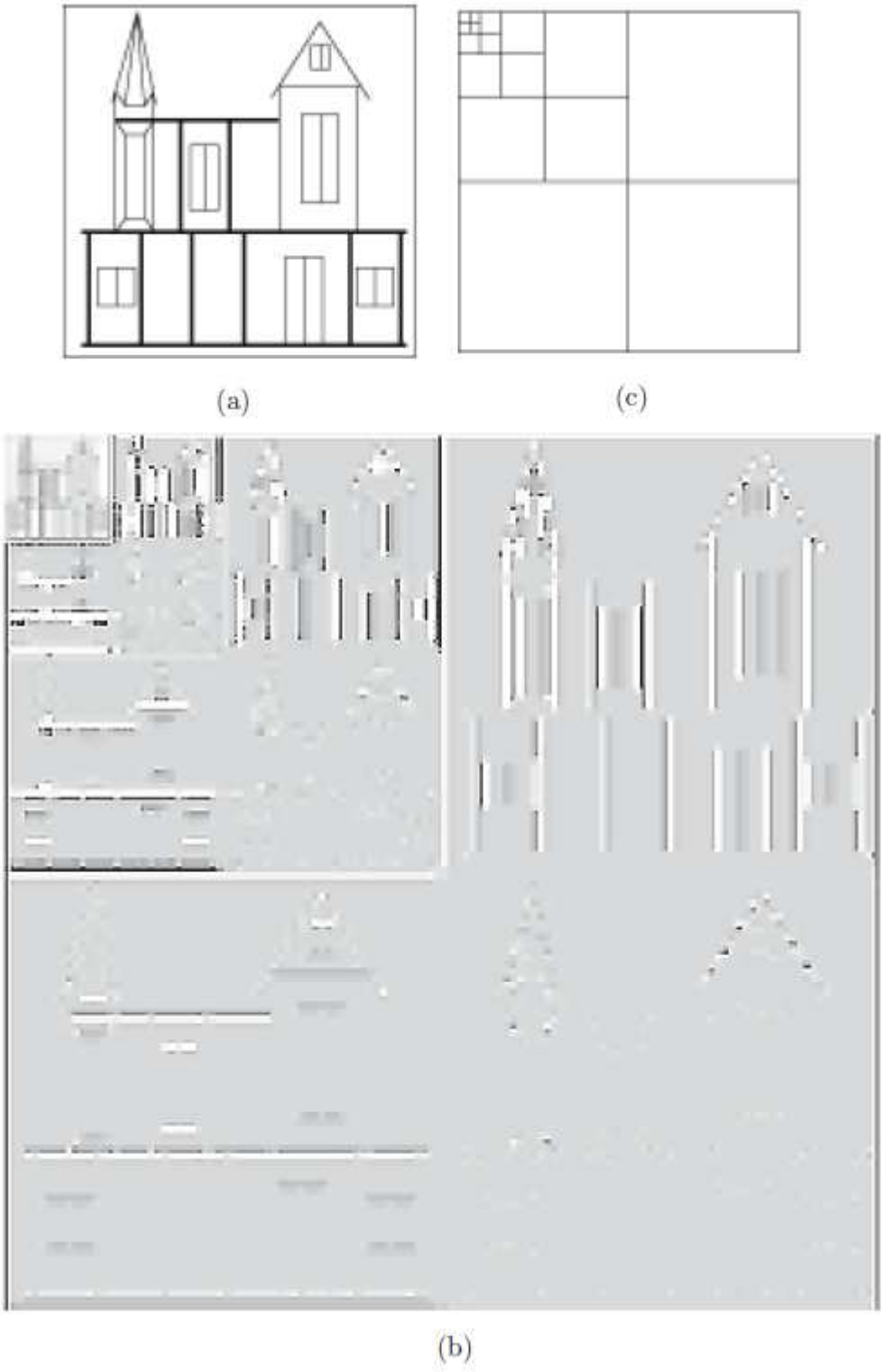


Figura 5.17: Ejemplo de transformada wavelet en pirámide de una imagen.

comprimir utilizando una combinación de métodos, tales como RLE, mover–al–frente, y la codificación Huffman. Si la compresión con pérdida es aceptable, algunas de las diferencias más pequeñas pueden ser cuantificadas o incluso establecerse a cero, lo cual crea rachas de ceros, haciendo el uso de RLE aún más atractivo.

Whiter foam than thine, O wave,
Wavelet never wore,
Stainless wave; and now you lave
The far and stormless shore —
Ever — ever — evermore![a]

—Abram J. Ryan, *Poems*

[a]Más blanca espuma que la tuya, ¡oh ola!, La olita nunca vestía, Ola inoxidable; y ahora tú moldeas el paisaje de; La remota y agitada orilla; una — y otra vez — ¡siempre!

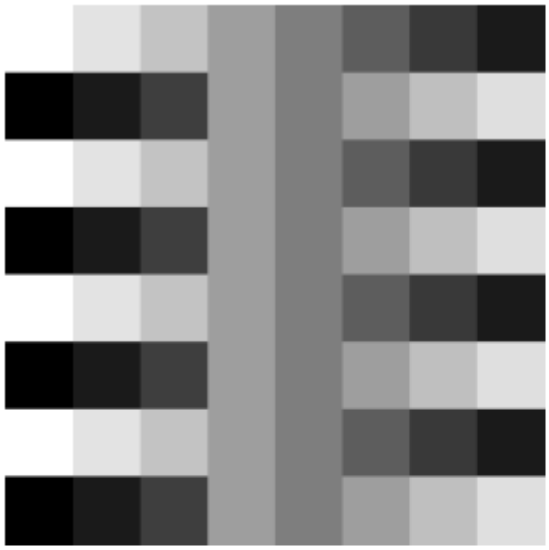
Imágenes en color: Hasta ahora hemos asumido que cada píxel es un único número (i.e., tenemos una imagen con sólo un componente, en el que todos los píxeles son tonos de un mismo color, normalmente gris). Cualquier método de compresión de imágenes de un único componente puede ser extendido a imágenes en color (tres componentes) separando los tres componentes, y después transformando y comprimiendo cada uno individualmente. Si el método de compresión es con pérdidas, tiene sentido convertir los tres componentes de la imagen desde su representación de color original, que es normalmente RGB, hasta la representación de color YIQ. El componente Y de esta representación se llama *luminancia*, y las componentes I y Q (la crominancia) son responsables de la información del color [Salomon 99]. La ventaja de esta representación del color es que el ojo humano es más sensible a Y y menos sensible a Q. Un método con pérdidas debe, por lo tanto, dejar el componente Y solo, y borrar algunos datos de las I, y más datos de los componentes Q, produciendo una buena compresión, y ocasionando unas pérdidas a las que el ojo es no que sensible.

Es interesante observar que la transmisión de televisión a color de EE.UU. también aprovecha la representación YIQ. Las señales se emiten con anchos de banda de 4 MHz para Y, 1,5 MHz para I, y sólo 0,6 MHz para Q.

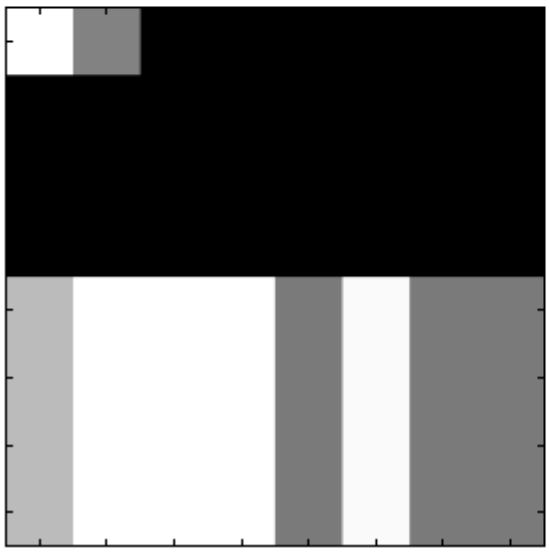
5.6.2. Propiedades de la transformada de Haar

Los ejemplos en esta sección ilustran algunas propiedades de la transformada de Haar, y de la transformada wavelet discreta en general. La Figura 5.18 muestra una imagen de 8×8 altamente correlacionada y su transformada wavelet de Haar. Se muestran tanto la escala de grises como los valores numéricos de los píxeles y los *coeficientes de la transformada*. Debido a que la imagen original está tan correlacionada, los coeficientes wavelet son pequeños y hay muchos ceros.

◇ **Ejercicio 5.5 (sol. en pág. 1097):** Un vistazo a la Figura 5.18 sugiere que la última sentencia es incorrecta. Los coeficientes de la transformada wavelet mostrados en la figura son muy grandes en comparación con los valores de píxel de la imagen original. De hecho, sabemos que los coeficientes de la parte superior izquierda de la transformada de Haar deben ser el promedio de todos los píxeles de la imagen. Puesto que los píxeles de nuestra imagen tienen valores que están (más o menos) distribuidos uniformemente en el intervalo $[0, 255]$, este promedio debe estar alrededor de 128, sin embargo, el coeficiente de la parte superior izquierda de transformada es 1051. ¡Explíquese ésto!



255	224	192	159	127	95	63	32
0	32	64	159	127	159	191	223
255	224	192	159	127	95	63	32
0	32	64	159	127	159	191	223
255	224	192	159	127	95	63	32
0	32	64	159	127	159	191	223
255	224	192	159	127	95	63	32
0	32	64	159	127	159	191	223



1051	34.0	-44.5	-0.7	-1.0	-62	0	-1.0
0	0.0	0.0	0.0	0.0	0	0	0.0
0	0.0	0.0	0.0	0.0	0	0	0.0
0	0.0	0.0	0.0	0.0	0	0	0.0
48	239.5	112.8	90.2	31.5	64	32	31.5
48	239.5	112.8	90.2	31.5	64	32	31.5
48	239.5	112.8	90.2	31.5	64	32	31.5
48	239.5	112.8	90.2	31.5	64	32	31.5

Figura 5.18: La imagen de 8×8 reconstruida en la Figura 5.20 y transformada de Haar.

En una transformada wavelet discreta, la mayoría de los coeficientes wavelet son detalles (o diferencias). Los detalles de los niveles inferiores representan los detalles finos de la imagen. A medida que nos movemos a un nivel de subbanda superior, encontramos detalles que corresponden a características más toscas de la imagen. La Figura 5.19a ilustra este concepto. Muestra una imagen que es suave en la izquierda y tiene "actividad" (i.e., los píxeles adyacentes que tienden a ser diferentes) a la derecha. La parte (b) muestra la transformada wavelet de la imagen. Los niveles bajos (correspondientes a los detalles finos) han transformado los coeficientes de la derecha, ya que aquí es donde se encuentra la actividad de la imagen. Los niveles altos (detalles gruesos) parecen similares, pero también tienen coeficientes en el lado izquierdo, porque la imagen no aparece completamente en blanco a la izquierda.

La transformada de Haar es la transformada wavelet más simple, pero incluso este sencillo método ilustra el poder de la transformada wavelet. Resulta que los niveles bajos de la transformada wavelet

Figura 5.19: (a) Una imagen de 128×128 con actividad en la derecha. (b) Su transformada.

discreta contienen las características sin importancia de la imagen, por lo que la cuantificación o descarte de estos coeficientes puede conducir a una compresión con pérdida que es a la vez eficiente y de de alta calidad. A menudo, la imagen puede ser reconstruida a partir de muy pocos coeficientes de la transformada sin ninguna pérdida de calidad notable. La Figura 5.20a−c muestra tres reconstrucciones de la sencilla imagen de 8×8 de la Figura 5.18. Éstas se obtuvieron a partir de tan sólo 32, 13 y 5 coeficientes wavelet, respectivamente.

La Figura 5.21 es un ejemplo similar. Muestra una imagen binivel completamente reconstruida a partir de sólo el 4 % de los coeficientes de su transformada (653 coeficientes de 128×128).

La experimentación es la clave para entender estos conceptos. El software matemático adecuado hace que sea fácil introducir imágenes y experimentar con diversas características de la transformada wavelet discreta. Con el fin de ayudar al lector interesado, la Figura 5.22 lista un programa en Matlab que introduce una imagen, calcula su transformada wavelet de Haar, descarta un porcentaje determinado de los coeficientes más pequeños de la transformada, y luego calcula la transformada inversa para reconstruir la imagen.

La compresión con pérdida de imágenes wavelet implica el descarte de los coeficientes, por lo que se define el concepto de *proporción de raleza* (*sparseness ratio*) para medir la cantidad de coeficientes descartados. La raleza o dispersión se define como el número de coeficientes de la wavelet distintos de cero dividido por el número de coeficientes que quedan después desechar algunos. Cuanto mayor sea la proporción de raleza, menos coeficientes permanecen. Las mayores razones de raleza conducen a una mejor compresión, pero puede generar imágenes pobremente reconstruidas. La proporción de raleza está lejanamente relacionada con el factor de compresión, una medida de la compresión que se define en la introducción.

La línea "`filename = 'lena128'; dim = 128;`" contiene el nombre del archivo de imagen y la dimensión de la imagen. Los archivos de imagen utilizados por el autor estaban sin formato y contenían únicamente los valores en escala de grises, cada uno en un solo byte. No incluían ninguna cabecera, e incluso ninguna resolución de la imagen (número de filas y columnas) en el archivo. Sin embargo, Matlab puede leer otros tipos de archivos. Se supone que la imagen es cuadrada, y el parámetro "`dim`" debe ser una potencia de 2. La asignación "`thresh=`" especifica el porcentaje de coeficientes de la transformada que se desean eliminar. Esto proporciona una forma fácil de experimentar con la compresión con pérdida de imágenes wavelet.

El archivo "`harmatt.m`" contiene dos funciones que calculan los coeficientes wavelet de Haar en

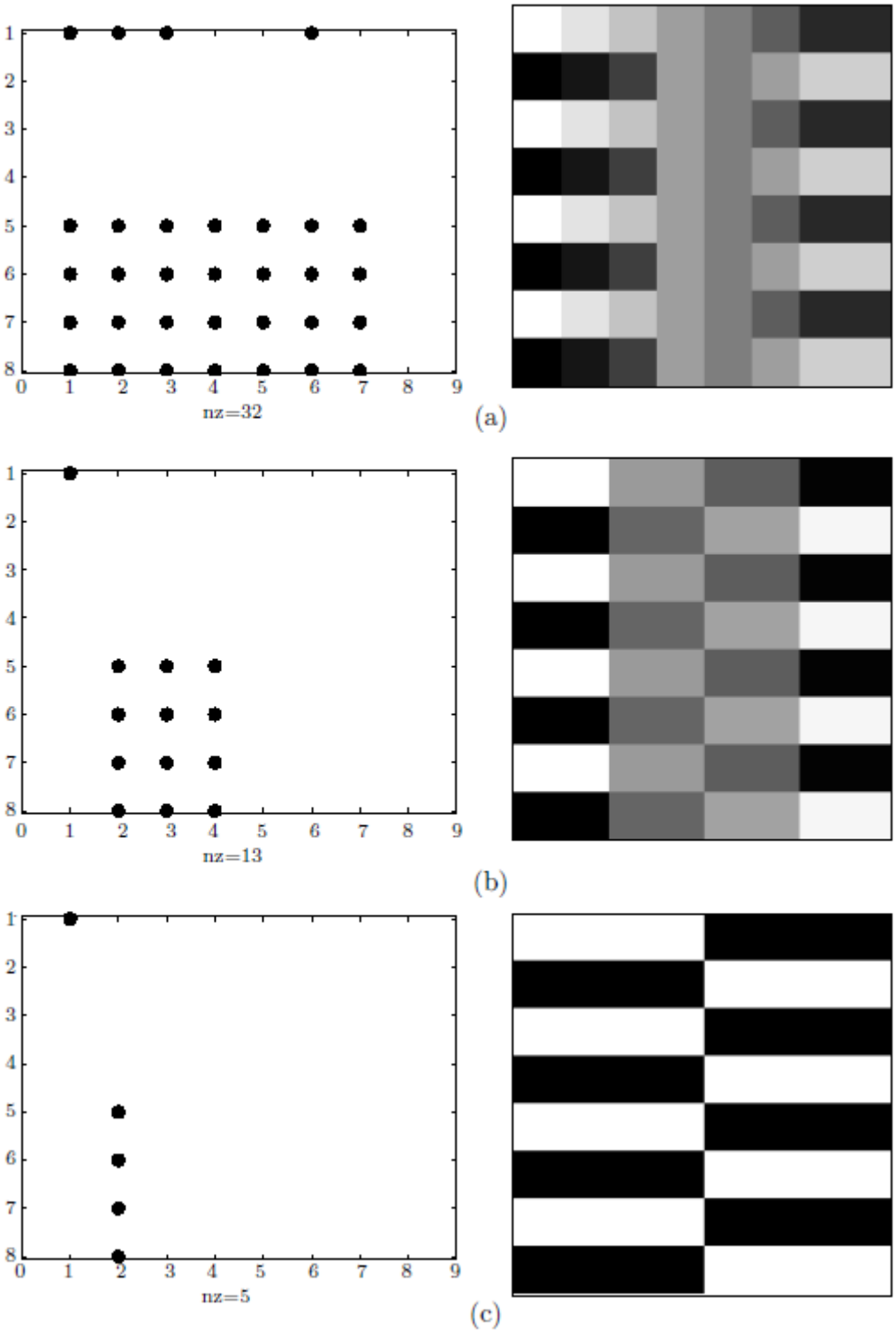


Figura 5.20: Tres reconstrucciones con pérdidas de una imagen de 8×8.

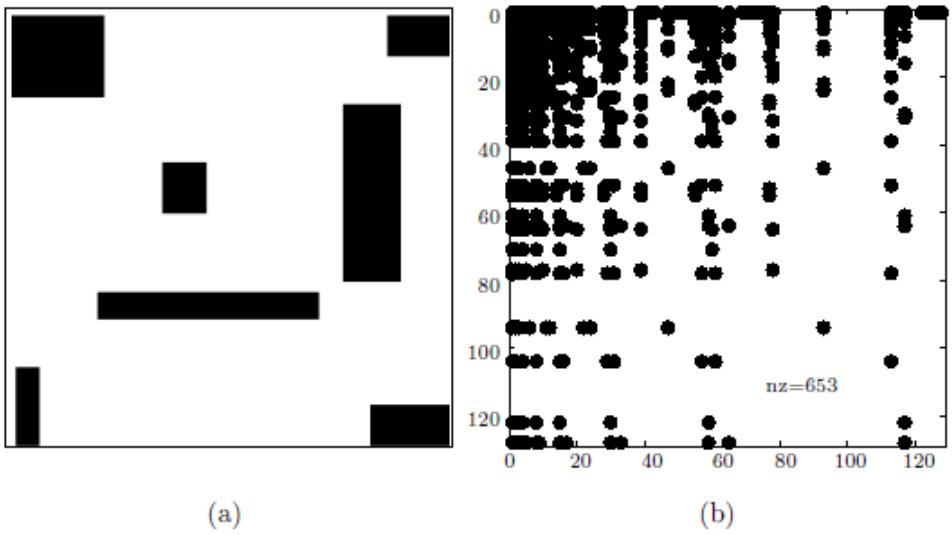


Figura 5.21: Reconstrucción de una imagen simple de 128×128 a partir del 4 % de sus coeficientes.

forma de matriz (Sección 5.6.3).

(Nota técnica: Un archivo de Matlab `m` puede incluir comandos o una función, pero no ambos. Puede, sin embargo, contener más de una función, siempre que sólo la función de la parte superior se invoque desde fuera del archivo. Todas las demás funciones deben llamarse desde dentro del archivo. En nuestro caso, la función `harmatt(dim)` llama a la función `individ(n)`.)

◇ **Ejercicio 5.6 (sol. en pág. 1097):** Utilícese el código de la Figura 5.22 (o un código similar) para calcular la transformada de Haar de la imagen de "Lena" (Figura 4.53) y reconstrúyase tres veces descartando cada vez más coeficientes de detalle.

5.6.3. Un enfoque con matrices

El principio de la transformada de Haar es calcular los promedios y las diferencias. Resulta que ésto se puede realizar por medio de la multiplicación de matrices ([Mulcahy 96] y [Mulcahy 97]). A modo de ejemplo, nos fijamos en la fila superior de la sencilla imagen de la Figura 5.18. Cualquiera con un poco de experiencia con matrices puede construir una matriz que cuando se multiplica por este vector crea un vector con cuatro promedios y cuatro diferencias. La matriz A_1 de la Ecuación (5.6) hace ésto y, cuando se multiplica por la fila superior de píxeles de la Figura 5.18, genera (239,5, 175,5, 111,0, 47,5, 15,5, 16,5, 16,0, 15,5). Similarmente, las matrices A_2 y A_3 realizan los pasos segundo y tercero de la transformada, respectivamente. Los resultados se muestran en la Ecuación (5.7):

$$A_1 = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & -\frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & -\frac{1}{2} \end{pmatrix}, \quad A_1 \begin{pmatrix} 255 \\ 224 \\ 192 \\ 159 \\ 127 \\ 95 \\ 63 \\ 32 \end{pmatrix} = \begin{pmatrix} 239{,}5 \\ 175{,}5 \\ 111{,}0 \\ 47{,}5 \\ 15{,}5 \\ 16{,}5 \\ 16{,}0 \\ 15{,}5 \end{pmatrix}, \tag{5.6}$$

```
clear;% programa principal
filename='lena128'; dim=128;
fid=fopen(filename,'r');
if fid==-1 disp('archivo no encontrado')
else img=fread(fid,[dim,dim])'; fclose(fid);
end
thresh=0.0;% tanto por ciento de coeficientes de la transformada borrados
figure(1), imagesc(img), colormap(gray), axis off, axis square
w=harmatt(dim);% calcular la matriz dim x dim de la transformada de Haar
timg=w*img*w';% transformada de Haar directa
tsort=sort(abs(timg(:)));
tthresh=tsort(floor(max(thresh*dim*dim,1)));
cim=timg.*(abs(timg) > tthresh);
[i,j,s]=find(cim);
dimg=sparse(i,j,s,dim,dim);
% figure(2) muestra los coeficientes restantes de la transformada
% figure(2), spy(dimg), colormap(gray), axis square
figure(2), image(dimg), colormap(gray), axis square
cimg=full(w'*sparse(dimg)*w);% transformada de Haar inversa
density = nnz(dimg);
disp([num2str(100*thresh) '% de coeficientes más pequeños eliminados.'])
disp([num2str(density) ' coeficientes permanecen fuera de ' ...
  num2str(dim) 'x' num2str(dim) '.'])
figure(3), imagesc(cimg), colormap(gray), axis off, axis square


File harmatt.m with two functions


function x = harmatt(dim)
num=log2(dim);
p = sparse(eye(dim)); q = p;
i=1;
while i≤dim/2;
  q(1:2*i,1:2*i) = sparse(individ(2*i));
  p=p*q; i=2*i;
end
x=sparse(p);


function f=individ(n)
x=[1, 1]/sqrt(2);
y=[1,-1]/sqrt(2);
while min(size(x)) < n/2
  x=[x, zeros(min(size(x)),max(size(x)));...
    zeros(min(size(x)),max(size(x))), x];
end
while min(size(y)) < n/2
  y=[y, zeros(min(size(y)),max(size(y)));...
    zeros(min(size(y)),max(size(y))), y];
end
f=[x;y];
```

Figura 5.22: Código en Matlab para la transformada de Haar de una imagen.

$$A_2 = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 \\ \frac{1}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}, \quad A_3 = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix},$$

$$A_2 \begin{pmatrix} 239{,}5 \\ 175{,}5 \\ 111{,}0 \\ 47{,}5 \\ 15{,}5 \\ 16{,}5 \\ 16{,}0 \\ 15{,}5 \end{pmatrix} = \begin{pmatrix} 207{,}5 \\ 79{,}25 \\ 32{,}0 \\ 31{,}75 \\ 15{,}5 \\ 16{,}5 \\ 16{,}0 \\ 15{,}5 \end{pmatrix}, \quad A_3 \begin{pmatrix} 207{,}5 \\ 79{,}25 \\ 32{,}0 \\ 31{,}75 \\ 15{,}5 \\ 16{,}5 \\ 16{,}0 \\ 15{,}5 \end{pmatrix} = \begin{pmatrix} 143{,}375 \\ 64{,}125 \\ 32{,}0 \\ 31{,}75 \\ 15{,}5 \\ 16{,}5 \\ 16{,}0 \\ 15{,}5 \end{pmatrix}. \tag{5.7}$$

En lugar de calcular los promedios y las diferencias, lo único que tenemos que hacer es construir matrices A_1, A_2, y A_3, multiplicarlas para obtener $W = A_1 A_2 A_3$, aplicar W a todas las columnas de la imagen I multiplicando $W \cdot I$:

$$W \begin{pmatrix} 255 \\ 224 \\ 192 \\ 159 \\ 127 \\ 95 \\ 63 \\ 32 \end{pmatrix} = \begin{pmatrix} \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} \\ \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{-1}{8} & \frac{-1}{8} & \frac{-1}{8} & \frac{-1}{8} \\ \frac{1}{4} & \frac{1}{4} & \frac{-1}{4} & \frac{-1}{4} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{-1}{4} & \frac{-1}{4} \\ \frac{1}{2} & \frac{-1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{-1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{-1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{-1}{2} \end{pmatrix} \begin{pmatrix} 255 \\ 224 \\ 192 \\ 159 \\ 127 \\ 95 \\ 63 \\ 32 \end{pmatrix} = \begin{pmatrix} 143{,}375 \\ 64{,}125 \\ 32{,}0 \\ 31{,}75 \\ 15{,}5 \\ 16{,}5 \\ 16{,}0 \\ 15{,}5 \end{pmatrix}.$$

Ésto, por supuesto, es sólo la mitad del trabajo. Con el fin de calcular la transformada completa, todavía tenemos que aplicar W a las filas del producto $W \cdot I$, y lo efectuamos aplicándola a las columnas de la transpuesta $(W \cdot I)^T$, y a continuación transponiendo resultado. En consecuencia, la transformación completa es (véase la línea `timg=w*img*w'` en la Figura 5.22)

$$I_{\text{tr}} = \left(W \left(W \cdot I \right)^T \right)^T = W \cdot I \cdot W^T.$$

La transformada inversa se realiza mediante:

$$W^{-1} \left(W^{-1} \cdot I_{\text{tr}}^T \right)^T = W^{-1} \left(I_{\text{tr}} \cdot \left(W^{-1} \right)^T \right),$$

y aquí es donde la transformada de Haar normalizada (mencionada en la página 555) se vuelve importante. En lugar de calcular los promedios [cantidades de la forma $^{(d_i+d_{i+1})}/_2$] y las diferencias [cantidades de la forma $^{(d_i-d_{i+1})}/_2$], es mejor utilizar las cantidades $^{(d_i+d_{i+1})}/\sqrt{2}$ y $^{(d_i-d_{i+1})}/\sqrt{2}$. Estos resultados producen una matriz *ortonormal* W, y es sabido que la inversa de esta matriz es simplemente su transpuesta. Por consiguiente, podemos escribir la transformada inversa en la forma simple: $W^T \cdot I_{\text{tr}} \cdot W$ [véase la línea `cimg=full(w'*sparse(dimg)*w)` en la Figura 5.22].

Entre las transformadas directa e inversa, algunos coeficientes de la transformada pueden ser cuantificados o eliminados. Alternativamente, la matriz I_{tr} puede ser comprimida mediante la codificación run length y/o códigos de Huffman.

La función `individ(n)` de la Figura 5.22 comienza con una matriz de la transformada de Haar de 2×2 (observe que usa $\sqrt{2}$ en lugar de 2) y luego se utiliza para la construcción de tantas matrices A_i individuales como sea necesario. La función `harmatt(dim)` combina las matrices individuales para formar la matriz de Haar final para una imagen tenue de `dim` filas y `dim` columnas.

◇ **Ejercicio 5.7 (sol. en pág. 1097):** Efectúese el cálculo de $W \cdot I \cdot W^T$ para la imagen de 8×8 de la Figura .

5.7. Bancos de filtros

En esta sección se utiliza el enfoque matricial de la transformación de Haar para introducir al lector en la idea de *bancos de filtros*. Mostrramos cómo la transformada de Haar puede ser interpretada como un banco de dos filtros, uno paso bajo y otro paso alto. Explicamos los términos "filtro", "paso bajo", y "paso alto" y mostramos cómo la idea de bancos de filtros conduce de forma natural al concepto de *transformada de subbanda*. La transformada de Haar, por supuesto, es la transformada wavelet más sencilla, por lo que se utiliza aquí para ilustrar los nuevos conceptos. Sin embargo, su uso como un banco de filtros puede no ser muy eficiente. Las aplicaciones de filtros wavelet más prácticas emplean conjuntos de coeficientes de filtros más sofisticados, pero todos se basan en el concepto de filtros y bancos de filtros [Strang y Nguyen 96].

> And just like the wavelet that moans on the beach,
> And, sighing, sinks back to the sea,
> So my song—it just touches the rude shores of speech,
> And its music melts back into me.[a]
>
> —Abram J. Ryan, *Poems*

[a] Y así como la ola pequeña que gime en la playa; Y, suspirando, se hunde de nuevo en la mar; Así mi canción —toca las rudas costas del lenguaje; Y su música se funde de nuevo en mí. *Poemas* de Abram J. Ryan.

Un *filtro* es un operador lineal definido en términos de sus *coeficientes de filtro* $h(0)$, $h(1)$, $h(2)$,.... Puede ser aplicado a un vector de entrada x para producir un vector de salida y acorde con:

$$y(n) = \sum_k h(k)\, x(n-k) = h \star x,$$

donde el símbolo $\star$ indica una convolución. Observe que los límites de la suma anterior no se han indicado explícitamente. Ellos dependen de los tamaños de los vectores x y h. Puesto que nuestra variable independiente es el tiempo t, es conveniente asumir que las entradas (y, en consecuencia, también las salidas) llegan continuamente en los instantes $t = \ldots, -2, -1, 0, 1, 2, \ldots$. Por consiguiente, utilizamos la notación:

$$x = (\ldots, a, b, c, d, e, \ldots),$$

donde el valor central c es la entrada en el tiempo cero $[c = x(0)]$; los valores d y e son las entradas en los tiempos 1 y 2, respectivamente; y $b = x(-1)$ y $a = x(-2)$. En la práctica, la entradas son siempre finitas, por lo que el vector infinito x tendrá sólo un número finito de elementos distintos de cero.

Una visión más profunda del comportamiento de un filtro lineal puede obtenerse considerando la sencilla entrada $x = (\ldots, 0, 0, 1, 0, 0, \ldots)$. Esta entrada es cero en todo momento excepto en $t = 0$. Se conoce como *unidad de pulsos* o *impulso unitario*. Aunque los límites de la suma en la convolución

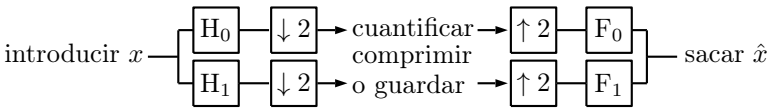


Figura 5.23: Un banco de filtros de dos canales.

han sido especificados, es fácil ver que para cualquier n sólo hay un término distinto de cero en la suma, de modo que $y(n) = h(n)\,x(0) = h(h)$. Decimos que la salida $y(n) = h(h)$ en el tiempo n es la *respuesta* en el tiempo n a la unidad de impulso $x(0) = 1$. Puesto que el número de coeficientes de filtro $h(i)$ es finito, éste es un filtro de *impulsos de respuesta finita* (FIR o *finite impulse response*).

La Figura 5.23 presenta la idea básica de un banco de filtros. Muestra un *banco de análisis* consistente en dos filtros, un filtro paso bajo H_0 y un filtro paso alto H_1. El filtro paso bajo emplea la convolución para eliminar las frecuencias altas de la señal de entrada x y deja pasar las frecuencias bajas a través de él. El filtro de paso alto hace lo contrario. Juntos, separan la entrada en *bandas de frecuencia*.

La entrada x puede ser una señal unidimensional (un vector de números reales, que es lo que suponemos en esta sección) o una señal bidimensional, una imagen. Los elementos $x(n)$ de x son introducidos en los filtros de uno en uno, y cada filtro calcula y produce un número $y(n)$ en respuesta a $x(n)$. El número de respuestas es, por lo tanto, el doble del número de entradas (porque tenemos dos filtros); un mal resultado, ya que estamos interesados en la compresión de datos. Para corregir esta situación, cada filtro es seguido por un proceso de *downsampling* (reducción de las muestras) donde se descartan las salidas impares. Esta operación se denomina también *decimación* y está representado por las cajas marcadas "$\downarrow 2$". Después de la decimación, el número de salidas de los dos filtros juntos es igual al número de entradas.

Observe que el banco de filtros descrito aquí, seguido por la decimación, realiza exactamente los mismos cálculos que la matriz $W = A_1A_2A_3$ de la Sección 5.6.3. Los bancos de filtros son simplemente una forma más general de observar la transformada de Haar (o, en general, la transformada discreta wavelet). Vemos esta transformación como una operación de filtrado, seguida de decimación, y después tratamos de encontrar mejores filtros.

La razón de tener un banco de filtros en lugar de sólo un filtro es que varios filtros que trabajan juntos, con *downsampling*, pueden manifestar un comportamiento que es imposible obtener con un único filtro. La característica más importante de un banco de filtros es su capacidad para reconstruir la entrada a partir de la salidas H_0x y H_1x, a pesar de que cada uno haya sufrido la decimación.

El *downsampling* no es invariante en el tiempo. Después de la reducción de muestras, la salida está formada por los valores de numeración par $y(0), y(2), y(4), \ldots$, Pero si retrasamos las entradas una unidad de tiempo, las nuevas salidas serán $y(-1), y(1), y(3), \ldots$, y éstas son distintas e independientes de las salidas originales. Estas dos secuencias de señales son dos fases del vector y.

Las salidas del banco de análisis se llaman *coeficientes de subbanda*. Pueden ser cuantificados (si la compresión con pérdida es aceptable), y pueden ser comprimidos mediante RLE, Huffman, codificación aritmética, o cualquier otro método. Eventualmente, se alimentan en el *banco de síntesis*, donde primero se efectúa el denominado *upsampled* (que aumenta las muestras insertando ceros para los coeficientes de numeración impar que fueron desechados), luego pasan a través de los filtros inversos F_0 y F_1, y finalmente se combinan para formar un único vector de salida $\hat{x}$. La salida de cada de filtro de análisis (tras la decimación) es:

$$(\downarrow y) = (\ldots, y(-4), y(-2), y(0), y(2), y(4), \ldots).$$

El *upsampling* inserta ceros para los valores ya decimados, lo que convierte el vector de salida anterior en:

$$(\downarrow y) = (\ldots, y(-4), 0, y(-2), 0, y(0), 0, y(2), 0, y(4), 0, \ldots).$$

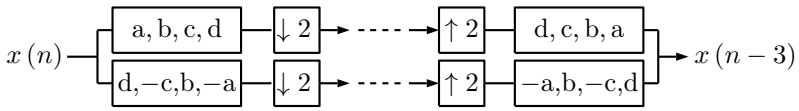


Figura 5.24: Un banco de filtros ortogonal con cuatro coeficientes de filtro.

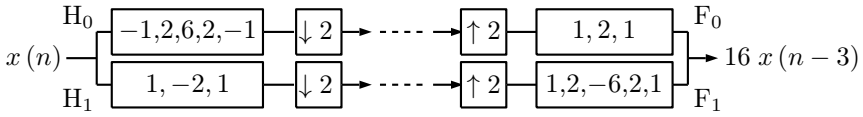


Figura 5.25: Un banco de filtros biortogonal con reconstrucción perfecta.

El *downsampling* causa pérdida de datos. El *upsampling* por sí solo no puede compensarla, porque simplemente inserta ceros para los datos que faltaban. Con el fin de lograr la reconstrucción sin pérdidas de la señal original x, los filtros deben ser diseñados de tal forma que compensen estas pérdidas de datos. Una característica que se usa comúnmente en el diseño de buenos filtros es la ortogonalidad. La Figura 5.24 muestra un conjunto de filtros ortogonales de tamaño 4. Los filtros del conjunto son ortogonales debido a que su producto escalar es cero:

$$(a, b, c, d) \cdot (d, -c, b, -a) = 0.$$

Observe cuán similares son H_0 y F_0 (y también H_1 y F_1). Queda aún, por supuesto, la elección de los valores reales para los cuatro *coeficientes de filtro* a, b, c y d. Una discusión completa sobre ésto está fuera del alcance de este libro, pero la Sección 5.7.1 ilustra algunos de los métodos y reglas que se usan en la práctica para determinar los valores de los diversos coeficientes de filtro. Un ejemplo es el filtro D4 de Daubechies, cuyos valores se muestran en la Ecuación (5.11).

La simulación manual del funcionamiento de este filtro muestra que la entrada reconstruida es idéntica a la entrada original, pero se queda tres unidades de tiempo detrás de ella.

Un banco de filtros también puede ser *biortogonal*, un tipo menos restringido de filtro. La Figura 5.25 muestra un ejemplo de un conjunto de filtros que puede reconstruir una señal exactamente. Observe la similitud de H_0 y F_1 y también de H_1 y F_0.

Ya sabemos, desde la discusión de la Sección 5.6, que las salidas del filtro paso bajo H_0 se transmiten normalmente a través de los filtros de análisis varias veces, creando salidas cada vez más cortas. Este proceso recursivo puede ser ilustrado como un árbol (Figura 5.26). Debido a que cada nodo de este árbol produce la mitad el número de salidas que su predecesor, el árbol se llama *árbol logarítmico*. La Figura 5.26 muestra cómo la función de escalado $\phi(t)$ y la wavelet $\psi(t)$ se obtienen en el límite del árbol logarítmico. Ésta es la conexión entre la transformada discreta wavelet (usando bancos de filtros) y la CWT.

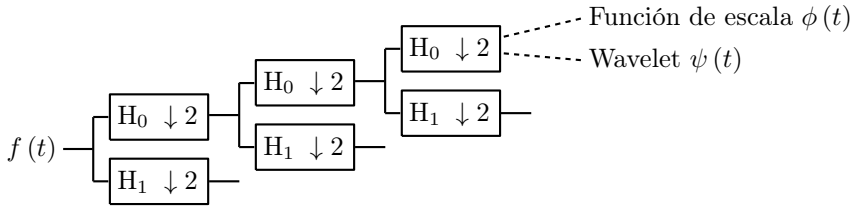


Figura 5.26: Función de escalado y wavelet como límites de un árbol logarítmico.

A medida que "ascendemos" por el árbol logarítmico del nivel i al nivel más próximo $i+1$, calculamos las nuevas medias de las nuevas funciones de escalado de mayor resolución $\phi\left(2^i t-k\right)$ y los nuevos detalles de las nuevas wavelets $\psi\left(2^i t-k\right)$

$$\begin{array}{rl} \text{señal a nivel } i \text{ (promedios)} & \searrow \\ & + \quad \text{señal a nivel } i+1. \\ \text{detalles a nivel } i \text{ (diferencias)} & \nearrow \end{array}$$

Cada nivel en el árbol corresponde al doble de la frecuencia (o el doble de la resolución) del nivel precedente, por lo que el árbol logarítmico también se llama *árbol multirresolución*. Los sucesivos filtrados a través del árbol separaran frecuencias cada vez más bajas.

La gente que hace el trabajo cuantitativo con el sonido y la música saben que dos tonos con frecuencias ω y 2ω suenan como la misma nota y difieren únicamente en el tono. El intervalo de frecuencia entre ω y 2ω se divide en 12 subintervalos (llamados *escala cromática*), pero la música occidental tiene una tradición de favorecer a sólo ocho de los doce tonos que resultan de esta división (una *escala diatónica*, compuesta por siete notas, con la octava nota como "octava"). Por ello, el intervalo de frecuencia básico utilizado en la música se llama tradicionalmente una *octava*. Decimos, por lo tanto, que los niveles adyacentes del árbol multirresolución difieren en una octava de frecuencias.

Para comprender el significado de *paso bajo* y *paso alto* necesitamos trabajar en el dominio de la frecuencia, donde se sustituye la convolución de dos vectores por una multiplicación de sus transformadas de Fourier. El vector $x(n)$ está en el dominio del tiempo, por lo que su dominio de la frecuencia equivalente es su transformada de Fourier discreta (Ecuación (5.1))

$$X(\omega) \stackrel{\text{def}}{=} X\left(e^{i\omega}\right)=\sum_{-\infty}^{\infty} x(n) e^{-in\omega},$$

que en ocasiones se escribe en el dominio de z,

$$X(z)=\sum_{-\infty}^{\infty} x(n) z^{-n},$$

donde $z \stackrel{\text{def}}{=} e^{i\omega}$. La convolución de h en el dominio del tiempo se convierte ahora en una multiplicación mediante la función $H(\omega)=\sum h(n) e^{-in\omega}$ en el dominio de la frecuencia, por lo que podemos expresar la salida en el dominio de la frecuencia con

$$Y\left(e^{i\omega}\right)=H\left(e^{i\omega}\right) X\left(e^{i\omega}\right),$$

o, en notación reducida, $Y(\omega)=H(\omega) X(\omega)$, o, en el dominio de z, $Y(z)=H(z) X(z)$. Cuando todas las entradas son $X(\omega)=1$, la salida a la frecuencia ω es $Y(\omega)=H(\omega)$.

Ahora podemos entender el funcionamiento del filtro de Haar paso bajo. Funciona calculando el promedio de dos entradas consecutivas, por lo que produce la salida:

$$y(n)=\frac{1}{2} x(n)+\frac{1}{2} x(n-1). \tag{5.8}$$

Ésta es una convolución con sólo los dos términos $k=0$ y $k=1$ en la suma. Los coeficientes de filtro son $h(0)=h(1)=1/2$, y podemos denominar la salida *promedio móvil*, ya que cada $y(n)$ depende de la entrada actual y de su predecesor. Si la entrada es la unidad de impulso $x=(\ldots, 0,0,1,0,0, \ldots)$, entonces la salida es $y(0)=y(1)=1/2$, o $y=(\ldots, 0,0,1/2,1/2,0,0, \ldots)$. Los valores de salida son simplemente los coeficientes de filtro como vimos anteriormente.

Podemos ver este filtro promedio como la combinación de un operador identidad y uno de retardo. La salida producida por el operador identidad es igual a la entrada actual, mientras que la salida producida por el de retardo es la entrada que había una unidad de tiempo antes. En consecuencia, podemos escribir:

$$\text{filtro promedio} = \frac{1}{2}\,(\text{identidad}) + \frac{1}{2}\,(\text{retardo})\,.$$

En notación matricial ésto puede ser expresado mediante:

$$\begin{pmatrix} \cdots \\ y\,(-1) \\ y\,(0) \\ y\,(1) \\ \cdots \end{pmatrix} = \begin{pmatrix} \cdots & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ & \frac{1}{2} & \frac{1}{2} & & \\ & & \frac{1}{2} & \frac{1}{2} & \\ & & & \cdots & \end{pmatrix} \begin{pmatrix} \cdots \\ x\,(-1) \\ x\,(0) \\ x\,(1) \\ \cdots \end{pmatrix}.$$

Los valores $1/2$ en la diagonal principal son copias del peso del operador identidad. Todos ellos son iguales al coeficiente $h\,(0)$ del filtro de Haar. Los valores $1/2$ por debajo de la diagonal son copias de los pesos del operador de retardo. Todos ellos son iguales al coeficiente $h\,(1)$ del filtro de Haar. En consecuencia, la matriz es una matriz *diagonal constante* (o una matriz *con bandas*). Un filtro wavelet que tenga un coeficiente $h\,(3)$ correspondería a una matriz en la que dicho coeficiente de filtro aparece en la segunda diagonal por debajo de la diagonal principal. La regla de multiplicación de matrices produce la familiar convolución:

$$y\,(n) = h\,(0)\,x\,(n) + h\,(1)\,x\,(n-1) + h\,(2)\,x\,(n-2) + \cdots = \sum_k h\,(k)\,x\,(n-k)\,.$$

Observe que la matriz es triangular inferior. La diagonal superior que, naturalmente corresponde a los coeficientes de filtro $h\,(-1)\,, h\,(-2)\,, \ldots$, Es cero. Todos los coeficientes de filtro con índices negativos deben ser cero, ya que dichos coeficientes conducen a salidas que preceden a las entradas en el tiempo. En el mundo real, estamos acostumbrados a que una causa es anterior a su efecto, por lo que nuestros filtros de respuesta de impulsos finitos también deben ser *causales*.

Resumen: Un filtro FIR causal con $N+1$ coeficientes de filtro $h\,(0)\,, h\,(1)\,, \ldots, h\,(N)$ (un filtro con $N+1$ "taps" o "derivaciones") tiene $h\,(i) = 0$ para todos los i negativos y para $i > N$. Cuando se expresa en términos de una matriz, la matriz es triangular inferior y con bandas. Tales filtros se utilizan comúnmente y son importantes.

Del diccionario

Derivación (en inglés, *tap*) (nombre).

1. Una conexión cilíndrica o tapón para cerrar una abertura a través de la cual el líquido es drenado, como en un barril; espita.
2. Un grifo o llave.
3. Una conexión hecha en un punto intermedio en un circuito eléctrico o dispositivo.
4. Un acto o una instancia de escuchas telefónicas.

Para ilustrar la respuesta en frecuencia de un filtro seleccionamos un vector de entrada de la forma:

$$x\,(n) = e^{in\omega} = \cos\,(n\omega) + i\,\sin\,(n\omega)\,,\ \text{para}\ -\infty < n < \infty.$$

Ésta es una función compleja cuyas partes real e imaginaria son un coseno y un seno, respectivamente, ambos con frecuencia ω. Recordemos que la transformada de Fourier de un pulso contiene todas las frecuencias (Figura 5.2d,e), pero la transformada de Fourier de una onda sinusoidal tiene una sola frecuencia. La frecuencia más pequeña es $\omega = 0$, para la que el vector se convierte en $x = (\ldots, 1, 1, 1, 1, 1, \ldots)$. La frecuencia más alta es $\omega = \pi$, donde el mismo vector se convierte en $x = (\ldots, 1, -1, 1, -1, 1, \ldots)$. La característica especial de esta entrada es que el vector de salida $y(n)$ es un múltiplo de la entrada.

Para el promedio móvil, la salida (respuesta del filtro) es:

$$y(n) = \frac{1}{2}x(n) + \frac{1}{2}x(n-1) = \frac{1}{2}e^{in\omega} + \frac{1}{2}e^{i(n-1)\omega} = \left(\frac{1}{2} + \frac{1}{2}e^{-i\omega}\right)e^{in\omega} = H(\omega)\,x(n),$$

donde $H(\omega) = \left(\frac{1}{2} + \frac{1}{2}e^{-i\omega}\right)$ es la *función de respuesta en frecuencia* del filtro. Puesto que $H(0) = 1/2 + 1/2 = 1$, vemos que la entrada $x = (\ldots, 1, 1, 1, 1, 1, \ldots)$ se transforma en sí misma. Además, $H(\omega)$ para valores pequeños de ω genera una salida que es muy similar a la de entrada. Este filtro "permite" el paso de las frecuencias bajas, de ahí el nombre de "filtro paso bajo". Para $\omega = \pi$, la entrada es $x = (\ldots, 1, -1, 1, -1, 1, \ldots)$ y la salida es todo ceros (ya que el promedio de 1 y -1 es cero). Este filtro paso bajo suaviza las regiones de alta frecuencia (los abultamientos) de la señal de entrada.

Observe que podemos escribir:

$$H(\omega) = \left(\cos\frac{\omega}{2}\right)e^{i\omega/2}.$$

Cuando representamos la magnitud $|H(\omega)| = \cos(\omega/2)$ de $H(\omega)$ (Figura 5.27a), es fácil ver que tiene un máximo en $\omega = 0$ (la frecuencia más baja) y dos mínimos en $\omega = \pm\pi$ (las frecuencias más altas).

El filtro paso alto utiliza las diferencias para recoger las frecuencias altas en la señal de entrada y reduce o elimina las partes suaves (bajas frecuencia). En el caso de la transformada de Haar, el filtro paso alto calcula

$$y(n) = \frac{1}{2}x(n) + \frac{1}{2}x(n-1) = h \star x,$$

donde los coeficientes del filtro son $h(0) = 1/2$ y $h(1) = -1/2$, ó

$$h = (\ldots, 0, 0, 1/2, -1/2, 0, \ldots).$$

En notación matricial ésto puede ser expresado mediante:

$$\begin{pmatrix} \cdots \\ y(-1) \\ y(0) \\ y(1) \\ \cdots \end{pmatrix} = \begin{pmatrix} \cdots & & & & \\ -\frac{1}{2} & \frac{1}{2} & & & \\ & -\frac{1}{2} & \frac{1}{2} & & \\ & & -\frac{1}{2} & \frac{1}{2} & \\ & & & \cdots & \end{pmatrix} \begin{pmatrix} \cdots \\ x(-1) \\ x(0) \\ x(1) \\ \cdots \end{pmatrix}.$$

La diagonal principal contiene copias de $h(0)$, y la diagonal inmediatamente por debajo de la diagonal contiene $h(1)$. Utilizando los operadores identidad y de retardo, ésto también puede escribirse:

$$\text{filtro paso alto} = \frac{1}{2}(\text{identidad}) - \frac{1}{2}(\text{retardo}).$$

Seleccionando de nuevo la entrada $x(n) = e^{in\omega}$, es fácil ver que la salida es:

$$y(n) = \frac{1}{2}e^{in\omega} - \frac{1}{2}e^{i(n-1)\omega} = \left(\frac{1}{2} - \frac{1}{2}e^{-i\omega}\right)e^{-i\omega/2} = \sin(\omega/2)\,ie^{-i\omega/2}.$$

Esta vez, la función de respuesta de paso alto es:

$$H(\omega) = \frac{1}{2} - \frac{1}{2}e^{-i\omega} = \frac{1}{2}\left(e^{i\omega/2} - e^{-i\omega/2}\right)e^{-i\omega/2} = \sin(\omega/2)\,e^{-i\omega/2}.$$

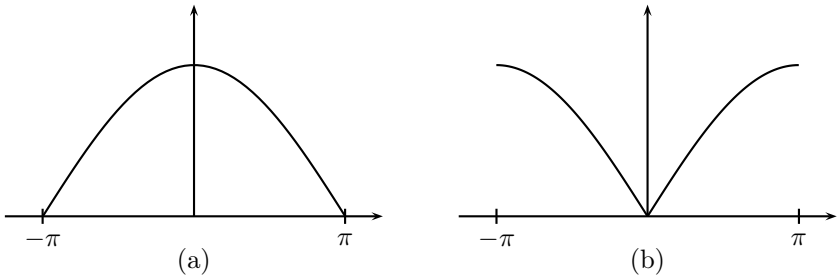


Figura 5.27: Magnitudes de los filtros (a) Paso bajo y (b) Paso alto.

La magnitud es $|H_1(\omega)| = \left|\sin\left(\frac{\omega}{2}\right)\right|$. Se muestra en la Figura 5.27b, y es obvio que tiene un mínimo para la frecuencia cero y dos máximos para las frecuencias grandes.

Una propiedad importante de los bancos de filtros es que ninguno de los filtros individuales es invertible, pero el banco en su conjunto tiene que ser diseñado de tal manera que la señal de entrada pueda ser perfectamente reconstruida a partir de la salida a pesar de la pérdida de datos provocada por la disminución de las muestras (*downsampling*). Es fácil ver, por ejemplo, que la señal constante $x = (\ldots, 1, 1, 1, 1, 1, \ldots)$ es transformada por el filtro paso alto H_1 en un vector de salida de todo ceros. Obviamente, no puede existir un filtro inverso H_1^{-1} que sea capaz de reconstruir la entrada original a partir de un vector cero. Lo mejor que esa transformación inversa puede hacer es utilizar el vector cero para reconstruir otro vector cero.

◇ **Ejercicio 5.8 (sol. en pág. 1097):** Muéstrese un ejemplo de un vector de entrada x que es transformado por el filtro paso H_0 en un vector de todo ceros.

Resumen: El análisis de los bancos de filtros en esta sección debe ser comparado con la discusión de las transformadas de imagen de la Sección 4.4. A pesar de que ambas secciones describen transformaciones, difieren en su enfoque, ya que describen diferentes clases de transformaciones. Cada una de las transformaciones descritas en la Sección 4.4 se basa en un conjunto de funciones base *ortogonales* (o imágenes básicas ortogonales), y se calcula como un producto interno de la señal de entrada con las funciones de base. El resultado es un conjunto de coeficientes de transformada que posteriormente se comprime, ya sea sin pérdidas (mediante RLE o algún codificador de entropía), ya sea con pérdidas (mediante cuantificación seguida de codificación de entropía).

Esta sección aborda las *transformaciones de subbanda* [Simoncelli y Adelson 90], un tipo diferente de transformación que se calcula tomando la *convolución* de la señal de entrada con un conjunto de filtros de paso de banda y efectuando la *decimación* de los resultados. Cada conjunto de coeficientes de transformada decimado es una señal de subbanda que codifica un rango específico de las frecuencias de entrada. La reconstrucción se realiza mediante *upsampling* (añadiendo muestras), seguido por el cálculo de las transformadas inversas, y la mezcla de los conjuntos resultantes de las salidas de los filtros inversos.

La principal ventaja de las transformaciones de subbanda es que aislan las frecuencias diferentes de la señal de entrada, lo que hace posible al usuario controlar con precisión la pérdida de datos en cada rango de frecuencia. En la práctica, tal transformación descompone una imagen en varias subbandas, que corresponden a distintas frecuencias de la imagen, y cada subbanda puede ser cuantificada de manera diferente.

La principal desventaja de este tipo de transformación es la introducción de artefactos, tales como *aliasing*[2] y *ringing*,[3] en la imagen reconstruida, debido al *downsampling* (disminuyendo las muestras).

[2]Los artefactos del aliasing se aprecian en las curvas y líneas inclinadas de la imagen añadiendo un efecto de "escalón", apareciendo ante la vista con bordes "dentados" debido a que se aprecian los bloques de construcción de las mismas.

[3]Los artefactos del ringing aparecen como señales espurias cercanas a las trasiciones bruscas en la señal. Produce

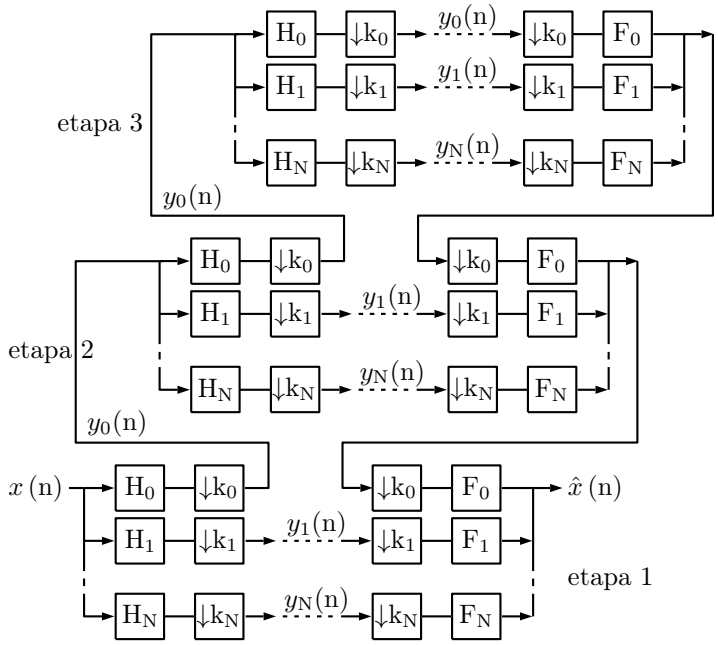


Figura 5.28: Un banco de filtros general.

Por esta razón la transformada de Haar es insatisfactoria, y la mayor parte de la investigación en este campo se refiere a la búsqueda de mejores conjuntos de filtros.

La Figura 5.28 ilustra un caso típico de un banco de filtros de subbanda general con N filtros paso banda y tres etapas. Observe cómo la salida del filtro paso bajo H_0 de cada etapa se envía a la siguiente etapa para una descomposición adicional, y cómo la salida combinada del banco de síntesis de una etapa se envía al filtro inverso superior del banco de síntesis de la etapa precedente.

5.7.1. Derivación de los coeficientes de filtro

Después de presentar el funcionamiento básico de los bancos de filtros, la pregunta natural es: ¿Cómo son derivados los coeficientes de filtro? Una respuesta completa está fuera del alcance de este libro (véase, por ejemplo, [Akansu y Haddad 92]), pero esta sección proporciona un vistazo a las reglas y los métodos utilizados para calcular los valores de diversos bancos de filtros.

Dado un conjunto de dos filtros N-tap (con N derivaciones) directos y dos inversos —H_0 y H_1, y F_0 y F_1— (donde N es par), denotamos sus coeficientes por:

$$h_0 = (h_0(0), h_0(1), \ldots, h_0(N-1)), \quad h_1 = (h_1(0), h_1(1), \ldots, h_1(N-1)),$$
$$f_0 = (f_0(0), f_0(1), \ldots, f_0(N-1)), \quad h_1 = (f_1(0), f_1(1), \ldots, f_1(N-1)).$$

Los cuatro vectores h_0, h_1, f_0, y f_1 son las *respuestas de impulso* de los cuatro filtros. El conjunto más simple de condiciones que estas cantidades tienen que satisfacer es:

1. Normalización: El vector h_0 está normalizado (i.e., su longitud es una unidad).
2. Ortogonalidad: Para cualquier entero i que satisfaga $1 \leq i < N/2$, el vector formado por los primeros $2i$ elementos de h_0 debe ser ortogonal al vector formado por los últimos $2i$ elementos del mismo h_0.

bandas cerca de los bordes de las imágenes.

3. El vector f_0 es el inverso de h_0.

4. El vector h_1 es una copia de f_0, donde los signos de los elementos de numeración impar (el primero, el tercero, etc) se invierten. Podemos expresar ésto diciendo que h_1 se calcula mediante la multiplicación de coordenadas de h_1 y $(-1, 1-1, 1, \ldots, -1, 1)$.

5. El vector f_1 es una copia de h_0 donde los signos de los elementos pares (el segundo, el cuarto, etc) se invierten. Podemos expresar esto diciendo que f_1 se calcula mediante la multiplicación de coordenadas de h_0 y $(1, -1, 1, -1, \ldots, 1, -1)$.

Para los filtros dos-tap, regla 1 implica:

$$h_0^2(0) + h_0^2(1) = 1. \tag{5.9}$$

La regla 2 no es aplicable porque $N = 2$, por lo que $i < {}^N/_2$ implica $i < 1$. Las reglas 3–5 producen:

$$f_0 = (h_0(1), h_0(0)), \quad h_1 = (-h_0(1), h_0(0)), \quad f_1 = (h_0(0), -h_0(1)).$$

Todo depende de los valores $h_0(0)$ y $h_0(1)$, pero sólo la Ecuación (5.9) no es suficiente para determinarlos. Sin embargo, no es difícil ver que la elección $h_0(0) = h_0(1) = {}^1/\sqrt{2}$ satisface la Ecuación (5.9).

Para los filtros cuatro-tap, las reglas 1 y 2 implican:

$$h_0^2(0) + h_0^2(1) + h_0^2(2) + h_0^2(3) = 1, \quad h_0(0)h_0(2) + h_0(1)h_0(3) = 0, \tag{5.10}$$

y las reglas 3–5 generan:

$$\begin{aligned} f_0 &= (h_0(3), h_0(2), h_0(1), h_0(0)), \\ h_1 &= (-h_0(3), h_0(2), -h_0(1), h_0(0)), \\ f_1 &= (h_0(0), -h_0(1), h_0(2), -h_0(3)). \end{aligned}$$

Una vez más, la Ecuación (5.10) no es suficiente para determinar cuatro incógnitas, y son necesarias otras consideraciones (más la intuición matemática) para derivar los cuatro valores. Ellas se enumeran en la Ecuación (5.11) (ésta es el filtro D4 de Daubechies).

◇ **Ejercicio 5.9 (sol. en pág. 1099):** Escríbanse las cinco condiciones anteriores para un filtro ocho-tap.

La determinación de los N coeficientes de filtro para cada uno de los cuatro filtros H_0, H_1, F_0, y F_1 depende de $h_0(0)$ a $h_0(N-1)$, por lo que requiere N ecuaciones. Sin embargo, en cada uno de los casos anteriores, las reglas 1 y 2 proporcionan sólo ${}^N/_2$ ecuaciones. Otras condiciones tienen a ser impuestas y satisfechas antes de que las N cantidades, $h_0(0)$ a $h_0(N-1)$ puedan ser determinadas. He aquí algunos ejemplos:

Filtro *paso bajo* H_0: Queremos que H_0 sea un filtro paso bajo, por lo que tiene sentido exigir que la respuesta en frecuencia $H_0(\omega)$ sea cero para la frecuencia más alta $\omega = \pi$.

Filtro de *fase mínima*: Esta condición requiere que los ceros de la función compleja $H_0(z)$ se encuentren en o dentro del círculo unidad en el plano complejo.

Colinealidad controlada: La linealidad de la respuesta de fase puede ser controlada requiriendo que la suma

$$\sum_i \left(h_0(i) - h_0(N-1-i)\right)^2$$

sea un mínimo.

Otras condiciones se discuten en [Akansu y Haddad 92].

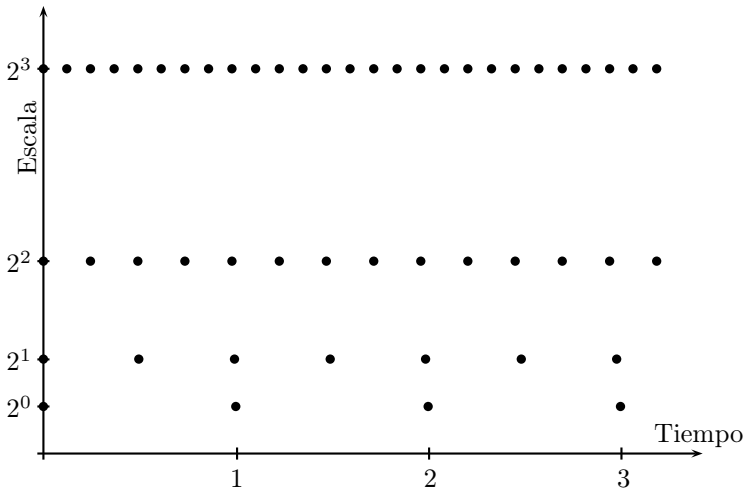


Figura 5.29: La red diádica muestra la relación entre los factores de escala y el tiempo.

5.8. La DWT

La información que es producida analizada en situaciones de la vida real es discreta. Se presenta en forma de números, en lugar de como una función continua. Por esta razón en la práctica se utiliza la transformada wavelet discreta en vez de la continua ([Daubechies 88], [DeVore et al. 92], y [Vetterli y Kovacevic 95]). Recordemos que la CWT [Ecuación (5.3)] es la integral del producto $f(t)\,\psi^*\left(\frac{t-b}{a}\right)$, donde a, el factor de escala, y b, el desplazamiento de tiempo, pueden ser cualesquier números reales. El cálculo correspondiente para el caso discreto (la DWT) implica una *convolución*, pero la experiencia demuestra que la calidad de este tipo de transformación depende en gran medida de dos factores: la elección de los factores de escala y los desplazamientos de tiempo, y la elección de la wavelet.

En la práctica, la DWT se calcula con factores de escala que son potencias negativas de 2 y desplazamientos de tiempo que son potencias no negativas de 2. La Figura 5.29 muestra la llamada *red diádica* (*dyadic lattice*) que ilustra esta elección particular. Las ondas utilizadas son aquellas que generan bases wavelet ortonormales (o biortogonales).

El principal objetivo de la investigación sobre wavelets ha sido, pues, la búsqueda de familias wavelet que forman bases ortogonales. De esas wavelets, las preferidas son aquellas que tienen soporte compacto, ya que permiten efectuar los cálculos de la DWT mediante filtros de *impulsos de respuesta finita* (FIR).

La forma más sencilla de describir la transformada wavelet discreta es mediante la multiplicación de matrices, siguiendo las líneas desarrolladas en la Sección 5.6.3. La transformada de Haar depende de *dos coeficientes* de filtro c_0 y c_1, ambos con un valor de $1/\sqrt{2} \approx 0{,}7071$. La matriz de transformación más pequeña que puede construirse en este caso es $\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}/\sqrt{2}$. Ésta es una matriz de 2×2, y genera dos coeficientes de transformación, un promedio y una diferencia. (Nótese que éstos no son exactamente un promedio y una diferencia, porque se utiliza $\sqrt{2}$ en vez de 2. Mejores nombres para ellos son *detalle grueso —coarse detail—* y *detalle fino —fine detail—*, respectivamente.) En general, la DWT puede utilizar cualquier conjunto de filtros wavelet, pero se calcula de la misma manera, independientemente del filtro particular utilizado.

Comenzamos con una de las wavelets más populares, la D4 de Daubechies. Como su nombre indica, se basa en cuatro coeficientes de filtro c_0, c_1, c_2, y c_3, cuyos valores se muestran en la Ecuación (5.11).

La matriz de transformación W es [compárese con la matriz A_1, Ecuación (5.6)]:

$$W = \begin{pmatrix} c_0 & c_1 & c_2 & c_3 & 0 & 0 & \dots & 0 \\ c_3 & -c_2 & c_1 & -c_0 & 0 & 0 & \dots & 0 \\ 0 & 0 & c_0 & c_1 & c_2 & c_3 & \dots & 0 \\ 0 & 0 & c_3 & -c_2 & c_1 & -c_0 & \dots & 0 \\ \vdots & \vdots & & & & \ddots & & \\ 0 & 0 & \dots & 0 & c_0 & c_1 & c_2 & c_3 \\ 0 & 0 & \dots & 0 & c_3 & -c_2 & c_1 & -c_0 \\ c_2 & c_3 & 0 & \dots & 0 & 0 & c_0 & c_1 \\ c_1 & -c_0 & 0 & \dots & 0 & 0 & c_3 & -c_2 \end{pmatrix}.$$

Cuando esta matriz se aplica a un vector columna de ítems de datos $(x_1, x_2, \dots, x_n)$, su fila superior genera la suma ponderada $s_1 = c_0x_1 + c_1x_2 + c_2x_3 + c_3x_4$, su tercera fila genera la suma ponderada $s_1 = c_0x_3 + c_1x_4 + c_2x_5 + c_3x_6$, y las otras filas de numeración impar generan sumas ponderadas semejantes s_i. Dichas sumas son *convoluciones* del vector de datos x_i con los cuatro coeficientes de filtro. En el lenguaje de wavelets, cada una de ellas se denomina *coeficiente de suavizado* (*smooth coefficient*), y juntos constituyen un filtro de suavizado H.

De forma similar, la segunda fila de la matriz genera las cantidades $d_1 = c_3x_1 - c_2x_2 + c_1x_3 - c_0x_4$, y las otras filas de numeración par generan convoluciones similares. Cada d_i se denomina *coeficiente de detalle*, y juntos forman el llamado filtro G. G no es un filtro de suavizado. De hecho, los coeficientes de filtro se eligen de manera que el filtro G genera valores pequeños cuando los ítems de datos x_i están correlacionados. Juntos, H y G se denominan *filtros espejo de cuadratura* (QMF o *quadrature mirror filters*).

La transformada wavelet discreta de una imagen puede, por lo tanto, ser vista como pasar la imagen original a través de un QMF que consta de un par de filtros paso bajo (H) y paso alto (G).

Si W es una matriz de $n \times n$, genera $n/2$ coeficientes de suavizado s_i y $n/2$ coeficientes de detalle d_i. La matriz transpuesta es:

$$W^T = \begin{pmatrix} c_0 & c_3 & 0 & 0 & \dots & & & & & c_2 & c_1 \\ c_1 & -c_2 & 0 & 0 & \dots & & & & & c_3 & -c_0 \\ c_2 & c_1 & c_0 & c_3 & \dots & & & & & 0 & 0 \\ c_3 & -c_0 & c_1 & -c_2 & \dots & & & & & 0 & 0 \\ & & & & \ddots & & & & & & \\ & & & & & c_2 & c_1 & c_0 & c_3 & 0 & 0 \\ & & & & & c_3 & -c_0 & c_1 & -c_2 & 0 & 0 \\ & & & & & & & c_2 & c_1 & c_0 & c_3 \\ & & & & & & & c_3 & -c_0 & c_1 & -c_2 \end{pmatrix}.$$

Se puede demostrar que para que W sea ortonormal, los cuatro coeficientes deben satisfacer las dos relaciones: $c_0^2 + c_1^2 + c_2^2 + c_3^2 = 1$ y $c_2c_0 + c_3c_1 = 0$. Las otras dos ecuaciones usadas para calcular los cuatro coeficientes de filtro son: $c_3 - c_2 + c_1 - c_0 = 0$ y $0c_3 - 1c_2 + 2c_1 - 3c_0 = 0$. Ellos representan la desaparición de los dos primeros momentos de la secuencia $(c_3, -c_2, c_1, -c_0)$. Las soluciones son:

$$\begin{aligned} c_0 &= (1+\sqrt{3})/(4\sqrt{2}) \approx 0{,}48296, \quad c_1 = (3+\sqrt{3})/(4\sqrt{2}) \approx 0{,}8365, \\ c_2 &= (3-\sqrt{3})/(4\sqrt{2}) \approx 0{,}2241, \quad c_3 = (1-\sqrt{3})/(4\sqrt{2}) \approx -0{,}1294. \end{aligned} \tag{5.11}$$

El uso de una matriz de transformación W es conceptualmente simple, pero no es muy práctico, ya que W debe ser del mismo tamaño que la imagen, que puede ser grande. Sin embargo, una mirada a W muestra que es muy regular, por lo que realmente no hay ninguna necesidad de construir la matriz

completa. Es suficiente tener sólo la fila superior de W. De hecho, es suficiente tener una matriz con los coeficientes de filtro. La Figura 5.30 muestra el código en Matlab que realiza este cálculo. La función `fwt1(dat, coarse, filtro)` toma un vector fila `dat` de 2^n ítems de datos, y otro array, `filtro`, con los coeficientes de filtro. A continuación, calcula los primeros niveles `coarse` de la transformada wavelet discreta (los más toscos).

◇ **Ejercicio 5.10 (sol. en pág. 1099):** Escríbase un código similar para la transformada wavelet discreta unidimensional inversa.

Representación de funciones: Las wavelets se utilizan en muchos campos y tienen muchas aplicaciones, pero la prueba más sencilla de la Figura 5.30 sugiere otra aplicación, a saber, la representación de funciones. Cualquier programa de gráficos o paquete de software gráfico debe incluir una rutina gráfica para representar funciones. Trabaja calculando la función en ciertos puntos y conectando dichos puntos con segmentos rectos. En las regiones donde la función tiene una curvatura pequeña (parece una línea recta) sólo son necesarios unos pocos puntos, mientras que en las zonas donde la función tiene una gran curvatura (cambia de dirección rápidamente) se requieren más puntos. Una rutina de trazado ideal debe, por lo tanto, ser adaptativa. Ésta debe seleccionar los puntos dependiendo de la curvatura de la function.

La curvatura, sin embargo, puede no ser fácil de calcular (está esencialmente determinada por el segunda derivada de la función) por lo que muchas rutinas de trazado usan en su lugar el ángulo entre segmentos consecutivos. La Figura 5.31 muestra cómo trabaja una rutina típica de trazando. Se inicia con un número fijo (digamos, 50) de puntos. Ésto implica 49 segmentos rectos conectándolos. Antes de que cualquiera de los segmentos sean realmente dibujados, la rutina mide los ángulos entre segmentos consecutivos. Si un ángulo en el punto $\mathbf{P}_i$ es extremo (cercano a cero o a 360º, como ocurre en torno a los puntos 4 y 10 en la figura), entonces se calculan más puntos entre los puntos $\mathbf{P}_{i-1}$ y $\mathbf{P}_{i+1}$, de lo contrario (si el ángulo está más cercano a 180º, como, por ejemplo, en torno a los puntos 5 y 9 en la figura), $\mathbf{P}_i$ se considera el único punto necesario en esa región.

Los mejores y más rápidos resultados pueden obtenerse usando una transformada wavelet discreta. La función es evaluada en los n puntos (donde n, un parámetro, es grande), y los valores son recogidos en un vector v. Entonces se calcula una transformada wavelet discreta de v, para producir n coeficientes de transformación. El siguiente paso es descartar m de los coeficientes más pequeños (donde m es otro parámetro). Sabemos, a partir de la discusión anterior, que los coeficientes menores representan pequeños detalles de la función, por lo que ignorarlos deja los detalles importantes prácticamente intactos. A continuación, se efectúa la transformada inversa sobre los restantes $n - m$ coeficientes de transformación, lo que resulta en $n - m$ nuevos puntos que posteriormente se conectan con segmentos rectos. Cuanto mayor es m, menos segmentos son necesarios, pero peor es el ajuste.

Los lectores que se tomen la molestia de leer y comprender las funciones `fwt1` e `iwt1` (Figuras 5.30 y 5.32) pueden estar interesados en sus equivalentes bidimensionales, las funciones `fwt2` e `iwt2`, que se detallan en las Figuras 5.33 y 5.34, respectivamente, con una sencilla rutina de prueba.

La Tabla 5.35 muestra los coeficientes de filtro para algunas de las wavelets más comunes actualmente en uso. Tenga en cuenta que cada uno de esos grupos aún debe ser normalizado. A continuación se presentan las principales características de cada conjunto:

- La familia de filtros Daubechies maximizan la suavidad de la wavelet padre (la función de escalado) maximizando la tasa de descomposición de su transformada de Fourier.
- La wavelet de Haar puede ser considerada un filtro Daubechies de orden 2. Es el filtro más antiguo. Es fácil trabajar con él, pero que no produce los mejores resultados, ya que no es continuo.
- El filtro Beylkin coloca las raíces de la función de respuesta en frecuencia cerca de la tasa de Nyquist (página 544) sobre el eje real.

```
function wc1=fwt1(dat,coarse,filtro)
% La Transformada Wavelet directa 1D
% dat debe ser un vector fila 1D de tamaño 2^n,
% coarse es el nivel de la transformada más tosco (grueso)
% (Observe que el grueso debe ser ≪n)
% filtro es un filtro de espejo de cuadratura ortonormal
% cuya longitud debe ser <2 ^ (grueso +1)
n=length(dat); j=log2(n); wc1=zeros(1,n);
beta=dat;
for i=j-1:-1:coarse
  alfa=HiPass(beta,filtro);
  wc1((2^(i)+1):(2^(i+1)))=alfa;
  beta=LoPass(beta,filtro);
end
wc1(1:(2^coarse))=beta;

function d=HiPass(dt,filtro)% downsampling paso alto
d=iconv(mirror(filtro),lshift(dt));
% iconv es una utilidad de convolución de matlab
n=length(d);
d=d(1:2:(n-1));

function d=LoPass(dt,filtro)% downsampling paso bajo
d=aconv(filtro,dt);
% aconv es una utilidad de convolución de matlab con
% filtro de inversión temporal
n=length(d);
d=d(1:2:(n-1));

function sgn=mirror(filt)
% devuelve los coeficientes de filtro con signos alternantes
sgn=-((-1).^(1:length(filt))).*filt;
```

Un sencillo test de `fwt1` es:

```
n=16; t=(1:n)./n;
dat=sin(2*pi*t)
filt=[0.4830 0.8365 0.2241 -0.1294];
wc=fwt1(dat,1,filt)
```

con salidas:

```
dat=
0.3827 0.7071 0.9239 1.0000 0.9239 0.7071 0.3827 0
-0.3827 -0.7071 -0.9239 -1.0000 -0.9239 -0.7071 -0.3827 0
wc=
1.1365 -1.1365 -1.5685 1.5685 -0.2271 -0.4239 0.2271 0.4239
-0.0281 -0.0818 -0.0876 -0.0421 0.0281 0.0818 0.0876 0.0421
```

Figura 5.30: Código para la transformada wavelet discreta directa unidimensional.

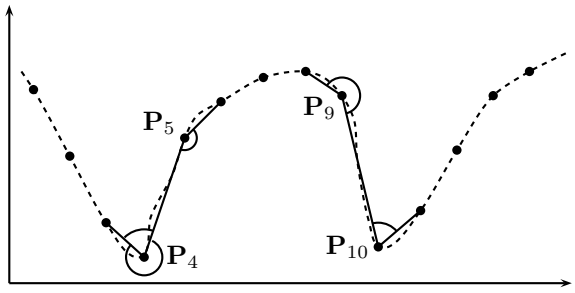


Figura 5.31: Uso de ángulos entre segmentos para añadir más puntos.

```
function dat=iwt1(wc,coarse,filtro)
% Transformada Wavelet Discreta Inversa
dat=wc(1:2^coarse);
n=length(wc); j=log2(n);
for i=coarse:j-1
  dat=ILoPass(dat,filtro)+ ...
    IHiPass(wc((2^(i)+1):(2^(i+1))),filtro);
end

function f=ILoPass(dt,filtro)
f=iconv(filtro,AltrntZro(dt));

function f=IHiPass(dt,filtro)
f=aconv(mirror(filtro),rshift(AltrntZro(dt)));

function sgn=mirror(filt)
% Devuelve los coeficientes de filtro con signos alternantes
sgn=-((-1).^(1:length(filt))).*filt;

function f=AltrntZro(dt)
% Devuelve un vector de longitud 2*n con ceros
% situados entre valores consecutivos
n =length(dt)*2; f =zeros(1,n);
f(1:2:(n-1))=dt;
```

Un sencillo test de `iwt1` es:

```
n=16; t=(1:n)./n;
dat=sin(2*pi*t)
filt=[0.4830 0.8365 0.2241 -0.1294];
wc=fwt1(dat,1,filt)
rec=iwt1(wc,1,filt)
```

Figura 5.32: Código para la transformada wavelet discreta inversa unidimensional.

```
function wc=fwt2(dat,coarse,filtro)
% La Transformada Wavelet Directa 2D
% dat debe ser una matriz 2D de tamaño (2^n:2^n),
% ''coarse'' es el nivel más tosco (grueso) de la transformada
% (observe que coarse debe ser <<n)
% filtro es un qmf orthonormal de longitud <2^(coarse+1)
q=size(dat); n = q(1); j=log2(n);
if q(1)~=q(2), disp('¡Imagen no cuadrada!'), end;
wc = dat; nc = n;
for i=j-1:-1:coarse,
  top = (nc/2+1):nc; bot = 1:(nc/2);
  for ic=1:nc,
    row = wc(ic,1:nc);
    wc(ic,bot)=LoPass(row,filtro);
    wc(ic,top)=HiPass(row,filtro);
  end
  for ir=1:nc,
    row = wc(1:nc,ir)';
    wc(top,ir)=HiPass(row,filtro)';
    wc(bot,ir)=LoPass(row,filtro)';
  end
nc = nc/2;
end

function d=HiPass(dt,filtro)% downsampling paso alto
d=iconv(mirror(filtro),lshift(dt));
% iconv is una utilidad de convolución de matlab
n=length(d);
d=d(1:2:(n-1));

function d=LoPass(dt,filtro)% downsampling paso bajo
d=aconv(filtro,dt);
% aconv es una utilidad de convolución de matlab con
% filtro de inversión temporal
n=length(d);
d=d(1:2:(n-1));

function sgn=mirror(filt)
% devuelve los coeficientes de filtro con signos alternantes
sgn=-((-1).^(1:length(filt))).*filt;
```

Un sencillo test de `fwt2` e `iwt2` es:

```
filename='house128'; dim=128;
fid=fopen(filename,'r');
if fid==-1 disp('archivo no encontrado')
  else img=fread(fid,[dim,dim])'; fclose(fid);
end
filt=[0.4830 0.8365 0.2241 -0.1294];
fwim=fwt2(img,4,filt);
figure(1), imagesc(fwim), axis off, axis square
rec=iwt2(fwim,4,filt);
figure(2), imagesc(rec), axis off, axis square
```

Figura 5.33: Código para la transformada wavelet discreta directa bidimensional.

```
function dat=iwt2(wc,coarse,filtro)
% La Transformada Wavelet Discreta Inversa 2D
n=length(wc); j=log2(n);
dat=wc;
nc=2^(coarse+1);
for i=coarse:j-1,
  top=(nc/2+1):nc; bot=1:(nc/2); all=1:nc;
  for ic=1:nc,
    dat(all,ic)=ILoPass(dat(bot,ic)',filtro)' ...
      +IHiPass(dat(top,ic)',filtro)';
  end% ic
  for ir=1:nc,
  dat(ir,all)=ILoPass(dat(ir,bot),filtro) ...
      +IHiPass(dat(ir,top),filtro);
  end% ir
nc=2*nc;
end% i

function f=ILoPass(dt,filtro)
f=iconv(filtro,AltrntZro(dt));

function f=IHiPass(dt,filtro)
f=aconv(mirror(filtro),rshift(AltrntZro(dt)));

function sgn=mirror(filt)
% devuelve los coeficientes de filtro con signos alternantes
sgn=-((-1).^(1:length(filt))).*filt;

function f=AltrntZro(dt)
% devuelve un vector de longitud 2*n con ceros
% situados entre valores consecutivos
n =length(dt)*2; f =zeros(1,n);
f(1:2:(n-1))=dt;
```

Un sencillo test de `fwt2` e `iwt2` es:

```
filename='house128'; dim=128;
fid=fopen(filename,'r');
if fid==-1 disp('archivo no encontrado')
  else img=fread(fid,[dim,dim])'; fclose(fid);
end
filt=[0.4830 0.8365 0.2241 -0.1294];
fwim=fwt2(img,4,filt);
figure(1), imagesc(fwim), axis off, axis square
rec=iwt2(fwim,4,filt);
figure(2), imagesc(rec), axis off, axis square
```

Figura 5.34: Código para la transformada wavelet discreta inversa bidimensional.

- El filtro de Coifman (o "Coiflet") de orden p (donde p es un número entero positivo) proporciona tanto la wavelet $2p$ madre como la padre en momentos cero.
- Los filtros simétricos (symmlets) son las wavelets más simétricas de soporte compacto con un número máximo de momentos cero.
- El filtro de Vaidyanathan no satisface ninguna condición sobre los momentos, pero produce una reconstrucción exacta. Este filtro es especialmente útil en la compresión de voz. Las Figuras 5.36 y 5.37 son diagramas de algunos de esos wavelets.

La familia de wavelets de Daubechies es un conjunto de funciones ortonormales de soporte compacto donde los miembros consecutivos son cada vez más suaves. Algunos de ellas se muestran en la Figura 5.37. El término *soporte compacto* significa que estas funciones son cero (exactamente iguales a cero, no sólo muy pequeñas) fuera de un intervalo finito.

Las wavelets D4 de Daubechies se basan en cuatro coeficientes, que se muestran en la Ecuación (5.11). La wavelet D6 está, de manera similar, basada en seis coeficientes. Éstos se determinan resolviendo seis ecuaciones, tres de las cuales representan los requisitos de ortogonalidad y las otras tres representan la desaparición de los tres primeros momentos. El resultado se muestra en la Ecuación (5.12):

$$\begin{aligned}
c_0 &= \left(1+\sqrt{10}+\sqrt{5+2\sqrt{10}}\right) / \left(16\sqrt{2}\right) \approx 0{,}3326,\\
c_1 &= \left(5+\sqrt{10}+3\sqrt{5+2\sqrt{10}}\right) / \left(16\sqrt{2}\right) \approx 0{,}8068,\\
c_2 &= \left(10-2\sqrt{10}+2\sqrt{5+2\sqrt{10}}\right) / \left(16\sqrt{2}\right) \approx 0{,}4598,\\
c_3 &= \left(10-2\sqrt{10}-2\sqrt{5+2\sqrt{10}}\right) / \left(16\sqrt{2}\right) \approx -0{,}1350,\\
c_4 &= \left(5+\sqrt{10}-3\sqrt{5+2\sqrt{10}}\right) / \left(16\sqrt{2}\right) \approx -0{,}0854,\\
c_5 &= \left(1+\sqrt{10}-\sqrt{5+2\sqrt{10}}\right) / \left(16\sqrt{2}\right) \approx 0{,}0352.
\end{aligned} \tag{5.12}$$

Cada miembro de esta familia tiene dos coeficientes más que su predecesor y es más suave. La derivación de estas funciones está fuera del alcance de este libro y puede encontrarse en [Daubechies 88]. Ellas se derivan de forma recursiva, no tienen un formato cerrado, y son no diferenciables en infinitos puntos. ¡Unas funciones verdaderamente inusuales!

Funciones no diferenciables

La mayor parte de las funciones que se utilizan en la ciencia y la ingeniería son suaves. Tienen una dirección bien definida (o tangente) en cada punto. Matemáticamente, podemos decir que son *diferenciables*. Algunas funciones pueden tener unos pocos puntos en los que de repente cambian de dirección, y por tanto no tienen una tangente. La wavelet de Haar es un ejemplo de tal función. Una función puede tener muchos de estos ángulos "puntiagudos", incluso infinitos. Un ejemplo sencillo es un onda cuadrada infinita. Ésta cuenta con infinitos puntos puntiagudos, pero no parece extraño o inusual, porque los puntos están separados por áreas lisas de la función.

Lo difícil para nosotros es imaginar (e incluso más duro, aceptar la existencia de) una función continua, pero ¡en ninguna parte diferenciable! Tal función no tiene "huecos" o "aberturas". Continúa

,099305765374	,424215360813	,699825214057	,449718251149	−,110927598348	−,264497231446
,026900308804	,155538731877	−,017520746267	−,088543630623	,019679866044	,042916387274
−,017460408696	−,014365807969	,010040411845	,001484234782	−,002736031626	,000640485329

Beylkin

,038580777748	−,126969125396	−,077161555496	,607491641386	,745687558934	,226584265197

Coifman 1-tap

,016387336463	−,041464936782	−,067372554722	,386110066823	,812723635450	,417005184424
−,076488599078	−,059434418646	,023680171947	,005611434819	−,001823208871	−,000720549445

Coifman 2-tap

−,003793512864	,007782596426	,023452696142	−,065771911281	−,061123390003	,405176902410
,793777222626	,428483476378	−,071799821619	−,082301927106	,034555027573	,015880544864
−,009007976137	−,002574517688	,001117518771	,000466216960	−,000070983303	−,000034599773

Coifman 3-tap

,000892313668	−,001629492013	−,007346166328	,016068943964	,026682300156	−,081266699680
−,056077313316	,415308407030	,782238930920	,434386056491	−,066627474263	−,096220442034
,039334427123	,025082261845	−,015211731527	−,005658286686	,003751436157	,001266561929
−,000589020757	−,000259974552	,000062339034	,000031229876	−,000003259680	−,000001784985

Coifman 4-tap

−,000212080863	,000358589677	,002178236305	−,004159358782	−,010131117538	,023408156762
,028168029062	−,091920010549	−,052043163216	,421566206729	,774289603740	,437991626228
−,062035963906	−,105574208706	,041289208741	,032683574283	−,019761779012	−,009164231153
,006764185419	,002433373209	−,001662863769	−,000638131296	,000302259520	,000140541149
−,000041340484	−,000021315014	,000003734597	,000002063806	−,000000167408	−,000000095158

Coifman 5-tap

,482962913145	,836516303738	,224143868042	−,129409522551

Daubechies 4-tap

,332670552950	,806891509311	,459877502118	−,135011020010	−,085441273882	,035226291882

Daubechies 6-tap

,230377813309	,714846570553	,630880767930	−,027983769417	−,187034811719	,030841381836
,032883011667	−,010597401785				

Daubechies 8-tap

,160102397974	,603829269797	,724308528438	,138428145901	−,242294887066	−,032244869585
,077571493840	−,006241490213	−,012580751999	,003335725285		

Daubechies 10-tap

,111540743350	,494623890398	,751133908021	,315250351709	−,226264693965	−,129766867567
,097501605587	,027522865530	−,031582039317	,000553842201	,004777257511	−,001077301085

Daubechies 12-tap

,077852054085	,396539319482	,729132090846	,469782287405	−,143906003929	−,224036184994
,071309219267	,080612609151	−,038029936935	−,016574541631	,012550998556	,000429577973
−,001801640704	,000353713800				

Daubechies 14-tap

Tabla 5.35: Coeficientes de filtro para algunas wavelets comunes (Parte I).

,054415842243	,312871590914	,675630736297	,585354683654	−,015829105256	−,284015542962
,000472484574	,128747426620	−,017369301002	−,044088253931	,013981027917	,008746094047
−,004870352993	−,000391740373	,000675449406	−,000117476784		

Daubechies 16-tap

,038077947364	,243834674613	,604823123690	,657288078051	,133197385825	−,293273783279
−,096840783223	,148540749338	,030725681479	−,067632829061	,000250947115	,022361662124
−,004723204758	−,004281503682	,001847646883	,000230385764	−,000251963189	,000039347320

Daubechies 18-tap

,026670057901	,188176800078	,527201188932	,688459039454	,281172343661	−,249846424327
−,195946274377	,127369340336	,093057364604	−,071394147166	−,029457536822	,033212674059
,003606553567	−,010733175483	,001395351747	,001992405295	−,000685856695	−,000116466855
,000093588670	−,000013264203				

Daubechies 20-tap

−,107148901418	−,041910965125	,703739068656	1,136658243408	,421234534204	−,140317624179
−,017824701442	,045570345896				

Symmlet 4-tap

,038654795955	,041746864422	−,055344186117	,281990696854	1,023052966894	,896581648380
,023478923136	−,247951362613	−,029842499869	,027632152958		

Symmlet 5-tap

,021784700327	,004936612372	−,166863215412	−,068323121587	,694457972958	1,113892783926
,477904371333	−,102724969862	−,029783751299	,063250562660	,002499922093	−,011031867509

Symmlet 6-tap

,003792658534	−,001481225915	−,017870431651	,043155452582	,096014767936	−,070078291222
,024665659489	,758162601964	1,085782709814	,408183939725	−,198056706807	−,152463871896
,005671342686	,014521394762				

Symmlet 7-tap

,002672793393	−,000428394300	−,021145686528	,005386388754	,069490465911	−,038493521263
−,073462508761	,515398670374	1,099106630537	,680745347190	−,086653615406	−,202648655286
,010758611751	,044823623042	−,000766690896	−,004783458512		

Symmlet 8-tap

,001512487309	−,000669141509	−,014515578553	,012528896242	,087791251554	−,025786445930
−,270893783503	,049882830959	,873048407349	1,015259790832	,337658923602	−,077172161097
,000825140929	,042744433602	−,016303351226	−,018769396836	,000876502539	,001981193736

Symmlet 9-tap

,001089170447	,000135245020	−,012220642630	−,002072363923	,064950924579	,016418869426
−,225558972234	−,100240215031	,667071338154	1,088251530500	,542813011213	−,050256540092
−,045240772218	,070703567550	,008152816799	−,028786231926	−,001137535314	,006495728375
,000080661204	−,000649589896				

Symmlet 10-tap

−,000062906118	,000343631905	−,000453956620	−,000944897136	,002843834547	,000708137504
−,008839103409	,003153847056	,019687215010	−,014853448005	−,035470398607	,038742619293
,055892523691	−,077709750902	−,083928884366	,131971661417	,135084227129	−,194450471766
−,263494802488	,201612161775	,635601059872	,572797793211	,250184129505	,045799334111

Vaidyanathan

Tabla 5.35: Coeficientes de filtro para algunas wavelets comunes (Parte II).

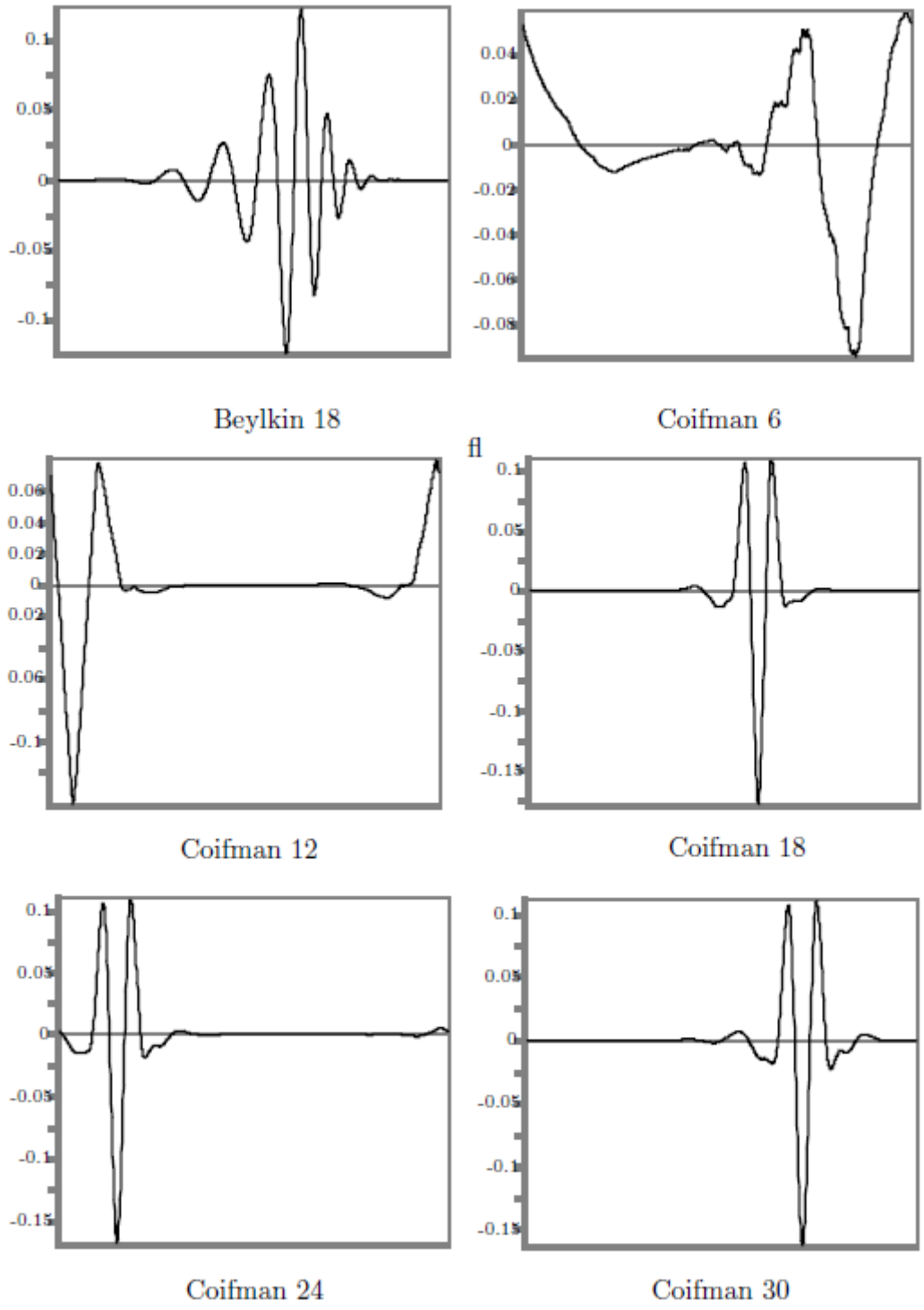


Figura 5.36: Ejemplos de wavelets comunes.

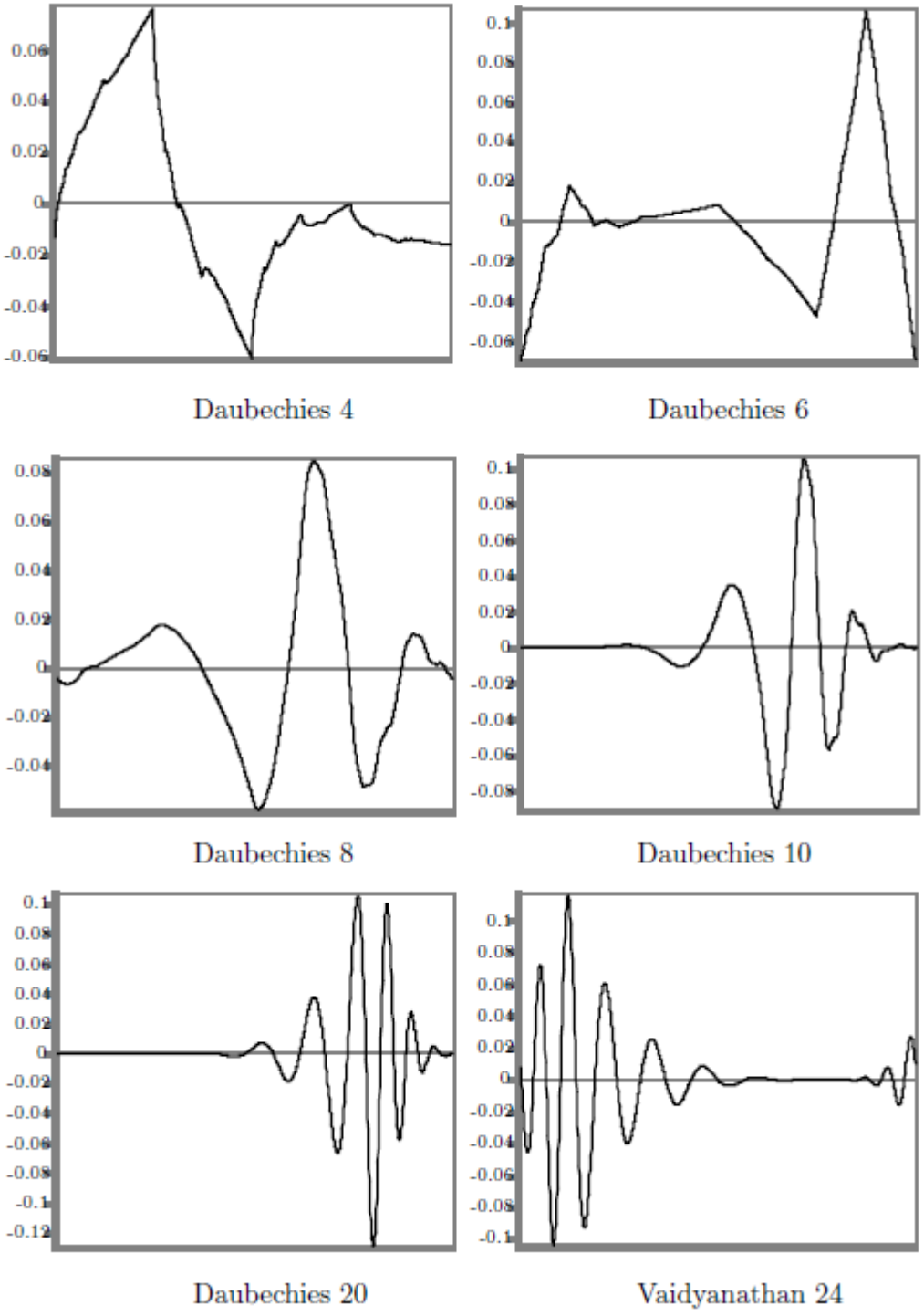


Figura 5.37: Ejemplos de wavelets comunes.

sin interrupciones, pero cambia su dirección bruscamente en *todos los* puntos. Es como si tuviera una cierta dirección en el punto x y una dirección diferente en el punto inmediatamente posterior a x; excepto, por supuesto, que un número real x no tenga un sucesor inmediato.

Tales funciones existen. La primera fue descubierta en 1875 por Karl Weierstrass. Es la suma de la sucesión infinita:

$$W_{b,\omega}(t) = \frac{\sum_{n=0}^{\infty} \omega^n e^{2\pi i b^n t}}{\sqrt{1-\omega^2}},$$

donde $b > 1$ es un número real, ω se escribe ya sea como $\omega = b^h$, con $0 < h < 1$, ya sea como $\omega = b^{d-2}$, con $1 < d < 2$, e $i = \sqrt{-1}$. Observe que $W_{b,\omega}(t)$ es compleja; sus partes real e imaginaria son llamadas funciones coseno y seno de Weierstrass, respectivamente.

Weierstrass demostró el comportamiento inusual de esta función, y también mostró que para $d < 1$ es diferenciable. En su tiempo, esta función era tan contraria al sentido común y a la intuición matemática que él no publicó sus hallazgos. Hoy en día, simplemente llamamos a esta función y otras similares *fractales*.

◇ **Ejercicio 5.11 (sol. en pág. 1099):** Utilícense las funciones `fwt2` e `iwt2` de las Figuras 5.33 y 5.34 para emborronar una imagen. La idea consiste en calcular la transformada de subbanda de 4 pasos de una imagen (terminando así con 13 subbandas) y entonces establecer la mayoría de los coeficientes de la transformada a cero y cuantificar fuertemente algunos de los otros. Ésto, por supuesto, produce una pérdida de información de la imagen, y una reconstrucción imperfecta de la misma. El objetivo de este ejercicio, sin embargo, es que la transformada inversa produzca una *imagen borrosa*. Ésto ilustra una propiedad importante de la transformada wavelet discreta, a saber, su capacidad para reconstruir imágenes que se degradan con elegancia a medida que más y más coeficientes de la transformada se ponen a cero o son fuertemente cuantificados. Otras transformadas, especialmente la DCT, pueden introducir artefactos en la reconstrucción de la imagen, pero esta característica de la DWT la hace ideal para aplicaciones tales como la compresión de huellas dactilares (Sección 5.18).

5.9. Descomposición multirresolución

La idea principal del análisis wavelet, ilustrada en detalle en la Sección 5.5, es analizar una función a diferentes escalas. Se utiliza una wavelet madre para construir wavelets en diferentes escalas (dilataciones) y efectuar una traslacíon de cada una, relativa a la función que está siendo analizada. Los resultados de la traslación de una wavelet dependerán de cuánto se asemeja a la función que se analiza. Wavelets a escalas diferentes (resoluciones) producen resultados diferentes. El principio de la descomposición multirresolución, debida a Stephane Mallat e Yves Meyer, consiste en agrupar todos los coeficientes de la transformada wavelet para una escala determinada, mostrar su superposición y repetir para todas las escalas.

La Figura 5.39 lista una función `multres` en Matlab para esta operación, junto con un test. La Figura 5.38 muestra dos ejemplos: un pico único y picos múltiples. Es fácil ver cómo los coeficientes de resolución fina se concentran en los valores de t que corresponden a los picos (i.e., las zonas de alta actividad de los datos).

5.10. Varias descomposiciones de imágenes

La Sección 5.6.1 muestra dos maneras de aplicar una transformada wavelet discreta a una imagen, con el fin de particionarla en varias subbandas. Esta sección (basada en [Strømme 99], que también contiene comparaciones de resultados experimentales) analiza siete maneras de hacer lo mismo, cada

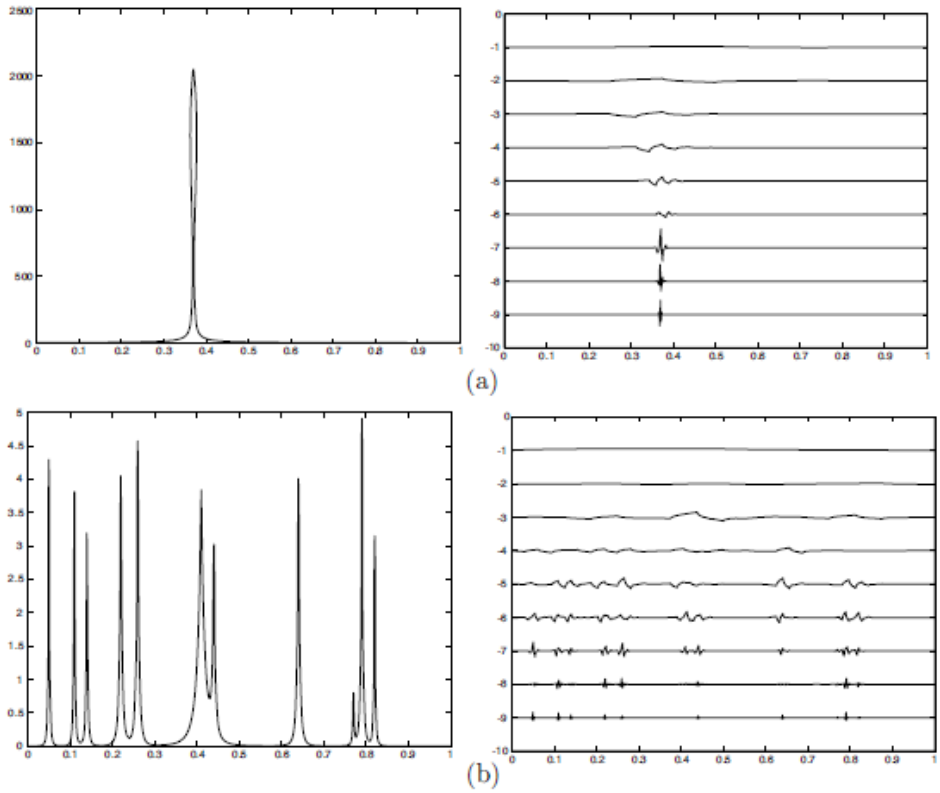


Figura 5.38: Ejemplos de descomposición multirresolución: (a) Un pico. (b) Varios picos.

una con un algoritmo diferente y generando subbandas con distintas compactaciones energía. Otras descomposiciones también son posibles (La Sección 5.18 describe una descomposición especial, simétrica).

Es importante darse cuenta de que los filtros wavelet y el método de descomposición son independientes. La transformada wavelet discreta de una imagen puede utilizar cualquier conjunto de filtros wavelet y puede descomponer la imagen de cualquier manera. La única limitación es que debe tener suficientes puntos de datos en la subbanda para cubrir todas las tomas (*taps*) del filtro. Por ejemplo, si se utiliza un filtro de Daubechies 12-tap, y los tamaños de imagen y de subbanda son potencias de dos, entonces la subbanda más pequeña que puede ser producida tiene un tamaño de 8×8. Ésto es así, porque una subbanda de tamaño 16×16 es la más pequeña que puede ser multiplicada por los 12 coeficientes de este filtro en particular. Una vez que tal subbanda ha sido descompuesta, las subbandas resultantes de 8×8 son demasiado pequeñas para ser multiplicadas por los 12 coeficientes y deben descomponerse aún más.

1. Pirámide Laplaciana: Esta técnica de descomposición de imágenes se describe en detalle en la Sección 5.13. Su característica principal es la transmisión progresiva de imágenes. Durante la descompresión y reconstrucción de la imagen, el usuario ve pequeñas imágenes borrosas, que crecen y se hacen más nítidas. La principal referencia es [Burt y Adelson 83].

La pirámide laplaciana se genera restando una versión paso bajo remuestreada de la imagen a partir de la imagen original. La imagen es particionada en una pirámide Gaussiana (la subbandas paso bajo) y una pirámide Laplaciana formada por los coeficientes que proporcionan detalle (la subbandas paso

alto). Sólo es necesaria la pirámide Laplaciana para reconstruir la imagen. La imagen transformada es más grande que la imagen original, que es la principal diferencia entre la descomposición en pirámide Laplaciana y la descomposición en pirámide (método 4 de más abajo).

2. Línea: Esta técnica es una versión más simple de la descomposición wavelet estándar (método 5). La transformada wavelet es aplicada a cada fila de la imagen, lo que produce coeficientes de suavizado en los coeficientes de la izquierda (subbanda L1) y coeficientes de detalle a la derecha (subbanda H1). La subbanda L1 se divide a continuación en L2 y H2, y el proceso se repite hasta que la matriz de coeficientes completa se convierte en coeficientes de detalle, a excepción de la columna del extremo izquierdo, que contiene coeficientes de suavizado. La transformada wavelet se aplica de forma recursiva a la columna de la izquierda, lo que produce un coeficiente de suavizado en la esquina superior izquierda de la matriz de coeficientes. Este último paso puede omitirse si el método de compresión utilizado requiere que las filas de la imagen se compriman individualmente (nótese la diferencia entre el transformada wavelet y el algoritmo de compresión real).

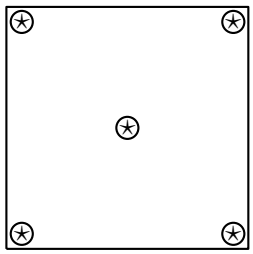
Esta técnica explota correlaciones sólo dentro de una fila de la imagen para calcular los coeficientes de la transformada. También, descartando un coeficiente que se encuentra en la columna del extremo izquierdo puede afectar a sólo un grupo particular de filas y de esta manera puede introducir artefactos en la imagen reconstruida.

La implementación de este método es sencilla, y la ejecución es rápida, aproximadamente el doble que con la descomposición estándar. Este tipo de descomposición se ilustra en la Figura 5.40.

Es posible aplicar esta descomposición a las columnas de la imagen, en lugar de a las filas. Idealmente, la transformada debe aplicarse en la dirección de más alta redundancia de la imagen, y la experiencia indica que para ofrecer imágenes naturales esta es la dirección horizontal. Por lo tanto, en la práctica, la descomposición de línea se aplica a las filas de la imagen.

Del Diccionario

QUINCUNCE (*Quincunx*): Una disposición de cinco objetos en un cuadrado o rectángulo, uno en cada esquina, y uno en el centro. La palabra parece tener su origen en el latín *quincunx*, hacia 1640–1650. Procede de "cinco doceavos". *Quinc* es una variación de *quinque*, y *unx* o *unc* es una forma de *uncia*, que significa duodécimo. Fue usada para indicar una moneda romana de valor las cinco doceavas partes de un AS y marcada con un quincunce de puntos. (AS es una unidad romana antigua de peso, equivalente a aproximadamente 12 onzas.)



---

3. Quincunce o tresbolillo: Algo similar a la pirámide Laplaciana, la descomposición en quincunce procede nivel por nivel y descompone la subbanda L_i del nivel i en las subbandas H_{i+1} y L_{i+1} del nivel $i+1$. La Figura 5.41 ilustra este tipo de descomposición. El método se debe a Strømme y McGregor [Strømme y McGregor 97], quienes originalmente la llamaron descomposición no estándar. (Véase también [Starck et al. 98] para una presentación diferente de este método.) Es computacionalmente eficiente y simple. En promedio, se logra más de cuatro veces la energía de compactación del método de línea.

La descomposición en quincunce produce menos subbandas que la mayoría de las otras descomposiciones wavelet, una característica que puede dar lugar a imágenes reconstruidas con una calidad visual ligeramente más baja. El método no se utiliza mucho en la práctica, pero [Strømme 99] presenta

```
function multres(wc,coarse,filtro)
% Un gráfico multirresolución de una transformada wavelet
1D
scale=1./max(abs(wc));
n=length(wc); j=log2(n);
LockAxes([0 1 -(j) (-coarse+2)]);
t=(.5:(n-.5))/n;
for i=(j-1):-1:coarse
  z=zeros(1,n);
  z((2^(i)+1):(2^(i+1)))=wc((2^(i)+1):(2^(i+1)));
  dat=iwt1(z,i,filtro);
  plot(t,-(i)+scale.*dat);
end
z=zeros(1,n);
z(1:2^(coarse))=wc(1:2^(coarse));
dat=iwt1(z,coarse,filtro);
plot(t,(-(coarse-1))+scale.*dat);
UnlockAxes;
```

Y un test de la rutina:

```
n=1024; t=(1:n)./n;
dat=spikes(t);
% varios picos
%p=floor(n*.37); dat=1./abs(t-(p+.5)/n);% un pico
figure(1), plot(t,dat)
filt=[0.4830 0.8365 0.2241 -0.1294];
wc=fwt1(dat,2,filt);
figure(2), plot(t,wc)
figure(3)
multres(wc,2,filt);

function dat=spikes(t)
pos=[.05 .11 .14 .22 .26 .41 .44 .64 .77 .79 .82];
hgt=[5 5 4 4.5 5 3.9 3.3 4.6 1.1 5 4];
wth=[.005 .005 .006 .01 .01 .03 .01 .01 .005 .008 .005];
dat=zeros(size(t));
for i=1:length(pos)
  dat=dat+hgt(i)./(1+abs((t-pos(i))./wth(i))).^4;
end;
```

Figura 5.39: Código en Matlab: descomposición multirresolución de un vector fila unidimensional.

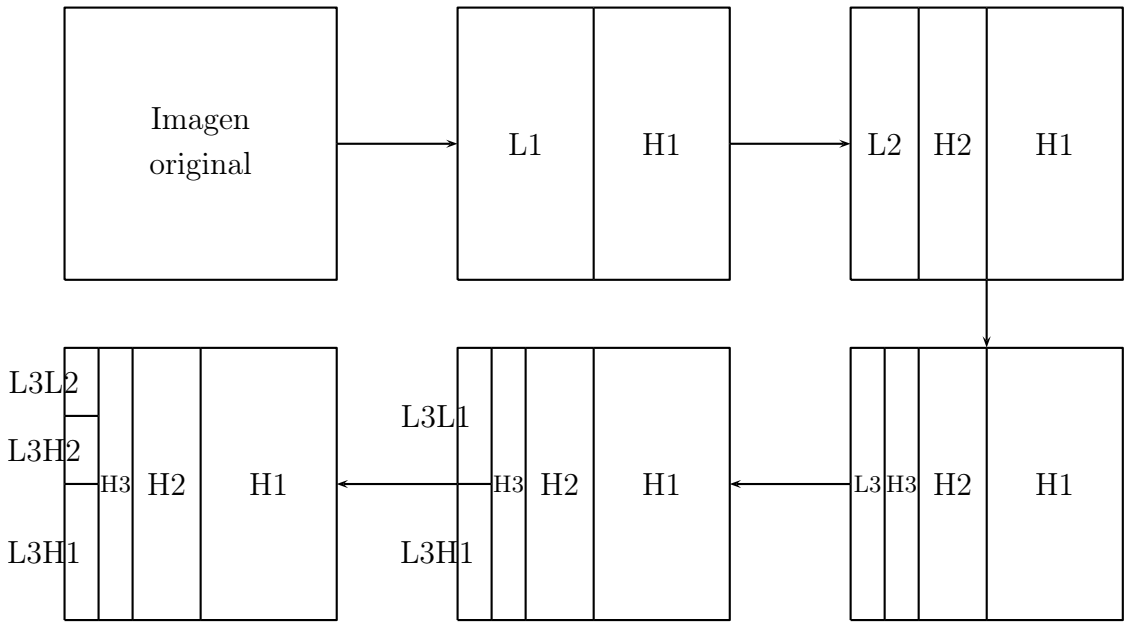


Figura 5.40: Descomposición wavelet de línea.

resultados que sugieren que la descomposición en tresbolillo funciona extremadamente bien y puede ser la de mejor rendimiento en muchas situaciones prácticas.

4. Pirámide: La descomposición en pirámide es, con mucho, el método más común utilizado para descomponer las imágenes que son transformadas wavelet. Esto produce subbandas con detalles de la imagen horizontal, vertical y diagonal, como se ilustra en la Figura 5.17. Las tres subbandas en cada nivel contienen características de la imagen horizontal, vertical y diagonal a una escala particular, y cada escala se divide por una octava en el espacio de frecuencias (división de la frecuencia por dos).

La descomposición en pirámide resulta ser una forma muy eficiente de transferir datos visuales significativos a los coeficientes de detalle. Su complejidad computacional es aproximadamente un 30 % mayor que el del método quincunce, pero sus capacidades de reconstrucción de la imagen son mayores. Las razones de la popularidad del método en pirámide pueden deberse a que: (1) es simétrica, (2) su descripción matemática es simple, y (3) fue utilizado por el prestigioso documento [Mallat 89].

La Figura 5.42 ilustra la descomposición en pirámide. Es evidente que el primer paso es idéntico al de la descomposición en quincunce. Sin embargo, mientras que el método de tresbolillo deja la subbanda de alta frecuencia intacta, el método de pirámide la separa en dos bandas. Por otra parte, la descomposición en pirámide involucra más cálculos con el fin de separar la banda de alta frecuencia asimétrica en dos bandas simétricas, de alta y de baja frecuencia.

5. Estándar: El primer paso en la descomposición estándar es aplicar cualquier filtro wavelet discreto que esté siendo utilizado a todas las filas de la imagen, obteniendo las subbandas L_1 y H_1. Ésto se repite en L_1 para obtener L_2 y H_2, y así sucesivamente k veces. Ésto es seguido por una segunda etapa donde se aplica un cálculo similar k veces a las columnas. Si $k = 1$, la descomposición alterna entre filas y columnas, pero k puede ser mayor que 1. El resultado final es tener un coeficiente de suavizado en la esquina superior izquierda de la matriz de coeficientes. Este método es algo similar a la descomposición de línea. Una característica importante de la descomposición estándar es que cuando un coeficiente es cuantificado, puede afectar a una larga y estrecha área rectangular en la imagen reconstruida. En consecuencia, una cuantificación muy tosca puede producir artefactos en la

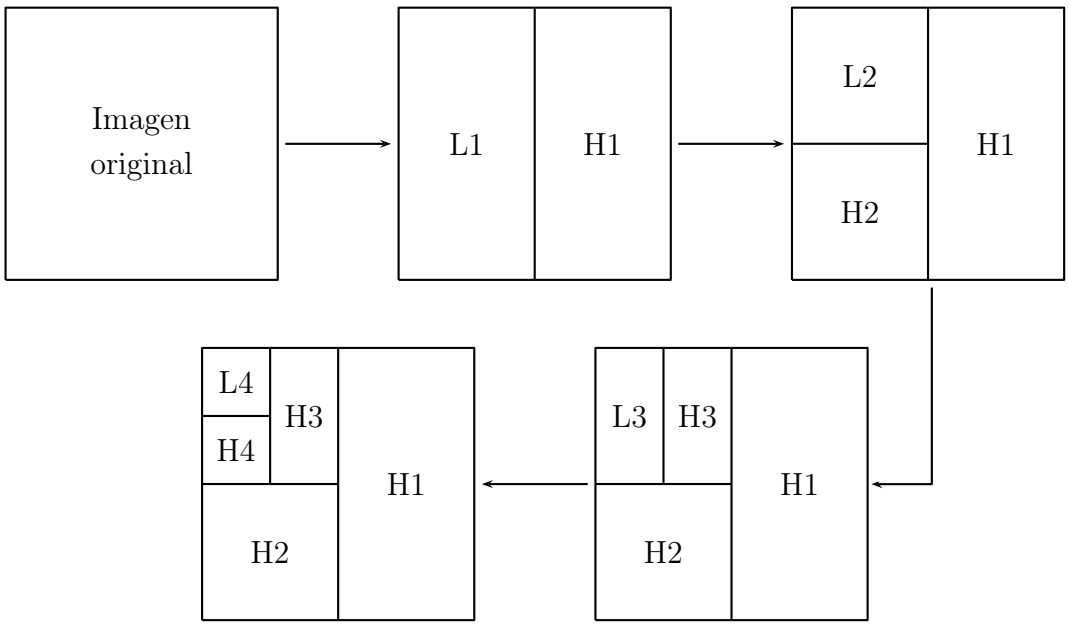


Figura 5.41: Descomposición wavelet en quincuce o tresbolillo.

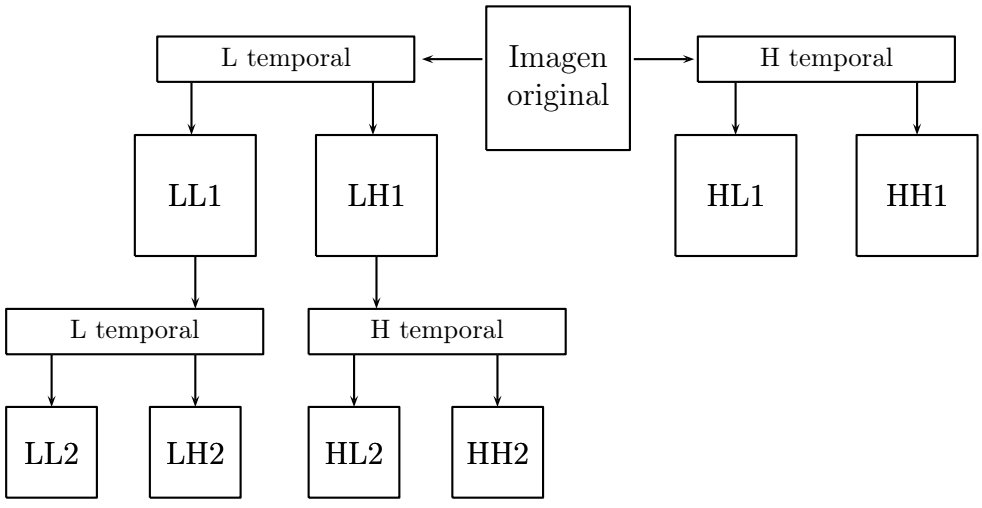


Figura 5.42: Descomposición wavelet en pirámide.

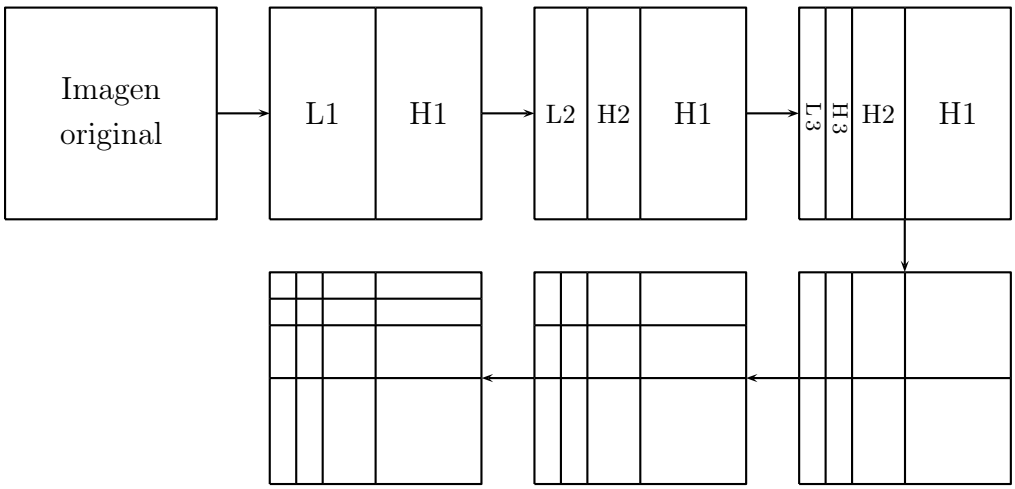


Figura 5.43: Descomposición wavelet estándar.

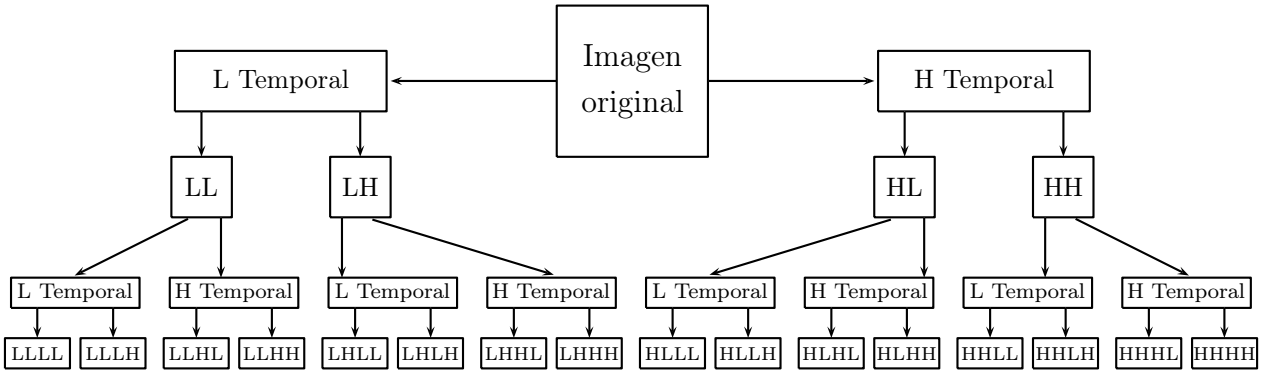
imagen reconstruida en forma de rectángulos horizontales.

La descomposición estándar, ilustrada en la Figura 5.43, tiene la segunda más alta calidad de reconstrucción de los siete métodos aquí descritos. La razón de la mejora en comparación con la descomposición en pirámide puede ser que la mayor resolución direccional proporciona una mejor oportunidad de encontrar valores umbral para cubrir grandes áreas uniformes. Por otro lado, la descomposición estándar es computacionalmente más costosa que la descomposición en pirámide.

6. Descomposición uniforme: Este método es también llamado *transformada de paquetes wavelet* (*wavelet packet transform*). Se ilustra en la Figura 5.44. En el caso donde el usuario elige calcular todos los niveles de la transformada, la descomposición uniforme se vuelve similar a la transformada de Fourier discreta (DFT) en la que cada coeficiente representa una frecuencia espacial para toda la imagen. En tal caso (donde todos los niveles son calculados), la eliminación de un coeficiente en la imagen transformada afecta a la imagen reconstruida completa.

El coste computacional de la descomposición uniforme es muy alta, ya que, efectivamente calcula n^2 coeficientes para cada nivel de descomposición, donde n es la longitud lateral de la imagen (cuadrada). A pesar de tener cualidades promedio de reconstrucción comparativamente altas, la calidad de la percepción de la imagen comienza a degradarse a ratios menores que para los otros métodos de descomposición. La razón de ésto es la misma que para los métodos de Fourier: Puesto que el soporte a un coeficiente único es global, su eliminación tiene el efecto de borrar la imagen reconstruida. La conclusión es que el aumento de la complejidad computacional de la descomposición uniforme no da lugar a una mayor calidad de la imagen reconstruida.

6.1. Descomposición wavelet completa: (Este es un caso especial de descomposición uniforme). Denotamos la imagen original por I_0. Asumimos que su tamaño es $2^l \times 2^l$. Después de la aplicación de la transformada wavelet discreta a la misma, obtenemos una matriz I_1 particionada en cuatro subbandas. La misma transformada wavelet discreta (i.e., utilizando los mismos filtros wavelet) es luego aplicada recursivamente a cada una de las cuatro subbandas de forma individual. El resultado es una matriz de coeficientes consistente en 16 subbandas. Cuando este proceso se lleva a cabo r veces, el resultado es una matriz de coeficientes formada por $2^r \times 2^r$ subbandas, cada una de tamaño $2^{l-r} \times 2^{l-r}$. La subbanda superior izquierda contiene los coeficientes de suavizado (dependiendo del filtro wavelet particular utilizado, puede parecer una versión más pequeña de la imagen original), y las otras subbandas contienen los coeficientes de detalle. Cada subbanda corresponde a una banda de



LLLL	LLHL	HLLL	HLHL
LLLH	LLHH	HLLH	HLHH
LHLL	LHHL	HHLL	HHHL
LHLH	LHHH	HHLH	HHHH

Figura 5.44: Descomposición wavelet uniforme.

frecuencias, mientras que cada coeficiente de transformada individual corresponde a una región espacial local. Al aumentar el nivel de recursividad r, podemos incrementar la resolución de la frecuencia a expensas de resolución espacial.

Este tipo de descomposición de imágenes wavelet fue propuesta por Wong y Kuo [Wong y Kuo 93], quienes recomiendan encarecidamente su uso. Sin embargo, parece que ha sido ignorado por los investigadores e implementadores en el campo de la compresión de imágenes.

7. Descomposición adaptativa de paquetes wavelet: El método de descomposición uniforme es costoso en términos de cómputos, y el método adaptativo de paquetes wavelet lo es potencialmente aún más. La idea es saltar aquellas divisiones de subbandas que no contribuyen significativamente a la compactación de la energía. El resultado es una matriz de coeficientes con subbandas de distintos (posiblemente muchos) tamaños. En la parte inferior derecha de la Figura 5.45 se muestra tal caso (posterior a [Meyer et al 98], el cual muestra la matriz adaptativa de la transformada de paquetes wavelet para la imagen binivel "mandril", la Figura 4.54).

La justificación de este complejo método de descomposición es la prevalencia de imágenes de tonos continuos (naturales). Estas imágenes, que se discuten al inicio del capítulo 4, son en su mayoría suaves pero normalmente tienen también algunas regiones con datos de alta frecuencia. Tales regiones deberían terminar como muchas subbandas pequeñas (para permitir una representación más fiel de la frecuencia espacial de la imagen), con el resto de la imagen, lo que da lugar a unas pocas subbandas grandes. Los coeficientes de la parte inferior izquierda de la matriz en la Figura 5.45 son un ejemplo de imagen muy uniforme, lo que produce 10 subbandas. (El test para una división depende de la magnitud absoluta de los coeficientes de la transformada. Por consiguiente, el test puede ser ajustado tan alto que se efectúen muy pocas divisiones.)

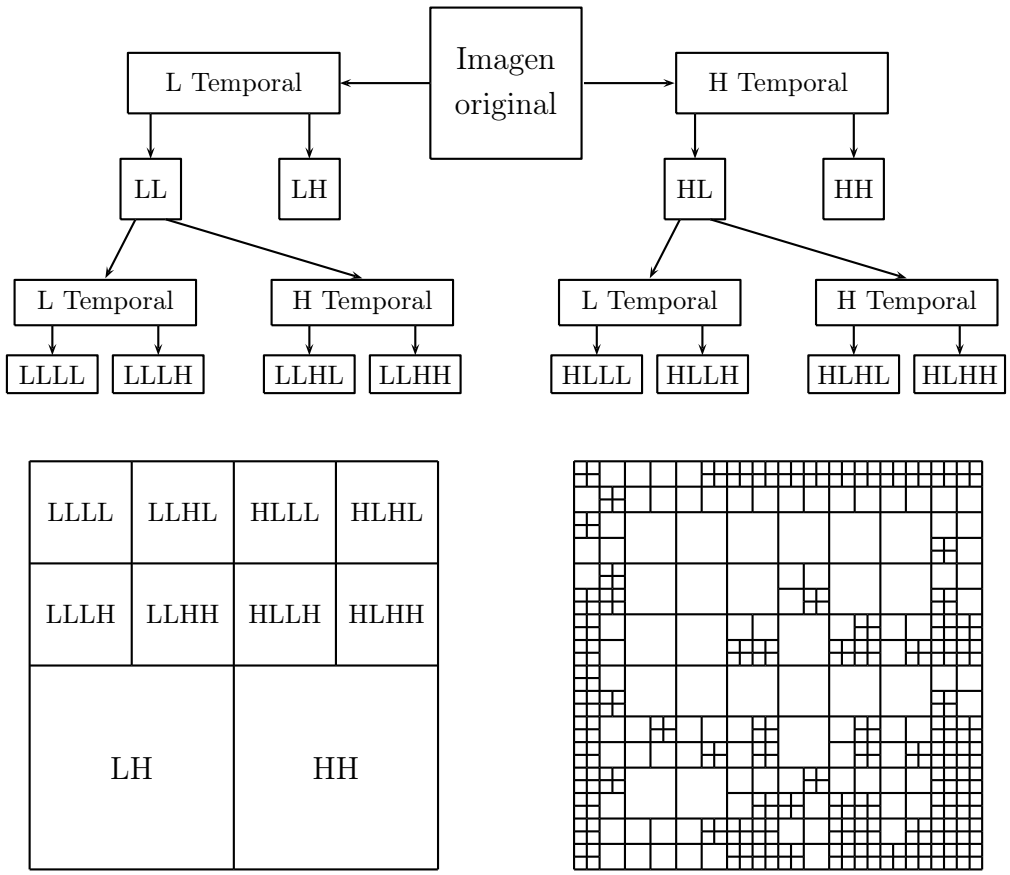


Figura 5.45: Descomposición adaptativa de paquetes wavelet.

La desventaja de este tipo de descomposición es encontrar un algoritmo que determine que divisiones de subbanda se pueden omitir. Dicho algoritmo utiliza cálculos de entropía y debe ser eficiente. Debe identificar todas las divisiones que no tienen que ser realizadas, y debería identificar el mayor número posible de ellas. Un algoritmo ineficiente puede conducir a la división de cada subbanda, efectuando de esta manera muchos cálculos innecesarios y terminando con una matriz de coeficientes donde cada coeficiente es una subbanda, en cuyo caso esta descomposición se reduce a la descomposición uniforme.

Este tipo de descomposición tiene la más alta calidad de reproducción de todos los métodos aquí discutidos, una característica que puede justificar los altos costos computacionales en ciertas aplicaciones especiales. Esta calidad, sin embargo, no es mucho mayor que lo que se consigue con métodos de descomposición más sencillos, como el estándar, en pirámide, o en quincunce.

5.11. El esquema lifting

El esquema lifting ([Stollnitz et al. 96] y [Sweldens y Schröder 96]) es una nueva y útil manera de ver la transformada wavelet discreta. Es fácil de comprender, ya que realiza todas las operaciones en el dominio del tiempo, en lugar de en el dominio de la frecuencia, y tiene también otras ventajas. Esta sección ilustra el método de lifting usando la transformada de Haar, que ya es familiar para el lector,

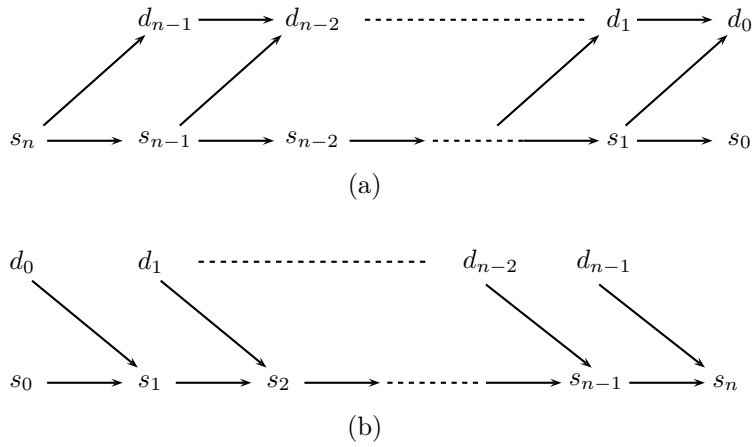


Figura 5.46: (a) La transformada wavelet de Haar y (b) Su inversa.

como ejemplo. La sección siguiente extiende el mismo enfoque a otras transformadas.

La transformada de Haar, descrita Sección 5.6, se basa en los cálculos de promedios y diferencias (referencia). Dados dos píxeles adyacentes a y b, el principio es calcular el promedio $s = {}^{(a+b)}/_2$ y la diferencia $d = b - a$. Si a y b son similares, s será similar a ambos, y d será pequeño, i.e., requiere menos bits para la representación. Esta transformación es reversible, porque $a = s - {}^{d}/_2$ y $b = s + {}^{d}/_2$, y puede ser escrita utilizando la notación matricial como:

$$(s,d) = (a,b)\begin{pmatrix} {}^1/_2 & -1 \\ {}^1/_2 & 1 \end{pmatrix} = (a,b)\,\mathbf{A}, \quad (a,b) = (s,d)\begin{pmatrix} 1 & 1 \\ -{}^1/_2 & {}^1/_2 \end{pmatrix} = (s,d)\,\mathbf{A}^{-1}.$$

Considere una fila de 2^n píxeles de valores $s_{n,l}$ para $0 \le l < 2^n$. Existen 2^{n-1} pares de píxeles $s_{n,2l}, s_{n,2l+1}$ para $l = 0, 2, 4, \ldots, 2^{n-2}$. Cada par se transforma en un promedio $s_{n-1,l} = (s_{n,2l} + s_{n,2l+1})/2$ y una diferencia $d_{n-1,l} = s_{n,2l+1} - s_{n,2l}$. El resultado es un conjunto s_{n-1} de 2^{n-1} promedios y un conjunto d_{n-1} de 2^{n-1} diferencias.

Las mismas operaciones pueden aplicarse a los 2^{n-1} promedios $s_{n-1,l}$ del conjunto s_{n-1}, produciendo 2^{n-2} promedios $s_{n-2,l}$, y 2^{n-2} diferencias $d_{n-2,l}$. Tras aplicar estas operaciones n veces nos encontramos con un conjunto s_0 que consta de un promedio $s_{0,0}$, y con n conjuntos de diferencias $d_{j,l}$ donde $j = 0, 1, \ldots, n$ y $l = 0, 1, \ldots, 2^j - 1$. El conjunto j está formado por j diferencias, por lo que el número total de diferencias es:

$$\sum_{j=0}^{n-1} 2^j = 2^n - 1.$$

La adición del único promedio $s_{0,0}$ lleva el número total de resultados a 2^n, el mismo que el número de valores de píxel originales. Observe que el promedio final $s_{0,0}$ es el promedio S de los 2^n valores de píxel originales, por lo que puede ser llamado componente DC de los valores originales. De hecho, si nos fijamos en cualquier conjunto s_j de promedios $s_{j,l}$, $l = 0, 1, \ldots, 2^j - 1$, nos encontramos con que la media:

$$S = \frac{1}{2^j} \sum_{l=0}^{2^j-1} s_{j,l}$$

es el promedio de todos los 2^n valores de píxel originales. En consecuencia, la media S de un conjunto s_j es independiente de j. La Figura 5.46a,b ilustra la transformada y su inversa.

La idea principal en el esquema lifting es efectuar todas las operaciones necesarias sin utilizar espacio extra. La transformada completa se realiza *in situ*, reemplazando la imagen original. Comenzamos con un par de píxeles consecutivos a y b. Éstos se sustituyen por su media s y su diferencia d primeramente reemplazando b con $d = b - a$, después reemplazando a con $s = a + {}^{d}/_{2}$ [puesto que $d = b - a$, $a + {}^{d}/_{2} = a + {}^{(b-a)}/_{2}$ es igual a ${}^{(a+b)}/_{2}$]. En lenguaje C, esto se escribe:

```
b-=a;  a+=b/2;
```

◇ **Ejercicio 5.12 (sol. en pág. 1099):** Escríbanse las operaciones opuestas en C.

Ésto es fácil de aplicar a toda una fila de píxeles. Supongamos que tenemos una fila s_j con 2^j valores y queremos transformarla en una fila s_{j-1} con 2^{j-1} promedios y 2^{j-1} diferencias. El esquema lifting efectúa esta transformación en tres pasos: *dividir*, *predecir* y *actualizar*.

La operación de división escinde la fila s_j en dos conjuntos separados denotados $\text{even}_{\text{j}-1}$ y $\text{odd}_{\text{j}-1}$. El primero contiene todos los valores pares $s_{j,2l}$, y el segundo contiene todos los valores impares $s_{j,2l+1}$. El rango de l es de 0 a $2^j - 1$. Este tipo de división en valores pares e impares se denomina *transformada wavelet lenta* (*lazy wavelet transform*). La denotamos por:

$$(\text{even}_{\text{j}-1}, \text{odd}_{\text{j}-1}) := \text{Split}\,(s_j)\,.$$

La operación de predicción utiliza el conjunto par $\text{even}_{\text{j}-1}$ para predecir el conjunto impar $\text{odd}_{\text{j}-1}$. Está basado en el hecho de que cada valor $s_{j,2l+1}$ en el conjunto impar es adyacente al valor correspondiente $s_{j,2l}$ en el conjunto par. Por consiguiente, los dos valores están correlacionados y, el uno puede utilizarse para predecir el otro. Recuérdese que una diferencia general $dj - 1, l$ se calcula como la diferencia $d_{j-1,l} = s_{j,2l+1} - s_{j,2l}$ entre un valor impar y un valor par adyacente incluso (o entre un valor impar y su predicción), por lo que podemos definir el operador de predicción P como:

$$d_{j-1} = \text{odd}_{\text{j}-1} - P\,(\text{even}_{\text{j}-1})\,.$$

La operación de actualización U sigue a la etapa de predicción. Calcula las 2^{j-1} medias $s_{j-1,l}$ como la suma:

$$s_{j-1,l} = s_{j,2l} + d_{j-1,l}/2. \tag{5.13}$$

Esta operación se definine formalmente mediante:

$$s_{j-1} = \text{even}_{\text{j}-1} + U\,(d_{j-1})\,.$$

◇ **Ejercicio 5.13 (sol. en pág. 1101):** Utilícese la Ecuación (5.13) para demostrar que los conjuntos s_j y s_{j-1} tienen el mismo promedio.

El punto importante es observar que las tres operaciones pueden efectuarse in situ. Las ubicaciones pares del conjunto s_j son sobrescritas con los promedios (i.e., con el conjunto $\text{even}_{\text{j}-1}$), y las ubicaciones impares son sobrescritas con las diferencias (conjunto $\text{odd}_{\text{j}-1}$). La secuencia de tres operaciones pueden resumirse mediante:

$$(\text{odd}_{j-1}, \text{even}_{j-1}) := \text{Split}\,(s_j)\,; \quad \text{odd}_{j-1} -= P\,(\text{even}_{j-1})\,; \quad \text{even}_{j-1} += U\,(\text{odd}_{j-1})\,;$$

La Figura 5.47a es un diagrama de conexiones de este proceso.

La transformada inversa es similar. Se basa en las tres operaciones: *deshacer actualización*, *deshacer predicción*, y *fusionar* (*merge*).

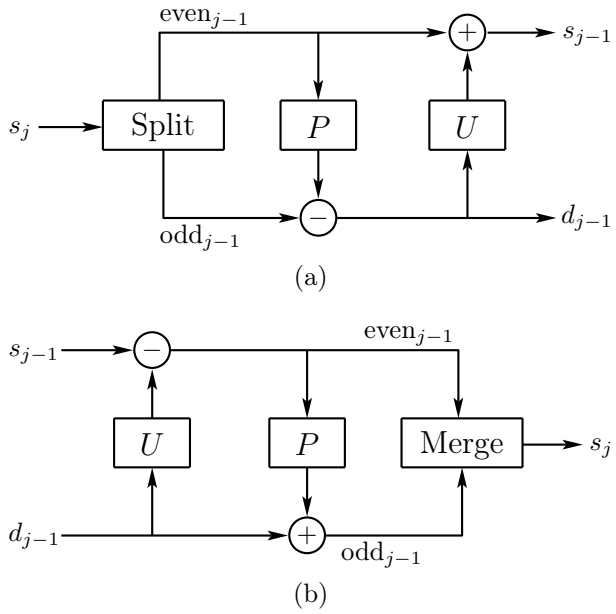


Figura 5.47: Esquema de lifting. Transformada (a) directa, (b) inversa.

Dados dos conjuntos s_{j-1} y d_{j-1}, la operación "deshacer actualización" reconstruye los promedios $s_{j,2l}$ (los valores pares del conjunto s_j) restando la operación de actualización. Genera el conjunto $\text{even}_{\text{j}-1}$ restando los conjuntos $s_{j-1} - U\left(d_{j-1}\right)$. Escrita explícitamente, esta operación se convierte en:

$$s_{j,2l} = s_{j-1,l} - d_{j-1,l}/2, \text{ para } 0 \leq l < 2^j.$$

Dados los dos conjuntos $\text{even}_{\text{j}-1}$ y d_{j-1}, la operación "deshacer predicción" reconstruye las diferencias $s_{j,2l+1}$ (los valores impares del conjunto s_j) mediante la adición del operador de predicción P. Genera el conjunto $\text{odd}_{\text{j}-1}$ sumando los conjuntos $d_{j-1} + P\left(\text{even}_{\text{j}-1}\right)$. Escrito de forma explícita, esta operación se convierte en:

$$s_{j,2l+1} = d_{j-1,l} + s_{j,2l}, \text{ para } 0 \leq l < 2^j.$$

Ahora que los dos conjuntos $\text{even}_{\text{j}-1}$ y $\text{odd}_{\text{j}-1}$ han sido reconstruidos, se combinan mediante la operación de "fusión" en el conjunto s_j. Ésta es la transformada wavelet perezosa inversa, formalmente denotada por:

$$s_j = \text{Merge}\left(\text{odd}_{j-1}, \text{even}_{j-1}\right).$$

La cual traslada las medias y las diferencias en las ubicaciones par e impar de s_j, respectivamente, sin utilizar ningún espacio adicional. Las tres operaciones se resumen mediante:

$$\text{even}_{j-1} = U\left(\text{odd}_{j-1}\right); \quad \text{odd}_{j-1} += P\left(\text{even}_{j-1}\right); \quad s_j := \text{Merge}\left(\text{odd}_{j-1}, \text{even}_{j-1}\right);$$

La Figura 5.47b es un diagrama de conexiones de este proceso.

Los diagramas de conexiones muestran una de las características más importantes del esquema lifting, a saber, su inherente paralelismo. Dado un computador SIMD[4] (Single Instruction, Multiple Data) con 2^n unidades de procesamiento, cada unidad puede ser "responsable" de un valor de píxel. Tal ordenador ejecuta un único programa en el que cada instrucción es ejecutada en paralelo por todas

[4]Única instrucción, múltiples datos.

las unidades de procesamiento. Otra ventaja del lifting es la simplicidad de su transformada inversa. Es simplemente el código de la transformada directa, ejecutado hacia atrás. La principal ventaja del lifting es el hecho de que es fácil de extender. Se ha presentado aquí para la transformada de Haar, donde los promedios y las diferencias se calculan de una manera sencilla. Esto puede extenderse a los casos más complejos, donde las operaciones de predicción y actualización son más complejas.

5.11.1. La transformada wavelet lineal

La razón para extender el esquema lifting más allá de la transformada de Haar es que esta transformada no produce resultados de alta calidad, ya que utiliza una predicción tan simple. Recordemos que la operación "predicción" utiliza el conjunto par even_{j-1} para predecir el conjunto impar odd_{j-1}. Ésto proporciona una predicción precisa sólo en los (muy raros) casos donde los dos conjuntos son idénticos. Podemos decir que la transformada de Haar elimina la correlación de orden cero entre los píxeles utilizando un predictor de orden uno. La transformada de Haar también preserva el promedio de todos los píxeles de la imagen, y este promedio puede ser llamado *momento de orden de cero* de la imagen.

Puede lograrse una mejor compresión mediante transformaciones que usan mejores predictores, predictores que explotan las correlaciones entre los píxeles vecinos, y también preservan momentos de orden superior de la imagen. Las operaciones de predicción y actualización descritas aquí son de orden dos. Ésto implica que el predictor proporcionará una predicción exacta si los píxeles de la imagen varían linealmente, y la operación de actualización preservará los dos primeros momentos (de orden cero y de orden uno). El principio es fácil de describir. De nuevo nos concentramos en una fila s_j de valores de píxel. Un valor de numeración impar $s_{j,2l+1}$ es predicho como el promedio de sus dos vecinos inmediatos $s_{j,2l}$ y $s_{j,2l+2}$. Para ser capaz de reconstruir $s_{j,2l+1}$, tenemos que calcular un valor de detalle $d_{j,l}$ que ya no es una simple diferencia, sino que viene dado por:

$$d_{j,l} = s_{j,2l+1} - \frac{1}{2}\left(s_{j,2l} + s_{j,2l+2}\right).$$

(Esta notación asume que cada píxel tiene dos vecinos inmediatos. Ésto no es cierto para los píxeles localizados en los límites de la imagen, por lo que las reglas de borde tendrán que ser desarrolladas y justificadas.)

La Figura 5.48 ilustra el significado de los valores de detalle en este caso. La Figura 5.48a muestra los valores de nueve píxeles hipotéticos numerados de 0 a 8. La Figura 5.48b muestra segmentos rectos que unen los píxeles pares. Ésta es la razón para el nombre "transformación lineal". En la Figura 5.48c los píxeles impares son predichos por estos segmentos rectos, y la Figura 5.48d muestra (en negrita discontinua) la diferencia entre cada píxel de numeración impar y su predicción. La ecuación de una línea recta es $y = ax+b$, un polinomio de grado 1, por lo que podemos pensar en los valores de detalle como la cantidad en que los valores de píxel se desvían localmente de un polinomio de grado 1. Si el píxel x tiene un valor $ax + b$, todos los valores de detalle serían cero. Por ello, podemos decir que los valores de detalle capturan las altas frecuencias de la imagen.

La operación "actualización" reconstruye los promedios $s_{j-1,l}$, a partir de los promedios $s_{j,l}$ y de las diferencias $d_{j-1,l}$. En el caso de la transformada de Haar, esta operación se define mediante la Ecuación (5.13): $s_{j-1,l} = s_{j,2l} + d_{j-1,l}/2$. En el caso de la transformación lineal, la operación es algo más complicada. Derivamos la operación de actualización para la transformación lineal usando el requisito de que conserva el momento de orden cero de la imagen, i.e., la media de los promedios $s_{j,l}$ no debe depender de j. Tratamos una actualización de la forma

$$s_{j-1,l} = s_{j,2l} + A\left(d_{j-1,l-1} + d_{j-1,l}\right), \tag{5.14}$$

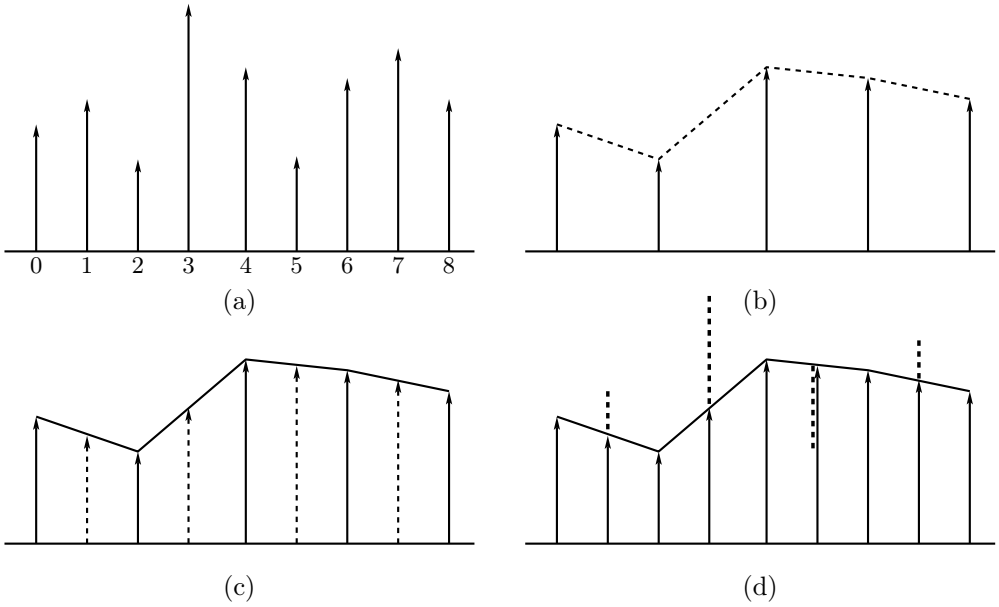


Figura 5.48: Predicción lineal.

donde A es un coeficiente desconocido. La suma de $s_{j-1,l}$ se convierte ahora en

$$\sum_l s_{j-1,l} = \sum_l s_{j,2l} + 2A \sum d_{j-1,l} = (1-2A) \sum_l s_{j,2l} + 2A \sum_l s_{j,2l+1},$$

por lo que la elección $A = 1/4$ produce como resultado:

$$\sum_{l=0}^{2^{j-1}-1} s_{j-1,l} = \left(1 - \frac{1}{2}\right) \sum_{l=0}^{2^j-1} s_{j,2l} + \frac{2}{4} \sum_{l=0}^{2^j-1} s_{j,2l+1} = \frac{1}{2} \sum_{l=0}^{2^j-1} s_{j,l}.$$

Comparando ésto con la Ecuación (5.15) se aprecia que el momento de orden cero de la imagen se preserva mediante la operación de actualización de la Ecuación (5.14). Una comprobación directa también verifica que

$$\sum_l l s_{j-1,l} = \frac{1}{2} \sum_l l s_{j,l},$$

lo que demuestra que esta operación de actualización también preserva el momento de primer orden de la imagen y por lo tanto es de orden 2.

La Ecuación (5.15), duplicada abajo, se deriva de la respuesta al Ejercicio 5.13:

$$\sum_{l=0}^{2^{j-1}-1} s_{j-1,l} = \sum_{l=0}^{2^{j-1}-1} (s_{j,2l} + d_{j-1,l}/2) = \frac{1}{2} \sum_{l=0}^{2^{j-1}-1} (s_{j,2l} + s_{j,2l+1}) = \frac{1}{2} \sum_{l=0}^{2^j-1} s_{j,l}. \tag{5.15}$$

La transformada lineal inversa reconstruye los valores promedio pares e impares mediante

$$s_{j,2l} = s_{j-1,l} - \frac{1}{4}\left(d_{j-1,l-1} + d_{j-1,l}\right), \quad \text{y} \quad s_{j,2l+1} = d_{j,l} + \frac{1}{2}\left(s_{j,2l} + s_{j,2l+2}\right),$$

respectivamente.

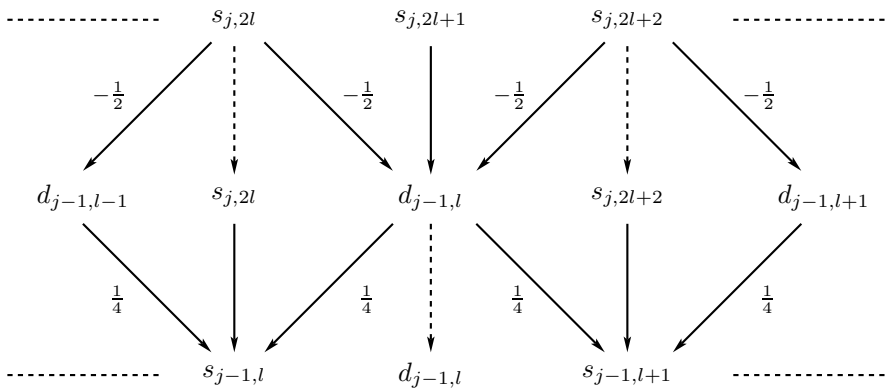


Figura 5.49: Resumen de la transformada wavelet lineal.

La Figura 5.49 resume la transformación lineal. La fila superior muestra los promedios pares e impares $s_{j,l}$. La fila central muestra cómo se calcula un valor detalle $d_{j-1,l}$ como la diferencia entre un promedio impar $s_{j,2l+1}$ y la mitad de la suma de sus dos vecinos pares $s_{j,2l}$ y $s_{j,2l+2}$. La fila inferior muestra cómo se calcula el siguiente conjunto s_{j-1} de promedios. Cada promedio $s_{j-1,l}$ en este conjunto es la suma de un promedio par $s_{j,2l}$ y un cuarto de los dos valores de detalle $d_{j-1,l-1}$ y $d_{j-1,l}$. La figura también ilustra la característica principal del esquema lifting, a saber, cómo se reemplazan los promedios pares $s_{j,2l}$ por el siguiente conjunto de promedios $s_{j-1,l}$ y cómo son reemplazados los promedios impares $s_{j,2l+1}$ por los valores de detalle $d_{j-1,l}$ (las flechas discontinuas indican los ítems que no se mueven). Todas estas operaciones se realizan in situ.

5.11.2. Subdivisión por interpolación

El método de subdivisión por interpolación comienza con un conjunto de píxeles s_0 donde el píxel $s_{0,k}$ (para $k = 0, 1, \ldots$) se almacena en la ubicación k de un arreglo (o array) a. Crea un conjunto s_1 (del doble del tamaño de s_0) de píxeles $s_{1,k}$ tal que los píxeles de numeración par $s_{1,2k}$ son simplemente los píxeles de numeración par $s_{0,k}$, y cada uno de los píxeles de numeración impar $s_{1,2k+1}$ se obtiene mediante la interpolación de algunos de los píxeles (impares y/o pares) del conjunto s_0. El nuevo contenido del array a es:

$$s_{1,0} = s_{0,0}, s_{1,1}, s_{1,2} = s_{0,1}, s_{1,3}, s_{1,4} = s_{0,2}, s_{1,5}, \ldots s_{1,2k} = s_{0,k}, s_{1,2k+1}, \ldots$$

Los elementos originales $s_{0,k}$ de s_0 se almacenan ahora en a in las posiciones $2k = k \cdot 2^1$. Decimos que el conjunto s_1 fue creado a partir de s_0 por un proceso de subdivisión (o refinamiento).

A continuación, el conjunto s_2 (del doble del tamaño de s_1) se crea de la misma manera a partir de s_1. Los píxeles de numeración par $s_{2,2k}$ son simplemente los píxeles de numeración par $s_{1,k}$, y cada uno de los píxeles de numeración impar $s_{1,2k+1}$ se obtiene mediante la interpolación de algunos de los píxeles (impares y/o pares) del conjunto s_1. El nuevo contenido del array a se convierte en:

$$s_{2,0} = s_{1,0} = s_{0,0}, s_{2,1}, s_{2,2} = s_{1,1}, s_{2,3}s_{2,4} = s_{1,2} = s_{0,1}, s_{2,5}, \ldots$$
$$s_{2,2k} = s_{1,k}, s_{2,2k+1}, \ldots, s_{2,4k} = s_{1,2k} = s_{0,k}, s_{2,4k+1}, \ldots$$

Los elementos originales $s_{0,k}$ de s_0 se almacenan ahora en a en las posiciones $4k = k \cdot 2^2$.

En un paso de subdivisión general, se usa un conjunto s_j de valores de píxel $s_{j,k}$ para construir un nuevo conjunto s_{j+1}, el doble de grande. Los píxeles de numeración par $s_{j+1,2k}$ son simplemente los píxeles de numeración par $s_{j,k}$, y cada uno de los píxeles de numeración impar $s_{j+1,2k+1}$ se obtiene

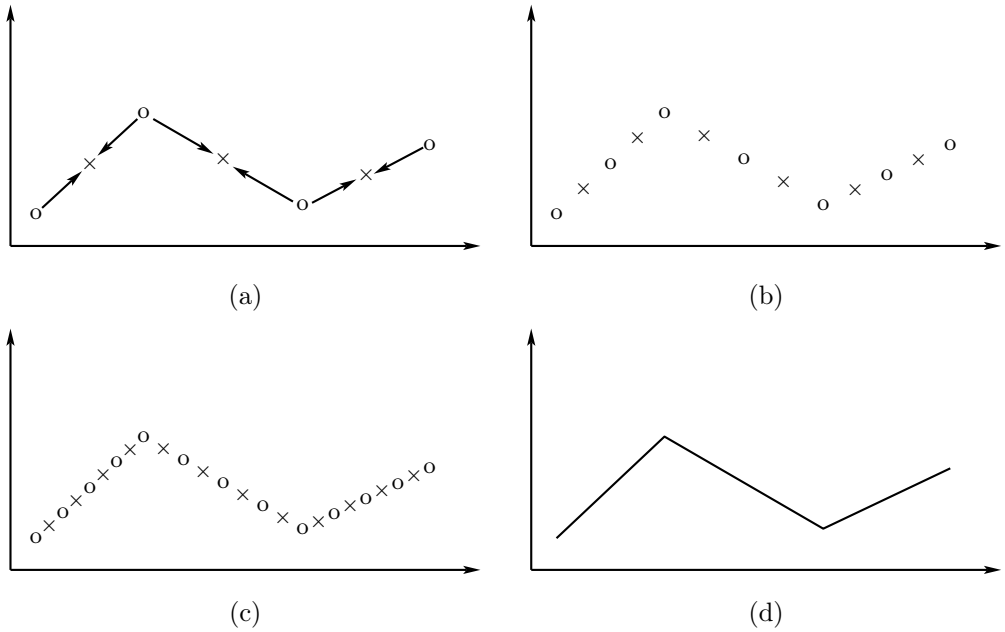


Figura 5.50: Un ejemplo de una subdivisión lineal.

mediante la interpolación de algunos de los píxeles (impares y/o pares) del conjunto s_j. Los elementos originales $s_{0,k}$ de s_0 son ahora almacenados en a en las posiciones $k \cdot 2^j$.

Empleamos la interpolación lineal para ilustrar este proceso de refinamiento. Cada uno de los píxeles de numeración impar $s_{1,2k+1}$ se calcula como la media de los dos píxeles $s_{0,k}$ y $s_{0,k+1}$ del conjunto s_0. En general, obtenemos:

$$s_{j+1,2k} = s_{j,k}, \quad s_{j+1,2k+1} = \frac{1}{2}\left(s_{j,k} + s_{j,k+1}\right). \tag{5.16}$$

La Figura 5.50a–d muestra varios pasos de este proceso, comenzando con un conjunto s_0 de cuatro píxeles (es útil visualizar cada píxel como un punto bidimensional). Es obvio que el valores de los píxeles convergen a la *línea poligonal* que pasa a través de los cuatro puntos originales.

Dada una imagen donde los valores de los píxeles suben y bajan linealmente (en una línea poligonal), podemos comprimirla seleccionando los píxeles en las esquinas de la línea poligonal (i.e., aquellos píxeles donde el valor cambia de dirección) y escribiéndolos en el archivo comprimido. La imagen podría ser perfectamente reconstruida a cualquier resolución utilizando la subdivisión lineal para computar tantos píxeles como fuera necesario entre los píxeles de esquina. La mayoría de las imágenes, sin embargo, presentan un comportamiento más complejo en los valores de píxel, por lo que requieren una interpolación más compleja.

Vamos ahora a demostrar cómo extender la subdivisión lineal a una *subdivisión polinómica*. En vez de calcular un píxel de numeración impar $s_{j+1,2k+1}$ como el promedio de sus dos vecinos inmediatos del nivel j, $s_{j,k}$ y $s_{j,k+1}$, se calcula como la suma ponderada de sus **cuatro** vecinos inmediatos del nivel j, $s_{j,k-1}$, $s_{j,k}$, $s_{j,k+1}$, y $s_{j,k+2}$. Es obvio que debe asignarse más peso a los dos vecinos más cercanos, $s_{j,k}$ y $s_{j,k+1}$, que a los dos vecinos extremos, $s_{j,k-1}$, y $s_{j,k+2}$; pero ¿Cuáles deben ser los pesos? La respuesta la proporciona la Ecuación (4.44) (Sección 4.21.4), que demuestra que el polinomio $\mathbf{P}(t)$ de

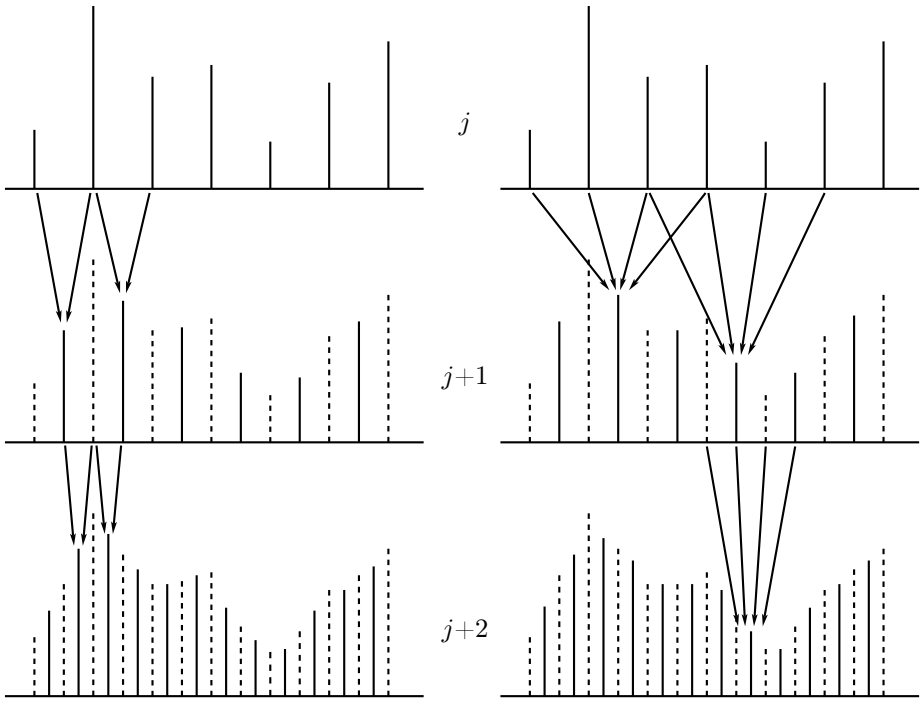


Figura 5.51: Subdivisiones lineal y cúbica.

grado 3 (cúbico) que interpola cuatro puntos arbitrarios $\mathbf{P}_1$, $\mathbf{P}_2$, $\mathbf{P}_3$, y $\mathbf{P}_4$ viene dado por:

$$\mathbf{P}(t) = \left(t^3, t^2, t, 1\right) \begin{pmatrix} -4{,}5 & 13{,}5 & -13{,}5 & 4{,}5 \\ 9{,}0 & -22{,}5 & 18 & -4{,}5 \\ -5{,}5 & 9{,}0 & -4{,}5 & 1{,}0 \\ 1{,}0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} \mathbf{P}_1 \\ \mathbf{P}_2 \\ \mathbf{P}_3 \\ \mathbf{P}_4 \end{pmatrix}. \tag{5.17}$$

Vamos a colocar el nuevo píxel de numeración impar $s_{j+1,2k+1}$ en el medio del grupo de sus cuatro vecinos del nivel j, por lo que tiene sentido asignarle el valor de interpolación del polinomio en la mitad de su intervalo, i.e., $\mathbf{P}(0{,}5)$. Calculado a partir de la Ecuación (5.17), este valor es:

$$\mathbf{P}(0{,}5) = -0{,}0625\mathbf{P}_1 + 0{,}5625\mathbf{P}_2 + 0{,}5625\mathbf{P}_3 - 0{,}0625\mathbf{P}_4.$$

(Observe que los cuatro pesos suman uno. Éste es un ejemplo de funciones baricéntricas. Véase también, el Ejercicio 4.45 para una explicación de los pesos negativos.)

La regla de subdivisión en este caso es [por analogía con la Ecuación (5.16)]

$$s_{j+1,2k} = s_{j,k}, \quad s_{j+1,2k+1} = \mathbf{P}_{3,j,k-1}(0{,}5), \tag{5.18}$$

donde la notación $\mathbf{P}_{3,j,k-1}(t)$ indica el polinomio de interpolación de grado 3 para el grupo de cuatro píxeles del nivel j, que comience en $s_{j,k-1}$. Definimos el *orden* de esta subdivisión como 4 (el número de píxeles interpolados). Puesto que la subdivisión lineal interpola dos píxeles, su orden es 2. La Figura 5.51 muestra tres niveles de píxeles generados mediante subdivisiones lineales (5.51a) y cúbicas (5.51b).

Ahora es obvio cómo la subdivisión por interpolación puede extenderse a órdenes superiores. Seleccionar un entero par n, derivar el polinomio de grado $(n-1)$, $\mathbf{P}_{n-1,j,k-(n/2-1)}(t)$ que interpolará los n píxeles del nivel j

$$s_{j,k-(n/2-1)}, s_{j,k-n/2}, s_{j,k-n/2+1}, \ldots s_{j,k}, s_{j,k+1}, \ldots, s_{j,k+n/2},$$

calcular el punto medio $\mathbf{P}_{n-1,j,k-(n/2-1)}(0{,}5)$, y generar los píxeles del nivel $(j+1)$ de acuerdo con

$$s_{j+1,2k} = s_{j,k}, \quad s_{j+1,2k+1} = \mathbf{P}_{n-1,j,k-(n/2-1)}(0{,}5), \tag{5.19}$$

◇ **Ejercicio 5.14 (sol. en pág. 1101):** Calcúlese el punto medio $\mathbf{P}(0{,}5)$ del polinomio de interpolación de grado 5 para seis puntos $\mathbf{P}_1$ a $\mathbf{P}_6$ como una función de los puntos.

5.11.3. Funciones de escalado

Las funciones de escalado $\phi_{j,k}(x)$ han sido mencionadas en la Sección 5.6, en relación con la transformada de Haar. Aquí mostramos cómo se construyen para una subdivisión por interpolación. Cada coeficiente $s_{j,k}$ calculado en el nivel j tiene una función de escalado $\phi_{j,k}(x)$ asociada con él, lo que se define como sigue: Seleccionar los valores n, j, y k. Establecer el píxel $s_{j,k}$ a 1 y todos los demás $s_{j,i}$ a 0 (ésto se puede expresar como $s_{j,k} = \delta_{0,k}$). Utilizar un esquema de subdivisión (basado en n puntos) para calcular los niveles $(j+1)$, $(j+2)$, etc.. Cada nivel tiene el doble del número de píxeles de su predecesor. En el límite, nos encontramos con un número infinito de píxeles. Podemos ver estos píxeles como números reales y definir el rango de la función de escalado $\phi_{j,k}(x)$ con estos números. Cada par de valores j y k define una función de escalado $\phi_{j,k}(x)$ diferente, pero intuitivamente podemos ver que las funciones de escalado dependen de j y k de manera simple. La forma de $\phi_{j,k}(x)$ no depende de k, ya que calculamos ϕ estableciendo $s_{j,k} = 1$ y todos los demás $s_{j,i} = 0$. Por consiguiente, la función $\phi_{j,8}(x)$ es una copia desplazada de $\phi_{j,7}(x)$, una copia desplazada dos veces de $\phi_{j,6}(x)$, etc.. En general, se puede escribir $\phi_{j,k}(x) = \phi_{j,0}(x-k)$. Para comprender la dependencia de $\phi_{j,k}(x)$ de j, el lector debe recordar la siguiente frase (de la página 605):

> "Los elementos originales $s_{0,k}$ de s_0 se almacenan ahora en a in las posiciones $k \cdot 2^j$."

Ésto implica que si seleccionamos un valor pequeño para j, terminamos con una función de escalado amplia $\phi_{j,k}(x)$. En general, tenemos:

$$\phi_{j,k}(x) = \phi_{0,0}\left(2^j x - k\right) \stackrel{\text{def}}{=} \phi\left(2^j x - k\right),$$

lo que implica que todas las funciones de escalado para valores diferentes de j y k son traslaciones y dilataciones (escalados) de $\phi(x)$, la solución fundamental del proceso de subdivisión. $\phi(x)$ se muestra en la Figura 5.52 para $n = 2, 4, 6$, y 8.

Las principales propiedades de $\phi(x)$ son el soporte compacto y la suavidad (que puede ser distinto de cero sólo dentro del intervalo $[-(n-1), n-1]$), pero también satisface una *relación de refinamiento* de la forma:

$$\phi(x) = \sum_{l=-n}^{n} h_l \phi(2x - l),$$

donde los h_l se llaman *coeficientes de filtro*. Un cambio de variables nos permite escribir la misma relación de refinamiento en la forma:

$$\phi_{j,k}(x) = \sum_{l} h_{l-2k} \phi_{j+1,l}(x).$$

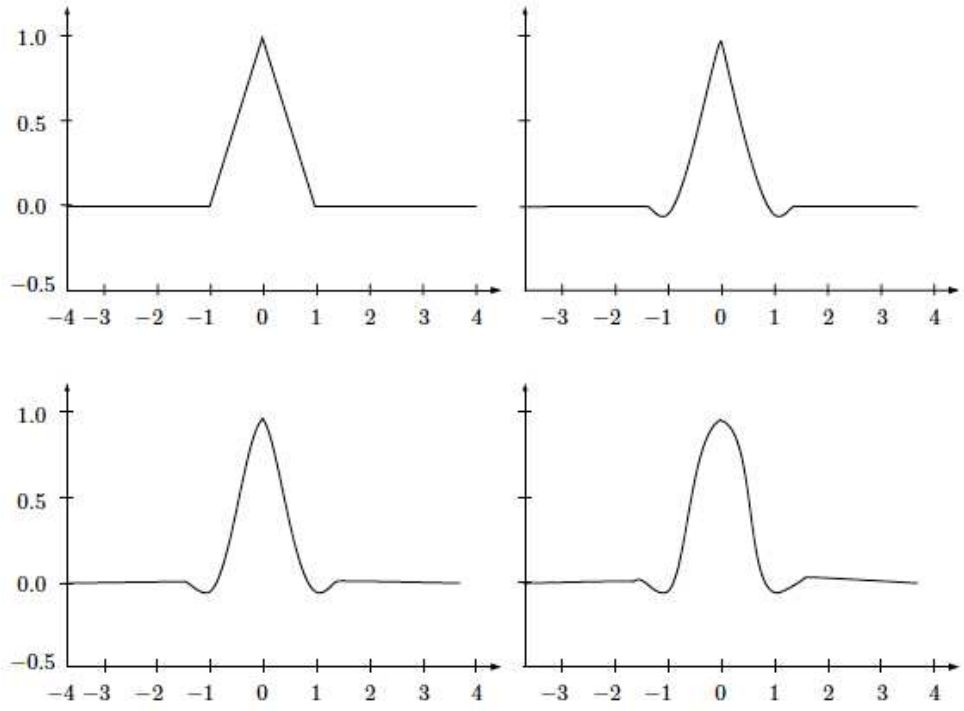


Figura 5.52: Funciones de escalado $\phi_{j,k}(x)$ para $n = 2, 4, 6$, y 8.

Los coeficientes de filtro de numeración impar son los coeficientes del polinomio de interpolación en su punto medio $\mathbf{P}_{n-1,j,k-(n/2-1)}(0{,}5)$. Los coeficientes de numeración par son cero, excepto para $l = 0$ (esta propiedad puede expresarse como $h_{2l} = \delta_{0,l}$). La expresión general para h_k es:

$$h_k = \begin{cases} k \text{ par}, & \delta_{k,0}, \\ k \text{ impar}, & (-1)^{d+k} \frac{\prod_{i=0}^{2d-1}(i-d+1/2)}{(k+1/2)(d+k)!(d-k-1)!} \end{cases}$$

donde $d = n/2$. Para la subdivisión lineal, los coeficientes de filtro son $(1/2, 1, 1/2)$. Para la interpolación cúbica los siete coeficientes de filtro son $(-1/16, 0, 9/16, 1, 9/16, 0, -1/16)$. Los coeficientes de filtro son útiles, ya que permiten que escribamos la subdivisión por interpolación en la forma:

$$s_{j+1,l} = \sum_k h_{l-2k} s_{j,k}.$$

5.12. La IWT

La DWT es sencilla pero tiene un inconveniente importante, a saber, utiliza un filtro de coeficientes no entero, lo que produce coeficientes de transformada no enteros. Existen varias formas de modificar la DWT básica de manera que produzca coeficientes de transformada enteros. Esta sección describe una sencilla *transformada wavelet entera* (*integer wavelet transform* o IWT) que puede usarse para descomponer una imagen en cualquiera de las formas descritas en la Sección 5.10. La transformación es reversible, i.e., la imagen puede ser totalmente reconstruida a partir de los coeficientes (enteros) de la transformada. Esta IWT puede utilizarse para comprimir la imagen, ya sea con pérdidas (mediante

la cuantificación de los coeficientes de la transformada), ya sea sin pérdidas (mediante la codificación de entropía de los coeficientes de la transformada).

El simple principio de esta transformada es ilustrado aquí para el caso unidimensional. Dado un vector de datos de N enteros x_i, donde $i = 0, 1, \ldots N-1$, definimos $k = {}^{N}/_{2}$ y calculamos la transformada del vector y_i calculando los componentes pares e impares de y por separado. En primer lugar, discutimos el caso en que N es par. Los ${}^{N}/_{2}$ componentes impares y_{2i+1} (donde $i = 0, 1, \ldots, k-1$) se calculan como la diferencia de los x_i's. Ellos se convierten los coeficientes de detalle (alta frecuencia) de la transformada. Cada uno de los componentes pares y_{2i} (donde i varía en el mismo intervalo $[0, k-1]$) es calculado como una media ponderada de cinco ítems de datos x_i. Estos ${}^{N}/_{2}$ números se convierten en los coeficientes de baja frecuencia de la transformada, y normalmente se transforman de nuevo, en ${}^{N}/_{4}$ coeficientes de baja frecuencia y ${}^{N}/_{4}$ de alta frecuencia.

La regla básica para los coeficientes de la transformada impares es

$$y_{2i+1} = -\frac{1}{2}x_{2i} + x_{2i+1} - \frac{1}{2}x_{2i+2},$$

excepto el último coeficiente, donde $i = k-1$, que se calcula como la simple diferencia $y_{2k-1} = x_{2k-1} - x_{2k-2}$. Ésto puede resumirse como:

$$y_{2i+1} = \begin{cases} x_{2i+1} - (x_{2i} + x_{2i+2})/2, & \text{para } i = 0, 1, \ldots, k-2, \\ x_{2i+1} - x_{2i}, & \text{para } i = k-1. \end{cases} \tag{5.20}$$

Los coeficientes pares de la transformada se calculan como el promedio ponderado

$$y_{2i} = -\frac{1}{8}x_{2i-2} + \frac{1}{4}x_{2i-1} + \frac{3}{4}x_{2i} + \frac{1}{4}x_{2i+1} - \frac{1}{8}x_{2i+2},$$

excepto el primer coeficiente, donde $i = 0$, que se calcula como

$$y_0 = \frac{3}{4}x_0 + \frac{1}{2}x_1 - \frac{1}{4}x_2.$$

En la práctica, este cálculo se efectúa calculando cada coeficiente par y_{2i} en términos de x_{2i} y los dos coeficientes impares y_{2i-1} e y_{2i+1}. Ésto se puede resumir como:

$$y_{2i} = \begin{cases} x_{2i} + y_{2i+1}/2, & \text{para } i = 0, \\ x_{2i} + (y_{2i-1} + y_{2i+1})/4, & \text{para } i = 1, 2, \ldots, k-1. \end{cases} \tag{5.21}$$

La transformada inversa es fácil de determinar. Utiliza los coeficientes y_i de la transformada para calcular ítems de datos z_i que son idénticos a los originales x_i. Primero calcula los elementos pares

$$z_{2i} = \begin{cases} y_{2i} - y_{2i+1}/2, & \text{para } i = 0, \\ y_{2i} - (y_{2i-1} + y_{2i+1})/4, & \text{para } i = 1, 2, \ldots, k-1, \end{cases} \tag{5.22}$$

y luego los elementos impares

$$z_{2i+1} = \begin{cases} y_{2i+1} + (z_{2i} + z_{2i+2})/2, & \text{para } i = 0, 1, \ldots, k-2, \\ y_{2i+1} + z_{2i}, & \text{para } i = k-1. \end{cases} \tag{5.23}$$

Ahora viene la parte interesante. Los coeficientes de la transformada calculados mediante las Ecuaciones (5.20) y (5.21) generalmente no son enteros, debido a las divisiones por 2 y por 4. Lo mismo es cierto para los ítems de datos reconstruidos a partir de las Ecuaciones (5.22) y (5.23). La principal

caracteristica de la IWT particular descrita aquí es el uso del *truncamiento*. El truncamiento, denotado por los símbolos "de pie" $\lfloor$ y $\rfloor$, se utiliza para producir coeficientes de transformada enteros y_i y también ítems de datos reconstruidos enteros z_i. Las Ecuaciones (5.20) a (5.23) son modificadas a:

$$\begin{aligned}
y_{2i+1} &= \begin{cases} x_{2i+1} - \lfloor (x_{2i} + x_{2i+2})/2 \rfloor, & \text{para } i = 0, 1, \ldots, k-2, \\ x_{2i+1} - x_{2i}, & \text{para } i = k-1. \end{cases} \\
y_{2i} &= \begin{cases} x_{2i} + \lfloor y_{2i+1}/2 \rfloor, & \text{para } i = 0, \\ x_{2i} + \lfloor (y_{2i-1} + y_{2i+1})/4 \rfloor, & \text{para } i = 1, 2, \ldots, k-1. \end{cases} \\
z_{2i} &= \begin{cases} y_{2i} - \lfloor y_{2i+1}/2 \rfloor, & \text{para } i = 0, \\ y_{2i} - \lfloor (y_{2i-1} + y_{2i+1})/4 \rfloor, & \text{para } i = 1, 2, \ldots, k-1, \end{cases} \\
z_{2i+1} &= \begin{cases} y_{2i+1} + \lfloor (z_{2i} + z_{2i+2})/2 \rfloor, & \text{para } i = 0, 1, \ldots, k-2, \\ y_{2i+1} + z_{2i}, & \text{para } i = k-1. \end{cases}
\end{aligned} \tag{5.24}$$

Debido al truncamiento, se pierde alguna información cuando se calculan los y_i. Sin embargo, el truncamiento también se utiliza en el cálculo de z_i, lo que restaura la información perdida. Por consiguiente, la Ecuación (5.24) es una auténtica IWT directa e inversa que reconstruye los ítems de datos originales exactamente.

◇ **Ejercicio 5.15 (sol. en pág. 1101):** Dado el vector de datos $x = (112, 97, 85, 99, 144, 120, 77, 80)$, utilícese la Ecuación (5.24) para calcular sus transformadas wavelet enteras, directa e inversa.

Los mismos conceptos pueden aplicarse al caso en que el número N de ítems de datos es impar. Primero definimos k para $N = 2k + 1$, después definimos las transformadas enteras directa e inversa mediante:

$$\begin{aligned}
y_{2i+1} &= x_{2i+1} - \lfloor (x_{2i} + x_{2i+2})/2 \rfloor, \text{ para } i = 0, 1, \ldots, k-1, \\
y_{2i} &= \begin{cases} x_{2i} + \lfloor y_{2i+1}/2 \rfloor, & \text{para } i = 0, \\ x_{2i} + \lfloor (y_{2i-1} + y_{2i+1})/4 \rfloor, & \text{para } i = 1, 2, \ldots, k-1, \\ x_{2i} + \lfloor y_{2i-1}/2 \rfloor, & \text{para } i = k. \end{cases} \\
z_{2i} &= \begin{cases} y_{2i} - \lfloor y_{2i+1}/2 \rfloor, & \text{para } i = 0, \\ y_{2i} - \lfloor (y_{2i-1} + y_{2i+1})/4 \rfloor, & \text{para } i = 1, 2, \ldots, k-1, \\ y_{2i} - \lfloor y_{2i-1}/2 \rfloor, & \text{para } i = k, \end{cases} \\
z_{2i+1} &= y_{2i+1} + \lfloor (z_{2i} + z_{2i+2})/2 \rfloor, \text{ para } i = 0, 1, \ldots, k-1.
\end{aligned}$$

Tenga en cuenta que la IWT produce un vector y_i donde los coeficientes de detalle y los promedios ponderados están intercalados. El algoritmo debe ser modificado para colocar las medias en la primera mitad de y y los detalles en la segunda mitad.

La extensión de esta transformada para el caso bidimensional es obvio. La IWT se aplica a las filas y a las columnas de la imagen utilizando los métodos de descomposición de imagen descritos en la Sección 5.10.

5.13. La pirámide Laplaciana

La característica principal del método de la pirámide Laplaciana [Burt y Adelson 83] es la compresión progresiva. El decodificador introduce el stream comprimido sección por sección, y cada sección mejora la apariencia en pantalla de la imagen procesada hasta el momento. El método utiliza tanto

la predicción como técnicas de transformación, pero sus cálculos son simples y locales (i.e., no hay necesidad de examinar o utilizar valores lejanos al píxel actual). El nombre "Laplaciana" procede del campo de la mejora de la imagen, donde se utiliza para indicar operaciones similares a las usadas aquí. Comenzamos con una descripción general del método.

Denotamos por $g_0(i,j)$ a la imagen original. Se calcula una nueva imagen reducida g_1 a partir de g_0 de forma que cada píxel de g_1 sea una suma ponderada de un grupo de 5×5 píxeles de g_0. La imagen g_1 es calculada [véase la Ecuación (5.25)] de tal manera que tiene la mitad del número de filas y la mitad del número de columnas de g_0, por lo que es de un cuarto del tamaño de g_0. Es una versión borrosa (o de paso bajo filtrada) de g_0. El siguiente paso es expandir g_1 a una imagen $g_{1,1}$, del tamaño de g_0 mediante la interpolación de valores de píxel [Ecuación (5.26)]. Una imagen de diferencia (también llamada imagen de error) L_0 es calculada como la diferencia $g_0 - g_{1,1}$, y se convierte en el nivel inferior de la pirámide Laplaciana. La imagen original g_0 puede ser reconstruida a partir de L_0 y $g_{1,1}$ y también desde L_0 y g_1. Puesto que g_1 es más pequeña que $g_{1,1}$, tiene sentido escribir L_0 y g_1 en el stream comprimido. El tamaño de L_0 es igual al de g_0, y el tamaño de g_1 es $1/4$ de éste, por lo que parece que hemos generado expansión, pero de hecho, se consigue compresión, porque los valores de error de L_0 están altamente correlacionados, y por tanto son pequeños (y por consiguiente tienen una varianza pequeña y una baja entropía) y pueden representarse con menos bits que los píxeles originales en g_0.

Con el fin de conseguir una representación progresiva de la imagen, sólo se escribe L_0 en la salida, y se repite el proceso en g_1. Se calcula una nueva reducción de la imagen g_2 a partir de g_1 (el tamaño de g_2 es $1/6$ del de g_0). Es expandida a una imagen $g_{2,1}$, y se calcula una nueva imagen de diferencia como la diferencia $g_1 - g_{2,1}$ y se convierte en el siguiente nivel, por encima de L_0, de la pirámide Laplaciana. El resultado final es una secuencia $L_0, L_1, \ldots, L_{k-1}, L_k$, donde los k primeros ítems son imágenes de diferencia, y el último —L_k— es simplemente la (muy pequeña) imagen reducida g_k. Estos ítems constituyen la pirámide Laplaciana. Ellos son escritos en el stream comprimido en orden inverso, por lo que el decodificador introduce primero L_k, lo utiliza para mostrar una pequeña imagen borrosa, introduce L_{k-1}, reconstruye y muestra g_{k-1} (que es cuatro veces más grande), y reitera hasta que reconstruye y muestra g_0. Este proceso puede resumirse en:

$$\begin{aligned} g_k &= L_k, \\ g_i &= L_i + \text{Expandir}\,(g_{i+1}) = L_i + g_{i+1,1}, \quad \text{para } i = k-1, k-2, \ldots, 2, 1, 0. \end{aligned}$$

El usuario ve imágenes borrosas pequeñas, que crecen y se vuelven más nítidas. El decodificador puede ser modificado para expandir cada imagen intermedia g_i (antes de ser mostrada) varias veces, mediante la interpolación de valores de píxel, hasta obtener el tamaño de la imagen original g_0. Así, el usuario ve una imagen que progresa desde muy borrosa a nítida, mientras permanece del mismo tamaño. Por ejemplo, la imagen g_3 —que es $1/64$ del tamaño de g_0— puede ser llevada a este tamaño expandiendo tres veces, generando la cadena:

$$g_{3,3} = \text{Expandir}\,(g_{3,2}) = \text{Expandir}\,(g_{3,1}) = \text{Expandir}\,(g_3)\,.$$

Para que todas las imágenes intermedias g_i y L_i tengan dimensiones bien definidas, la imagen original g_0 debe tener $R = M_R 2^M + 1$ filas y $C = M_C 2^M + 1$ columnas, donde M_R, M_C, y M son enteros. La selección de, por ejemplo, $M_R = M_C$ resulta en una imagen cuadrada. La imagen g_1 tiene dimensiones $\left(M_R 2^{M-1} + 1\right) \times \left(M_C 2^{M-1} + 1\right)$, y la imagen g_p tiene dimensiones $\left(M_R 2^{M-p} + 1\right) \times \left(M_C 2^{M-p} + 1\right)$. Un ejemplo es $M_R = M_C = 1$ y $M = 8$. Las dimensiones de la imagen original son $\left(2^8 + 1\right) \times \left(2^8 + 1\right) = 257 \times 257$ y las imágenes reducidas g_1 a g_5 tienen dimensiones $\left(2^7 + 1\right) \times \left(2^7 + 1\right) = 129 \times 129$, 65×65, 33×33, 17×17, y 9×9, respectivamente.

◇ **Ejercicio 5.16 (sol. en pág. 1102):** Calcúlense las dimensiones de las seis primeras imágenes, de g_0 a g_5 para el caso $M_C = 3$, $M_R = 4$, y $M = 5$.

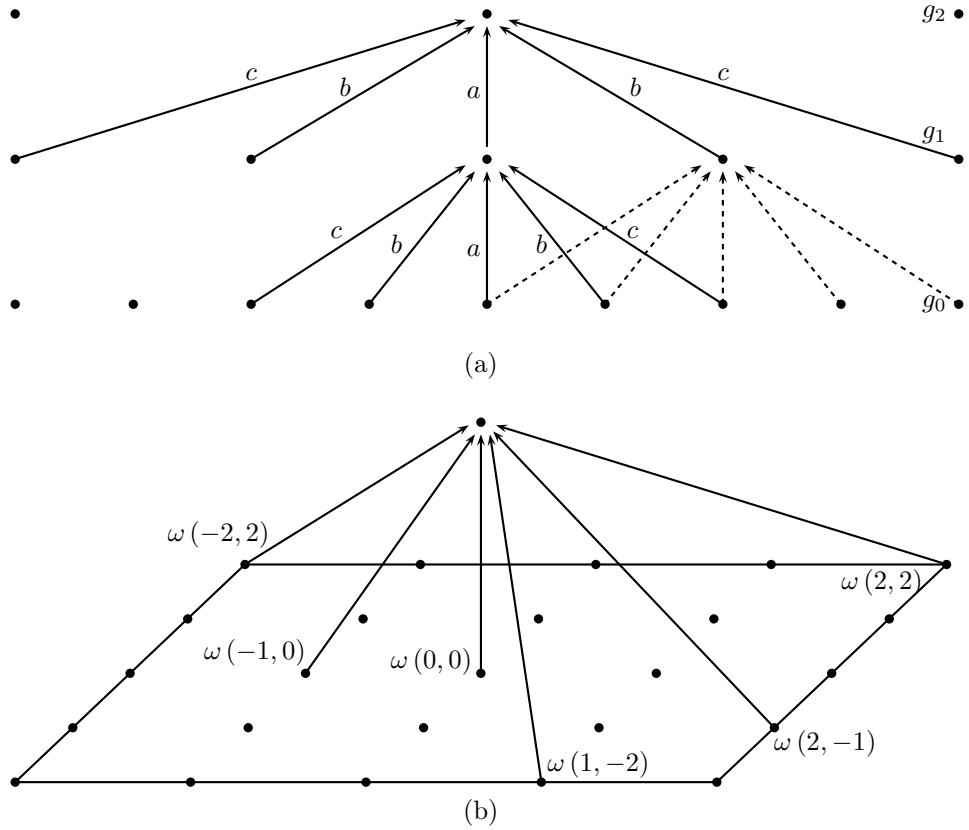


Figura 5.53: Ilustración de la reducción.

Pasamos ahora a los detalles de la reducción y la expansión de imágenes. La reducción de una imagen g_{p-1} a una imagen g_p de dimensiones $R_p \times C_p$ (donde $R_p = M_R 2^{M-p}+1$, y $C_p = M_C 2^{M-p}+1$) se efectúa mediante:

$$g_p(i,j) = \sum_{m=-2}^{2} \sum_{n=-2}^{2} \omega(m,n)\, g_{p-1}(2i+m, 2j+n), \tag{5.25}$$

donde $i = 0, 1, \ldots, C_p - 1$, $j = 0, 1, \ldots, R_p - 1$, y p (el nivel) varía de 1 a $k-1$. Cada píxel de g_p es una suma ponderada de 5×5 píxeles de g_{p-1} con pesos $\omega(m,n)$, que son los mismos para todos los p niveles. La Figura 5.53a ilustra este proceso en una dimensión. Muestra cómo cada píxel en un nivel superior de la pirámide se genera como una suma ponderada de cinco píxeles del nivel inferior, y cómo cada nivel tiene (aproximadamente) la mitad el número de píxeles de su predecesor. La Figura 5.53b es en dos dimensiones. Muestra cómo un píxel en un nivel alto se obtiene a partir de 25 píxeles situados en un nivel inferior. Algunos de los pesos también se muestran. Observe que en este caso, cada nivel tiene aproximadamente $1/4$ del número de píxeles de su predecesor.

Los pesos $\omega(m,n)$ (también llamados núcleo de generación) se determinan primero separando cada uno en la forma $\omega(m,n) = \hat{\omega}(m)\,\hat{\omega}(n)$, donde las funciones $\hat{\omega}(m)$ deberían ser normalizadas, i.e.,

$$\sum_{m=-2}^{2} \hat{\omega}(m) = 1,$$

	−0,05	0,25	0,6	0,25	−0,05
−0,05	0,0025	−0,0125	−0,03	−0,01	0,0025
0,25		0,0625	0,15	0,0625	−0,0125
0,6			0,36	0,15	−0,03
0,25				0,0625	−0,0125
−0,05					0,0025

Tabla 5.54: Valores de $\omega(m,n)$ para $a = 0{,}6$.

y simétricas, $\hat{\omega}(m) = \hat{\omega}(-m)$. Estas dos restricciones no son suficientes para determinar $\hat{\omega}(m)$, por lo que añadir una tercera, llamada *contribución de igualdad*, que exige que todos los píxeles de un nivel dado contribuyan con el mismo peso total ($= 1/4$) a los píxeles del siguiente nivel superior. Si establecemos $\hat{\omega}(0) = a$, $\hat{\omega}(-1) = \hat{\omega}(1) = b$, y $\hat{\omega}(-2) = \hat{\omega}(2) = c$, entonces las tres restricciones se satisfacen si:

$$\hat{\omega}(0) = a, \quad \hat{\omega}(-1) = \hat{\omega}(1) = 0{,}25, \quad \hat{\omega}(-2) = \hat{\omega}(2) = 0{,}25 - a/2.$$

(El lector debe comparar ésto con la discusión de los polinomios de interpolación de la Sección 4.21.4.)

La experiencia recomienda establecer $a = 0{,}6$, que produce (véanse los valores de $\omega(m,n)$ en Tabla 5.54)

$$\hat{\omega}(0) = 0{,}6, \quad \hat{\omega}(-1) = \hat{\omega}(1) = 0{,}25, \quad \hat{\omega}(-2) = \hat{\omega}(2) = 0{,}25 - 0{,}3 = -0{,}05.$$

Los mismos pesos usados en la reducción de imágenes también se utilizan en la expansión de las mismas. La expansión de una imagen g_p de dimensiones $R_p \times C_p$ (donde $R_p = M_R 2^{M-p} + 1$, y $C_p = M_C 2^{M-p} + 1$ a una imagen $g_{p,1}$ que tiene las mismas dimensiones que g_{p-1} (i.e., es cuatro veces más grande que g_p) se realiza mediante:

$$g_{p,1}(i,j) = 4 \sum_{m=-2}^{2} \sum_{n=-2}^{2} \omega(m,n)\, g_p\left(\frac{[i-m]}{2}, \frac{[j-n]}{2}\right), \tag{5.26}$$

donde $i = 0, 1, \ldots, C_p - 1$, $j = 0, 1, \ldots, R_p - 1$, y las sumas incluyen sólo los términos para los que tanto $(i-m)/2$ y $(j-n)/2$ son números enteros. A modo de ejemplo calculamos el píxel único:

$$g_{1,1}(4,5) = 4 \sum_{m=-2}^{2} \sum_{n=-2}^{2} \omega(m,n)\, g_1\left(\frac{[4-m]}{2}, \frac{[5-n]}{2}\right).$$

De los 25 términos de esta suma, sólo seis, a saber, aquellos con $m = -2, 0, 2$ y $n = -1, 1$ satisfacen las condiciones anteriores y son incluidos. Los seis términos corresponden a

$$(m,n) = (-2,-1), (-2,1), (0,-1), (0,1), (2,-1)(2,1),$$

y la suma es:

$$\begin{aligned} 4[&\omega(-2,-1)\, g_1(3,3) + \omega(-2,1)\, g_1(3,2) + \omega(0,-1)\, g_1(2,3) \\ &+ \omega(0,1)\, g_1(2,2) + \omega(2,-1)\, g_1(1,3) + \omega(2,1)\, g_1(1,2)]. \end{aligned}$$

Se puede conseguir una versión con pérdidas de la pirámide Laplaciana cuantificando los valores de cada imagen L_i antes de ser codificada y escrita en el stream comprimido.

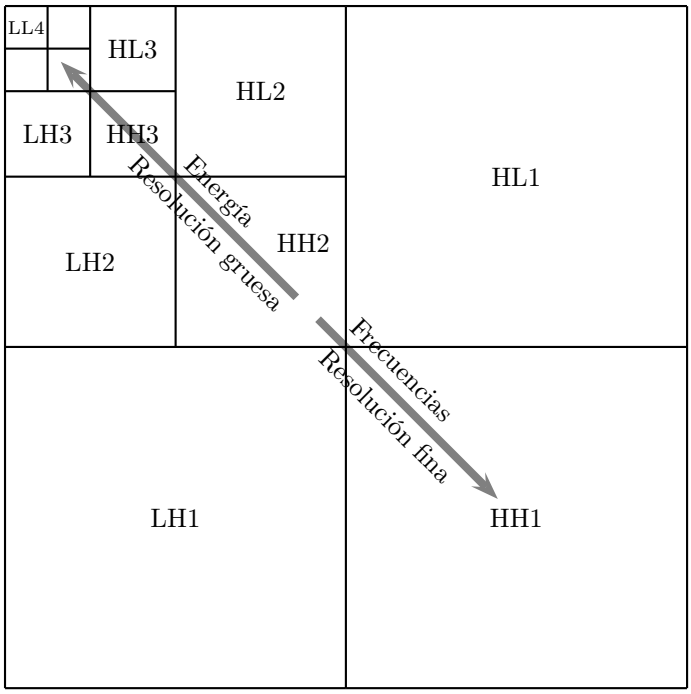


Figura 5.55: Subbandas y niveles en la descomposición wavelet.

5.14. SPIHT

La Sección 5.6 muestra cómo puede aplicarse la transformada de Haar varias veces a una imagen, creando regiones (o subbandas) de promedios y detalles. La transformada de Haar es sencilla, y la mejor compresión puede conseguirse mediante otros filtros wavelet. Al parecer, distintos filtros wavelet producen resultados diferentes dependiendo del tipo de imagen, pero actualmente no está claro qué filtro es el mejor para un tipo de imagen dado. Independientemente del filtro particular utilizado, la imagen es descompuesta en subbandas, de tal manera que las subbandas inferiores corresponden a las frecuencias más altas de la imagen (son los niveles de paso alto) y las subbandas superiores corresponden a las frecuencias más bajas de la imagen (niveles de paso bajo), donde se concentra la mayor parte de la energía de la imagen (Figura 5.55). Por ello, podemos esperar que los coeficientes de detalle sean más pequeños a medida que no movemos de los niveles altos a los bajos. Además, existen similitudes espaciales entre las subbandas (Figura 5.17b). Una parte de la imagen, tal como un borde, ocupa la misma posición espacial en cada subbanda. Estas características de la descomposición wavelet son explotadas por el método SPIHT (*Set Partitioning In Hierarchical Trees* o conjunto de particionamiento en árboles jerárquicos) [Said y Pearlman 96].

SPIHT fue diseñado para la transmisión progresiva óptima, así como para la compresión. Una de las características importantes de SPIHT (quizás una característica única) es que en cualquier punto durante la decodificación de una imagen, la calidad de la imagen visualizada es la mejor que se puede lograr para el número de bits introducidos por el decodificador hasta ese momento.

Otra característica importante de SPIHT es su uso de la codificación embebida. Esta característica se define como sigue: Si un codificador (de codificación embebida) produce dos archivos, uno grande de tamaño M y uno pequeño de tamaño m, entonces el archivo más pequeño es idéntico a los primeros m bits del archivo más grande.

El siguiente ejemplo ilustra bien el significado de esta definición. Supongamos que tres usuarios

esperan que se les envíe una cierta imagen comprimida, pero necesitan diferentes calidades de imagen. El primero necesita la calidad contenida en un archivo de 10 Kb. Las calidades de imagen requeridas por los usuarios segundo y tercero son las contenidas en archivos de tamaños 20 Kb y 50 Kb, respectivamente. La mayoría de los métodos de compresión con pérdida de imagen tendría que comprimir la misma imagen tres veces, en diferentes calidades, para generar tres archivos con los tamaños adecuados. SPIHT, en cambio, produce un archivo, y después tres fragmentos de longitudes 10 Kb, 20 Kb y 50 Kb, todos empezando por el principio de ese archivo —pueden ser enviados a los tres usuarios, satisfaciendo de esta manera sus necesidades—.

Comenzamos con una descripción general de SPIHT. Denotamos los píxeles de la imagen original $\mathbf{p}$ por $p_{i,j}$. Cualquier conjunto $\mathbf{T}$ de filtros wavelet puede ser utilizado para transformar los píxeles a coeficientes wavelet (o coeficientes de la transformada) $c_{i,j}$. Estos coeficientes constituyen la imagen transformada $\mathbf{c}$. La transformación se denota por $\mathbf{c} = \mathbf{T}(\mathbf{p})$. En un método de transmisión progresivo, el decodificador comienza estableciendo la reconstrucción de la imagen $\hat{\mathbf{c}}$ a cero. A continuación, introduce los coeficientes de la transformada (codificados), los decodifica, y los usa para generar una reconstrucción de la imagen mejorada $\hat{\mathbf{c}}$, la cual, a su vez, se utiliza para producir una mejor imagen $\hat{\mathbf{p}}$. Podemos resumir esta operación mediante $\hat{\mathbf{p}} = \mathbf{T}^{-1}(\hat{\mathbf{c}})$.

El objetivo principal de la transmisión progresiva es transmitir la información más importante de la imagen en primer lugar. Ésta es la información que resulta en la mayor reducción de la distorsión (la diferencia entre las imágenes original y reconstruida). SPIHT utiliza como medida de distorsión el error cuadrático medio (MSE) [Ecuación (4.2)]

$$D_{\text{mse}}(\mathbf{p}-\hat{\mathbf{p}}) = \frac{|\mathbf{p}-\hat{\mathbf{p}}|^2}{N} = \frac{1}{N}\sum_i\sum_j (p_{i,j}-\hat{p}_{i,j})^2,$$

donde N es el número total de píxeles. Una consideración importante en el diseño de SPIHT es el hecho de que esta medida es invariante para la transformada wavelet, una característica que nos permite escribir:

$$D_{\text{mse}}(\mathbf{p}-\hat{\mathbf{p}}) = D_{\text{mse}}(\mathbf{c}-\hat{\mathbf{c}}) = \frac{|\mathbf{p}-\hat{\mathbf{p}}|^2}{N} = \frac{1}{N}\sum_i\sum_j (c_{i,j}-\hat{c}_{i,j})^2, \tag{5.27}$$

La Ecuación (5.27) muestra que el MSE disminuye en $|c_{i,j}|^2/N$ cuando el decodificador recibe el coeficiente de transformada $c_{i,j}$ (asumimos que el decodificador recibe el valor exacto del coeficiente, i.e., no hay ninguna pérdida de precisión debido a las limitaciones impuestas por la aritmética del ordenador). Ahora es claro que los mayores coeficientes $c_{i,j}$ (los más grandes en valor absoluto, independientemente de sus signos) contienen la información que reduce más la distorsión MSE, por lo que un codificador progresivo debe enviar esos coeficientes en primer lugar. Éste es un principio importante de SPIHT.

Otro principio se basa en la observación de que los bits más significativos de un entero binario cuyo valor es próximo al máximo tienden a ser pequeños. Ésto sugiere que los bits más significativos contienen la información más importante de la imagen, y que se debe ser enviada en primer lugar al decodificador (o escrita primero en el stream comprimido).

El método utilizado por la transmisión progresiva SPIHT incorpora estos dos principios. SPIHT ordena los coeficientes y transmite sus bits más significativos primero. Para simplificar la descripción, asumimos que la primera información de ordenación es transmitida explícitamente al decodificador; la siguiente sección muestra una forma eficiente para codificar esta información.

Vamos a demostrar cómo utiliza el codificador SPIHT estos principios para transmitir progresivamente los coeficientes wavelet para el decodificador (o los escribe en el stream comprimido), comenzando con la información más importante. Asumimos que ya ha sido aplicada una transformada wavelet a la imagen (SPIHT es un método de codificación, por lo que puede trabajar con cualquier transformada wavelet) y que los coeficientes transformados $c_{i,j}$ ya están almacenados en la memoria.

k		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	sign	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s
msb	14	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	13	a	b	1	1	1	1	0	0	0	0	0	0	0	0	0	0
	12	c	d	e	f	g	h	1	1	1	0	0	0	0	0	0	0
	11	i	j	k	l	m	n	o	p	q	1	0	0	0	0	0	0
	$\vdots$	$\vdots$	$\vdots$														$\vdots$
lsb	0	r	s	t	u	v	w	x	y		$\ldots$		$\ldots$		$\ldots$		z
$m(k) = i, j$		$2,3$	$3,4$	$3,2$	$4,4$	$1,2$	$3,1$	$3,3$	$4,2$	$4,1$	$\ldots$		$\ldots$		$\ldots$		$4,3$

Tabla 5.56: Coeficientes de transformada ordenados por magnitudes absolutas.

Los coeficientes se ordenan (ignorando sus signos), y la información de ordenación está contenida en un array m tal que un elemento del array $m(k)$ contiene las coordenadas (i, j) de un coeficiente $c_{i,j}$, y tal que $\left|c_{m(k)}\right| \geq \left|c_{m(k+1)}\right|$ para todos los valores de k. La Tabla 5.56 muestra los valores hipotéticos de 16 coeficientes. Cada uno se muestra como un número de 16 bits donde el bit más significativo (bit 15) es el signo y los restantes 15 bits (numerados de 14 a 0, de arriba a abajo) constituyen la magnitud. El primer coeficiente $c_{m(1)} = c_{2,3}$ es $s1aci\ldots r$ (donde s, a, etc., son bits). El segundo $c_{m(2)} = c_{3,4}$ es $s1bdj\ldots s$, y así sucesivamente.

La información de ordenación que el codificador tiene que transmitir es la secuencia $m(k)$, o

$$(2,3),(3,4),(3,2),(4,4),(1,2),(3,1),(3,3),(4,2),\ldots,(4,3).$$

Además, se tienen que transmitir las 16 señales, y los 16 coeficientes en orden de bits significativos. Una transmisión directa enviaría los 16 números

$$ssssssssssssssss,\quad 1100000000000000,\quad ab11110000000000,$$
$$cdefgh1110000000,\quad ijklmnopq1000000,\ldots,rstuvwxy\ldots z,$$

pero esto es claramente un desperdicio. En vez de éso, el codificador entra en un bucle, donde en cada iteración realiza un *etapa de clasificación u ordenación* y una *etapa de refinamiento*. En la primera iteración transmite el número $l = 2$ (el número de coeficientes $c_{i,j}$ en nuestro ejemplo que satisfacen $2^{14} \leq |c_{i,j}| < 2^{15}$) seguido por los dos pares de coordenadas $(2,3)$ y $(3,4)$ y por los signos de los dos primeros coeficientes. Esto se efectúa en la primera etapa de clasificación. Esta información permite al decodificador construir versiones aproximadas de los 16 coeficientes como sigue: Los coeficientes $c_{2,3}$ y $c_{3,4}$ son construidos como los números de 16 bits $s100\ldots0$. Los 14 coeficientes restantes se construyen como todo ceros. Así es como los bits más significativos de los mayores coeficientes son transmitidos en primer lugar al decodificador.

La siguiente etapa del codificador es el paso de refinamiento, pero esto no se lleva a cabo en la primera iteración.

En la segunda iteración el codificador realiza ambos pasos: En el paso de clasificación transmite el número $l = 4$ (el número de coeficientes $c_{i,j}$ en nuestro ejemplo que satisfacen $2^{13} \leq |c_{i,j}| < 2^{14}$), seguidos de los cuatro pares de coordenadas $(3,2)$, $(4,4)$, $(1,2)$, y $(3,1)$ y por los signos de los cuatro coeficientes. En el paso de refinamiento transmite los dos bits a y b. Éstos son el 14-ésimo bit más significativos de los dos coeficientes transmitidos en la iteración anterior.

La información recibida hasta el momento permite al decodificador mejorar los 16 coeficientes aproximados construidos en la iteración previa. Los seis primeros se convierten en

$$c_{2,3} = s1a0\ldots0,\ c_{3,4} = s1b0\ldots0,\ c_{3,2} = s0100\ldots0,$$
$$c_{4,4} = s0100\ldots0,\ c_{1,2} = s0100\ldots0,\ c_{3,1} = s0100\ldots0,$$

y los 10 coeficientes restantes no cambian.

◇ **Ejercicio 5.17 (sol. en pág. 1102):** Efectúense los pasos de clasificación y de refinamiento de la siguiente (tercera) iteración.

Los principales pasos del codificador SPIHT ahora deberían ser fáciles de comprender. Son los siguientes:

- *Paso 1:* Dada una imagen para ser comprimida, se calcula su transformada wavelet utilizando cualquier filtro wavelet adecuado, se descompone ésta en los coeficientes de transformada $c_{i,j}$, y se representan los coeficientes resultantes con un número fijo de bits. (En la discusión que sigue se usan los términos *píxel* y *coeficiente* indistintamente.) Asumimos que los coeficientes son representados como números de 16 bits de magnitud con signo. El bit más a la izquierda es el signo, y los 15 bits restantes son la magnitud. (Tenga en cuenta que la representación de signo-magnitud es diferente del método de complemento a 2, que es utilizado por los equipos informáticos a representar números con signo.) Tales números pueden tener valores desde $-\left(2^{15}-1\right)$ a $2^{15}-1$. Se establece n a $\left|\log_2 \text{máx}_{i,j}\left(c_{i,j}\right)\right|$. En nuestro caso n se establecerá en $\left|\log_2\left(2^{15}-1\right)\right| = 14$.
- *Paso 2:* Etapa de clasificación u ordenación: Se transmite el número l de coeficientes $c_{i,j}$ que satisfacen $2^n \leq |c_{i,j}| < 2^{n+1}$. Se sigue con los l pares de coordenadas y los l bits de signo de esos coeficientes.
- *Paso 3:* Etapa de refinamiento: Se transmite el n-ésimo bit más significativo de todos los coeficientes $c_{i,j}$ que satisfacen $|c_{i,j}| \geq 2^{n+1}$. Éstos son los coeficientes que han sido seleccionados en etapas de clasificación previas (sin incluir el paso 2 precedente de ordenación).
- *Paso 4:* Iteración: Decrementar n en 1. Si se requiren más iteraciones (o se desean), se vuelve al Paso 2.

La última iteración se realiza normalmente para $n = 0$, pero el codificador puede detenerse antes, en cuyo caso la información de la imagen menos importante (algunos de los bits menos significativos de todos los coeficientes wavelet) no se transmitirá. Ésta es la opción natural, con pérdidas de SPIHT. Es equivalente a la cuantificación escalar, pero produce mejores resultados que los que se consiguen normalmente con la cuantificación escalar, puesto que los coeficientes se transmiten de forma ordenada. Una alternativa es que el codificador transmita la imagen completa (i.e., todos los bits de todos los coeficientes wavelet) y el decodificador pueda detenerse cuando la decodificación imagen reconstruida alcance una cierta calidad. Esta calidad puede ser predeterminada por el usuario o determinada automáticamente por el decodificador en tiempo de ejecución.

5.14.1. Algoritmo de ordenación del conjunto de particionamiento

El método como se ha descrito hasta ahora es sencillo, ya que hemos asumido que los coeficientes habían sido ordenados antes del bucle iniciado. En la práctica, la imagen puede tener $1K \times 1K$ píxeles o más; puede haber más de un millón de coeficientes, por lo que la clasificación de todos ellos es demasiado lenta. En lugar de ordenar los coeficientes, SPIHT utiliza el hecho de que la clasificación se efectúa comparando dos elementos a la vez, y cada resultado de la comparación es un simple resultado sí/no. Por lo tanto, si ambos, codificador y decodificador, utilizan el mismo algoritmo de ordenación, el codificador puede sólo tener que enviar al decodificador la secuencia de resultados sí/no, y el decodificador puede utilizarlos para duplicar las operaciones del codificador. Ésto es cierto no sólo para la ordenación, sino para cualquier algoritmo basado en comparaciones o en cualquier tipo de ramificación.

El algoritmo real utilizado por SPIHT se basa en la constatación de que realmente no hay necesidad de ordenar *todos* los coeficientes. La tarea principal de la etapa de clasificación en cada iteración

es seleccionar aquellos coeficientes que satisfacen $2^n \leq |c_{i,j}| < 2^{n+1}$. Esta tarea se divide en dos partes. Para un valor dado de n, si un coeficiente $c_{i,j}$ satisface $|c_{i,j}| \geq 2^n$, entonces decimos que es *significativo*; en caso contrario, se le denomina *insignificante*. En la primera iteración, relativamente pocos coeficientes serán significativos, pero su número aumenta de iteración a iteración, porque n se va decrementando cada vez más. La etapa de clasificación debe determinar cuál de los coeficientes significativos satisface $|c_{i,j}| < 2^{n+1}$ y transmitir sus coordenadas al decodificador. Ésta es una parte importante del algoritmo utilizado por SPIHT.

El codificador particiona todos los coeficientes en cierto número de conjuntos T_k y realiza la prueba de significación

$$¿ \max_{(i,j)\in T} |c_{i,j}| \geq 2^n \ ?$$

en cada conjunto T_k. El resultado puede ser, o bien "no" (todos los coeficientes de T_k son insignificantes, en cuyo caso T_k en sí mismo se considera insignificante) o bien "sí" (algunos coeficientes en T_k son significativos, por lo que T_k es en sí mismo significativo). Este resultado se transmite al decodificador. Si el resultado es "sí", entonces T_k se divide tanto en el codificador como en el decodificador, utilizando la misma regla, en subconjuntos y la misma prueba de significación es efectuada en todos los subconjuntos. Este particionamiento se repite hasta que todos los conjuntos significativos se reducen al tamaño 1 (i.e., contienen un coeficiente cada uno, y ese coeficiente es significativo). Así es como se identifican los coeficientes significativos en la etapa de clasificación en cada iteración.

La prueba de significación efectuada sobre un conjunto T se puede resumir mediante

$$S_n(T) = \begin{cases} 1, & \text{máx}_{(i,j)\in T} |c_{i,j}| \geq 2^n, \\ 0, & \text{en otro caso.} \end{cases} \tag{5.28}$$

El resultado, $S_n(T)$, es un único bit que es transmitido al decodificador. El resultado de cada prueba de significación se convierte en un único bit que es escrito en el stream comprimido, por lo que el número de pruebas debe ser minimizado. Para lograr este objetivo, los conjuntos deben ser creados y particionados de manera que se espera que los conjuntos que sean significativos sean grandes y los conjuntos que se espera que sean insignificantes contengan sólo un elemento.

5.14.2. Árboles de orientación espacial

Los conjuntos T_k son creados y particionados con una estructura de datos especial llamada *árbol de orientación espacial*. Esta estructura se define de manera que explota las relaciones espaciales entre los coeficientes wavelet en los diferentes niveles de la pirámide de subbanda. La experiencia ha demostrado que las subbandas en cada nivel de la pirámide exhiben similitud espacial (Figura 5.17b). Las características especiales, tales como un borde recto o una región uniforme, son visibles en todos los niveles en el mismo lugar.

Los árboles de orientación espacial se ilustran en la Figura 5.57a,b para una imagen de 16×16. La figura muestra dos niveles, nivel 1 (paso alto) y nivel 2 (paso bajo). Cada nivel se divide en cuatro subbandas. La subbanda LL2 (la subbanda paso bajo) se divide en cuatro grupos de 2×2 coeficientes cada uno. La Figura 5.57a muestra el grupo superior izquierda, y la Figura 5.57b muestra el grupo inferior derecha. En cada grupo, cada uno de los cuatro coeficientes (excepto el superior izquierda, marcado en gris) se convierte en la raíz de un árbol de orientación espacial. Las flechas muestran ejemplos de cómo están relacionados los distintos niveles de estos árboles. Las flechas gruesas indican cómo cada grupo de 4×4 coeficientes del nivel 2 es el padre de los cuatro grupos en el nivel 1. En general, un coeficiente en la posición (i, j) en la imagen es el padre de los cuatro coeficientes en las ubicaciones $(2i, 2j)$, $(2i + 1, 2j)$, $(2i, 2j + 1)$, y $(2i + 1, 2j + 1)$.

Las raíces de los árboles orientación espacial de nuestro ejemplo se encuentran en la subbanda LL2 (en general, están localizadas en la subbanda LL superior izquierda, que puede ser de cualquier

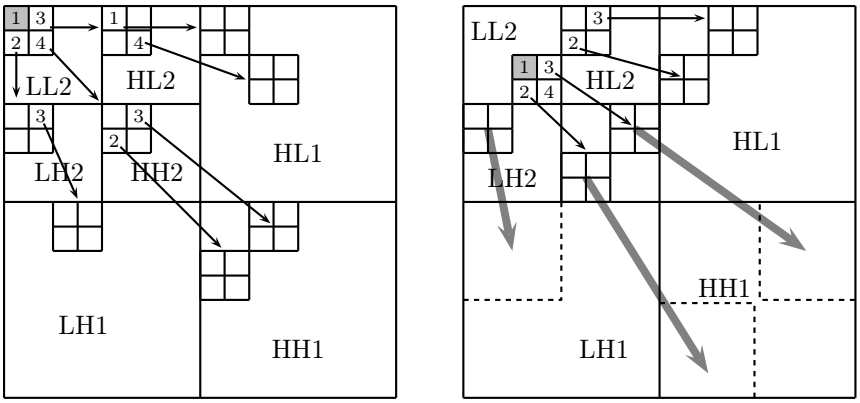


Figura 5.57: Árboles de orientación espacial en SPIHT.

tamaño), pero cualquier coeficiente wavelet, excepto los grises en el nivel 1 (también excepto las hojas), puede ser considerado la raíz de algún subárbol de orientación espacial. Las hojas de todos los árboles se encuentran en el nivel 1 de la pirámide subbanda.

En nuestro ejemplo, la subbanda LL2 es de tamaño 4×4, por lo que se divide en cuatro grupos de 2×2, y tres de los cuatro coeficientes de un grupo se convierte en raíces de árboles. Por consiguiente, el número de árboles en nuestro ejemplo es 12. En general, el número de árboles es $3/4$ del tamaño de las subbandas más altas LL.

Cada una de las 12 raíces en la subbanda LL2 en nuestro ejemplo es el padre de cuatro hijos localizados en el mismo nivel. Sin embargo, los hijos de estos hijos se encuentran en el nivel 1. Ésto es cierto en general. Las raíces de los árboles están localizadas en el nivel más alto, y sus hijos están en el mismo nivel, pero a partir de ese momento, los cuatro hijos de un coeficiente en el nivel k están ubicados en el nivel $k-1$.

Usamos los términos *descendencia directa* o *prole* (*offspring*) para los cuatro hijos de un nodo, y *descendientes* para los hijos, nietos y todos sus descendientes. El algoritmo de ordenación del conjunto de particionamiento utiliza los siguientes cuatro conjuntos de coordenadas:

1. $\mathcal{O}(i,j)$: El conjunto de coordenadas de los cuatro descendientes directos del nodo (i,j). Si el nodo (i,j) es una hoja de un árbol de orientación espacial, entonces $\mathcal{O}(i,j)$ está vacío.

2. $\mathcal{D}(i,j)$: El conjunto de las coordenadas de los descendientes del nodo (i,j).

3. $\mathcal{H}(i,j)$: El conjunto de coordenadas de las raíces de todos los árboles de orientación espacial ($3/4$ de los coeficientes wavelet en subbanda más alta LL).

4. $\mathcal{L}(i,j)$: El conjunto diferencia $\mathcal{D}(i,j) - \mathcal{O}(i,j)$. Este grupo contiene todos los descendientes del nodo (i,j) del árbol, excepto sus cuatro descendientes inmediatos.

Los árboles de orientación espacial se utilizan para crear y particionar los conjuntos T_k. El conjunto de reglas de particionamiento son las siguientes:

1. Los conjuntos iniciales son $\{(i,j)\}$ y $\mathcal{D}(i,j)$, para todo $(i,j) \in \mathcal{H}$ (i.e., para todas las raíces de los árboles de orientación espacial). En nuestro ejemplo hay 12 raíces, por lo que inicialmente serán 24 conjuntos: 12 conjuntos, cada uno conteniendo las coordenadas de una raíz, y 12 conjuntos más, cada uno con las coordenadas de todos los descendientes de una raíz.

1. Inicialización: Fijar n a $\lfloor \log_2 \text{máx}_{i,j} (c_{i,j}) \rfloor$ y transmitir n. Establecer la LSP a vacío. Establecer la LIP a las coordenadas de todas las raíces $(i,j) \in \mathcal{H}$. Establecer la LIS a las coordenadas de todas las raíces $(i,j) \in \mathcal{H}$ que tengan descendientes.
2. Etapa de clasificación (ordenación):
   2.1 para cada entrada (i,j) en la LIP hacer:
      2.1.1 emitir $S_n(i,j)$;
      2.1.2 si $S_n(i,j) = 1$, mover (i,j) a la LSP y emitir el signo de $c_{i,j}$;
   2.2 para cada entrada (i,j) en la LIS hacer:
      2.2.1 si la entrada es de tipo A, entonces
         - emitir $S_n(\mathcal{D}(i,j))$;
         - si $S_n(\mathcal{D}(i,j)) = 1$, entonces
            * para cada $(k,l) \in \mathcal{O}(i,j)$ hacer:
               · emitir $S_n(k,l) = 1$;
               · si $S_n(k,l) = 1$, añadir (k,l) a la LSP, emitir el signo de $c_{k,l}$;
               · si $S_n(k,l) = 0$, añadir (k,l) a la LIP;
            * si $\mathcal{L}(i,j) \neq 0$, mover (i,j) hasta el final de la LIS, como una entrada de tipo B, e ir al paso 2.2.2; en otro caso, eliminar la entrada (i,j) de la LIS;
      2.2.2 si la entrada es de tipo B, entonces
         - emitir $S_n(\mathcal{L}(i,j))$;
         - si $S_n(\mathcal{L}(i,j)) = 1$, entonces
            * añadir cada $(k,l) \in \mathcal{O}(i,j)$ de la LIS como una entrada de tipo A;
            * eliminar (i,j) de la LIS;
3. Etapa de refinamiento: para cada entrada (i,j) de la LSP, excepto aquellas incluidas en la última etapa de clasificación (la que tiene el mismo n), emitir el n-ésimo bit más significativo de $|c_{i,j}|$;
4. Bucle: decrementar n en 1 e ir al paso 2 si es necesario.

Figura 5.58: Algoritmo de codificación SPIHT.

2. Si el conjunto $\mathcal{D}(i,j)$ es significativo, entonces es particionado en $\mathcal{L}(i,j)$, más los cuatro conjuntos de un único elemento con los cuatro descendientes directos de (i,j). En otras palabras, si alguno de los descendientes de nodo (i,j) es significativo, entonces sus cuatro descendientes directos se convierten en cuatro nuevos conjuntos y todos sus otros descendientes se convierten en otro conjunto (para probar su significación en la regla 3).

3. Si $\mathcal{L}(i,j)$ es significativo, entonces es particionado en los cuatro conjuntos $\mathcal{D}(k,l)$, donde (k,l) son la descendencia directa de (i,j).

Una vez comprendidos los árboles de orientación espacial y los conjuntos de particionamiento, el algoritmo de codificación puede ser descrito.

5.14.3. Codificación SPIHT

Es importante mantener los conjuntos de significación del codificador y del decodificador en el mismo punto. Por eso, el algoritmo de codificación utiliza tres listas llamadas: *lista de pixels significativos* (LSP), *lista de píxeles insignificantes* (LIP), y *lista de conjuntos insignificantes* (LIS). Estas son las listas de coordenadas (i,j) que en la LIP y la LSP representan coeficientes individuales, y en la LIS representan, ya sea el conjunto $\mathcal{D}(i,j)$ (un tipo de entrada A), ya sea el conjunto $\mathcal{L}(i,j)$ (un tipo de entrada B).

1. Fijar el umbral. Establecer la LIP a todos los coeficientes de nodos raíz. Establecer la LIS a todos los árboles (asignarlos el tipo D). Establecer la LSP a conjunto vacío.
2. Etapa de clasificación: Comprobar la significación de todos los coeficientes en la LIP:
   2.1 Si es significativo, emitir 1, emitir un bit de signo, y mover el coeficiente a la LSP.
   2.2 Si no es significativo, emitir 0.
3. Comprobar la significación de todos los árboles de la LIS, acorde con el tipo de árbol:
   3.1 Para un árbol de tipo D:
      3.1.1 Si es significativo, emitir 1, y codificar sus hijos:
         3.1.1.1 Si un hijo es significativo, emitir 1, luego un bit de signo, y añadirlo a la LSP.
         3.1.1.2 Si un hijo es insignificante, emitir 0 y agregar el hijo al final de la LIP.
         3.1.1.3 Si los hijos tienen descendientes, mover el árbol hasta el final de la LIS como de tipo de L, en caso contrario eliminarlo de la LIS.
      3.1.2 Si es insignificante, emitir 0.
   3.2 Para un árbol de tipo L:
      3.2.1 Si es significativo, emitir 1, añadir a cada uno de los hijos al final de la LIS como una entrada de tipo D y eliminar el árbol padre de la LIS.
      3.2.2 Si es insignificante, emitir 0.
4. Bucle: Decrementar el umbral e ir al paso 2 si es necesario.

Figura 5.59: Un algoritmo de codificación SPIHT simplificado.

La LIP contiene las coordenadas de los coeficientes que eran insignificantes en la etapa de clasificación previa. En el paso actual son probados, y aquellos que sean significativos se trasladan a la LSP. De manera similar, los conjuntos en la LIS son probados en orden secuencial, y cuando se encuentra un conjunto significativo, se elimina de la LIS y es particionado. Los nuevos subconjuntos con más de un coeficiente se colocan de nuevo en la LIS, para ser testeados más tarde, y los subconjuntos con un elemento se prueban y se anexan a la LIP o a la LSP, en función de los resultados del test. El paso de refinamiento transmite los n-ésimos bits más significativos de las entradas en la LSP.

La Figura 5.58 muestra este algoritmo en detalle. La Figura 5.59 es una versión simplificada, para los lectores que se sienten intimidados por demasiados detalles.

El decodificador ejecuta el algoritmo detallado de la Figura 5.58. Siempre funciona al unísono con el codificador, no obstante, las notas siguientes arrojan más luz sobre su funcionamiento:

1. El paso 2.2 del algoritmo evalúa todas las entradas de la LIS. No obstante, el paso 2.2.1 añade ciertas entradas a la LIS (como tipo B) y el paso 2.2.2 agrega otras entradas a la LIS (como tipo A). Es importante darse cuenta de que todas estas entradas también son evaluadas en el paso 2.2 en la misma iteración.

2. El valor de n es decrementado en cada iteración, pero no hay necesidad de llegar hasta cero. El bucle puede detenerse tras cualquier iteración, lo que produce una compresión con pérdida. Normalmente, el usuario especifica el número de iteraciones, pero también es posible que el usuario especifique la cantidad de distorsión aceptable (en unidades de MSE), y el codificador puede utilizar la Ecuación (5.27) para decidir cuándo detener el bucle.

3. El codificador conoce los valores de los coeficientes wavelet $c_{i,j}$ y los utiliza para calcular los bits de S_n (Ecuación (5.28)), que transmite (i.e., escribe en el flujo de datos comprimido). Estos bits son introducidos por el decodificador, que los usa para calcular los valores de $c_{i,j}$. El algoritmo

ejecutado por el decodificador es el de la Figura 5.58, pero cambiando palabra "emitir" por "introducir".

4. La información de ordenación, previamente denotada por $m(k)$, es recuperada cuando las coordenadas de los coeficientes significativos son añadidos a la LSP en los pasos 2.1.2 y 2.2.1. Ésto implica que los coeficientes indicados mediante coordenadas en la LSP son ordenados según

$$\lfloor \log_2 |c_{m(k)}| \rfloor \geq \lfloor \log_2 |c_{m(k+1)}| \rfloor ,$$

para todos los valores de k. El decodificador recupera el orden, ya que sus tres listas (LIS, LIP, y LSP) se actualizan de la misma manera que las del codificador (recordemos que el decodificador funciona al unísono con el codificador). Cuando el decodificador introduce los datos, sus tres listas son idénticas a las del codificador en el momento en que (el codificador) emite esos datos.

5. El codificador comienza con los coeficientes wavelet $c_{i,j}$; nunca llega a "ver" la imagen actual. El decodificador, sin embargo, tiene que mostrar la imagen y actualizar la presentación en cada iteración. En cada iteración, cuando se mueven las coordenadas (i,j) de un coeficiente $c_{i,j}$ a la LSP como una entrada, se sabe (para ambos, codificador y decodificador) que $2^n \leq |c_{i,j}| < 2^{n+1}$. Como resultado, el mejor valor que el decodificador puede dar del coeficiente $\hat{c}_{i,j}$ que está siendo reconstruido está a medio camino entre 2^n y $2^{n+1} = 2 \times 2^n$. En consecuencia, el decodificador establece $\hat{c}_{i,j} = \pm 1{,}5 \times 2^n$ (el signo de $\hat{c}_{i,j}$ es introducido por el decodificador justo después de la inserción). Durante el paso de refinamiento, cuando el decodificador introduce el valor real del n-ésimo bit de $c_{i,j}$, mejora el valor $1{,}5 \times 2^n$ ya sea sumando 2^{n-1} al mismo (si el bit introducido era un 1), ya sea restando 2^{n-1} del mismo (si el bit introducido era un 0). De esta manera, el decodificador puede mejorar el aspecto de la imagen (o, equivalentemente, reducir la distorsión) durante *ambas* etapas, de clasificación y de refinamiento.

Es posible mejorar el rendimiento de SPIHT mediante la codificación de entropía de la salida del codificador, pero la experiencia muestra que la ganancia de compresión adicional que se logra de esta manera es mínima y no justifica el gasto adicional de tiempo tanto en la codificación como en la decodificación. Al parecer los signos y los bits individuales de los coeficientes emitidos en cada iteración se distribuyen de manera uniforme, por lo que una codificación de entropía sobre ellos no produce ningún tipo de compresión. Los bits $S_n(i,j)$ y $S_n(\mathcal{D}(i,j))$, por otra parte, están distribuidos de forma no uniforme y pueden beneficiarse de este tipo de codificación.

5.14.4. Ejemplo

Asumimos que una imagen de 4×4 ha sido transformada por completo, y los 16 coeficientes están almacenados en la memoria como números con signo de 6 bits de magnitud (un bit de signo seguido por cinco bits de magnitud). Éstos se muestran en la Figura 5.60, junto con el único árbol de orientación espacial. El algoritmo de codificación inicializa la LIP al conjunto de elementos uno $\{(1,1)\}$, la LIS al conjunto $\{\mathcal{D}(1,1)\}$ y la LSP al conjunto vacío. El coeficiente más grande es 18, por lo que n se establece en $\lfloor \log_2 18 \rfloor = 4$. Se muestran las dos primeras iteraciones.

Etapa de clasificación 1:
$2^n = 16$
¿Es $(1,1)$ significativo? sí: emitir un 1.
LSP = $\{(1,1)\}$, emitir el bit de signo: 0.

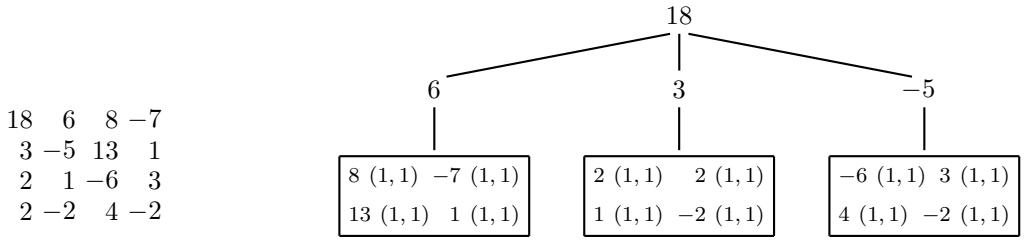


Figura 5.60: Dieciséis coeficientes y un árbol de orientación espacial.

¿Es $\mathcal{D}(1,1)$ significativo? no: emitir un 0.
LSP = $\{(1,1)\}$, LIP = $\{\ \}$, LIS = $\{\mathcal{D}(1,1)\}$.
Tres bits emitidos.
Etapa de refinamiento 1: no se emite ningún bit (esta etapa trata con coeficientes de la etapa de clasificación $n-1$).
Decrementar n a 3.
Etapa de clasificación 2:
$2^n = 8$.
¿Es $\mathcal{D}(1,1)$ significativo? sí: emitir un 1.
¿Es $(1,2)$ significativo? no: emitir un 0.
¿Es $(2,1)$ significativo? no: emitir un 0.
¿Es $(2,2)$ significativo? no: emitir un 0.
LIP = $\{(1,2),(2,1),(2,2)\}$, LIS = $\{\mathcal{L}(1,1)\}$.
¿Es $\mathcal{L}(1,1)$ significativo? sí: emitir un 1.
LIS = $\{\mathcal{D}(1,2),\mathcal{D}(2,1),\mathcal{D}(2,2)\}$.
¿Es $\mathcal{D}(1,2)$ significativo? sí: emitir un 1.
¿Es $(1,3)$ significativo? sí: emitir un 1.
LSP = $\{(1,1),(1,3)\}$, emitir el bit de signo: 1.
¿Es $(2,3)$ significativo? sí: emitir un 1.
LSP = $\{(1,1),(1,3),(2,3)\}$, emitir el bit de signo: 1.
¿Es $(1,4)$ significativo? no: emitir un 0.
¿Es $(2,4)$ significativo? no: emitir un 0.
LIP = $\{(1,2),(2,1),(2,2),(1,4),(2,4)\}$,
LIS = $\{\mathcal{D}(2,1),\mathcal{D}(2,2)\}$.
¿Es $\mathcal{D}(2,1)$ significativo? no: emitir un 0.
¿Es $\mathcal{D}(2,2)$ significativo? no: emitir un 0.
LIP = $\{(1,2),(2,1),(2,2),(1,4),(2,4)\}$,
LIS = $\{\mathcal{D}(2,1),\mathcal{D}(2,2)\}$.
LSP = $\{(1,1),(1,3),(2,3)\}$
Catorce bits emitidos.
Etapa de refinamiento 2: Tras la iteración 1, la LSP incluye la entrada $(1,1)$, cuyo valor es $18 = 10010_2$.
Un bit es emitido.
Etapa de clasificación 3:
$2^n = 4$.
¿Es $(1,2)$ significativo? sí: emitir un 1.
LSP = $\{(1,1),(1,3),(2,3),(1,2)\}$, emitir un bit de signo: 1.
¿Es $(2,1)$ significativo? no: emitir un 0.
¿Es $(2,2)$ significativo? sí: emitir un 1.

LSP = $\{(1,1),(1,3),(2,3),(1,2),(2,2)\}$, emitir un bit de signo: 0.
¿Es $(1,4)$ significativo? sí: emitir un 1.
LSP = $\{(1,1),(1,3),(2,3),(1,2),(2,2),(1,4)\}$, emitir un bit de signo: 1.
¿Es $(2,4)$ significativo? no: emitir un 0.
LIP = $\{(2,1),(2,4)\}$.
¿Es $\mathcal{D}(2,1)$ significativo? no: emitir un 0.
¿Es $\mathcal{D}(2,2)$ significativo? sí: emitir un 1.
¿Es $(3,3)$ significativo? sí: emitir un 1.
LSP = $\{(1,1),(1,3),(2,3),(1,2),(2,2),(1,4),(3,3)\}$, emitir un bit de signo: 0.
¿Es $(4,3)$ significativo? sí: emitir un 1.
LSP = $\{(1,1),(1,3),(2,3),(1,2),(2,2),(1,4),(3,3),(4,3)\}$, emitir un bit de signo: 1.
¿Es $(3,4)$ significativo? no: emitir un 0.
LIP = $\{(2,1),(2,4),(3,4)\}$.
¿Es $(4,4)$ significativo? no: emitir un 0.
LIP = $\{(2,1),(2,4),(3,4),(4,4)\}$.
LIP = $\{(2,1),(2,4),(3,4),(4,4)\}$, LIS = $\{\mathcal{D}(2,1)\}$,
LSP = $\{(1,1),(1,3),(2,3),(1,2),(2,2),(1,4),(3,3),(4,3)\}$.
Dieciséis bits emitidos.
Etapa de refinamiento 3:
Tras la iteración 2, la LSP incluye las entradas $(1,1)$, $(1,3)$, y $(2,3)$ cuyos valores respectivos son $18 = 10010_2$, $8 = 1000_2$, y $13 = 1101_2$. Tres bits son emitidos.
Tras dos iteraciones, han sido emitidos 37 bits.

5.14.5. QTCQ

En estrecha relación con SPIHT, el método QTCQ (*quadtree classification and trellis coding quantization* o quadtree de clasificación y cuantificación de codificación reticular) [Banister y Fischer 99] utiliza un menor número de listas que SPIHT y explícitamente forma clases de los coeficientes wavelet para cuantificarlos posteriormente por medio de los algoritmos ACTCQ y TCQ (arithmetic and trellis coded quantization o cuantificación por codificación aritmética y reticular) de [Joshi, Crump, y Fisher 93].

El método utiliza los árboles de orientación espacial originalmente desarrollados por SPIHT. Este tipo de árbol es un caso especial de quadtree (Sección 4.30). El algoritmo de codificación es iterativo. En la n-ésima iteración, si cualquier elemento de este quadtree se detecta como significativo, entonces los cuatro elementos más altos en el árbol se definen como de la clase n. También se convierten en las raíces de cuatro nuevos quadtrees. Cada uno de los cuatro nuevos árboles es probado para la significación, moviéndose hacia abajo por cada árbol hasta encontrar todos los elementos significativos. Todos los coeficientes wavelet declarados en la clase n se almacenan en una *lista de píxeles* (LP). La LP es inicializada con todos los coeficientes wavelet en la subbanda de frecuencia más baja (LFS). La prueba de significación se lleva a cabo mediante la función $S_T(k)$, que se define por

$$S_T(k) = \begin{cases} 1, & \text{máx}_{(i,j)\in k} |C_{i,j}| \geq T, \\ 0, & \text{en otro caso,} \end{cases}$$

donde T es el umbral actual para la significación y k es un árbol de coeficientes wavelet. El algoritmo de codificación QTCQ utiliza este test, y se muestra en la Figura 5.61.

El decodificador QTCQ es similar. Todos los "emitir" en la Figura 5.61 deben ser reemplazados por "introducir", y codificación ACTCQ debe ser reemplazada por decodificación ACTCQ.

La implementación de QTCQ, como se describe en [Banister y Fischer 99], no transmite la imagen de forma progresiva, pero los autores afirman que puede añadírsele esta propiedad.

```
1. Inicialización:
    Inicializar LP con todos los C_{i,j} en la LFS,
    Inicializar LIS todos los nodos padres,
    Emitir n = ⌊log2 (max |C_{i,j}| /q)⌋.
    Establecer el umbral en T = q2^n, donde q es un factor de calidad.
2. Clasificación (ordenación):
    for cada nodo k en LIS do
      emitir S_T (k)
      if S_T (k) = 1 then
        for cada hijo de k do
          mover los coeficientes a LP
          añadir a LIS como un nuevo nodo
        endfor
        eliminar k de LIS
      endif
    endfor
3. Cuantificación: Para cada elemento en LP,
    cuantificar y codificar utilizando ACTCQ.
    (usar TCQ con tamaño de paso Δ = α · q).
4. Actualización: Eliminar todos los elementos en LP. Fijar T = T/2.
    Ir al paso 2.
```

Figura 5.61: Codificación QTCQ.

5.15. CREW

El método CREW (*compression with reversible embedded wavelets* o compresión con wavelets embebidas reversibles) fue desarrollado en 1994 por A. Zandi en Ricoh Silicon Valley para la compresión con pérdida y sin pérdidas de calidad, de imágenes médicas. Más tarde se dio cuenta de que él desarrolló independientemente un método muy similar a SPIHT (Sección 5.14), que es por lo que no se describen los detalles de CREW aquí. El lector interesado puede consultar [Zandi et al. 95], sin embargo, se pueden encontrar descripciones más recientes y detalladas en [CREW 00].

5.16. EZW

El método SPIHT es de alguna manera una extensión de EZW, por lo que este método, cuyo nombre completo es "embedded coding using zerotrees of wavelet coefficients" o "codificación embebida utilizando árboles-cero de coeficientes wavelet", se describe aquí esbozando sus principios y se muestra un ejemplo. Algunos de los detalles, tales como la relación entre los padres y los descendientes en un árbol de orientación espacial, y el significado del término "embebido", se describen en la Sección 5.14.

El método EZW, tal como se aplica en la práctica, comienza mediante el cálculo de la transformada wavelet del filtro espejo en cuadratura (QMF o quadrature mirror filter) simétrico 9-tap [Adelson et al. 87]. A continuación se repite el bucle principal para los valores umbral que se redujeron a la mitad al final cada iteración. El umbral es utilizado para calcular un *mapa de significación* de los coeficientes wavelet signiticativos e insignificantes. Los árboles-cero (*zerotrees*) se utilizan para representar el mapa de significación de una forma eficiente. Los pasos principales son los siguientes:

1. Inicialización: establecer el umbral T a la menor potencia de 2 que es mayor que $\max_{(i,j)} |c_{i,j}| /2$, donde $c_{i,j}$ son los coeficientes wavelet.

2. Codificación del mapa de significación: Escanear todos los coeficientes en un camino predeterminado y emitir un símbolo cuando $|c_{i,j}| > T$. Cuando el decodificador introduce este símbolo, establece $c_{i,j} = \pm 1{,}5T$.

3. Refinamiento: Refinar cada coeficiente significativo enviando un bit más que su representación binaria. Cuando el decodificador recibe éste, incrementa el valor del coeficiente actual en $\pm 0{,}25T$.

4. Establecer $T = {}^{T}/_{2}$, e ir al paso 2 si es necesario efectuar más iteraciones.

Un coeficiente de wavelet $c_{i,j}$ es considerado insignificante con respecto al umbral actual T si $|c_{i,j}| \leq T$. La estructura de datos zerotree se basa en el siguiente bien conocido resultado experimental: Si un coeficiente wavelet a una escala gruesa (i.e., en la parte superior de la pirámide de la imagen) es insignificante con respecto a un umbral dado T, entonces todos los coeficientes con la misma orientación en la misma ubicación espacial a escalas más finas (i.e., situado más bajo en la pirámide) es muy probable que sean insignificantes con respecto a T.

En cada iteración, todos los coeficientes se escanean en el orden mostrado en la Figura 5.62a. Ésto garantiza que, cuando se visita un nodo, todos sus padres ya han sido escaneados. El escaneo comienza en la subbanda de más baja frecuencia LL_n, continúa con las subbandas HL_n, LH_n, y HH_n, y desciende hasta el nivel $n-1$, donde escanea HL_{n-1}, LH_{n-1}, y HH_{n-1}. Cada subbanda es escaneada completamente antes de que el algoritmo proceda con la siguiente subbanda.

Cada coeficiente visitado en el escaneo es clasificado como una raíz zerotree (ZTR), un cero aislado (IZ), significación positiva (POS) o significación negativa (NEG). Una raíz zerotree es un coeficiente que es insignificante y todos sus descendientes (en el mismo árbol de orientación espacial) también son insignificantes. Tal coeficiente se convierte en la raíz de un zerotree. Es codificado con un símbolo especial (denotado por ZTR), y el aspecto importante es que sus descendientes no tienen que ser codificados en la iteración actual. Cuando el decodificador introduce un símbolo ZTR, asigna un valor cero a los coeficientes y a todos sus descendientes en el árbol de la orientación espacial. Sus valores son mejorados (refinados) en posteriores iteraciones. Un cero aislado es un coeficiente que es insignificante, pero tiene algunos descendientes significativos. Tal coeficiente es codificado con el símbolo especial IZ. Las otras dos clases están formadas por coeficientes que son significativos y son positivos o negativos. El diagrama de flujo de la Figura 5.62b ilustra esta clasificación. Observe que un coeficiente se clasifica en una de cinco clases, pero la quinta clase (un nodo zerotree) no se codifica.

Los coeficientes en el nivel más bajo de la pirámide no tienen hijos, por lo que no pueden ser raíces de zerotrees. En consecuencia, son clasificados como cero aislado, positivo significativo, o negativo significativo.

El zerotree puede ser visto como una estructura que ayuda a encontrar la insignificancia. La mayor parte de los métodos que tratan de encontrar la estructura de una imagen intentan encontrar la significación. El método IFS de la Sección 4.35, por ejemplo, intenta localizar partes de la imagen que son similares en tamaño y/o transformación, y ésto es mucho más difícil que localizar las partes que son insignificantes en relación a otras partes.

Dos listas son utilizadas por el codificador (y también por el decodificador, el cual funciona al unísono) en el proceso de escaneo. La *lista dominante* contiene las coordenadas de los coeficientes que no han sido considerados significativos. Ellos se almacenan en orden scan (de izquierda a derecha y de arriba abajo), según los niveles de la pirámide y dentro de cada nivel por subbandas. La *lista subordinada* contiene las *magnitudes* (no las coordenadas) de los coeficientes que se han considerado significativos. Cada lista es escaneada una vez por iteración.

Una iteración consiste en una *pasada dominante* seguida de una *pasada subordinada*. En la pasada dominante, los coeficientes de la lista dominantes son probados para la significación. Si un coeficiente se encontró significativo, entonces (1) su signo es determinado (2) se clasifica como POS o NEG, (3) su magnitud es agregada a la lista subordinada, y (4) se establece a cero en la memoria (en el array que

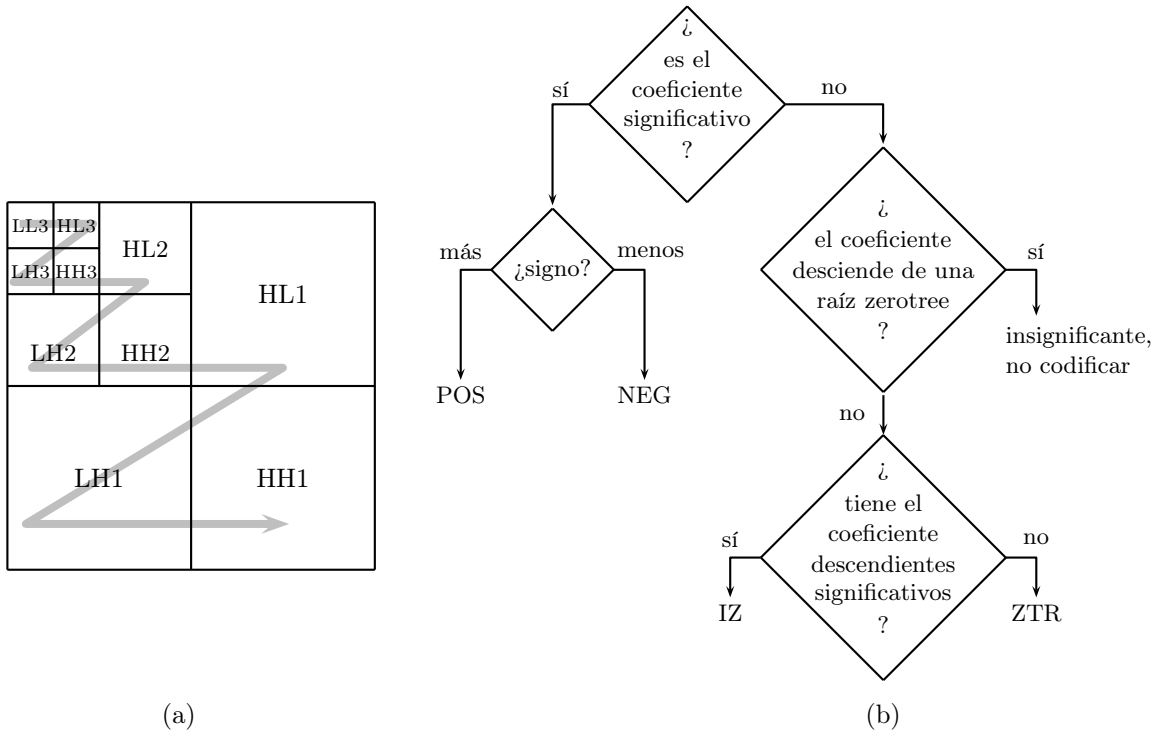


Figura 5.62: (a) Escaneado de un zerotree. (b) Clasificación de un coeficiente.

contiene todos los coeficientes wavelet). El último paso se efectúa de forma que el coeficiente no impida la ocurrencia de un zerotree en las pasadas dominantes posteriores para umbrales más pequeños.

Imagine que el umbral inicial es $T = 32$. Cuando un coeficiente $c_{i,j} = 63$ es localizado en la primera iteración, se encuentra que es significativo. Dado que es positivo, es codificado como POS. Cuando el decodificador introduce este símbolo, no conoce su valor, pero sabe que el coeficiente es positivo significativo, i.e., satisface $c_{i,j} > 32$. El decodificador también sabe que $c_{i,j} \leq 64 = 2 \times 32$, por lo que el mejor valor que el decodificador puede asignar al coeficiente es ${}^{(32+64)}/_{2} = 48$. A continuación, el coeficiente es establecido en 0, por lo que las subsiguientes iteraciones no lo identificarán como significativo.

Podemos pensar en el umbral T como un indicador que especifica una posición de bit. En cada iteración, el umbral indica la siguiente posición del bit menos significativo. También podemos ver T como el ancho de la cuantificación actual. En cada iteración esa anchura es dividida por 2, de esta manera se conoce otro bit menos significativo de los coeficientes.

Durante una pasada subordinada, la lista subordinada es escaneada y el codificador emite un 0 ó un 1 para cada coeficiente para indicar al decodificador cómo debe ser mejorada la magnitud del coeficiente . En el ejemplo de $c_{i,j} = 63$, el codificador transmite un 1, indicando al decodificador que el valor real del coeficiente es mayor que 48. El decodificador utiliza esa información para mejorar la magnitud del coeficiente de 48 a ${}^{(46+64)}/_{2} = 56$. Si se hubiera transmitido un 0, el decodificador habría refinado el valor del coeficiente a ${}^{(32+48)}/_{2} = 40$.

La cadena de bits generada por el codificador durante la pasada subordinada es codificada entrópicamente utilizando una versión personalizada de la codificación aritmética adaptativa (Sección 2.15).

Al final de la pasada subordinada, el codificador (y, al mismo paso, también el decodificador)

63	−34	49	10	7	13	−12	7
−31	23	14	−13	3	4	6	−1
15	14	3	−12	5	−7	3	9
−9	−7	−14	8	4	−2	3	2
−5	9	−1	47	4	6	−2	2
3	0	−3	2	3	−2	0	4
2	−3	6	−4	3	6	3	6
5	11	5	6	0	3	−4	4

(a)

Subbanda	Valor coef.	Símbolo	Valor reconstr.	Nota
LL3	63	POS	48	1
HL3	−34	NEG	−48	
LH3	−31	IZ	0	2
HH3	23	ZTR	0	3
HL2	49	POS	48	
HL2	10	ZTR	0	4
HL2	14	ZTR	0	
HL2	−13	ZTR	0	
LH2	15	ZTR	0	
LH2	14	IZ	0	5
LH2	−9	ZTR	0	
LH2	−7	ZTR	0	
HL1	7	Z	0	
HL1	13	Z	0	
HL1	3	Z	0	
HL1	4	Z	0	
LH1	−1	Z	0	
LH1	47	POS	48	6
LH1	−3	Z	0	
LH1	−2	Z	0	

(b)

Figura 5.63: Un ejemplo EZW: Tres niveles de una imagen de 8×8.

ordena las magnitudes en la lista subordinada en orden decreciente.

El codificador detiene el bucle cuando se cumple una condición determinada. El usuario puede, por ejemplo, especificar la tasa de bits deseada (número de bits por píxel). El codificador conoce el tamaño de la imagen (número de píxeles), por lo que detecta cuándo se ha alcanzado o excedido la tasa de bits deseada. La ventaja es que la razón de compresión se conoce de antemano (de hecho, es determinada por el usuario). La desventaja es que puede perderse demasiada información si la razón de compresión es demasiado alta, lo que conduce a una imagen mal reconstruida. También es posible para el codificador parar en medio de una iteración, cuando la tasa de bits exacta especificada por el usuario ha sido alcanzada. Sin embargo, en tal caso, la última palabra de código puede estar incompleta, lo que lleva a la decodificación incorrecta de un coeficiente.

El usuario también puede especificar el monto de bits (el tamaño del stream comprimido) como una condición de parada. Ésto es similar a la especificación de la tasa de bits. Una alternativa es que el usuario especifique la distorsión máxima aceptable (la diferencia entre el original y la imagen comprimida). En tal caso, el codificador itera hasta que el umbral se convierte en 1, y el decodificador para la decodificación cuando la distorsión máxima aceptable ha sido alcanzada.

5.16.1. Ejemplo

El siguiente ejemplo es el que se encuentra en [Shapiro 93]. La Figura 5.63a muestra tres niveles de la transformada wavelet de una imagen de 8×8. El valor máximo es 63, por lo que el umbral inicial puede estar en cualquier lugar dentro del intervalo $(31, 64]$. Lo hemos establecido en 32. La Tabla 5.63b muestra los resultados de la primera pasada dominante.

Notas:

1. El coeficiente superior izquierda es 63. Es mayor que el umbral, y es positivo, por lo que se genera un símbolo POS que es transmitido por el codificador (y el 63 se cambia a 0). El decodificador asigna a este símbolo POS el valor 48, el punto medio del intervalo $[32, 64)$.

2. El coeficiente 31 es insignificante con respecto a 32, pero no es una raíz zerotree, ya que uno de sus descendientes (el 47 en LH1) es significativo. Por lo tanto, el 31 es un cero aislado (IZ).

3. El 23 es menor que 32. Además, todos sus descendientes (el 3, -12, -14, y 8 en HH2, y todos los HH1) son insignificantes. Por lo tanto, el 23 es una raíz zerotree (ZTR). Como resultado, el codificador no generará ningún símbolo en la pasada dominante para sus descendientes (es por éso, que ninguno de los coeficientes HH2 y HH1 aparece en la tabla).

4. El 10 es menor que 32, y todos sus descendientes (el -12, 7, 6, y -1 en HL1) son también menores que 32. En consecuencia, el 10 se convierte en una raíz zerotree (ZTR). Observe que el -12 es mayor, en valor absoluto, que el 10, pero es aún menor que el umbral.

5. El 14 es insignificante con respecto al umbral, pero uno de sus hijos (que son -1, 47, -3, y 2) es significativo. En consecuencia, el 14 se convierte en un IZ.

6. El 47 en la subbanda LH1 es significativo con respecto al umbral, por lo que es codificado como POS. Es entonces cambiado a cero, de modo que una pasada futura (con un umbral de 16) codificará a su padre, 14, como una raíz zerotree.

Cuatro coeficientes significativos son transmitidos durante la primera pasada dominante. Todo lo que el decodificador conoce acerca de ellos es que se encuentran en el intervalo $[32, 64)$. Serán refinados durante la primera pasada subordinada, por lo que el decodificador será capaz de colocarlos ya sea en $[32, 48)$ (si recibe un 0), ya sea en $[48, 64)$ (si recibe un 1). El codificador genera y transmite los bits "1010" para los cuatro coeficientes significativos 63, 34, 49, y 47. En consecuencia, el decodificador los refina a 56, 40, 56, y 40, respectivamente.

En la segunda pasada dominante, sólo aquellos coeficientes que aún no se han considerado significativos son escaneados y testados. Los que se consideran importantes son tratados como cero cuando el codificador comprueba las raíces zerotree. Esta segunda pasada termina identificando el -31 en LH3 como NEG, el 23 en HH3 como POS, el 10, 14, y -3 en LH2 como raíces zerotree, y también los cuatro coeficientes en LH2 y los cuatro en HH2 como raíces zerotree. La segunda pasada dominante se detiene en este punto, ya que todos los otros coeficientes son conocidos por ser insignificantes desde la primera pasada dominante.

La lista subordinada contiene, en este punto, las seis magnitudes 63, 49, 34, 47, 31, y 23. Ellas representan los intervalos de 16 bits de ancho $[48, 64)$, $[32, 48)$, y $[16, 31)$. El codificador emite bits que definen un nuevo subintervalo para cada uno de los tres. Al final de la segunda pasada subordinada, el decodificador podría haber identificado el 34 y el 47 para ser de diferentes intervalos, por lo que las seis magnitudes se organizan como 63, 49, 47, 34, 31 y 23. El decodificador les asigna los valores refinados 60, 52, 44, 36, 28, y 20. (Fin del ejemplo.)

5.17. DjVu

Los métodos de compresión de imágenes se diseñan normalmente para un tipo de imagen. JBIG (Sección 4.11), por ejemplo, fue diseñado para imágenes binivel, el método de descomposición de bloques FABD (Sección 4.28) está diseñado para la compresión de imágenes de tonos discretos, y JPEG (Sección 4.8) funciona mejor con imágenes de tonos continuos. Ciertas imágenes, sin embargo, combinan las propiedades de los tres tipos de imágenes. Un ejemplo importante de dichas imágenes

es un documento escaneado que contiene texto, dibujos lineales, y regiones con imágenes en tonos continuos, tales como pinturas o fotografías. Las bibliotecas de todo el mundo están actualmente digitalizando sus posesiones escaneándolas y almacenándolas en formato comprimido en discos y CD-ROMs. Las organizaciones interesadas en hacer sus documentos disponibles para el público (tal como una empresa de pedidos por correo con un catálogo a todo color, o un instituto de investigación con artículos científicos) tienen también colecciones de documentos. Todos pueden beneficiarse de un método de compresión con pérdida eficiente que comprima dichos documentos. El ojeo de dicho documento se efectúa normalmente en un navegador web, por lo que este método debe ofrecer una rápida decodificación. Un método tal es DjVu (pronunciado[5] "déjà vu"), de los laboratorios AT&T ([ATT 96] y [Haffner et al. 98]). Comenzamos con un breve resumen de su eficiencia.

DjVu logra habitualmente factores de compresión tan altos como 1000 —que es de 5 a 10 veces mejor que los métodos de compresión de imágenes con los que compite—. Las páginas escaneadas a 300 dpi a todo color se comprimen típicamente de 25 Mb a 30–60 Kb con una excelente calidad. Las páginas en blanco y negro son aún más pequeñas, típicamente son comprimidas a 10 a 30 Kb. Ésto crea páginas escaneadas de alta calidad, cuyo tamaño es comparable, en promedio, al de una página HTML.

Para documentos con texto e imágenes en color, los archivos DjVu son típicamente 5–10 veces más pequeñas que los JPEG con una calidad similar. Para las páginas en blanco y negro, los archivos DjVu son típicamente 10–20 veces más pequeños que los JPEG y cinco veces más pequeños que los GIF. Los archivos DjVu También están cercanos a cinco veces más pequeños que los archivos PDF (Sección 8.13) producidos a partir del escaneo documentos.

Para ayudar a los usuarios a leer documentos DjVu comprimidos en un navegador web, los desarrolladores han implementado un decodificador en forma de un plug-in para navegadores web estándar. Con este decodificador (disponible gratuitamente para todas las plataformas populares), es fácil desplazarse por la imagen y hacer zoom sobre ella. El decodificador también utiliza poca memoria, típicamente 2 Mbytes de RAM para imágenes que normalmente requieren 25 Mbytes de RAM para mostrarlas completamente. El decodificador guarda la imagen en RAM en forma compacta, y descomprime, en tiempo real, sólo la parte que es realmente mostrada en la pantalla.

El método DjVu es progresivo. El espectador ve una versión inicial del documento muy rápidamente, y la calidad visual de la pantalla mejora progresivamente a medida que llegan más bits. Por ejemplo, el texto de una página de una revista típica aparece en sólo tres segundos con una conexión de módem de 56 Kbps. En uno o dos segundos, aparecen las primeras versiones de las imágenes y de los fondos. La versión final de la página, de máxima calidad, está completa tras unos pocos segundos más.

A continuación resumimos las principales características de DjVu. La idea principal es que los distintos elementos de un documento escaneado, a saber, textos, dibujos, e imágenes, tienen diferentes características perceptuales. El texto digitalizado y los dibujos lineales requieren una alta resolución espacial pero poca resolución de color. Imágenes y fondos, por otro lado, pueden ser codificados a menor resolución espacial, pero con más bits por píxel (para color de alta resolución). Sabemos por experiencia que el texto y los diagramas lineales deben ser escaneados y mostrados a 300 dpi o una mayor resolución, ya que cualquier texto en baja resolución es apenas legible y las líneas pierden sus bordes afilados. Además, el texto suele utilizar un solo color, y los dibujos utilizan varios colores. Las imágenes, por otro lado, se pueden escanear y visualizar a 100 dpi y sin mucha pérdida de detalle de imagen si los píxeles adyacentes pueden tener colores similares (i.e., si el número de colores disponibles es grande).

Por consiguiente, DjVu comienza descomponiendo el documento en tres componentes: máscara (*mask*), primer plano (*foreground*), y fondo (*background*). El componente fondo contiene los píxeles que constituyen las imágenes y el fondo del documento. La máscara contiene el texto y las líneas en

[5] "Déjà vu" es una expresión francesa para indicar que lo que estamos viendo ya lo hemos vivido en el pasado. La j se pronuncia como la y de yate (pero más suave), y la v es semejante a una mezcla entre f y b.

formato binivel (i.e., un bit por píxel). El primer plano contiene el color de la máscara de píxeles. El fondo es una imagen de tonos continuos y puede ser comprimida en la baja resolución de 100 dpi. El primer plano normalmente contiene grandes áreas uniformes y también se comprime como una imagen de tonos continuos en la misma baja resolución. La máscara se deja a 300 dpi, pero puede ser comprimida eficientemente, porque es binivel. El fondo y el primer plano son comprimidos con un método basado en wavelets llamado IW44 ("IW" procede de "integer wavelet" o "entero wavelet"), mientras que la máscara es comprimida con JB2, una versión de JBIG2 (Sección 4.12), desarrollado en AT&T.

El decodificador decodifica los tres componentes, aumenta la resolución de los componentes del fondo y del primer plano de nuevo a 300 dpi, y genera cada píxel de la imagen final descomprimida acorde con la máscara. Si un píxel de máscara es 0, el correspondiente píxel de la imagen es tomado del fondo. Si el píxel de la máscara es 1, el píxel correspondiente de la imagen es generado en el color del píxel de primer plano.

El resto de esta sección describe el método de separación de la imagen utilizada por DjVu. Éste es un algoritmo de agrupación bicolor multiescala basado en el concepto de *colores dominantes*. Imagine un documento que contiene sólo texto negro sobre un fondo blanco. Para obtener mejores resultados, este documento debe ser escaneado como una imagen en escala de grises, suavizada (*antialiased*). Ésto da como resultado una imagen que tiene la mayoría de píxeles en blanco y negro, pero también píxeles grises de varios tonos que se encuentran en y cercanos a los bordes de los caracteres de texto. Es obvio que los colores dominantes de tal imagen son el blanco y el negro. En general, dada una imagen con varios colores o tonos de gris, sus dos colores dominantes pueden ser identificados mediante el siguiente algoritmo:

- *Paso 1:* Inicializar el color de fondo b a blanco y el color de primer plano f a negro.
- *Paso 2:* Bucle sobre todos los píxeles de la imagen. Para cada píxel, calcular las distancias f y b entre el color del píxel y los colores de primer plano y de fondo actuales. Seleccionar la más corta de las dos distancias y marcar el píxel como f o b, como corresponda.
- *Paso 3:* Calcular el color promedio de todos los píxeles que están marcados con f. Éste se convierte en el nuevo color de primer plano. Hacer lo mismo con el color de fondo.
- *Paso 4:* Repetir los pasos 2 y 3 hasta que los dos colores dominantes converjan (i.e., hasta que varíen por menos que el valor de un umbral preestablecido).

Este Algoritmo es simple, y converge rápidamente. Sin embargo, un documento real, rara vez tiene dos colores dominantes, por lo que DjVu utiliza dos extensiones de este algoritmo. La primera extensión se denomina *agrupación bicolor de bloques* o *block bicolor clustering*. Ésta divide el documento en pequeños bloques rectangulares, y ejecuta el algoritmo de agrupación anterior en cada bloque para conseguir dos colores dominantes. Cada bloque es entonces separado en un primer plano y un fondo usando estos dos colores dominantes. Esta extensión no es del todo satisfactoria por los siguientes problemas que afectan al tamano del bloque:

1. Un bloque debe ser lo suficientemente pequeño como para tener un color de primer plano dominante. Imagine, por ejemplo, una palabra en rojo en una región de texto en negro. Idealmente, queremos que tal palabra esté en un bloque propio, con el rojo como uno de los colores dominantes del bloque. Si los bloques son grandes, esta palabra puede llegar a ser una pequeña parte de un bloque cuyos colores dominantes sean el blanco y el negro; el rojo efectivamente desaparecerá.
2. Por otro lado, un bloque pequeño puede estar ubicado completamente fuera de cualquier área de texto. Tal bloque contiene sólo los píxeles de fondo y no debe ser separado en fondo y primer plano. Un bloque también puede estar localizado completamente dentro de un carácter grande.

Tal bloque es todo primer plano y no debe ser separado. En cualquier caso, esta extensión del algoritmo de agrupación no encuentra significativamente colores dominantes.

3. La experiencia muestra que en un bloque pequeño este algoritmo no siempre selecciona colores correctos de primer plano y de fondo.

La segunda extensión se denomina *agrupación bicolor multiescala* o *multiscale bicolor clustering*. Éste es un algoritmo iterativo que comienza con una cuadrícula de grandes bloques y aplica el algoritmo de agrupación bicolor de bloques a cada uno de ellos. El resultado de esta primera iteración es un par de colores dominantes para cada bloque grande. En la segunda y subsiguientes iteraciones, la cuadrícula se divide en bloques cada vez más pequeños, y cada uno es procesado con el algoritmo de agrupación original. Sin embargo, este algoritmo se modifica ligeramente, ya que ahora atrae los nuevos colores de primer plano y de fondo de una iteración, no hacia el blanco y negro, sino hacia los dos colores dominantes que se encontraron en la iteración anterior. He aquí cómo funciona:

1. Para cada bloque pequeño b, identificar el bloque más grande B (de la iteración precedente) en cuyo interior se encuentra b. Inicializar los colores de fondo y de primer plano de b a los de B.

2. Bucle sobre la imagen completa. Comparar el color de un píxel con los colores de fondo y de primer plano de su bloque y etiquetar el píxel acorde con la más pequeña de los dos distancias.

3. Para cada bloque pequeño b, calcular un nuevo color de fondo haciendo un promedio de (1) los colores de todos los píxeles de b que han sido etiquetados como fondo, y (2) el color de fondo que se ha usado para b en esta iteración. Éste es un promedio ponderado donde a los píxeles de (1) se les da un peso del 80 %, y al color de fondo de (2) se le asigna un peso del 20 %. El nuevo color de primer plano se calcula de forma similar.

4. Los pasos 2 y 3 se repiten hasta que tanto el fondo como el primer plano convergen.

Posteriormente, el algoritmo de agrupación bicolor multiescala divide cada pequeño bloque b en bloques más pequeños y repite el proceso.

Este proceso suele tener éxito en la separación de la imagen en los componentes correctos de primer plano y de fondo. Para mejorarlo aún más, es seguido por una variedad de filtros que se aplican a distintas áreas del primer plano. Ellos están diseñados para encontrar y eliminar los errores más obvios en la identificación de las partes en primer plano.

déjà vu — Expresión francesa para "ya visto".
Vuja De — La sensación de que nunca has estado aquí.

5.18. WSQ, Compresión de huellas dactilares

La mayoría de nosotros no se da cuenta, pero las huellas dactilares (o digitales) son "un gran negocio". El FBI comenzó a recoger huellas en forma de impresiones entintadas en tarjetas de papel en 1924, y hoy tiene alrededor de 200 millones de tarjetas, que ocupa un acre[6] de archivadores en el edificio J. Edgar Hoover en Washington, D.C. (El FBI, al igual que muchos de nosotros, nunca

[6] Según la wikipedia, un acre son 40,4687260987 áreas; un área son 100 m^2. (El diccionario de la RAE lo define como: "Medida inglesa de superficie equivalente a 40 áreas y 47 centiáreas.")

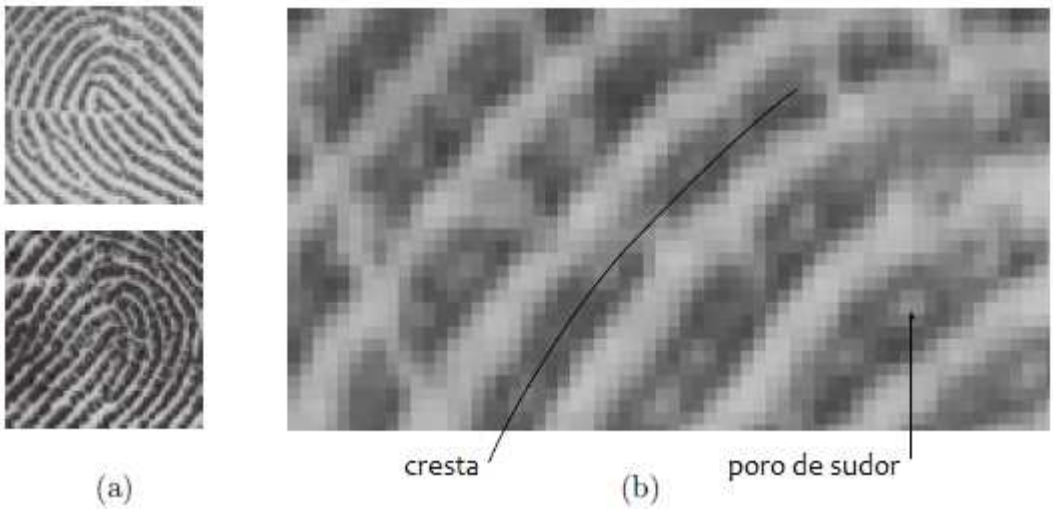


Figura 5.64: Ejemplos de huellas dactilares escaneadas (cortesía de Christopher Brislawn)

tira nada. También tienen muchos "clientes habituales", por lo que "sólo" unos 29 millones de los 200 millones de tarjetas son distintas; son las que se utilizan para efectuar las comprobaciones de antecedentes.)

Lo que es más, estas tarjetas se van acumulando a una tasa de 30 000–50 000 nuevas tarjetas por día (¡ésto es por día, no por año!). Hay una clara necesidad de digitalizar la colección, por lo que ocupará menos espacio y se prestará a la búsqueda y la clasificación automática. El problema principal es el tamaño (en bits). Cuando una tarjeta de huellas digitales típica es escaneada a 500 dpi, con ocho $^{\text{bits}}/_{\text{píxel}}$, se traduce en aproximadamente 10 Mb de datos. En consecuencia, el tamaño total de la colección digitalizada sería de más de 2000 terabytes (un terabyte son 2^{40} bytes); enormes incluso para los estándares actuales (2006).

◇ **Ejercicio 5.18 (sol. en pág. 1104):** Aplíquense estos números para calcular el tamaño de una tarjeta de huellas dactilares.

Por lo tanto, la compresión es una necesidad. En un primer momento, parece que la compresión de la huella dactilar debe ser sin pérdidas a causa de los pequeños pero importantes detalles involucrados. Sin embargo, los métodos de compresión de imágenes sin pérdidas producen razones de compresión típicas de 0,5, mientras que para para hacer una seria disminución en la enorme cantidad de datos en esta colección, son necesarias compresiones de alrededor de 1 bpp ó mejor. Lo que se necesita es un método de compresión con pérdida que dé lugar a la degradación progresiva de los datos de las imágenes, y no introduzca ningún artefacto en la imagen reconstruida. La mayoría de los métodos de compresión de imágenes con pérdidas implican la pérdida de los pequeños detalles y, por lo tanto, son inaceptables, ya que los pequeños detalles de identificación, tales como los poros de sudor, son puntos de identificación admisibles en los tribunales. Es aquí donde las wavelets entran en juego. La compresión wavelet con pérdidas, si se diseña cuidadosamente, puede satisfacer los criterios anteriores y dar lugar a la compresión eficiente, donde los pequeños detalles importantes son conservados o, por lo menos, son identificables. La Figura 5.64a,b (obtenida, con autorización, de Christopher M. Brislawn), muestra dos ejemplos de huellas dactilares y un detalle, cuando las crestas y los poros del sudor pueden verse claramente.

La compresión también es necesaria, ya que las imágenes de huellas dactilares son enviadas rutinariamente entre las fuerzas del orden. La entrega más rápida de la tarjeta real es demasiado lenta y

Toma (*tap*)	Valor exacto	Valor aproximado
$h_0(0)$	$-5\sqrt{2}x_1\left(48\left\|x_2\right\|^2-16\mathcal{R}x_2+3\right)/32$	0,852698790094000
$h_0(\pm1)$	$-5\sqrt{2}x_1\left(8\left\|x_2\right\|^2-\mathcal{R}x_2\right)/8$	0,377402855612650
$h_0(\pm2)$	$-5\sqrt{2}x_1\left(4\left\|x_2\right\|^2+4\mathcal{R}x_2-1\right)/16$	−0,110624404418420
$h_0(\pm3)$	$-5\sqrt{2}x_1(\mathcal{R}x_2)/8$	−0,023849465019380
$h_0(\pm4)$	$-5\sqrt{2}x_1/64$	0,037828455506995
$h_1(-1)$	$\sqrt{2}(6x_1-1)/16x_1$	0,788485616405660
$h_1(-2,0)$	$-\sqrt{2}(16x_1-1)/64x_1$	−0,418092273222210
$h_1(-3,1)$	$\sqrt{2}(2x_1+1)/32x_1$	−0,040689417609558
$h_1(-4,2)$	$-\sqrt{2}/64x_1$	0,064538882628938

Tabla 5.65: Coeficientes de filtro wavelet simétricos para la WSQ.

arriesgada (no hay copia de seguridad de las tarjetas), y el envío de 10 Mb de datos a través de un módem de 9600 baudios toma alrededor de tres horas.

El método descrito aquí [Bradley, Brislawn y Hopper 93] ha sido adoptado por el FBI como su estándar para la compresión de huellas dactilares [Federal Bureau of Investigations 93]. Involucra tres etapas: (1) una transformada wavelet discreta, (2) una cuantificación escalar adaptativa de los coeficientes de la transformada wavelet, y (3) una codificación de Huffman de dos pasadas de los índices de cuantificación. Ésta es la razón del nombre *cuantificación wavelet/escalar*, (*wavelet/scalar quantization* o WSQ). El método normalmente produce factores de compresión de alrededor de 20. La decodificación es la opuesta a la codificación, por lo que WSQ es un método de compresión simétrica.

El primer paso es una transformada wavelet discreta simétrica (SWT) que utiliza los coeficientes de filtro simétricos mostrados en la Tabla 5.65 (donde $\mathcal{R}$ indica la parte real de un número complejo). Son filtros simétricos con siete y nueve tomas (*taps*) de respuesta de impulso, y que dependen de los dos números x_1 (real) y x_2 (complejo). El estándar final aprobado por el FBI utiliza los valores:

$$x_1 = A + B - \frac{1}{6}, \quad x_2 = \frac{-(A+B)}{2} - \frac{1}{6} + \frac{i\sqrt{3}(A-B)}{2},$$

donde

$$A = \left(\frac{-14\sqrt{15}+63}{1080\sqrt{15}}\right)^{1/3}, \quad \text{y } B = \left(\frac{-14\sqrt{15}-63}{1080\sqrt{15}}\right)^{1/3}.$$

La descomposición de imágenes wavelet es distinta de la descrita en la Sección 5.10. Se le puede llamar simétrica y se muestra en la Figura 5.66. La SWT se aplica primero a las filas y columnas de la imagen, produciendo $4\times4=16$ subbandas. A continuación, se aplica la SWT de la misma manera a tres de las 16 subbandas, descomponiendo cada una en 16 subbandas más pequeñas. El último paso consiste en descomponer la subbanda superior izquierda en cuatro más pequeñas.

Las subbandas más grandes (51–63) contienen la información de alta frecuencia, los detalles finos de la imagen. Posteriormente, éstos pueden ser fuertemente cuantificados sin pérdida de información importante (i.e., la información necesaria para clasificar e identificar las huellas dactilares). De hecho, las subbandas 60–63 son descartadas por completo. Las subbandas 7–18 son importantes. Contienen esa parte de las frecuencias de la imagen correspondiente a las crestas en una huella dactilar. Esta información es importante y debe ser cuantificada levemente.

Los coeficientes de transformación en las 64 subbandas son números en coma flotante denotados por a. Ellos son cuantificados en una cantidad finita de números en coma flotante que son denotados

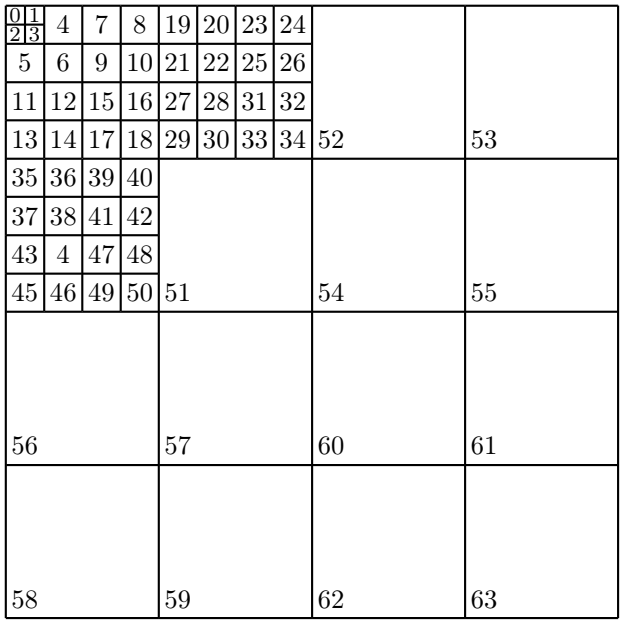


Figura 5.66: Descomposición wavelet simétrica de imágenes.

por $\hat{a}$. El codificador WSQ asigna un coeficiente de transformada a a un índice de cuantificación p (un entero que luego es mapeado a un código que es en sí mismo una codificación de Huffman). El índice p puede ser considerado como un puntero a un recipiente de cuantificación, donde se encuentra a. El decodificador WSQ recibe un índice p y lo mapea a un valor $\hat{a}$ que es cercano, pero no idéntico, a a. Así es cómo WSQ pierde la información de la imagen. El conjunto de valores $\hat{a}$ es un conjunto discreto de números en coma flotante llamados *coeficientes wavelet cuantificados*. La cuantificación depende de parámetros que pueden variar de una subbanda a otra, ya que distintas subbandas tienen diferentes requisitos de cuantificación.

La Figura 5.67 muestra la configuración de los recipientes de cuantificación para la subbanda k. El parámetro Z_k es la anchura del recipiente del cero, y el parámetro Q_k es la anchura de los otros contenedores. El parámetro C está en el rango $[0, 1]$. Éste determina el valor reconstruido $\hat{a}$. Para $C = 0{,}5$, por ejemplo, el valor reconstruido para cada recipiente de cuantificación es el centro del

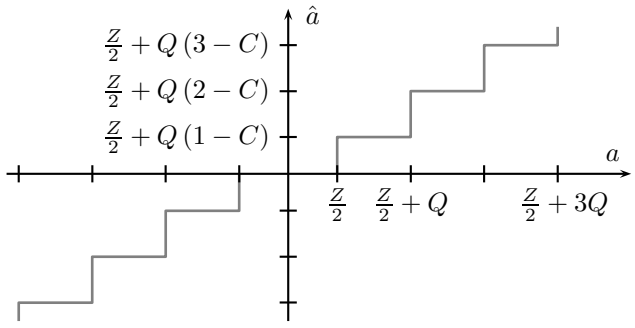


Figura 5.67: Cuantificación escalar WSQ.

contenedor. La Ecuación (5.29) muestra cómo son utilizados los parámetros Z_k y Q_k por el codificador WSQ para cuantificar un coeficiente de transformada $a_k(m,n)$ (i.e., un coeficiente en la posición (m,n) en la subbanda k) a un índice $p_k(m,n)$ (un entero), y cómo calcula el decodificador WSQ un coeficiente cuantificado $\hat{a}_k(m,n)$ a partir de ese índice:

$$p_k(m,n) = \begin{cases} \left\lfloor \frac{a_k(m,n)-Z_k/2}{Q_k} \right\rfloor + 1, & a_k(m,n) > Z_k/2, \\ 0 & -Z_k/2 \le a_k(m,n) \le Z_k/2, \\ \left\lceil \frac{a_k(m,n)+Z_k/2}{Q_k} \right\rceil + 1 & a_k(m,n) < -Z_k/2, \end{cases} \tag{5.29}$$

$$\hat{a}_k(m,n) = \begin{cases} (p_k(m,n) - C)\, Q_k + Z_k/2, & p_k(m,n) > 0, \\ 0, & p_k(m,n) = 0, \\ (p_k(m,n) + C)\, Q_k - Z_k/2, & p_k(m,n) < 0. \end{cases}$$

El estándar final aprobado por el FBI utiliza el valor $C = 0{,}44$ y determina la anchura Q_k y Z_k de los recipientes a partir de las varianzas de los coeficientes en las distintas subbandas siguiendo estos pasos:

- *Paso 1:* Denotamos la anchura y la altura de la subbanda k por X_k y Y_k, respectivamente. Calculamos las seis cantidades:

$$W_k = \left\lfloor \frac{3X_k}{4} \right\rfloor, \quad H_k = \left\lfloor \frac{7Y_k}{16} \right\rfloor,$$
$$x_{0k} = \left\lfloor \frac{X_k}{8} \right\rfloor, \quad x_{1k} = x_{0k} + W_k - 1,$$
$$y_{0k} = \left\lfloor \frac{9Y_k}{32} \right\rfloor, \quad y_{1k} = y_{0k} + H_k - 1.$$

- *Paso 2:* Asumiendo que la posición $(0,0)$ es la esquina superior izquierda de la subbanda, usamos la región desde la posición (x_{0k}, y_{0k}) hasta la posición (x_{1k}, y_{1k}) de la misma para estimar la varianza σ_k^2 de la subbanda mediante:

$$\sigma_k^2 = \frac{1}{W \cdot H - 1} \sum_{n=x_{0k}}^{x_{1k}} \sum_{m=y_{0k}}^{y_{1k}} \left(a_k(m,n) - \mu_k\right)^2,$$

  donde μ_k denota el promedio de $a_k(m,n)$ en la región.

- *Paso 3:* Parámetro Q_k se calcula mediante

$$qQ_k = \begin{cases} 1, & 0 \le k \le 3, \\ \frac{10}{A_k \log_e(\sigma_k^2)}, & 4 \le k \le 59, \text{ y } \sigma_k^2 \ge 1{,}01, \\ 0, & 60 \le k \le 63, \text{ ó } \sigma_k^2 < 1{,}01, \end{cases}$$

  donde q es una constante de proporcionalidad que controla la anchura de los recipientes Q_k y con ello el nivel general de compresión. El procedimiento para el cálculo de q es complejo y no

Código	Índice o run length
1	run length de 1 cero
2	run length de 2 ceros
3	run length de 3 ceros
⋮	
100	run length de 100 ceros
101	código de escape para índices positivos de 8 bits
102	código de escape para índices negativos de 8 bits
103	código de escape para índices positivos de 16 bits
104	código de escape para índices negativos de 16 bits
105	código de escape para racha (run) de ceros, 8 bits
106	código de escape para racha (run) de ceros, 16 bits
107	valor de índice -73
108	valor de índice -72
109	valor de índice -71
⋮	
179	valor de índice -1
180	sin uso
181	valor de índice 1
⋮	
253	valor de índice 73
254	valor de índice 74

Tabla 5.68: Códigos WSQ para índices de cuantificación y run lengths.

será descrito aquí. Los valores de las constantes A_k son:

$$A_k = \begin{cases} 1{,}32, & k = 52,\, 56, \\ 1{,}08, & k = 53,\, 58, \\ 1{,}42, & k = 54,\, 57, \\ 1{,}08, & k = 55,\, 59, \\ 1, & \text{en otro caso.} \end{cases}$$

Nótese que los anchos de recipiente para las subbandas 60–63 son cero. Como resultado, estas subbandas, que contienen los coeficientes de detalle más finos, son simplemente descartados.

- *Paso 4:* La anchura de recipiente cero se establece en $Z_k = 1{,}2Q_k$.

El codificador WSQ calcula los índices de cuantificación $p_k\,(m,n)$ tal como se muestra, después los mapea a los 256 códigos listados en la Tabla 5.68. Estos valores están codificados con códigos de Huffman (mediante un proceso de dos pasadas), y los códigos de Huffman son escritos en el stream comprimido. Un índice de cuantificación $p_k\,(m,n)$ puede ser cualquier número entero, pero la mayoría son pequeños y no hay muchos ceros. En consecuencia, los códigos de la Tabla 5.68 se dividen en tres grupos. El primer grupo consiste en 100 códigos (códigos 1 a 100) para run lengths de 1 a 100 índices cero. El segundo grupo está formado por los códigos, de 107 a 254. Éstos especifican índices pequeños, en el rango $[-73, +74]$. El tercer grupo se compone de los seis códigos *de escape* 101 a 106. Ellos indican índices grandes o rachas (run lengths) de más de 100 índices cero. El código 180 (el cual corresponde a un índice $p_k\,(m,n) = 0$) no se utiliza, porque este caso es en realidad un run length

de un único cero. Un código de escape es seguido por el valor en bruto (8 ó 16 bits) del índice (o el tamaño del run length). He aquí algunos ejemplos:

Un índice $p_k\,(m,n) = -71$ se codifica como 109. Un índice $p_k\,(m,n) = -1$ se codifica como 179. Un índice $p_k\,(m,n) = 75$ se codifica como 101 (código de escape para los índices positivos de 8 bits) seguido por 75 (en ocho bits). Un índice de $p_k\,(m,n) = -259$ se codifica como 104 (código de escape para índices negativos grandes) seguido por 259 (el valor absoluto del índice, en 16 bits). Un índice cero aislado se codifica como 1, y un run length de 260 ceros se codifica como 106 (código de escape para run lengths grandes) seguido por 260 (en 16 bits). Los índices o run lengths que requieran más de 16 bits no pueden ser codificados, pero la particular elección de los parámetros de cuantificación y la transformada wavelet prácticamente garantizan que no se generarán índices grandes.

> O'Day pensó que eso era más de lo que él había tenido derecho a esperar dadas las circunstancias. Una identificación de huellas dactilares normalmente requiere de diez puntos individuales —las irregularidades que constituyen el arte de la identificación de huellas dactilares— pero ese número siempre era arbitrario. El inspector estaba seguro de que Cutter había manejado este disco de ordenador, incluso si un jurado no pudiera estar completamente seguro, si ese momento llegara.
>
> —Tom Clancy, *Peligro inminente*

El último paso consiste en preparar las tablas de códigos de Huffman. Éstas dependen de la imagen, por lo que tienen que ser escritos en el stream comprimido. El estándar adoptado por el FBI especifica que las subbandas se pueden agrupar en tres bloques y todas las subbandas en un grupo utilizan la misma tabla de códigos de Huffman. Ésto facilita la transmisión progresiva de la imagen. El primer bloque está formado por las subbandas de baja y media frecuencia 0–18. El segundo y tercer bloques contienen las subbandas de detalle paso alto 19–51 y 52–59, respectivamente (recordemos que las subbandas 60–63 se han descartado completamente). Se preparan las dos tablas de códigos de Huffman, una para el primer bloque y la otra para el segundo y tercer bloques.

Una tabla de código de Huffman para un bloque de subbandas se prepara contando del número de veces que aparece en el bloque cada uno de los 254 códigos de la Tabla 5.68. Los conteos se utilizan para determinar la longitud de cada código y para construir el árbol de código de Huffman. Éste es un trabajo de dos etapas (una para determinar las tablas de códigos y otra para codificar) y se efectúa de una manera similar cómo utiliza el código de Huffman JPEG (Sección 4.8.4).

5.19. JPEG 2000

Esta sección fue escrita a mediados del 2000 y ha sido mejorada ligeramente a principios del 2003.

El campo de la compresión de datos es muy activo; nuevos enfoques, ideas y técnicas están siendo desarrolladas e implementadas continuamente. JPEG (Sección 4.8) se utiliza ampliamente para la compresión de imágenes, pero no es perfecto. El uso de la DCT (transformada discreta del coseno) en bloques de 8×8 píxeles produce en ocasiones una imagen reconstruida que tiene apariencia de bloques (especialmente cuando los parámetros de JPEG han sido fijados para una gran pérdida de información). Por ello, el comité de JPEG decidió, hacia 1995, desarrollar un nuevo estándar para la compresión de imágenes estáticas basado en wavelets, que se conoce como JPEG 2000 (o JPEG Y2K). Quizás el hito más importante en el desarrollo de JPEG 2000 se produjo en diciembre 1999, cuando el comité JPEG se reunió en Maui, Hawái, y aprobó el primer borrador del comité sobre la parte 1 del estándar JPEG 2000. En su reunión de Rochester, en agosto del 2000, el comité JPEG aprobó la versión final de este estándar internacional. En diciembre del 2000 este borrador fue finalmente aceptado como un estándar internacional completo por la ISO y la UIT-T. Lo siguiente es una lista de las áreas donde se espera que este nuevo estándar mejore los métodos existentes:

- Alta eficiencia de compresión. Se esperan ratios de bits de menos de 0,25 bpp para imágenes en escala de grises altamente detalladas.
- Capacidad de manejar imágenes grandes, de hasta $2^{32} \times 2^{32}$ píxeles (el JPEG original, puede manejar imágenes de hasta $2^{16} \times 2^{16}$).
- Transmisión de imagen progresiva (Sección 4.10). El estándar propuesto puede descomprimir una imagen progresivamente mediante SNR, resolución, componente de color, o región de interés.
- De acceso fácil y de rápido a distintos puntos del stream comprimido.
- El decodificador puede efectuar desplazamientos/zooms en la imagen, mientras descomprime sólo partes de la misma.
- El decodificador puede rotar y recortar la imagen, mientras la descomprime.
- Capacidad de recuperación de errores. Pueden incluirse códigos correctores de errores en el stream comprimido, para mejorar la fiabilidad de la transmisión en ambientes ruidosos.

Las principales fuentes de información sobre JPEG 2000 son [JPEG 00] y [Taubman y Marcellin 02]. Esta sección, sin embargo, está basada en [ISO/IEC 00], el último borrador del comité (FCD o final committee draft), lanzado en marzo del 2000. Este documento define el flujo de datos (stream) comprimido (referido como *bitstream*) y las operaciones del decodificador. Contiene secciones informativas sobre el codificador, pero cualquier codificador que produzca un bitstream válido es considerado un codificador JPEG 2000 válido.

Uno de los nuevos e importantes enfoques introducidos por la compresión JPEG 2000 es el paradigma "comprimir una vez, descomprimir de muchas maneras". El codificador JPEG 2000 selecciona una calidad de imagen máxima Q y una resolución máxima R, y comprime una imagen utilizando estos parámetros. El decodificador puede descomprimir la imagen en cualquier calidad inferior o igual a Q y en cualquier resolución menor o igual que R. Supongamos que una imagen I fue comprimida en B bits. El decodificador puede extraer A bits del stream comprimido (donde $A < B$) y produce una imagen descomprimida con pérdidas que será idéntica a la imagen que se obtendría si I hubiera sido originalmente comprimida con pérdidas en A bits.

En general, el decodificador puede descomprimir la imagen completa en baja calidad y/o menor resolución. También puede descomprimir partes de la imagen (regiones de interés) en cualquier calidad y resolución, máxima o inferior. Incluso más, el decodificador puede *extraer* partes del stream comprimido y ensamblarlas para crear un nuevo stream comprimido sin tener que hacer descompresión alguna. En consecuencia, una imagen de baja resolución y/o de calidad inferior puede ser creada sin que el decodificador tenga que descomprimir nada. Las ventajas este enfoque son: (1) ahorra tiempo y espacio, y (2) previene la acumulación de ruido de imagen, común en los casos donde una imagen es comprimida y descomprimida con pérdidas varias veces.

JPEG 2000 también permite recortar y transformar la imagen. Cuando una imagen es comprimida en un principio, se pueden especificar varias regiones de interés. El decodificador puede acceder a los datos comprimidos de cualquier región y escribirlos como un nuevo stream comprimido. Ésto exige algún tipo de procesamiento especial en torno a los bordes de la región, pero no es necesario descomprimir toda la imagen y volver a comprimir la región deseada. Además, el reflejo (volteado o *flipping*) de la imagen o la rotación de 90°, 180°, 270° pueden llevarse a cabo casi enteramente en el stream comprimido sin descompresión.

La imagen progresiva también está soportada por JPEG 2000 y tiene cuatro aspectos: calidad, resolución, ubicación espacial y componente.

A medida que el decodificador introduce más datos del stream comprimido mejora la calidad de la imagen visualizada. Una imagen normalmente se hace reconocible después de sólo haber sido

introducidos y procesados 0,05 bits/píxel por el decodificador. Después de introducirse y procesarse 0,25 bits/píxel, la imagen parcialmente descomprimida aparece suave, con sólo artefactos de compresión sin importancia. Cuando el decodificador comienza, utiliza los primeros bytes de datos de entrada para crear una pequeña versión (una miniatura) de la imagen. A medida que son leídos y procesados más datos, el decodificador añade más píxeles a la imagen visualizada, aumentando de este modo su tamaño, en etapas. Cada etapa aumenta la imagen en un factor de 2 en cada lado. Así es cómo la resolución es incrementada progresivamente. Este modo de JPEG 2000 corresponde aproximadamente al modo jerárquico de JPEG.

La ubicación espacial significa que la imagen puede visualizarse progresivamente en orden *raster* (de arriba abajo y de izquierda a derecha), fila por fila. Ésto es útil cuando la imagen tiene que ser descomprimida y enviada inmediatamente a una impresora. La progresión de componentes tiene que ver con los componentes de color de la imagen. La mayor parte de las imágenes tienen uno o tres componentes de color; pero también son comunes cuatro componentes, como en CMYK. JPEG 2000 permite hasta 16 384 componentes. Los componentes extra pueden ser superposiciones que contienen texto y gráficos. Si el stream comprimido es preparado para la progresión por componentes, entonces la imagen es decodificada primero como escala de grises, luego como una imagen a color, y a continuación, se descomprimen las diversas superposiciones y se visualizan, para agregar realismo y detalle.

Los cuatro aspectos de la progresión pueden convivir en un único stream comprimido. Un stream en particular puede prepararse por el codificador para crear, por ejemplo, el siguiente resultado. El inicio del stream puede contener datos suficientes para mostrar una pequeña imagen en escala de grises. Los datos que siguen pueden añadir color a esta pequeña imagen, seguidos por los datos que aumentan la resolución varias veces, seguidos, a su vez, por un incremento de la calidad hasta que el usuario decide que la calidad es suficientemente alta para permitir la impresión de la imagen. En ese punto, la resolución puede ser aumentada para que coincida con la de la impresora disponible. Si la impresora es en blanco y negro, la información de color en el resto de la imagen comprimida, puede ser omitida. Todo ésto puede efectuarse mientras que sólo se necesitan ser introducidos y descomprimidos los datos requeridos por el usuario. Independientemente del tamaño original del stream comprimido, la razón de compresión es incrementada eficazmente cuando sólo una parte del stream comprimido necesita ser introducido y procesado.

¿Cómo funciona JPEG 2000? El párrafo siguiente es un breve resumen del algoritmo. Ciertos pasos se describen en más detalle en el resto de esta sección.

Si la imagen que está siendo comprimida es en color, se divide en tres componentes. Cada componente es particionado en regiones rectangulares no superpuestas llamadas teselas (*tiles*), que son comprimidas de forma individual. Una tesela se comprime en cuatro pasos principales. El primer paso es calcular una transformada wavelet que produce las subbandas de coeficientes wavelet. Dos de tales transformaciones, una entera y una de punto flotante, se especifican en el estándar. Hay $L+1$ niveles de resolución de subbandas, donde L es un parámetro determinado por el codificador. En el segundo paso, los coeficientes wavelet son cuantificados. Ésto se realiza si el usuario especifica como objetivo un ratio de bits. Cuanto menor sea el ratio de bits, más toscamente deben ser cuantificados los coeficientes wavelet. El tercer paso utiliza el codificador MQ (un codificador similar al codificador QM, Sección 2.16) para codificar aritméticamente los coeficientes wavelet. El algoritmo EBCOT [Taubman 99] ha sido adoptado para la etapa de codificación. El principio de EBCOT es dividir cada subbanda en bloques (denominados *bloques de código*) que son codificados de forma individual. Los bits resultantes de la codificación de varios bloques de código se convierten en un *paquete* y los paquetes son los componentes del bitstream. El último paso consiste en construir el bitstream. Este paso coloca los paquetes, así como muchos *marcadores*, en el bitstream. Los marcadores pueden ser utilizados por el decodificador para omitir ciertas áreas del bitstream y llegar a ciertos puntos rápidamente. Usando marcadores, el decodificador puede, e.g., descifrar ciertos bloques de código antes que otros, visualizando así ciertas regiones de la imagen antes que otras. Otro uso de los marcadores es para que el decodificador decodifique la imagen progresivamente en una de varias maneras. El bitstream se orga-

niza en *capas* (*layers*), donde cada capa contiene información de la imagen de mayor resolución. En consecuencia, la decodificación la imagen capa por capa es una forma natural de lograr la transmisión y descompresión progresiva de una imagen.

Antes de entrar en detalle, seguidamente narramos una breve historia de los esfuerzos de desarrollo de JPEG 2000. La historia de JPEG 2000 se inicia en 1995, cuando Ricoh Inc. presentó el algoritmo CREW (compresión con wavelets embebidas reversibles, Sección 5.15) para la ISO/IEC como un candidato a JPEG-LS (Sección 4.9). CREW no fue seleccionado como el algoritmo para JPEG-LS, pero era lo suficientemente avanzado como para ser considerado un candidato para el nuevo método que estaba siendo examinado por la ISO/IEC. Este método, que más tarde sería conocido como JPEG 2000, fue aprobado como un nuevo elemento de trabajo oficial, y fue creado un grupo de trabajo (WG1) para él en 1996. En marzo de 1997, WG1 solicitó propuestas y comenzó la evaluación de las mismas. De los muchos algoritmos presentados, el método WTCQ (wavelet/trellis-coded quantization o wavelet/cuantificación de retículas codificadas) ofreció el mejor rendimiento y fue elegido en noviembre de 1997 como el algoritmo de referencia de JPEG 2000. El algoritmo WTCQ incluye una transformada wavelet y un método de cuantificación.

En noviembre de 1998, el algoritmo EBCOT fue presentado al grupo de trabajo por su creador David Taubman y fue adoptado como el método para codificar los coeficientes wavelet. En marzo de 1999, el codificador MQ se presentó al grupo de trabajo y fue adoptado por él como el codificador aritmético para ser utilizado en JPEG 2000. Durante 1999, el formato del bitstream fue desarrollado y probado, con el resultado de que a finales de 1999 todos los componentes principales de JPEG 2000 estaban en su lugar. En diciembre de 1999, el grupo de trabajo presentó su borrador de comité[7] (CD), y en marzo del 2000, emitió su borrador de comité definitivo[8] (FCD), el documento en que se basa esta sección. En agosto del 2000, el grupo se reunió JPEG y decidió aprobar el FCD como un estándar internacional completo en diciembre del 2000. Esta norma se conoce ahora como ISO/IEC-15444, y su descripción formal está disponible (en 13 partes, de las cuales la parte 7 ha sido abandonada) en la ISO, la UIT-T, y en ciertas organizaciones nacionales de normalización.

En esta sección se continúa con los detalles de algunos convenios y operaciones de JPEG 2000. El objetivo es aclarar los conceptos clave con el fin de dar al lector una comprensión general de este nuevo estándar internacional.

Componentes de color: Una imagen en color se compone de tres componentes de color. El primer paso del codificador JPEG 2000 es transformar los componentes, ya sea por medio de una transformación de componentes reversible (RCT), ya sea mediante una transformación de componentes irreversible (ICT). Cada componente transformado es posteriormente comprimido por separado.

Si los píxeles de la imagen tienen valores sin signo (que es el caso normal), entonces la transformación de componentes (o bien ECA, o bien ICT) es precedida por un nivel de desplazamiento DC. Este proceso traslada los valores de todos los píxeles desde su intervalo sin signo original $[0, 2^s - 1]$ (donde s es la profundidad de los píxeles) hasta el intervalo con signo $\left[-2^{s-1}, 2^{s-1} - 1\right]$ restando 2^{s-1} de cada valor. Para $s = 4$, e.g., los $2^4 = 16$ valores de píxel posibles se transforman desde el intervalo $[0, 15]$ hasta el intervalo $[-8, +7]$ restando $2^{4-1} = 8$ a cada valor.

La RCT es una transformación de descorrelación. Sólo se puede utilizar con la transformada wavelet entera (que es reversible). Denotando los valores de píxel del componente i de la imagen (después de un posible nivel de desplazamiento DC) por $I_i(x, y)$ para $i = 0, 1$, y 2, la RCT produce nuevos valores

[7] CD proviene de *Committee Draft*.
[8] FCD proviene de *Final Committee Draft*.

$Y_i(x,y)$ de acuerdo con:

$$Y_0(x,y) = \left\lfloor \frac{I_0(x,y) + 2I_1(x,y) + I_2(x,y)}{4} \right\rfloor,$$
$$Y_1(x,y) = I_2(x,y) - I_1(x,y),$$
$$Y_2(x,y) = I_0(x,y) - I_1(x,y).$$

Observe que los valores de los componentes Y_1 e Y_2 (pero no Y_0) requieren un bit más que los valores originales I_i.

La ICT es también una transformación descorrelación. Sólo se puede utilizar con la transformada wavelet de punto flotante (que es irreversible). Las ICT se define mediante:

$$Y_0(x,y) = 0{,}299I_0(x,y) + 0{,}587I_1(x,y) + 0{,}144I_2(x,y),$$
$$Y_1(x,y) = -0{,}16875I_0(x,y) - 0{,}33126I_1(x,y) + 0{,}5I_2(x,y),$$
$$Y_2(x,y) = 0{,}5I_0(x,y) - 0{,}41869I_1(x,y) - 0{,}08131I_2(x,y).$$

Si los componentes originales de la imagen son rojo, verde, y azul, entonces la ICT es muy similar a la representación de color YCbCr (Sección 6.2).

Teselas: Cada componente de color (transformado mediante RCT o ICT) de la imagen es particionado en teselas rectangulares, no superpuestas. Dado que los componentes de color pueden tener distintas resoluciones, pueden usar tamaños diferentes de teselas. Las teselas pueden tener cualquier tamaño, inferior o igual al de la imagen completa (i.e., una tesela). Todas las teselas de un componente de color dado tienen el mismo tamaño, excepto aquellos en los bordes. Cada tesela se comprime de forma individual.

La Figura 5.69a muestra un ejemplo de teselado de imagen. JPEG 2000 permite que la imagen tenga un desplazamiento vertical desde la parte superior y un desplazamiento horizontal desde la izquierda (los desplazamientos pueden ser cero). El origen de la rejilla de teselado (la esquina superior izquierda) puede ubicarse en cualquier lugar dentro del área de intersección de los dos desplazamientos. Todas las teselas en la rejilla tienen el mismo tamaño, pero las situadas en los bordes, tales como $T0$, $T4$ y $T19$ en la figura, deben ser truncadas. Las teselas se numeran en orden raster.

La razón principal para tener teselas es permitir al usuario decodificar partes de la imagen (regiones de interés). El decodificador puede identificar cada tesela en el bitstream y descomprimir sólo los píxeles incluidos en la tesela. La Figura 5.69b muestra una imagen con una relación de aspecto (altura/anchura) de 16 : 9 (la relación de aspecto de la televisión de alta definición, HDTV; Sección 6.3.1). La imagen está teselada con cuatro teselas cuya relación de aspecto es de 4 : 3 (la relación de aspecto normal de la televisión analógica), de manera que la tesela $T0$ cubre el área central de la imagen. Ésto hace que sea fácil recortar la imagen desde la relación de aspecto original al ratio 4 : 3.

Transformada wavelet: Dos transformadas wavelet han sido especificadas por el estándar: la wavelet $(9,7)$ de punto flotante (irreversible) y la wavelet $(5,3)$ entera (reversible). Cualquiera de ellas permite la transmisión progresiva, pero sólo la transformada entera puede producir compresión sin pérdidas.

Denotamos una fila de píxeles en una tesela como $P_k, P_{k+1}, \ldots, P_m$. Debido a la naturaleza de la transformada wavelet usada por JPEG 2000, puede ser necesario utilizar unos pocos píxeles con índices menores que k o mayores que m. Por consiguiente, antes de calcular cualquier transformada wavelet para una tesela, los píxeles de la misma pueden tener que ser ampliados. El estándar JPEG 2000 especifica un método simple de extensión denominado *extensión simétrica periódica*, que se ilustra en la Figura 5.70. La figura muestra una fila de siete símbolos "ABCDEFG" y cómo éstos son reflejados hacia la izquierda y hacia la derecha para extender la fila l y r símbolos, respectivamente. La Tabla 5.71 muestra los valores mínimos de l y r como funciones de la paridad de k y m y de la transformada en particular utilizada.

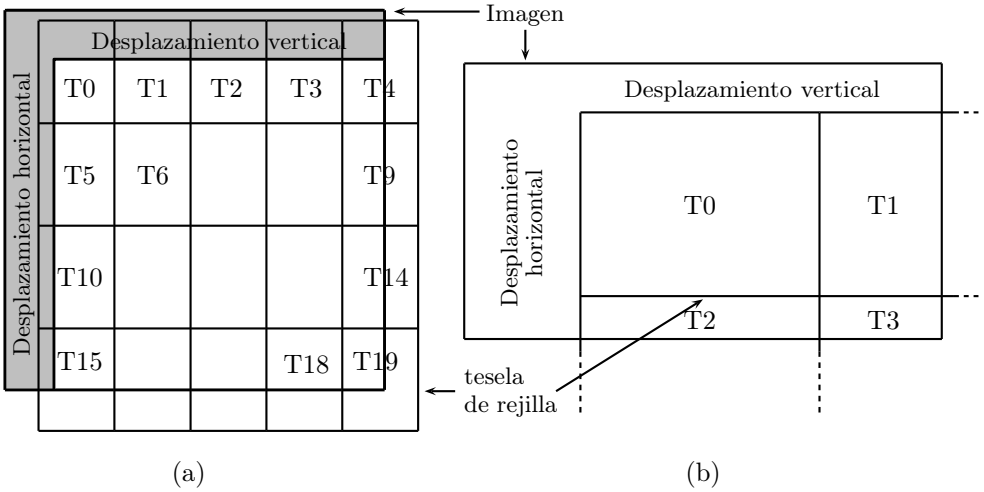


Figura 5.69: Enrejado de imagen en JPEG 2000.

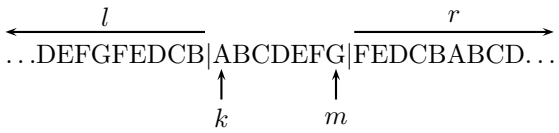


Figura 5.70: Extensión de una fila de píxeles.

k	$l\,(5,3)$	$l\,(9,7)$	m	$r\,(5,3)$	$r\,(9,7)$
par	2	4	impar	2	4
impar	1	3	par	1	3

Tabla 5.71: Extensiones izquierda y derecha mínimas.

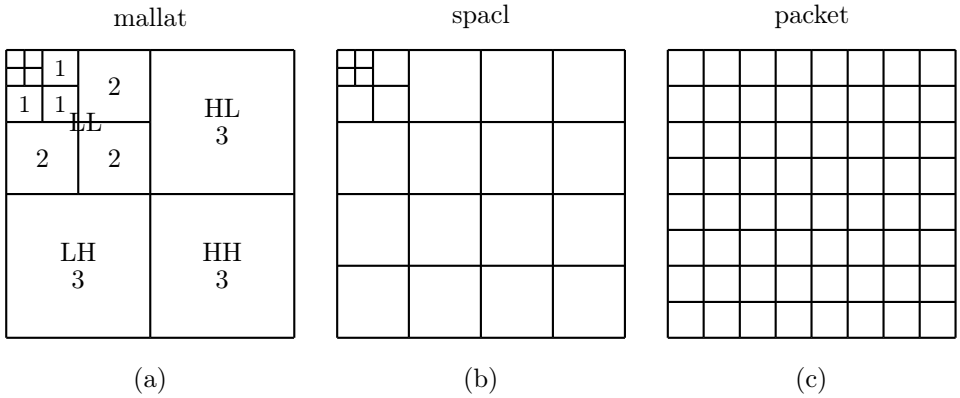


Figura 5.72: Organización de la subbanda en JPEG 2000.

Ahora denotamos una fila de píxeles extendidos en una tesela como $P_k, P_{k+1}, \ldots, P_m$. Dado que los píxeles han sido ampliados, los valores de índice por debajo de k y por encima de m pueden ser utilizados. la transformada wavelet $(5,3)$ entera calcula los coeficientes wavelet $C(i)$ calculando primero los valores impares $C(2i+1)$ y después los usa para calcular los valores pares $C(2i)$. Los cálculos son:

$$C(2i+1) = P(2i+1) - \left\lfloor \frac{P(2i) + P(2i+2)}{2} \right\rfloor, \quad \text{para } k-1 \le 2i+1 < m+1,$$
$$C(2i) = P(2i) + \left\lfloor \frac{C(2i-1) + C(2i+1) + 2}{4} \right\rfloor, \quad \text{para } k \le 2i < m+1.$$

La transformada wavelet $(9,7)$ de punto flotante se calcula mediante la ejecución de cuatro pasos de "lifting" seguidos por dos pasos "de escalado" en los valores de píxel extendidos, de P_k a P_m. Cada paso se aplica a todos los píxeles en la tesela antes de comenzar el paso siguiente. El paso 1 se efectúa para todos los valores i que satisfacen $k-3 \le 2i+1 < m+3$. El paso 2 se lleva a cabo para todos los i tales que $k-2 \le 2i < m+2$. El paso 3 se realiza para $k-1 \le 2i+1 < m+1$. El paso 4 se efectúa para $k \le 2i < m$. El paso 5 se leva a cabo para $k \le 2i+1 < m$. Finalmente, el paso 6 se ejecuta para $k \le 2i < m$. Los cálculos son:

$$\begin{aligned}
C(2i+1) &= P(2i+1) + \alpha\left[P(2i) + P(2i+2)\right], && \text{paso 1}\\
C(2i) &= P(2i) + \beta\left[C(2i-1) + C(2i+1)\right], && \text{paso 2}\\
C(2i+1) &= C(2i+1) + \gamma\left[C(2i) + C(2i+2)\right], && \text{paso 3}\\
C(2i) &= C(2i) + \delta\left[C(2i-1) + C(2i+1)\right], && \text{paso 4}\\
C(2i+1) &= -K \times C(2i+1), && \text{paso 5}\\
C(2i) &= (1/K) \times C(2i), && \text{paso 6}
\end{aligned}$$

donde las cinco constantes (coeficientes del filtro wavelet) usadas por JPEG 2000 vienen dadas por $\alpha = -1{,}586134342$, $\beta = -0{,}052980118$, $\gamma = 0{,}882911075$, $\delta = 0{,}443506852$, y $K = 1{,}230174105$.

Estas transformaciones wavelet unidimensionales se aplican L veces, donde L es un parámetro (ya sea controlado por el usuario, ya sea establecido por el codificador), y se intercalan en las filas y las columnas para formar L niveles (o resoluciones) de subbandas. La resolución $L-1$ es la imagen original (resolución 3 en la Figura 5.72a), y la resolución 0 es la subbanda de frecuencia más baja. Las subbandas pueden ser organizadas en una de tres maneras, como se ilustra en la Figura 5.72a–c.

Cuantificación: Cada subbanda puede tener un tamaño de paso de cuantificación diferente. Cada coeficiente wavelet en la subbanda es dividido por el tamaño de paso de la cuantificación y el resultado

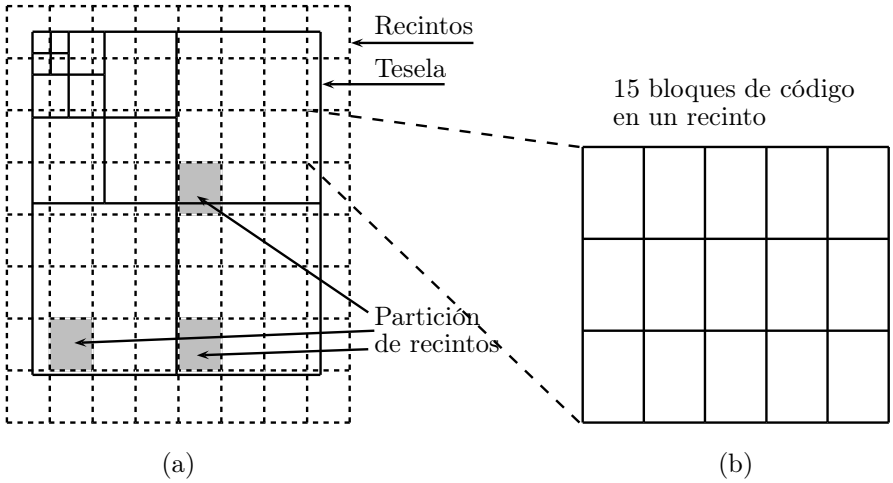


Figura 5.73: Subbandas, recintos, y bloques de código.

es truncado. El tamaño de paso de la cuantificación puede determinarse iterativamente con el fin de lograr una tasa de bits como objetivo (i.e., el factor de compresión puede ser especificado de antemano por el usuario) o con el fin de alcanzar un nivel de calidad de imagen predeterminado. Si se desea una compresión con pérdida, el paso de la cuantificación se establece en 1.

Precintos y bloques de código: Considere una tesela en un componente de color. Los píxeles originales son transformadas wavelet, lo que produce subbandas de L niveles de resolución. La Figura 5.73a muestra una tesela y cuatro niveles de resolución. Hay tres subbandas en cada nivel de resolución (excepto en el nivel más bajo). El tamaño total de todas las subbandas es igual al tamaño de la tesela. Ahora se superpone una rejilla de rectángulos conocidos como *recintos* (*precincts*) a la imagen completa, como se muestra en la Figura 5.73b. El origen de la rejilla de recintos se fija en la esquina superior izquierda de la imagen y las dimensiones de un recinto (su anchura y altura) son potencias de 2. Observe que los límites de la subbanda no son en general idénticos a los límites del recinto. Examinamos ahora las tres subbandas de una cierta resolución y seleccionamos los tres recintos ubicados en las mismas regiones en las tres subbandas (los tres rectángulos grises en la Figura 5.73a). Estos tres recintos constituyen una *partición de recinto*s. La rejilla de recintos se divide ahora en una fina rejilla de *bloques de código*, que son las unidades básicas a ser codificadas aritméticamente. La Figura 5.73b muestra cómo se ha particionado un recinto en 15 bloques de código. En consecuencia, una partición del recinto en este ejemplo está formada por 45 bloques de código. Un bloque de código es un rectángulo de tamaño 2^{xcb} (ancho) por 2^{ycb} (alto), donde $2^{ycb} \leq xcb$, $ycb \leq 10$ y $xcb + ycb \leq 12$.

Podemos pensar en las teselas, recintos y bloques de código como particiones gruesas, medias y finas de la imagen, respectivamente. La partición de la imagen en unidades cada vez más pequeñas ayuda en: (1) la creación de implementaciones de bajo consumo de memoria, (2) el *streaming* (transmisión continua), y (3) permitir un fácil acceso a muchos puntos en el bitstream. Se espera que los codificadores JPEG 2000 sencillos puedan ignorar esta partición y tener sólo una tesela, un recinto y un bloque de código. Los codificadores sofisticados, por otro lado, pueden terminar con un gran número de bloques de código, permitiendo así al decodificador realizar una descompresión progresiva, una transmisión en continuo (streaming) rápida, zums, panorámicas y otras operaciones especiales mientras decodifica sólo partes de la imagen.

Codificación de entropía: Los coeficientes wavelet de un bloque de código son codificados aritméticamente por planos de bits. La codificación se realiza desde el plano de bits más significativo (que contiene los bits más importantes de los coeficientes) al plano de bits menos significativo. Cada plano

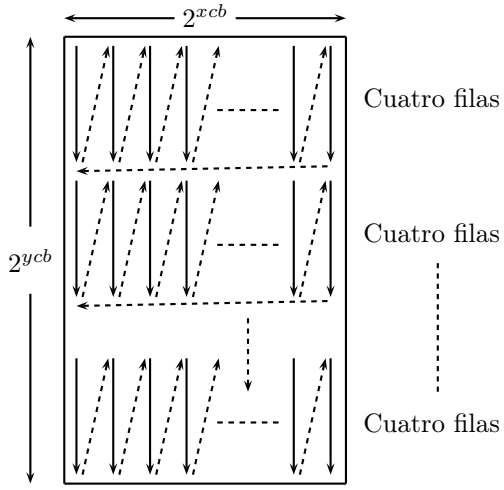


Figura 5.74: Bandas en un bloque de código.

de bits es escaneado tal como se muestra en la Figura 5.74. Se determina un contexto para cada bit, se estima una probabilidad a partir del contexto, y el bit y su probabilidad se envían al codificador aritmético.

Muchos métodos de compresión de imagen funcionan de manera diferente. Un algoritmo de compresión de imágenes típico puede utilizar varios de los vecinos de un píxel como su contexto y codificar el píxel (no sólo un bit individual) basándose en el contexto. Dicho contexto puede incluir sólo los píxeles que serán conocidos por el decodificador (normalmente los píxeles ubicados sobre el píxel actual o a su izquierda). JPEG 2000 es diferente en este aspecto. Codifica bits individuales (por eso utiliza el codificador MQ, que codifica bits, no números) y utiliza contextos simétricos. El contexto de un bit se calcula a partir de sus ocho vecinos cercanos. Sin embargo, puesto que en el momento de la decodificación el decodificador no conocerá a todos los vecinos, el contexto no puede usar los valores de los vecinos. En su lugar, utiliza la *significación* de los vecinos. Cada coeficiente wavelet tiene una variable de 1 bit (un flag o bandera) asociado con él, que indica su importancia. Éste es el *estado de significación* del coeficiente. Cuando se inicia la codificación de un bloque de código, todos sus coeficientes wavelet se consideran insignificantes y todos los estados de significación son puestos a cero.

Algunos de los planos de bits más significativos pueden ser todo ceros. El número de tales planos de bit es almacenado en el bitstream, para uso del decodificador. La codificación comienza desde el primer plano de bits que no es idénticamente cero. Ese plano de bits es codificado en una sola pasada (una *pasada de eliminación de datos superfluos* o *cleanup pass*). Cada uno de los planos de bits menos significativos que le siguen son codificados en tres pasadas, referidas como *pasada de propagación de significación*, *pasada de refinamiento de magnitud*, y *pasada de eliminación de datos superfluo o de limpieza*. Cada pasada divide el plano de bits en bandas que son de cuatro filas de alto cada una (Figura 5.74). Cada banda es escaneada columna por la columna, de izquierda a derecha. Cada bit en el plano de bits es codificado en una de las tres pasadas. Como se mencionó anteriormente, la codificación de un bit implica: (1) la determinación de su contexto, (2) la estimación de una probabilidad para ello, y (3) el envío del bit y su probabilidad para el codificador aritmético.

La primera pasada de codificación (propagación de significación) de un plano de bits codifica todos los bits que pertenecen a los coeficientes wavelet satisfaciendo: (1) el coeficiente es insignificante y (2) al menos uno de sus ocho vecinos más cercanos es significativo. Si un bit es codificado en esta pasada y si el bit es 1, su coeficiente wavelet es marcado como significativo estableciendo su estado

de significación a 1. Los bits posteriores codificados en esta pasada (y las dos pasadas siguientes) considerarán este coeficiente significativo.

Es claro que en esta pasada para codificar *ciertos* bits, algunos coeficientes wavelet deben ser declarados significativos antes incluso de que la pasada comience. Por ello, el primer plano de bits que está siendo codificado es codificado en un solo paso conocido como *cleanup pass*. En ese paso, todos los bits del plano de bits son codificados. Si ocurre que un bit está a 1, su coeficiente es declarado significativo.

La segunda pasada de codificación (refinamiento de magnitud) de un plano de bits codifica todos los bits de los coeficientes wavelet que se convirtieron en significativos en el plano de bits *previo*. En consecuencia, una vez que un coeficiente se vuelve significativo, todos sus bits menos significativos se codifican uno a uno, cada uno en la segunda pasada de un plano de bits diferente.

La tercera y última pasada de codificación (limpieza) de un plano de bits codifica todos los bits no codificados en las dos primeras pasadas. Consideremos un coeficiente C en un plano de bits B. Si C es insignificante, su bit en B no será codificado en la segunda pasada. Si los ocho vecinos cercanos de C son insignificantes, el bit de C en el plano de bits B tampoco será codificado en la primera pasada. Ese bit será, por lo tanto, codificado en la tercera pasada. Si ocurre que el bit es 1, C se convertirá en significativo (el codificador fijará su estado de significación a 1).

Los coeficientes wavelet son enteros con signo y tienen un bit de signo. En JPEG 2000, son representados con el método de *signo-magnitud*. El bit de signo es 0 para un coeficiente positivo (o un cero) y 1 para un coeficiente negativo. Los bits de magnitud son los mismos, independientemente del signo. Si un coeficiente tiene un bit de signo y ocho bits de magnitud, entonces el valor +7 se representa como 0 | 00000111, y −7 se representa como 1 | 00000111. El bit de signo de un coeficiente se codifica siguiendo al primer bit 1 del coeficiente.

El comportamiento de las tres pasadas se ilustra mediante un sencillo ejemplo (Tabla 5.76). Asumimos cuatro coeficientes con valores $10 = 0 \mid 00001010$, $1 = 0 \mid 00000001$, $3 = 0 \mid 00000011$ y $-7 = 1 \mid 00000111$. Existen ocho planos de bits numerados de 7 a 0, de izquierda (el más significativo) a derecha (el menos significativo). El plano de bits con los bits de signo es inicialmente ignorado. Los primeros cuatro planos de bits 7–4 son todo ceros, por lo que la codificación comienza con el plano de bits 3 (Figura 5.75). Hay una única pasada para este plano de bits —la pasada de limpieza—. En esta pasada, se codifica un bit de cada uno de los cuatro coeficientes. El bit para el coeficiente 10 es 1, por lo que este coeficiente es declarado significativo (los bits restantes, en los planos de bits 2, 1 y 0, serán codificados en la pasada 2). También se codifica el bit de signo de 10, siguiendo a este 1. A continuación, se codifica el plano de bits 2. El coeficiente 10 es significativo, por lo que su bit en este plano de bits (un cero) es codificado en la pasada 2. El coeficiente 1 es insignificante, pero uno de sus vecinos cercanos (el 10) es significativo, por lo que el bit de 1 en el plano de bits 2 (un cero) es codificado en la pasada 1. Los bits de los coeficientes de 3 y −7 en este plano de bits (un cero y un uno, respectivamente) se codifican en la pasada 3. El bit de signo del coeficiente −7 se codifica siguiendo al bit 1 de ese coeficiente. Además, el coeficiente de -7 es declarado significativo.

A continuación se codifica el plano de bits 1. El bit del coeficiente 10 se codifica en la pasada 2. Los bits del coeficiente 1 se codifican en la pasada 1, de la misma forma que en el plano de bits 2. El bit del coeficiente 3, sin embargo, es codificado en la pasada 1 ya que su vecino cercano, el −7, es ahora significativo. Este bit es 1, por lo que el coeficiente 3 se hace significativo. Este bit es el primer 1 del coeficiente 3, por lo que el signo de 3 es codificado siguiendo a este bit.

◇ **Ejercicio 5.19 (sol. en pág. 1104):** Descríbase el orden de la codificación del último plano de bits.

El contexto de un bit es determinado de una forma diferente para cada pasada. Aquí mostramos cómo se determina para la pasada propagación de significación. Los ocho vecinos cercanos del coeficiente wavelet actual X se utilizan para determinar el contexto usado en la codificación de cada bit de X. Se selecciona uno de nueve contextos y se utiliza para estimar la probabilidad del bit que está siendo

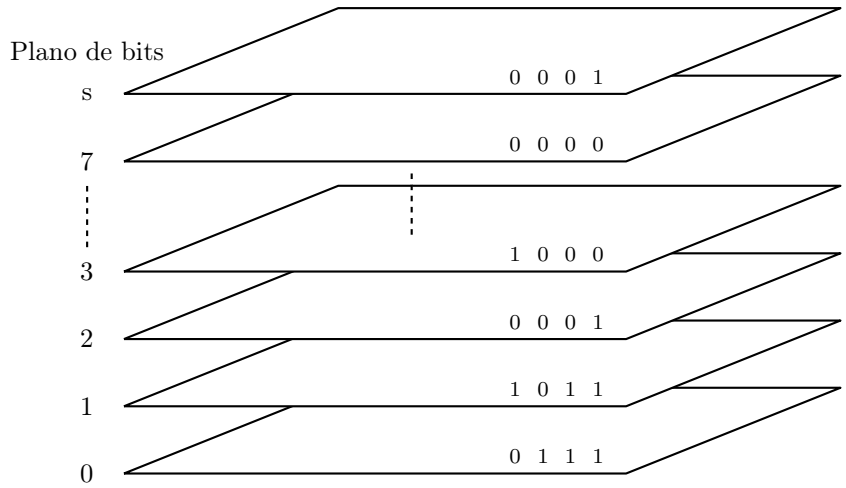


Figura 5.75: Planos de bits de cuatro coeficientes en un bloque de código.

	Coeficientes			
Plano de bits	10	1	3	−7
signo	0	0	0	1
7	0	0	0	0
6	0	0	0	0
5	0	0	0	0
4	0	0	0	0
3	1	0	0	0
2	0	0	0	1
1	1	0	1	1
0	0	1	1	1

(a)

		Coeficientes			
Plano de bits	Pasada	10	1	3	−7
3	limpieza	1+	0	0	0
2	significación		0		
2	refinamiento	0			
2	limpieza			0	1−
1	significación		0	1+	
1	refinamiento	1			1
1	limpieza				
0	significación		1+		
0	refinamiento	0		1	1
0	limpieza				

(b)

Tabla 5.76: Codificación de cuatro planos de bits para cuatro coeficientes.

D_0	V_0	D_1
H_0	X	H_1
D_2	V_1	D_3

Figura 5.77: Ocho vecinos.

Subbandas LL y LH (paso alto vertical)			Subbanda HL (paso alto horizontal)			Subbanda HH (paso alto diagonal)		Contexto
$\sum H_i$	$\sum V_i$	$\sum D_i$	$\sum H_i$	$\sum V_i$	$\sum D_i$	$\sum (H_i + V_i)$	$\sum D_i$	
2				2			≥ 3	8
1	≥ 1		≥ 1	1		≥ 1	2	7
1	0	≥ 1	0	1	≥ 1	0	2	6
1	0	0	0	1	0	≥ 2	1	5
0	2		2	0		1	1	4
0	1		1	0		0	1	3
0	0	≥ 2	0	0	≥ 2	≥ 2	0	2
0	0	1	0	0	1	1	0	1
0	0	0	0	0	0	0	0	0

Tabla 5.78: Nueve contextos para la significación de la pasada de propagación.

codificado. Sin embargo, como se mencionó anteriormente, son los *estados de significación* actuales de los ocho vecinos, no sus valores, los que se utilizan en la determinación. La Figura 5.77 muestra los nombres asignados a los estados de significación de los ocho vecinos. La Tabla 5.78 lista los criterios utilizados para seleccionar uno de los nueve contextos. Observe que estos criterios dependen de cuál de las cuatro subbandas (HH, HL, LH, o LL) está siendo codificada. El contexto 0 es seleccionado cuando el coeficiente X no tiene vecinos cercanos significativos. El contexto 8 es seleccionado cuando los ocho vecinos de X son significativos.

El estándar JPEG 2000 especifica reglas similares para determinar el contexto de un bit en las pasadas de refinamiento y de limpieza, así como para el contexto de un bit de signo. La determinación del contexto en la pasada de limpieza es idéntica a la de la pasada de significación, con la diferencia de que se utiliza la codificación run-length si los cuatro bits en una columna de una banda son insignificantes y cada uno tiene sólo vecinos insignificantes. En tal caso, es codificado un único bit, para indicar si la columna es de todo ceros o no. Si no, entonces la columna tiene una racha (*run*) de (entre cero y tres) ceros. En dicho caso, la longitud de este run es codificada. Este run debe, por supuesto, ser seguido por un 1. Este 1 no tiene que codificado, ya que su presencia es deducida fácilmente por el decodificador. La codificación normal (bit a bit) continúa para el siguiente bit a este 1.

Una vez que se ha determinado un contexto para un bit, se utiliza para estimar una probabilidad para la codificación del bit. Ésto se efectúa usando una tabla de estimación de probabilidad, similar a la Tabla 2.70. Observe que JBIG y JBIG2 (Secciones 4.11 y 4.12, respectivamente) utilizan tablas similares, pero usan miles de contextos, en contraste con los pocos contextos (nueve o menos) utilizados por JPEG 2000.

Paquetes: Después de que todos los bits de todos los coeficientes de todos los bloques de código de una partición de recintos han sido codificados en un corto bitstream, se añade una cabecera a ese bitstream, convirtiendolo así en un paquete. La Figura 5.73a,b muestra una partición de recintos consistente en tres recintos, cada uno dividido en 15 bloques de código. La codificación de esta partición produce, por tanto, un paquete con 45 bloques de código codificados. La cabecera contiene toda la información necesaria para decodificar el paquete. Si todos los bloques de código en una partición de recintos son idénticamente cero, el cuerpo del paquete está vacío. Recordemos que una partición de recintos corresponde a la misma ubicación espacial en tres subbandas. Como resultado, un paquete puede ser considerado un incremento de calidad para un nivel de resolución en una localización espacial determinada.

Capas (layers): Una capa es un conjunto de paquetes. Contiene un paquete de cada partición de recintos de cada nivel de resolución. En consecuencia, una capa es un incremento de calidad para la totalidad de la imagen a resolución completa.

Transmisión progresiva: Ésta es una característica importante de JPEG 2000. El estándar proporciona cuatro modos de transmisión progresiva y decodificación de una imagen: mediante resolución, calidad, ubicación espacial y componente. La progresión se logra simplemente almacenando los paquetes en un orden específico en el bitstream. Por ejemplo, la progresión de calidad (también conocida como SNR) puede lograrse organizando los paquetes en capas, dentro de cada capa por el componente, dentro de cada componente por el nivel de resolución, y dentro de cada nivel de resolución por la partición de recintos. La resolución progresiva se logra cuando los paquetes se disponen según la partición de recintos (la de anidación más interna), la capa, el componente de imagen y el nivel de resolución (el de anidación más externa).

Cuando una imagen es codificada, los paquetes se colocan en el bitstream en un cierto orden, que corresponde a una cierta progresión. Si un usuario o una aplicación requieren una progresión diferente (y, por lo tanto, un orden diferente de los paquetes), debe ser fácil leer el bitstream, identificar los paquetes y reorganizarlos. Este proceso se conoce como *análisis sintáctico* (*parsing*), y es una tarea fácil debido a los muchos *marcadores* incrustados en el bitstream. Existen distintos tipos de marcadores y se usan para diferentes propósitos. Ciertos marcadores identifican el tipo de progresión utilizada en el bitstream, y otros contienen las longitudes de todos los paquetes. Por lo tanto, el bitstream puede ser analizado sin tener que decodificar nada de eso.

Un ejemplo típico de análisis sintáctico es la impresión de una imagen en color en una impresora de escala de grises. En tal caso, no tiene ningún sentido el envío de información de color a la impresora. Un analizador sintáctico (*parser*) puede utilizar los marcadores para identificar todos los paquetes que contienen información de color y desecharlos. Otro ejemplo es la disminución del tamaño del bitstream (aumentando la cantidad de pérdidas en la imagen). El analizador debe identificar y descartar las capas que contribuyen menos a la calidad de imagen. Ésto se efectúa varias veces, hasta que se alcanza el tamaño de bitstream deseado.

El analizador sintáctico puede formar parte de un *servidor de imágenes*. Un cliente envía una petición a tal servidor con el nombre de una imagen y un atributo deseado. El servidor ejecuta el analizador para obtener la imagen con ese atributo y transmite el bitstream al cliente.

Regiones de interés: Un cliente puede desear decodificar sólo una parte de una imagen —una región de interés (ROI)—. El analizador sintáctico (parser) debe ser capaz de identificar las partes del bitstream correspondientes a la ROI y transmitir sólo aquellas partes. Una ROI puede ser especificada en tiempo de compresión. En tal caso, el codificador identifica los bloques de código localizados en la ROI y los escribe al principio del bitstream. En muchos casos, sin embargo, se especifica una ROI a un parser después de que la imagen ha sido comprimida. En dicho caso, el analizador puede utilizar las teselas para identificar la ROI. Cualquier ROI se incluirá en una o varias teselas, y el parser puede identificar esas teselas y transmitir al cliente un bitstream formado por sólo esas teselas. Ésta es una aplicación importante de las teselas. Teselas pequeñas hacen posible especificar pequeñas

ROI, pero dan lugar a una compresión deficiente. La experiencia sugiere que las teselas de tamaño de 256×256 tienen un impacto negativo insignificante sobre la relación de compresión y normalmente son lo suficientemente pequeñas para especificar ROIs.

Puesto que cada tesela es comprimida individualmente, es fácil encontrar la información de la tesela en el bitstream. Cada tesela tiene un encabezado y los marcadores que simplifican esta tarea. Cualquier análisis sintáctico que pueda realizarse en la imagen completa, también se puede hacerse en las teselas individuales. Otras teselas pueden, o bien ser ignoradas, o bien ser transmitidas con una calidad inferior.

Alternativamente, el analizador sintáctico puede manejar pequeñas ROIs extrayendo del bitstream información para los bloques de código individuales. El parser tiene que: (1) determinar qué bloques de código contienen un píxel dado, (2) encontrar los paquetes que contienen los bloques de código, (3) decodificar las cabeceras de los paquetes, (4) utilizar los datos de cabecera para encontrar la información del bloque de código dentro cada paquete, y (5) transmitir inmediatamente esta información al decodificador.

Resumen: La experimentación actual indica que JPEG 2000 tiene un mejor rendimiento que el JPEG original, especialmente para imágenes donde se requieren tasas de bits muy bajas (grandes factores de compresión) o muy alta calidad de imagen. Para la compresión sin pérdidas o casi sin pérdidas, JPEG 2000 ofrece sólo mejoras modestas sobre JPEG.

> En las secciones siguientes voy a presentar la transformada wavelet y desarrollar un plan que nos permitirá implementar la transformada wavelet de una manera eficiente en un ordenador digital. La transformada será tan eficiente que ni siquiera utiliza wavelet alguna. (El lector atento puede levantar aquí una ceja y preguntar: "¡Seguramente usted no puede hablar en serio?")
>
> —Clemens Valens, [polyvalens 06]

Capítulo 6

Compresión de vídeo

La grabación de sonido y la cámara de cine se encuentran entre los grandes inventos de Thomas Edison. Ambas se unieron más tarde, cuando se desarrolló el cine sonoro, y todavía se utilizan juntas en las grabaciones de vídeo. Esta unificación es una de las razones de la popularidad de las películas y el video. Con los rápidos avances en las computadoras en las décadas de 1980 y 1990 llegaron las aplicaciones multimedia, en las que las imágenes y el sonido se combinan en el mismo archivo. Estos archivos tienden a ser grandes, por lo que su compresión se convirtió en una aplicación natural.

Este capítulo comienza con discusiones básicas sobre el video analógico y el digital, continúa con los principios de la compresión de vídeo, y concluye con una descripción de varios métodos de compresión diseñados específicamente para vídeo, a saber, MPEG-1, MPEG-4, H.261 y H.264.

6.1. Vídeo analógico

Una cámara de vídeo analógica convierte la imagen que "ve" a través de su lente a un voltaje eléctrico (una señal) que varía con el tiempo de acuerdo con la intensidad y el color de la luz emitida por las diferentes partes de la imagen. Dicha señal se denomina *analógica*, porque es análoga (proporcional) a la intensidad de la luz. La mejor manera de comprender ésta señal es ver cómo responde a la misma un receptor de televisión.

Consulta del diccionario

Analógico (adjetivo).

Existencia de un mecanismo que representa los datos mediante mediciones de una cantidad variable continua (como la tensión eléctrica).

6.1.1. El CRT

Un receptor de televisión (tradicionalmente un CRT, o tubo de rayos catódicos), Figura 6.1a), es un tubo de cristal con una forma familiar. En la parte posterior tiene un cañón de electrones (cátodo) que emite una corriente de electrones. Su superficie frontal está cargada positivamente, por lo que atrae los electrones (que tienen carga eléctrica negativa). La parte frontal está recubierta con un compuesto de fósforo que convierte la energía cinética de los electrones que lo golpean en luz. El destello de luz dura sólo una fracción de un segundo, por consiguiente, para obtener una visualización constante, la imagen tiene que actualizarse varias veces por segundo. La tasa de actualización real depende de la *persistencia*

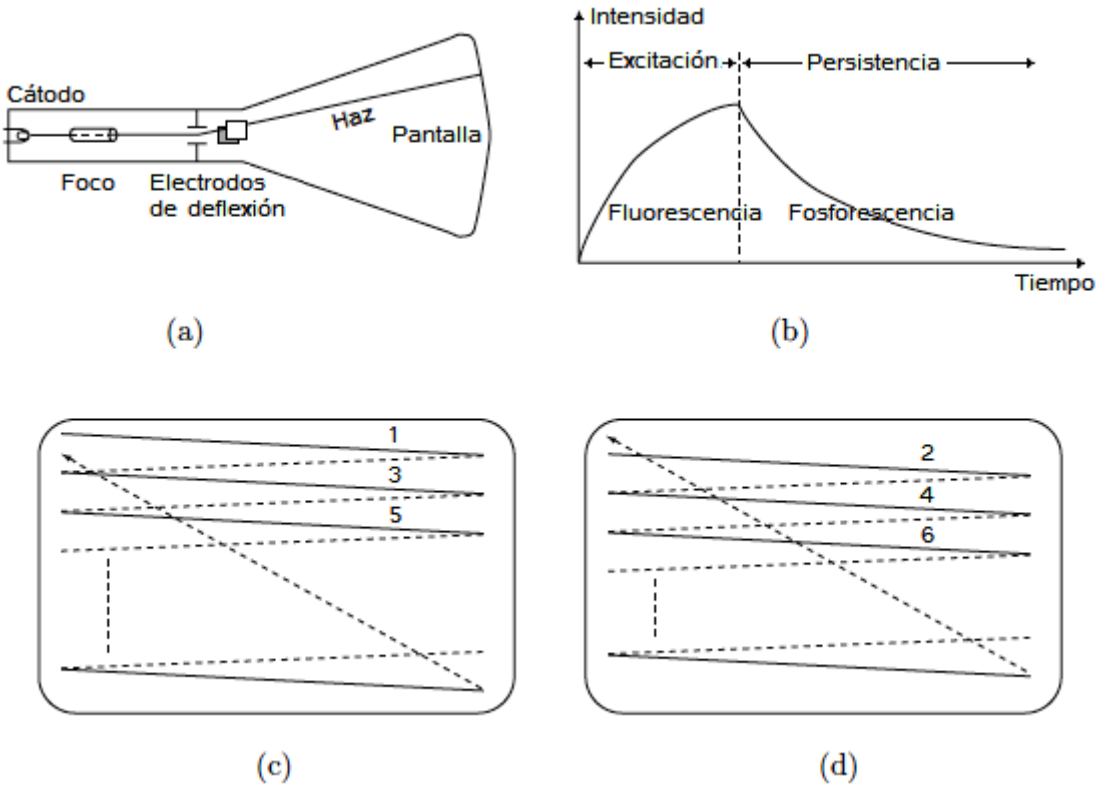


Figura 6.1: (a) Operación CRT. (b) Persistencia. (c) Líneas escaneo impares. (d) Líneas escaneo pares.

del compuesto (Figura 6.1b). Para ciertos tipos de trabajo, tales como el dibujo arquitectónico, una larga persistencia es aceptable. Para la animación, la persistencia corta es una necesidad.

Los pioneros del cine descubrieron, después de mucha experimentación, que la frecuencia de actualización mínima requerida para una animación suave es de 15 imágenes (fotogramas o *frames*) por segundo (fps), por lo que adoptaron 16 fps como frecuencia de actualización para sus cámaras y proyectores. Sin embargo, cuando empezaron las películas de acción rápida (como los *westerns*), la industria cinematográfica decidió aumentar la tasa de refresco a 24 fps, una tasa que se utiliza en la actualidad. En un momento dado, se descubrió que esta tasa se puede duplicar artificialmente, a 48 fps (que produce animación más suave), proyectando cada fotograma dos veces. Ésto se realiza empleando un obturador de doble hoja giratoria en el proyector de películas El obturador expone una imagen, la cubre, y la expone de nuevo, todo en $1/24$ de segundo, con lo que consigue una tasa de refresco efectiva de 48 fps. Los proyectores de cine modernos tienen lámparas muy brillantes y permiten incluso el uso de un obturador de triple hoja, para obtener una tasa de refresco efectiva de 72 fps.

La frecuencia de la corriente eléctrica en Europa es de 50 Hz, por lo que los estándares de televisión utilizados allí, como PAL y SECAM, emplean una velocidad de refresco de 25 fps. Esto es conveniente para transmisión de una película en la televisión. La película, que fue filmada a 24 fps, se muestra a 25 fps, una diferencia imperceptible.

La frecuencia de la corriente eléctrica en los Estados Unidos es de 60 Hz, por lo que cuando llegó la televisión, en la década de 1930, se utilizó una tasa de refresco de 30 fps. Cuando se añadió el color, en 1953, tasa que se redujo en un 1 %, a 29,97 fps, por la necesidad de una separación precisa de las señales portadoras de video y de audio. Debido al entrelazado, una imagen de televisión completa

consta de dos fotogramas, por lo que una frecuencia de actualización de 29,97 frames por segundo requiere una tasa de 59,94 frames por segundo.

Resulta que la frecuencia de refresco para la televisión debe ser más alta que la de las películas. Una película se ve normalmente en la oscuridad, mientras que la televisión se ve en una habitación iluminada, y la visión humana es más sensible a parpadear en condiciones de iluminación brillante. Ésta es la razón por la cual 30 (ó 29,97) fps es mejor que 25.

El haz de electrones se puede apagar y encender muy rápidamente. También puede ser desviado horizontal y verticalmente por dos pares (X e Y) de electrodos. La visualización de un solo punto de la pantalla se efectúa girando el haz apagado, desplazándolo a la parte de la pantalla donde el punto debe aparecer y encendiéndolo. Ésto se consigue mediante un hardware especial que responde a la señal analógica recibida por el aparato de televisión.

La señal indica al hardware que desactive el haz, lo mueva a la esquina superior izquierda de la pantalla, lo active, y realice un barrido siguiendo una línea horizontal en la pantalla. Mientras el haz efectúa el barrido horizontal a lo largo de la línea de escaneo superior, se utiliza la señal analógica para ajustar la intensidad de los haces de acuerdo con la imagen que se está mostrando. Al final de la primera línea de escaneo, la señal da instrucciones al hardware de la televisión para desactivar el haz, moverlo hacia atrás y ligeramente hacia abajo, hasta el inicio de la tercera (no la segunda) línea de escaneo, vuelve a activarlo, y realiza el barrido de esa línea. El movimiento del haz al inicio de la siguiente línea de exploración se conoce como *retrazado* (*retrace*). El tiempo que tarda el retrazado es el *tiempo de borrado horizontal* (*horizontal blanking time*).

De esta manera, un campo de la imagen es creado en la pantalla línea por línea, usando sólo el escaneo de la líneas impares (Figura 6.1c). Al final de la última línea, la señal contiene instrucciones para un el retrazo de un frame. Ésto apaga el haz y lo mueve al inicio del campo siguiente (la segunda línea de escaneo) para explorar las líneas de escaneo de numeración par del campo (Figura 6.1d). El tiempo que tarda en hacer el retrazo vertical es el *tiempo de borrado vertical* (*vertical blanking time*). Por consiguiente, la imagen se crea en dos campos que juntos constituyen un *frame*. Se dice que la imagen está *entrelazada*.

Este proceso se repite varias veces cada segundo, para actualizar la imagen. Esta orden de escaneado (de izquierda a derecha, de arriba abajo, con o sin entrelazado) se llama escaneo *raster*. La palabra raster se deriva del latín *rastrum*, que significa rastrillo, ya que este escaneo se realiza en un patrón similar al que deja un rastrillo en un campo.

Un aparato de televisión dirigido al consumidor utiliza uno de los tres estándares internacionales. El estándar utilizado en los Estados Unidos se llama NTSC (National Television Standards Committee o Comité de eStándares sobre Televisión Nacional), aunque el nuevo estándar digital (Sección 6.3.1) se está haciendo popular rápidamente. NTSC especifica una transmisión de televisión de 525 líneas (hoy en día, esto sería $2^9 = 512$ líneas, pero puesto que que la televisión se desarrolló antes de la llegada de las computadoras con su preferencia por los números binarios, el estándar NTSC no tiene nada que ver con las potencias de dos). Debido al borrado vertical, sin embargo, sólo son visibles 483 líneas en la pantalla. Dado que la relación de aspecto (anchura/altura) de una pantalla de televisión es 4:3, cada línea tiene un tamaño de $\frac{4}{3}483 = 644$ píxeles. La resolución de un aparato de televisión estándar es, por lo tanto, 483×644. Ésto puede ser considerado como la mejor resolución media. (Esta es la razón por la cual el texto es tan difícil de leer en una televisión estándar.)

◇ **Ejercicio 6.1 (sol. en pág. 1105):** (Fácil) ¿Cuál sería la resolución si las 525 líneas fueran visibles en la pantalla?

La relación de aspecto 4:3 fue seleccionada por Thomas Edison cuando construyó la primeras cámaras de cine y proyectores, y fue adoptada por la televisión a principios de la década de 1930. En la década de 1950, tras muchas pruebas con espectadores, la industria del cine decidió que la gente prefiere relaciones de aspecto mayores y comenzó a hacer películas para pantalla ancha, con relaciones de aspecto de 1,85 o superior. Influenciado por éso, los desarrolladores de video digital han optado

Formatos de imagen	Relación de aspecto
NTSC, PAL, y SECAM TV	1,33
Película de 16 mm y 35 mm	1,33
HDTV	1,78
Película de pantalla ancha	1,85
Película de 70 mm	2,10
Película en cinemascope	2,35

Tabla 6.2: Relaciones de aspecto de la televisión y el cine.

(Sección 6.3.1) por la gran relación de aspecto 16:9. El Ejercicio 6.4 compara las dos relaciones de aspecto, y en la Tabla 6.2 se muestran algunas relaciones de aspecto comunes de la televisión y del cine.

El concepto de *relación de aspecto del pel* también es útil y debe ser mencionado. Normalmente, pensamos en un pel (o un píxel) como un punto matemático, sin dimensiones y sin forma. En la práctica, sin embargo, los pels se imprimen o se visualizan, por lo que tienen forma y dimensiones. El uso de una máscara de sombra (véase más abajo) crea pels circulares, pero los monitores de ordenador normalmente muestran píxeles cuadrados o rectangulares, creando así una imagen clara y nítida (porque los píxeles cuadrados o rectangulares llenan completamente el espacio). MPEG-1 (Sección 6.5) incluso tiene un parámetro `pel_aspect_ratio`, cuyos 16 valores se muestran en la Tabla 6.26.

Se debe enfatizar que la televisión analógica no muestra píxeles. Cuando una línea es escaneada, la intensidad del haz varía continuamente. La imagen se muestra línea a línea, pero cada línea es continua. Por consiguiente, la imagen visualizada por la televisión analógica se muestra sólo en la dimensión vertical.

NTSC también especifica una frecuencia de actualización de 59,94 (ó $^{60}/_{1,001}$) frames por segundo y puede resumirse mediante la notación 525/59,94/2:1, donde 2:1 indica el entrelazado. La notación 1:1 indica *escaneo progresivo* (no es lo mismo que la compresión de imágenes progresiva). La televisión PAL estándar (línea de fase alternada), utilizada en Europa y Asia, se resume en 625/50/2:1. La cantidad $262{,}5 \times 59{,}94 = 15734{,}25$ kHz se denomina la *velocidad de línea* de la 525/59,94/2:1 estándar. Éste es el producto del tamaño del frame (número de líneas por frame) y la frecuencia de actualización.

Se debe mencionar que NTSC y PAL son estándares para la codificación de color. En ellos se especifica cómo codificar el color en la señal de vídeo analógica en negro y blanco.

Sin embargo, por razones históricas, los sistemas de televisión que utilizan normalmente un escaneo 525/59,94 emplean un código de colores NTSC, mientras que los sistemas de televisión que usan normalmente un escaneo 625/50 emplean una codificación de color PAL. Ésta es la razón por la que 525/59,94 y 625/50 se llaman ilógicamente NTSC y PAL, respectivamente.

Unas palabras sobre el color: Muchos tubos CRTs de color utilizan la técnica de *máscara de sombra* (Figura 6.3). Tienen tres cañones de electrones que emiten tres haces separados. Cada haz está asociado con un color, pero los propios rayos, por supuesto, están formados por electrones y no tienen ningún color. Los haces están ajustados de tal manera que siempre convergen a una corta distancia detrás de la pantalla. En el momento en que llegan a la pantalla han divergido un poco, y encuentran un grupo de tres puntos diferentes (pero muy cercanos) llamados *tríada*.

La pantalla está recubierta con puntos hechos de tres tipos de compuestos de fósforo que emiten luz roja, verde, y azul, respectivamente, cuando son excitados. En el plano de convergencia hay una pantalla delgada, de metal perforado: la máscara de sombra. Cuando los tres haces convergen en un agujero de la máscara, lo atraviesan, divergen, y golpean una tríada de puntos recubiertos con diferentes compuestos fosforescentes. Los puntos brillan en los tres colores, y el observador ve una mezcla de rojo, verde y azul cuyo color preciso depende de las intensidades de los tres haces. Cuando

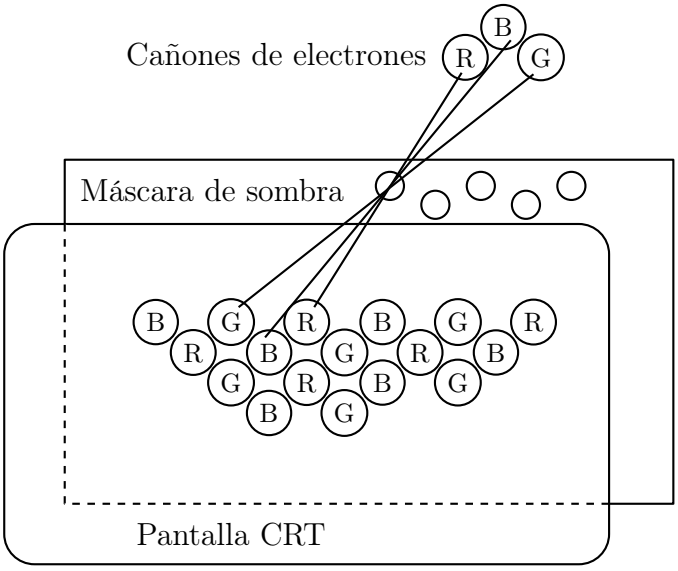
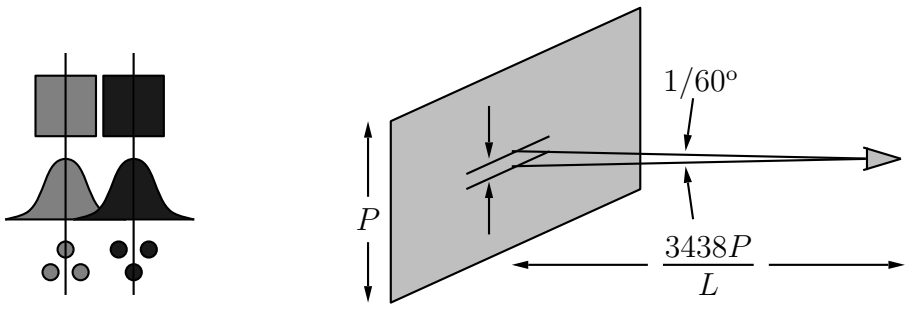


Figura 6.3: Una máscara de sombra.

los haces son desviados un poco, golpean la máscara y son absorbidos. Después de alguna deflexión más, convergen en otro agujero y golpean la pantalla en otra tríada.

En una resolución de pantalla de 72 ppp (puntos por pulgada) esperamos 72 píxeles cuadrados ideales por cada pulgada de pantalla. Cada píxel debe ser un cuadrado de lado $^{25,4}/_{72} \approx 0{,}353$ mm. Sin embargo, como muestra la Figura 6.4a, cada tríada produce una mancha circular ancha, con un diámetro de 0,63 mm, en la pantalla. Estas manchas se superponen, y cada una afecta a los colores percibidos de su vecinos.

Cuando vemos la televisión, tendemos a situarnos a una distancia desde la que es cómodo verla. Cuando la observamos desde una distancia mayor, echamos de menos algunos detalles, y al mirar de cerca, las líneas de exploración individuales son visibles. Los experimentos muestran que la distancia cómoda para la vista se determina por la regla: El más pequeño detalle que queremos ver debe subtender un ángulo de alrededor de un minuto de arco $(^1/_{60})^o$. Denotamos P a la altura de la imagen, y L, el número de líneas de escaneo. La relación entre grados y radianes es: $360^o = 2\pi$ radianes.



(a) (b)

Figura 6.4: (a) Píxeles cuadrados y circulares. (b) Distancia cómoda para la visualización.

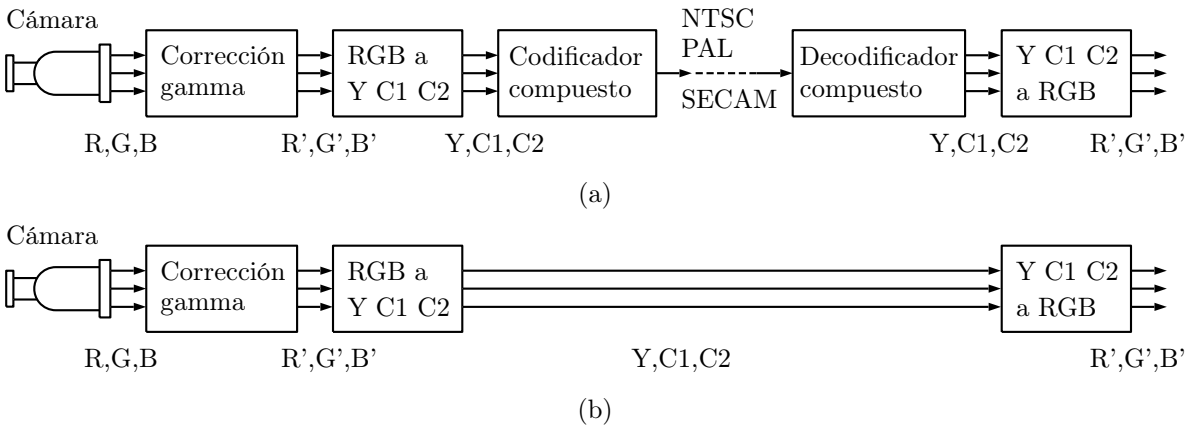


Figura 6.5: Transmisión de televisión (a) compuesta y (b) por componentes.

Combinando ésto con la Figura 6.4b se produce la expresión:

$$\frac{{}^{P}/_{L}}{\text{Distancia}} = \left(\frac{1}{60}\right)^{\text{o}} = \frac{2\pi}{360 \cdot 60} = \frac{1}{3438},$$

o

$$\text{Distancia} = \frac{3438P}{L}. \tag{6.1}$$

Si $L = 483$, la distancia confortable es $7{,}12P$. Para $L = 1080$, la Ecuación (6.1) sugiere una distancia de $3{,}18P$.

◇ **Ejercicio 6.2 (sol. en pág. 1105):** Mídase la altura de la imagen en su televisor y calcúlese la distancia confortable para la vista a partir de la Ecuación (6.1). Compárese con la distancia que realmente utiliza usted.

> Todos los libros del mundo no contienen más información que la que se difunde como vídeo en una única gran ciudad americana en un solo año. No todos los bits tienen el mismo valor.
>
> —Carl Sagan

6.2. Video compuesto y por componentes

El receptor de televisión común que se encuentra en muchos hogares recibe desde el transmisor una señal compuesta, donde las componentes de luminancia y crominancia [Salomon 99] están multiplexadas. Este tipo de señal fue diseñada en la década de 1950, cuando se añadió el color a las transmisiones de televisión. La señal básica en negro y blanco se convierte en el componente de luminancia (Y), y los dos componentes de crominancia $C1$ y $C2$ son adicionados. Ésos pueden ser U y V, Cb y Cr, I y Q, o cualesquier otros componentes de crominancia. La Figura 6.5a muestra los componentes principales de un transmisor y un receptor que utilizan una señal compuesta. El punto principal es que sólo se necesita una señal. Si la señal se envía por el aire, sólo es necesaria una frecuencia. Si es enviada por un cable, sólo se utiliza un cable.

> Televisión, un medio. Se llama así porque ni es maravillosa ni se hace bien.
>
> —Proverbio

NTSC utiliza los componentes $Y\ IQ$, que se definen mediante:

$$\begin{aligned} Y =& 0{,}299R' + 0{,}587B' + 0{,}114B', \\ I =& 0{,}596R' - 0{,}274G' - 0{,}322B' \\ =& -(\sin 33^{\circ})\, U + (\cos 33^{\circ})\, V, \\ Q =& 0{,}211R' - 0{,}523G' + 0{,}311B' \\ =& (\cos 33^{\circ})\, U + (\sin 33^{\circ})\, V. \end{aligned}$$

En el receptor, los componentes $R'G'B'$ de corrección gamma se extraen mediante la transformación inversa:

$$\begin{aligned} R' =& Y + 0{,}956I + 0{,}621Q, \\ G' =& Y - 0{,}272I - 0{,}649Q, \\ B' =& Y - 1{,}106I + 1{,}703Q. \end{aligned}$$

PAL utiliza el espacio de color básico $Y\ UV$, que se define por:

$$\begin{aligned} Y =& 0{,}299R' + 0{,}587G' + 0{,}114B', \\ U =& -0{,}147R' - 0{,}289G' + 0{,}436B' = 0{,}492\,(B' - Y)\,, \\ V =& 0{,}615R' - 0{,}515G' - 0{,}100B' = 0{,}877\,(R' - Y)\,, \end{aligned}$$

cuya transformada inversa es:

$$\begin{aligned} R' =& Y + 1{,}140V, \\ G' =& Y - 0{,}394U - 0{,}580V, \\ B' =& Y - 2{,}030U. \end{aligned}$$

SECAM utiliza el espacio de color compuesto $Y\ DrDb$, definido mediante:

$$\begin{aligned} Y =& 0{,}299R' + 0{,}587G' + 0{,}114B', \\ Db =& -0{,}450R' - 0{,}833G' + 1{,}333B' = 3{,}059U, \\ Dr =& -1{,}333R' + 1{,}116G' - 0{,}217B' = -2{,}169V. \end{aligned}$$

La transformación inversa es:

$$\begin{aligned} R' =& Y - 0{,}526Dr, \\ G' =& Y - 0{,}129Db + 0{,}268Dr, \\ B' =& Y + 0{,}665Db. \end{aligned}$$

El vídeo compuesto es barato, pero tiene problemas como la luminancia y la crominancia transversales, artefactos en la imagen representada. Los sistemas de alta calidad de vídeo a menudo utilizan el *vídeo por componentes*, donde tres cables o tres frecuencias llevan los componentes de color individuales (Figura 6.5b). Un vídeo por componentes estándar común es la recomendación 601 de la ITU-R, que adopta el espacio de colorYCbCr (página 642). En este estándar, la luminancia Y tiene valores en el rango [16, 235], mientras que cada uno de los dos componentes de crominancia tiene valores en el rango [16, 240] centrado en 128, lo que indica una crominancia de cero.

6.3. Vídeo Digital

El vídeo digital es el caso donde una cámara (digital) genera una imagen digital, i.e., una imagen que se compone de píxeles. Muchas personas pueden sentir intuitivamente que una imagen producida de esta manera es inferior a una imagen analógica. Una imagen analógica parece tener una resolución infinita, mientras que una imagen digital tiene una resolución fija y finita que no puede ser aumentada sin pérdidas en la calidad de la imagen. En la práctica, sin embargo, la alta resolución de las imágenes analógicas no es una ventaja, ya que las vemos en una pantalla de televisión o un monitor de ordenador en una resolución fija determinada. El vídeo digital, por otra parte, tiene las importantes ventajas siguientes:

1. Pueden ser fácilmente editadas. Ésto hace que sea posible producir efectos especiales. Las imágenes generadas por ordenador, tales como naves espaciales o personajes de dibujos animados, pueden combinarse con la acción de la vida real para producir efectos complejos, que parecen reales. Las imágenes de un actor en una película pueden ser editadas para que parezca joven al principio y viejo posteriormente. El software para la edición de vídeo digital está disponible para la mayoría de las plataformas informáticas. Los usuarios pueden editar un archivo de vídeo y adjuntarlo a un mensaje de correo electrónico (*email*), creando así un *vmail*. Las aplicaciones multimedia, donde el texto, el sonido, y las imágenes estáticas y de vídeo están integrados, son comunes hoy en día e implican la edición de video.

2. Se puede almacenar en cualquier medio digital, tales como discos duros, cartuchos extraíbles, CD-ROMs, o DVDs. Un código de corrección de errores puede ser añadido, si es necesario, para una mayor fiabilidad. Ésto hace que sea posible duplicar una película larga o transmitirla entre equipos sin pérdida de calidad (de hecho, sin un solo bit corrupto). En contraste, el vídeo analógico se almacena típicamente en la cinta, cada copia es de calidad ligeramente inferior a la original, y el medio está sujeto al desgaste.

3. Puede ser comprimido. Ésto permite una mayor capacidad de almacenamiento (cuando el vídeo se almacena en un medio digital) y también una transmisión rápida. El envío de vídeo comprimido entre ordenadores hace posible la telefonía de vídeo (videoteléfono), que, a su vez, hace posible la videoconferencia. La transmisión de vídeo comprimido también hace que sea posible aumentar la capacidad de los cables de televisión y por lo tanto la adición de canales.

El video digital es, en principio, una secuencia de imágenes, llamadas frames, representadas a una cierta *tasa de frames* (tantos fotogramas por segundo o fps) para crear la ilusión de animación. Esta tasa, así como el tamaño de la imagen y la profundidad del píxel, dependen en gran medida de la aplicación. Las cámaras de vigilancia, por ejemplo, utilizan la muy baja tasa de cinco fps, mientras que la HDTV se muestra a 25 fps. La Tabla 6.6 muestra algunas aplicaciones de vídeo típicas y sus parámetros de vídeo.

La tabla ilustra la necesidad de compresión. Incluso la aplicación más económica, una cámara de vigilancia, genera ¡$5 \times 640 \times 480 \times 12 = 18\,432\,000$ bits por segundo! Ésto es equivalente a más de 2,3 millones de bytes por segundo, y esta información tiene que ser guardada al menos un par de días antes de que pueda ser borrada. La mayoría de las aplicaciones de vídeo también involucran el audio. Éste forma parte de los datos de video globales y tiene que ser comprimido con la imagen de vídeo.

◇ **Ejercicio 6.3 (sol. en pág. 1105):** ¿Qué aplicaciones de vídeo no incluyen sonido?

Una pieza completa de vídeo a veces se llama una *presentación*. Ésta se compone de un número de *actos*, donde cada acto se divide en varias *escenas*. Una escena se compone de varios *planos* (*shots*) o *secuencias* de acción, cada una compuesta por una sucesión de *frames*, donde existe un pequeño

Aplicación	Tasa de frames	Resolución	Profundidad del píxel
Vigilancia	5	640×480	12
Videoteléfono	10	320×240	12
Multimedia	15	320×240	16
TV analógica	25	640×480	16
HDTV (720p)	60	1280×720	24
HDTV (1080i)	60	1920×1080	24
HDTV (1080p)	30	1920×1080	24

Tabla 6.6: Parámetros de vídeo para aplicaciones típicas.

cambio en la posición de la escena y de la cámara entre fotogramas consecutivos. En consecuencia, la jerarquía de cinco pasos es:

$$\text{pieza} \rightarrow \text{acto} \rightarrow \text{escena} \rightarrow \text{secuencia} \rightarrow \text{frame}.$$

6.3.1. Televisión de alta definición

El estándar NTSC fue creado en la década de 1930, para las transmisiones de televisión en blanco y negro. NTSC es un acrónimo de National Television Standards Committee (Comité Nacional de eStándares de Televisión). Éste es un estándar que especifica la forma de la señal enviada por un transmisor de televisión. La señal es analógica, con una amplitud que va de arriba abajo en cada línea de escaneo en respuesta a las partes en negro y blanco de la línea. Cuando se incorporó el color en el presente estándar, en 1953, tuvo que ser añadido de forma que los aparatos de televisión en blanco y negro fueran capaces de mostrar la señal de color en blanco y negro. El resultado fue modulación de fase de la portadora del blanco y negro, una chapuza —los ingenieros de televisión se refieren a ella en tono de broma como NSCT (Never the Same Color Twice), "nunca el mismo color dos veces"—.

Con la proliferación de las computadoras y equipos digitales en las dos últimas décadas llegó la constatación de que una señal digital es una manera mejor y más confiable para enviar imágenes a través del aire. En tal señal, la imagen se envía píxel a píxel, donde cada píxel es representado por un número que especifica su color. La señal digital es todavía una onda, pero la amplitud de la onda ya no representa la imagen. Más bien, la onda es *modulada* para llevar información binaria. El término modulación significa que algo en la onda es modificado para distinguir entre los unos y ceros que se envían. Una señal digital FM, por ejemplo, modifica (modula) la frecuencia de la onda. Este tipo de onda utiliza una frecuencia para representar un 0 binario y otra para representar un 1. El estándar DTV (TV digital) usa una técnica de modulación llamada 8-VSB (de *vestigial sideband* o *banda secundaria residual*), que proporciona una ransmisión terrestre robusta y fiable. La técnica de modulación 8-VSB permite una amplia área de cobertura, reduce las interferencias con las transmisiones analógicas existentes, y es en sí misma inmune a las interferencias.

Historia de la DTV: El Comité de Sistemas de Televisión Avanzados (ATSC), establecido en 1982, es una organización internacional que desarrolla estándares técnicos para los sistemas de video avanzados. A pesar de que estos estándares son voluntarios, por lo general son adoptados por los miembros del ATSC y otros fabricantes. Actualmente existen alrededor de ochenta empresas y organizaciones miembros del ATSC, que representan las múltiples facetas de la televisión, el computador, el teléfono, y las industrias cinematográficas.

El estándar de televisión digital ATSC adoptado por la comisión Federal de comunicaciones de Estados Unidos (FCC o United States Federal Communications Commission) se basa en un diseño de la Gran Alianza (una coalición de fabricantes de electrónica e institutos de investigación), que

fue finalista en la primera ronda de Propuestas DTV bajo el Comité Consultivo de la FCC sobre sistemas avanzados de televisión (ACATS). Los ACATS están compuestos por representantes de los ordenadores, la radiodifusión, las telecomunicaciones, la manufactura, la televisión por cable, y las industrias cinematográficas. Su misión es la de contribuir a la adopción de un estándar de transmisión HDTV y promover la rápida implementación de la HDTV en los EE.UU.

Los ACATS anunciaron un concurso abierto: cualquiera podría presentar una propuesta para el estándar HDTV, y el mejor sistema sería seleccionado como el nuevo estándar de televisión para los Estados Unidos. Para asegurar una transición rápida a la HDTV, la FCC prometió que todas las estaciones de televisión del país deberían prestar temporalmente un canal adicional del espectro de transmisión.

Los ACATS trabajaron con el ATSC para revisar el estándar DTV propuesta y dio su visto bueno a las especificaciones finales para el formato de las diversas partes —audio, transporte, formato, compresión, y transmisión. El ATSC documentó el sistema como un estándar, y los ACATS adoptaron el sistema de la Gran Alianza, en su recomendación a la FCC a finales de 1995.

A finales de 1996, los miembros corporativos del ATSC habían llegado a un acuerdo sobre la Estándar DTV (Documento A/53) y pidió a la FCC aprobarlo. El 31 de diciembre, 1996, la FCC adoptó formalmente todos los aspectos del estándar ATSC, excepto para los formatos de video. Estos formatos de vídeo, sin embargo siguen siendo una parte del estándar ATSC, y se espera que sean utilizado por los organismos de radiodifusión y por los fabricantes de televisión en el futuro próximo.

Especificaciones de la HDTV: El estándar NTSC, en uso desde la década de 1930, especifica una imagen entrelazada compuesta por 525 líneas en la que las líneas impares $(1, 3, 5, \ldots)$ son dibujadas en la pantalla primero, seguidas por las líneas pares $(2, 4, 6, \ldots)$. Los dos campos son entrelazados y dibujados en $1/30$ de un segundo, lo que permite 30 actualizaciones (*refrescos*) de pantalla cada segundo. Por el contrario, una imagen no entrelazada muestra la imagen completa línea a línea. Este tipo de *escaneo progresivo* de la imagen es el utilizado por los monitores de ordenador de hoy en día.

Los aparatos de televisión digitales, que han estado disponibles desde mediados de 1998, utilizan una proporción de aspecto de 16/9 y pueden visualizar tanto imágenes entrelazadas como de escaneo progresivo en diversas resoluciones distintas —una de las mejores características del video digital—. Estos formatos incluyen: escaneo progresivo de 525 líneas (525P), escaneo progresivo de 720 líneas (720p), escaneo progresivo de 1050 líneas (1050P), y entrelazado de 1080 (1080I), todos con píxeles cuadrados.

Nuestros televisores analógicos actuales, no pueden hacer frente a la nueva señal digital emitida por las estaciones de televisión, pero ya están disponibles convertidores de bajo costo (en forma de una pequeña caja que puede situarse cómodamente en la parte superior de un aparato de televisión) para convertir las señales digitales en analógicas (y perder información de la imagen en el proceso).

El estándar NTSC pide 525 líneas de escaneo y una relación de aspecto de 4/3. Ésto implica $\frac{4}{3} \times 525 = 700$ píxeles por línea, lo que arroja un total de $525 \times 700 = 367\,500$ píxeles en la pantalla. (Éste es el total teórico, en la práctica sólo son en realidad visibles 483 líneas.) En comparación, un formato de DTV que demanda el escaneo de 1080 líneas y una relación de aspecto de 16/9 es equivalente a 1920 píxeles por línea, llevando el número total de píxeles a $1080 \times 1920 = 2\,073\,600$, alrededor de 5,64 veces más que el estándar NTSC entrelazado.

◇ **Ejercicio 6.4 (sol. en pág. 1105):** La relación de aspecto de NTSC es $4/3 = 1{,}33$ y la de DTV es $16/9 = 1{,}77$. ¿Cuál se ve mejor?

Además del formato de DTV 1080×1920, el estándar DTV ATSC demanda un formato de baja resolución, con sólo 720 líneas de escaneo, lo que implica $\frac{16}{9} \times 720 = 1280$ píxeles por línea. Cada una de estas resoluciones puede ser refrescada con una de las tres diferentes tasas: 60 $^{\text{frames}}/_{\text{segundo}}$ (para video en vivo) y 24 ó 30 $^{\text{frames}}/_{\text{segundo}}$ (para el material originalmente producido en las películas). Las frecuencias de refresco pueden considerarse *resolución temporal*. El resultado es un total de

líneas × píxeles	# total de píxeles	tasa de refresco 24	30	60
1080 × 1920	2 073 600	1 194 393 600	1 492 992 000	2 985 984 000
720 × 1280	921 600	530 841 600	663 552 000	1 327 104 000

Tabla 6.7: Resoluciones y capacidades de seis formatos de DTV.

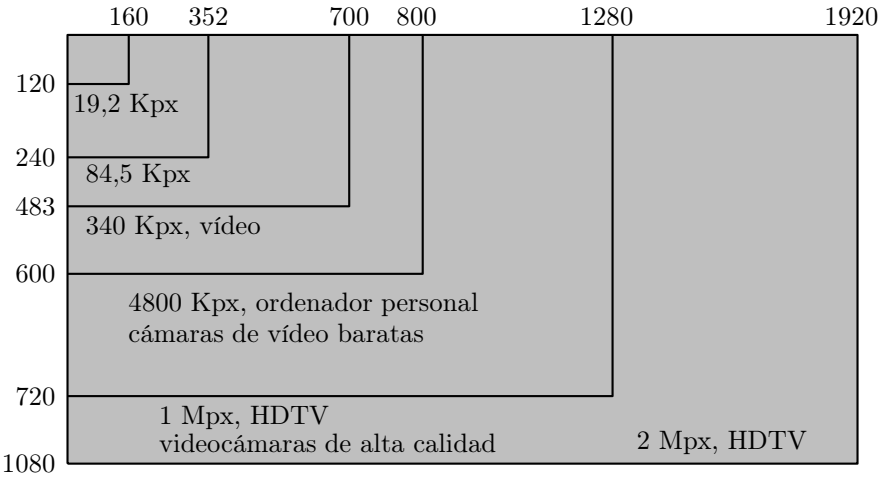


Figura 6.8: Diversas resoluciones de vídeo.

seis formatos diferentes. La Tabla 6.7 resume las capacidades de pantalla y las tasas de transmisión necesarias para los seis formatos. En alta resolución y 60 frames por segundo el transmisor debe ser capaz de enviar 2 985 984 000 $^{\text{bits}}/_{\text{seg.}}$ (alrededor de 356 $^{\text{Mbytes}}/_{\text{seg.}}$), por lo que este formato utiliza compresión. (Emplea la compresión MPEG-2. Otros formatos de vídeo también pueden utilizar este método de compresión.) El hecho de que la DTV puede tener diferentes escalas espaciales y resoluciones temporales permite compensaciones. Ciertos tipos de materiales de vídeo (como como un caballo en movimiento rápido o una carrera de autos) pueden verse mejor a altas frecuencias de actualización, e incluso a baja resolución espacial, mientras que otros materiales (como las pinturas de alta calidad de un museo) deben idealmente ser vistos en alta resolución, incluso con bajas tasas de refresco.

Televisión Digital (DTV) es un término amplio que abarca todos los tipos de transmisión digital. HDTV es un subconjunto de la DTV que implica 1080 líneas de exploración o escaneo. Otro tipo de DTV es la televisión de definición estándar (SDTV), que tiene una calidad de imagen ligeramente mejor que una buena fotografía analógica. (SDTV tiene una resolución de 640 × 480 a 30 $^{\text{frames}}/_{\text{segundo}}$ y una relación de aspecto de 4:3.) Puesto que la generación de una imagen SDTV requiere un menor número de píxeles, una estación de transmisión debe ser capaz de transmitir múltiples canales de SDTV dentro de su rango de frecuencias de 6 MHz permitido. HDTV también incorpora tecnología de sonido Dolby Digital para reunir una presentación completa.

La Figura 6.8 muestra las resoluciones más importantes que se utilizan en los diferentes sistemas de vídeo. Sus capacidades van desde 19 000 píxeles a más de dos millones de píxeles.

A veces las cámaras y la televisión son buenas para la gente y, a veces no lo son. Yo no sé si es la forma en que lo dice, o cómo la ve.

—Dan Quayle

6.4. Compresión de video

La compresión de vídeo se basa en dos principios. El primero es la redundancia espacial que existe en cada fotograma o frame. El segundo es el hecho de que la mayor parte del tiempo, un frame de vídeo es muy similar a sus vecinos inmediatos. Ésto se llama *redundancia temporal*. Una técnica de compresión de video típica, por lo tanto, debe comenzar codificando el primer frame usando un método de compresión para imágenes estáticas. Después, debe codificar cada frame sucesivo identificando las diferencias entre el frame y su predecesor, y codificando estas diferencias. Si un frame es muy diferente de su predecesor (como sucede con el primer fotograma de una secuencia), debe ser codificado independientemente de cualquier otro marco. En la literatura sobre compresión de vídeo, un frame que es codificado usando su predecesor se llama *inter frame* (o simplemente *inter*), mientras que un frame que es codificado de forma independiente se llama *intra frame* (o simplemente *intra*).

La compresión de vídeo es normalmente con pérdidas. La codificación de un frame F_i en términos de su predecesor F_{i-1} introduce algunas distorsiones. Como resultado, la codificación del frame siguiente F_{i+1} en términos del (ya deformado) F_i aumenta la distorsión. Incluso en la compresión de vídeo sin pérdidas, un frame puede perder algunos bits. Ésto puede ocurrir durante la transmisión o después de una larga exposición. Si un frame F_i ha perdido algunos bits, entonces todos los frames que le siguen, hasta el próximo frame intra, es decodificado incorrectamente, dando incluso lugar a errores acumulados. Es por ésto que los frames intra deben ser usados de vez en cuando en el interior de una secuencia, no sólo en su inicio. Un frame intra se etiqueta I, y un frame inter se etiqueta P (por *predictivo*).

Una vez que se capta esta idea, es posible generalizar el concepto de un inter frame. Tal frame puede ser codificado basado en uno de sus predecesores y también en uno de sus *sucesores*. Sabemos que un codificador no debe utilizar toda la información que no está disponible para el decodificador, pero la compresión de vídeo es especial debido a las grandes cantidades de datos involucrados. Por lo general, no nos importa si el codificador es lento, pero el decodificador tiene que ser rápido. Un caso típico es la grabación de vídeo en un disco duro o en un DVD, para ser reproducidos. El codificador puede tardar minutos u horas en codificar los datos. El decodificador, sin embargo, tiene que reproducirlo de nuevo a la tasa de frames correcta (tantos frames por segundo), por lo que tiene que ser rápido. Ésta es la razón por la que un decodificador de vídeo típico funciona en paralelo. Dispone de varios circuitos de decodificación trabajando simultáneamente en varios frames.

Con esto en mente, es fácil imaginar una situación en la que el codificador codifica el frame 2 basándose tanto en el frame 1 como en el 3, y escribe los frames en el stream comprimido en el orden 1, 3, 2. El decodificador los lee en este orden, decodifica los frames 1 y 3 en paralelo, emite el frame 1, y después decodifica el frame 2 basándose en los frames 1 y 3. Naturalmente, los frames deben estar claramente etiquetados (o marcados en el tiempo). Un frame que es codificado basándose tanto en frames pasados como futuros se etiqueta como B (por *bidireccional*).

La predicción de un frame basándose en su sucesor tiene sentido en los casos donde el movimiento de un objeto en la película descubre gradualmente una zona del fondo. Tal zona puede ser sólo parcialmente conocida en el frame actual, pero puede ser mejor conocida en el frame posterior. En consecuencia, el frame siguiente es un candidato natural para la predicción de esta área en el frame actual.

La idea de un frame B es tan útil que la mayor parte de los frames en una presentación de vídeo comprimido puede ser de este tipo. Por consiguiente, terminamos con una secuencia de frames comprimidos de los tres tipos: I, P, y B. Un frame I es decodificado independientemente de cualquier otro frame. Un frame P es decodificado usando el frame anterior I o P. Un frame B es decodificado utilizando los frames anterior y posterior I o P. La Figura 6.9a muestra una secuencia de tales frames en el orden en el que son generados por el codificador (e introducidos por el decodificador). La Figura 6.9b muestra la misma secuencia en el orden en el que los frames son emitidos y mostrados por el decodificador. El frame etiquetado como 2 debe ser mostrado después del frame 5, por lo que cada

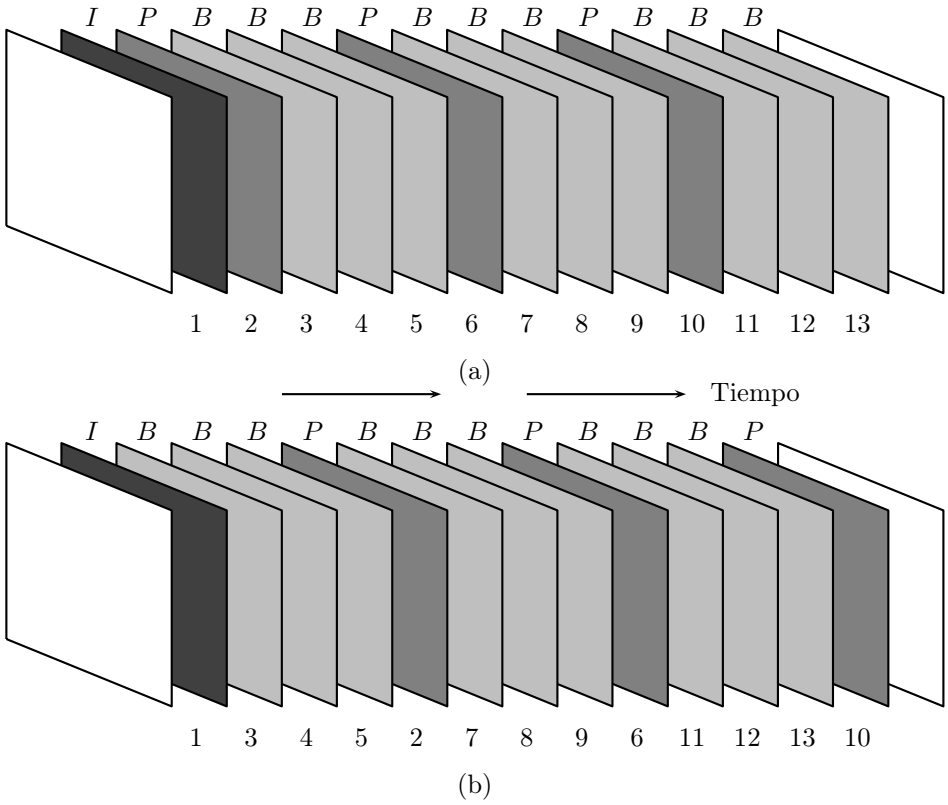


Figura 6.9: (a) Orden de codificación. (b) Orden de visualización.

frame debe tener dos marcas de tiempo, su tiempo de codificación y su tiempo de visualización.

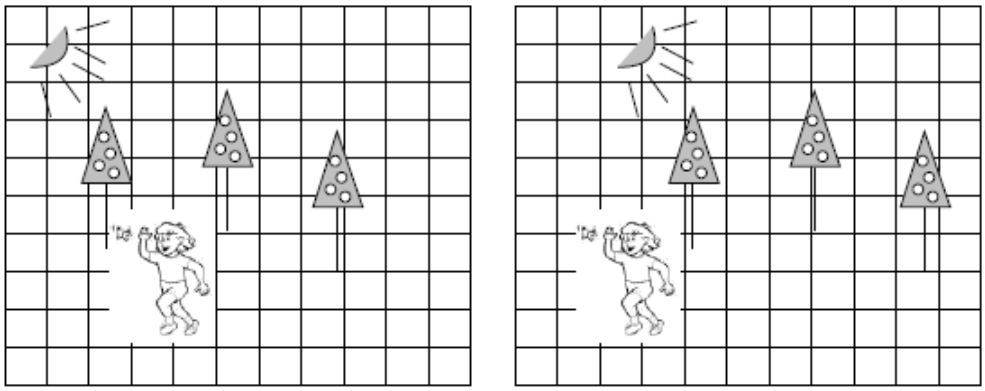
Comenzamos con algunos métodos de compresión de vídeo intuitivos.

Submuestreo: El codificador selecciona cada frame y lo escribe en el stream comprimido. Ésto produce un factor de compresión de 2. El decodificador introduce y un frame y lo duplica para crear dos frames.

Diferenciación: Un frame es comparado con su predecesor. Si la diferencia entre ellos es pequeña (de sólo unos pocos píxeles), el codificador codifica los píxeles que son diferentes escribiendo tres números en el stream comprimido para cada píxel: sus coordenadas de imagen, y la diferencia entre los valores de píxel en los dos marcos. Si la diferencia entre los frames es grande, el frame actual es escrito en la salida tal cual (en formato raw). Compárese este método con la codificación relativa, Sección 1.3.1.

Una versión de diferenciación con pérdidas observa en cuánto cambia un píxel. Si la diferencia entre las intensidades de un píxel en el frame precedente y el actual es menor que un umbral determinado (controlado por el usuario), el píxel no se considera diferente.

Diferenciación de bloque: Ésta es una mejora adicional de la diferenciación. La imagen se divide en bloques de píxeles, y cada bloque B en el frame actual se compara con el bloque correspondiente P en el frame precedente. Si los bloques difieren en más de una cierta cantidad, entonces B es comprimido escribiendo su coordenadas de imagen, seguidas por los valores de todos sus píxeles (expresados como diferencias) en el stream comprimido. La ventaja es que las coordenadas del bloque son números pequeños (menores que las coordenadas de un píxel), y estas coordenadas tienen que escribirse una sola vez para todo el bloque. Como parte negativa, los valores de todos los píxeles del bloque, incluso



(a) (b)

Figura 6.10: Compensación de movimiento.

los que no han cambiado, tienen que ser escritos en la salida. Sin embargo, puesto que estos valores se expresan como diferencias, son números pequeños. Consecuentemente, este método es sensible al tamaño del bloque.

Compensación de movimiento: Cualquiera que haya visto las películas sabe que la diferencia entre fotogramas consecutivos es pequeña, ya que es el resultado del movimiento de la escena, la cámara, o ambos entre frames. Por consiguiente, esta característica puede ser explotada para lograr una mejor de compresión. Si el codificador descubre que una parte P del frame precedente ha sido movido estrictamente a una ubicación diferente en el frame actual, entonces P puede ser comprimido escribiendo los tres elementos siguientes en el stream comprimido: su localización previa, su ubicación actual y la información que identifica los límites de P. La siguiente discusión sobre la compensación de movimiento se basa en [Manning 98].

En principio, dicha parte puede tener cualquier forma. En la práctica, estamos limitados a bloques de igual tamaño (normalmente cuadrados, pero también pueden ser rectangulares). El codificador analiza el frame actual bloque a bloque. Para cada bloque B busca en el frame precedente un bloque idéntico C (si la compresión es sin pérdidas) o uno similar (si puede ser con pérdidas). Cuando descubre tal bloque, el codificador escribe la diferencia entre sus ubicaciones pasadas y presentes en la salida. Esta diferencia es de la forma

$$(C_x - B_x, C_y - B_y) = (\Delta x, \Delta y)\,,$$

por lo que se llama *vector de movimiento*. Figura 6.10a,b muestra un sencillo ejemplo donde el sol y los árboles se mueven estrictamente a la derecha (a causa del movimiento de la cámara), mientras que el niño se mueve una distancia diferente hacia la izquierda (este es el movimiento de la escena).

La compensación de movimiento es efectivo si los objetos únicamente se trasladan, no son escalados o rotados. Los cambios drásticos en la iluminación de un frame a otro también reducen la efectividad de este método. En general, la compensación de movimiento es con pérdidas. Los siguientes párrafos discuten los principales aspectos de la compensación de movimiento en detalle.

Segmentación del frame: El frame actual se divide en bloques de igual tamaño no superpuestos. Los bloques pueden ser cuadrados o rectángulos. La última opción presupone que movimiento en el vídeo es principalmente horizontal, de modo que los bloques horizontales reducen el número de vectores de movimiento sin degradar la relación de compresión. El tamaño del bloque es importante, ya que los bloques grandes reducen la posibilidad de encontrar una coincidencia, y los bloques pequeños generan

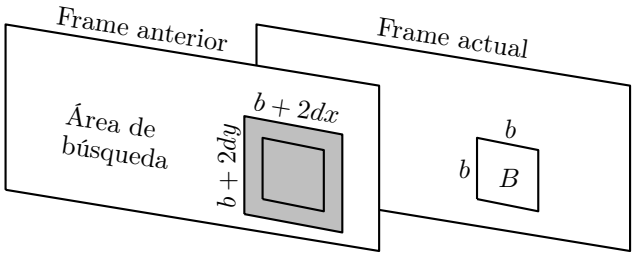


Figura 6.11: Área de búsqueda.

muchos vectores de movimiento. En la práctica se utilizan tamaños de bloque que son potencias enteras de 2, tales como 8 ó 16, ya que ésto simplifica el software.

Búsqueda de un umbral: Cada bloque B en el frame actual es comparado primero con su contraparte C en el frame precedente. Si son idénticos, o si la diferencia entre ellos es menor que un umbral predefinido, el codificador asume que el bloque no ha sido movido.

Búsqueda de bloque: Este es un proceso que consume tiempo, y por lo tanto tiene que ser diseñado cuidadosamente. Si B es el bloque actual en el frame actual, el frame anterior tiene que ser explorado para encontrar un bloque idéntico o muy parecido a B. La búsqueda está normalmente restringida a un área pequeña (llamada *zona o área de búsqueda*) alrededor de B, definida por los parámetros de *desplazamiento máximo* dx y dy. Estos parámetros especifican las distancias horizontal y vertical máximas, en píxeles, entre B y cualquier bloque con correspondencia en el frame anterior. Si B es un cuadrado de lado b, el área de búsqueda contendrá $(b + 2dx)\,(b + 2dy)$ píxeles (Figura 6.11) y estará formado por $(2dx + 1)\,(2dy + 1)$ cuadrados diferentes de $b \times b$, superpuestos. El número de bloques candidatos en esta área es, por lo tanto, proporcional a $dx \cdot dy$.

Medida de distorsión: Esta es la parte más sensible del codificador. La distorsión medida selecciona la mejor correspondencia para el bloque B. Tiene que ser sencillo y rápido, pero también fiable. Algunas opciones —similares a las de la Sección 4.14— son discutidas aquí.

La *diferencia absoluta media* (o *error absoluto medio*) calcula el promedio de las diferencias absolutas entre un píxel B_{ij} en B y su contraparte C_{ij} en un bloque candidato C:

$$\frac{1}{b^2}\sum_{i=1}^{b}\sum_{j=1}^{b}|B_{ij} - C_{ij}|\,.$$

Ésto involucra b^2 sustracciones y operaciones de valor absoluto, b^2 adiciones, y una división. Esta medida se calcula para cada uno de los $(2dx + 1)\,(2dy + 1)$ bloques candidatos de $b \times b$ superpuestos distintos, y la distorsión más pequeña (digamos, para el bloque C_k) es examinado. Si es menor que el umbral de búsqueda, entonces se selecciona C_k como la pareja para B. De lo contrario, no hay correspondencia para B, y B tiene que ser codificado sin compensación de movimiento.

◇ **Ejercicio 6.5 (sol. en pág. 1105):** ¿Cómo puede suceder tal cosa? ¿Cómo puede un bloque en el frame actual no tener correspondencia en el frame precedente?

La *diferencia cuadrática media* es una medida similar, donde se calcula el cuadrado, en lugar del valor absoluto, de una diferencia de píxeles:

$$\frac{1}{b^2}\sum_{i=1}^{b}\sum_{j=1}^{b}(B_{ij} - C_{ij})^2\,.$$

La medida *clasificación por diferencia de pels* (PDC o *pel difference classification*) cuenta cuántas diferencias $|B_{ij} - C_{ij}|$ son más pequeñas que el parámetro p de PDC.

La medida de *proyección integral* calcula la suma de una fila de B y la resta de la suma de la fila correspondiente de C. El valor absoluto de la diferencia es añadido al valor absoluto de la diferencia de la suma de columnas:

$$\sum_{i=1}^{b}\left|\sum_{j=1}^{b}B_{ij}-\sum_{j=1}^{b}C_{ij}\right|+\sum_{j=1}^{b}\left|\sum_{i=1}^{b}B_{ij}-\sum_{i=1}^{b}C_{ij}\right|.$$

Métodos de búsqueda subóptimos: Estos métodos buscan algunos, en lugar de todos, los bloques candidatos en el área $(b+2dx)\,(b+2dy)$. Aceleran la búsqueda para una coincidencia de bloque, a expensas de la eficiencia de la compresión. Varios de tales métodos son discutidos en detalle en la Sección 6.4.1.

Corrección del vector de movimiento: Una vez que un bloque C ha sido seleccionado como la mejor correspondencia para B, se calcula un vector de movimiento como la diferencia entre la esquina superior izquierda de C y la esquina superior izquierda de B. Independientemente de cómo fue determinada la concordancia, el vector de movimiento puede ser erróneo debido al ruido, mínimos locales en el frame, o porque el algoritmo de emparejamiento no es perfecto. Es posible aplicar técnicas de suavizado a los vectores de movimiento después de haber sido calculados, en un intento de mejorar la correspondencia. Las correlaciones espaciales en la imagen sugieren que los vectores de movimiento también deben estar correlacionados. Si se encuentran ciertos vectores que violan ésto, pueden ser corregidos.

Este paso es costoso y puede incluso ser contraproducente. Una presentación de video puede implicar un movimiento lento y suave de la mayoría de los objetos, pero también un movimiento rápido y desigual de algunos objetos pequeños. La corrección de vectores de movimiento puede interferir con los vectores de movimiento de tales objetos y provocar distorsiones en los frames comprimidos.

Codificación de los vectores de movimiento: Una gran parte del frame actual (quizás cerca de la mitad del mismo) puede ser convertido en vectores de movimiento, por lo que la forma en que estos vectores son codificados es crucial; también debe ser sin pérdidas. Dos propiedades de los vectores de movimiento ayudan en su codificación: (1) Están correlacionados y (2) su distribución no es uniforme. A medida que escaneamos el frame bloque por bloque, los bloques adyacentes tienen vectores de movimiento que normalmente no difieren en mucho; están correlacionados. Los vectores tampoco apuntan en todas las direcciones. A menudo existen una o dos direcciones preferidas a las que todos o la mayoría vectores de movimiento apuntan; los vectores no están distribuidos uniformemente.

Ningún método ha demostrado ser ideal para la codificación de los vectores de movimiento. La codificación aritmética, la codificación de Huffman adaptativa, y varios códigos prefijo han sido probados, y todos parecen desenvolverse bien. Aquí hay dos métodos diferentes que pueden funcionar mejor:

1. Predecir un vector de movimiento basándose en sus predecesores en la misma fila y sus predecesores en la misma columna del frame actual. Calcular la diferencia entre la predicción y el vector real, y codificarlo con Huffman. Este algoritmo es importante. Se utiliza en MPEG y otros métodos de compresión.

2. Agrupar los vectores de movimiento en bloques. Si todos los vectores en un bloque son idénticos, el bloque es codificado mediante la codificación de este vector. Los otros bloques se codifican como en el punto 1 de arriba. Cada bloque codificado comienza con un código de identificación de su tipo.

Codificación del error de predicción: La compensación de movimiento es con pérdidas, ya que un bloque B normalmente se corresponde con un bloque algo diferente C. La compresión puede ser mejorada mediante la codificación de las diferencias entre los frames sin comprimir y comprimidos actuales bloque a bloque, y sólo para los bloques que difieran mucho. Ésto se efectúa generalmente mediante la codificación por transformada. La diferencia es escrita en la salida, después de cada frame, y es utilizada por el decodificador para mejorar el frame después de haber sido decodificado.

6.4.1. Métodos de búsqueda subóptimos

La compresión de vídeo requiere muchos pasos y cálculos, por lo que los investigadores han ido buscando optimizaciones y algoritmos más rápidos, especialmente para las etapas que involucran muchos cálculos. Una de tales medidas es la búsqueda de un bloque C en el frame anterior para que empareje con un bloque B dado en el frame actual. Una búsqueda exhaustiva consume mucho tiempo, por lo que compensa explorar métodos de búsqueda óptimos que busquen sólo algunos de los muchos bloques superpuestos candidatos. Estos métodos no siempre encuentran la mejor coincidencia, pero generalmente pueden acelerar el proceso de compresión, mientras incurren solamente en una pequeña pérdida de eficiencia en la compresión.

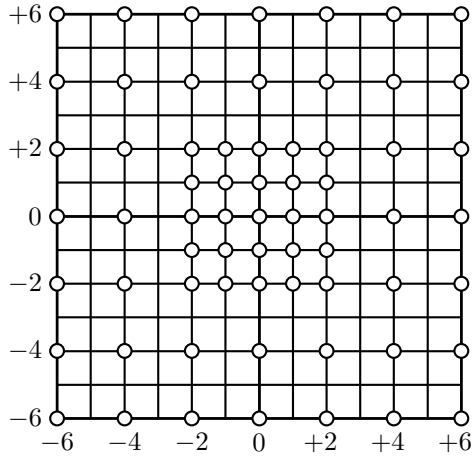
Métodos basados en firmas: Este método realiza una serie de pasos, restringiendo el número de bloques de candidatos en cada paso. En el primer paso, todos los bloques candidatos se buscan usando una medida de distorsión simple y rápida como la clasificación por diferencia de pels. Sólo los mejores bloques emparejados son incluidos en la siguiente etapa, donde se evalúan mediante una medida de distorsión más restrictiva, o con la misma medida, pero con un parámetro menor. Un método de firma puede implicar varios pasos, usando diferentes medidas de distorsión en cada uno.

Búsqueda diluida en distancia: Sabemos por experiencia que los objetos en movimiento rápido se ven borrosos en una animación, incluso si son nítidos en todos los frames. Ésto sugiere una manera de perder datos. Podemos requerir un buen bloque de emparejamiento para objetos de movimiento lento, pero permitir una peor correspondencia para los de movimiento rápido. El resultado es un algoritmo de coincidencia de bloques que busca en todos los bloques cercanos a B, pero cada vez menos bloques a medida que la búsqueda se aleja de B. La Figura 6.12a muestra cómo puede funcionar tal método para unos parámetros de desplazamiento máximos $dx = dy = 6$. El número total de bloques C que se buscan va desde $(2dx + 1) \cdot (2dy + 1) = 13 \times 13 = 169$ a sólo 65, ¡menos de un 39 %!

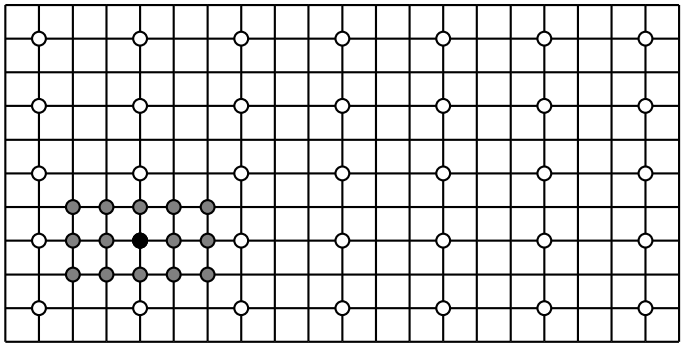
Búsqueda basada en la localidad: Este método se basa en la asunción de que una vez se ha descubierto un buen emparejamiento, probablemente incluso mejores correspondencias tienden a ubicarse cerca del mismo (recuerde que los bloques C se buscaron para coincidencias altamente superpuestas). Un algoritmo obvio es empezar a buscar un emparejamiento en un conjunto de bloques disperso, luego utilizar el bloque C que mejor empareja como el centro de una segunda oleada de búsquedas, esta vez en un conjunto de bloques más denso. La Figura 6.12b muestra dos olas de búsqueda, la primera considera bloques muy separados entre sí, seleccionando uno como el mejor emparejamiento. La segunda ola busca cada bloque en las proximidades de la mejor coincidencia.

Búsqueda monótona de cuadrantes: Esta es una variante de una búsqueda basada en la localidad. Comienza con un conjunto disperso de bloques C que se buscan para un emparejamiento. La medida de la distorsión es calculada para cada uno de los bloques, y el resultado es un conjunto de valores de distorsión. La idea es que los valores de distorsión aumentan a medida que nos alejamos del mejor emparejamiento. Examinando el conjunto de valores de distorsión obtenidos en el primer paso, el segundo paso puede predecir dónde es probable que se encuentre el mejor emparejamiento. La Figura 6.13 muestra cómo la búsqueda de una región de 4×3 bloques sugiere una dirección bien definida en la que seguir buscando.

Este método es menos seguro que los anteriores porque la dirección propuesta por el conjunto de valores de distorsión puede conducir a un mejor bloque local, mientras que *el* mejor bloque puede



(a)



○ — primera ola. ● — mejor emparejamiento de primera ola. ● — segunda ola.

(b)

Figura 6.12: (a) Búsqueda diluida en distancia $dx = dy = 6$. (b) Una búsqueda local.

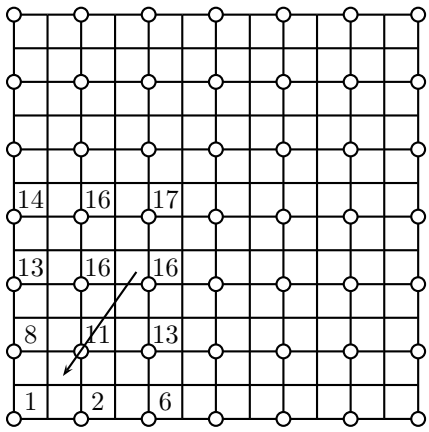


Figura 6.13: Búsqueda monótona de cuadrantes.

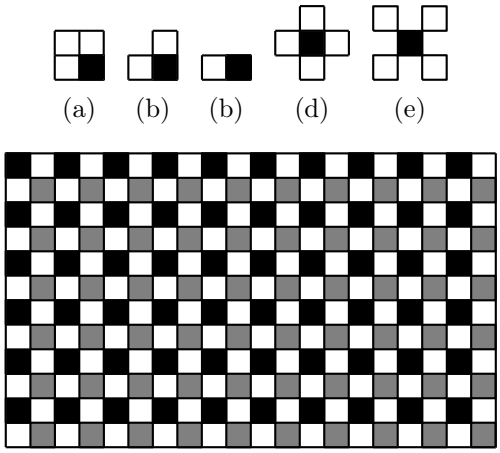


Figura 6.14: Estrategias para algoritmos de búsqueda espaciales dependientes.

estar ubicado en otro lugar.

Algoritmos dependientes: Como se mencionó anteriormente, el movimiento en un frame es el resultado, ya sea del movimiento de la cámara, ya sea del movimiento del objeto. Si asumimos que los objetos en el frame son más grandes que un bloque, concluimos que es razonable esperar que los vectores de movimiento de los bloques adyacentes estén correlacionados. Por consiguiente, el algoritmo de búsqueda puede comenzar estimando el vector de movimiento de un bloque B a partir de los vectores de movimiento que ya han sido encontrados para sus vecinos, y luego mejorar esta estimación comparando B con algunos bloques candidatos C. Ésta es la base de varios *algoritmos dependientes*, que pueden ser espaciales o temporales.

Dependencia espacial: En un algoritmo de dependencia espacial, los vecinos de un bloque B en el frame actual se utilizan para estimar el vector de movimiento de B. Éstos, por supuesto, deben ser vecinos cuyos vectores de movimiento ya han sido calculados. La mayoría de los bloques tienen ocho vecinos cada uno, pero considerar los ocho puede no ser la mejor estrategia (además, cuando un bloque B es considerado, sólo algunos de sus vecinos pueden tener ya calculados sus vectores de movimiento). Si los bloques son emparejados en orden raster, entonces tiene sentido usar uno, dos, o tres vecinos previamente emparejados, como se muestra en la Figura 6.14a,b,c. Debido a la simetría, sin embargo, es mejor utilizar cuatro vecinos simétricos, como en la Figura 6.14d,e. Ésto puede realizarse mediante un método de tres pasadas que escanea los bloques como en la Figura 6.14f. La primera pasada escanea todos los bloques que se muestran en negro (una cuarta parte de los bloques en el frame). Los vectores de movimiento para esos bloques se calculan mediante algún otro método. La segunda pasada escanea los bloques mostrados en gris (el 25 % de los bloques) y calcula un vector de movimiento para cada uno usando los vectores de movimiento de sus cuatro vecinos en las esquinas. Los bloques blancos (el 50 % restante) se escanean en la tercera pasada, y el vector de movimiento de cada uno se estima utilizando los vectores de movimiento de sus vecinos en los cuatro lados. Si los vectores de movimiento de los vecinos son muy diferentes, no deben ser usados, y se calcula un vector de movimiento para el bloque B con un método diferente.

Dependencia temporal: El vector de movimiento del bloque B en el frame actual puede ser estimado como el vector de movimiento del mismo bloque en el frame previo. Ésto tiene sentido si podemos asumir un movimiento uniforme. Después de que el vector de movimiento de B es estimado de esta manera, debe mejorarse y corregirse utilizando otros métodos.

Métodos de búsqueda monótonos de más cuadrantes: Los siguientes métodos subóptimos de emparejamiento de bloques utilizan la asunción principal del método de búsqueda monótona de cuadrantes.

Búsqueda logarítmica bidimensional: Este método multipasada reduce la zona de búsqueda en cada pasada hasta que se restringe a un bloque. Asumimos que el bloque actual B se encuentra en la posición (a, b) en el frame actual. Esta posición se convierte en el centro inicial de la búsqueda. El algoritmo utiliza un parámetro distancia d que define el área de búsqueda. Este parámetro está controlado por el usuario y tiene un valor predeterminado. El área de búsqueda se compone de los $(2d + 1) \times (2d + 1)$ bloques centrados en el bloque actual B.

- *Paso 1:* Se calcula un tamaño de paso s mediante

$$s = 2^{\lfloor \log_2 d \rfloor - 1},$$

  y el algoritmo compara B con los cinco bloques en las posiciones (a, b), $(a, b + s)$, $(a, b - s)$, $(a + s, b)$, y $(a - s, b)$ del frame anterior. Estos cinco bloques forman el patrón de un signo más "+".

- *Paso 2:* Se selecciona el mejor emparejamiento entre los cinco bloques. Denotamos la posición de este bloque por (x, y). Si $(x, y) = (a, b)$, entonces s se reduce a la mitad (ésta es la razón del nombre *logarítmica*). De lo contrario, s sigue siendo el mismo, y el centro (a, b) de la búsqueda se mueve a (x, y).

- *Paso 3:* Si $s = 1$, entonces se buscan los nueve bloques alrededor del centro (a, b) de la búsqueda, y el mejor emparejamiento entre ellos se convierte en el resultado del algoritmo. De lo contrario el algoritmo vuelve al paso 2.

Cualquier bloque que necesite ser buscado, pero que se encuentre fuera del área de búsqueda es ignorado y no es utilizado en la búsqueda. La Figura 6.15 ilustra el caso donde $d = 8$. Por simplicidad asumimos que el bloque actual B tiene como coordenadas de frame $(0, 0)$. La búsqueda se limita al área del bloque de (17×17) centrado en el bloque B. El paso 1 calcula

$$s = 2^{\lfloor \log_2 8 \rfloor - 1} = 2^{3-1} = 4,$$

y busca los cinco bloques (etiquetados 1) en las ubicaciones $(0, 0)$, $(4, 0)$, $(-4, 0)$, $(0, 4)$, y $(0, -4)$. Suponemos que el mejor emparejamiento de estos cinco está en $(0, 4)$, por lo que éste se convierte en el nuevo centro de la búsqueda, y los tres bloques (etiquetados 2) en las ubicaciones $(4, -4)$, $(4, 4)$, y $(8, 0)$ son buscados en el segundo paso.

Suponiendo que el mejor emparejamiento entre estos tres está localizado en $(4, 4)$, el siguiente paso busca los dos bloques (etiquetados 3) en las ubicaciones $(8, 4)$ y $(4, 8)$, el bloque (etiquetado 2) en $(4, 4)$, y los bloques "1" en $(0, 4)$ y $(4, 0)$.

◇ **Ejercicio 6.6 (sol. en pág. 1105):** Suponiendo que $(4, 4)$ es de nuevo el mejor emparejamiento, úsese la figura para describir el resto de la búsqueda.

Búsqueda de tres pasos: Ésta es algo similar a la búsqueda logarítmica bidimesional. En cada paso se testan ocho bloques, en lugar de cuatro, en torno al centro de búsqueda, luego se reduce el tamaño de paso a la mitad. Si inicialmente $s = 3$, el algoritmo termina tras tres pasos, de ahí su nombre.

Búsqueda ortogonal: Ésta es una variación de tanto la búsqueda logarítmica bidimensional como la búsqueda de tres pasos. Cada paso de la búsqueda ortogonal involucra una búsqueda horizontal y una vertical. El tamaño de paso s se inicializa a $\lfloor (d + 1)/2 \rfloor$, y se buscan, el bloque en el centro de la búsqueda y los dos bloques candidatos situados a cada lado del mismo a una distancia s. La

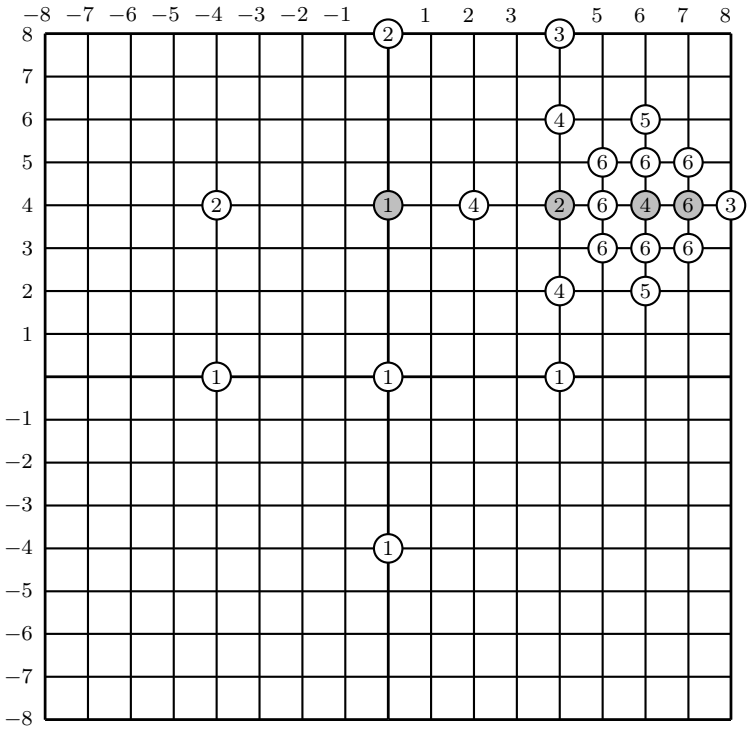


Figura 6.15: Método de búsqueda logarítmica bidimensional.

localización de la distorsión más pequeña se convierte en el centro de la búsqueda vertical, donde se buscan dos bloques candidatos por encima y por debajo del centro, a una distancia s. La mejor de estas posiciones se convierte en el centro de la siguiente búsqueda. Si el tamaño de paso s es 1, el algoritmo termina y devuelve el mejor bloque encontrado en el paso actual. De lo contrario, s se reduce a la mitad, y se lleva a cabo un nuevo conjunto similar de búsquedas horizontales y verticales.

Búsqueda de uno en uno: En este tipo de búsqueda existen de nuevo dos pasadas, una horizontal y una vertical. La pasada horizontal busca todos los bloques en el área de búsqueda cuya coordenada y es igual que la del bloque B (i.e., que se encuentra en el mismo eje horizontal que B). Suponiendo que el bloque H tiene la distorsión mínima entre aquellos, la pasada vertical busca todos los bloques en el mismo eje vertical que H y devuelve el mejor de ellos. Una variación repite ésto en áreas de búsqueda cada vez más pequeñas.

Búsqueda en cruz: Todas las pasdas de este algoritmo, excepto la última, buscan los cinco bloques en los extremos de un signo de multiplicación "×". El tamaño de paso se reduce a la mitad en cada pasada hasta que llega a 1. En la última pasada, se utiliza el signo más "+" para buscar las áreas ubicadas en las esquinas superior izquierda e inferior derecha de la pasada precedente.

Ésto ha sido un estudio de métodos de búsqueda monótonos de cuadrante. Seguimos con un esbozo de dos métodos de búsqueda avanzados.

Métodos de búsqueda jerárquica: Los métodos jerárquicos toman ventaja del hecho de que el emparejamiento de bloques es sensible al tamaño del bloque. Un método de búsqueda jerárquica comienza con grandes bloques y usa sus vectores de movimiento como puntos de partida para efectuar más búsquedas con bloques más pequeños. Los bloques grandes son menos propensos a caer en un máximo local, mientras que un bloque pequeño generalmente produce un mejor vector de movimiento. Un método de búsqueda jerárquico es por tanto computacionalmente intensivo, y la cuestión principal

es acelerarlo reduciendo el número de operaciones. Ésto se puede conseguir de varias maneras como sigue:

1. En las etapas iniciales, cuando los bloques aún son grandes, se busca sólo una muestra de bloques. Los vectores de movimiento resultantes no son los mejores, pero sólo se van a utilizar como puntos de partida para otros mejores.

2. Cuando se buscan bloques grandes, se pasan por alto algunos de los píxeles de un bloque. El algoritmo puede, por ejemplo, examinar sólo una cuarta parte de los píxeles de los bloques grandes, la mitad de los píxeles de los bloques más pequeños, y así sucesivamente.

3. Se seleccionan los tamaños de bloque de manera que el bloque utilizado en la etapa i se divide en varios bloques (típicamente cuatro o nueve) que se utilizan en la etapa siguiente. De esta manera, se calcula un único vector de movimiento en la etapa i que puede usarse como una estimación de varios vectores de movimiento mejores en la etapa $i+1$.

Métodos de espacios de búsqueda multidimensionales: Éstos métodos son más complejos. Cuando se busca un emparejamiento para el bloque B, este método busca coincidencias que son rotaciones o zums de B, no sólo traslaciones.

Un método de espacio de búsqueda multidimensional también puede encontrar un bloque C que coincida con B pero que tenga diferentes condiciones de iluminación. Ésto es útil cuando un objeto se mueve entre áreas que se iluminan de manera diferente. Todos los métodos descritos hasta ahora comparan dos bloques cotejando los valores de luminancia de los píxeles correspondientes. Dos bloques B y C, que contienen los mismos objetos pero difieren en luminosidad deberían declararse diferentes por tales métodos.

Cuando un método de espacio de búsqueda multidimensional encuentra un bloque C que coincide con B, pero tiene diferente luminancia, podrá declarar C como el emparejamiento de B y añadir un valor de luminancia al frame B comprimido. Este valor (que puede ser negativo) es añadido por el decodificador a los píxeles del frame descomprimido, para devolverlos a sus valores originales.

Un método de espacio de búsqueda multidimensional puede comparar un bloque B con versiones rotadas de los bloques candidatos C. Ésto es útil si los objetos en la presentación de vídeo pueden rotarse, además de ser movidos. El algoritmo también puede tratar de emparejar un bloque B con un bloque C que contiene una versión a escala de los objetos en B. Si, por ejemplo, B tiene un tamaño de 8×8 píxeles, el algoritmo puede considerar bloques C de tamaño 12×12, reducir cada uno a 8×8, y compararlo con B.

Este tipo de búsqueda de bloques requiere muchas operaciones y comparaciones adicionales. Decimos que aumenta el tamaño del *espacio de búsqueda* de manera significativa, de ahí el nombre espacio de búsqueda multidimensional. Parece que en la actualidad no hay ningún método de espacio de búsqueda multidimensional que pueda considerar el escalado, la rotación y los cambios en la iluminación y además ser lo suficientemente rápido para el uso práctico.

> ¿Televisión? Nada bueno saldrá de este dispositivo. La palabra es mitad griega y mitad latina.
>
> —C. P. Scott

6.5. MPEG

Desde 1988, el proyecto MPEG fue desarrollado por un grupo de cientos de expertos bajo los auspicios de la ISO (Organización Internacional de eStandarización) y la IEC (Comisión Electrotécnica Internacional). El nombre de MPEG es el acrónimo de Moving Pictures Experts Group (grupo de

expertos sobre imágenes en movimiento). MPEG es un método para la compresión de vídeo, el cual involucra la compresión de imágenes digitales y de sonido, así como la sincronización de las dos. Actualmente existen varios estándares MPEG. MPEG-1 está destinado para velocidades de datos intermedias, del orden de 1,5 $^{\text{Mbits}}/_{\text{seg.}}$; MPEG-2 se destina para altas tasas de datos de menos de 10 $^{\text{Mbits}}/_{\text{seg.}}$; MPEG-3 fue pensado para la compresión de la HDTV, pero se encontró que era redundante y se fusionó con MPEG-2; MPEG-4 ha sido diseñada para tasas de datos muy bajas de menos de 64 $^{\text{Kbits}}/_{\text{seg.}}$. Una tercera organización internacional, la ITU-T (UIT-T en español), se ha involucrado en el diseño de los formatos MPEG-2 y MPEG-4. Esta sección se concentra en MPEG-1 y discute sólo sus características de compresión de imágenes.

El nombre formal de MPEG-1 es el estándar internacional para la compresión de vídeo de imágenes en movimiento, IS11172-2. Al igual que otros estándares elaborados por la ITU y la ISO, el documento que describe MPEG-1 tiene secciones *normativas* e *informativas*. Una sección normativa es parte de la especificación del estándar. Está destinado a los implementadores; está escrito en un lenguaje preciso, y debe respetarse estrictamente cuando se aplica el estándar en plataformas informáticas reales. Una sección informativa, por otro lado, ilustra conceptos discutidos en otra parte, explica las razones que llevaron a ciertas opciones y decisiones, y contiene material de fondo. Un ejemplo de sección normativa son las diversas tablas de códigos variables utilizadas en MPEG. Un ejemplo de sección informativa es el algoritmo utilizado por MPEG para estimar el movimiento y el emparejamiento de bloques. MPEG no se requiere ningún algoritmo en particular y un codificador MPEG puede usar cualquier método para emparejamiento de bloques. La sección en sí se limita a describir las distintas alternativas.

La discusión de MPEG en esta sección es informal. La primera subsección (componentes principales) describe todos los términos importantes, principios y códigos utilizados en MPEG-1. Las subsecciones que siguen, entran en más detalles, especialmente en la descripción y el listado de los diversos parámetros y los códigos de tamaño variable.

La importancia de un estándar ampliamente aceptado para la compresión de vídeo es aparente del hecho de que muchos fabricantes (de juegos de ordenador, películas en DVD, televisión digital, y grabadores digitales, entre otros) implementaron MPEG-1 y comenzaron a usarlo incluso antes de que fuera finalmente aprobado por el comité sobre MPEG. Por esta razón, MPEG-1 tuvo que ser congelado en una etapa temprana y MPEG-2 tuvo que ser desarrollado para acomodar las aplicaciones de vídeo con la alta velocidad de datos.

Hay muchas fuentes de información sobre MPEG. [Mitchell et al. 97] es una fuente detallada sobre MPEG-1, y el consorcio sobre MPEG [MPEG 98] contiene una lista de otros recursos. Además, hay muchas páginas web con descripciones, explicaciones, y respuestas a las preguntas más frecuentes acerca de MPEG.

Para comprender el significado de las palabras "tasa de datos intermedia" consideramos un ejemplo típico de vídeo con una resolución de 360×288, una profundidad de 24 bits por píxel, y una tasa de refresco de 24 frames por segundo. La parte de la imagen de este vídeo requiere $360 \times 288 \times 24 \times 24 = 59\,719\,680$ $^{\text{bits}}/_{\text{seg.}}$. Para la parte del audio, asumimos dos pistas de sonido (sonido estéreo), cada una muestreada a 44 kHz con muestras de 16 bits. La tasa (velocidad) de datos es de $2 \times 44\,000 \times 16 = 1\,408\,000$ $^{\text{bits}}/_{\text{seg.}}$. El total es de unos 61,1 $^{\text{Mbits}}/_{\text{seg.}}$, y se supone que ésto debe ser comprimido por MPEG-1 a una tasa de datos intermedia de alrededor de 1,5 $^{\text{Mbits}}/_{\text{seg.}}$ (el tamaño de sólo la pista de sonido), ¡un factor de compresión de más de 40! Otro aspecto es la velocidad de decodificación. Una película MPEG comprimida puede ser almacenada en un CD-ROM o DVD y tiene que decodificarse y reproducirse en tiempo real.

MPEG utiliza su propio vocabulario. Una película entera se considera una *secuencia de vídeo*. Está formada por imágenes, cada una tiene tres *componentes*, uno de luminancia (Y) y dos de crominancia (Cb y Cr). El componente de luminancia (Sección 4.1) contiene la imagen en blanco y negro, y los componentes de crominancia proporcionan el tono de color y la saturación (véase [Salomon 99] para una discusión detallada). Cada componente es un array rectangular de las *muestras*, y cada fila del array se denomina una *línea raster*. Un *pel* es el conjunto de tres muestras. El ojo es sensible a pequeñas

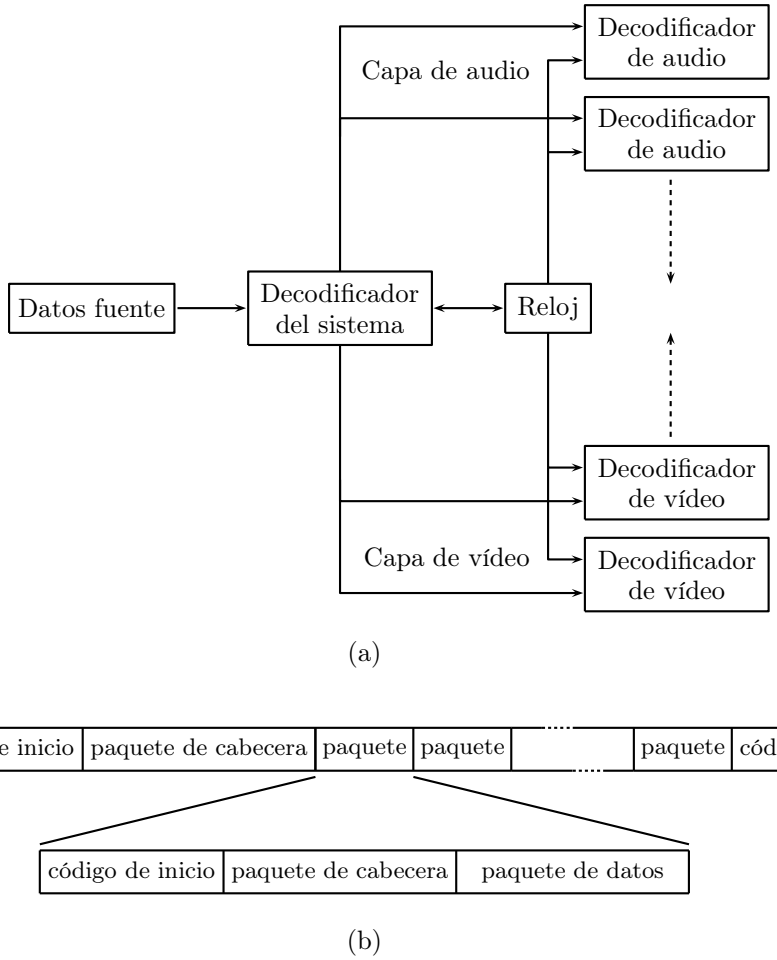


Figura 6.16: (a) Organización del decodificador MPEG. (b) Formato fuente.

variaciones espaciales de luminancia, pero es menos sensible a cambios similares en la crominancia. Como resultado, MPEG-1 muestra los componentes de crominancia a la mitad de la resolución del componente de luminancia. Se utiliza el término *intra*, pero *inter* y *nointra* se usan indistintamente.

La entrada a un codificador MPEG recibe el nombre de *datos fuente*, y la salida de un decodificador MPEG son los *datos reconstruidos*. Los datos fuente se organizan en paquetes (Figura 6.16b), donde cada paquete comienza con un código de inicio (32 bits) seguido por una cabecera, termina con un código de terminación de 32 bits, y contiene un número de paquetes en el medio. Un paquete contiene datos comprimidos, ya sea de audio, ya sea de vídeo. El tamaño de un paquete es determinado por el codificador MPEG de acuerdo con los requisitos del medio de almacenamiento o transmisión, que es por lo que un paquete no es necesariamente una imagen de vídeo completa. Puede ser cualquier parte de una imagen de vídeo o cualquier parte del audio.

El decodificador MPEG consta de tres partes principales, llamadas *capas*, para decodificar el audio, los datos de vídeo, y los datos del sistema. La capa del sistema lee e interpreta los distintos códigos y cabeceras de los datos fuente y dirige los paquetes a las capas, o bien de audio o bien de video (Figura 6.16a) para ser introducidos en un buffer y luego decodificados. Cada una de estas dos capas consta de varios decodificadores que trabajan simultáneamente.

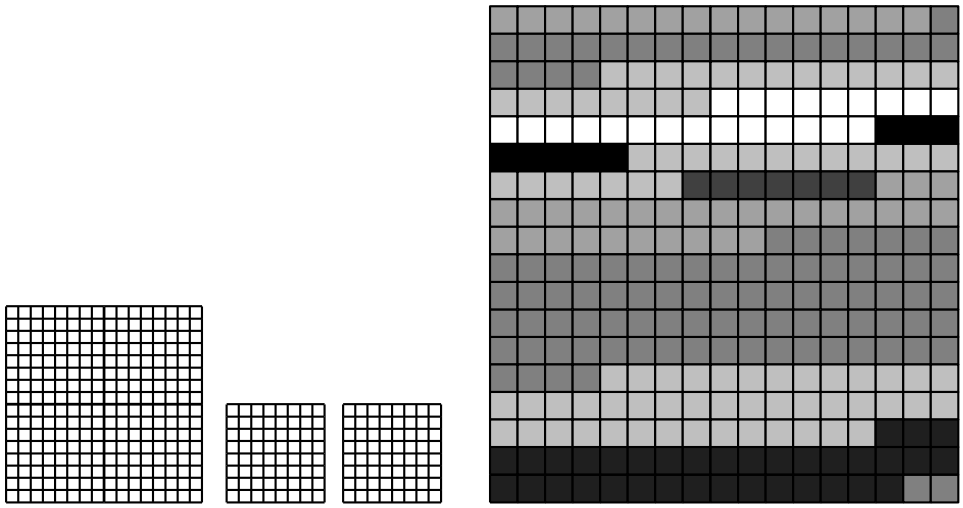


Figura 6.17: (a) Un macrobloque. (b) Una posible estructura de sectores (*slices*).

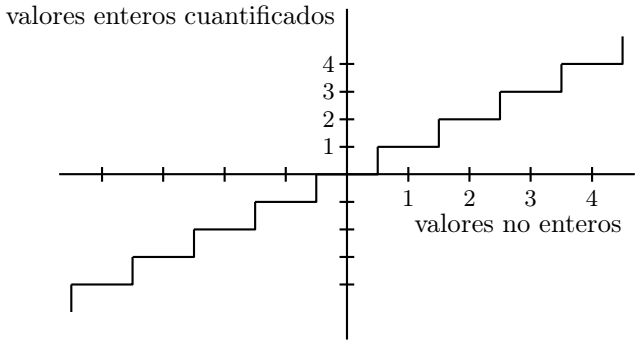
La página de inicio de MPEG está en `http://www.chiariglione.org/mpeg/index.htm`.

6.5.1. Componentes principales de MPEG-1

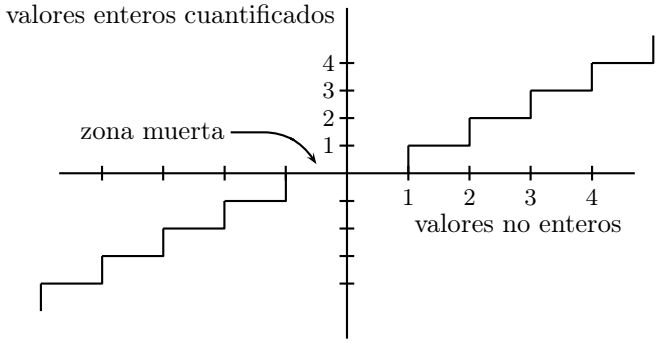
MPEG utiliza imágenes I, P, y B, como se discute en la Sección 6.4. Están dispuestas en grupos, en los que un grupo puede ser abierto o cerrado. Las imágenes están dispuestas en un cierto orden, llamado *orden de codificación*, pero (después de haber sido decodificadas) se emiten y se visualizan en un orden diferente, llamado *orden de visualización*. En un grupo cerrado, las imágenes P y B son decodificadas sólo a partir de otras imágenes en el grupo. En un grupo abierto, pueden ser decodificadas a partir de las imágenes de fuera del grupo. Las diferentes regiones de una imagen B pueden utilizar distintas imágenes para su decodificación. Una región puede ser decodificada a partir de imágenes precedentes, de algunas imágenes siguientes, de ambos tipos, o de ninguno. Similarmente, una región en una imagen P puede utilizar varias imágenes precedentes para su decodificación, o no usar ninguna en absoluto, en cuyo caso se decodifica usando métodos intra de MPEG.

El componente básico de una imagen MPEG es el *macrobloque* (Figura 6.17a). Éste consiste en un bloque de luminancia de 16×16 (en escala de grises), muestras (divididas en cuatro bloques de 8×8), y dos bloques de 8×8 de muestras de coincidencia de crominancia. La compresión MPEG de un macrobloque consiste principalmente en pasar cada uno de los seis bloques a través de una transformada discreta del coseno, lo que crea valores descorrelacionados, y a continuación cuantificar y codificar los resultados. Es muy similar a la compresión JPEG (Sección 4.8), las principales diferencias son que en MPEG se utilizan distintas tablas de cuantificación y distintas tablas de códigos para intra y nointra, y el redondeo se realiza de manera diferente.

Una imagen en formato MPEG se organiza en sectores (*slices*), donde cada sector un conjunto contiguo de macrobloques (en orden raster) que tienen la misma escala de grises (i.e., componente de luminancia). El concepto de *slices* tiene sentido porque una imagen a menudo puede contener grandes



(a) Para codificación intra



(b) Para codificación nointra

Figura 6.18: Redondeo de los coeficientes DCT cuantificados.

áreas uniformes, que producen muchos macrobloques contiguos que tienen la misma escala de grises. La Figura 6.17b muestra una imagen hipotética MPEG y la forma en que se divide en sectores. Cada cuadrado en la imagen es un macrobloque. Nótese que un sector puede continuar de una línea de escaneo a la siguiente.

◇ **Ejercicio 6.7 (sol. en pág. 1105):** ¿Cuántas muestras hay en la imagen MPEG hipotética de la Figura 6.17b?

Cuando una imagen es codificada en modo nointra (i.e., se codifica mediante otra imagen, normalmente su predecesora), el codificador MPEG genera las diferencias entre las imágenes; luego aplica la DCT a las diferencias. En tal caso, la DCT no contribuye mucho a la compresión, debido a que las diferencias ya están correlacionadas. No obstante, la DCT es útil incluso en este caso, ya que es seguida por la cuantificación, y la cuantificación en la codificación nointra puede ser muy profunda.

La precisión de los números procesados por la DCT en MPEG también depende de si se utiliza la codificación intra o nointra. Las muestras MPEG en la codificación intra son enteros sin signo de 8 bits, mientras que en la nointra son enteros con signo de 9 bits. Ésto es porque una muestra en nointra es la diferencia de dos enteros sin signo, y por lo tanto puede ser negativa. Las dos sumas de la DCT bidimensional, Ecuación (4.15), pueden a lo sumo multiplicar una muestra por $64 = 2^6$ y por lo tanto produce un entero de $8 + 6 = 14$ bits (véase el Ejercicio 4.19 para un caso similar). En esas sumas, una muestra es multiplicada por funciones coseno, que pueden generar un número negativo. El resultado de la suma doble es por lo tanto un entero con signo de 15 bits. Este entero es luego multiplicado por

el factor $C_iC_j/4$ que es al menos $1/8$, reduciendo de este modo el resultado en un entero de 12 bits.

Este entero de 12 bits es cuantificado a continuación dividiéndolo por un coeficiente de cuantificación (QC) tomado de una tabla de cuantificación. El resultado es, en general, un valor no entero y tiene que ser redondeado. Es en la cuantificación y en el redondeo donde la información se pierde irremediablemente. MPEG especifica unas tablas de cuantificación por defecto, pero también se pueden utilizar tablas personalizadas. En la codificación intra, el redondeo se realiza de la manera normal, al entero más próximo, mientras que en la nointra, el redondeo se efectúa truncando un no entero al menor entero más cercano. La Figura 6.18a,b muestra los resultados gráficamente. Nótese el gran intervalo en torno al cero en la codificación nointra. Ésta es la llamada *zona muerta*.

Las etapas de cuantificación y de redondeo son complejas e involucran más operaciones que solamente la división de un coeficiente de la DCT por un coeficiente de cuantificación. Ellas dependen de un factor de escala llamado `quantizer_scale`, un parámetro de MPEG que es un entero en el intervalo $[1, 31]$. Los resultados de la cuantificación, y por lo tanto el rendimiento de la compresión, son sensibles al valor de `quantizer_scale`. El codificador puede cambiar este valor de vez en cuando y debe insertar un código especial en el stream comprimido para indicar ésto.

Denotamos por DCT al coeficiente de la DCT que está siendo cuantificado, por Q al QC de la tabla de cuantificación, y por $QDCT$ al valor cuantificado de DCT. La regla de cuantificación para la codificación intra es:

$$QDCT = \frac{(16 \times DCT) + \text{Sign}\,(DCT) \times \texttt{quantizer\_scale} \times Q}{2 \times \texttt{quantizer\_scale} \times Q}, \tag{6.2}$$

donde la función $\text{Sign}\,(DCT)$ es el signo de DCT, definido por:

$$\text{Sign}\,(DCT) = \begin{cases} +1 & \text{cuando } DCT > 0, \\ 0, & \text{cuando } DCT = 0, \\ -1, & \text{cuando } DCT < 0. \end{cases}$$

El segundo término de la Ecuación (6.2) se llama *término de redondeo* y es el responsable de la forma especial de redondeo ilustrada en la Figura 6.18a. Ésto es fácil de ver cuando consideramos el caso de un DCT positivo. En este caso, la Ecuación (6.2) se reduce a la expresión más simple:

$$QDCT = \frac{(16 \times DCT)}{2 \times \texttt{quantizer\_scale} \times Q} + \frac{1}{2}.$$

El término de redondeo es eliminado por la codificación nointra, donde la cuantificación se efectúa con:

$$QDCT = \frac{(16 \times DCT)}{2 \times \texttt{quantizer\_scale} \times Q}. \tag{6.3}$$

La descuantificación, que se realiza mediante el decodificador en la preparación de la IDCT, es la inversa de la cuantificación. Para la codificación intra se hace con:

$$DCT = \frac{(2 \times QDCT) \times \texttt{quantizer\_scale} \times Q}{16}$$

(observe que no existe un término de redondeo), y para nointra es la inversa de la Ecuación (6.2)

$$DCT = \frac{((2 \times QDCT) + \text{Sign}\,(QDCT)) \times \texttt{quantizer\_scale} \times Q}{16}.$$

La forma precisa para calcular la IDCT no se especifica por MPEG. Ésto puede conducir a distorsiones en los casos donde una imagen es codificada por una implementación y decodificada por otra,

8	16	19	22	26	27	29	34
16	16	22	24	27	29	34	37
19	22	26	27	29	34	34	38
22	22	26	27	29	34	37	40
22	26	27	29	32	35	40	48
26	27	29	32	35	40	48	58
26	27	29	34	38	46	56	69
27	29	35	38	46	56	69	83

Tabla 6.19: Tabla de cuantificación por defecto de luminancia para codificación intra.

donde la IDCT se efectúa de manera diferente. En una cadena de imágenes inter, donde cada imagen es decodificada mediante sus vecinos, ésto puede conducir a la acumulación de errores, un fenómeno conocido como *desfase* (*mismatch*) IDCT. Ésta es la razón por la que MPEG requiere una codificación periódica intra de cada parte de la imagen. Esta *actualización forzada* tiene que hacerse al menos una vez por cada 132 imágenes P en la secuencia. En la práctica, la actualización forzada es rara, ya que las imágenes I son bastante comunes, y aparecen cada 10 a 15 imágenes.

Los números $QDCT$ son codificados mediante Huffman, usando el método no adaptativo de Huffman y tablas de códigos de Huffman que fueron calculadas con la recopilación de las estadísticas de muchas secuencias de imágenes de entrenamiento. La tabla de códigos particular que se utilice depende del tipo de la imagen que se esté codificando. Para evitar el problema de la probabilidad cero (Sección 2.18), todas los entradas en las tablas de códigos se inicializan a 1 antes de recoger cualquier estadística.

La descorrelación de los pels originales mediante el cálculo de la DCT (o, en el caso de la codificación inter, mediante el cálculo de diferencias pel) es parte del modelo estadístico de MPEG. La otra parte es la creación de un conjunto de símbolos que se aprovecha de las propiedades de la codificación de Huffman. La Sección 2.8 explica que el método de Huffman se vuelve ineficiente cuando los datos contienen muchos símbolos con probabilidades altas. Si la probabilidad de un símbolo es de 0,5, lo ideal es asignarle un código de 1 bit. Si la probabilidad es más alta, debería asignarse al símbolo un código más corto, pero los códigos de Huffman son números enteros y por lo tanto no pueden ser menores de un bit. Para evitar símbolos con alta probabilidad, MPEG utiliza un alfabeto donde varios símbolos viejos (i.e., varias diferencias pel o coeficientes DCT cuantificados) se combinan para formar un nuevo símbolo. Un ejemplo son los *run lengths* de ceros. Tras la cuantificación de los 64 coeficientes DCT de un bloque, muchos de los números resultantes son ceros. Por tanto, la probabilidad de un cero es alta y puede superar fácilmente 0,5. La solución es tratar con rachas (*runs*) de ceros consecutivos. Cada run se convierte en un nuevo símbolo y se le asigna un código de Huffman. Este método crea un gran número de nuevos símbolos y, como consecuencia de ello, se necesitan muchos códigos de Huffman. Sin embargo, se mejora la eficiencia de la compresión.

La Tabla 6.19 es la tabla de coeficientes de cuantificación por defecto para las muestras de luminancia en la codificación intra. La documentación MPEG "explica" esta tabla, diciendo: "Esta tabla tiene una distribución de valores de cuantificación que son aproximadamente acordes con la respuesta en frecuencia del ojo humano, dada una distancia de visión de unas seis veces el ancho de la pantalla y una imagen de 360×240 pels". La cuantificación en la codificación nointra es completamente diferente, ya que las cantidades a cuantificar son diferencias pel, y carecen de frecuencias espaciales. Este tipo de cuantificación se efectúa dividiendo los coeficientes DCT de las diferencias por 16 (la tabla de cuantificación por defecto es por tanto plana), aunque también se pueden especificar tablas de cuantificación personalizadas.

En una imagen I, los coeficientes DC de los macrobloques se codifican por separado de los coeficientes AC, similarmente a como se hace en el formato JPEG (Sección 4.8.4). La Figura 6.20 muestra cómo

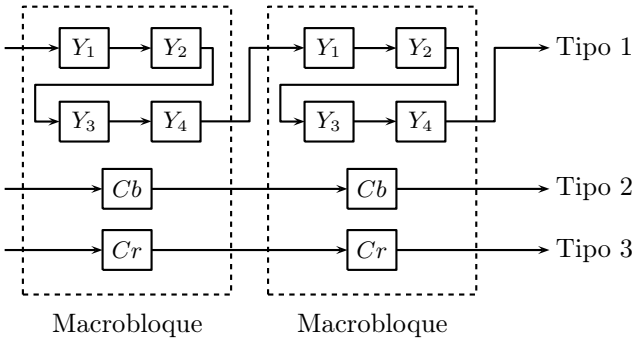


Figura 6.20: Los tres tipos de coeficientes DC.

Códigos Y	Códigos C	Tamaño	Rango de magnitud
100	00	0	0
00	01	1	$-1, 1$
01	10	2	$-3 \cdots -2, 2 \cdots 3$
101	110	3	$-7 \cdots -4, 4 \cdots 7$
110	1110	4	$-15 \cdots -8, 8 \cdots 15$
1110	11110	5	$-31 \cdots -16, 16 \cdots 31$
11110	111110	6	$-63 \cdots -32, 32 \cdots 63$
111110	1111110	7	$-127 \cdots -64, 64 \cdots 127$
1111110	11111110	8	$-255 \cdots -128, 128 \cdots 255$

Tabla 6.21: Códigos para los coeficientes DC (luminancia y crominancia).

tres tipos de coeficientes DC, para los componentes Y, Cb y Cr de la imagen I, son codificados por separado en un stream. Cada macrobloque consta de cuatro bloques Y, uno Cb, y un bloque Cr, por lo que contribuye con cuatro coeficientes DC del primer tipo, y un coeficiente de cada uno de los otros dos tipos. Un coeficiente de DC_i es utilizado para calcular primero una diferencia $\Delta DC = DC_i - P$ (donde P es el coeficiente DC previo *del mismo tipo*), y luego la diferencia es codificada codificando una categoría de tamaño seguida por algunos bits para la magnitud y el signo de la diferencia. La categoría de tamaño es el número de bits requeridos para codificar el signo y la magnitud de la diferencia ΔDC. A cada categoría se le asigna un código. Se necesitan tres pasos para codificar una diferencia ΔDC de DC: (1) En primer lugar, se determina el tamaño de la categoría y se emite su código, (2) si ΔDC es negativo, se resta un 1 de sus representaciones de complemento a 2, y (3) se emite el *tamaño* de los bits menos significativos de la diferencia. La Tabla 6.21 resume las categorías de tamaño, sus códigos y el rango de las diferencias ΔDC para cada tamaño. Observe que la categoría de tamaño de cero se define como 0. Esta tabla debería compararse con la Tabla 4.63, que lista los códigos correspondientes usados por JPEG.

Ejemplos: (1) Una luminancia ΔDC de 5. El número 5 puede expresarse en tres bits, por lo que la categoría de tamaño es 3 y el código 101 es emitido primero. Éste es seguido por los tres bits menos significativos de 5, que son 101. (2) Una crominancia ΔDC de -3. El número 3 puede expresarse en 2 bits, por lo que la categoría de tamaño es 2, y el código 10 es emitido primero. La diferencia -3 está representada en complemento a dos como ...11101. Cuando se resta 1 de este número, los 2 bits menos significativos son 00, y luego se emite este código.

◇ **Ejercicio 6.8 (sol. en pág. 1105):** Calcule el código de luminancia $\Delta DC = 0$ y el código de crominancia $\Delta DC = 4$.

$$\begin{array}{cccccccc} 127 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

(a)

$$\begin{array}{cccccccc} 118 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -2 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

(b)

Figura 6.22: Dos bloques de 8×8 de coeficientes DCT cuantificados.

Los coeficientes AC de una imagen I (codificación intra) se codifican escaneádolos en zigzag según muestra la Figura 1.8b. La secuencia de coeficientes AC resultante consiste en coeficientes distintos de cero y run lengths de coeficientes cero. Un código *run-level* es emitido para cada coeficiente distinto de cero C, donde *run* se refiere al número de coeficientes cero que preceden a C, y *level* (nivel) se refiere al tamaño absoluto de C. Cada código run-level para un coeficiente distinto de cero C es seguido por el signo de 1 bit de C (1 para negativo y 0 para positivo). El código run-level para el último coeficiente distinto de cero es seguido por un código especial fin-de-bloque de 2 bits (EOB o end-of-block). La Tabla 6.23 muestra los códigos EOB y los códigos run-level para los valores comunes de runs y niveles (levels). Las combinaciones de runs y niveles que no están en la tabla son codificados mediante el código de escape, seguido de un código de 6 bits para la longitud del run (run length) y un código de 8 ó 16 bits para el nivel.

La Figura 6.22a muestra un ejemplo de un bloque de 8×8 de coeficientes cuantificados. La secuencia en zigzag de estos coeficientes es:

$$127, 0, 0, -1, 0, 2, 0, 0, 0, 1,$$

donde 127 es el coeficiente DC. En consecuencia, los coeficientes AC son codificados mediante los tres códigos run-level $(2, -1)$, $(1, 2)$, $(3, 1)$, seguidos por el código EOB. La Tabla 6.23 muestra que los códigos son (nótese el bit de signo siguiendo a los códigos run-level):

$$0101\,1 \mid 000110\,0 \mid 00111\,0 \mid 10$$

(sin las barras verticales).

◇ **Ejercicio 6.9 (sol. en pág. 1105):** Calcúlese la secuencia en zigzag y los códigos run-level para los coeficientes AC de la Figura 6.22b.

◇ **Ejercicio 6.10 (sol. en pág. 1105):** Dado un bloque con 63 coeficientes AC a cero ¿cómo es codificado?

Una característica peculiar de la Tabla 6.23 es que muestra dos códigos para el run-level $(0, 1)$. Además, el primero de esos códigos (etiquetado "primero") es $1s$, que puede entrar en conflicto con el código EOB. La explicación es que el segundo de esos códigos (con la etiqueta "siguiente"), $11s$, es el normalmente utilizado, y ésto no causa ningún conflicto. El primer código, $1s$, sólo se usa en la codificación nointra, donde un bloque de coeficientes DCT de todo ceros es codificado de una manera especial.

La discusión hasta ahora se ha concentrado en la codificación de los coeficientes DCT cuantificados para la codificación intra (imágenes I). Para la codificación nointra (i.e., imágenes P y B), la situación es diferente. El proceso de predecir una imagen a partir de otra imagen ya descorrelaciona las muestras, y la principal ventaja de la DCT en la codificación nointra es la cuantificación. La cuantificación profunda de los coeficientes DCT aumenta la compresión, e incluso una tabla bidimensional de

0/1	1s (primero)	2	1/1	011s	4
0/1	11s (siguente)	3	1/2	0001 10s	7
0/2	0100 s	5	1/3	0000 0101 s	9
0/3	0010 1s	6	1/4	0000 0011 00s	11
0/4	0000 110s	8	1/5	0000 0001 1011 s	13
0/5	0010 0110 s	9	1/6	0000 0000 1011 0s	14
0/6	0010 0001 s	9	1/7	0000 0000 1010 1s	14
0/7	0000 0010 10s	11	1/8	0000 0000 0011 111s	16
0/8	0000 0001 1101 s	13	1/9	0000 0000 0011 110s	16
0/9	0000 0001 1000 s	13	1/10	0000 0000 0011 101s	16
0/10	0000 0001 0011 s	13	1/11	0000 0000 0011 100s	16
0/11	0000 0001 0000 s	13	1/12	0000 0000 0011 011s	16
0/12	0000 0000 1101 0s	14	1/13	0000 0000 0011 010s	16
0/13	0000 0000 1100 1s	14	1/14	0000 0000 0011 001s	16
0/14	0000 0000 1100 0s	14	1/15	0000 0000 0001 0011 s	17
0/15	0000 0000 1011 1s	14	1/16	0000 0000 0001 0010 s	17
0/16	0000 0000 0111 11s	15	1/17	0000 0000 0001 0001 s	17
0/17	0000 0000 0111 10s	15	1/18	0000 0000 0001 0000 s	17
0/18	0000 0000 0111 01s	15	2/1	0101 s	5
0/19	0000 0000 0111 00s	15	2/2	0000 100s	8
0/20	0000 0000 0110 11s	15	2/3	0000 0010 11s	11
0/21	0000 0000 0110 10s	15	2/4	0000 0001 0100 s	13
0/22	0000 0000 0110 01s	15	2/5	0000 0000 1010 0s	14
0/23	0000 0000 0110 00s	15	3/1	0011 1s	6
0/24	0000 0000 0101 11s	15	3/2	0010 0100 s	9
0/25	0000 0000 0101 10s	15	3/3	0000 0001 1100 s	13
0/26	0000 0000 0101 01s	15	3/4	0000 0000 1001 1s	14
0/27	0000 0000 0101 00s	15	4/1	0011 0s	6
0/28	0000 0000 0100 11s	15	4/2	0000 0011 11s	11
0/29	0000 0000 0100 10s	15	4/3	0000 0001 0010 s	13
0/30	0000 0000 0100 01s	15	5/1	0001 11s	7
0/31	0000 0000 0100 00s	15	5/2	0000 0010 01s	11
0/32	0000 0000 0011 000s	16	5/3	0000 0000 1001 0s	14
0/33	0000 0000 0010 111s	16	6/1	0001 01s	7
0/34	0000 0000 0010 110s	16	6/2	0000 0001 1110 s	13
0/35	0000 0000 0010 101s	16	6/3	0000 0000 0001 0100 s	17
0/36	0000 0000 0010 100s	16	7/1	0001 00s	7
0/37	0000 0000 0010 011s	16	7/2	0000 0001 0101 s	13
0/38	0000 0000 0010 010s	16	8/1	0001 111s	8
0/39	0000 0000 0010 001s	16	8/2	0000 0001 0001 s	13
0/40	0000 0000 0010 000s	16			

Tabla 6.23: Códigos run-level de longitud variable (Parte 1).

9/1	0000 101s	8	18/1	0000 0001 1010 s	13
9/2	0000 0000 1000 1s	14	19/1	0000 0001 1001 s	13
10/1	0010 0111 s	9	20/1	0000 0001 0111 s	13
10/2	0000 0000 1000 0s	14	21/1	0000 0001 0110 s	13
11/1	0010 0011 s	9	22/1	0000 0000 1111 1s	14
11/2	0000 0000 0001 1010 s	17	23/1	0000 0000 1111 0s	14
12/1	0010 0010 s	9	24/1	0000 0000 1110 1s	14
12/2	0000 0000 0001 1001 s	17	25/1	0000 0000 1110 0s	14
13/1	0010 0000 s	9	26/1	0000 0000 1101 1s	14
13/2	0000 0000 0001 1000 s	17	27/1	0000 0000 0001 1111 s	17
14/1	0000 0011 10s	11	28/1	0000 0000 0001 1110 s	17
14/2	0000 0000 0001 0111 s	17	29/1	0000 0000 0001 1101 s	17
15/1	0000 0011 01s	11	30/1	0000 0000 0001 1100 s	17
15/2	0000 0000 0001 0110 s	17	31/1	0000 0000 0001 1011 s	17
16/1	0000 0010 00s	11	EOB	10	2
16/2	0000 0000 0001 0101 s	17	ESC	0000 01	6
17/1	0000 0001 1111 s	13			

Tabla 6.23: Códigos run-level de longitud variable (Parte 2).

cuantificación por defecto (que no se aprovecha de las propiedades de la visión humana) es efectiva en este caso. Otra característica de la DCT en la codificación nointra es que los coeficientes DC y AC no son sustancialmente diferentes, ya que son transformadas DCT de diferencias. No es por eso necesario separar la codificación de los coeficientes DC y AC.

El proceso de codificación comienza localizando secuencias (runs) de macrobloques que sean completamente ceros. Tales runs son codificados mediante un incremento de la dirección del macrobloque. Si un macrobloque no es todo ceros, algunos de sus seis bloques componentes todavía pueden ser completamente ceros. Para tales macrobloques el codificador prepara un *patrón de bloque codificado* (cbp o *coded block pattern*). Éste es una variable de 6 bits donde cada bit especifica si uno de los seis bloques está formado por todo ceros (bloque cero) o no (bloque nocero). Un bloque cero se identifica como tal mediante el correspondiente bit de cbp. Un bloque nocero se codifica utilizando los códigos de la Tabla 6.23. Cuando se codifica un bloque nocero, el codificador sabe que no puede ser todo ceros. Debe existir al menos un coeficiente distinto de cero entre los 64 coeficientes cuantificados en el bloque. Si el primer coeficiente no nulo tiene un código run-level de $(0, 1)$, se codifica como "1s" y no hay ningún conflicto con el código EOB ya que el código EOB no puede ser el primer código de tal bloque. Los otros coeficientes distintos de cero con un código run-level de $(0, 1)$ se codifican utilizando el código "siguiente", que es "11s".

6.5.2. Sintaxis del vídeo MPEG-1

Algunos de los muchos parámetros que utiliza MPEG para especificar y controlar la compresión de una secuencia de vídeo son descritos en esta sección en detalle. Los lectores que estén interesados sólo en la descripción general de MPEG pueden saltarse esta sección. Los conceptos de secuencia de vídeo, imagen, slice, macrobloque y bloque ya se han discutido. La Figura 6.24 muestra el formato del stream comprimido de MPEG y cómo se organiza en seis capas. Las partes opcionales están encerradas en cajas de trazos. Tenga en cuenta que sólo se muestra la secuencia de vídeo del stream comprimido; las partes del sistema se han omitido.

La secuencia de vídeo comienza con una cabecera de secuencia, seguida por un grupo de imágenes (GOP o *Group Of Pictures*) y, opcionalmente, por más GOPs. Puede haber otras cabeceras de secuencia seguidas por más GOP, y la secuencia termina con un código-fin-de-secuencia. Las cabece-

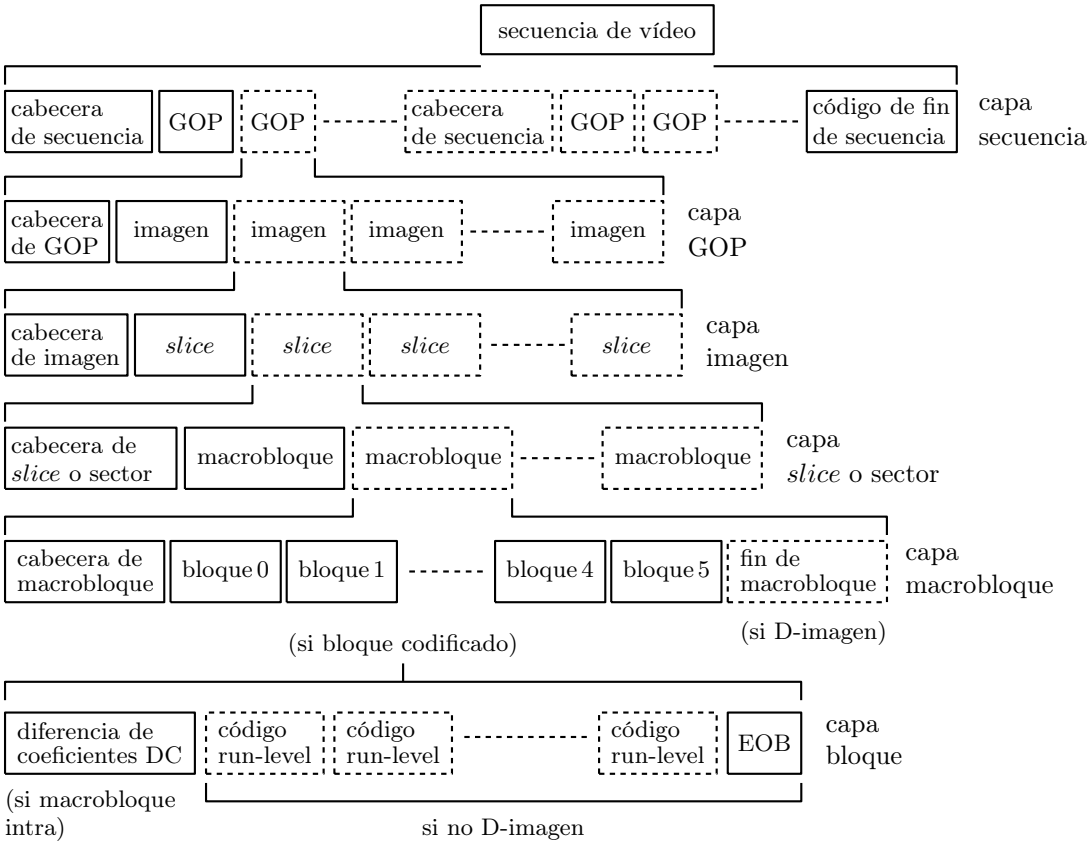


Figura 6.24: Las capas de una secuencia (*stream*) de video.

ras de secuencia adicionales pueden ser incluidas para ayudar en la reproducción aleatoria del vídeo o su edición, pero la mayoría de los parámetros en los encabezados de secuencia adicionales deben permanecer sin cambios desde la primera cabecera.

Un grupo de imágenes (GOP) comienza con una cabecera del GOP, seguida por una o más imágenes. Cada imagen en un GOP comienza con una cabecera de imagen, seguida de una o más rebanadas o sectores (*slices*). Cada slice, a su vez, se compone de una cabecera de sector seguida de uno o más macrobloques de coeficientes DCT codificados y cuantificados. Un macrobloque es un conjunto de seis bloques de 8×8, cuatro bloques de muestras de luminancia y dos bloques de muestras de crominancia. Algunos bloques pueden ser completamente cero y no pueden ser codificados. Cada bloque codificado es intra o nointra. Un bloque intra comienza con una diferencia entre su coeficiente DC y el coeficiente DC previo (del mismo tipo), seguido por códigos run-level para los coeficientes AC distintos de cero. El código EOB termina el bloque. En un bloque nointra, tanto el coeficiente DC como los coeficientes AC son codificados mediante códigos run-level.

Cabe mencionar que, además de los tipos de imágenes I, P y B, existe en MPEG un cuarto tipo, una imagen D (para DC *codificado*). Tales imágenes contienen sólo información de coeficientes DC; ningún código run-level o EOB es incluido. Sin embargo, a las imágenes D no se las permite mezclarse con los otros tipos de imágenes, por lo que son poco comunes y no se discutirán aquí.

Todos los encabezados de una secuencia, GOP, imagen, y slice empiezan con un código de inicio de 32 bits, byte-alineación. Además de estos códigos de inicio de vídeo existen otros códigos de inicio para la capa de sistema, los datos de usuario, y el error de etiquetado. Un código de inicio comienza

Código de inicio (*Start code*)	Hexadecimal	Binario
extensión.inicio (*extension.start*)	`000001B5`	00000000 00000000 00000001 10110101
GOP.inicio (*GOP.start*)	`000001B8`	00000000 00000000 00000001 10111000
imagen.inicio (*picture.start*)	`00000100`	00000000 00000000 00000001 00000000
reservado (*reserved*)	`000001B0`	00000000 00000000 00000001 10110000
reservado (*reserved*)	`000001B1`	00000000 00000000 00000001 10110001
reservado (*reserved*)	`000001B6`	00000000 00000000 00000001 10110110
secuencia.fin (*sequence.end*)	`000001B7`	00000000 00000000 00000001 10110111
secuencia.error (*sequence.error*)	`000001B4`	00000000 00000000 00000001 10110100
secuencia.cabecera (*sequence.header*)	`000001B3`	00000000 00000000 00000001 10110011
slice.inicio.1 (*slice.start.1*)	`00000101`	00000000 00000000 00000001 00000001
...	...	
slice.inicio.175 (*slice.start.175*)	`000001AF`	00000000 00000000 00000001 10101111
usuario.datos.inicio (*user.data.start*)	`000001B2`	00000000 00000000 00000001 10110010

Tabla 6.25: Códigos de inicio de video de MPEG.

con 23 bits a cero, seguidos por un solo bit a 1, seguido por un único byte. La Tabla 6.25 muestra todos los códigos de inicio de vídeo. El código "secuencia.error" es para los casos donde el codificador descubre errores irrecuperables en una secuencia de vídeo y como resultado no pueden codificarla. Los códigos run-level tienen longitud variable, por lo que normalmente deben adicionarse algunos bits 0 a la secuencia de vídeo antes de un código de inicio, para asegurarse de que el código comienza en los límites de un byte.

Capa secuencia de vídeo: Esta capa comienza con un código de inicio `000001B3`, seguido por nueve elementos de datos de longitud fija. Los parámetros `horizontal_size` y `vertical_size` son de 12 bits, y definen la anchura y la altura de la imagen, respectivamente. Tampoco está permitido que sean cero, y `vertical_size` debe ser par. El parámetro `pel_aspect_ratio` es de 4 bits, y especifica la relación de aspecto de un pel. Sus 16 valores se muestran en la Tabla 6.26. El parámetro `picture_rate` es de 4 bits, y especifica una de las 16 tasas de refresco de imágenes. Sus ocho valores no reservados se muestran en la Tabla 6.27. El elementos de datos `bit_rate` de 18 bits especifica la tasa de datos comprimidos al decodificador (en unidades de 400 $^{\text{bits}}/_{\text{s}}$). Este parámetro tiene que ser positivo y está relacionado con la tasa de bits verdadera R mediante:

$$\texttt{bit\_rate} = \lceil R/400 \rceil .$$

A continuación viene un bit bandera `marker_bit`. Este único bit a 1 evita la generación accidental de un código de inicio en caso de que algunas partes se dañaran. Estos bits marcadores son comunes en MPEG. Los 10 bits de `vbv buffer size` siguen al bit bandera. En ellos se especifica al decodificador el límite inferior para el tamaño del buffer de datos comprimido. El tamaño del buffer, en bits, viene dado por

$$B = 8 \times 2048 \times \texttt{vbv\_buffer\_size},$$

y es un múltiplo de 2K bytes. El parámetro de 1 bit `constrained_parameter_flag` es normalmente 0. Cuando se establece en 1, significa que algunos de los otros parámetros tienen valores listados en la Tabla 6.28.

Los dos últimos elementos de datos son de 1 bit cada uno y controlan la carga de las tablas de cuantificación intra y nointra. Cuando `load_intra_quantizer_matrix` se establece en 1, esto significa que es seguido por los 64 QCs de 8 bits de `intra_quantizer_matrix`. De manera similar, el parámetro `load_non_intra_quantizer_matrix` indica si le sigue la tabla de cuantificación `non_intra_quantizer_matrix` o si debe usarse la tabla por defecto.

Código	alto/ancho	Video fuente
0000	prohibido	
0001	1,0000	computadoras (VGA)
0010	0,6735	
0011	0,7031	16:9, 625 líneas
0100	0,7615	
0101	0,8055	
0110	0,8437	16:9, 525 líneas
0111	0,8935	
1000	0,9157	CCIR Rec., 601, 625 líneas
1001	0,9815	
1010	1,0255	
1011	1,0695	
1100	1,0950	CCIR Rec., 601, 525 líneas
1101	1,1575	
1110	1,2015	
1111	reservado	

Tabla 6.26: Códigos pel de relación de aspecto de MPEG.

Código	Tasa de imágenes nominal	Aplicaciones típicas
0000		Prohibido
0001	23,976	Películas en monitores de difusión (*broadcast*) NTSC
0010	24	Películas, clips comerciales, animaciones
0011	25	PAL, SECAM, componente de video genérica 625/50Hz
0100	29,97	Tasa de difusión (*broadcast*) NTSC
0101	30	Estudio profesional NTSC, componente de vídeo 525/60Hz
0110	50	Video no entrelazado PAL/SECAM/625
0111	59,94	Difusión (*broadcast*) NTSC no entrelazado
1000	60	Tasa NTSC de estudio 525 no entrelazado
1001		
...		
1111	reservado	

Tabla 6.27: Tasa de imágenes en MPEG y aplicaciones típicas.

`horizontal_size` $\leq$ 768 pels.
`vertical_size` $\leq$ 576 líneas.
(número de macrobloques) $\leq$ 396.
(número de macrobloques)$\times$`picture_rate` $\leq 396 \times 25$.
`picture_rate` $\leq$ 30 imágenes por segundo.
`vbv_buffer_size` $\leq$ 160.
`bit_rate` $\leq$ 4640.
`forward_f_code` $\leq$ 4.
`backward_f_code` $\leq$ 4.

Tabla 6.28: Restricción de límites en los parámetros.

Código	Tipo de imagen
000	prohibido
001	I
010	P
011	B
100	D
101	reservado
...	...
111	reservado

Tabla 6.29: Códigos de tipos de imágenes.

Capa GOP: Esta capa se inicia con nueve elementos obligatorios, seguido opcionalmente por extensiones y datos de usuario, y por las imágenes mismas (comprimidas).

El código de inicio de grupo de 32 bits `000001B8` es seguido por `time_code`, de 25 bits, que está formado por los siguientes seis elementos de datos: `drop_frame_flag` (un bit) es cero si la tasa de imágenes no es de 29,97 Hz; `time_code_hours` (cinco bits, en el rango $[0, 23]$), los elementos de datos `time_code_minutes` (seis bits, en el rango $[0, 59]$), y `time_code_seconds` (seis bits, en el mismo rango) indican las horas, minutos y segundos en el intervalo de tiempo que abarca desde el inicio de la secuencia hasta la visualización de la primera imagen en el GOP. El parámetro de 6 bits `time_code_pictures` indica el número de imágenes en un segundo. Existe un `marker_bit` entre `time_code_minutes` y `time_code_seconds`.

Tras el `time_code` hay dos parámetros de 1 bit. El flag `closed_gop` se establece si GOP es cerrado (i.e., sus imágenes pueden ser decodificadas sin hacer referencia a imágenes residentes fuera del grupo). El flag `broken_link` se pone a 1 si la edición ha trastocado la secuencia original de grupos de imágenes.

Capa imagen: Los parámetros en esta capa especifican el tipo de imagen (I, P, B, o D) y los vectores de movimiento para la imagen. La capa comienza con los 32 bits de `picture_start_code`, cuyo valor hexadecimal es `00000100`. Ésto es seguido por un parámetro de 10 bits `temporal_reference`, que es el número de la imagen (módulo 1024) en la secuencia. El siguiente parámetro, de 3 bits, es `picture_coding_type` (Tabla 6.29), y éste es seguido por los 16 bits de `vbv_delay` que indican al decodificador cuántos bits deben estar en el buffer de datos comprimidos antes de que la imagen pueda ser decodificada. Este parámetro ayuda evitar desbordamiento de buffer, tanto superior (*overflow*) como inferior (*underflow*).

Si el tipo de imagen es P o B, entonces ésto es seguido por la información de la escala de los vectores de movimiento hacia delante, un parámetro de 3 bits llamado `forward_f_code` (véase Tabla 6.34). Para las imágenes B, sigue la información de la escala de los vectores de movimiento hacia atrás, un parámetro de 3 bits llamado `backward_f_code`.

Capa *slice* o sector: Puede haber muchos sectores en una imagen, por lo que el código de inicio de un slice termina con un valor en el rango $[1, 175]$. Este valor define la fila del macrobloque donde comienza el sector (por lo tanto, una imagen puede tener hasta 175 filas de macrobloques). La posición horizontal donde comienza el slice en esa fila del macrobloque está determinado por otros parámetros.

El parámetro `quantizer_scale` (cinco bits) inicializa el factor de escala del cuantificador, discutido anteriormente en relación con el redondeo de los coeficientes DCT cuantificados. El flag *extra_bit_slice* que le sigue es siempre 0 (el valor 1 está reservado para los futuros estándares ISO). Después de ésto, se escriben los macrobloques codificados.

Capa macrobloque: Esta capa identifica la posición del macrobloque relativa a la posición del macrobloque actual. Codifica los vectores de movimiento para el macrobloque, e identifica los bloques

Valor del incremento	Incremento de dirección del macrobloque	Valor del incremento	Incremento de dirección del macrobloque
1	1	19	0000 00
2	011	20	0000 0100 11
3	010	21	0000 0100 10
4	0011	22	0000 0100 011
5	0010	23	0000 0100 010
6	0001 1	24	0000 0100 001
7	0001 0	25	0000 0100 000
8	0000 111	26	0000 0011 111
9	0000 011	27	0000 0011 110
10	0000 1011	28	0000 0011 101
11	0000 1010	29	0000 0011 100
12	0000 1001	30	0000 0011 011
13	0000 1000	31	0000 0011 010
14	0000 0111	32	0000 0011 001
15	0000 0110	33	0000 0011 000
16	0000 0101 11	relleno	0000 0001 111
17	0000 0101 10	escape	0000 0001 000
18	0000 0101 01		

Tabla 6.30: Códigos para incremento de dirección del macrobloque.

	Código	quant	forward	backward	patrón	intra
I	1	0	0	0	0	1
	01	1	0	0	0	1
	001	0	1	0	0	0
	01	0	0	0	1	0
	00001	1	0	0	1	0
P	1	0	1	0	1	0
	00010	1	1	0	1	0
	00011	0	0	0	0	1
	000001	1	0	0	0	1
	0010	0	1	0	0	0
	010	0	0	1	0	0
	10	0	1	1	0	0
	0011	0	1	0	1	0
	000011	1	1	0	1	0
B	011	0	0	1	1	0
	000010	1	0	1	1	0
	11	0	1	1	1	0
	00010	1	1	1	1	0
	00011	0	0	0	0	1
	000001	1	0	0	0	1
D	1	0	0	0	0	1

Tabla 6.31: Códigos de longitud variable para los tipos de macrobloque.

cbp dec.	cbp binario	# bloque 012345	código cbp	cbp dec.	cbp binario	# bloque 012345	código cbp
0	000000	`.....`	prohibido	32	100000	`c.....`	1010
1	000001	`.....c`	01011	33	100001	`c....c`	0010100
2	000010	`....c.`	01001	34	100010	`c...c.`	0010000
3	000011	`....cc`	001101	35	100011	`c...cc`	00011100
4	000100	`...c..`	1101	36	100100	`c..c..`	001110
5	000101	`...c.c`	0010111	37	100101	`c..c.c`	0001110
6	000110	`...cc.`	0010011	38	100110	`c..cc.`	00001100
7	000111	`...ccc`	00011111	39	100111	`c..ccc`	000000010
8	001000	`..c...`	1100	40	101000	`c.c...`	10000
9	001001	`..c..c`	0010110	41	101001	`c.c..c`	00011000
10	001010	`..c.c.`	0010010	42	101010	`c.c.c.`	00010100
11	001011	`..c.cc`	00011110	43	101011	`c.c.cc`	00010000
12	001100	`..cc..`	10011	44	101100	`c.cc..`	01110
13	001101	`..cc.c`	00011011	45	101101	`c.cc.c`	00001010
14	001110	`..ccc.`	00010111	46	101110	`c.ccc.`	00000110
15	001111	`..cccc`	00010011	47	101111	`c.cccc`	000000110
16	010000	`.c....`	1011	48	110000	`cc....`	10010
17	010001	`.c...c`	0010101	49	110001	`cc...c`	00011010
18	010010	`.c..c.`	0010001	50	110010	`cc..c.`	00010110
19	010011	`.c..cc`	00011101	51	110011	`cc..cc`	00010010
20	010100	`.c.c..`	10001	52	110100	`cc.c..`	01101
21	010101	`.c.c.c`	00011001	53	110101	`cc.c.c`	00001001
22	010110	`.c.cc.`	00010101	54	110110	`cc.cc.`	00000101
23	010111	`.c.ccc`	00010001	55	110111	`cc.ccc`	000000101
24	011000	`.cc...`	001111	56	111000	`ccc...`	01100
25	011001	`.cc..c`	00001111	57	111001	`ccc..c`	00001000
26	011010	`.cc.c.`	00001101	58	111010	`ccc.c.`	00000100
27	011011	`.cc.cc`	000000011	59	111011	`ccc.cc`	000000100
28	011100	`.ccc..`	01111	60	111100	`cccc..`	111
29	011101	`.ccc.c`	00001011	61	111101	`cccc.c`	01010
30	011110	`.cccc.`	00000111	62	111110	`ccccc.`	01000
31	011111	`.ccccc`	000000111	63	111111	`cccccc`	001100

Tabla 6.32: Códigos para incremento de dirección de los macrobloques.

cero y distintos de cero en el macrobloque.

Cada macrobloque tiene una dirección, o índice, en la imagen. Los valores de índice comienzan en 0 en la esquina superior izquierda de la imagen y continúan en orden raster. Cuando el codificador comienza a codificar una nueva imagen, establece la dirección del macrobloque a -1. El parámetro `macroblock_address_increment` contiene la cantidad necesaria para incrementar dirección del macrobloque con el fin de alcanzar el macrobloque está siendo codificado. Este parámetro es normalmente 1. Si `macroblock_address_increment` es mayor que 33, se codifica como una secuencia de códigos `macroblock_escape`, incrementando la dirección de cada macrobloque en 33, seguido por el código apropiado de la Tabla 6.30.

El parámetro `macroblock_type` es de tamaño variable, de entre 1 y 6 bits de longitud, cuyos valores se enumeran en la Tabla 6.31. Cada valor de este código de tamaño variable es decodificado en cinco bits que se convierten en los valores de los cinco flags siguientes:

1. `macroblock_quant`. Si se establece, se envía una nueva escala de cuantificación de 5 bits.

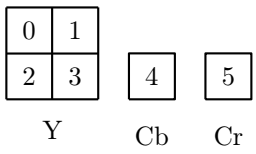


Figura 6.33: Índices de seis bloques en un macrobloque.

2. `macroblock_motion_forward`. Si se establece, se envía un vector de movimiento hacia adelante.

3. `macroblock_motion_backward`. Si se establece, se envía un vector de movimiento hacia atrás.

4. `macroblock_pattern`. Si se establece, sigue el código `coded_block_pattern` (de longitud variable) mostrado en la Tabla 6.32 para codificar los seis bits `pattern_code` de la variable `cbp` discutida anteriormente (los bloques etiquetados "." en la tabla son omitidos, y los bloques etiquetados `c` son codificados). Estos seis bits identifican los seis bloques del macrobloque como completamente cero o no. La correspondencia entre los seis bits y los bloques se muestra en la Figura 6.33, donde el bloque 0 corresponde al bit más significativo de `pattern_code`.

5. `macroblock_intra`. Si se establece, los seis bloques de este macrobloque se codifican como intra.

Estos cinco flags o banderas determinan el resto de los pasos de procesamiento para el macrobloque.

Una vez que se conocen los bits de `pattern_code`, los bloques que corresponden a bits 1 son codificados.

Capa bloque: Esta capa es la más baja de la secuencia de vídeo. Contiene los bloques de 8×8 codificados de los coeficientes DCT cuantificados. La codificación depende de si el bloque contiene muestras de luminancia o de crominancia y de si el macrobloque es intra o nointra. En la codificación nointra, los bloques que son completamente cero son omitidos; no tienen que ser codificados.

El flag `macroblock_intra` obtiene su valor de `macroblock_type`. Si se establece, el coeficiente DC del bloque es codificado por separado de los coeficientes AC.

6.5.3. Compensación de movimiento

Un elemento importante de MPEG es la *compensación de movimiento*, que se utiliza únicamente en la codificación inter. En este modo, los pels de la imagen actual se predicen mediante aquéllos de una imagen de referencia previa (y, posiblemente, los de una imagen de referencia futura). Los pels son restados, y las diferencias (que deben ser números pequeños) son transformados con la DCT, cuantificados, y codificados. Las diferencias entre la imagen actual y la de referencia son normalmente causadas por el movimiento (ya sea movimiento de la cámara, ya sea el movimiento de la escena), por lo que la mejor predicción se obtiene emparejando una región en la imagen actual con una región diferente en la imagen de referencia. MPEG no requiere el uso de un algoritmo de correspondencia concreto, y cualquier implementación puede utilizar su propio método para emparejar los macrobloques (véase la Sección 6.4 para ejemplos de algoritmos de correspondencia o coincidencia). La discusión aquí se concentra en las operaciones del decodificador.

Las diferencias entre imágenes consecutivas también pueden ser causadas por el ruido aleatorio en la cámara de vídeo, o por variaciones en la iluminación, que puede cambiar el brillo de una forma no uniforme. En tales casos, no se utiliza la compensación de movimiento, y cada región termina siendo emparejada con la misma región espacial en la imagen de referencia.

Si la diferencia entre imágenes consecutivas es causada por el movimiento de la cámara, es suficiente un vector de movimiento para toda la imagen. Normalmente, sin embargo, también existen escenas en movimiento y desplazamiento de sombras, por lo que se necesitan cierto número de vectores de

movimiento, para describir el movimiento de las distintas regiones de la imagen. El tamaño de estas regiones es crítico. Un gran número de regiones pequeñas mejora la precisión de la predicción, mientras que la situación inversa simplifica los algoritmos utilizados para encontrar regiones coincidentes y también conduce a menos vectores de movimiento y, en ocasiones, a una mejor compresión. Dado que un macrobloque es una unidad tan importante en MPEG, también fue seleccionada como la región primaria para la compensación de movimiento.

Otra consideración importante es la precisión de los vectores de movimiento. Un vector de movimiento tal como $(15, -4)$ para un macrobloque M normalmente significa que M ha sido movido desde la imagen de referencia a la imagen actual desplazando 15 pels a la derecha y cuatro pels hacia arriba (un desplazamiento vertical positivo es hacia abajo). Los componentes del vector se dan en unidades de píxeles. Sin embargo, pueden estar en unidades de medio pel, o incluso ser más pequeñas. En MPEG-1, la precisión de los vectores de movimiento puede ser, ya sea pel-completa (*full-pel*), ya sea medio-pel (*half-pel*), y el codificador señala esta decisión al decodificador mediante un parámetro en la cabecera de la imagen (este parámetro puede variar de una imagen a otra).

A menudo sucede que grandes áreas de una imagen se mueven a velocidades iguales o similares, y ésto implica que los vectores de movimiento de los macrobloques adyacentes están correlacionados. Razón por la cual el codificador MPEG codifica un vector de movimiento restándolo del vector de movimiento del macrobloque precedente y codificando la diferencia.

Una imagen P utiliza una imagen I o una imagen P previa como imagen de referencia. Decimos que las imágenes P usan una predicción con compensación de movimiento hacia delante. Cuando un vector de movimiento `MD` para un macrobloque es determinado (`MD` proviene de *motion displacement* o *desplazamiento de movimiento*, ya que el vector consta de dos componentes, los desplazamientos horizontal y vertical), MPEG denota el vector de movimiento del macrobloque previo en el sector o *slice* por `PMD` y calcula la diferencia `dMD` = `MD` − `PMD`. `PMD` se pone a cero al comienzo de un slice, después de que un macrobloque es codificado como intra, cuando el macrobloque es omitido, y cuando el parámetro `block_motion_forward` es cero.

El parámetro de 1 bit `full_pel_forward_vector` en la cabecera de la imagen define el precisión de los vectores de movimiento (1 =full-pell, 0 =half-pel). El parámetro de 3 bits `forward_f_code` define el rango.

Una imagen B puede utilizar la compensación de movimiento hacia delante (*forward*) o hacia atrás (*backward*). Si se usan ambos, entonces el codificador calcula dos vectores de movimiento para el macrobloque actual, y cada vector es codificado por primera vez calculando una diferencia, como en las imágenes P. Además, si se utilizan ambos vectores de movimiento, entonces la predicción es el promedio de los dos. Así es como la predicción es efectuada en el caso de la predicción conjunta hacia delante y hacia atrás. Supongamos que el codificador ha determinado que el macrobloque M en la imagen actual coincide con el macrobloque `MB` en la siguiente imagen y el macrobloque `MF` en la imagen precedente. Cada pel $M[i, j]$ en el macrobloque M de la imagen actual se predice mediante el cálculo de la diferencia:

$$M[i, j] - \frac{MF[i, j] + MB[i, j]}{2},$$

donde el cociente de la división por 2 es redondeado al entero más cercano.

Los parámetros `full_pel_forward_vector` y `forward_f_code` tienen el mismo significado que para una imagen P. Además, existen los dos parámetros correspondientes (para hacia atrás) `full_pel_backward_vector` y `backward_f_code`.

Las siguientes reglas son aplicadas al codificar los vectores de movimiento en imágenes B:

1. La cantidad `PMD` se establece a cero al comienzo de un slice y después de ser omitido un macrobloque.

2. En un macrobloque omitido, `MD` se predice a partir del macrobloque previo en el slice.

3. Cuando `motion_vector_forward` es cero, los `MD`s hacia adelante se predicen a partir del macrobloque precedente en el slice.

4. Cuando `motion_vector_backward` es cero, los `MD`s hacia atrás se predicen a partir del macrobloque precedente en el slice.

5. Una imagen B se predice por medio de imágenes I y P, pero no mediante otra imagen B.

6. Los dos componentes de un vector de movimiento (los desplazamientos de movimiento) se calculan con una precisión, o bien full-pel, o bien half-pel, como se especifica en el encabezamiento de la imagen.

Desplazamiento principal y residual: Los dos componentes de un vector de movimiento son desplazamientos de movimiento. Cada desplazamiento de movimiento tiene dos partes, una principal y una residual. La parte principal se denota por $dM\ D_p$. Es un entero con signo que viene dado por el producto

$$dM\ D_p = \texttt{motion\_code} \times f,$$

donde (1) `motion_code` es un parámetro entero en el intervalo $[-16, +16]$, incluido en el stream comprimido usando un código de longitud variable, y (2) f, el factor de escala, es una potencia de 2 dada por

$$f = 2^{rsize}.$$

El entero $rsize$ es simplemente `f_code` $- 1$, donde `f_code` es un parámetro de 3 bits con valores en el intervalo $[1, 7]$. Ésto implica que $rsize$ tiene valores $[0, 6]$ y f es una potencia de 2 en el rango $[1, 2, 4, 8, 16, 32, 64]$.

La parte residual se denota por `r` y es un número entero positivo, definido por

$$\texttt{r} = |dM\ D_p| - |dM\ D|\,.$$

Tras el cálculo de `r`, el codificador lo codifica concatenando los complementos a uno de `r` y `motion_r` al código de longitud variable para el parámetro `motion_code`. El parámetro `motion_r` está relacionada con `r` por

$$\texttt{motion\_r} = (f - 1) - \texttt{r},$$

y f debe ser elegido como el valor más pequeño que satisface las desigualdades siguientes para el mayor desplazamiento diferencial (positivo o negativo) en la imagen completa

$$-(16 \times f) \leq \texttt{dMD} < (16 \times f)\,.$$

Una vez ha sido seleccionado f, el valor del parámetro `motion_code` para el desplazamiento diferencial de un macrobloque viene dado por

$$\texttt{motion\_code} = \frac{\texttt{dMD} + \text{Sign}\,(\texttt{dMD}) \times (f - 1)}{f},$$

donde el cociente es redondeado de forma que `motion_code` $\times f \geq$ `dMD`.

La Tabla 6.34 muestra los parámetros de cabecera para el cálculo del vector de movimiento ("`p`" denota la cabecera de la imagen (picture) y "`mb`" denota la cabecera del macrobloque). La Tabla 6.35 muestra los nombres genérico y completo de los parámetros aquí mencionados, y la Tabla 6.36 muestra el rango de valores de `motion_r` como una función de `f_code`.

Parámetro de cabecera	Situado en cabecera	Número de bits	Parámetro de desplazamiento
full_pel_forward_vector	p	1	precisión
forward_f_code	p	3	rango
full_pel_backward_vector	p	1	precisión
backward_f_code	p	3	rango
motion_horizontal_forward_code	mb	vlc	principal
motion_horizontal_forward_r	mb	forward_r_size	residual
motion_vertical_forward_code	mb	vlc	principal
motion_vertical_forward_r	mb	forward_r_size	residual

Tabla 6.34: Parámetros de cabecera para el cálculo del vector de movimiento.

Nombre genérico	Nombre completo	Rango
full_pel_vector	full_pel_forward_vector full_pel_backward_vector	$0, 1$
f_code	forward_f_code backward_f_code	1–7
r_size	forward_r_size backward_r_size	0–6
f	forward_f backward_f	$1, 2, 4, 8, 16, 32, 64$
motion_code	motion_horizontal_forward_code motion_vertical_forward_code motion_horizontal_backward_code motion_vertical_backward_code	$-16 \rightarrow +16$
motion_r	motion_horizontal_forward_r motion_vertical_forward_r motion_horizontal_backward_r motion_vertical_backward_r	$0 \rightarrow (f-1)$
r	compliment_horizontal_forward_r compliment_vertical_forward_r compliment_horiontal_backward_r compliment_vertical_backward_r	$0 \rightarrow (f-1)$

Tabla 6.35: Nombres genéricos para parámetros de desplazamiento de movimiento.

f_code	r_size	f	$f-1$	r	motion_r
1	0	1	0		0
2	1	2	1	$0, 1$	$1, 0$
3	2	4	3	0–3	$3, \ldots, 0$
4	3	8	7	0–7	$7, \ldots, 0$
5	4	16	15	0–15	$15, \ldots, 0$
6	5	32	31	0–31	$31, \ldots, 0$
7	6	64	63	0–63	$63, \ldots, 0$

Tabla 6.36: Rango de valores de motion_r en función de f_code.

6.5.4. Reconstrucción pel

La principal tarea del decodificador MPEG es reconstruir los pels de la secuencia de vídeo completa. Ésto se hace leyendo los códigos de un bloque del stream comprimido, decodificándolos, descuantificándolos, y calculando la IDCT. Para los bloques nointra en imágenes P y B, el decodificador tiene que añadir la predicción con compensación de movimiento a los resultados de la IDCT. Ésto se repite seis veces (o menos, si algunos bloques son completamente cero) para los seis bloques de un macrobloque. Toda la secuencia es decodificada imagen por imagen, y dentro de cada imagen, macrobloque por macrobloque.

Ya se ha mencionado que la IDCT no se define de forma estricta en el formato MPEG, lo cual puede conducir a la acumulación de errores, llamados *desfases* (*mismatch*) *IDCT*, durante la decodificación.

Para los bloques con codificación intra, el decodificador lee el código diferencial del coeficiente DC y utiliza el valor decodificado del coeficiente DC anterior (del mismo tipo) para decodificar el coeficiente DC del bloque actual. A continuación, lee el código run-level hasta que encuentra un código EOB, y lo decodifica, generando una secuencia de 63 coeficientes AC, normalmente con unos pocos coeficientes distintos de cero y runs de ceros entre ellos. El coeficiente DC y los 63 coeficientes AC son luego recolectados siguiendo un orden en zigzag para crear un bloque de 8×8. Tras la descuantificación y el cálculo de la DCT inversa, el bloque resultante se convierte en uno de los seis bloques que conforman un macrobloque (en la codificación intra todos los seis bloques son codificados siempre, incluso aquellos que son completamente cero).

Para los bloques nointra, no hay distinción entre los coeficientes DC y AC y entre los bloques de luminancia y de crominancia. Todos ellos se decodifican de la misma manera.

6.6. MPEG-4

MPEG-4 es un nuevo estándar para los datos audiovisuales. A pesar de que la compresión de vídeo y audio sigue siendo aún una característica central de MPEG-4, este estándar incluye mucho más que sólo la compresión de datos. Como resultado, MPEG-4 es enorme y esta sección sólo puede describir sus principales características. No se proporcionan detalles. Empezamos con un poco de historia (véase también la Sección 6.8).

El proyecto MPEG-4 comenzó en mayo de 1991 e inicialmente dirigido a encontrar formas de comprimir los datos multimedia a muy bajas tasas de bits con distorsiones mínimas. En julio de 1994, este objetivo se modificó significativamente en respuesta a la evolución de las tecnologías audiovisuales. El comité de MPEG-4 comenzó a pensar en la futura evolución y trató de adivinar qué características deben ser incluidas en MPEG-4 para satisfacerlas. La convocatoria de propuestas fue emitida en julio de 1995 y las respuestas se recibieron en octubre de ese año. (Las propuestas eran supuestamente para hacer frente a las ocho principales funcionalidades de MPEG-4, que se enumeran a continuación.) Las pruebas de las propuestas comenzaron a finales de 1995. En enero de 1996, se definió el primer modelo de verificación, y el ciclo de convocatorias de propuestas —la implementación y verificación de propuestas se repitió varias veces en 1997 y 1998—. Muchas de las propuestas fueron aceptadas por las muchas facetas de MPEG-4, y la primera versión de MPEG-4 fue aceptada y aprobada a finales de 1998. La descripción formal se publicó en 1999 con muchas de las modificaciones que seguían saliendo.

En la actualidad (mediados de 2006), el estándar MPEG-4 se designa, estándar ISO/IEC 14496, y su descripción formal, que está disponible desde [ISO 03], consta de 17 partes más las nuevas modificaciones. Descripciones más legibles se pueden encontrar en [Pereira y Ebrahimi 02] y [Symes 03].

MPEG-1 se desarrolló originalmente como un estándar de compresión de vídeo interactivo en CDs y para la radiodifusión (*broadcasting*) de audio digital. Resultó ser un triunfo tecnológico pero un fallo visionario. Por un lado, no se encontró ni un solo error de diseño durante la implementación de este complejo algoritmo y funcionó como se esperaba. Por otro lado, los CDs interactivos y la radiodifusión

de audio digital han tenido poco éxito comercial, por lo que MPEG-1 se utiliza hoy en día para la compresión de vídeo en general. Un aspecto de MPEG-1 que se supuso de menor importancia, a saber, MP3, ha crecido de forma desproporcionada y se utiliza comúnmente hoy para el audio (Sección 7.14). MPEG-2, por otra parte, fue diseñado específicamente para la televisión digital y de este estándar ha tenido un tremendo éxito comercial.

Las lecciones aprendidas de MPEG-1 y MPEG-2 no se perdieron en los miembros del comité MPEG y ayudaron a formar sus ideas para MPEG-4. El proyecto MPEG-4 comenzó como un estándar para la compresión de vídeo a tasas de bits muy bajas. Se suponía que iba a ofrecer datos de vídeo razonables de sólo unos pocos miles de bits por segundo. Tal compresión es importante para los videoteléfonos, videoconferencias o para recibir vídeo en un pequeño dispositivo de mano, especialmente en un entorno móvil, tal como un automóvil en movimiento. Después de trabajar en este proyecto durante dos años, los miembros del comité, al darse cuenta que el rápido desarrollo de aplicaciones y servicios multimedia requería cada vez más estándares de compresión, revisaron su enfoque. En lugar de un estándar de compresión, decidieron desarrollar un conjunto de herramientas (un *toolbox*) para tratar con los productos audiovisuales en general, en el presente y en el futuro. Esperaban que ese conjunto alentara a la industria a invertir en nuevas ideas, tecnologías y productos de confianza, al hacer posible que los consumidores puedieran generar, distribuir, y recibir distintos tipos de datos multimedia con facilidad y por un costo razonable.

Tradicionalmente, los métodos para la compresión de vídeo se han basado en píxeles. Cada frame de vídeo es un conjunto rectangular de píxeles, y el algoritmo busca correlaciones entre píxeles en un frame y entre frames. El paradigma de compresión adoptado para MPEG-4, por otro lado, se basa en objetos. (El nombre del proyecto MPEG-4 fue también cambiado en este punto a "codificación de objetos audiovisuales.") Además de producir una película de la manera tradicional con una cámara o con la ayuda de la animación por ordenador, una generación individual de un fragmento de datos audiovisuales puede comenzar definiendo los objetos, tales como una flor, un rostro o un vehículo, y luego describiendo cómo debe moverse cada objeto y ser manipulado en frames sucesivos. Una flor se abre lentamente, una cara puede cambiar, sonreír, y desvanecerse, un vehículo puede moverse hacia el espectador y parecer más grande. MPEG-4 incluye un lenguaje de descripción de objetos que proporciona una descripción compacta de tanto los objetos como de sus movimientos e interacciones.

Otra característica importante de MPEG-4 es la *interoperabilidad*. Este término se refiere a la capacidad de intercambiar cualquier tipo de datos, ya sea texto, gráficos, vídeo, o audio. Obviamente, la interoperabilidad sólo es posible en presencia de estándares. Todos los dispositivos que producen datos, los entregan, y los consumen (reproducción, muestra, o impresión) deben obedecer las mismas reglas, y leer y escribir las mismas estructuras de archivos.

Durante su importante reunión en julio de 1994, el comité MPEG-4 decidió revisar su objetivo original y también comenzó a pensar en la evolución futura en el ámbito audiovisual y de las características que deben incluirse en MPEG-4 para satisfacerlas. Ellos terminaron con ocho puntos que consideraban funcionalidades importantes para MPEG-4.

1. **Herramientas de acceso multimedia basadas en contenidos.** El estándar MPEG-4 debe proporcionar herramientas para el acceso y organización de los datos audiovisuales. Estas herramientas pueden incluir: indexación, interconexión (*linking*), consulta, navegación, entrega de archivos y su eliminación. Las principales herramientas existentes en la actualidad se encuentran más adelante en esta sección.

2. **Manipulación basada en contenidos y la edición de cadenas bits (*bitstream*).** Una sintaxis y un esquema de codificación deben ser parte de MPEG-4. La idea es permitir a los usuarios manipular y editar archivos comprimidos (bitstreams) sin descomprimirlos completamente. Un usuario debe poder seleccionar un objeto y modificarlo en el archivo comprimido sin descomprimir el archivo completo.

3. **Codificación de datos híbridos, naturales y sintéticos.** Una escena natural es normalmente producida por una cámara de vídeo. Una escena sintética consta de texto y gráficos. MPEG-4 reconoce la necesidad de herramientas para comprimir escenarios naturales y sintéticos, y mezclarlos interactivamente.

4. **Mejora del acceso aleatorio temporal.** A menudo los usuarios desean acceder a una parte del archivo comprimido, por lo que el estándar MPEG-4 debe incluir etiquetas para que sea fácil llegar rápidamente a cualquier punto del archivo. Ésto puede ser importante cuando el archivo se almacena en una ubicación central, y el usuario está tratando de manipularlo de forma remota, a través de un canal de comunicaciones lento.

5. **Mejora de la eficiencia de codificación.** Esta característica significa simplemente una mejor compresión. Imagine un caso donde los datos audiovisuales tienen que ser transmitidos sobre un canal de bajo ancho de banda (tal como una línea telefónica) y almacenados en un dispositivo de baja capacidad, tal como una tarjeta inteligente. Ésto es posible sólo si los datos se comprimen bien, y las altas tasas de compresión (o equivalentemente, bajas tasas de bits) normalmente implican una compensación en forma de imágenes de tamaño más pequeño, resolución reducida (píxeles por pulgada), y una menor calidad.

6. **Codificación de múltiples streams de datos simultáneos.** Parece que las aplicaciones audiovisuales futuras permitirán al usuario no sólo ver y escuchar, sino también interactuar con la imagen. Como resultado, el stream comprimido de MPEG-4 puede incluir varios puntos de vista de la misma escena, permitiendo al usuario seleccionar cualquiera de ellos para ver y cambiar puntos de vista a voluntad. La cuestión es que los diferentes puntos de vista pueden ser similares, por lo que cualquier redundancia debe ser eliminada por medio de una compresión eficiente que tenga en cuenta los patrones idénticos en los distintos puntos de vista. Lo mismo es cierto para la parte de audio (las bandas sonoras o *soundtracks*).

7. **Robustez en entornos propensos a errores.** MPEG-4 debe proporcionar códigos de corrección de errores para los casos en que los datos audiovisuales se transmiten a través de un canal ruidoso. Ésto es especialmente importante para los streams de baja tasa de bits, donde hasta el más mínimo error puede ser notable y puede propagarse y afectar a gran parte de la presentación audiovisual.

8. **Escalabilidad basada en el contenido.** El stream comprimido puede incluir datos audiovisuales de resolución fina y de alta calidad, pero cualquier decodificador MPEG-4 debe ser capaz de decodificarlo a baja resolución y baja calidad. Esta característica es útil en los casos en que los datos son decodificados y mostrados en una pantalla pequeña, de baja resolución, o en los casos en que el usuario está en un apuro y prefiere ver una imagen aproximada en lugar de esperar a una decodificación completa.

Una vez que las ocho funcionalidades fundamentales antes mencionadas fueron identificadas y enumeradas, el comité de MPEG-4 comenzó el proceso de desarrollo de herramientas separadas para satisfacer estas funcionalidades. Éste es un proceso en curso que continúa hasta nuestros días y continuará en el futuro. Un autor de MPEG-4 frente a una aplicación tiene que identificar los requisitos de la aplicación y seleccionar las herramientas adecuadas. Ahora está claro que la compresión es un requisito central en MPEG-4, pero no el único requisito, como lo fue para MPEG-1 y MPEG-2.

Un ejemplo puede servir para ilustrar el concepto de objetos naturales y sintéticos. En una sesión de noticias en la televisión, unos pocos segundos pueden estar dedicados al tiempo. Los espectadores ven el mapa del tiempo de su región geográfica local (una imagen generada por ordenador) que puede acercar y alejar, y de la que puede tomar una vista panorámica. Las imágenes gráficas del sol, las nubes, las gotas de lluvia, o un arco iris (escenas sintéticas) aparecen, se mueven y desaparecen. Una

persona se está moviendo, apuntando, y hablando (un escenario natural), y también puede aparecer texto (otra escena sintética) de vez en cuando. Todas las escenas se mezclan por los productores en una presentación audiovisual que es comprimida, transmitida (por el cable de televisión, por el aire, o por internet), recibida por los ordenadores o televisores, descomprimida y visualizada (consumida).

En general, el contenido audiovisual pasa por tres etapas: producción, distribución y consumo. Cada una de estas etapas se resume a continuación para el enfoque tradicional y para el enfoque de MPEG-4.

Producción. Tradicionalmente, los datos audiovisuales están formados por escenas bidimensionales; se generan con una cámara y micrófonos, y se componen de objetos naturales. Toda la mezcla de objetos (composición de la imagen) se efectúa durante la producción. El enfoque MPEG-4 consiste en permitir tanto objetos bidimensionales como tridimensionales, y escenarios naturales y sintéticos. La composición de los objetos es especificada explícitamente por los productores durante la producción por medio de un lenguaje especial. Ésto permite su posterior edición.

Distribución. El enfoque tradicional es transmitir los datos audiovisuales en unas pocas redes, tales como redes de área local y transmisiones vía satélite. El enfoque de MPEG-4 es permitir que prácticamente cualquier red de datos lleve datos audiovisuales. Existen protocolos para transmitir datos audiovisuales a través de cualquier tipo de red.

Consumo. Tradicionalmente, un espectador no puede ver el vídeo y escuchar el audio de fondo. Todo está precompuesto. El enfoque de MPEG-4 es permitir al usuario toda la libertad de composición como sea posible. El usuario debe ser capaz de interactuar con los datos audiovisuales, ver sólo partes de ellos, modificar de forma interactiva el tamaño, la calidad, y la resolución de las partes que está observando, y ser tan activo en la fase de consumo como posible.

Debido a los amplios objetivos y la rica variedad de herramientas disponibles como parte de MPEG-4, se espera que este estándar tenga muchas aplicaciones. Las que se muestran aquí son sólo algunos ejemplos importantes.

1. Transmisión de datos multimedia a través de Internet o a través de redes de área local. Ésto es importante para el entretenimiento y la educación.

2. Comunicaciones, tanto visuales como de audio, entre los vehículos y/o personas. Ésto tiene aplicaciones militares y policiales.

3. Multimedia digital de radiodifusión (*broadcasting*). Ésto, de nuevo, tiene muchas aplicaciones de entretenimiento y educativas.

4. Almacenamiento y recuperación basado en el contexto. Los datos audiovisuales pueden ser almacenados en formato comprimido y recuperados para la entrega o el consumo.

5. Estudio y postproducción de televisión. Una película producida originalmente en inglés puede ser traducida a otro idioma mediante el doblaje o la subtitulación.

6. Vigilancia. El vídeo de baja calidad y los datos de audio pueden comprimirse y transmitirse desde una cámara de vigilancia a una central de vigilancia ubicada sobre un barato y lento canal de comunicaciones. Las señales de control pueden ser enviadas de nuevo a la cámara a través del mismo canal para girar o ampliar las imágenes, con el fin de seguir los movimientos de un sospechoso.

7. Conferencia virtual. Esta aplicación de ahorro de tiempo es la favorita de los ejecutivos ocupados.

Nuestra breve descripción de MPEG-4 concluye con una lista de las principales herramientas especificadas por el estándar MPEG-4.

Sistema descriptor de objetos. Imaginemos a un individuo participando en una videoconferencia. Existe un objeto MPEG-4 que representa a este individuo, y hay streams de vídeo y de audio asociados a este objeto. El descriptor de objetos (OD) proporciona información sobre los streams elementales disponibles para representar un objeto MPEG-4 dado. El OD también tiene información sobre la ubicación de origen de los streams (tal vez una URL) y sobre diversos decodificadores MPEG-4 disponibles para consumir (i.e., mostrar y reproducir sonido) los streams. Ciertos objetos ponen restricciones a su consumo, y éstos también son incluidos en el OD del objeto. Un ejemplo común de una restricción es la necesidad de pagar antes de que un objeto pueda ser consumido. Un video, por ejemplo, puede ser visto solamente si se ha pagado, y el consumo puede ser restringido a sólo la transmisión (*streaming*), por lo que el consumidor no puede copiar la película original.

Modelo decodificador de sistemas. Toda la sincronización básica y las características de streaming del estándar MPEG-4 están incluidas en esta herramienta. En ella se especifica cómo los búferes del receptor deben ser inicializados y administrados durante la transmisión y el consumo. También incluye especificaciones para la identificación de las temporizaciones y los mecanismos para la recuperación de errores.

Formato binario para las escenas. Una escena MPEG-4 se compone de objetos, pero para que la escena tenga sentido, los objetos deben ser colocados en los lugares adecuados y moverse y manipularse en el momento adecuado. Esta importante herramienta (*Binary Format For Scenes* o BIFS para abreviar) es la responsable de describir una escena, tanto espacial como temporalmente. Contiene funciones que se utilizan para describir objetos bidimensionales y tridimensionales y sus movimientos. También proporciona maneras de describir y manipular escenas sintéticas, como texto y gráficos.

MPEG-J. Un usuario puede querer usar el lenguaje de programación Java para implementar ciertas partes de un contenido en MPEG-4. MPEG-J permite al usuario escribir tales *MPEGlets*[1] y también incluye útiles APIs de Java que ayudan a la interfaz de usuario con el dispositivo de salida y con las redes utilizadas para entregar el contenido. Además, MPEG-J también define un mecanismo de entrega que permite a los MPEGlets y otras clases de Java ser transmitidas por streaming como salida por separado.

Formato de texto MPEG-4 extensible. Esta herramienta es un formato XMT abreviado, que permite a los autores intercambiar el contenido MPEG-4 con otros autores. XMT puede describirse como un sistema que utiliza una sintaxis textual para representar las descripciones de las escenas MPEG-4.

Herramientas de transporte. Dos de estas herramientas, MP4 y FlexMux, han sido definidas para ayudar a los usuarios a transportar contenidos multimedia. La primera escribe el contenido MPEG-4 en un archivo, mientras que la última se utiliza para intercalar múltiples streams en uno único, incluyendo la información de sincronismo.

Compresión de vídeo. Ya se ha mencionado que la compresión es sólo uno de los muchos objetivos de MPEG-4. Las herramientas de compresión de vídeo se componen de varios algoritmos que pueden comprimir datos de vídeo a tasas de bits de entre 5 $^{\text{Kbits}}/_{\text{seg.}}$ (muy baja tasa de bits, lo que implica baja resolución y vídeo de baja calidad) y 1 $^{\text{Gbit}}/_{\text{seg.}}$. Los métodos de compresión varían desde muchas pérdidas a casi sin pérdidas, y algunos también soportar video progresivo y entrelazado. Muchos objetos MPEG-4 consisten en mallas poligonales, por lo que la mayor parte de las herramientas de compresión de vídeo están diseñadas para comprimir dichas mallas. La Sección 8.11 es un ejemplo de tal método.

Herramientas robustez. La compresión de datos se basa en la eliminación de redundancias de los datos originales, pero esto también hace que los datos sean más vulnerables a los errores. Todos los métodos para la detección y corrección de errores se basan en el aumento de la redundancia de

[1] Let se utiliza aquí con el significado de "ir con"; mantengo la palabra original por quedar más "artística".

Estándar	Propósito
H.261	Vídeo
H.221	Comunicaciones
H.230	Establecimiento de enlace (*handshake*) inicial
H.320	Sistemas terminales
H.242	Protocolo de control
G.711	Audio compandido (64 Kbits/s)
G.722	Audio de alta calidad (64 Kbits/s)
G.728	Habla (LD-CELP @16kbits/s)

Tabla 6.37: Los estándares $p \times 64$.

los datos. MPEG-4 incluye herramientas para agregar robustez, en forma de códigos correctores de errores, al contenido comprimido. Estas herramientas son importantes en aplicaciones en las que los datos tienen que transmitirse a través de líneas poco fiables. La robustez también tiene que ser añadida a streams MPEG-4 de muy baja tasa de bits debido a que éstos son quienes más sufren errores.

Escalabilidad de grano fino. Cuando el contenido de MPEG-4 es transmitido por streaming, a veces es deseable enviar primero una imagen aproximada, y luego mejorar su calidad visual mediante la adición de capas de información adicionales. Ésta es la función de las herramientas de escalabilidad de grano fino (*fine-grain scalability* o FGS).

Animación de cara y cuerpo. A menudo, un archivo MPEG-4 contiene rostros humanos y cuerpos, y ellos tienen que ser animados. Por tanto, el estándar MPEG-4 ofrece herramientas para la construcción y animación de tales superficies.

Codificación de voz. El habla a menudo puede ser parte del contenido de MPEG-4, y se proporcionan herramientas especiales para comprimirlo de manera eficiente en tasas de bits desde 2 $^{\text{Kbits}}/_{\text{seg.}}$ hasta 24 $^{\text{Kbits}}/_{\text{seg.}}$. El algoritmo principal para la compresión de voz es CELP, pero también hay un codificador paramétrico.

Codificación de audio. Diversos algoritmos están disponibles como herramientas de MPEG-4 para la compresión de audio. Algunos ejemplos son: (1) codificación de audio avanzado (*Advanced Audio Coding* o AAC, Sección 7.15), (2) cuantificación vectorial entrelazada ponderada en el dominio de la transformada (*transform-domain weighted interleave vector quantization* o Twin VQ, puede producir tasas de bits bajas tales como 6 kbit/seg/canal), y (3) líneas armónicas e individuales más ruido (*harmonic and individual lines plus noise* o HILN, un codificador de audio paramétrico).

Codificación de audio sintético. Algoritmos que proporcionan la generación del sonido de instrumentos musicales familiares. Pueden ser utilizados para producir música sintética en formato comprimido. El formato MIDI, popular entre los usuarios de la música por ordenador, también se incluye entre estas herramientas. Las utilidades *text-to-speech*[2] permiten a los autores escribir el texto que será pronunciado cuando el contenido de MPEG-4 es consumido. Este texto puede incluir parámetros tales como la definición de la afinación y la duración del fonema que mejoran la calidad de la voz.

6.7. H.261

A finales de 1984, el CCITT (actualmente la ITU-T, o en español UIT-T), organizó un grupo de expertos para desarrollar un estándar para la telefonía visual para los servicios ISDN (RDSI). La idea era enviar imágenes y sonido entre terminales especiales, de manera que los usuarios pudieran

[2]Literalmente "texto-a-habla"; estas utilidades son capaces de leer en voz alta, esto es, pronuncian lo escrito.

hablar y verse. Este tipo de aplicación requiere el envío de grandes cantidades de datos, por lo que la compresión se convirtió en una consideración importante. El grupo finalmente llegó con algunos estándares, conocidos como las recomendaciones para las series H (para vídeo) y para las series G (para audio), todas operando a velocidades de $p \times 64$ $^{\text{Kbits}}/_{\text{seg.}}$ para $1 \leq p \leq 30$. Estos estándares se conocen hoy bajo el nombre genérico de $p \times 64$ y se resumen en la Tabla 6.37. En esta sección ofrece un breve resumen de el estándar H.261 [Liou 91].

Los miembros de $p \times 64$ también participaron en el desarrollo de MPEG, por lo que los dos métodos tienen muchos elementos comunes. Existe, sin embargo, una diferencia importante entre ellos. En MPEG, el decodificador debe ser rápido, ya que puede tener que operar en tiempo real, pero el codificador puede ser lento. Ésto conduce a una compresión muy asimétrica, y el codificador puede ser cientos de veces más complejo que el decodificador. En H.261, tanto el codificador y el decodificador operan en tiempo real, por lo que ambos deben ser rápidos. Aún así, el H.261 estándar define sólo el stream de datos y el decodificador. El codificador puede utilizar cualquier método siempre y cuando cree un stream comprimido válido. El stream comprimido se organiza en capas, y los macrobloques se utilizan como en MPEG. También se utilizan, la misma DCT de 8×8 y el mismo orden en zigzag que en MPEG. El coeficiente DC intra se cuantifica siempre dividiéndolo por 8, y no tiene ninguna zona muerta. El coeficiente DC inter y todos los coeficientes AC son cuantificados con una zona muerta.

La compensación de movimiento se utiliza cuando las imágenes son predichas a partir de otras imágenes, y los vectores de movimiento son codificados como diferencias. Los bloques que son completamente cero pueden ser omitidos dentro de un macrobloque, y se usan los códigos de tamaño variable que son muy similares a los de MPEG (tales como los códigos run-level), o son incluso idénticos (tales como los códigos de los vectores de movimiento). En todos estos aspectos, H.261 y MPEG son muy similares.

Existen, sin embargo, importantes diferencias entre ellos. H.261 utiliza un único coeficiente de cuantificación en lugar de una tabla de 8×8 de QCs, y este coeficiente puede cambiarse sólo después de 11 macrobloques. Los coeficientes AC que están codificados con intra tienen una zona muerta. El stream comprimido tiene sólo cuatro capas, en lugar de las seis de MPEG. Los vectores de movimiento son siempre full-pel y están limitados a un rango de sólo ± 15 pels. No hay imágenes B, y sólo puede utilizarse la imagen inmediatamente anterior para predecir una imagen P.

6.7.1. Flujo de datos (*stream*) comprimido H.261

H.261 limita la imagen a sólo dos tamaños, el formato intermedio común (CIF), que es opcional, y el cuarto de CIF (*quarter* CIF o QCIF). Éstos se muestran en la Figura 6.38a,b. El formato CIF tiene unas dimensiones de 288×360 para la luminancia, pero sólo 352 de las 360 columnas de pels se codifican en realidad, creando un área activa de 288×352 pels. Para la crominancia, las dimensiones son de 144×180, pero sólo son codificadas 176 columnas, generando un área activa de 144×176 pels. El formato QCIF es un cuarto de CIF, por lo que el componente de luminancia es de 144×180 con un área activa de 144×176, y las componentes de crominancia son 72×90, con un área activa de 72×88 pels.

Los macrobloques se organizan en grupos (conocidos como grupos de bloques, *Groups Of Blocks* o GOB) de 33 macrobloques de cada uno. Cada GOB es una matriz de 3×11 (48×176 pels), como se muestra en la Figura 6.38c. Una imagen CIF consta de 12 GOBs, y una imagen QCIF son tres GOBs, numerados como en la Figura 6.38d.

La Figura 6.39 muestra las cuatro capas de la secuencia de vídeo H.261. La capa principal es la capa de imagen. Comienza con una cabecera de imagen, seguida por los GOBs (12 ó 3, dependiendo del tamaño de la imagen). El stream comprimido puede contener tantas imágenes como sea necesario. La siguiente capa es el GOB, que consta de una cabecera, seguida de los macrobloques. Si la compensación de movimiento es buena, algunos macrobloques pueden ser completamente nulos y se pueden omitir. La capa de macrobloque es la siguiente, con una cabecera que puede contener un vector de movimiento,

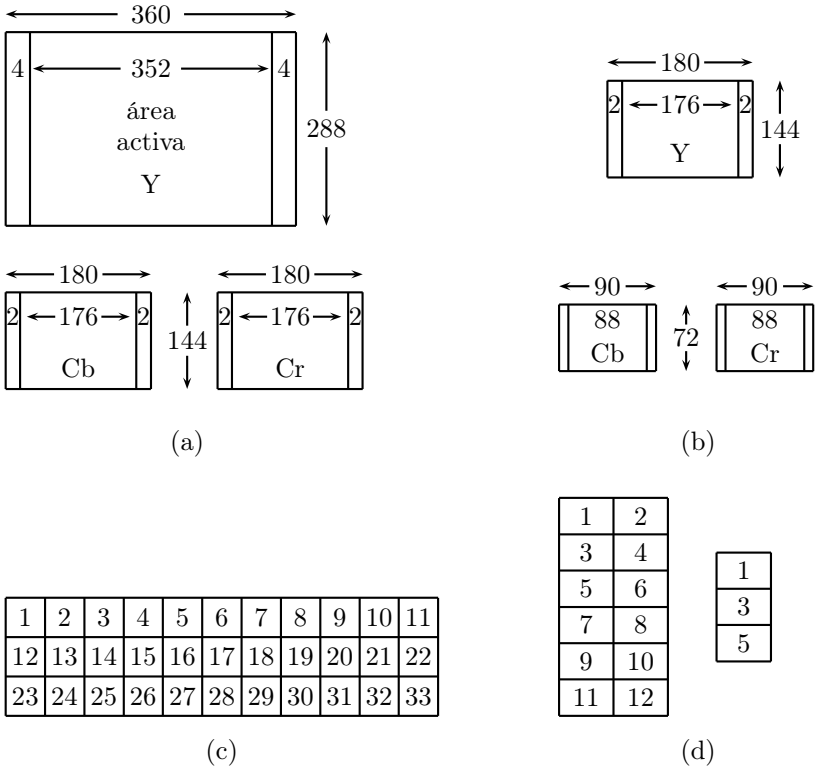


Figura 6.38: (a) CIF. (b) QCIF. (c) GOB.

seguido por los bloques. Hay seis bloques por macrobloque, pero los bloques cero son omitidos. La capa de bloque contiene los bloques que son codificados. Si un bloque es codificado, sus coeficientes no nulos se codifican con un código run-level. Un código EOB de 2 bits termina cada bloque.

6.8. H.264

Los últimos años del siglo 20 fueron testigos de un progreso sin precedentes e imprevisible en la potencia de cálculo, aplicaciones de video y soporte de redes para los datos de vídeo. Ambos, productores y consumidores de vídeo, sintieron la necesidad de un codificador (*codec*) de vídeo avanzado para reemplazar los estándares de compresión de vídeo existentes: H.261, H.262 y H.263. El último de estos estándares, H.263, fue desarrollado hacia 1995, y en el 2001 ya era anticuado. Los dos grupos responsables del desarrollo de los estándares de compresión de vídeo, ISO-MPEG y ITU-VCEG (grupo de expertos en imágenes en movimiento y grupo expertos en codificación de vídeo), consideraron que el nuevo estándar debería ofrecer: (1) aumento de la eficiencia de la compresión, (2) soporte para aplicaciones de vídeo especiales como la videoconferencia, almacenamiento en DVD, transmisión de video, y *streaming* a través de internet, y (3) una mayor fiabilidad. En el 2001, en respuesta a esta demanda, la ITU comenzó dos proyectos. El primer proyecto, a corto plazo, fue un intento para añadir características adicionales a H.263. Este proyecto dio lugar a la versión 2 de este estandar.

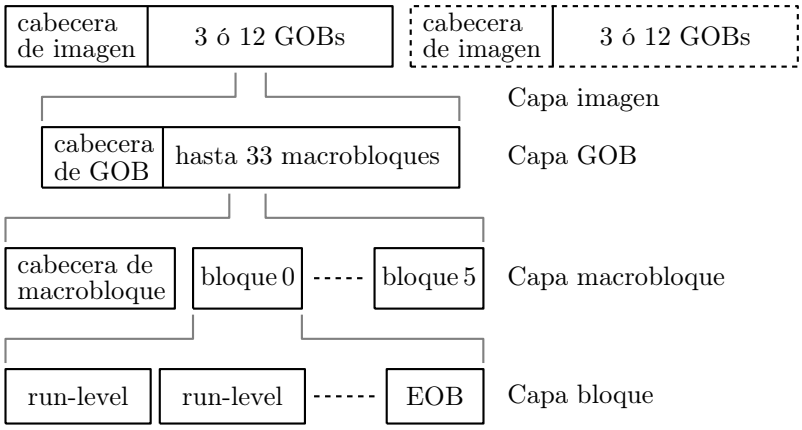


Figura 6.39: H.261 Vídeo Secuencia Layers.

El segundo proyecto, a largo plazo, fue un esfuerzo para desarrollar un nuevo estándar para una compresión de video eficiente (i.e., la baja tasa de bits). Este proyecto recibió el nombre en clave H.26L. En el mismo año, la ITU estableció un nuevo grupo de estudio, SG 16, cuyos miembros provenían de MPEG y VCEG, y cuya misión era estudiar nuevos métodos, seleccionar los prometedores, e implementar, probar, y aprobar el nuevo estándar. El nuevo estándar, H.264, fue aprobado en mayo de 2003 [ITU-T264 02], y una corrección fue agregada y aprobada en mayo de 2004. Este estándar, que ahora forma parte del gran proyecto MPEG-4, es conocido por varios nombres. La ITU la llama Recomendación ITU-T H.264, codificación avanzada de vídeo para servicios audiovisuales genéricos. Su título oficial es codificación avanzada de vídeo (*Advanced Video Coding* o AVC). La ISO lo considera la parte 10 del estándar MPEG-4. A pesar de su título, este estándar (mencionado en esta sección como H.264) es el mejor codificador de vídeo disponible en el momento de este escrito (mediados de 2006).

Lo siguiente es una lista de algunas de las muchas referencias disponibles en la actualidad para H.264. El estándar oficial es [H.264Draft 06]. Como es habitual en tales estándares, este documento describe sólo el formato del stream comprimido y el funcionamiento del decodificador. Por lo tanto, es difícil de seguir. El texto principal es [Richardson 03], un libro detallado sobre H.264 y MPEG-4. La transformada entera de H.264 ha sido estudiada en [H.264Transform 06]. Muchos documentos y artículos están disponibles en línea en [H.264Standards 06]. Dos documentos de estudio de Iain Richardson [H264PaperIR 06 y 10] y Rico Malvar [H.264PaperRM 06] están también ampliamente disponibles y sirven como introducción a este tema. La discusión en esta sección sigue tanto [H.264Draft 06] como [Richardson 03].

Principios de funcionamiento

El estándar H.264 no especifica las operaciones del codificador, pero tiene sentido suponer que las implementaciones prácticas del codificador y del decodificador estarán formados por los bloques de construcción mostrados en las Figuras6.40 y 6.41, respectivamente. Aquellos que estén familiarizados con los viejos codificadores de vídeo se darán cuenta de que los principales componentes de H.264 (predicción, transformación, cuantificación, y codificación de la entropía) no son muy diferentes de las de sus predecesores MPEG-1, MPEG-2, MPEG-4, H.261 y H.263. Lo que hace a H.264 nuevo y original son los detalles de los componentes y la presencia de un único componente nuevo, el filtro. Debido a esta similitud, la discusión aquí se concentra en las características de H.264 que lo distinguen de sus predecesores. Los lectores que estén interesados en más detalles deberían estudiar los principios

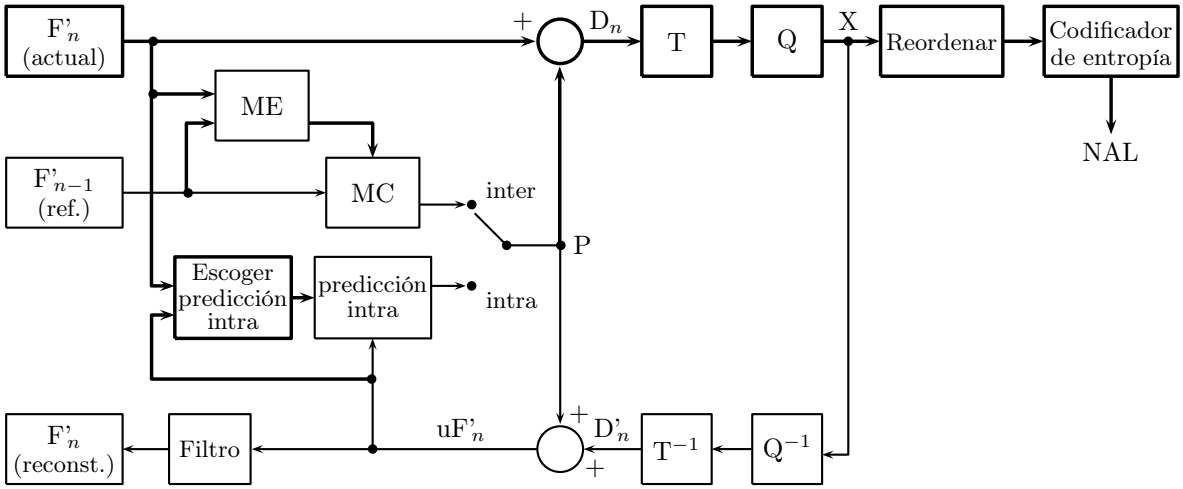


Figura 6.40: Codificador H.264.

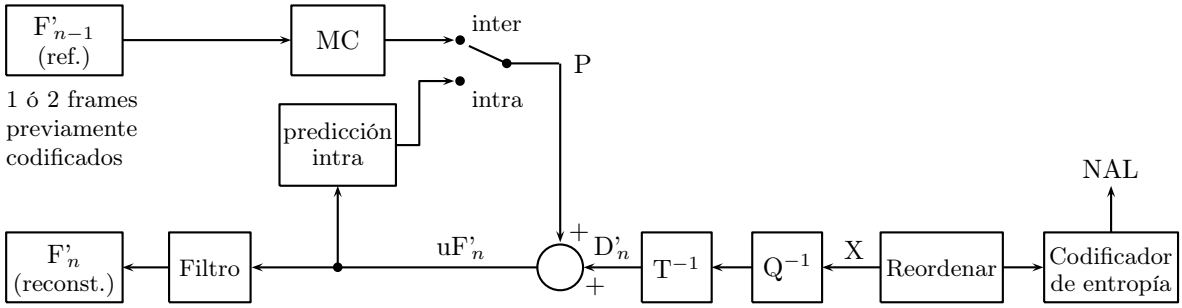


Figura 6.41: Decodificador H.264.

de MPEG-1 (Sección 6.5) antes de seguir leyendo.

La entrada a un codificador de vídeo es un conjunto de frames de vídeo (donde un frame puede consistir en vídeo progresivo o entrelazado). Cada frame se codifica por separado, y el frame codificado se conoce como "imagen codificada". Existen varios tipos de frames, principalmente I, P, y B como se explica al principio de la Sección 6.4, y el orden en que los frames son codificados puede ser diferente del orden en el que tienen que ser mostrados (Figura 6.9). Un frame se divide en sectores (*slices*), y cada slice se divide además en macrobloques como se ilustra en la Figura 6.17.

En H.264, los slices y los macrobloques también tienen tipos. Un slice I puede tener sólo macrobloques de tipo I, un slice P pueden tener macrobloques P e I, y un slice B pueden tener macrobloques B e I (también hay slices de tipos SI y SP). La Figura 6.40 muestra los pasos principales del codificador H.264. Se predice un frame F_n (cada macrobloque en el frame es predicho mediante otros macrobloques del mismo frame o usando macrobloques de otros frames). El frame predicho P se resta del original F_n para producir una diferencia D_n, que es transformada (en la caja etiquetada T), filtrada (en Q), reordenada y codificada mediante entropía. Lo que es nuevo en el codificador H.264 es la ruta de la reconstrucción.

Incluso una mirada superficial a la Figura 6.40 muestra que el codificador consta de dos rutas de datos, una ruta "hacia adelante" (principalmente de izquierda a derecha, mostrada en líneas gruesas) y una de "reconstrucción" (principalmente de derecha a izquierda, mostrada en líneas delgadas). El

decodificador (Figura 6.41) es muy similar a la ruta de reconstrucción del codificador y tiene la mayor parte de los componentes de este último.

La parte principal del codificador es su ruta hacia delante. El frame de video siguiente a ser comprimido se denota por F_n. El frame se divide en macrobloques de 16×16 píxeles cada uno, y cada macrobloque es codificado en modo intra o inter. En cualquier modo, se construye un macrobloque de predicción P basándose en un frame de video reconstruido. En el modo intra, P se construye a partir de muestras previamente codificadas en el frame actual n. Estas muestras son decodificadas y reconstruidas, convirtiéndose en uF'_n en la figura. En el modo inter, P se construye mediante predicción con compensación de movimiento a partir de uno o varios frames de referencia (F_{n-1} en la figura). Observe que la predicción para cada macrobloque puede estar basada en uno o dos frames que ya han sido codificados y reconstruidos (pueden ser frames pasados o futuros). El macrobloque de predicción P es entonces restado del macrobloque actual para producir un macrobloque residual o diferencia D_n. Este macrobloque es transformado y cuantificado para producir un conjunto X de coeficientes de transformación cuantificados. Los coeficientes se reordenan en zigzag y son codificados mediante entropía en una corta cadena de bits. Esta cadena, junto con la información adicional para el decodificador (también codificada con entropía; e incluye el modo de predicción del macrobloque, el tamaño de paso del cuantificador, la información del vector de movimiento que especifica la compensación de movimiento, así como otros ítems), se convierten en el stream comprimido que se pasa a una capa de abstracción de red (*network abstraction layer* o NAL) para la transmisión al exterior de la computadora o para almacenamiento propio. NAL está formado por unidades, cada una con un encabezado y una secuencia de bytes en bruto de carga útil (*raw byte sequence payload* o RBSP) que puede ser enviada como paquetes a través de una red o almacenada como registros que constituyen el archivo comprimido.

Cada slice o sector comienza con una cabecera que contiene el tipo slice, el número de la imagen codificada que incluye el slice, y otra información. Los datos del slice están formados principalmente por macrobloques codificados, pero también pueden existir indicadores de salto u omisión, indicando los macrobloques que han de ser omitidos (no codificados).

Los slices de tipo I (intra) sólo contienen macrobloques I (macrobloques que han sido predichos a partir de macrobloques previos del mismo slice). Este tipo de slice se utiliza en los tres perfiles (los perfiles se discuten más adelante).

Los slices de tipo P (predicho) contienen macrobloques P y/o I. El primer tipo se predice a partir de una imagen de referencia. Este tipo de slice se utiliza en los tres perfiles.

Los slices de tipo B (bipredictivo) contienen macrobloques B y/o I. El primer tipo se predice a partir de una lista de imágenes de referencia. Este tipo de slice se utiliza en los perfiles principal y extendido.

Los slices de tipo SP (*switching* P) son especiales. Facilitan el intercambio entre los streams de codificación y contienen macrobloques extendidos P y/o I. Este tipo de slice se utiliza sólo en el perfil extendido.

Los slices de tipo SI (*switching* I) son también especiales y contienen macrobloques SI extendidos. Este tipo de slice se utiliza sólo en el perfil extendido.

El codificador tiene una ruta de reconstrucción especial cuyo propósito es reconstruir un frame para la codificación de más macrobloques. El paso principal en esta ruta es decodificar los coeficientes X del macrobloque cuantificado. Éstos son reescalados en el bloque Q^{-1} y sufre la transformación inversa en T^{-1}, produciendo un macrobloque diferencia D'_n. Observe que este macrobloque es diferente del macrobloque diferencia original D_n debido a las pérdidas introducidas por la cuantificación. Podemos considerar a D'_n como una versión distorsionada de D_n. El siguiente paso es crear un macrobloque reconstruido uF'_n (una versión distorsionada del macrobloque original) añadiendo el macrobloque de predicción P a D'_n. Finalmente, se aplica un filtro a una serie de macrobloques uF'_n con el fin de suavizar los efectos de bloque. Ésto crea un frame de referencia reconstruido F'_n.

El decodificador (Figura 6.41) introduce una cadena de bits comprimida desde la NAL. Los pri-

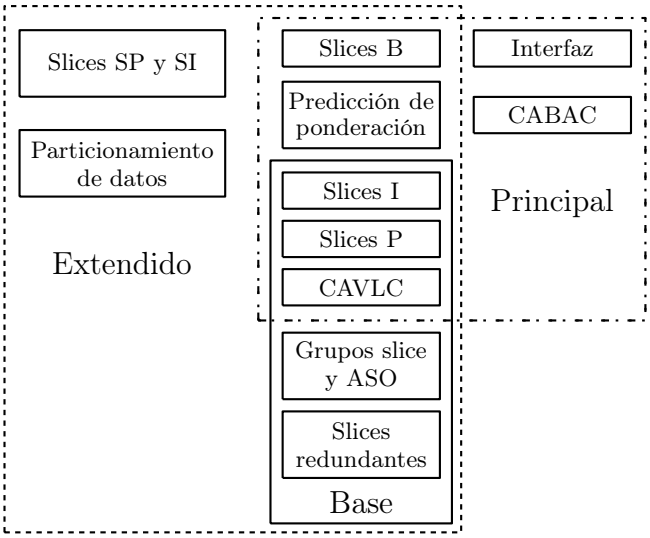


Figura 6.42: Perfiles en H.264.

meros dos pasos (decodificación de entropía y reordenación) producen un conjunto de coeficientes cuantificados X. Una vez que estos son reescalados y se les ha aplicado la transformación inversa generan una D'_n idéntica a la D'_n del codificador). Usando la información adicional de la cabecera de la cadena de bits, el decodificador construye un macrobloque de predicción P, idéntico a la predicción original P creada por el codificador. En el siguiente paso, se añade P a D'_n para producir uF'_n. En el paso final, uF'_n es filtrado para crear el macrobloque decodificado F'_n.

Las Figuras 6.40 y 6.41 y los párrafos anteriores dejan claro que la reconstrucción de la ruta en el codificador tiene una tarea importante. Asegura que tanto el codificador como el decodificador utilizan marcos de referencia idénticos para crear la predicción P. Es importante que las predicciones P en el codificador y el decodificador sean idénticas, ya que cualquier cambio entre ellas tiende a acumularse y dar lugar a un error magnificado o "una derivación" entre el codificador y el decodificador.

El resto de esta sección analiza varios aspectos importantes de H.264. Comenzamos con los perfiles y niveles. H.264 define tres perfiles: básico, principal, y extendido. Cada perfil está formado por un cierto conjunto de funciones de codificación, y cada uno está destinada a diferentes aplicaciones de vídeo. El perfil básico está diseñado para la videotelefonía, la videoconferencia, y la comunicación inalámbrica. Tiene codificación inter e intra con slices I y P, y realiza la codificación de entropía con códigos de tamaño variable de contexto adaptativo (*context-adaptive variable-size codes* o CAVLC). El perfil principal está diseñado para manejar la transmisión de televisión y el almacenamiento de vídeo. Puede tratar el vídeo entrelazado, realizar la codificación inter con slices B, y codificar macrobloques con codificación aritmética basada en el contexto (*context-based arithmetic coding* o CABAC). El perfil extendido está destinado a la transmisión de vídeo y audio. Éste sólo soporta el vídeo progresivo y tiene modos para codificar slices SP y SI. También emplea la partición de datos para un mayor control de errores. La Figura 6.42 muestra los principales componentes de cada perfil y deja claro que el perfil de línea base es un subconjunto del perfil extendido.

Esta sección continúa con los principales componentes del perfil básico. Los que estén interesados en los detalles de los perfiles principales y extendidos pueden consultar [Richardson 03].

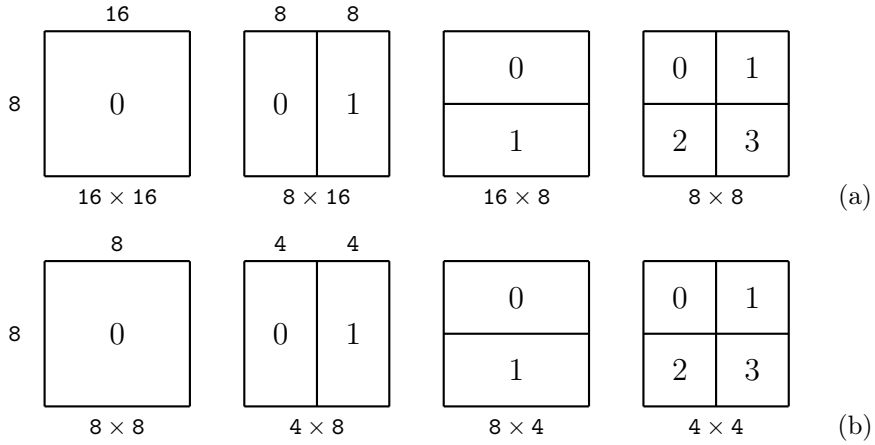


Figura 6.43: Particiones de macrobloque en H.264.

El perfil básico

Este perfil maneja los slices I y P. Los macrobloques en un slice I son codificados como intra, lo que significa que cada macrobloque se predice a partir de macrobloques previos en el mismo slice. Los macrobloques en un slice P pueden ser omitidos, *intra* codificados o *inter* codificados. El último término significa que un macrobloque es predicho a partir del mismo macrobloque en imágenes previamente codificadas. H.264 utiliza un algoritmo de compensación de movimiento con estructura de árbol basado en vectores de movimiento como se describe en la Sección 6.4.

Una vez que un macrobloque F_n ha sido predicho, la predicción es restada de F_n para obtener un macrobloque D_n de residuos. Los residuos se transforman con una transformada entera de 4×4, similar a la DCT. Los coeficientes de la transformada son reordenados (recogidos siguiendo un orden en zigzag), cuantificados, y codificados mediante un código de tamaño variable de contexto adaptativo (*context-adaptive variable-size code* o CAVLC). Otra información que tiene que ir en el stream comprimido es codificada mediante un código de Golomb.

Predicción inter. El modelo de predicción se basa en diversos frames que ya han sido codificados. El modelo es similar al empleado por los viejos estándares de vídeo MPEG, siendo las principales diferencias: (1) puede tratar muestras de video con tamaños de bloque de 16×16 hasta 4×4 y (2) utiliza vectores de movimiento de resolución fina.

El algoritmo de compensación de movimiento con estructura de árbol comienza dividiendo el componente de luminancia de un macrobloque de 16×16 en particiones y calculando un vextor de movimiento separado para cada partición. En consecuencia, la compensación de movimiento de cada partición se efectúa por separado. Un macrobloque puede dividirse de cuatro formas diferentes como se ilustra en la Figura 6.43a. El macrobloque, o bien se deja como un único bloque de 16×16, o bien se divide de una de tres maneras: (1) dos particiones de 8×16, (2) dos particiones de 16×8, ó (3) cuatro particiones de 8×8. Si se elige esta última división, entonces cada partición puede ser dividida adicionalmente en una de cuatro maneras como muestra la Figura 6.43b.

Los tamaños de las particiones aquí enunciadas son para el componente de color de luminancia del macrobloque. Cada uno de los dos componentes de crominancia (Cb y Cr) es particionado de la misma manera que el componente de luminancia, pero con la mitad de los tamaños horizontal y vertical. En consecuencia, cuando un vector de movimiento se aplica a las particiones de cromáticas, sus componentes verticales y horizontales tienen que ser reducidos a la mitad.

Es obvio que el esquema de particionamiento y los vectores de movimiento individuales tienen que ser enviados al decodificador como información adicional. Existe, por lo tanto, un compromiso

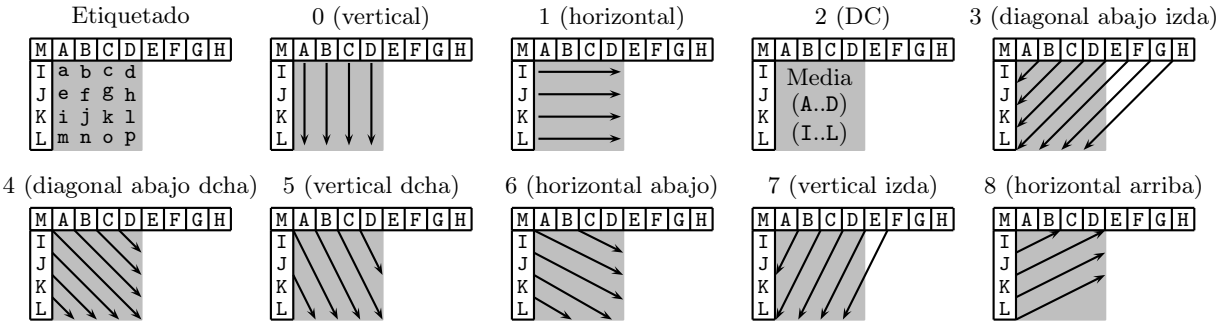


Figura 6.44: Nueve modos de predicción de luminancia.

importante entre la mejor compresión que resulta de las pequeñas particiones y la mayor información complementaria que las particiones pequeñas requieren. La elección del tamaño de la partición es por lo tanto importante, y la regla general es que las grandes particiones deben ser seleccionadas por el codificador para las regiones uniformes (o casi uniformes) de un frame de vídeo, mientras que las particiones pequeñas funcionan mejor para las regiones con mucho ruido de vídeo.

Predicción intra. En este modo, el componente de luminancia de un macrobloque, o bien permanece como un solo bloque de 16×16, o bien se divide en pequeñas particiones de 4×4. En el primer caso, el estándar especifica cuatro métodos (que se resumen más adelante) para predecir el macrobloque. En el último caso, los 16 elementos de una partición de 4×4 son predichos con uno de los nueve modos de 12 elementos de tres particiones previamente codificadas de su izquierda y por encima de ella. Estos 12 elementos se muestran en la parte superior izquierda de la Figura 6.44 en la que están etiquetados, de `A` a `H`, de `I` a `L`, y `M`. Una de las coordenadas, x o y, de estos elementos es -1. Nótese que estos elementos ya han sido codificados y reconstruidos, lo que implica que también están disponibles en el decodificador, no sólo en el codificador. Los 16 elementos de la partición a ser predichos se etiquetan `a`–`p`, y sus coordenadas x e y están en el intervalo $[0, 3]$. Las nueve partes restantes de la Figura 6.44 muestran los nueve modos de predicción a disposición del codificador. Un codificador sofisticado debe tratar los nueve modos, predecir la partición de 4×4 nueve veces, y seleccionar la mejor predicción. El número de modo de cada partición debe ser escrito en el stream comprimido para uso del decodificador. A continuación se presentan los detalles de cada uno de los nueve modos.

El *modo* 0 (*vertical*) es sencillo. Los elementos `a`, `e`, `i`, y `m` se establecen en `A`. En general, $p(x, y) = p(x, -1)$ para $x, y = 0, 1, 2$, y 3.

El *modo* 1 (*horizontal*) es similar. Los elementos `a`, `b`, `c`, y `d` se establecen en `I`. En general $p(x, y) = p(-1, y)$ para $x, y = 0, 1, 2$, y 3.

Modo 2 *(DC).* Cada uno de los 16 elementos se establece en la media $(\texttt{I}+\texttt{J}+\texttt{K}+\texttt{L}+\texttt{A}+\texttt{B}+\texttt{C}+\texttt{D}+4)/8$. Si cualquiera de `A`, `B`, `C`, y `D` no ha sido codificado previamente, entonces cada uno de los 16 elementos se establece en la media $(\texttt{I}+\texttt{J}+\texttt{K}+\texttt{L}+2)/4$. Similarmente, si cualquiera de `I`, `J`, `K`, y `L` no ha sido codificado previamente, entonces, cada uno de los 16 elementos se establece en la media $(\texttt{A}+\texttt{B}+\texttt{C}+\texttt{D}+2)/4$. En el caso donde alguno de `A`, `B`, `C`, y `D` y alguno de `I`, `J`, `K`, y `L` no han sido codificados previamente, cada uno de los 16 elementos se establece en 128. Este modo siempre puede ser seleccionado y utilizado por el codificador.

Modo 3 (*diagonal abajo-izquierda*). Este modo puede utilizarse sólo si todos los ocho elementos, de `A` a `H` han sido codificados. El elemento `p` se establece en $(\texttt{G}+3\texttt{H}+2)/4$, y los elementos $p(x, y)$ donde $x \neq 3$ ó $y \neq 3$ se establecen en $[p(x+y,-1)+2p(x+y+1,-1)+p(x+y+2,-1)+2]/4$. Como ejemplo, el elemento `g`,

cuyas coordenadas son $(2,1)$, se establece en la media $[p(3,-1)+2p(4,-1)+p(5,-1)+2]/4 = [\texttt{D}+2\texttt{E}+\texttt{F}+2]/4$.

Modo 4 (*diagonal abajo-derecha*). Este modo puede utilizarse sólo si los elementos `A` a `D`, `M`, e `I` a `L` han sido codificados. Los cuatro elementos de la diagonal `a`, `f`, `k`, y `p` se establecen en $[\texttt{A}+2\texttt{M}+\texttt{I}+2]/4$. Los elementos $p(x,y)$ donde $x > y$, se establecen en $[p(x-y-2,-1)+2p(x-y-1,-1)+p(x-y,-1)+2]/4$. Los elementos donde $x < y$ se establecen en un promedio similar pero con $y-x$ en lugar de $x-y$. Ejemplo: El elemento `h`, cuyas coordenadas son $(3,1)$, se establece en $[p(0,-1)+2p(1,-1)+p(2,-1)+2]/4 = [\texttt{A}+2\texttt{B}+\texttt{C}+2]/4$.

Modo 5 (*vertical derecha*). Este modo puede utilizarse sólo si los elementos `A` a `D`, `M`, e `I` a `L` han sido codificados. Establecemos la variable z a $2x-y$. Si z es par (igual a 0, 2, 4, ó 6), el elemento $p(x,y)$ se establece en $\left[p\left(x-\frac{y}{2}-1,-1\right)+p\left(x-\frac{y}{2},-1\right)+1\right]/2$. Si z es igual a 1, 3, ó 5, el elemento $p(x,y)$ se establece en $\left[p\left(x-\frac{y}{2}-2,-1\right)+2p\left(x-\frac{y}{2}-1,-1\right)+p\left(x-\frac{y}{2},-1\right)+2\right]/4$. Si $z=-1$, el elemento $p(x,y)$ se establece en $[\texttt{A}+2\texttt{M}+\texttt{I}+2]/4$. Finalmente, si $z=-2$ ó -3, el elemento $p(x,y)$ se establece en $[p(-1,y-1)+2p(-1,y-2)+p(-1,y-3)+2]/4$.

Modo 6 (*abajo horizontal*). Este modo puede utilizarse sólo si los elementos `A` a `D`, `M`, e `I` a `L` han sido codificados. Establecemos la variable z a $2y-x$. Si z es par (igual a 0, 2, 4, ó 6), el elemento $p(x,y)$ se establece en $\left[p\left(-1,y-\frac{x}{2}-1\right)+p\left(-1,y-\frac{x}{2}\right)+1\right]/2$. Si z es igual a 1, 3, ó 5, el elemento $p(x,y)$ se establece en $\left[p\left(-1,y-\frac{x}{2}-2\right)+2p\left(-1,y-\frac{x}{2}-1\right)+p\left(-1,y-\frac{x}{2}\right)+2\right]/4$. Si $z=-1$, el elemento $p(x,y)$ se establece en $[\texttt{A}+2\texttt{M}+\texttt{I}+2]/4$. Finalmente, si $z=-2$ ó -3, el elemento $p(x,y)$ se establece en $[p(x-1,-1)+2p(x-2,-1)+p(x-3,-1)+2]/4$.

Modo 7 (*vertical izquierda*). Este modo puede utilizarse sólo si los elementos `A` a `H`, han sido codificados. Si y es igual a 0 ó 2, el elemento elemento $p(x,y)$ se establece en $\left[p\left(x+\frac{y}{2},-1\right)+p\left(x+\frac{y}{2}+1,-1\right)+1\right]/2$. Si y es igual a 1 ó 3, el elemento elemento $p(x,y)$ se establece en $\left[p\left(x+\frac{y}{2},-1\right)+2p\left(x+\frac{y}{2}+1,-1\right)+p\left(x+\frac{y}{2}+2,-1\right)+2\right]/4$.

Modo 8 (*arriba horizontal*). Este modo se puede utilizar sólo si los elementos `I` a `L`, han sido codificados. Establecemos z en $x+2y$. Si z es par (igual a 0, 2, 4, ó 6), el elemento $p(x,y)$ se establece en $\left[p\left(-1,y+\frac{x}{2}-1\right)+p\left(-1,y+\frac{x}{2}\right)+1\right]/2$. Si z es igual a 1 ó 3, el elemento $p(x,y)$ se establece en $\left[p\left(-1,y+\frac{x}{2}\right)+2p\left(-1,y+\frac{x}{2}+1\right)+p\left(-1,y+\frac{x}{2}+2\right)+2\right]/4$. Si z es igual a 5, $p(x,y)$ se establece en $[\texttt{K}+3\texttt{L}+2]/4$. Si $z>5$, $p(x,y)$ se establece en `L`.

Si el codificador decide predecir un macrobloque entero de 16×16 de componentes de luminancia sin dividirlo, el estándar especifica cuatro modos de predicción, donde los modos 0, 1, y 2 son idénticos a los modos correspondientes para las particiones de 4×4 descritas anteriormente, y el modo 3 especifica una función lineal especial de los elementos de arriba y a la izquierda del macrobloque de 16×16.

También existen cuatro modos de predicción para los bloques de 4×4 de componentes de crominancia.

Filtro de desagregación (*deblocking*). Los macrobloques individuales son parte de un frame de video. El hecho de que cada macrobloque sea predicho por separado introduce distorsiones de bloque; por ello, H.264 tiene un nuevo componente, el filtro de desagregación, diseñado para reducir estas distorsiones. El filtro se aplica después de la transformada inversa. Naturalmente, la transformada inversa es calculada por el decodificador, pero el lector debe recordar, con la ayuda de la Figura 6.40, que en H.264 la transformada inversa también se lleva a cabo mediante la reconstrucción de la ruta del codificador (bloque T^{-1}, antes de que el macrobloque sea reconstruido y almacenado para predicciones posteriores). El filtro mejora la apariencia de los frames de vídeo decodificados mediante el suavizado de los bordes de los bloques. Sin el filtro, los bloques descomprimidos a menudo presentan artefactos debidos a las pequeñas diferencias entre los bordes de los bloques adyacentes.

El filtro se aplica a los bordes horizontales o verticales de los bloques de 4×4 de un macrobloque, excepto los bordes situados en el límite de un slice, en el siguiente orden (Figura 6.45):

1. Filtrar los cuatro límites verticales `a`, `b`, `c`, y `d` del componente de luminancia de 16×16 del macrobloque.

2. Filtrar los cuatro límites horizontales `e`, `f`, `g`, y `h`, del componente de luminancia de 16×16 del macrobloque.

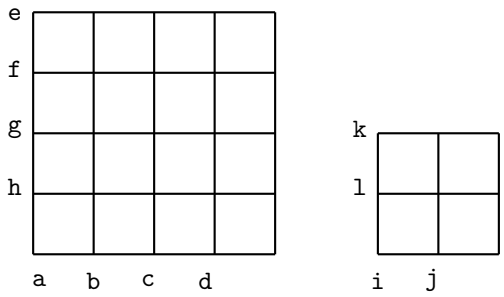


Figura 6.45: Designación de bordes en un macrobloque.

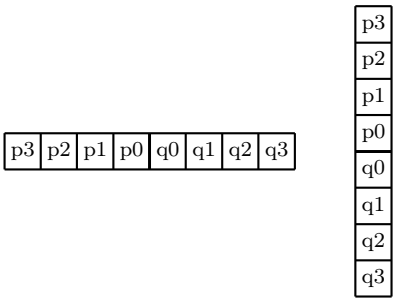


Figura 6.46: Límites horizontales y verticales.

3. Filtrar los dos límites verticales `i` y `j` de cada componente de luminancia de 8×8 del macrobloque.
4. Filtrar los dos límites horizontales `k` y `l` de cada componente de luminancia de 8×8 del macrobloque.

La operación de filtrado es una transformada wavelet discreta n-tap (Sección 5.7), donde n puede ser 3, 4, ó 5. En la Figura 6.46 se pueden ver ocho muestras, de `p0` a `p3` y de `q0` a `q3` que están siendo filtradas, cuatro en cada lado de la frontera (vertical y horizontal) entre macrobloques. Las muestras `p0` y `q0` se encuentran en la frontera. La operación de filtrado puede tener intensidades diferentes, y afecta hasta tres muestras en cada lado de la frontera. La intensidad del filtro es un número entero entre 0 (sin filtrado) y 4 (filtrado máximo), y depende de la cuantificación actual, de los modos de decodificación (inter o intra) de los macrobloques adyacentes, y del gradiente de muestras de vídeo a través de la frontera.

El estándar H.264 define dos parámetros de umbral α y β cuyos valores dependen de los parámetros promedio del cuantificador de los dos bloques `p` y `q` en cuestión. Estos parámetros ayudan a determinar si el filtrado debe llevarse a cabo. Un conjunto de ocho muestras de vídeo `pi qi` se filtran, sólo si $|\texttt{p0} - \texttt{q0}| < \alpha$, $|\texttt{p1} - \texttt{p0}| < \beta$, y $|\texttt{q1} - \texttt{q0}| \leq \beta$.

Cuando la intensidad de filtrado está entre 1 y 3, el proceso de filtrado procede como sigue: Las muestras `p1`, `p0`, `q0`, y `q1` son procesadas mediante un filtro 4-tap para producir $\texttt{p}'\texttt{0}$ y $\texttt{q}'\texttt{0}$. Si $|\texttt{p2} - \texttt{p0}| < \beta|$, se aplica otro filtro 4-tap a `p2`, `p1`, `p0`, y `q0` para producir $\texttt{p}'\texttt{1}$. Si $|\texttt{q2} - \texttt{q0}| < \beta$, se aplica un tercer filtro 4-tap a `q2`, `q1`, `q0`, y `p0` para producir $\texttt{q}'\texttt{1}$ (los dos últimos pasos se llevan a cabo sólo para el componente de luminancia). Cuando la intensidad del filtro es 4, el proceso de filtrado es similar, pero más largo. Cuenta con más pasos y emplea filtros 3-, 4-, y 5-tap.

Transformación. Las Figuras 6.40 y 6.41 muestran que los bloques D_n de residuos se transforman (en el bloque T) por el codificador y la transformada inversa (bloque T^{-1}) tanto por el decodificador como en la ruta de reconstrucción por el codificador. La mayoría de los residuos se transforman en

H.264 mediante una versión especial de 4×4 de la transformada discreta del coseno (DCT, Sección 4.6). Recordemos que un bloque de $n \times n$ genera, tras haber sido transformado por la DCT, un bloque de $n \times n$ de coeficientes de transformada donde el coeficiente superior izquierdo es DC y los $n^2 - 1$ coeficientes restantes son AC. En H.264, ciertos coeficientes DC son transformados mediante la transformada de Walsh-Hadamard (WHT, Sección 4.5.2). El DCT y las transformadas WHT empleadas por H.264 son especiales y tienen las siguientes características:

1. Utilizan sólo números enteros, por lo que no hay ninguna pérdida de precisión.
2. Las partes principales (núcleo o *core*) de las transformadas usan sólo sumas y restas.
3. La parte de la DCT es la multiplicación de escalado y ésto está integrado en el cuantificador, reduciendo de este modo el número de multiplicaciones.

Dado un bloque $\mathbf{X}$ de 4×4 de residuos, la DCT especial empleada por H.264 puede escribirse en la forma:

$$\mathbf{Y} = \mathbf{A}\mathbf{X}\mathbf{A}^T = \begin{bmatrix} a & a & a & a \\ b & c & -c & -b \\ a & -a & -a & a \\ c & -b & b & -c \end{bmatrix} \mathbf{X} \begin{bmatrix} a & b & a & c \\ a & c & -a & -b \\ a & -c & -a & b \\ a & -b & a & -c \end{bmatrix},$$

donde

$$a = \frac{1}{2}, \quad b = \sqrt{\frac{1}{2}} \cos\frac{\pi}{8}, \quad \text{y} \quad c = \sqrt{\frac{1}{2}} \cos\frac{3\pi}{8}.$$

Ésto también puede escribirse en la forma:

$$\begin{aligned} \mathbf{Y} &= \left(\mathbf{C}\mathbf{X}\mathbf{C}^T\right) \otimes \mathbf{E} \\ &= \left\{ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & d & -d & -1 \\ 1 & -1 & -1 & 1 \\ d & -1 & 1 & -d \end{bmatrix} \mathbf{X} \begin{bmatrix} 1 & 1 & 1 & d \\ 1 & d & -1 & -1 \\ 1 & -d & -1 & 1 \\ 1 & -1 & 1 & -d \end{bmatrix} \right\} \otimes \begin{bmatrix} a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \\ a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \end{bmatrix}, \end{aligned}$$

donde $\left(\mathbf{C}\mathbf{X}\mathbf{C}^T\right)$ es una transformada núcleo o *core transform* (donde $d = c/b \approx 0{,}414$), la matriz $\mathbf{E}$ está formada por factores de escala, y la operación $\otimes$ indica la multiplicación escalar de matrices (se multiplican pares de elementos de matriz correspondientes).

Para simplificar los cálculos y asegurar que la transformada sigue siendo ortogonal, se usan los siguientes valores aproximados en la realidad:

$$a = \frac{1}{2}, \quad b = \sqrt{\frac{2}{5}}, \quad d = \frac{1}{2},$$

llevando esta transformada directa especial a la forma:

$$\mathbf{Y} = \left\{ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix} \mathbf{X} \begin{bmatrix} 1 & 2 & 1 & 1 \\ 1 & 1 & -1 & -2 \\ 1 & -1 & -1 & 2 \\ 1 & -2 & 1 & -1 \end{bmatrix} \right\} \otimes \begin{bmatrix} a^2 & ab/2 & a^2 & ab/2 \\ ab/2 & b^2/4 & ab/2 & b^2/4 \\ a^2 & ab/2 & a^2 & ab/2 \\ ab/2 & b^2/4 & ab/2 & b^2/4 \end{bmatrix}.$$

Después de la transformación, cada coeficiente de transformada Y_{ij} es cuantificado mediante la sencilla operación $Z_{ij} = \text{round}\left(Y_{ij}/Q_{\text{paso}}\right)$ donde Q_{paso} es el paso de cuantificación. La operación de redondeo (*round*) también es especial. El codificador puede decidir utilizar "*floor* (piso)" o "*ceiling*

QP	0	1	2	3	4	5	6	7	8	9	10	11	12	...
Q_{paso}	0,625	0,6875	0,8125	0,875	1	1,125	1,25	1,375	1,625	1,75	2	2,25	2,5	...
QP	...	18	...	24	...	30	...	36	...	42	...	48	...	51
Q_{paso}		5		10		20		40		80		160		224

Tabla 6.47: Tamaños de paso de la cuantificación.

(techo)" en lugar de *round* (redondeo)[3]. El estándar H.264 especifica 52 pasos de cuantificación Q_{paso} con rango, de 0,625 a 224 que dependen del parámetro de cuantificación QP cuyos valores abarcan de 0 a 51. La Tabla 6.47 lista estos valores y muestra que cada incremento de 6 en QP duplica Q_{paso}. El amplio rango de valores de la última cantidad permite a un codificador sofisticado controlar de forma precisa el equilibrio entre una tasa de bits baja y una alta calidad. Los valores de QP y Q_{paso} pueden ser diferentes para los bloques de luminancia y crominancia transformados.

Si un macrobloque es codificado en modo de predicción intra 16×16, entonces cada bloque residual de 4×4 en el macrobloque es transformado primero por la transformada núcleo descrita anteriormente, y luego se aplica la transformada de Walsh-Hadamard de 4×4 a los coeficientes DC resultantes como sigue:

$$\mathbf{Y}_D = \frac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \mathbf{W}_D \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix},$$

donde $\mathbf{W}_D$ es un bloque de coeficientes DC de 4×4.

Después de haber predicho un macrobloque y de que los residuos hayan sido transformados y cuantificados, se reordenan. Cada bloque de 4×4 de coeficientes de transformada cuantificados es escaneado en zigzag (Figura 1.8b). Si el macrobloque está codificado en modo intra, habrá 16 coeficientes DC (el coeficiente superior izquierda de cada bloque de luminancia de 4×4). Estos 16 coeficientes están dispuestos en su propio bloque de 4×4 que es escaneado en zigzag. Los 15 coeficientes AC restantes en cada bloque de luminancia de 4×4 también es escaneado siguiendo el mismo orden en zigzag. El mismo reordenamiento se aplica a los bloques de crominancia más pequeños.

Codificación de entropía

H.264 involucra muchos ítems —recuentos, parámetros, y flags o banderas— que tienen que ser codificados. El perfil básico emplea códigos de tamaño variable y un sistema de codificación aritmética adaptativa de contexto especial (CABAC) para codificar parámetros como el tipo de macrobloque, el índice de referencia del frame, las diferencias de vectores de movimiento, y las diferencias de los parámetros de cuantificación. La discusión en esta sección se concentra en la codificación de la secuencia en zigzag de los coeficientes de la transformada cuantificados. Esta secuencia se codifica en un sistema de codificación de longitud variable adaptativo de contexto especial (CAVLC) basado en los códigos de Golomb (Sección 2.5).

CAVLC se utiliza para codificar secuencias en zigzag reordenadas de bloques (tanto de 4×4 como de 2×2) de coeficientes de transformada cuantificados. El método está diseñado para tomar ventaja de las siguientes características de este tipo especial de datos.

1. Tales secuencias son dispersas, tienen runs de ceros, y normalmente terminan con ese run.

[3] *Round* efectúa un redondeo normal; *floor*, siempre redondea hacia abajo; y *ceiling*, siempre redondea hacia arriba. Ejemplos de redondeo son: round (1,4) = 1; round (1,5) = 2; round (1,6) = 2; y floor (1.x) = 1, ceiling (1.x) = 2, para $x = 0 \ldots 9$.

nC:	0	1	2	3	4	5	6	7	8	9	...
Tabla:	1	1	2	2	3	3	3	3	4	4	...

Tabla 6.48: Valores de nC para seleccionar una tabla.

2. Entre los coeficientes distintos de cero, tal secuencia tiene muchos +1s y −1s y éstos se conocen como *trailing* (sucesión o serie de) ±1s.

3. El número de coeficientes distintos de cero en tal secuencia es a menudo similar a los números correspondientes en bloques vecinos. En consecuencia, estos números están correlacionados a través bloques adyacentes.

4. El primer coeficiente (DC) tiende a ser el más grande en la secuencia, y coeficientes no nulos posteriores tienden a ser más pequeños.

CAVLC comienza escaneando y analizando la secuencia de 16 coeficientes. Este paso asigna valores a los seis parámetros de codificación: `coeff_token`, `trailing_ones_sign_flag`, `level_prefix`, `level_suffix`, `total_zeros`, y `run_before`. Estos parámetros son posteriormente utilizados en cinco pasos para seleccionar las tablas de códigos de tamaño variable y los códigos del interior de esas tablas. Los códigos se concatenan para formar una cadena binaria que luego es formateada por la etapa NAL y finalmente se escribe en el stream comprimido. Los cinco pasos de la codificación son los siguientes:

Paso 1. El parámetro `coeff_token` es el número de coeficientes distintos de cero y *trailing* ±1s entre los 16 coeficientes en la secuencia. El número de coeficientes distinto de cero puede ser de 0 a 16. El número de *trailing* ±1s puede ser de hasta el número de coeficientes no nulos, pero `coeff_token` tiene en cuenta (i.e., marca) hasta tres *trailing* ±1s. Cualquier *trailing* ±1s más allá de los tres primeros se codifica por separado como los otros coeficientes distintos de cero. Si el número de coeficientes no nulos es cero, no puede haber *trailing* ±1s. Si el número de coeficientes no nulos es uno, puede haber cero o un *trailing* ±1s. Si el número de coeficientes no nulos es dos, el número de *trailing* ±1s puede ser 0, 1, ó 2. Si el número de coeficientes no nulos es mayor que dos, puede haber entre cero y tres *trailing* ±1s que se marcan mediante `coeff_token`. Puede haber entre cero y 16 coeficientes no nulos, por lo que el número total de valores del parámetro `coeff_token` es $1 + 2 + 3 + 14 \times 4 = 62$. En consecuencia, el estándar H.264 debe especificar una tabla de 62 códigos de tamaño variable para codificar el valor de `coeff_token`. Sin embargo, debido a que el número de coeficientes distintos de cero está correlacionado a través de bloques vecinos, el estándar especifica cuatro de tales tablas (tabla 9–5 en la página 159 de [H.264Draft 06]).

Para seleccionar una de las cuatro tablas, el codificador cuenta los números nA y nB de coeficientes no nulos en los bloques a la izquierda y por encima del bloque actual (si tales bloques existen, ya han sido codificados). Un valor nC se calcula a partir de nA y nB como sigue. Si existen los dos bloques, entonces, $\text{nC} = \text{round}\left[(\text{nA} + \text{nB})/2\right]$. Si sólo existe el bloque de la izquierda, entonces nC = nA. Si sólo existe el bloque de arriba, entonces nC = nB. De lo contrario nC = 0. Por consiguiente, el valor resultante de nC está en el intervalo $[0, 16]$ y se utiliza para seleccionar una de las cuatro tablas como muestra la Tabla 6.48.

Paso 2. El parámetro `trailing_ones_sign_flag` se codifica como un único bit para cada *trailing* ±1 marcado por `coeff_token`. Los *trailing* ±1s se escanean en orden zigzag inverso, y los bits de signo (hasta tres) que se generan se añaden a la cadena binaria generada hasta ese instante.

Paso 3. El siguiente conjunto de bits adicionado a la cadena binaria codifica los valores de los coeficientes no nulos restantes (i.e., los coeficientes distintos de cero excepto aquellos *trailing* ±1s que fueron marcados por `coeff_token`). Estos coeficientes no nulos son escaneados en orden zigzag inverso, y cada uno es codificado en dos partes, como un prefijo (`level_prefix`) y como un sufijo

suffixLength:	0	1	2	3	4	5	6
Umbral:	0	3	6	12	24	48	na

Tabla 6.49: Umbrales para el incremento de `suffixLength`.

(`level_suffix`). El prefijo es un entero entre 0 y 15 y se codifica con un código de tamaño variable especificado por el estándar (tabla 9–6 en la página 162 de [H.264Draft 06]). La longitud de la parte del sufijo es determinada por el parámetro `suffixLength`. Este parámetro puede tener valores entre 0 y 6 y es adaptativo mientras los coeficientes no nulos sean localizados y codificados. El principio de adaptación de `suffixLength` es para emparejar el sufijo de un coeficiente con las magnitudes de los coeficientes no nulos recientemente codificados. Las siguientes reglas regulan la forma en que se adapta `suffixLength`:

1. El parámetro `suffixlength` se inicializa a 0 (pero si hay más de 10 coeficientes no nulos y menos de tres *trailing* $\pm$1s, se inicializa a 1).

2. El siguiente coeficiente según el orden zigzag inverso es codificado.

3. Si la magnitud de este coeficiente es mayor que el umbral actual, se incrementa `suffixLength` en 1. Ésto también selecciona otro umbral. Los valores de los umbrales se muestran en la Tabla 6.49.

Paso 4. El parámetro `total_zeros` es el número de ceros que precede al último coeficiente no nulo. (Este número no incluye el último run de ceros, el run que sigue al último coeficiente distinto de cero.) `total_zeros` es un número entero en el intervalo $[0, 15]$ y es codificado con un código de tamaño variable especificado en el estándar (9–7 y 9–8 en la página 163 de [H.264Draft 06]). Este parámetro y el siguiente (`run_before`) juntos son suficientes para codificar todos los runs de ceros hasta el último coeficiente no nulo.

Paso 5. El parámetro `run_before` es la longitud de un run de ceros que precede a un coeficiente no nulo. Este parámetro es determinado y codificado para cada coeficiente distinto de cero (excepto, por supuesto, el primero) en orden zigzag inverso. La longitud de tal run puede ser cero, pero su mayor valor es 14 (ésto ocurre en el caso improbable de que el coeficiente DC vaya seguido de un run de 14 ceros, que a su vez vaya seguido por un coeficiente no nulo). Mientras el codificador codifica estos run lengths, realiza un seguimiento de su longitud total y este paso termina cuando el total alcanza `total_zeros` (i.e., cuando no quedan más ceros por ser codificados). El estándar especifica una tabla de códigos de tamaño variable (tabla 9–10 en la página 164 de [H.264Draft 06]) para codificar estos run lengths.

Una vez que los cinco pasos han codificado todos los coeficientes no nulos y todos los runs de ceros entre ellos, no hay necesidad de codificar el último run de ceros, el primero (si lo ha) que sigue al último coeficiente distinto de cero, debido a que la longitud de este run será conocida por el decodificador una vez que haya decodificado toda la cadena de bits.

El lector concienzudo debe consultar [Richardson 03] para ejemplos de codificación secuencias y comparar el método CAVLC de codificación de una secuencia de coeficientes de transformada DCT con las (mucho más simples) secuencias similares que son codificadas por JPEG (Sección 4.8.4).

Tengo que decir que encuentro la televisión muy educativa. Al minuto que alguien la enciende me voy a la biblioteca y leo un buen libro.

—Groucho Marx [1890–1977]

Capítulo 7

Compresión de audio

El texto no ocupa mucho espacio en el ordenador. Un libro promedio, formado por un millón de caracteres, se puede almacenar sin comprimir en alrededor de 1 Mbyte, ya que cada carácter de texto ocupa un byte (el Colofón al final del libro ilustra ésto con datos más precisos del libro original, en inglés).

◇ **Ejercicio 7.1 (sol. en pág. 1105):** El hecho de que un libro promedio ocupe alrededor de un millón de bytes es una regla práctica. Explíquese por qué ésto tiene sentido.

En contraste, las imágenes ocupan mucho más espacio, dando otro significado a la frase "una imagen vale más que mil palabras". Dependiendo del número de colores utilizados en una imagen, un solo píxel ocupa entre uno y tres bytes. Por consiguiente, una imagen de 4 Mpíxeles tomada por una cámara digital corriente típica (2006) ocupa entre 512 Kbytes y 12 Mbytes antes de la compresión. Con el advenimiento de las potentes y baratas computadoras personales, en las décadas de 1980 y 1990 llegaron las aplicaciones multimedia, donde el texto, las imágenes, las películas, y el *sonido* se almacenan en el ordenador, y pueden cargarse, descargarse, mostrarse, editarse y reproducirse. Los requerimientos de almacenamiento del sonido son más pequeños que los de las imágenes o las películas, pero más grande que los del texto. Por todo ello, la compresión de audio se ha vuelto importante y ha sido objeto de muchas investigaciones y experimentaciones a lo largo de la década de 1990.

Dos características importantes de la compresión de audio son: (1) puede ser con pérdidas y (2) se requiere una decodificación rápida. La compresión de texto debe ser sin pérdidas, pero las imágenes y el audio pueden perder muchos datos sin una degradación notable de la calidad. Por consiguiente, hay tanto algoritmos de compresión de audio sin pérdidas como con pérdidas. Sólo rara vez ocurre que un usuario desea leer el texto mientras es decodificado y descomprimido, pero ésto es común con el audio. A menudo, el audio se almacena en forma comprimida y tiene que ser descomprimido en tiempo real cuando el usuario quiere escucharlo. Ésta es la razón por la que la mayoría de los métodos de compresión de audio son asimétricos. El codificador puede ser lento, pero el decodificador tiene que ser rápido. Ésta es también la razón por la que los métodos de compresión de audio no están basados en diccionarios. Una compresión basada en métodos de diccionarios puede tener muchas ventajas, pero la decodificación rápida no es una de ellas.

El campo de la compresión de audio se está desarrollando rápidamente y cada vez hay más referencias disponibles. Entre toda esta gran cantidad de datos, cabe destacar especialmente la referencia [wiki.audio 06]. Hay que mantenerse al día y proveer al lector con las últimas técnicas, recursos y enlaces relativos a los trabajos en este importante campo. Un documento importante sobre la compresión de audio sin pérdidas es [Hans y Schafer 01].

Este capítulo comienza con una breve introducción al sonido y la digitalización del sonido. A continuación, analiza las propiedades del sistema auditivo humano (oído y el cerebro) que hacen posible

eliminar mucha información de audio sin afectar negativamente a la calidad del audio reconstruido. El capítulo continúa con un estudio de varios métodos de compresión de audio con y sin pérdidas. La mayoría de los métodos son generales y pueden comprimir cualquier archivo de datos de audio. Otros métodos están dirigidos a la compresión del habla. De especial interés es la detallada descripción de los tres métodos de compresión de audio con pérdidas (capas o *layers*) que utilizan los formatos estándares MPEG-1 y MPEG-2 (Sección 7.14).

7.1. El sonido

Para la mayoría de nosotros, el sonido es un fenómeno muy familiar, porque lo oigo todo el tiempo. Sin embargo, cuando tratamos de definir el sonido, nos encontramos con que podemos acercarnos a este concepto desde dos puntos de vista diferentes, y terminamos con dos definiciones, de la siguiente manera:

Una definición intuitiva: El sonido es la sensación detectada por el oído e interpretado por nuestro cerebro de una manera determinada.

Una definición científica: El sonido es una alteración física en un medio. Se propaga en el medio como una onda de presión mediante el movimiento de átomos o moléculas.

Normalmente escuchamos el sonido cuando se propaga a través del aire y golpea el diafragma en nuestros oídos. Sin embargo, el sonido puede propagarse en muchos medios diferentes. Los animales marinos producen sonidos bajo el agua y responden a sonidos similares. Golpeando el extremo de una barra de metal con un martillo se producen ondas sonoras que se propagan a través de la barra y pueden ser detectado en el otro extremo. Los buenos aislantes de sonido son raros, y el mejor aislante es el vacío, donde no hay partículas para vibrar y propagar la perturbación.

El sonido también puede considerarse una onda, a pesar de que su frecuencia puede cambiar continuamente. Es una onda longitudinal, una donde la perturbación se produce en la dirección de la onda misma. En contraste, las ondas electromagnéticas y las olas del mar son ondas transversales. Sus ondulaciones son perpendiculares a la dirección de la onda.

Al igual que cualquier otra onda, el sonido tiene tres atributos importantes, su velocidad, su amplitud, y su período. La frecuencia de una onda no es un atributo independiente; es el número de periodos que se producen en una unidad de tiempo (un segundo). La unidad de frecuencia es el hercio (Hz). La velocidad del sonido depende principalmente del medio a través del que pasa, y de la temperatura. En el aire, a nivel del mar (una atmósfera de presión), y a 20º Celsius (68º Fahrenheit), la velocidad del sonido es de 343,8 metros por segundo (alrededor de 1128 pies por segundo).

El oído humano es sensible a una amplia gama de frecuencias de sonido, normalmente desde aproximadamente 20 Hz hasta aproximadamente 22 000 Hz, dependiendo de la edad y la salud de una persona. Éste es el rango de *frecuencias audibles*. Algunos animales, especialmente perros y murciélagos, pueden escuchar frecuencias mayores (ultrasonidos). Un cálculo rápido revela las longitudes de onda asociadas con las frecuencias audibles. A 22 000 Hz, cada longitud de onda es de unos 1,56 cm de largo, mientras que a 20 Hz, una longitud de onda es de aproximadamente 17,19 metros de largo.

◇ **Ejercicio 7.2 (sol. en pág. 1106):** Verifíquense estos cálculos.

La amplitud del sonido es también una propiedad importante. Lo percibimos como sonoridad. Tenemos la sensación de sonido cuando las moléculas de aire golpean el diafragma del oído y ejercen presión sobre él. Las moléculas lo mueven atrás y adelante pequeñas distancias que están relacionadas con la *amplitud*, no con el período del sonido. El período de una onda de sonido puede ser de varios metros, sin embargo, una molécula individual en el aire puede desplazar sólo una millonésima parte de un centímetro en sus oscilaciones. Con un ruido muy fuerte, una molécula individual puede desplazar alrededor de un milésima de centímetro. Un dispositivo para medir los niveles de ruido debe basarse en

un diafragma sensible, donde la presión de la onda de sonido sea detectada y convertida a un voltaje eléctrico, que a su vez se muestre como un valor numérico.

El problema con la medición de la intensidad del ruido es que el oído es sensible a una gama muy amplia de niveles de sonido (amplitudes). La relación entre el nivel de sonido de un cañón en la boca del cañón y el nivel más bajo que podemos oir (el umbral de la audición) es de aproximadamente $11-12$ órdenes de magnitud. Si denotamos el menor nivel de sonido audible con 1, entonces el ruido de los cañones ¡tendría una magnitud de 10^{11}! No es conveniente tratar las mediciones en un rango tan amplio, razón por la que las unidades de volumen del sonido utilizan una *escala logarítmica*. El logaritmo (base 10) de $1 = 10^0$ es 0, y el logaritmo de 10^{11} es 11. Utilizando logaritmos, sólo debemos tratar con números en el rango de 0 a 11. De hecho, este intervalo es demasiado pequeño, y típicamente se multiplica por 10 ó por 20, para obtener números entre 0 y 110 ó 220. Éste es el bien conocido (y a veces confuso) sistema de medición de decibelios.

◇ **Ejercicio 7.3 (sol. en pág. 1106):** (Para el matemáticamente débil.) ¿Qué es exactamente un logaritmo?

El decibelio (dB) unidad se define como el logaritmo en base 10 de la relación de dos cantidades físicas cuyas unidades son potencias (energía por unidad de tiempo). El logaritmo se multiplica posteriormente por el conveniente factor de escala 10. (Si el factor de escala no se utiliza, el resultado se mide en unidades llamadas "Bel". El Bel, sin embargo, fue abandonado hace mucho tiempo en favor del decibelio). Por consiguiente, tenemos:

$$\text{Nivel} = 10 \log_{10} \frac{P_1}{P_2} \text{dB},$$

donde P_1 y P_2 se miden en unidades de potencia, tales como vatios, $^{\text{julios}}/_{\text{seg.}}$, $^{\text{gram·cm}}/_{\text{seg.}}$, o caballos de fuerza. Ésto puede ser energía mecánica, energía eléctrica, o cualquier otra cosa. En la medición de la intensidad del sonido, tenemos que usar unidades de potencia acústica. Puesto que incluso un sonido fuerte puede ser generado con muy poca energía, utilizamos el microvatio (10^{-6} vatios) como una unidad más conveniente.

Del Diccionario

Acústica: (1) La ciencia del sonido, incluyendo la generación, transmisión y efectos de las ondas de sonido, tanto audibles como inaudibles. (2) Las cualidades físicas de una habitación u otro recinto (tales como tamaño, forma, cantidad de ruido) que determinan la audibilidad y la percepción del habla y la música dentro de la sala.

El decibel es el logaritmo de un cociente. El numerador, P_1 es la potencia (en microvatios) del sonido cuyo nivel de intensidad se está midiendo. Es conveniente seleccionar como denominador el número de microvatios que produce el más mínimo sonido audible (el umbral de audición). Este número, según muestran los experimentos es de 10^{-6} microvatios $= 10^{-12}$ vatios. En consecuencia, un equipo de música que produce 1 vatio de potencia acústica tiene un nivel de intensidad de:

$$10 \log \frac{10^6}{10^{-6}} = 10 \log \left(10^{12}\right) = 10 \times 12 = 120 \text{ dB}$$

(esto ocurre alrededor del umbral de sensibilidad; véase la Figura 7.1), mientras que un auricular produciendo 3×10^4 microvatios tiene un nivel de:

$$10 \log \frac{3 \times 10^{-4}}{10^{-6}} = 10 \log \left(3 \times 10^2\right) = 10 \times (\log 3 + \log 100) = 10 \times (0{,}477 + 2) \approx 24{,}77 \text{ dB}.$$

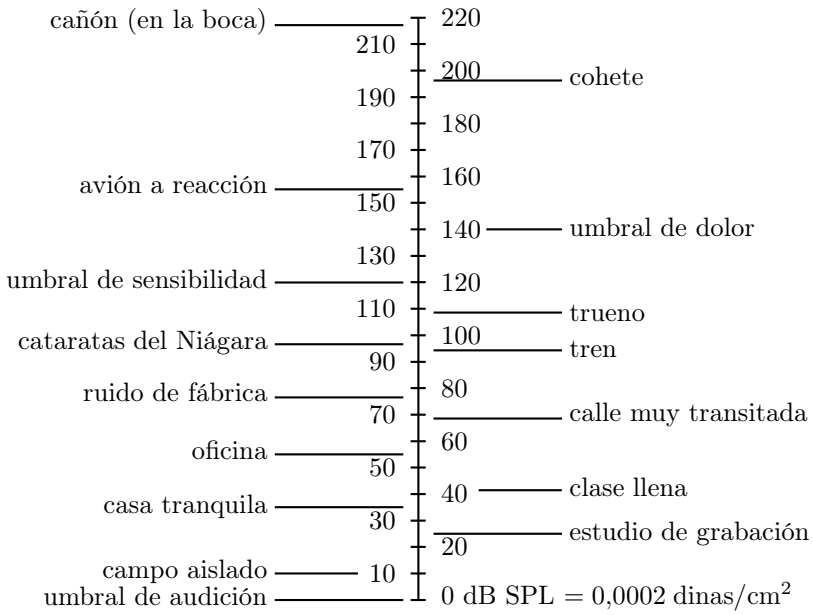


Figura 7.1: Niveles de sonido normales en unidades dB SPL.

En el campo de la electricidad, hay una relación simple entre potencia (eléctrica) P y presión (tensión o voltaje) V. La potencia eléctrica es el producto de la corriente por la tensión $P = I \cdot V$. La actual, sin embargo, es proporcional al voltaje por medio de la ley de Ohm $I = {}^{V}/_{R}$ (donde R es la resistencia). Por tanto, podemos escribir $P = {}^{V^2}/_{R}$, y utilizar presión (voltaje) en nuestras mediciones de decibelios eléctricos.

En trabajos de acústica prácticos, no siempre se tiene acceso a la fuente del sonido, por lo que no podemos medir su potencia eléctrica de salida. En la práctica, podemos encontrarnos en un lugar concurrido, con un medidor de decibelios de sonido en nuestras manos, tratando de medir el nivel de ruido a nuestro alrededor. El medidor de decibelios mide la presión Pr aplicado por las ondas de sonido en su diafragma. Afortunadamente, la potencia acústica por unidad de superficie (denotada por P) es proporcional al cuadrado de la presión del sonido Pr. Ésto es porque la potencia acústica P es el producto de la presión Pr y de la velocidad v del sonido, y porque la velocidad puede expresarse como la presión dividida por la impedancia específica del medio a través del que el sonido se propaga. Ésta es la razón por la que el volumen del sonido se mide comúnmente en unidades de dB SPL (nivel de presión sonora) en lugar de en potencia acústica. La definición es:

$$\text{Nivel} = 10 \log_{10} \frac{P_1}{P_2} = 10 \log_{10} \frac{Pr_1^2}{Pr_2^2} = 20 \log_{10} \frac{Pr_1}{Pr_2} \text{ dB SPL.}$$

El nivel de referencia cero para dB SPL se convierte en 0,0002 ${}^{\text{dinas}}/_{\text{cm}^2}$, donde la dina, una pequeña unidad de la fuerza, es de unos 0,0010197 gramos. Puesto que una dina es igual a 10^{-5} newtons, y ya que un centímetro es 0,01 metros, ese nivel de referencia cero (el umbral de audición) es igual a 0,00002 ${}^{\text{newton}}/_{\text{metro}^2}$. La Tabla 7.2 muestra los valores dB típicos en ambas unidades de potencia y en SPL.

La sensibilidad del oído humano para un nivel de sonido depende de la frecuencia. Experimentos indican que la gente es más sensible a (y, por lo tanto, más molesta por) sonidos de alta frecuencia (por eso las sirenas tienen un tono alto). Es posible modificar el sistema dB SPL para que sea más sensible a las altas frecuencias y menos sensibles a las bajas frecuencias. Ésto se conoce como dBA

vatios	dB	presión $^{n}/_{m^2}$	dB SPL	fuente
30000,0	165	2000,0	160	avión a reacción
300,0	145	200,0	140	umbral de dolor
3,0	125	20,0	120	ruido de fábrica
0,03	105	2,0	100	tráfico en carreteras
0,0003	85	0,2	80	aparatos
0,000003	65	0,02	60	conversación
0,00000003	45	0,002	40	habitación tranquila
0,0000000003	25	0,0002	20	susurro
0,000000000001	0	0,00002	0	umbral de audición

Tabla 7.2: Niveles de ruido en unidades de potencia y presión.

estándar (estándar ANSI S1.4-1983). Existen también estándares dBB y dBC de medición del ruido. (Los ingenieros eléctricos usan también estándares de decibelios llamados dBm, dBm0, y dBrn; véase, por ejemplo, [Shenoi 95].)

Debido al uso de los logaritmos, las medidas de dB no se limitan a sumar. Si el primer trompetista empieza a tocar su trompeta justo antes del concierto, generando, digamos, un nivel de ruido de 70 dB, y el segundo trombonista sigue con su trombón, generando el mismo nivel de sonido, entonces los (pobres) oyentes escuchan el doble de intensidad de ruido, pero esto corresponde a tan sólo 73 dB, no a 140 dB. Para mostrar por qué sucede ésto démonos cuenta de que si

$$10 \log \left(\frac{P_1}{P_2} \right) = 70,$$

entonces

$$10 \log \left(\frac{2P_1}{P_2} \right) = 10 \left[\log_{10} 2 + \log \left(\frac{P_1}{P_2} \right) \right] = 10 \left(0{,}3 + {}^{70}/_{10} \right) = 73.$$

Duplicar el nivel de ruido aumenta el nivel de dB en 3 (si se utilizan unidades SPL, el 3 deberá duplicarse a 6).

◇ **Ejercicio 7.4 (sol. en pág. 1106):** Dos fuentes de sonido A y B producen niveles de dB de 70 y 79 dB, respectivamente. ¿Cuánto más fuerte es la fuente B frente a la A?

7.2. Audio digital

Así como una imagen pueda ser digitalizada y dividida en píxeles, donde cada píxel es un número, el sonido también puede ser digitalizado y dividido en números. Cuando se reproduce el sonido en un micrófono, se convierte en un voltaje que varía continuamente con el tiempo. La Figura 7.3 muestra un ejemplo típico de sonido que comienza en cero y oscila varias veces. Tal tensión es la representación analógica del sonido. La digitalización del sonido se efectúa midiendo el voltaje en muchos puntos en el tiempo, traduciendo cada medición en un número, y escribiendo los números en un archivo. Este proceso se denomina *muestreo* (*sampling*). La onda de sonido es muestreada, y las muestras constituyen el sonido digitalizado. El dispositivo utilizado para el muestreo se llama conversor analógico a digital (ADC).

La diferencia entre una onda de sonido y sus muestras puede ser comparada con la diferencia entre un reloj analógico, donde las manecillas parecen moverse continuamente, y un reloj digital, donde la pantalla cambia abruptamente a cada segundo.

Dado que las muestras de audio son números, son fáciles de editar. Sin embargo, el principal uso de un archivo de audio es para reproducirlo. Para ello se convierten las muestras numéricas de nuevo

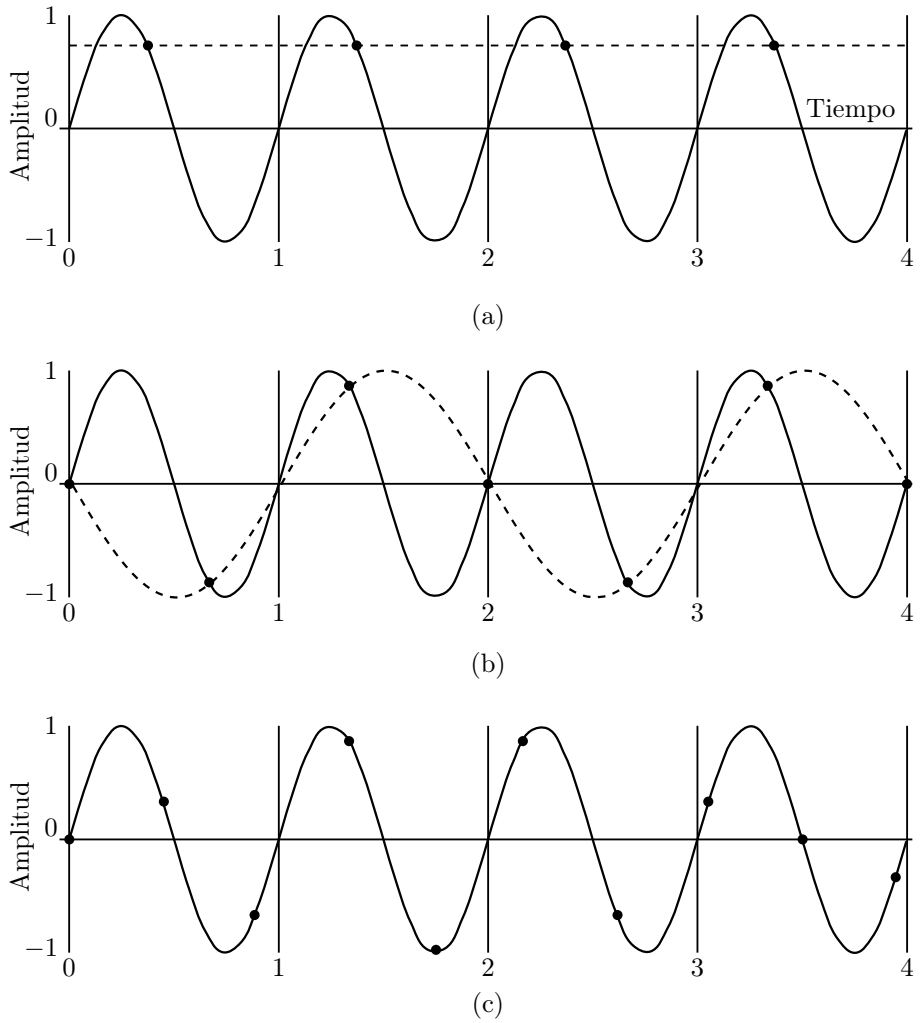


Figura 7.3: Muestreo de una onda de sonido.

en voltajes que son continuamente alimentados en un altavoz. El dispositivo que realiza ésto recibe el nombre de convertidor digital a analógico (DAC). Intuitivamente, es claro que una tasa de muestreo alta daría lugar a una mejor reproducción del sonido, pero también, cuantas más muestras los archivos serán más grandes. En consecuencia, el principal problema en el muestreo de audio es cuán a menudo muestrear un sonido dado.

La Figura 7.3a muestra lo que puede ocurrir si la tasa de muestreo es demasiado baja. La onda de sonido en la figura se muestrea cuatro veces, y las cuatro muestras resultan ser idénticas. Cuando estas muestras se utilizan para reproducir el sonido, el resultado es silencio. En la Figura 7.3b se recogen siete muestras, y parecen "seguir" la onda original muy de cerca. Desafortunadamente, cuando se utilizan para reproducir el sonido, producen la curva mostrada en discontinuo. Simplemente no hay suficientes muestras para reconstruir la onda de sonido original.

La solución al problema de muestreo es tomar muestras de sonido cada algo más de la *frecuencia de Nyquist* (página 544), que es dos veces la frecuencia máxima contenida en el sonido. Por lo tanto, si un sonido contiene frecuencias de hasta 2 kHz, debe ser muestreado en un poco más de 4 kHz.

(Un detalle que a menudo es ignorado en la literatura es que la reconstrucción de la señal es perfecta sólo si se utiliza algo llamado teorema de muestreo de Nyquist-Shannon [Wikipedia 03].) Dicha tasa de muestreo garantiza una reproducción fiel del sonido. Ésto está ilustrado en la Figura 7.3c, en la que se recogen 10 muestras igualmente espaciadas tomadas durante cuatro períodos. Observe que las muestras no tienen que ser tomadas de los máximos o mínimos de la onda; pueden proceder de cualquier punto.

El rango de la audición humana es típicamente, de $16 - 20$ Hz a $20\,000 - 22\,000$ Hz, dependiendo de la persona y de la edad. Cuando el sonido se digitaliza en alta fidelidad, debería, por lo tanto, ser muestreado a poco más de la tasa de Nyquist de $2 \times 22000 = 44000$ Hz. Ésto es por lo que el sonido de alta calidad digital se basa en una velocidad de muestreo de 44 100 Hz. Cualquier tasa inferior a ésta produce distorsiones, mientras que las altas tasas de muestreo no producen ninguna mejora en la reconstrucción (reproducción) del sonido. Podemos considerar la tasa de muestreo de 44 100 Hz un filtro paso bajo, ya que elimina eficientemente todas las frecuencias por encima de los 22 000 Hz.

Muchas aplicaciones de baja fidelidad de sonido muestrean a 11 000 Hz, y el sistema telefónico, originalmente diseñado para las conversaciones, no para las comunicaciones digitales, muestrea el sonido a sólo 8 kHz. En consecuencia, cualquier frecuencia superior a 4000 Hz se distorsiona cuando se envía por teléfono, por lo que es difícil distinguir, por teléfono, entre los sonidos "f" y "s". También es po ésto por lo que cuando alguien te da una dirección a través del teléfono usted debe preguntar: "¿Is H street, as in EFGH?" Con frecuencia, la respuesta es: "No, this is Eighth street, as in sixth, seventh, eighth".[1]

> La reunión fue en la oficina de la ley del Sr. Rogers, en 1415 H Street. Mi hoja de papel dijo 1415 *8th* Street. (La dirección se había dado por teléfono.)
>
> —Richard P. Feynman, *¿Qué importa lo que piensen los demás?*

El segundo problema en el muestreo de sonido es el tamaño de la muestra. Cada muestra se convierte en un número, pero ¿cuán grande debe ser este número? En la práctica, las muestras son normalmente de 8 ó 16 bits, aunque algunas tarjetas de alta calidad de sonido que están disponibles para muchas plataformas de computadoras pueden opcionalmente usar muestras de 32 bits. Suponiendo que el voltaje más alto en una onda de sonido es de 1 voltio, una muestra de 8 bits puede distinguir voltajes tan bajos como $1/256 \approx 0{,}004$ voltios, ó 4 milivoltios (mv). Un sonido bajo, generando una onda inferior a 4 mv, podría ser muestreada como cero y reproducida como silencio. En contraste, con una muestra de 16 bits es posible distinguir sonidos tan bajos como $1/65\,536 \approx 15$ microvoltios (μv). Podemos pensar en el tamaño de la muestra como una cuantificación de los datos de audio originales. Las muestras de ocho bits son más toscamente cuantificadas que las muestras de 16 bits. Como resultado, se produce una mejor compresión pero una reconstrucción más pobre del audio (el sonido reconstruido cuenta solamente con 256 amplitudes diferentes).

◇ **Ejercicio 7.5 (sol. en pág. 1106):** Supongamos que el tamaño de la muestra es de un bit. Cada muestra tiene un valor; ya sea 0, ya sea 1. ¿Qué escuchamos cuando estas muestras se reproducen?

El muestreo de audio también se conoce como *modulación de pulsos codificados* (PCM). Todos hemos oído hablar de la radio AM y FM. Estos términos representan la *modulación en amplitud* y la *modulación en frecuencia*, respectivamente. Indican métodos para modular (i.e., para incluir información binaria en) ondas continuas. El término *modulación de pulsos* se refiere a las técnicas para la conversión de una onda continua a una cadena de números binarios (muestras de audio). Los

[1] Este ejemplo es sólo válido en inglés debido a la semejanza entre las pronunciaciones de H y de Eighth. La traducción: "¿Es la calle H, como en EFGH?" ... "No, es la calle octava, como en sexto, séptimo, octavo".

posibles métodos de modulación de pulsos comprenden la modulación en amplitud de pulsos (PAM), la modulación de la posición de los pulsos (PPM), la modulación del ancho de los pulsos (PWM), y la modulación del número de pulsos (PNM). [Pohlmann 85] es una buena fuente de información sobre estos métodos. En la práctica, sin embargo, PCM ha demostrado ser la forma más eficaz de convertir las ondas de sonido a números. Cuando el sonido estéreo es digitalizado, el codificador PCM multiplexa las muestras de sonido derecho e izquierdo. Como consecuencia, el sonido estéreo muestreado a 22 000 Hz con muestras de 16 bits genera 44 000 muestras a 16 bits por segundo, para dar un total de 704 000 $^{\text{bits}}/_{\text{seg}}$, u 88 000 $^{\text{bytes}}/_{\text{seg}}$.

7.2.1. Audio digital y distribución de Laplace

La distribución de Laplace ya se ha mencionado en la Sección 4.21. Esta corta sección explica la importancia de esta distribución para la compresión de audio. Un archivo de audio grande con una pieza de música larga y compleja tiende a tener todos los posibles valores de muestras de audio. Considere el sencillo caso de muestras de audio de 8 bits, que tienen valores en el intervalo $[0, 255]$. Un gran archivo de audio, con millones de muestras de audio, tenderá a tener muchas muestras de audio centradas en torno a la mitad de este intervalo (alrededor de 128), un menor número de muestras grandes (cercanas al máximo 255), y unas pocas muestras pequeñas (aunque puede haber muchas muestras de audio a 0, ya que muchos tipos de sonido tienden a tener periodos de silencio). La distribución de las muestras puede tener un máximo en su centro y otro pico en el 0. Por consiguiente, las muestras de audio en sí mismas no tienen normalmente una distribución simple.

Sin embargo, cuando examinamos las diferencias entre muestras adyacentes, se observa un comportamiento completamente diferente. Muestras de audio consecutivas tienden a estar correlacionadas, que es por lo que las diferencias entre muestras consecutivas tienden a ser números pequeños. Experimentos con muchos tipos de sonido indican que la distribución de las diferencias de audio se asemejan a la distribución de Laplace. El siguiente experimento refuerza esta conclusión. La Figura 7.4a muestra la distribución de alrededor de 2900 muestras de audio tomadas de un archivo de audio real (el archivo `a8.wav`, donde se pronuncia en inglés la letra única "a"). Es fácil ver que la distribución tiene un pico en 128, pero es muy rudo y no se parece a las distribuciones Gaussiana ni Laplaciana. Además, hay muy pocas muestras más pequeñas que 75 y mayores que 225 (debido al pequeño tamaño del archivo y su particular contenido, una única letra).

No obstante, la parte b de la figura es diferente. Muestra la distribución de las diferencias y es claramente suave, estrecha, y tiene un pico agudo en el 0 (en la figura, el pico es de 256 debido a que los índices de Matlab deben ser no negativos, por lo que se ha añadido un 256 artificial a los índices de diferencias).

Las partes c, d de la figura muestran la distribución de unos 2900 números aleatorios. Los números mismos tienen una distribución plana (aunque no suave), mientras que sus diferencias tienen una distribución muy ruda, centrada en el 0, pero muy diferente de la de Laplace (sólo aproximadamente 24 diferencias son cero).

La distribución de las diferencias no es plana; se asemeja a una distribución Gaussiana muy ruda. Dados R_i enteros aleatorios entre 0 y n, las diferencias de números consecutivos pueden estar entre $-n$ a $+n$, pero estas diferencias se producen con distintas probabilidades. Las diferencias extremas son raras. La única forma de obtener una diferencia de n es tener una $R_{i+1} = 0$ seguida de una $R_i = n$ y restar $n - 0$. Las diferencias pequeñas, por otra parte, tienen una mayor probabilidad debido a que pueden obtenerse en más casos. Una diferencia de 2 se obtiene restando $56 - 42$, pero también $57 - 55$, $58 - 56$ y así sucesivamente.

La conclusión es que las diferencias de valores consecutivos correlacionados tienden a tener una distribución estrecha y puntiaguda, parecida a la distribución de Laplace. Ésto es cierto para las diferencias de muestras de audio, así como para las diferencias de los píxeles consecutivos de una imagen. Un algoritmo de compresión puede tomar ventaja de este hecho y codificar las diferencias con

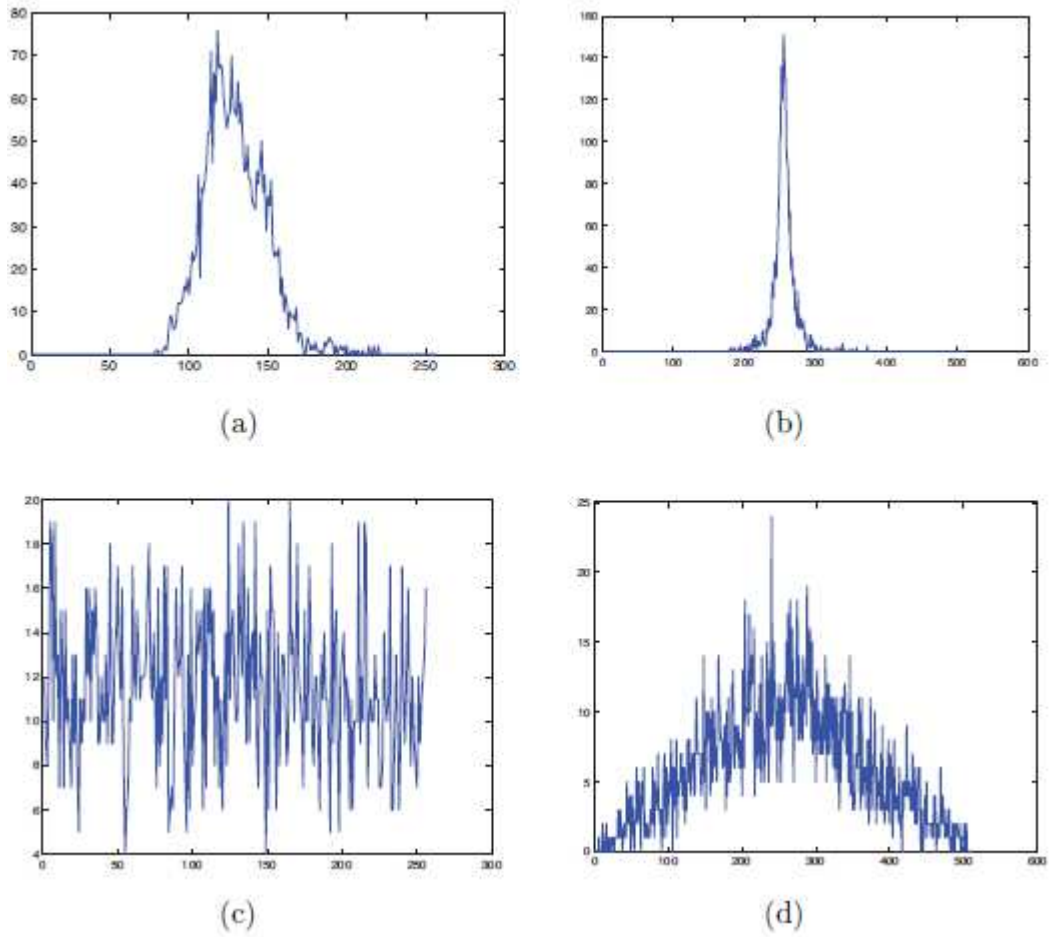


Figura 7.4: Distribuciones de muestras de audio, de números aleatorios, y de diferencias.

```
% Archivo 'LaplaceWav.m'
filename='a8.wav'; dim=2950;% size of file a8.wav
dist=zeros(256,1); ddist=zeros(512,1);
fid=fopen(filename,'r');
buf=fread(fid,dim,'uint8');%introducir enteros sin signo
  for i=46:dim% skip .wav file header
    x=buf(i)+1; dif=buf(i)-buf(i-1)+256;
    dist(x)=dist(x)+1; ddist(dif)=ddist(dif)+1;
  end
figure(1), plot(dist), colormap(gray)% dist de muestras de audio
figure(2), plot(ddist), colormap(gray)% dist de diferencias
dist=zeros(256,1); ddist=zeros(512,1);% limpiar búferes
buf=randint(dim,1,[0 255]);% algunos números aleatorios
  for i=2:dim
    x=buf(i)+1; dif=buf(i)-buf(i-1)+256;
    dist(x)=dist(x)+1; ddist(dif)=ddist(dif)+1;
  end
figure(3), plot(dist), colormap(gray)%dist de números aleatorios
figure(4), plot(ddist), colormap(gray)%dist de diferencias
```

Código para la Figura 7.4.

códigos de tamaño variable que sigan una distribución de Laplace. Una versión más sofisticada puede calcular las diferencias entre los valores reales (muestras de audio o píxeles) y sus valores predichos y después codificar las diferencias (distribuidas siguiendo Laplace). Dos de tales métodos son: MLP de imagen (Sección 4.21) y FLAC (Sección 7.10).

7.3. El sistema auditivo humano

El rango de frecuencias del oído humano abarca aproximadamente desde los 20 Hz hasta unos 20 000 Hz, pero la sensibilidad del oído a los sonidos no es uniforme. Depende de la frecuencia, y los experimentos indican que en un entorno tranquilo la sensibilidad del oído es máxima para las frecuencias en el rango 2 kHz a 4 kHz. La Figura 7.5a muestra el *umbral de audición* para un ambiente tranquilo (véase también la Figura 7.67).

◇ **Ejercicio 7.6 (sol. en pág. 1106):** Indíquese una forma adecuada para llevar a cabo tales experimentos.

También hay que señalar que el rango de la voz humana es mucho más limitada. Abarca sólo desde aproximadamente 500 Hz a cerca de 2 kHz (Sección 7.8).

La existencia del umbral de audición sugiere un enfoque para la compresión de audio con pérdidas. Tan solo hay que eliminar cualquier muestra de audio que esté por debajo del umbral. Puesto que el umbral depende de la frecuencia, el codificador necesita conocer el espectro en frecuencia del sonido que está siendo comprimido en todo momento. El codificador, por lo tanto, tiene que guardar varias de las muestras de audio previamente introducidas en algún instante ($n - 1$ muestras, donde n es una constante o un parámetro controlado por el usuario). Cuando la muestra actual es introducida, el primer paso es transformar las n muestras más recientes en el dominio de la frecuencia (Sección 5.2). El resultado es un número m de valores (llamados *señales*) que indican la intensidad del sonido en m frecuencias diferentes. Si una señal para la frecuencia f es menor que el umbral de audición para f, ésta (la señal) debería ser eliminada.

Además de ésto, se utilizan dos propiedades más del sistema auditivo humano en la compresión de audio. Ellas son el *enmascaramiento de la frecuencia* y el *enmascaramiento temporal*.

El enmascaramiento de la frecuencia (también conocido como *enmascaramiento auditivo*) se produce cuando un sonido que normalmente podemos oir (porque es lo suficientemente fuerte) está enmascarado por otro sonido con una frecuencia cercana. La flecha gruesa en la Figura 7.5b representa una fuente de sonido fuerte en 800 kHz. Esta fuente eleva el umbral normal en sus inmediaciones (la curva de trazos), con el resultado de que el sonido cercano representado por la flecha en "x", un sonido que sería normalmente audible, ya que está por encima del umbral, está ahora enmascarado, y es inaudible (véase también la Figura 7.67). Un buen método de compresión de audio con pérdidas debe identificar este caso y eliminar las señales correspondientes al sonido "x", porque de todos modos no se puede oir. Ésta es una forma de comprimir el sonido con pérdidas.

El enmascaramiento de la frecuencia (el ancho de la curva de trazos de la Figura 7.5b) depende la frecuencia. Ésta varía desde aproximadamente 100 Hz para las frecuencias más bajas hasta las más audibles de 4 kHz para las más altas. El rango de frecuencias audibles, por lo tanto, puede dividirse en un número de *bandas críticas* que indican la sensibilidad decreciente del oído (más bien, la disminución de su poder de resolución) para las frecuencias más altas. Podemos pensar en las bandas críticas como una medida similar a la frecuencia. Sin embargo, en contraste con la frecuencia, que es absoluta y no tiene nada que ver con la audición humana, las bandas críticas se determinan de acuerdo con la percepción de sonido del oído. Por consiguiente, constituyen una medida perceptualmente uniforme de la frecuencia. La Tabla 7.6 muestra 27 bandas críticas aproximadas.

Otra forma de describir bandas críticas es decir que debido a la limitada percepción del oído de las frecuencias, el umbral para una frecuencia f se eleva por un sonido cercano sólo si el sonido está

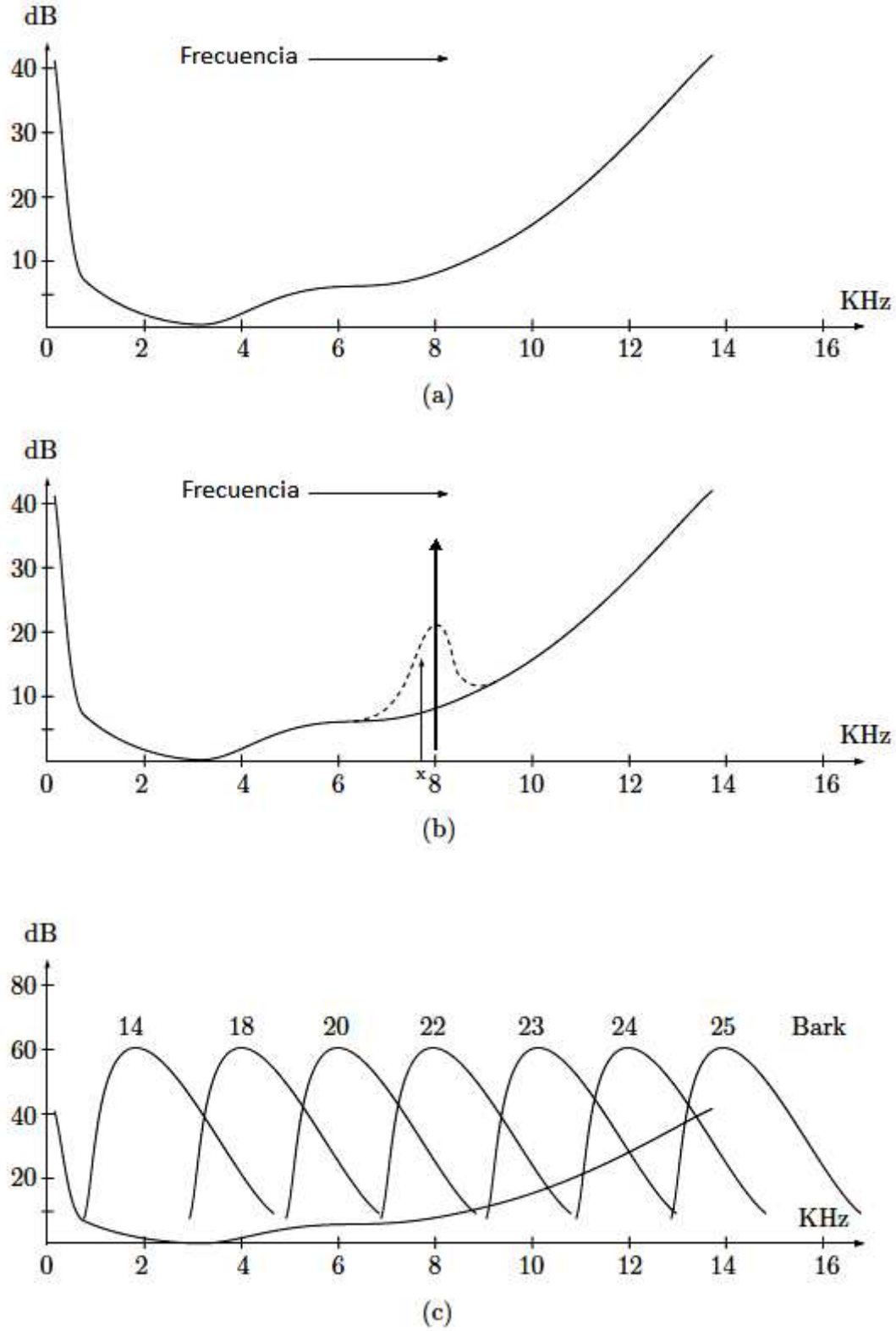


Figura 7.5: Umbral y enmascaramiento del sonido.

banda	rango	banda	rango	banda	rango
0	$0-50$	9	$800-940$	18	$3280-3840$
1	$50-95$	10	$940-1125$	19	$3840-4690$
2	$95-140$	11	$1125-1265$	20	$4690-5440$
3	$140-235$	12	$1265-1500$	21	$5440-6375$
4	$235-330$	13	$1500-1735$	22	$6375-7690$
5	$330-420$	14	$1735-1970$	23	$7690-9375$
6	$420-560$	15	$1970-2340$	24	$9375-11625$
7	$560-660$	16	$2340-2720$	25	$11625-15375$
8	$660-800$	17	$2720-3280$	26	$15375-20250$

Tabla 7.6: Veintisiete bandas críticas aproximadas.

dentro de la banda crítica de f. Ésto también señala el camino hacia el diseño de un algoritmo de compresión con pérdida práctico. La primera señal de audio debe ser transformada en su dominio de la frecuencia, y los valores resultantes (el espectro en frecuencia) debe ser dividido en subbandas que se asemejen a las bandas críticas tanto como sea posible. Una vez realizado ésto, las señales en cada subbanda deben ser cuantificadas de manera que el ruido de cuantificación (la diferencia entre la muestra de sonido original y su valor cuantificado) debe ser inaudible.

Sin embargo, otra forma de ver el concepto de bandas críticas es considerar el sistema auditivo humano un filtro que deja pasar sólo las frecuencias en el rango de (paso de banda) de 20 Hz a 20000 Hz. Visualizamos el sistema auditivo del cerebro como una colección de filtros, cada uno con un paso de banda diferente. Los pasos de banda se denominan bandas críticas. Se superponen y tienen anchuras diferentes. Son estrechas (aproximadamente 100 Hz) a frecuencias bajas y se hacen más anchas (alrededor de 4−5 kHz) a altas frecuencias.

La anchura de una banda crítica se conoce como su tamaño. Las anchuras de las bandas críticas introducen una nueva unidad, el *Bark* (en honor a H. G. Barkhausen) de modo que un Bark es el ancho (en Hz) de una banda crítica. El Bark se define como:

$$1\text{ Bark} = \begin{cases} \frac{f}{100}, & \text{para frecuencias } f < 500\text{ Hz}, \\ 9 + 4\log_2\left(\frac{f}{100}\right), & \text{para frecuencias } f \geq 500\text{ Hz}. \end{cases}$$

La Figura 7.5c muestra algunas bandas críticas, con Barks entre 14 y 25, posicionados por encima el umbral.

Heinrich Georg Barkhausen

Heinrich Barkhausen nació el 2 de diciembre de 1881, en Bremen, Alemania. Pasó toda su carrera como profesor de ingeniería eléctrica en la Technische Hochschule en Dresden, donde se concentró en el desarrollo de los tubos de electrones. También descubrió el llamado "efecto Barkhausen", donde las ondas acústicas se generan en un sólido mediante el movimiento de barreras de dominio cuando el material es magnetizado. También acuñó el término "phon" como una unidad de volumen de sonido. El instituto de Dresden fue destruido, al igual que la mayor parte de la ciudad, en el famoso bombardeo de febrero de 1945. Después de la guerra, Barkhausen ayudó a reconstruir el instituto. Murió el 20 de febrero de 1956.

El enmascaramiento temporal puede producirse cuando un sonido fuerte A de una frecuencia f va precedido o seguido en el tiempo por un sonido más débil B a una frecuencia cercana (o igual). Si el

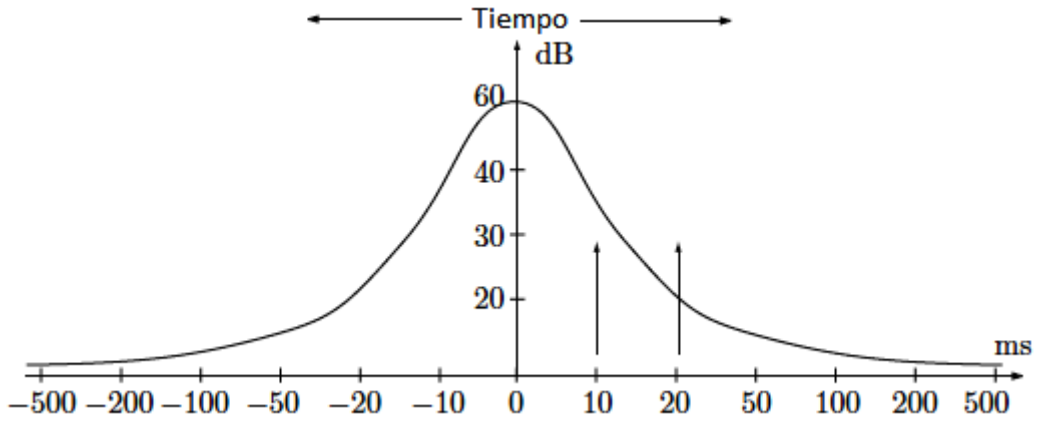


Figura 7.7: Umbral y enmascaramiento del sonido.

intervalo de tiempo entre los sonidos es corto, el sonido B puede ser inaudible. La Figura 7.7 ilustra un ejemplo de enmascaramiento temporal (véase también la Figura 7.68). El umbral de enmascaramiento temporal debido a un fuerte sonido en el tiempo 0 desciende, primero bruscamente, y luego lentamente. Un sonido más débil que 30 dB no será audible si se produce 10 ms antes o después de la fuerte de sonido, pero será audible si el intervalo de tiempo entre los sonidos es de 20 ms.

7.3.1. Métodos convencionales

Los métodos convencionales de compresión, como el RLE, el estadístico y el basado en diccionarios pueden ser utilizados para comprimir archivos de sonido sin pérdidas, pero los resultados dependen en gran medida del sonido específico. Algunos sonidos se pueden comprimir bien bajo RLE, pero no bajo un método estadístico. Otros sonidos pueden prestarse a la compresión estadística, pero pueden expandirse cuando son procesados por un método de diccionario. Aquí mostramos cómo responden los sonidos a cada uno de los tres tipos de métodos de compresión.

RLE puede funcionar bien cuando el sonido contiene largas rachas de muestras idénticas. Con muestras de 8 bits ésto puede ser común. Recordemos que la diferencia entre las dos muestras de 8 bits n y $n+1$ es de aproximadamente 4 mv. A los pocos segundos de música uniforme, donde la onda no oscila más que 4 mv, se puede producir una racha (*run*) de miles de muestras idénticas. Con muestras de 16 bits, las rachas largas pueden ser raras y RLE, en consecuencia, ineficaz.

Los métodos estadísticos asignan códigos de tamaño variable a las muestras de acuerdo con su frecuencia de ocurrencia. Con muestras de 8 bits, hay sólo 256 muestras diferentes, por lo que en un gran archivo de audio, las muestras a veces pueden tener una distribución plana. Este tipo de archivo, por lo tanto, no responde bien a la codificación de Huffman (Véase el Ejercicio 2.17). Con muestras de 16 bits hay más de 65 000 muestras posibles, por lo que a veces puede presentar probabilidades sesgadas (i.e., algunas muestras pueden producirse muy a menudo, mientras que otras pueden ser raras). Dicho archivo, por lo tanto, se puede comprimir mejor con la codificación aritmética, que funciona bien incluso para probabilidades asimétricas.

Los métodos basados en diccionarios esperan encontrar las mismas frases una y otra vez en los datos. Ésto sucede con el texto, donde ciertas cadenas se pueden repetir a menudo. El sonido, sin embargo, es una señal analógica y las muestras particulares generadas dependen de la manera precisa en que trabaja el ADC. Con muestras de 8 bits, por ejemplo, una onda de 8 mv se convierte en una muestra de tamaño 2, pero ondas muy cercanas a ésa, digamos, de 7,6 mv u 8,5 mv, pueden convertirse

en muestras de diferentes tamaños. Ésto es por lo que las partes del lenguaje que suenan igual para nosotros, y por lo tanto, deben convertirse en frases idénticas, terminan siendo digitalizadas de forma ligeramente distinta, y entran en el diccionario como frases diferentes, lo que reduce la compresión. Los métodos basados en diccionarios no son muy adecuados para la compresión de sonido.

> No me gusta como suena ese sonido.
>
> —Heather Graham como Judy Robinson en *Perdidos en el espacio* (1998)

7.3.2. Compresión de sonido con pérdidas

Es posible obtener una mejor compresión del sonido mediante el desarrollo de métodos con pérdidas que aprovechen la ventaja de nuestra percepción del sonido e ignoren aquellos datos para los que el oído humano no es sensible. Ésto es similar a la compresión de imagen con pérdidas, donde se descartan los datos a los que el ojo humano no es sensible. En ambos casos utilizamos el hecho de que la información original (imagen o sonido) es analógica y ya ha perdido cierta calidad al ser digitalizada. La pérdida de algunos datos más, si se hace con cuidado, puede no afectar significativamente el sonido reproducido y, por lo tanto, puede ser indistinguible del original. Describimos brevemente dos enfoques: *compresión de silencios y compansión* (*companding*).

El principio de la compresión de silencios consiste en el tratamiento de pequeñas muestras como si fueran silencios (i.e., como muestras 0). Ésto genera rachas (*run lengths*) de ceros, por lo que la compresión de silencios es en realidad una variante de RLE, adecuada para la compresión de sonido. Éste método utiliza el hecho de que algunas personas tienen un oído menos sensible que otros, y va a tolerar la pérdida de sonidos que son tan suaves que no se van oír todos modos. Los archivos de audio que contienen largos períodos de volumen de sonido bajo que responde a la compresión de silencios mejor que otros archivos con volumen de sonido alto. Este método requiere un parámetro controlado por el usuario que especifica la muestra más grande que debería ser suprimida. Son necesarios también otros dos parámetros, aunque no pueden ser controlados por el usuario. Uno especifica la longitud de la racha más corta de muestras pequeñas, típicamente 2 ó 3. El otro especifica el número mínimo de muestras largas consecutivas que deberían finalizar una racha de silencios. Por ejemplo, una racha de 15 muestras pequeñas, seguida por dos muestras largas, seguidas por 13 muestras pequeñas pueden ser consideradas una racha de silencios de 30 muestras, mientras que las rachas (*runs*) 15, 3, 13 pueden convertirse en dos rachas de silencios distintos de 15 y 13 muestras, sin silencios en el medio.

La compansión[2] (abreviatura de "compresión/expansión") utiliza el hecho de que el oído requiere muestras más precisas a bajas amplitudes (sonidos suaves), pero es más indulgente a mayores amplitudes. Un ADC típico utilizado en las tarjetas de sonido para ordenadores personales convierte los voltajes a números linealmente. Si una amplitud a se convierte en el número n, entonces la amplitud $2a$ se convierte en el número $2n$. Un método de compresión que utiliza la compansión examina cada muestra en el archivo de sonido, y emplea una fórmula no lineal para reducir el número de bits dedicados a ella. Para muestras de 16 bits, por ejemplo, un codificador de compansión puede utilizar una fórmula tan simple como:

$$\text{Muestra} = 32\,767\left(2^{\frac{\text{muestra}}{65536}} - 1\right) \tag{7.1}$$

para reducir cada muestra. Esta fórmula asigna las muestras de 16 bits no lineales a números de 15 bits (i.e., números en el rango [0, 32767]) de manera que las muestras pequeñas sean menos afectadas que las grandes. La Tabla 7.8 ilustra la no linealidad de esta asignación o mapeo. Muestra ocho pares de muestras, donde las dos muestras de cada par difieren en 100. Las dos muestras del primer par son asignadas a los números que difieren en 34, mientras que las dos muestras del último par se asignan a

[2] En el original, en inglés, companding abrevia "compressing/expanding".

Muestra	Mapeado	Dif.	Muestra	Mapeado	Dif.
100 →	35	34	30 000 →	12 236	47
200 →	69		30 100 →	12 283	
1 000 →	348	35	40 000 →	17 256	53
1 100 →	383		40 100 →	17 309	
10 000 →	3 656	38	50 000 →	22 837	59
10 100 →	3 694		50 100 →	22 896	
20 000 →	7 719	43	60 000 →	29 040	65
20 100 →	7 762		60 100 →	29 105	

Tabla 7.8: Muestras de 16 bits mapeadas a números de 15 bits.

los números que difieren en 65. Los números de 15 bits mapeados pueden ser decodificados de nuevo en las muestras originales de 16 bits mediante la fórmula inversa:

$$\text{Muestra} = 65\,536 \log_2 \left(1 + \frac{\text{mapeado}}{32\,767}\right). \tag{7.2}$$

La reducción de números de 16 a 15 bits no produce mucha compresión. La mejor compresión puede lograrse sustituyendo por un número menor el 32 767 en las Ecuaciones (7.1) y (7.2). Un valor de 127, por ejemplo, podría asignar a cada muestra de 16 bits una de 8 bits, produciendo una relación de compresión de 0,5. Sin embargo, la decodificación sería menos precisa. La muestra de 16 bits 60 100, por ejemplo, podría ser mapeada en el número de 8 bits 113, pero este número podría producir 60 172 al ser descifrado mediante la Ecuación (7.2). Incluso peor, la pequeña muestra de 16 bits 1000 podría ser mapeada en 1,35, que tiene que ser redondeado a 1. Cuando la Ecuación (7.2) es utilizada para decodificar un 1, produce 742, significativamente distinto de la muestra original. La cantidad de compresión debería, por lo tanto, ser un parámetro controlado por el usuario, y éste es un interesante ejemplo de un método de compresión donde la relación de compresión ¡se conoce de antemano!

En la práctica, no hay necesidad de pasar por las Ecuaciones (7.1) y (7.2), ya que el mapeo de todas las muestras pueden prepararse de antemano en una tabla. Ambas, codificación y decodificación, son por tanto rápidas.

La compansión no se limita a las Ecuaciones (7.1) y (7.2). Métodos más sofisticados, como μ-law y A-law[3], son de uso común y han sido designados estándares internacionales.

7.4. Formato de audio WAVE

`WAVE` (o simplemente Wave) es el formato de archivo nativo empleado por el sistema operativo Windows para el almacenamiento de datos de audio digitales. El audio se ha vuelto tan importante como las imágenes y el video, por lo que los modernos sistemas operativos soportan un formato nativo para archivos de audio. La popularidad de Windows y la gran cantidad de software disponible para la plataforma PC han garantizado la popularidad del formato Wave, de aquí esta breve descripción.

Primero, tres notas técnicas: (1) El formato de archivo Wave es nativo de Windows y, por lo tanto, se ejecuta en procesadores Intel que utilizan el orden de bytes *little endian*.[4] (2) Los archivos Wave normalmente contienen cadenas de texto para especificar los puntos de referencia, etiquetas, notas y otra información. Las cadenas cuyo tamaño no se conoce de antemano se almacenan en el denominado formato Pascal, donde el primer byte especifica el número de bytes de texto ASCII en la cadena. (3) El formato de archivo `WAV` (o `.WAV`) es un caso Wave especial, donde no se utiliza compresión.

[3]En español sería: ley μ y ley A; pero es más fácil encontrar información en internet con la terminología en inglés.

[4]El byte de menor peso se almacena en la dirección más baja de la memoria y el byte de mayor peso en la más alta.

Byte #	Descripción
$0-3$	`fmt␣`
$4-7$	Longitud del subbloque (24+bytes de formato extra)
$8-9$	Código de compresión de 16 bits
$10-11$	Número de cadenas (1=Mono, 2=Estéreo)
$12-15$	Frecuencia de Muestreo (*Sample Rate*) (Binario, en Hz)
$16-19$	Bytes Por Segundo
$20-21$	Bytes Por Muestra: 1 = 8 bits Mono, 2 = 8 bits Estéreo ó 16 bits Mono, 4 = 16 bits Estéreo
$22-23$	Bytes Por Muestra (*Bits Per Sample*)
$24-$	Bytes de formato extra

Tabla 7.9: Un bloque de formato.

Código	(Hex)	Compresión
0	`0000`	Desconocido
1	`0001`	PCM/descomrimido
2	`0002`	ADPCM Microsoft
6	`0006`	A-law ITU G.711
7	`0007`	μ-law ITU G.711
17	`0011`	ADPCM IMA
20	`0016`	ADPCM ITU G.723 (Yamaha)
49	`0031`	GSM 6.10
64	`0040`	ADPCM ITU G.721
80	`0050`	MPEG
65 536	`FFFF`	Experimental

Tabla 7.10: Códigos de compresión Wave.

Una referencia recomendada para Wave y otros formatos de archivos importantes es [Born 95]. Sin embargo, siempre es una buena idea buscar en Internet nuevas referencias.

La organización de un archivo Wave se basa en la estructura RIFF estándar. [RIFF es un acrónimo de Resource Interchange File Format (Formato de Ficheros de Intercambio de Recursos). Ésta es la base de varios formatos de archivo importantes.] El archivo se compone de bloques, donde cada bloque comienza con un encabezado y puede contener subbloques así como datos. Por ejemplo, los bloques `fmt␣` y `data` son en realidad subbloques del bloque `RIFF`. Se pueden añadir nuevos tipos de bloques en el futuro, lo que lleva a una situación en la que el software existente no reconozca los nuevos bloques. Por lo tanto, la regla general es saltarse cualquier fragmento irreconocible. El tamaño total de un bloque tiene que ser un número par de bytes (un múltiplo de dos bytes), que es por lo que un bloque puede terminar con un byte cero de relleno.

El primer bloque de un archivo Wave es `RIFF`. Comienza con la cadena `RIFF` de 4 bytes, seguida por el tamaño de la parte restante del archivo (en cuatro bytes), seguido por la cadena `WAVE`. Ésto es seguido inmediatamente por los subbloques de formato y de datos. El primero se muestra en la Tabla 7.9.

Las especificaciones principales incluidas en el bloque de formato son las siguientes:

- Código de compresión: Los archivos Wave pueden ser comprimidos. Un caso especial son los archivos `.WAV`, que no se comprimen. Los códigos de compresión se muestran en la Tabla 7.10.
- Número de canales: El número de canales de audio codificados en el bloque de datos. Un valor de 1 indica una señal mono, 2 indica estéreo, y así sucesivamente.

	Hexadecimal																ASCII
00	52	49	46	46	E6	16	00	00	57	41	56	45	66	6D	74	20	R I F F . . . . W A V E f m t ␣
10	10	00	00	00	01	00	01	00	10	27	00	00	20	4E	00	00	. . . . . ' . . . N . .
20	02	00	10	00	64	61	74	61	C2	16	00	00	8B	FC	12	F8	. . . . d a t a . . . . .
30	B9	F8	EB	F8	25	F8	31	F3	67	EE	66	ED	80	EB	F1	E7	. . . . % . 1 . g . f . . . . .
40	3D	EB	5C	09	C3	1B	44	E9	EF	FF	18	39	35	0E	5B	0A	= . \ . . . D . . . . 9 5 . [.

Tabla 7.11: Ejemplo de un archivo .WAV.

- Frecuencia de muestreo: El número de muestras de audio por segundo. Este valor no se ve afectado por el número de canales.
- Promedio de bytes por segundo: ¿Cuántos bytes de datos wave por segundo deben ser enviados a un convertidor D/A para poder reproducir el archivo wave? Este valor es el producto de la tasa de muestreo y el bloque de alineación.
- Bloque de alineación: El número de bytes por porción de muestra. Este valor no se ve afectado por el número de canales.
- Bits significativos por muestra: El número de bits por muestra de audio, los más comunes son 8, 16, 24, ó 32. Si el número de bits no es un múltiplo de 8, entonces el número de bytes utilizados por la muestra se redondea hasta el tamaño más cercano en bytes y los bytes no utilizados se establecen en 0 y son ignorados.
- Bytes de formato adicionales: El número de bytes de formato adicionales que siguen. Estos bytes se utilizan para los códigos de compresión distintos de cero. Su significado depende del método de compresión específico utilizado. Si este valor no es par (un múltiplo de 2), debe añadirse un byte de relleno de ceros al final de esta información para alinearlo, pero el propio valor debe permanecer impar.

El bloque de datos comienza con la cadena `data`, que es seguida por el tamaño de los datos que siguen (como un entero de 32 bits). Ésto es seguido por los bytes de datos (muestras de audio).

La Tabla 7.11 muestra los primeros 80 bytes de un sencillo archivo `.WAV`.

El archivo comienza con la cadena de identificación `RIFF`. Los siguientes cuatro bytes son la longitud, que es $\texttt{16E6}_{16} = 5\,862_{10}$ bytes. Éste número es la longitud de todo el archivo menos los ocho bytes para `RIFF` y la longitud. Las cadenas `WAVE` y $\texttt{fmt}_{\sqcup}$ siguen a continuación.

La segunda fila comienza con la longitud $\texttt{10}_{16} = 16_{10}$ del bloque de formato. Ésta indica que no hay bytes adicionales. Los siguientes dos bytes generan el valor 1 (código de compresión correspondiente a "sin comprimir"), seguido de otros dos bytes que también forman el valor 1 (audio mono, un único canal). La tasa de muestreo del audio es de $\texttt{2710}_{16} = 10\,000_{10}$, y los bytes por segundo son $\texttt{4E20}_{16} = 20\,000_{10}$.

La tercera fila comienza con los bytes por muestra (dos bytes que generan el 2, indicando estéreo de 8 bits ó mono de 16 bits). Los siguientes dos bytes son los bits por muestra (16 en nuestro caso). Ahora sabemos que este audio es mono con muestras de audio de 16 bits. El subbloque de datos sigue inmediatamente. Comienza con la cadena `data`, seguida de los cuatro bytes $\texttt{16C2}_{16}$ que indican que a continuación se encuentran 5826_{10} bytes de datos de audio. Los primeros cuatro bytes de este tipo (dos muestras de audio) son: `8B FC 12 F8`.

7.5. Companding (companding) μ-Law y A-Law

Estos dos estándares internacionales, conocidos formalmente como recomendación G.711, están documentados en [ITU-T 89]. Emplean logaritmos basados en funciones para codificar las muestras de audio para los servicios de telefonía digital RDSI[5] (red digital de servicios integrados), mediante cuantificación no lineal. El hardware RDSI muestrea la señal de voz del teléfono a 8 000 veces por segundo, y genera muestras de 14 bits (13 para A-law). El método de companding μ-law se utiliza en América del Norte y Japón, y A-law se utiliza en otros lugares. Los dos métodos son similares; difieren principalmente en sus cuantificaciones[6] (*midtread* vs. *midriser*).

Los experimentos (documentados en [Shenoi 95]) indican que las amplitudes bajas de las señales del habla contienen más información que las amplitudes altas. Es por ésto por lo que la cuantificación no lineal tiene sentido. Imagine una señal de audio enviada por una línea telefónica y digitalizada en muestras de 14 bits. Cuanto más fuerte sea la conversación, tanto mayor será la amplitud, y mayor el valor de la muestra. Puesto que las amplitudes altas son menos importantes, pueden ser toscamente cuantificadas. Si la muestra más grande, que es de $2^{14}-1=16\,383$, se cuantifica a 255 (el número más grande de 8 bits), entonces el factor de compresión es de $^{14}/_{8}=1{,}75$. Cuando es decodificado, un código 255 será muy diferente del original 16 383. Decimos ésto porque debido a la tosca cuantificación, las muestras grandes generan un ruido de cuantificación alto. Las muestras más pequeñas deben ser finamente cuantificadas, por lo que acaban con un bajo ruido de cuantificación.

El codificador μ-law introduce muestras de 14 bits y produce palabras de código de 8 bits. El A-law introduce muestras de 13 bits y también produce palabras de código de 8 bits. Las señales telefónicas son muestreadas a 8 kHz (8 000 veces por segundo), por lo que el codificador μ-law recibe $8\,000\times 14=112\,000$ $^{\text{bits}}/_{\text{seg}}$. Con un factor de compresión de 1,75, el codificador genera 64 000 $^{\text{bits}}/_{\text{seg}}$. El estándar G.711 [G.711 72] también especifica las tasas de salida de 48 Kbps y 56 Kbps.

El codificador μ-law recibe una muestra de entrada x *con signo* de 14 bits. Por consiguiente, la entrada está en el rango $[-8\,192,+8\,191]$. La muestra es normalizada para el intervalo $[-1,+1]$, y el codificador utiliza la expresión logarítmica:

$$\operatorname{sgn}(x)\frac{\ln(1+\mu|x|)}{\ln(1+\mu)},\ \text{donde}\ \operatorname{sgn}(x)=\begin{cases}+1, & x>0,\\ 0, & x=0\\ -1, & x<0\end{cases}$$

(y μ es un entero positivo), para calcular y producir un código de 8 bits en el mismo intervalo $[-1,+1]$. La salida es posteriormente escalada al rango $[-256,255]$. La Figura 7.12 muestra esta salida como una función de la entrada para los tres valores de μ: 25, 255, y 2555. Es evidente que los valores grandes de μ producen una cuantificación tosca para grandes amplitudes. Tales valores asignan más bits para las amplitudes más pequeñas e importantes. El estándar G.711 recomienda el uso de $\mu=255$. El diagrama muestra sólo los valores de entrada no negativos (i.e., de 0 a 8191). La zona negativa del diagrama tiene la misma forma pero con entradas y salidas negativas.

El codificador A-law utiliza la expresión similar:

$$\begin{cases}\operatorname{sgn}(x)\frac{A|x|}{1+\ln(A)}, & \text{para } 0\le|x|<\frac{1}{A},\\ \operatorname{sgn}(x)\frac{1+\ln(A|x|)}{1+\ln(A)} & \text{para } \frac{1}{A}\le|x|<1.\end{cases}$$

El estándar G.711 recomienda el uso de $A=87{,}6$.

[5] En inglés, ISDN (integrated services digital network).
[6] *Midtread* es "escalón en la mitad"; *midriser* es "aumento en la mitad".

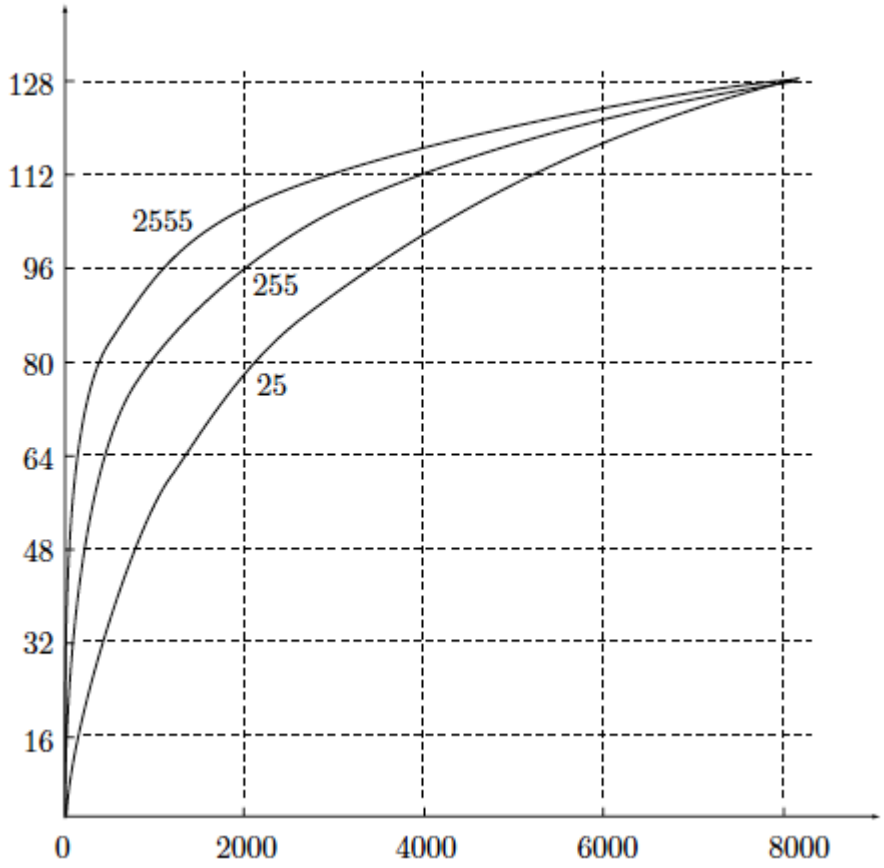


Figura 7.12: μ-law para valores de μ de 25, 255, y 2555.

```
dat=linspace(0,1,1000);
mu=255;
plot(dat*8159,128*log(1+mu*dat)/log(1+mu));
```

Código en Matlab de la Figura 7.12. Nótese cómo la entrada se normaliza en el rango $[0, 1]$ antes de efectuar los cálculos, y cómo la salida es escalada desde el intervalo $[0, 1]$ hasta el $[0, 128]$.

Los sencillos ejemplos siguientes ilustran la naturaleza no lineal de la μ-law. Las dos muestras (normalizadas) de entrada —0,15 y 0,16— son transformadas mediante la μ-law a las salidas 0,6618 y 0,6732. La diferencia entre las salidas es 0,0114. Por otro lado, los dos muestras de entrada 0,95 y 0,96 (las mayores entradas pero con la misma diferencia) se transforman a 0,9908 y 0,9927. La diferencia entre estas dos salidas es 0,0019; mucho más pequeño.

Las muestras más grandes se decodifican con más ruido, y las muestras más pequeñas se decodifican con menos ruido. Sin embargo, la relación señal–ruido (SNR, Sección 4.2.2) es constante porque tanto la μ-law como la SNR utilizan expresiones logarítmicas.

Los logaritmos son lentos de calcular, por lo que el codificador μ-law realiza cálculos mucho más simples que generan una aproximación. La salida especificada por el estándar G.711 es una palabra de código de 8 bits cuyo formato se muestra en la Figura 7.13.

El bit P en la Figura 7.13 es el bit de signo de la salida (el mismo que el bit de signo de la muestra

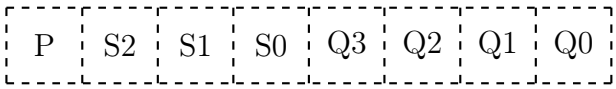


Figura 7.13: Palabra de código de μ-Law G.711.

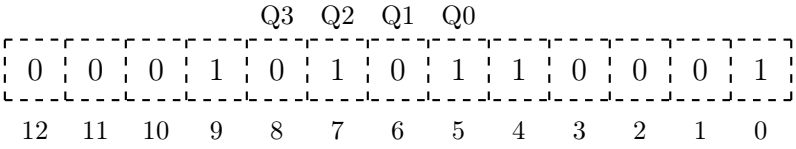


Figura 7.14: Ejemplo de codificación de entrada -656.

de entrada con signo de 14 bits). Los bits S2, S1 y S0 son el código de segmento, y los bits de Q3 a Q0 son el código de cuantificación. El codificador determina el código de segmento mediante: (1) la adición un sesgo de 33 al valor absoluto de la muestra de entrada, (2) la determinación de la posición de bit del bit a 1 más significativo entre los bits, de 5 a 12 de la entrada, y (3) la sustracción de 5 desde esa posición. El código de cuantificación de 4 bits se establece en los cuatro bits siguientes de la posición de bit determinada en el paso 2. El codificador ignora los bits restantes de la muestra de entrada, e invierte (complemento a 1) la palabra de código antes de ofrecerla como salida.

Usamos como ejemplo de muestra de entrada: -656. La muestra es negativa, por lo que el bit P es 1. La adición de 33 al valor absoluto de la entrada da $689 = 0001010110001_2$ (Figura 7.14). El bit a 1 más significativo en las posiciones, de 5 a 12, se encuentra en la posición 9. El segmento de código es, por tanto, $9 - 5 = 4$. El código de cuantificación son los cuatro bits 0101 en las posiciones 8−5, y los restantes cinco bits 10001 son ignorados. La palabra de código de 8 bits (que posteriormente es invertida) se convierte en:

P	Q2	S1	S0	Q3	Q2	Q1	Q0
1	1	0	0	0	1	0	1

El decodificador μ-law introduce la palabra de código de 8 bits y la invierte. A continuación, la decodifica como sigue:

1. Multiplica el código de cuantificación por 2 y añade 33 (el sesgo) al resultado.
2. Multiplica el resultado por 2 elevado a la potencia del código de segmento.
3. Decrementa el resultado en el valor del sesgo.
4. Usa el bit P para determinar el signo del resultado.

La aplicación de estos pasos a nuestro ejemplo produce:

1. El código de cuantificación es $101_2 = 5$, por tanto $5 \times 2 + 33 = 43$.
2. El segmento de código es $100_2 = 4$, por tanto $43 \times 2^4 = 688$.
3. El decremento en el valor del sesgo es $688 - 33 = 655$.
4. El bit P es 1, así que el resultado final es -655. Por consiguiente, el error de cuantificación (el ruido) es 1; muy pequeño.

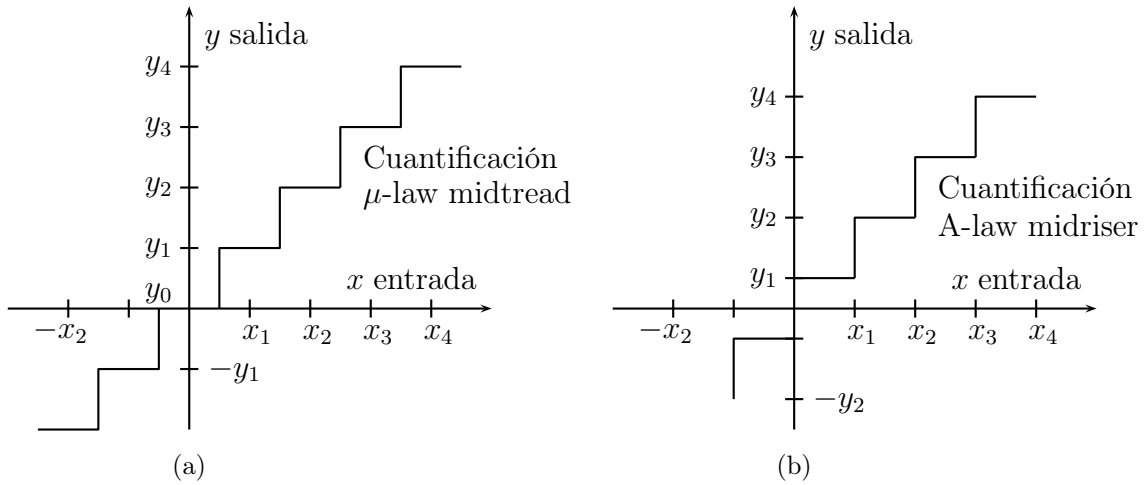


Figura 7.15: (a) Cuantificación μ-Law Midtread. (b) Cuantificación A-Law Midriser.

La Figura 7.15a ilustra la naturaleza de la cuantificación μ-law *midtread*. El cero es uno de los valores de salida válidos, y los pasos de cuantificación se centran en el valor de entrada 0. Los pasos están organizados en ocho segmentos de 16 pasos cada uno. Los pasos dentro de cada segmento tienen la misma anchura, pero doblan la anchura de un segmento al siguiente. Si denotamos el número de segmento por i (donde $i = 0, 1, \ldots, 7$) y la anchura de un segmento por k (donde $k = 1, 2, \ldots, 16$), entonces la mitad del escalón de cada paso en la Figura 7.15a (i.e., los puntos etiquetados x_j) está dado por:

$$x\,(16i + k) = T\,(i) + k \times D\,(i)\,, \tag{7.3}$$

donde las constantes $T\,(i)$ y $D\,(i)$ son el valor inicial y el tamaño de paso para el segmento i, respectivamente. Éstos vienen dados por:

i	0	1	2	3	4	5	6	7
$T\,(i)$	1	35	103	239	511	1055	2143	4319
$D\,(i)$	2	4	8	16	32	64	128	256

La Tabla 7.16 muestra algunos valores de los puntos de ruptura (puntos x_j) y las salidas (puntos y_j) representados en la Figura 7.15a.

El funcionamiento del codificador A-law es similar, excepto que la cuantificación (Figura 7.15b) es de la variedad *midriser*. Los puntos de ruptura x_j vienen dados por la Ecuación (7.3), pero el valor inicial $T\,(i)$ y el tamaño del paso $D\,(i)$ para el segmento i son distintos de los utilizados por el codificador μ-law y vienen dados por:

i	0	1	2	3	4	5	6	7
$T\,(i)$	0	32	64	128	256	512	1024	2048
$D\,(i)$	2	2	4	8	16	32	64	128

La Tabla 7.17 muestra algunos valores de los puntos de ruptura (puntos x_j) y las salidas (puntos y_j) representados en la Figura 7.15b

segmento 0		segmento 1		⋯	segmento 7	
Puntos de ruptura	Valores de salida	Puntos de ruptura	Valores de salida		Puntos de ruptura	Valores de salida
	$y_0 = 0$		$y_{16} = 33$	⋯		$y_{112} = 4191$
$x_1 = 1$		$x_{17} = 35$			$x_{113} = 4319$	
	$y_1 = 2$		$y_{17} = 37$	⋯		$y_{113} = 4447$
$x_2 = 3$		$x_{18} = 39$			$x_{114} = 4575$	
	$y_2 = 4$		$y_{18} = 41$	⋯		$y_{114} = 4703$
$x_3 = 5$		$x_{19} = 43$			$x_{115} = 4831$	
	$y_3 = 6$		$y_{19} = 45$	⋯		$y_{115} = 4959$
$x_4 = 7$		$x_{20} = 47$			$x_{116} = 5087$	
⋯				⋯		
⋯				⋯		
$x_{15} = 29$		$x_{31} = 91$			$x_{127} = 7903$	
	$y_{15} = 28$		$y_{31} = 93$	⋯		$y_{127} = 8031$
$x_{16} = 31$		$x_{32} = 95$			$x_{128} = 8159$	

Tabla 7.16: Especificaciones del cuantificador μ-Law.

segmento 0		segmento 1		⋯	segmento 7	
Puntos de ruptura	Valores de salida	Puntos de ruptura	Valores de salida		Puntos de ruptura	Valores de salida
$x_0 = 0$		$x_{16} = 32$			$x_{112} = 2048$	
	$y_1 = 1$		$y_{17} = 33$	⋯		$y_{113} = 2112$
$x_1 = 2$		$x_{17} = 34$			$x_{113} = 2176$	
	$y_2 = 3$		$y_{18} = 35$	⋯		$y_{114} = 2240$
$x_2 = 4$		$x_{18} = 36$			$x_{114} = 2304$	
	$y_3 = 5$		$y_{19} = 37$	⋯		$y_{115} = 2368$
$x_3 = 6$		$x_{19} = 38$			$x_{115} = 2432$	
	$y_4 = 7$		$y_{20} = 39$	⋯		$y_{116} = 2496$
⋯				⋯		
⋯				⋯		
$x_{15} = 30$		$x_{31} = 62$			$x_{128} = 4096$	
	$y_{16} = 31$		$y_{32} = 63$	⋯		$y_{127} = 4032$

Tabla 7.17: Especificaciones del cuantificador μ-Law.

El codificador A-law genera una palabra código de 8 bits con el mismo formato que el codificador μ-law. Establece el bit P al signo de la muestra de entrada. Después, determina el segmento código con los pasos siguientes:

1. Determina la posición de bit, del bit a 1 más significativo de entre los siete bits más significativos de la entrada.

2. Si encuentra dicho bit 1, el código de segmento se convierte en esa posición menos 4. De lo contrario, el segmento de código se convierte en cero.

El código de cuantificación de 4 bits se establece en los cuatro bits siguientes a la posición de bit determinada en el paso 1, o a la mitad del valor de entrada si el código de segmento es cero. El codificador ignora los bits restantes de la muestra de entrada, e invierte el bit P y los bits de numeración par de la palabra de código antes de ofrecerla como salida.

El decodificador A-law decodifica una palabra de código de 8 bits en una muestra de audio de 13 bits de la siguiente manera:

1. Invierte el bit P y los bits de numeración par de la palabra de código.

2. Si el código de segmento es distinto de cero, el decodificador multiplica el código de cuantificación por 2 e incrementa ésto en el sesgo (33). El resultado es multiplicado por 2 y elevado a la potencia de (código de segmento menos 1). Si el código de segmento es 0, el descodificador genera dos veces el código de cuantificación, más 1.

3. El bit P se utiliza entonces para determinar el signo de la salida.

Normalmente, las palabras de código de salida son generadas por el codificador a una tasa de 64 Kbps. El estándar G.711 [G.711 72] también prevé otras dos tasas, como sigue:

1. Para lograr una tasa de salida de 48 Kbps, el codificador enmascara los dos bits menos significativos de cada palabra de código. Ésto funciona, porque $^{6}/_{8} = {}^{48}/_{64}$.

2. Para lograr una tasa de salida de 56 Kpbs, el codificador enmascara el bit menos significativo de cada palabra de código. Ésto funciona, porque $^{7}/_{8} = {}^{56}/_{64} = 0{,}875$.

Ésto se aplica tanto al μ-law como al A-law. El decodificador generalmente rellena las posiciones de los bits de enmascaramiento con ceros antes de la decodificación de una palabra de código.

7.6. Compresión de audio ADPCM

La compresión es posible solamente porque el audio, y por lo tanto las muestras de audio tienden a tener redundancias. Muestras de audio adyacentes tienden a ser similares, de la misma manera que los píxeles vecinos en una imagen tienden a tener colores similares. La forma más sencilla para explotar esta redundancia es restar muestras adyacentes y codificar las diferencias, que tienden a ser números enteros pequeños. Cualquier método de compresión de audio basado en este principio se denomina modulación por codificación de pulsos diferencial (DPCM o *differential pulse code modulation*). Tales métodos, sin embargo, son ineficientes, debido a que no se adaptan ellos mismos a las variantes magnitudes de una cadena de audio. Se obtienen mejores resultados con una versión adaptativa, y cualquier versión de este tipo se denomina ADPCM [ITU-T 94].

ADPCM emplea predicción lineal (ésto también se utiliza comúnmente en la compresión de imágenes predictiva). Utiliza la muestra anterior (o varias muestras anteriores) para predecir la muestra actual. Luego, calcula la diferencia entre la muestra actual y su predicción, y cuantifica la diferencia. Para cada muestra de entrada X [n], la salida C [n] del codificador es simplemente un cierto número

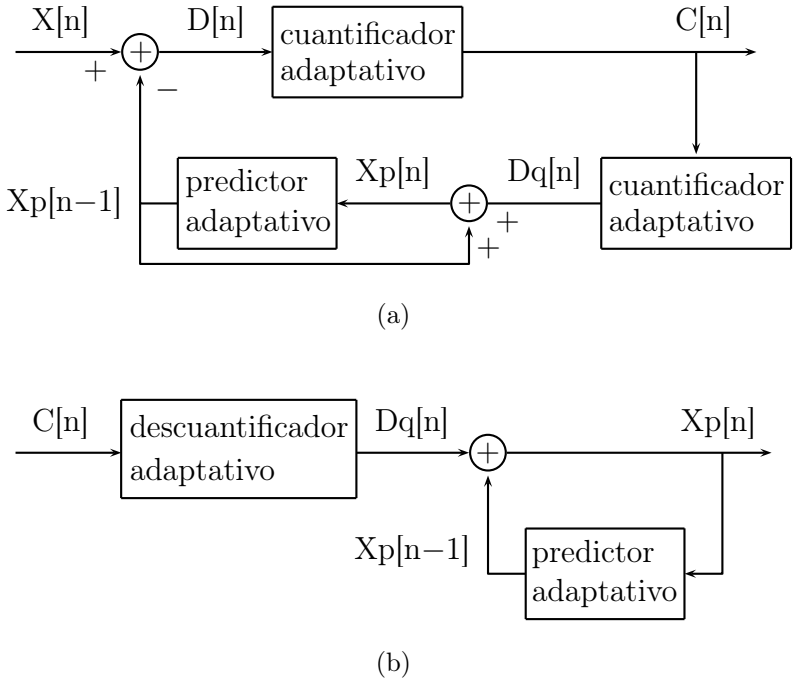


Figura 7.18: (a) Codificador ADPCM y (b) Decodificador.

de niveles de cuantificación. El decodificador multiplica este número por el paso de cuantificación (y puede añadir la mitad del paso de cuantificación, para una mejor precisión) y así obtener la muestra de audio reconstruida. El método es eficaz porque la etapa de cuantificación se actualiza continuamente, tanto en el codificador como en el decodificador, en respuesta a las magnitudes variantes de las muestras de entrada. También es posible modificar adaptativamente el algoritmo de predicción.

Los distintos métodos de ADPCM difieren en la forma en que predicen la muestra de audio actual y en la forma en que se adaptan a la entrada (cambiando el tamaño de paso de la cuantificación y/o el método de predicción).

Además de los valores cuantificados, un codificador ADPCM puede proporcionar al decodificador *información adicional*. Esta información aumenta el tamaño del *stream* comprimido, pero esta degradación es aceptable para los usuarios, porque hace más útil la compresión de datos de audio. Aplicaciones típicas de la información adicional son: (1) ayudar al decodificador a recuperarse de los errores y (2) señalar un punto de entrada en el stream comprimido. Una secuencia de audio original puede ser registrada en forma comprimida en un medio tal como un CD-ROM. Si el usuario (oyente) quiere escuchar la canción 5, el decodificador puede usar la información adicional para encontrar rápidamente el comienzo de la canción.

La Figura 7.18a,b muestra la organización general del codificador y del decodificador ADPCM. Observe que comparten dos unidades funcionales, una característica que ayuda en el software y las implementaciones en hardware. El cuantificador adaptativo recibe la diferencia D [n] entre la muestra de entrada actual X [n] y la predicción Xp [n − 1]. El cuantificador calcula y genera el código cuantificado C [n] de X [n]. El mismo código es enviado al descuantificador adaptativo (el mismo descuantificador usado por el decodificador), que produce la siguiente diferencia descuantificada de valor Dq [n]. Este valor se añade a la salida previa del predictor anterior Xp [n − 1], y la suma Xp [n] es enviada al predictor para ser utilizada en la siguiente etapa.

Una mejor predicción podría obtenerse alimentando la entrada real X [n] del predictor. Sin embargo, el decodificador no sería capaz de imitar ésto, ya que no dispone de X [n]. Vemos que el codificador ADPCM básico es simple, y el decodificador es incluso más sencillo. Introduce un código C [n], lo descuantifica a una diferencia Dq [n], que se añade al predictor anterior de salida Xp [n − 1] para formar la siguiente salida X [n]. La siguiente salida se alimenta también del predictor, para ser utilizada en la próxima etapa.

El resto de esta sección se describe el algoritmo concreto ADPCM adoptado por la Asociación Multimedia Interactiva (IMA o *Interactive Multimedia Association*) [IMA 06]. La IMA es un consorcio de fabricantes de hardware y de software para ordenadores, creada para desarrollar estándares para aplicaciones multimedia. El objetivo de la IMA en el desarrollo de su estándar de compresión de audio era disponer de un método de dominio público tan sencillo y lo suficientemente rápido que un computador personal a 20 MHz de la clase 386 fuera capaz de decodificar en tiempo real, el sonido grabado en estéreo a 44 100 muestras de 16 bits por segundo (ésto son 88 200 muestras de 16 bits por segundo).

El codificador cuantifica cada muestra de audio de 16 bits en un código de 4 bits. El factor de compresión es por tanto una constante: 4.

El "secreto" del algoritmo de la IMA es la simplicidad de su predictor. El valor predicho Xp [n − 1] que emite el predictor es simplemente el valor decodificado Xp [n] de la entrada anterior X [n]. El predictor sólo almacena Xp [n] para un ciclo (un intervalo de la muestra de audio), luego lo emite como Xp [n − 1]. No utiliza ninguno de los valores anteriores Xp [i] para obtener una mejor predicción. Por consiguiente, el predictor no es adaptativo (pero el cuantificador sí lo es). Además, el codificador no genera ninguna información adicional.

La Figura 7.19a es un diagrama de bloques del cuantificador de la IMA. Es a la vez sencillo y adaptativo, variando el tamaño de paso de la cuantificación basándose tanto en el tamaño de paso actual y el previo sacado por el cuantificador. La adaptación se realiza por medio de dos tablas de búsqueda, por lo que es rápido. El cuantificador genera códigos de 4 bits donde el bit de más a la izquierda es un signo y los tres bits restantes son el número de niveles de cuantificación calculados para la muestra de audio actual. Estos tres bits se utilizan como un índice para la primera tabla. El ítem encontrado en esta tabla sirve como un ajuste de índice para la segunda tabla. El ajuste de índice se añade a un índice previamente almacenado, y la suma, después de haber sido revisadas para mantener el rango adecuado, se utiliza como índice para la búsqueda en la segunda tabla. A continuación, la suma es almacenada, y se convierte en el índice almacenado que se utiliza en la siguiente etapa de adaptación. El ítem encontrado en la segunda tabla se convierte en el nuevo tamaño de paso de la cuantificación. La Figura 7.19b ilustra este proceso, y las Tablas 7.21 y 7.22 muestran las dos tablas. La Tabla 7.20 muestra la salida de 4 bits producida por el cuantificador como una función del tamaño de la muestra. Por ejemplo, si la muestra está en el rango $[1{,}5ss, 1{,}75ss)$, donde ss es el tamaño del paso, entonces la salida es 0 | 110.

La Tabla 7.21 ajusta el índice con pasos más grandes cuando la magnitud cuantificada es mayor. La Tabla 7.22 se construye de manera que la relación entre las entradas sucesivas es de aproximadamente 1,1.

> ADPCM: Acrónimo de *Adaptive Differential Pulse Code Modulation* (Modulación de pulsos codificados adaptivo y differential, una forma de modulación de pulsos codificados (PCM) que produce una señal digital con una tasa de bits más baja que el estándar PCM. ADPCM produce una menor tasa de bits al grabar únicamente las diferencias entre las muestras y ajustando la escala de codificación dinámicamente para adaptarse a diferencias grandes y pequeñas.
>
> —De `http://www.webopedia.com/`

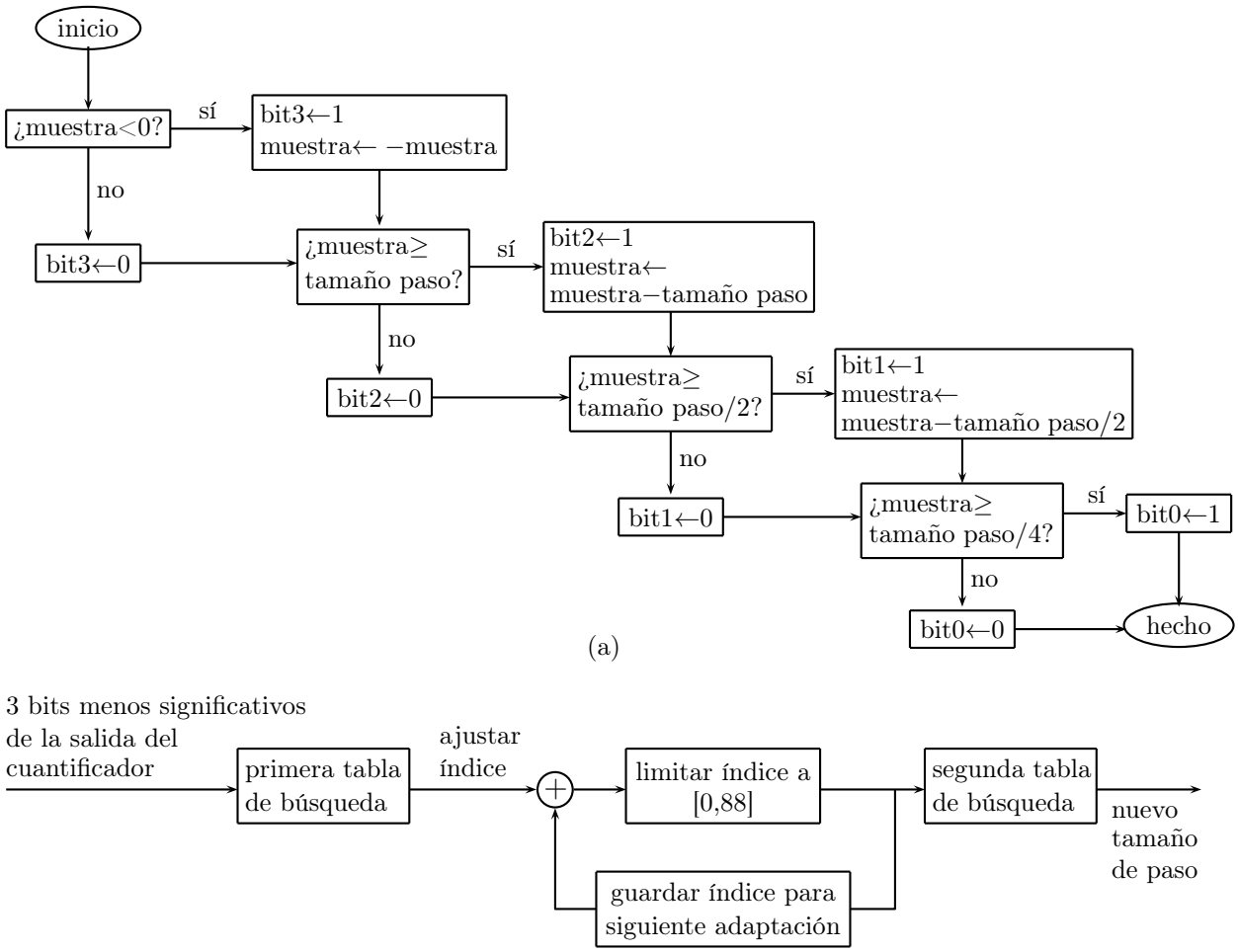


Figura 7.19: (a) Cuantificación ADPCM de IMA. (b) Adaptación del tamaño de paso.

Si la muestra está en el rango	cuantificación de 4 bits	Si la muestra está en el rango	cuantificación de 4 bits
$[1{,}75ss, \infty)$	0 | 111	$[-\infty, -1{,}75ss)$	1 | 111
$[1{,}5ss, 1{,}75ss)$	0 | 110	$[-1{,}75ss, -1{,}5ss)$	1 | 110
$[1{,}25ss, 1{,}5ss)$	0 | 101	$[-1{,}5ss, -1{,}25ss)$	1 | 101
$[1ss, 1{,}25ss)$	0 | 100	$[-1{,}25ss, -1ss)$	1 | 100
$[0{,}75ss, 1ss)$	0 | 011	$[-1ss, -0{,}75ss)$	1 | 011
$[0{,}5ss, 0{,}75ss)$	0 | 010	$[-0{,}75ss, -0{,}5ss)$	1 | 010
$[0{,}25ss, 0{,}5ss)$	0 | 001	$[-0{,}5ss, -0{,}25ss)$	1 | 001
$[0, 0{,}25ss)$	0 | 000	$[-0{,}25ss, 0)$	1 | 000

Tabla 7.20: Tamaño de paso y salidas de 4 bits del cuantificador.

Magnitud de cuantificación de tres bits	Índice de ajuste
000	−1
001	−1
010	−1
011	−1
100	2
101	4
110	6
111	8

Tabla 7.21: Primera tabla para el ADPCM de IMA.

Índice	Tamaño de paso	Índice	Tamaño de paso	Índice	Tamaño de paso	Índice	Tamaño de paso
0	7	22	60	44	494	66	4 026
1	8	23	66	45	544	67	4 428
2	9	24	73	46	598	68	4 871
3	10	25	80	47	658	69	5 358
4	11	26	88	48	724	70	5 894
5	12	27	97	49	796	71	6 484
6	13	28	107	50	876	72	7 132
7	14	29	118	51	963	73	7 845
8	16	30	130	52	1 060	74	8 630
9	17	31	143	53	1 166	75	9 493
10	19	32	157	54	1 282	76	10 442
11	21	33	173	55	1 411	77	11 487
12	23	34	190	56	1 552	78	12 635
13	25	35	209	57	1 707	79	13 899
14	28	36	230	58	1 878	80	15 289
15	31	37	253	59	2 066	81	16 818
16	34	38	279	60	2 272	82	18 500
17	37	39	307	61	2 499	83	20 350
18	41	40	337	62	2 749	84	22 358
19	45	41	371	63	3 024	85	24 633
20	50	42	408	64	3 327	86	27 086
21	55	43	449	65	3 660	87	29 794
						88	32 767

Tabla 7.22: Segunda tabla para el ADPCM de IMA.

7.7. Audio MLP

Nota. El método de compresión de audio MLP descrito en esta sección es diferente de y no está relacionado con el método de compresión de imágenes MLP (multinivel progresivo) de la Sección 4.21. Los acrónimos idénticos son una desafortunada coincidencia.

> La ambigüedad se refiere a la propiedad de palabras, términos y conceptos, (dentro de un contexto determinado) por ser indefinibles, indefinidos, o si no vagos, y tener así un significado poco claro. Una palabra, frase, oración, u otra comunicación se llama "ambigua" si puede ser interpretada de más de una manera.
>
> —De `http://en.wikipedia.org/wiki/Disambiguation`

Meridian [Meridian 03] es una empresa británica especializada en productos de alta calidad de audio, tales como reproductores CD y DVD, altavoces, sintonizadores de radio y amplificadores estéreo de sonido envolvente. La buena calidad del sonido digital, tal como el que se encuentra en un CD de audio, emplea dos canales (sonido estéreo), cada uno muestreado a 44,1 kHz con muestras de 16 bits (Sección 7.2). Un sonido digitalizado típico de alta calidad, por otro lado, puede utilizar seis canales (i.e., el sonido grabado originalmente por seis micrófonos, para sonido envolvente), muestreados con la elevada tasa de 96 kHz (para asegurar que todos los matices del audio puedan ser recogidos), con muestras de 24 bits (para obtener el rango dinámico más alto posible). Este tipo de datos de audio es representado a $6 \times 96\,000 \times 24 = 13{,}824$ Mbps (megabits, no megabytes). En contraste, un DVD (disco versátil digital) tiene 4,7 Gbytes, lo que a 13,824 Mbps sin compresión se traduce en tan sólo 45 minutos de reproducción. (Recuerde que incluso los CDs, que tienen una capacidad mucho menor, disponen de 74 minutos de tiempo de reproducción. Ésto es un estándar de la industria.) Además, la tasa de transferencia de datos máxima para el DVD-A (audio) es de 9,6 Mbps, muy inferior a 13,824 Mbps. Es obvio que la compresión es la clave para lograr un formato DVD-A práctico, pero la alta calidad (en lugar de sólo buena calidad) requiere compresión sin pérdidas.

El algoritmo que ha sido seleccionado como el estándar de compresión para el DVD-A (audio) es MLP (Meridian Lossless Packing o empaquetamiento sin pérdidas de Meridian). Este algoritmo está patentado, y algunos de sus detalles todavía se mantienen en propiedad, lo cual se refleja en la información proporcionada en esta sección. El término "empaquetado" tiene un doble significado. Se refiere a (1) la eliminación de la redundancia a partir de los datos originales con el fin de "empaquetarlo" más densamente, y (2) las muestras de audio son codificadas en paquetes.

MLP opera reduciendo o eliminando completamente las redundancias en el audio digitalizado, sin ninguna cuantificación u otra pérdida de datos. Observe que los formatos de audio de alta calidad como el de 96 kHz con muestras de 24 bits llevan más información que la estrictamente necesaria para el oyente humano (o más de lo disponible para los micrófonos y las técnicas de conversión modernos). En consecuencia, estos formatos de audio contienen mucha redundancia y pueden ser comprimidos eficientemente. MLP puede manejar hasta 63 canales de audio y soporta frecuencias de muestreo de hasta 192 kHz.

Las principales características del MLP son las siguientes:

1. Al menos 4 bits/muestra de compresión tanto para las tasas de datos promedio y máximos (de pico).
2. Fácil transformación entre los flujos de datos de tasa fija y de tasa variable.
3. El manejo cuidadoso y económico de tasas de muestras de entrada mezcladas.
4. Simple, de rápida decodificación.

5. Es en cascada. Una secuencia de audio puede ser codificada y decodificada varias veces en sucesión, y la salida siempre será una copia exacta del original. Con MLP, no hay pérdidas de generación.

El término "tasa (o velocidad) de datos variable" es importante. Una cadena de datos sin comprimir está formada por muestras de audio, y cada segundo de sonido requiere el mismo número de muestras. Tal secuencia tiene una tasa de datos fija. En contraste, una cadena comprimida generada por un codificador de audio sin pérdidas tiene una tasa de datos variable; cada segundo de sonido ocupa un número diferente de bits de esta cadena, dependiendo de la naturaleza del sonido. Un segundo de silencio ocupa muy pocos bits, mientras que un segundo de sonido al azar no se comprimirá y requerirá el mismo número de bits en la cadena comprimida que en el archivo original. La mayoría de los métodos de compresión de audio sin pérdidas están diseñados para reducir la tasa de datos media, pero MLP tiene la importante característica de que reduce el pico instantáneo de tasa de datos en una cantidad conocida. Esta característica hace posible la grabación de 74 min de cualquier tipo de sonido no aleatorio en un DVD-A de 4,7 Gbytes.

Además de ser sin pérdidas (lo que significa que los datos originales se entregan bit a bit en la reproducción), MLP es también robusto. No incluye ningún código de corrección de errores pero tiene características de protección de errores. Utiliza bits de comprobación para asegurarse de que cada paquete descomprimido por el decodificador es idéntico al comprimido por el codificador. El flujo de datos comprimidos contiene puntos de reinicio, colocados a intervalos de 10−30 ms. Cuando el decodificador detecta un error, simplemente salta al siguiente punto de reinicio, con una mínima pérdida de sonido. Éste es otro significado del término "sonido de alta calidad". Por primera vez, un oyente escucha exactamente lo que el compositor/intérprete pretende, bit por bit y nota por nota.

Con la compresión de audio con pérdidas, la cantidad de compresión se mide mediante el número de bits por segundo de sonido en el *stream* comprimido, independientemente del tamaño de la muestra de audio. Con la compresión sin pérdidas, un tamaño de muestra grande (que en realidad significa más bits menos significativos —o LSB—), debe ser comprimida sin pérdidas, por lo que aumenta el tamaño del *stream* comprimido, pero los LSBs adicionales suelen tener poca redundancia y, por tanto, son más difíciles de comprimir. Por esta razón la compresión de audio sin pérdidas debería medirse mediante el número de bits guardados en cada muestra de audio —una medida relativa de la compresión—.

MLP reduce las muestras de audio desde su tamaño original (típicamente 24 bits) dependiendo de la tasa de muestreo. Para velocidades de datos promedio, la reducción es la siguiente: Para tasas de muestreo de 44,1 kHz y 48 kHz, una muestra se reduce hasta 5 a 11 bits. A 88,2 kHz y 96 kHz, la reducción aumenta hasta 9 a 13 bits. A 192 kHz, MLP a veces puede reducir una muestra a 14 bits. Aún más importante es el ahorro para las tasas de datos máximas: muestras de 4 bits para 44,1 kHz, 8 bits para 96 kHz y 9 bits para 192 kHz. Estos ahorros en los picos de las tasas de datos equivalen a una garantía virtual, y son una de las razones principales para la adopción de MLP como el estándar de compresión de un DVD-A.

El resto de esta sección describe algunos de los detalles del MLP. Se basa en [Stuart et al. 99]. Las técnicas utilizadas por MLP para comprimir las muestras de audio incluyen:

1. Busca bloques consecutivos de muestras de audio que sean pequeños, i.e., que tengan varios bits más significativos a 0. Ésto se conoce como "aire viciado (*dead air*)". Los LSBs de dichas muestras son eliminados, lo cual es equivalente a un desplazamiento a la derecha (a nivel de bits) de las muestras .

2. Identifica los canales que no utilizan su ancho de banda completamente.

3. Identifica y elimina la correlación entre los canales.

4. Elimina la correlación entre muestras consecutivas de audio en cada canal.

5. Utiliza búferes para suavizar la tasa de producción de la salida comprimida.

Las etapas principales de la compresión son: (1) procesamiento sin pérdidas, (2) construcción de matrices (*matrixing*) sin pérdidas, (3) filtrado IIR sin pérdidas, (4) codificación de entropía, y (5) salida a través de un buffer FIFO. El procesamiento sin pérdidas se refiere a la identificación de bloques de muestras de audio con capacidad no utilizada. Tales muestras son desplazadas. El término "matrixing" se refiere a la reducción de las correlaciones entre canales por medio de una matriz de transformación afín. El filtro IIR descorrelaciona las muestras de audio en cada canal (descorrelación intracanal) prediciendo la siguiente muestra a partir de sus predecesoras y produciendo una predicción diferencia. Las diferencias se comprimen posteriormente mediante un codificador de entropía. El resultado es un *stream* comprimido para cada canal, y esos *streams* son introducidos en un buffer FIFO para suavizar la velocidad de producción de los datos codificados. Finalmente, la salida del buffer se divide en paquetes, y se añaden los bits de comprobación de errores y los puntos de reinicio a cada paquete para la protección contra errores.

Un archivo de audio con varios canales se obtiene normalmente mediante la grabación del sonido con varios micrófonos colocados en lugares diferentes. Cada micrófono emite una señal analógica que es digitalizada y se convierte en un canal de audio. En consecuencia, las muestras de audio en los diversos canales están correlacionados, y la reducción o eliminación de esta correlación es una etapa importante, denominada "matrixing", en la compresión MLP. Denotando la muestra de audio i del canal j por a_{ij}, la conversión a matrices se efectúa restando de a_{ij} una combinación lineal de las muestras de audio a_{ik} de todos los demás canales k. Por consiguiente,

$$a_{ij} \leftarrow a_{ij} - \sum_{k \neq j} a_{ik} m_{kj},$$

donde la columna j de la matriz m corresponde al canal de audio j. Esta combinación lineal También se puede ver como una suma ponderada donde se multiplica cada muestra de audio a_{ik} por un peso m_{kj}. También se puede interpretar como una transformación afín del vector de muestras de audio $(a_{i,1}, a_{i,2}, \ldots, a_{i,j-1}, a_{i,j+1}, \ldots, a_{i,n})$.

La etapa siguiente de la codificación MLP es la predicción. La etapa *matrixing* puede quitar toda o parte de la correlación entre los canales de audio, pero las muestras individuales en cada canal todavía están correlacionadas; una muestra tiende a ser similar a sus vecinos más cercanos, y las muestras en un canal de audio forman lo que se llama *componentes cuasi–periódicos*. MLP utiliza la predicción lineal donde un filtro especial IIR es aplicado a los datos para eliminar la correlación o una porción tímbrica de la señal. El filtro deja una señal residual de baja amplitud que es aperiódica y se asemeja a ruido. Esta señal residual representa tales componentes de sonido como transitorios de audio, relaciones de fase entre armónicos parciales e incluso resina extra en las cuerdas (no es broma).

La etapa de codificación de entropía continúa la creación de matrices (eliminando las correlaciones entre los canales) y la predicción (eliminando las correlaciones internas en los canales). Este paso se deshace de cualquier correlación que quede entre las muestras de audio original. En el codificador MLP están integrados varios codificadores de entropía, y se puede seleccionar cualquiera de ellos para codificar cualquier canal de audio. Generalmente, MLP asume que las señales de audio tienen una distribución Laplaciana cercana a cero, y sus codificadores de entropía asignan códigos de tamaño variable para las muestras de audio basadas en esta suposición (recordemos que el método de compresión de imágenes MLP de la Sección 4.21 también utiliza esta distribución, que se muestra gráficamente en la Figura 4.128b).

El uso de búferes es la siguiente etapa en la codificación MLP. Suaviza las variaciones en la tasa de bits causada por variaciones en el sonido que está siendo comprimido. Pasajes de sonido uniforme o incluso el silencio puede alternar con sonidos complejos y de semejanza aleatoria. Cuando comprime tales sonidos, el codificador MLP puede emitir unos pocos bits para cada segundo de los pasajes uniformes (que se comprimen bien) seguidos por muchos bits para cada segundo de las partes del

sonido que no se comprimen bien. El uso de un buffer asegura que el codificador saque los datos al *stream* comprimido a una velocidad constante. Ésto puede no ser muy importante cuando el stream comprimido es un archivo, pero puede haber una gran diferencia cuando el audio comprimido es transmitido y se supone que debe ser recibido por un decodificador MLP y reproducido en tiempo real y sin pausas.

Además de los datos comprimidos, el stream comprimido incluye instrucciones para el decodificador y los bits de comprobación de errores CRC. El archivo de audio de entrada se divide en bloques que son típicamente de 40−160 muestras de longitud cada uno. Los bloques están ensamblados en paquetes cuya longitud varía entre 640 y 2 560 muestras y es controlado por el usuario. Cada paquete es comprobado y contiene información de reinicio. Si el decodificador detecta datos erróneos y no puede continuar la descompresión de un paquete, simplemente salta al comienzo del siguiente paquete, con una pérdida típica de sólo 7 ms de tiempo de reproducción —apenas imperceptible para el oyente—.

MLP también tiene opciones con pérdidas. Una de ellas consiste en reducir todas las muestras de audio de 24 a 22 bits. Otra es pasar algunos de los canales de audio a través de un filtro de paso bajo. Ésto tiende a reducir la entropía de la señal de audio, por lo que es posible para el codificador MLP producir una mejor compresión. Tales opciones pueden ser importantes en casos donde deben grabarse más de 74 min de audio en un DVD-A.

7.8. Compresión del habla

Algunos codificadores (*codecs*) de audio están diseñados específicamente para comprimir señales de voz. Tales señales son de audio y se muestrean como cualesquier otros datos de audio, pero debido a la naturaleza del habla humana, tienen propiedades que pueden ser aprovechadas para conseguir una compresión eficiente. [Jayant 97] cubre varios tipos de codificadores de voz, sin embargo está disponible más información sobre este tema en la World Wide Web en los sitios de los investigadores. Esta sección comienza con una discusión de las propiedades del lenguaje humano, y continúa con una descripción de diversos codificadores de voz.

7.8.1. Propiedades de la voz

Producimos sonido al forzar la salida del aire de los pulmones a través de las cuerdas vocales situadas en el tracto vocal (Figura 7.23). El aire termina escapando a través de la boca, pero es el sonido generado en el tracto vocal (que se extiende desde las cuerdas vocales a la boca, y en un adulto promedio es 17 cm de largo) mediante vibraciones, de la misma manera como el aire que pasa a través de una flauta genera sonido. El tono del sonido se controla variando la forma del tracto vocal (principalmente moviendo la lengua) y mediante el movimiento de los labios. La intensidad (volumen) se controla variando la cantidad de aire enviado desde los pulmones. Los seres humanos son mucho más lentos que los ordenadores u otros dispositivos electrónicos, y ésto también es cierto con respecto al habla. Los pulmones funcionan lentamente, y el tracto vocal cambia de forma lentamente, por lo que el tono y el volumen de la voz varían lentamente. Cuando el habla es capturada por un micrófono y se muestrea, se encuentra que las muestras adyacentes son similares, e incluso muestras separadas por 20 ms están fuertemente correlacionadas. Esta correlación es la base de la compresión de la voz.

Las cuerdas vocales se pueden abrir y cerrar, y la abertura entre ellos se denomina glotis. Los movimientos del glotis y del tracto vocal dan lugar a diferentes tipos de sonido. Los tres tipos principales son los siguientes:

1. Sonidos de voz. Éstos son los sonidos que hacemos cuando hablamos. Las cuerdas vocales vibran, lo que abre y cierra la glotis, de este modo se envían pulsos de aire a diferentes presiones al tracto, donde se convierten en ondas de sonido. Variando la forma de la cuerdas vocales y su tensión se cambia la frecuencia de vibración de la glotis y, por lo tanto, se controla el tono del sonido.

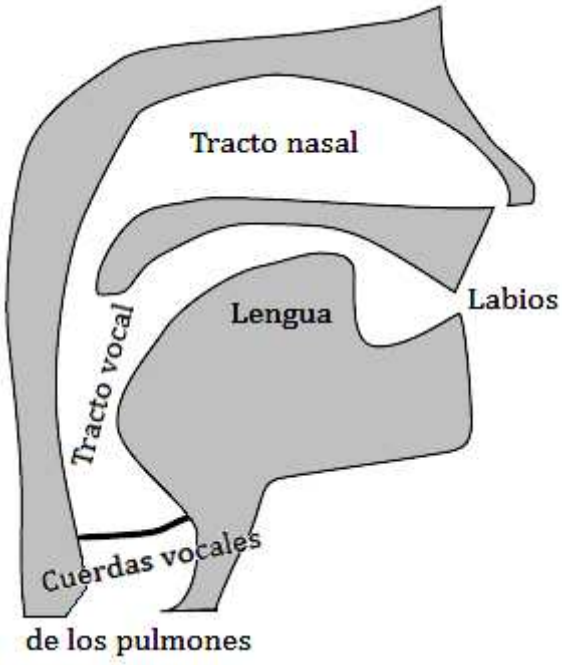


Figura 7.23: Un corte transversal de la cabeza humana.

Recordemos que el oído es sensible a frecuencias de sonido desde 16 Hz hasta aproximadamente $20\,000 - 22\,000$ Hz. Las frecuencias de la voz humana, por otro lado, son mucho más restringidas y están generalmente en el rango de 500 Hz a aproximadamente 2 kHz. Ésto equivale a períodos de tiempo de 2 ms a 20 ms, y para un ordenador, tales períodos son muy largos. En consecuencia, los sonidos de voz tienen una periodicidad de larga duración, y ésta es la clave para una buena compresión del habla. La Figura 7.24a es un ejemplo típico de formas de onda de sonido de voz.

2. Sonidos sin voz. Éstos son sonidos que se emiten y se pueden oir, pero que no forman parte del habla. Tales sonidos son el resultado de mantener la glotis abierta y forzar el paso del aire a través de una constricción en el tracto vocal. Cuando un sonido sin voz es muestreado, las muestras presentan poca correlación y son aleatorias o casi aleatorias. La Figura 7.24b es un ejemplo típico de las formas de onda de sonido sin voz.

3. Sonidos oclusivos. Éstos se producen cuando la glotis se cierra, los pulmones aplican presión de aire en ella, y ésta se abre de repente, dejando escapar el aire de golpe. El resultado es un sonido oclusivo.

Existen también combinaciones de los tres tipos. Un ejemplo es el caso cuando las cuerdas vocales vibran y también hay una constricción en el tracto vocal. Tal sonido se llama fricativo.

Codificadores de voz. Hay tres tipos principales de codificadores de voz. Los *codificadores* de voz *de forma de onda* producen una voz de buena a excelente después de comprimir y descomprimir ésta, pero genera tasas de bits de $10-64$ kbps. Los *codificadores de fuente* (también llamados *codificadores de voz* o *vocoders*) generalmente producen una voz entre pobre y razonable, pero puede comprimirse a tasas de bits muy bajas (por debajo de 2 kbps). Los *codificadores híbridos* son combinaciones de los dos primeros tipos y producen una voz que varía entre razonable y buena, con tasas de bits entre 2 y 16 kbps. La Figura 7.25 ilustra la calidad de la voz versus la tasa de bits de estos tres tipos.

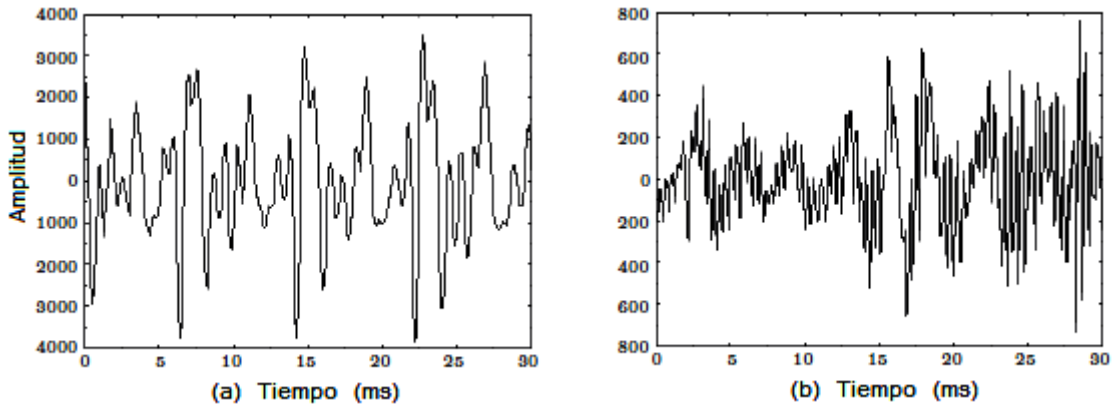


Figura 7.24: Ondas sonoras: (a) Con voz y (b) Sin voz.

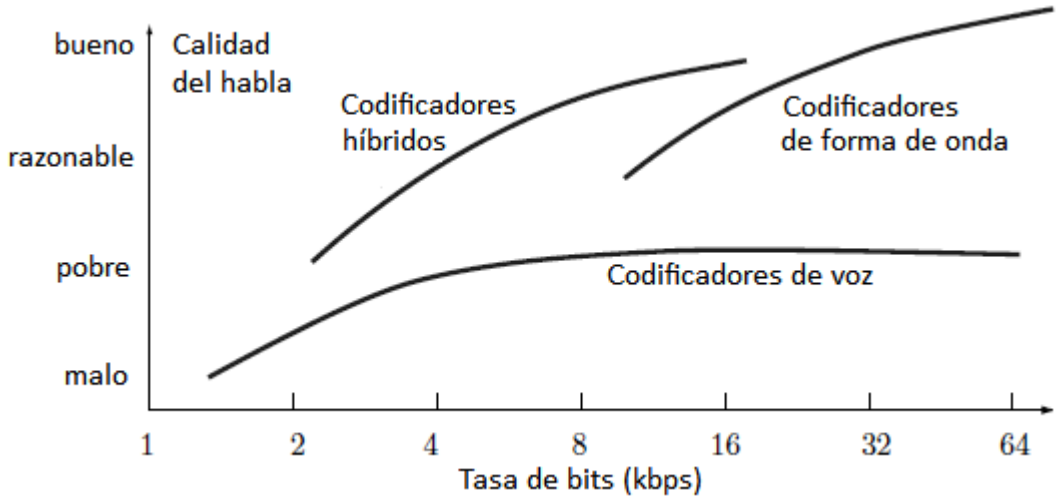


Figura 7.25: Calidad de voz versus tasa de bits para codificadores de voz.

7.8.2. Codificadores de forma de onda

Este tipo de codec no intenta predecir cómo se generó el sonido original. Sólo trata de producir, tras la descompresión, muestras de audio que estén tan cercanas a las originales como sea posible. Por consiguiente, estos codificadores no están diseñados específicamente para la codificación de voz y pueden funcionar igual de bien con todos los tipos de datos de audio. Como ilustra la Figura 7.25, cuando tal codificador es forzado a comprimir el sonido a menos de 16 kbps, la calidad del sonido reconstruido disminuye significativamente.

El codificador de forma de onda más simple es el modulador de pulsos codificados (PCM). Este codificador simplemente cuantifica cada muestra de audio. El habla es normalmente muestreada a sólo 8 kHz. Si cada muestra se cuantifica en 12 bits, la tasa de bits resultante es $8\text{k} \times 12 = 96$ kbps y la reproducción de los sonidos del habla es casi natural. Se obtienen mejores resultados con un cuantificador logarítmico, como los métodos de compansión μ-law y A-Law (Sección 7.5). Ellos cuantifican muestras de audio en un número variable de bits y pueden comprimir la voz a 8 bits por muestra en promedio, lo que resulta en una tasa de bits de 64 kbps, con una muy buena calidad de la voz reconstruida.

Un codificador de voz PCM diferencial (DPCM; Sección 7.6) utiliza el hecho de que las muestras de audio del habla con voz están correlacionadas. Este tipo de codificador calcula la diferencia entre la muestra actual y su predecesora y cuantifica la diferencia. Una versión adaptativa (ADPCM, Sección 7.6) puede comprimir el habla en buena calidad a una tasa de bits de 32 kbps.

Los codificadores de forma de onda también pueden operar en el dominio de la frecuencia. El algoritmo de codificación de subbanda (SBC) transforma las muestras de audio en el dominio de la frecuencia, particiona los coeficientes resultantes en varias bandas críticas (o subbandas de frecuencia), y codifica cada subbanda por separado con ADPCM o un método de cuantificación similar. El decodificador SBC decodifica los coeficientes de la frecuencia, los recombina, y efectúa la transformación inversa (con pérdidas) a la reconstrucción de las muestras de audio. La ventaja de SBC es que el oído es más sensible a ciertas frecuencias y menos sensible a otras (Sección 7.3, especialmente la Tabla 7.6). Las subbandas de frecuencias a las que el oído es menos sensible pueden ser, por lo tanto, toscamente cuantificadas sin pérdida de calidad de sonido. Este tipo de codificador produce típicamente una buena calidad de voz reconstruida a una tasa de bits de 16−32 kbps. Son, sin embargo, más complejos de implementar que los codificadores PCM y también pueden ser más lentos.

El algoritmo de compresión de voz por codificación adaptativa de transformadas (*Adaptive Transform Coding* o ATC) transforma las muestras de audio en el dominio de la frecuencia con la transformada discreta del coseno (DCT). El archivo de audio es dividido en bloques de muestras de audio y aplicando la DCT a cada bloque, se produce un número de coeficientes de frecuencia. Cada coeficiente es cuantificado de acuerdo con la frecuencia a la que corresponde. El habla puede ser reconstruida con buena calidad, alcanzando tasas de bits tan bajas como 16 kbps.

7.8.3. Codificadores de fuente

En general, un codificador de fuente utiliza un modelo matemático de la fuente de datos. El modelo depende de ciertos parámetros, y el codificador utiliza los datos de entrada para calcular esos parámetros. Una vez obtenidos los parámetros, son escritos (después de ser adecuadamente codificados) en el *stream* comprimido. El decodificador introduce los parámetros y emplea el modelo matemático para reconstruir los datos originales. Si los datos originales son de audio, el codificador de fuente se llama *vocoder* (de "vocal coder").[7] El vocoder descrito en esta sección es el codificador predictivo lineal (LPC, véase [Rabiner y Schafer 78]). La Figura 7.26 muestra un modelo simplificado de la producción del habla. La parte (a) ilustra el proceso en una persona, mientras que la parte (b) muestra el modelo matemático LPC correspondiente.

[7] En español, "codificador vocal" o "codificador de voz"

En este modelo, la salida es la secuencia de muestras de voz $s(n)$ que salen de el filtro LPC (que se corresponde con el tracto vocal y los labios). La entrada $u(n)$ al modelo (y al filtro) es ya sea un tren de pulsos (cuando el sonido es con voz) o ruido blanco (cuando el sonido es sin voz). Las cantidades $u(n)$ también se denominan *innovación*. El modelo ilustra cómo pueden ser generadas muestras $s(n)$ de voz mediante la mezcla de innovaciones (un tren de pulsos y ruido blanco). En consecuencia, representa matemáticamente la relación entre las muestras de voz y las innovaciones. La tarea del codificador de voz consiste en introducir muestras $s(n)$ de voz real, utilizar el filtro como una función matemática para determinar una secuencia equivalente de innovaciones $u(n)$, y emitir las innovaciones en formato comprimido. La correspondencia entre los parámetros del modelo y las partes de voz real es la siguiente:

1. El Parámetro V (con voz) corresponde a las vibraciones de las cuerdas vocales. UV expresa los sonidos sin voz.
2. T es el período de las vibraciones de las cuerdas vocales.
3. G (ganancia) corresponde a la intensidad o el volumen de aire enviado desde cada uno de los pulmones por segundo.
4. Las innovaciones $u(n)$ corresponden al aire que pasa a través del tracto vocal.
5. Los símbolos $\otimes$ y $\oplus$ denotan la amplificación y la combinación, respectivamente.

La ecuación principal del modelo LPC describe la salida del filtro LPC como:

$$H(z) = \frac{1}{1 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_{10} z^{-10}},$$

donde z es la entrada al filtro [el valor de uno de los $u(n)$]. Una ecuación equivalente describe la relación entre las innovaciones $u(n)$ por un lado, y los 10 coeficientes a_i y el audio de muestras $s(n)$ de voz por otro lado. La relación es:

$$u(n) = s(n) + \sum_{i=1}^{10} a_i s(n-i). \tag{7.4}$$

Esta relación implica que cada número $u(n)$ de entrada al filtro LPC es la suma de la muestra $s(n)$ de audio actual y una suma ponderada de las 10 muestras precedentes.

El modelo LPC se puede escribir como la 13–tupla:

$$\mathbf{A} = (a_1, a_2, \ldots, a_{10}, G, V/UV, T),$$

donde V/UV es un único bit que especifica la fuente (con voz o sin voz) de las muestras de entrada. El modelo asume que $\mathbf{A}$ permanece estable durante aproximadamente 20 ms, después se actualiza con las muestras de audio de los siguientes 20 ms. A una tasa de muestreo de 8 kHz, se producen 160 muestras de audio $s(n)$ cada 20 ms. El modelo calcula las 13 cantidades en $\mathbf{A}$ partiendo de estas 160 muestras, escribe $\mathbf{A}$ (como 13 números) en el *stream* comprimido, y luego repite lo mismo para los próximos 20 ms. El factor de compresión resultante es, por tanto, de 13 números para cada conjunto de 160 muestras de audio.

Es importante distinguir el funcionamiento del codificador a partir del diagrama del modelo matemático del LPC representado en la Figura 7.26b. La figura muestra cómo una secuencia de innovaciones $u(n)$ genera muestras de voz $s(n)$. El codificador, sin embargo, comienza con las muestras de voz. Introduce una secuencia de 20 ms de muestras de voz $s(n)$, calcula una secuencia equivalente de innovaciones, las comprime en 13 números, y envía los números después de codificarlos. Ésto se repite cada 20 ms.

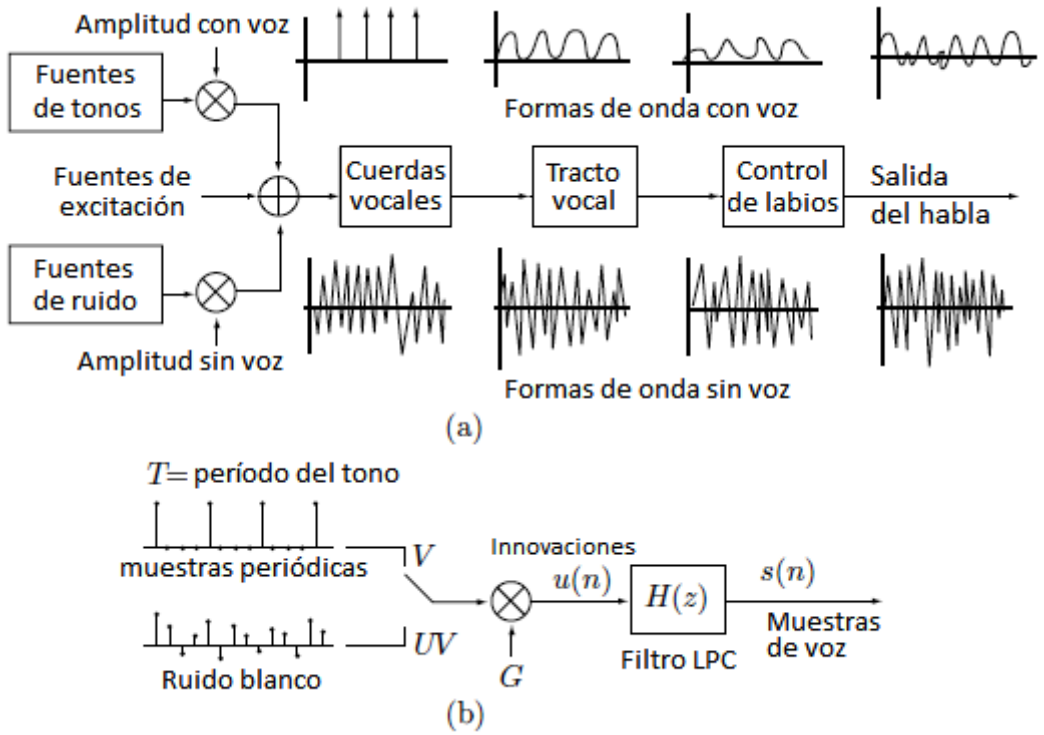


Figura 7.26: Producción del habla: (a) Real. (b) Modelo LPC.

La codificación LPC (o análisis) comienza con 160 muestras de sonido y calcula los 10 parámetros a_i del LPC minimizando la energía de la innovación $u\,(n)$. La energía es la función:

$$f\,(a_1, a_2, \ldots, a_{10}) = \sum_{i=0}^{159} u^2\,(n)$$

y su mínimo se calcula diferenciandolo 10 veces, con respecto a cada uno de sus 10 parámetros de configuración y estableciendo las derivadas a cero. Las 10 ecuaciones

$$\frac{\partial f}{\partial a_i} = 0, \quad \text{para } i = 1, 2, \ldots, 10$$

pueden escribirse en notación compacta:

$$\begin{bmatrix} R\,(0) & R\,(1) & R\,(2) & R\,(3) & R\,(4) & R\,(5) & R\,(6) & R\,(7) & R\,(8) & R\,(9) \\ R\,(1) & R\,(0) & R\,(1) & R\,(2) & R\,(3) & R\,(4) & R\,(5) & R\,(6) & R\,(7) & R\,(8) \\ R\,(2) & R\,(1) & R\,(0) & R\,(1) & R\,(2) & R\,(3) & R\,(4) & R\,(5) & R\,(6) & R\,(7) \\ R\,(3) & R\,(2) & R\,(1) & R\,(0) & R\,(1) & R\,(2) & R\,(3) & R\,(4) & R\,(5) & R\,(6) \\ R\,(4) & R\,(3) & R\,(2) & R\,(1) & R\,(0) & R\,(1) & R\,(2) & R\,(3) & R\,(4) & R\,(5) \\ R\,(5) & R\,(4) & R\,(3) & R\,(2) & R\,(1) & R\,(0) & R\,(1) & R\,(2) & R\,(3) & R\,(4) \\ R\,(6) & R\,(5) & R\,(4) & R\,(3) & R\,(2) & R\,(1) & R\,(0) & R\,(1) & R\,(2) & R\,(3) \\ R\,(7) & R\,(6) & R\,(5) & R\,(4) & R\,(3) & R\,(2) & R\,(1) & R\,(0) & R\,(1) & R\,(2) \\ R\,(8) & R\,(7) & R\,(6) & R\,(5) & R\,(4) & R\,(3) & R\,(2) & R\,(1) & R\,(0) & R\,(1) \\ R\,(9) & R\,(8) & R\,(7) & R\,(6) & R\,(5) & R\,(4) & R\,(3) & R\,(2) & R\,(1) & R\,(0) \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \\ a_8 \\ a_9 \\ a_{10} \end{bmatrix} = \begin{bmatrix} -R\,(1) \\ -R\,(2) \\ -R\,(3) \\ -R\,(4) \\ -R\,(5) \\ -R\,(6) \\ -R\,(7) \\ -R\,(8) \\ -R\,(9) \\ -R\,(10) \end{bmatrix},$$

donde $R(k)$, que se define como:

$$R(k) = \sum_{n=0}^{159-k} s(n)\, s(n+k),$$

es la función de autocorrelación de las muestras $s(n)$. Este sistema de 10 ecuaciones algebraicas es fácil de resolver numéricamente, y las soluciones son los 10 parámetros a_i del LPC. Los tres parámetros restantes, V/UV, G, y T, se determinan a partir de las 160 muestras de audio. Si esas muestras presentan periodicidad, entonces T se convierte en ese período y parámetro de 1 bit V/UV se fija en V. Si las 160 muestras no se caracterizan por ningún período bien definido, entonces T permanece indefinido y V/UV se fija en UV. El valor de G queda determinado por la muestra más grande.

La decodificación (o síntesis) LPC comienza con un conjunto de 13 parámetros del LPC y calcula 160 muestras de audio como salida del filtro LPC mediante:

$$H(z) = \frac{1}{1 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_{10} z^{-10}}.$$

Estas muestras son reproducidas a 8 000 muestras por segundo y producen 20 ms de habla (con voz o sin voz) reconstruido.

La versión de 2,4 kbps del vocoder LPC efectúa un paso más para codificar los 10 parámetros a_i. Éste los convierte en 10 parámetros de pares de líneas espectrales (*Line Spectrum pair* o LSP), denotados por ω_i, mediante las relaciones:

$$\begin{aligned} P(z) =& 1 + (a_1 - a_{10})\, z^{-1} + (a_2 - a_9)\, z^{-2} + \cdots + (a_{10} - a_1)\, z^{-10} - z^{-11} \\ =& \left(1 - z^{-1}\right) \prod_{k=2,4,\ldots,10} \left(1 - 2\cos\omega_k z^{-1} + z^{-2}\right), \\ Q(z) =& 1 + (a_1 + a_{10})\, z^{-1} + (a_2 + a_9)\, z^{-2} + \cdots + (a_{10} + a_1)\, z^{-10} + z^{-11} \\ =& \left(1 + z^{-1}\right) \prod_{k=1,3,\ldots,9} \left(1 - 2\cos\omega_k z^{-1} + z^{-2}\right). \end{aligned}$$

Los diez parámetros LSP satisfacen $0 < \omega_1 < \omega_2 < \cdots < \omega_{10} < \pi$. Cada uno se cuantifica en 3 tres o cuatro bits, como se muestra aquí, para formar un total de 34 bits.

ω_1	ω_2	ω_3	ω_4	ω_5	ω_6	ω_7	ω_8	ω_9	ω_{10}
3	4	4	4	4	3	3	3	3	3

El parámetro de ganancia G se codifica en siete bits, el período T se cuantifica en seis bits, y el parámetro V/UV requiere únicamente un bit. Por consiguiente, los 13 parámetros del LPC se cuantifican en 48 bits y proporcionan datos suficientes para una trama (*frame* o cuadro) de 20 ms de voz descomprimida. Cada segundo de voz consta de 50 frames, por lo que la tasa de bits es de $50 \times 48 = 2{,}4$ kbps. Suponiendo que las muestras originales de voz están formadas por ocho bits cada una, el factor de compresión es: $(8\,000 \times 8)/2\,400 = 26{,}67$ —muy impresionante—.

7.8.4. Codificadores híbridos

Este tipo de codificador de voz combina características tanto de los codificadores de forma de onda como los de código fuente. Los codificadores híbridos más populares son algoritmos de análisis por síntesis (Analysis-by-Synthesis o AbS) en el dominio del tiempo. Al igual que el vocoder LPC, estos codificadores modelan el tracto vocal mediante un filtro de predicción lineal, pero utilizan una señal de excitación en lugar del sencillo modelo de dos estados con voz–sin voz para suministrar la entrada $u(n)$

(innovación) al filtro. Por consiguiente, un codificador AbS comienza con un conjunto de muestras de voz (un frame), las codifica similarmente a LPC, las decodifica, y resta las muestras decodificadas de los originales. Las diferencias se envían a través de un proceso de minimización de errores que genera muestras codificadas mejoradas. Estas muestras son de nuevo descodificadas, restadas de las muestras originales y se calculan las nuevas diferencias. Ésto se repite hasta que las diferencias satisfagan una condición de terminación. El codificador entonces procede al siguiente conjunto de muestras de voz (siguiente frame).

Uno de los codificadores AbS más conocidos es CELP, un acrónimo que procede de *code-excited linear predictive* (predicción lineal excitada por código). El codificador CELP de 4.8 kbps es similar al vocoder LPC, con las siguientes diferencias:

1. El tamaño del frame es de 30 ms (i.e., 240 muestras de voz por frame y 33,3 frames por segundo de voz).
2. La innovación $u(n)$ se codifica directamente.
3. En el algoritmo está incluida una predicción de tono.
4. Se utiliza la cuantificación de vectores.

Cada frame se codifica en 144 bits de la siguiente manera: Los parámetros de LSP se cuantifican hasta un total de 34 bits. El filtro de predicción de tono se convierte en un número de 48 bits. Los índices del libro de códigos consumen 36 bits. Cuatro parámetros de ganancia requieren 5 bits cada uno. Un bit es usado para la sincronización, cuatro bits codifican el control progresivo de errores (forward error control o FEC, similar a CRC), y un bit está reservado para su uso futuro. A 33,3 frames por segundo, la tasa de bits es, por lo tanto, de $33{,}3 \times 144 = 4\,795 \approx 4{,}8$ kbps.

También existe un CELP algebraico de estructura conjugada (conjugate-structured algebraic o CS-ACELP) que utiliza frames de 10 ms (i.e., 80 muestras por frame) y codifica los parámetros de LSP con un cuantificador vectorial de dos estados. Las ganancias (dos por frame) también son codificadas mediante cuantización vectorial.

7.9. Shorten

Shorten es un sencillo compresor sin pérdidas de propósito especial para archivos de forma de onda. Cualquier archivo cuyos ítems de datos (que se denominan *muestras*) suban y bajen como en una onda pueden ser comprimidos eficientemente mediante este método. Su desarrollador [Robinson 94] tenía en mente aplicaciones para la compresión de voz (donde los archivos de audio con la voz estaban distribuidos en un CD-ROM), pero otros archivos de forma de onda pueden ser igualmente comprimidos. El rendimiento de la compresión de Shorten no es tan bueno como el de mp3, pero Shorten es sin pérdidas. Shorten funciona mejor en archivos con muestras de baja amplitud y baja frecuencia, donde obtiene factores de compresión de 2 ó mejor. Se ha implementado en UNIX y en MS-DOS y está disponible gratuitamente en [Softsound 03].

Shorten codifica las muestras individuales del archivo de entrada particionando el archivo en bloques, prediciendo cada muestra de algunas de sus predecesoras, restando la predicción de la muestra, y codificando la diferencia con un código especial de tamaño variable. También tiene un modo con pérdidas de datos, donde las muestras son cuantificadas antes de ser comprimidas. El algoritmo ha sido implementado por su desarrollador y admite archivos en los formatos de audio ulaw (Sección 7.5), s8, u8, s16 (éste es el formato de entrada por defecto), u16, s16x, u16x, s16hl, u16hl, s16lh, y u16lh, donde "s" y "u" proviene de "signed (con signo)" y "unsigned (sin signo)", respectivamente; una "x" final especifica datos bytes mapeados, "hl" implica que el byte de orden superior de una muestra va seguida en el archivo por el byte de orden inferior, y "lh" significa lo contrario.

Un archivo completo es codificado en bloques, donde el tamaño del bloque (típicamente 128 ó 256 muestras) es especificado por el usuario y tiene un valor por defecto de 256. (A una tasa de muestreo de 16 kHz, éste equivale a 16 ms de sonido.) Las muestras en el bloque se convierten en primer lugar a números enteros con una media esperada de 0. La idea es que las muestras dentro de cada bloque tengan la misma característica espectral y puedan, por lo tanto predecirse con precisión. Algunos archivos de audio están formados por varios canales intercalados, por eso Shorten comienza separando los canales en cada bloque. Por consiguiente, si el archivo tiene dos canales y las muestras se intercalan como L_1, R_1, L_2, R_2, y así sucesivamente hasta L_b, R_b, el primer paso crea los dos bloques $(L_1, L_2, \ldots, L_b)$ y $(R_1, R_2, \ldots, R_b)$ y cada bloque se comprime posteriormente por separado. En la práctica, los bloques que se corresponden con los canales de audio están a menudo altamente correlacionados, por lo que métodos tan sofisticados, como MLP (Sección 7.7), tratan de eliminar las correlaciones interbloque antes de abordar las muestras internas a un bloque.

Una vez que el bloque ha sido construido, sus muestras son predichas y los valores de diferencia son calculados. Un valor predicho $\hat{s}(t)$ para la muestra actual $s(t)$ se calcula a partir de las p muestras inmediatamente anteriores mediante una combinación lineal (véase también la Sección 4.26)

$$\hat{s}(t) = \sum_{i=1}^{p} a_i s(t-i).$$

Si la predicción se hace correctamente, entonces la diferencia (también denominada error o residual) $e(t) = s(t) - \hat{s}(t)$ será casi siempre un número pequeño (positivo o negativo). El tipo de onda más simple es la estacionaria. En tal onda, un único conjunto de coeficientes a_i siempre produce la mejor predicción. Naturalmente, la mayoría de las olas no son estacionarias y debe seleccionarse un conjunto diferente de coeficientes a_i para predecir cada muestra. Dicha selección puede efectuarse de distintas maneras, involucrando cada vez más muestras de vecinos, y ésto produce predictores de órdenes diferentes.

Un predictor de orden cero simplemente predice cada muestra $s(t)$ como cero. Un predictor de primer orden (Figura 7.27a) predice cada $s(t)$ como su predecesor $s(t-1)$. De manera similar, un predictor de segundo orden (Figura 7.27b) calcula un segmento recto (una función lineal o un polinomio de grado 1) a partir de $s(t-2)$ a $s(t-1)$ y lo continúa para predecir $s(t)$. Extendiendo esta idea a un punto más, un factor de predicción de tercer orden (Figura 7.27c) calcula un polinomio de grado 2 (una sección cónica) que pasa a través de los tres puntos $s(t-3)$, $s(t-2)$, y $s(t-1)$ y lo extrapola para predecir $s(t)$. En general, un predictor de orden n-ésimo calcula un polinomio de grado $(n-1)$ que pasa a través de los n puntos, de $s(t-n)$ a $s(t-1)$ y lo extrapola para predecir $s(t)$. Mostramos cómo calcular los predictores de segundo y tercer orden.

Dados los dos puntos $P_2 = (t-2, s_2)$ y $P_1 = (t-1, s_1)$, podemos escribir la ecuación paramétrica de la recta que los une como:

$$L(u) = (1-u)P_2 + uP_1 = (1-u)(t-2, s_2) + u(t-1, s_1) = (u+t-2, (1-u)s_2 + u s_1).$$

Es fácil ver que $L(0) = P_2$ y $L(1) = P_1$. La extrapolación al siguiente punto, en $u = 2$, produce $L(2) = (t, 2s_1 - s_2)$. Usando esta notación, llegamos a la conclusión de que el predictor de segundo orden predice la muestra $s(t)$ como la combinación lineal $2s(t-1) - s(t-1)$.

Para el predictor de tercer orden, empezamos con los tres puntos $P_3 = (t-3, s_3)$, $P_2 = (t-2, s_2)$, y $P_1 = (t-1, s_1)$. El polinomio de grado 2 que pasa a través de los puntos viene dado por el polinomio uniforme cuadrático de interpolación de Lagrange (véase, por ejemplo, [Salomon 06] p. 78, la Ecuación

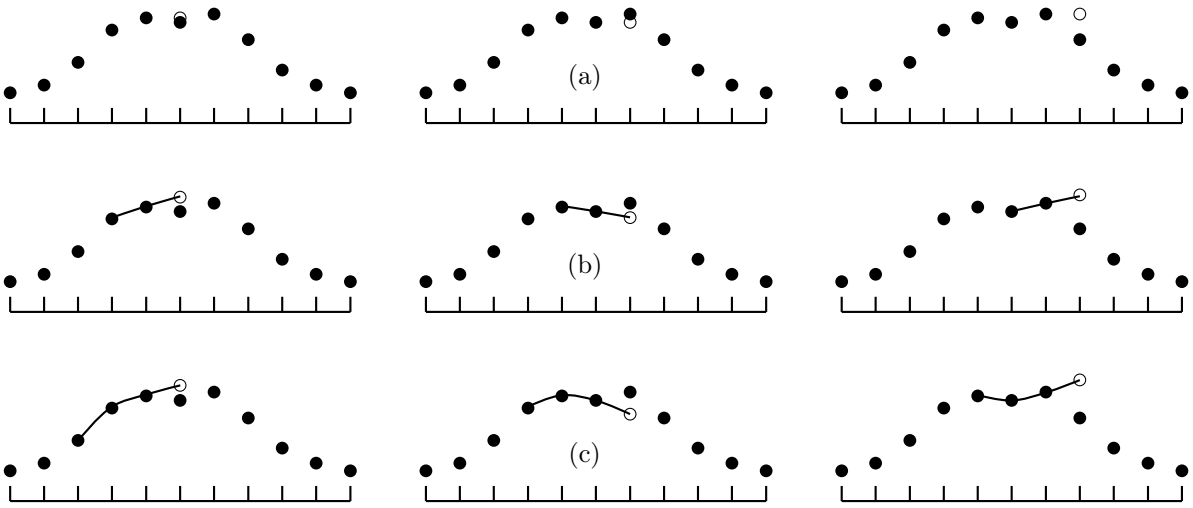


Figura 7.27: Predictores de órdenes 1, 2, y 3.

3.12).

$$L(u) = \left[u^2, u, 1\right] \begin{bmatrix} \frac{1}{2} & -1 & \frac{1}{2} \\ -\frac{3}{2} & 2 & -\frac{1}{2} \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_3 \\ P_2 \\ P_1 \end{bmatrix}$$
$$= \left[\frac{u^2}{2} - \frac{3u}{2} + 1\right] P_3 + \left(-u^2 + 2u\right) P_2 + \left[\frac{u^2}{2} - \frac{u}{2}\right] P_1.$$

Es fácil verificar que $L(0) = P_3$, $L(1) = P_2$, y $L(2) = P_1$. Extrapolando a $u = 3$ se obtiene $L(3) = 3P_1 - 3P_2 + P_3$. Cuando ésto se traduce a nuestras muestras, el resultado es $\hat{s}(t) = 3s(t-1) - 3s(t-2) + s(t-3)$. Los primeros cuatro predictores se resumen como:

$$\begin{aligned} \hat{s}_0(t) &= 0 \\ \hat{s}_1(t) &= s(t-1), \\ \hat{s}_2(t) &= 2s(t-1) - s(t-2), \\ \hat{s}_3(t) &= 3s(t-1) - 3s(t-2) + s(t-3). \end{aligned} \tag{7.5}$$

Estos predictores ahora pueden utilizarse para calcular los valores de error (o diferencia) para los primeros cuatro órdenes:

$$\begin{aligned} e_0(t) &= s(t) - \hat{s}_0(t) = s(t), \\ e_1(t) &= s(t) - \hat{s}_1(t) = s(t) - s(t-1) = e_0(t) - e_0(t-1), \\ e_2(t) &= s(t) - \hat{s}_2(t) = s(t) - 2s(t-1) + s(t-2) \\ &= [s(t) - s(t-1)] - [s(t-1) - s(t-2)] = e_1(t) - e_1(t-1), \\ e_3(t) &= s(t) - \hat{s}_3(t) = s(t) - 3s(t-1) + 3s(t-2) - s(t-3) \\ &= [s(t) - 2s(t-1) + s(t-2)] - [s(t-1) - 2s(t-2) + s(t-3)] \\ &= e_2(t) - e_2(t-1). \end{aligned} \tag{7.6}$$

Este cálculo es recursivo pero implica sólo tres pasos; es aritméticamente simple, y no requiere multiplicación alguna (véase también la Ecuación (7.8)).

i	Binario	Signo	LSB	No. de Ceros	Código	i	Binario	Signo	LSB	No. de Ceros	Código
0	0	0	00	0	0|1|00						
1	1	0	01	0	0|1|01	−1	1	1	01	0	1|1|01
2	10	0	10	0	0|1|10	−2	10	1	10	0	1|1|10
3	11	0	11	0	0|1|11	−3	11	1	11	0	1|1|11
4	100	0	00	1	0|01|00	−4	100	1	00	1	1|01|00
5	101	0	01	1	0|01|01	−5	101	1	01	1	1|01|01
6	110	0	10	1	0|01|10	−6	110	1	10	1	1|01|10
7	111	0	11	1	0|01|11	−7	111	1	11	1	1|01|11
8	1000	0	00	2	0|001|00	−8	1000	1	00	2	1|001|00
11	1011	0	11	2	0|001|11	−11	1011	1	11	2	1|001|11
12	1100	0	00	3	0|0001|00	−12	1100	1	00	3	1|0001|00
15	1111	0	11	3	0|0001|11	−15	1111	1	11	3	1|0001|11

Tabla 7.28: Varios códigos de Rice positivos y negativos.

Para la máxima compresión, es posible calcular los cuatro predictores y sus errores y seleccionar el error más pequeño. Sin embargo, la experiencia adquirida por el desarrollador del método indica que incluso un predictor de orden cero, produce una compresión típica del 48 %, y yendo hasta el final, la predicción de tercer orden mejora sólo hasta el 58 %. Para la mayoría de los casos, no hay ninguna necesidad, por lo tanto, de utilizar predictores de orden superior, y la precisión del predictor utilizado debe ser determinada valorando la calidad de compresión frente a consideraciones de tiempo de ejecución. El modo por defecto que Shorten utiliza es la predicción lineal (segundo orden).

Se asume que los valores de error (o diferencia) no están correlacionados y son reemplazados por códigos de tamaño variable. La fuerte correlación entre las muestras originales implica que la mayor parte los valores de error son pequeños o incluso cero, y sólo unos pocos son grandes. También son con signo. Los experimentos sugieren que la distribución de los errores está muy cercana a la bien conocida distribución de Laplace (Figura 4.128b), lo que sugiere que los códigos de tamaño variable utilizados para codificar las diferencias deben seguir esta distribución. (El desarrollador de este método asegura que el error introducido por la codificación de los valores de diferencia siguiendo la distribución de Laplace, en lugar de utilizar la distribución de probabilidad real de las diferencias, es de sólo 0,004 bits.)

Los códigos seleccionados para Shorten han sido desarrollados por Robert F. Rice y se conocen como códigos de Rice ([Rice 79], [Rice 91], y [Fenwick 96a]). Son un caso especial del código de Golomb (Sección 2.5) y también están relacionados con el código subexponential de la Sección 4.20.1. Un código de Rice depende de la elección de una base n y se calcula mediante los siguientes pasos: (1) Se separa el bit de signo del resto del número. Éste se convierte en el bit más significativo del código de Rice. (2) Se separan los n LSBs. Éstos se convierten en los LSBs del código de Rice. (3) Se codifican los bits restantes en unario y se convierten en la parte central del código de Rice. (Si los bits restantes son, digamos, 11, entonces el código unario es: o bien tres ceros seguidos por un 1, o bien tres 1's seguidos por un 0.) Por consiguiente, este código se calcula con unas pocas operaciones lógicas, que es más rápido que el cálculo de un código de Huffman, que requiere recorrer hacia abajo el árbol de Huffman a medida que se recogen los bits del código. Esta característica es especialmente importante para el decodificador, que tiene que ser sencillo y rápido. La Tabla 7.28 muestra ejemplos de este código para $n = 2$ (la columna "No. de ceros" indica el número de ceros en la parte unaria del código).

El código de Rice, similar al código de Huffman, asigna un número entero de bits a cada valor de diferencia, por lo que en general, no es un código de entropía. Sin embargo, el desarrollador de este método asegura que el error introducido por el uso de estos códigos es de sólo 0,12 bits por valor de

Número		Pod	Elias	Elias
Dec.	Binario		Gamma	Gamma sesgado
0	00000	1		1
1	00001	01	1	010
2	00010	0010	010	011
3	00011	0011	011	00100
4	00100	000100	00100	00101
5	00101	000101	00101	00110
6	00110	000110	00110	00111
7	00111	000111	00111	0001000
8	01000	00001000	0001000	0001001
9	01001	00001001	0001001	0001010
10	01010	00001010	0001010	0001011
11	01011	00001011	0001011	0001100
12	01100	00001100	0001100	0001101
13	01101	00001101	0001101	0001110
14	01110	00001110	0001110	0001111
15	01111	00001111	0001111	000010000
16	10000	0000010000	000010000	000010001
17	10001	0000010001	000010001	000010010
18	10010	0000010010	000010010	000010011

Tabla 7.29: Códigos Pod, Elias Gamma , y Elias Gamma sesgado.

diferencia.

Los códigos de Rice son ideales para ítems de datos con una distribución de Laplace, pero existen otros códigos prefijo que son más fáciles de construir y decodificar y que pueden, bajo ciertas circunstancias, superar a los códigos de Rice. La Tabla 7.29 enumera tres códigos. El código "pod", debido a Robin Whittle [firstpr 06], codifica el número cero con el bit único 1, y codifica el número binario $1\underbrace{b\ldots b}_{k}$ como $\underbrace{0\ldots0}_{k+1}1\underbrace{b\ldots b}_{k}$. En ambos casos, el código pod es un bit más largo que el código de Rice, en cuatro casos se tiene la misma longitud, y en todos los demás casos es más corto que los códigos de Rice. El código de Elias Gamma [Fenwick 96a] es idéntico al código pod menos su cero de más a la izquierda. Por tanto, es más corto, pero no incluye un código para el cero. El código de Elias Gamma sesgado corrige esta falla de una manera obvia, pero al costo de generar algunos códigos un bit más largos.

Queda la cuestión de qué valor de base n seleccionar para los códigos de Rice. La base determina cuántos bits de orden inferior de un valor de diferencia se incluyen directamente en el código de Rice, y ésto está linealmente relacionado con la varianza de los valores de diferencia. El desarrollador proporciona la fórmula $n = \log_2 [\log(2) E(|x|)]$, donde $E(|x|)$ es el valor esperado de las diferencias. Este valor es la suma $\sum |x| p(x)$ tomadas todas las posibles diferencias x.

Los pasos descritos hasta ahora producen una codificación sin pérdidas de las muestras de audio. A veces, es aceptable una cierta pérdida en las muestras, si con ello la compresión mejora significativamente. Shorten ofrece dos opciones de compresión con pérdida cuantificando las muestras de audio originales antes de ser compridas. La cuantificación se efectúa por separado para cada segmento. Una opción con pérdidas codifica cada segmento a la misma tasa de bits (el usuario especifica la tasa de bits máxima), y la otra opción mantiene una relación señal–a–ruido especificada por el usuario en cada segmento (el usuario especifica el mínimo aceptable de la relación señal–a–ruido en dB).

Pruebas de Shorten indican que generalmente funciona mejor y más rápido que los compresores de

UNIX compress y gzip, pero que las opciones con pérdidas son lentas.

7.10. FLAC

El nombre FLAC es un acrónimo que procede de *free lossless audio compression* (compresión libre de audio sin pérdidas). FLAC fue especialmente diseñado para la compresión de audio y también soporta la transmisión y almacenamiento de datos de audio. FLAC es una idea original de Josh Coalson que fue desarrollada en el año 1999 basada en las ideas de Shorten. Él entonces inició el proyecto FLAC en el bien conocido sitio web sourceforge [sourceforge.flac 06] liberando su implementación de referencia. Desde entonces, muchos desarrolladores han contribuido a mejorar la implementación de referencia y a la escritura de implementaciones alternativas. El proyecto FLAC, administrado y coordinado por Josh Coalson, mantiene el software y proporciona un *codec*[8] de referencia y *plugins*[9] de entrada para varios reproductores de audio populares.

FLAC es libre tanto en el sentido de que está disponible sin costo alguno como en que sus especificaciones y formato están totalmente abiertos al público, y puede ser utilizado para cualquier propósito. Ésto incluye el código fuente de FLAC, que está disponible bajo licencias open-source (de código abierto). Ni el formato FLAC ni ninguno de los algoritmos implementados de codificación/decodificación están cubiertos por patentes conocidas. Por consiguiente, FLAC es uno de los primeros métodos de compresión de audio sin pérdidas abierto y libre. Es posible que Squish (u Ogg Squish, por Chris Montgomery, [Ogg squish 06]), ostente el título de "primero", aunque no está muy extendido.]

El proyecto FLAC consta de: (1) el formato del *stream* comprimido, (2) codificadores y decodificadores de referencia en forma de biblioteca, (3) un software de línea de comandos FLAC codec (codificador FLAC), (4) metaflac, un editor de metadatos de línea de comandos para archivos FLAC, y (5) plugins de entrada para diversos reproductores musicales.

La implementación de referencia de FLAC compila en muchas plataformas y soporta muchos sistemas operativos. Algunos ejemplos son Windows, BeOS, OS/2, y la mayoría de los sistemas Unix y tipo Unix (incluyendo Linux, *BSD, Solaris, y Mac OS X).

SourceForge.net es el sitio web sobre desarrollo de software de código fuente abierto (*Open Source*) más grande del mundo; aloja más de 100 000 proyectos y más de 1 000 000 usuarios registrados con un recurso centralizado para la gestión de proyectos, problemas de comunicación, y código. SourceForge.net tiene el mayor repositorio de código fuente abierto y aplicaciones disponibles en Internet, y hospeda más productos de desarrollo de código abierto que cualquier otro sitio o red mundial. SourceForge.net ofrece una gran variedad de servicios para los proyectos que organizamos, y para la comunidad Open Source.

SourceForge.net proporciona alojamiento gratuito para los proyectos de desarrollo de software Open Source. La esencia del modelo de desarrollo Open Source es la generación rápida de soluciones dentro de un entorno abierto y colaborativo. La colaboración dentro de la comunidad Open Source (desarrolladores y usuarios finales) promueve un mayor nivel de calidad, y ayuda a garantizar la viabilidad a largo plazo de los datos y aplicaciones.

SourceForge.net es propiedad del Grupo de tecnologías de código fuente abierto (Open Source Technology Group u OSTG).

Desde `http://sourceforge.net/docs/about`

FLAC comprime el audio de entrada bloque por bloque y se basa en la predicción y los códigos de Rice. Por lo tanto, se parece bastante a Shorten (Sección 7.9). Un bloque de muestras de audio se

[8] Codificador.

[9] Un pluging es software que al instalarlo en un programa amplia las características de funcionamiento originales del mismo, ya sea proporcionándole la capacidad de tratar con un nuevo tipo de archivo, ya sea cualquier otra mejora.

comprime prediciendo las muestras, restando cada muestra de audio de su predicción, y codificando la diferencia con un código de Rice (Secciones 2.5 y 7.9). El stream comprimido está formado por los códigos de Rice, unos pocos parámetros especificando la predicción, e información en forma de metadatos. Ésta última incluye información tal como la velocidad de muestreo del audio, el número de canales de audio, las tasas de datos mínima y máxima, el tamaño de bloque mínimo y máximo, la firma MD5 de los datos de audio sin codificar, datos de relleno (*padding*), búsqueda de tablas, etiquetas, hojas cue o *cuesheets*[10], y datos específicos de la aplicación. Los usuarios que necesitan metadatos personalizados pueden definir su propio formato y solicitar un ID de metadatos a los desarrolladores de FLAC en [flacID 06].

FLAC no prohíbe que esquemas de prevención de copia como el DRM (*digital rights management* o gestión digital de derechos) aparezcan en streams de FLAC, simplemente carece de prestaciones para la prevención de copia en el formato. Es posible, por ejemplo, para Apple encriptar un stream de FLAC con FairPlay y almacenarlo en un contenedor `m4a`.

Muchos amantes de la música que se preocupan por el audio de alta calidad sospechan de los métodos de compresión con pérdida como mp3 o AAC, y es cierto que los métodos con pérdidas son sensibles a ciertos tipos de sonido y, a veces, pueden producir un sonido reconstruido de inferior calidad. Tales usuarios/oyentes prefieren FLAC, y han hecho este formato popular. Como resultado, los fabricantes de equipos de audio han comenzado a apoyar este formato en el hardware. Actualmente (mediados de 2006), muchos audios estéreo domésticos, equipos estéreo para el coche, dispositivos de audio portátiles y de mano pueden reproducir música en este formato. El software también está disponible en muchas plataformas de computadoras y en la mayoría de los sistemas operativos —incluyendo Windows, Unix (Linux, * BSD, Solaris, OS X, IRIX), BeOS, OS/2, y Amiga— para codificar y decodificar sonido en FLAC. En [flac.devices 06] se mantiene una lista de los dispositivos y del software disponibles.

Debido a su naturaleza sin pérdidas, FLAC se limita sólo a las muestras de audio enteras. Existen codificadores de audio que pueden admitir muestras de audio en coma flotante, pero los cálculos en coma flotante suelen devolver resultados ligeramente diferentes en equipos diferentes, debido a diferencias en el tamaño de palabra del computador y de la manera en que sus circuitos de la ALU (*arithmetic and logic unit* o unidad aritmética y lógica) controlan los errores de redondeo. La restricción de FLAC a muestras de audio enteras asegura que una implementación correcta de FLAC será capaz de reconstruir completa y fielmente cualquier archivo de audio.

FLAC puede manejar las tasas de muestreo de audio en un amplio rango. Cualquier velocidad de muestreo de 1 Hz a 1 048 570 Hz (en incrementos de 1 Hz) puede ser introducida y comprimida correctamente. FLAC está diseñada para manejar hasta ocho canales de audio (que pueden agruparse en estéreo o como canales de sonido envolvente 5.1, página 841) y aprovechar las correlaciones entre los canales para mejorar la compresión.

Objetivos de FLAC. El ser de código abierto significa que cualquier persona puede ser voluntaria para ayudar en la desarrollo de FLAC. Por consiguiente, es importante contar con un conjunto bien definido de objetivos para este proyecto. Los desarrolladores deben mantener estos objetivos en mente cuando tratan de mejorar, modificar o mantener cualquier parte de FLAC, pero el administrador del proyecto puede modificar los objetivos de cuando en cuando, para reflejar los cambios de pensamiento y las necesidades de los usuarios. Los objetivos actuales son:

- FLAC es y debe seguir siendo un formato abierto con una implementación de referencia de código fuente abierto.
- FLAC es y debe seguir siendo sin pérdidas. Los futuros desarrolladores no deben añadir compresión con pérdida incluso como una opción.

[10] Las hojas (o archivos) *cue* tienen información sobre la distribución de las pistas de datos en un disco compacto, con datos como el título que identifica el audio, el artista que lo creó, y la duración.

- La eficiencia de la compresión de FLAC debe estar a la par o ser mejor que la de otros codificadores sin pérdidas.
- La decodificación debe ser rápida. Un decodificador de FLAC debería funcionar al menos en tiempo real (i.e., decodificar el audio mientras se está reproduciendo), incluso en un hardware de velocidad media.
- FLAC debe soportar búsquedas precisas de muestras rápidas. Un usuario que está escuchando audio decodificado debe ser capaz de saltar a cualquier punto del audio con facilidad.
- Al ser sin pérdidas, FLAC debe permitir la reproducción de pistas de audio sin lagunas consecutivas.
- No se permite ninguna clase de protección de copia en un stream de FLAC comprimido .

Estos objetivos implican las principales características de FLAC, que son las siguientes:

- Sin pérdidas. Existen muchos métodos de compresión de audio con pérdidas excelentes, por lo que los desarrolladores de FLAC decidieron diseñarlo como sin pérdidas y mantenerlo sin pérdidas. El audio decodificado debe ser idéntico a la entrada original, y cada implementación tiene que ser probada para verificar esta propiedad. FLAC tiene una opción de "verificación" (`-V`) para verificar la salida durante la codificación. El decodificador funciona en paralelo con el codificador, y la salida del decodificador es comparada con la entrada original. El software señala un error si encuentra una discrepancia.

  Cada cuadro o *frame* en el stream comprimido incluye un CRC (código de redundancia cíclica) de 16 bits de los datos del frame para detectar (pero no corregir) los errores de transmisión. Ya se ha mencionado que el encabezado del archivo comprimido contiene la firma MD5 de los datos originales (sin comprimir). Ésto puede ser utilizado por el decodificador para verificar su salida, y puede ser también empleado por usuarios y desarrolladores para probar una implementación.
- Rápido. FLAC fue diseñado para una rápida decodificación. Ésto es importante en la compresión de audio porque el decodificador a menudo tiene que emitir el audio en tiempo real, para ser decodificado y reproducido simultáneamente. FLAC es, por lo tanto, un método de compresión asimétrico. La decodificación es rápida, ya que es sencilla (y mucho menos computacionalmente intensiva que en los decodificadores de percepción) y utiliza sólo operaciones con enteros. Soporte de hardware: Debido a la implementación de referencia libre de FLAC y a la baja complejidad de la decodificación, FLAC es actualmente el único codificador de audio sin pérdidas que tiene algún tipo de soporte en hardware.
- Transmisión continua. El stream comprimido de FLAC está formado por trozos llamados *frames*, donde cada trozo contiene todos los datos necesarios para decodificar el frame. La decodificación de un frame no requiere los frames anteriores. El stream de FLAC comprimido incluye los códigos de sincronización y CRCs (similares a los formatos MPEG y otros) que hacen que sea fácil para el decodificador saltar rápidamente a cualquier punto en el stream comprimido.
- Reubicable. La capacidad de saltar a cualquier punto en el audio es útil para la reproducción, así como para las aplicaciones de edición de audio.
- Metadatos flexibles. El stream de FLAC comprimido comienza con bloques de metadatos que incluyen información útil. Los usuarios pueden diseñar sus propios bloques de metadatos, con el objeto de guardar información especial y privada, y pueden aplicarlos al proyecto FLAC para los IDs de bloque.

- Adecuado para su almacenamiento en archivos. Un archivo puede beneficiarse de los datos comprimidos, pero requiere compresión sin pérdidas. La compresión sin pérdidas es también muy útil para la conversión entre formatos. El hecho de que FLAC es de código abierto también anima a los usuarios a seleccionar FLAC para guardar el audio en un archivo.

- Práctico para su almacenamiento en CD. Uno de los bloques de metadatos en el stream de FLAC comprimido tiene una "hoja de referencia" o "cue sheet" para el almacenamiento de la tabla de contenidos de un CD y todas las pistas, y los puntos del índice. En una aplicación típica, un CD se convierte a un archivo único y su *cue sheet* es extraído. Luego el archivo es comprimido en un único archivo de FLAC con un bloque de metadatos que contiene la hoja de referencia extraída. Si el CD original está dañado, el archivo FLAC con el cue sheet puede ser convertido de nuevo y "quemado" para convertirse en una copia exacta del CD.

- Resistente a errores. Muchos codificadores sin pérdidas no incluyen características de fiabilidad o integridad de los datos. Un solo error en el stream comprimido puede impedir que el decodificador pueda continuar. El stream de FLAC comprimido, por el contrario, está formador por frames, cada uno representando una pequeña fracción de un segundo de audio. Si un error daña un frame, el decodificador aún puede localizar el siguiente frame y continuar. Ésto limita cualquier daño causado al audio por errores durante la transmisión o el almacenamiento; una característica importante y práctica.

Antes de entrar en los detalles del algoritmo FLAC, indicamos aquí las definiciones de los dos conceptos clave: bloque y marco. Un bloque en FLAC es un número consecutivo de muestras de audio. Si la entrada tiene más de un canal de audio, un bloque está formado por un subbloque para cada canal. Por consiguiente, un subbloque es un conjunto de muestras de audio contiguos del mismo canal. El tamaño del bloque es en realidad el tamaño de un subbloque. Así, si el tamaño del bloque es 4 608 y hay dos canales de audio, entonces el tamaño real del bloque es de $4\,608 \times 2 = 9\,216$ muestras. El tamaño del bloque es importante, y afecta a la eficiencia, velocidad, y fiabilidad de un codificador de FLAC. Un bloque pequeño representa una fracción muy pequeña de un segundo de audio. En caso de error durante la decodificación, sólo se pierde un pequeño periodo de tiempo. Como aspecto negativo, los bloques pequeños implican muchos bloques, que representan más trabajo, tanto para el codificador como para el decodificador. Además, cada bloque se convierte en un frame en el stream comprimido y un frame tiene un encabezado. Un gran número de cabeceras se traduce en más sobrecarga, lo que lleva a una menor compresión. Otra consideración es que los bloques grandes presentan una correlación reducida entre sus muestras de audio y por tanto conducen a una predicción y compresión ineficientes.

En un caso típico donde la tasa de muestreo es de 44,1 kHz y se utiliza la predicción lineal, el tamaño de bloque óptimo es de 2 000–6 000 muestras y el valor predeterminado en tal caso es de 4 608 muestras (excepto cuando la opción controlada por el usuario `-l` es establecida a cero, en cuyo caso el tamaño de bloque se convierte en 1 152).

Para cada bloque del archivo de entrada se escribe un frame en el archivo de salida de FLAC. El frame comienza con una cabecera que es seguida por un número de subframes. Cada subframe comienza con su propio encabezado, seguido por los códigos de Rice para las muestras de audio codificadas desde el mismo canal. Un frame consta de un subframe para cada canal de audio, y cada subframe consta del mismo número de muestras de audio (codificadas con Rice).

Un stream de salida de FLAC comienza con la cadena de identificación de cuatro bytes `fLaC`. Ésta es seguida por el bloque de metadatos STREAMINFO. Opcionalmente pueden seguir otros bloques de metadatos. Ellos incluyen toda la información adicional que necesitan el decodificador y el usuario final. Los bloques de metadatos pueden ser de cualquier longitud, y pueden definirse otros nuevos por los usuarios. Por consiguiente, un decodificador de FLAC puede no reconocer ciertos bloques de metadatos y está permitido omitirlos silenciosamente. El resto del stream consta de frames (debe haber por lo menos uno de tales frames).

El codificador. El codificador de FLAC codifica cada bloque de muestras de audio por separado. El primer paso es explotar opcionalmente la correlación entre canales de muestras de audio. Si la entrada es estéreo (canales izquierdo y derecho), es descorrelacionada mediante la transformación sin pérdidas habitual central = (izquierdo + derecho) /2 y secundario = izquierdo − derecho. FLAC tiene dos opciones seleccionadas por el usuario para este descorrelación. La opción `−m` indica al codificador que codifique ambos, los canales izquierdo y derecho, y los canales mitad y lado, y posteriormente seleccione el mejor (i.e., el más pequeño) resultado. La opción `−M` indica al codificador que cambie adaptativamente entre izquierdo–derecho y central–secundario y seleccione siempre el frame más pequeño.

El paso siguiente del codificador es el modelado. Dado un bloque de muestras de audio a_i (de los canales izquierdo, derecho, central, secundario, o cualquier otro), el codificador intenta encontrar una función $f(t)$ tal que $f(t_i)$ se acerque a a_i para todos los valores de i y para ciertos valores de t_i. La función (denominada función de aproximación o de ajuste) queda completamente especificada mediante algunos parámetros que son enviados al decodificador como información adicional. El codificador entonces, efectúa la resta $a_i - f(t_i)$ para obtener una diferencia (o residuo) e_i. Si la función se ajusta mejor a las muestras de audio, los residuos serán números enteros pequeños (y pueden, por supuesto, ser negativos).

FLAC emplea cuatro métodos para la construcción de una función de aproximación (aunque sólo los dos últimos son de carácter general y proporcionan compresión).

- Literal. Este método siempre predice una muestra de audio cero. La diferencia (o residuo) es, por lo tanto, la muestra de audio original. No tiene sentido reemplazar las muestras de audio con un código de Rice, por lo que este predictor no codifica las muestras y no proporciona compresión. Literal es esencialmente un predictor de orden cero de muestras de audio. La ventaja del predictor literal es la rápida decodificación. Si al decodificador se le ha comunicado que un cierto bloque o subbloque emplea este predictor, el decodificador simplemente introduce las muestras de audio desde el stream comprimido sin decodificación alguna. Literal es la base sobre la que los otros predictores son medidos. Un buen codificador FLAC debe comprimir cada subbloque dos veces, primero con el método de predicción seleccionado por el usuario, y después con el predictor literal. El resultado más pequeño debe ser escrito en el stream comprimido. El predictor literal debe hacerlo por lo menos tan bien como cualquier otro predictor cuando el audio original es aleatorio o casi aleatorio.

- Constante. El sonido es una onda. Oímos el sonido cuando las vibraciones del aire golpean nuestros oídos. Tales vibraciones se convierten mediante un micrófono a una onda eléctrica (voltaje que varía con tiempo), y esta onda se muestrea para producir las muestras de audio. En ausencia de sonido, un micrófono emite una tensión constante (normalmente cero pero posiblemente distinta de cero), la cual se convierte en una larga secuencia de muestras de audio idénticas; un silencio digital. Un codificador FLAC sofisticado debe comprobar cada subbloque en busca de silencio digital antes de codificarlo. Cuando tal subbloque es encontrado, se comprime mediante la codificación run-length.

- Predictor lineal fijo. Éste es un predictor sencillo que ajusta un polinomio a las muestras de audio (este método es seleccionado si la opción `−l` de FLAC controlada por el usuario está puesta a cero). La predicción mediante ajuste polinómico es más rápida que LPC (el siguiente método) pero generalmente produce archivos un 5–10 % más grandes. El ajuste polinómico es similar al empleado por Shorten (predicción de orden cero a tercer orden, Sección 7.9), pero también incluye un polinomio de cuarto orden. La única información adicional incluida en el stream comprimido es el orden de la predicción, un número entero de 3 bits con valores de 0, 1, 2, 3, ó 4. La predicción polinómica de cuarto de orden se discute en la Sección 7.10.1.

- Predicción lineal FIR. Éste es un método más sofisticado que también se conoce como codificación predictiva lineal general (LPC) (véase la Sección 4.26) y [Rabiner y Schafer 78]). Este predictor es utilizado si la opción `-l` se establece en un valor comprendido entre 1 y 32. Este valor se convierte en el orden máximo de la LPC. Generalmente, cuanto mayor es el orden máximo, más exacta es la predicción (ajuste) proporcionada por la LPC, pero más lento es el codificador. Sin embargo, el aumento del orden (normalmente por encima de 9) conlleva unos rendimientos decrecientes y puede incluso degradar el ajuste de la LPC y de este modo disminuir la compresión. LPC se describe en la Sección 7.10.2, y la siguiente cita procede del sitio oficial de FLAC:

> La referencia del codificador utiliza el algoritmo de Levinson-Durbin para calcular los coeficientes de la LPC a partir de los coeficientes de autocorrelación, y los coeficientes son cuantificados antes de calcular el residuo. Mientras que los codificadores como Shorten utilizan una cuantificación fija para la entrada completa, FLAC permite que la precisión del coeficiente cuantificado varíe de subframe a subframe. El codificador de referencia de FLAC estima la precisión óptima a utilizar en función del tamaño del bloque y el rango dinámico de la señal original

Otra diferencia entre los dos últimos métodos es que el parámetro para el polinomio de ajuste puede ser escrito en el stream comprimido como un número de 3 bits, mientras que el tamaño de los parámetros de la LPC dependen del tamaño de la muestra (número de bits por muestra de audio) y del orden máximo de la LPC.

Residuo de la codificación. Una vez que las muestras de audio para un subbloque han sido calculadas, se restan de las muestras originales para obtener los residuos. Se ha observado experimentalmente (véase la Sección 7.2.1 para tal experimento) que los residuos tienen a menudo una distribución Laplaciana (Ecuación (4.39) y la Figura 4.128b), y se sabe que los códigos de Rice (Sección 7.9) son ideales para tal distribución. En consecuencia, FLAC codifica los residuos con códigos de Rice. Recordemos que cada bloque original de muestras de audio se convierte en un frame en el stream comprimido y cada subbloque se convierte en un subframe. La mayor parte del contenido de un subframe son códigos de Rice.

Los códigos de Rice dependen de la elección de un único parámetro m, y el mejor valor para el parámetro depende de la precisión en la forma de la distribución de Laplace. El codificador de FLAC estima m basándose en la distribución estadística de los residuos, mediante $m = \log_2((log_2 2)\, E\, \|r_i\|)$ donde E es el valor esperado de los residuos r_i (esta estimación fue propuesta originalmente por Shorten en [Robinson 94]).

Aún más, el codificador puede variar el parámetro de Rice dentro de un subframe si la distribución de los códigos de Rice varía significativamente. Los residuos en un subframe pueden dividirse en particiones, donde cada partición utiliza un parámetro de Rice diferente. La opción `-r` controlada por el usuario puede ser asignada a los valores m y n (para mín y max). El codificador de FLAC trata luego de dividir cada subframe en 2^m particiones, luego en 2^{m+1} particiones, y así sucesivamente, hasta 2^n particiones. En cada intento, a las particiones se les asignan diferentes valores del parámetro Rice, y el codificador mide el tamaño final del subframe. Cuando todos los tamaños son conocidos, el codificador selecciona el esquema de particionamiento que produce el menor subframe. Este proceso es lento y se hace aún más lento cuando el usuario selecciona valores m y n que son muy diferentes.

La literatura FLAC recomienda establecer tanto m como n a 2, a menos que el tamaño del bloque sea grande, en cuyo caso los valores recomendados son $m = n$ donde n satisface $^{\text{tamaño de bloque}}/_{(2^n)} = 128$. Los valores $m = 0$ y $n = 16$ producen una optimización máxima, pero la codificación es lenta.

El último paso de la codificación es el *framing*. El codificador prepara los frames y escribe cada uno de ellos en el stream comprimido. Cada frame contiene un encabezado de la frame, códigos de Rice, y el frame pie de página.

Formato de salida FLAC. FLAC puede ser modificado en el futuro, por lo que el formato de su stream de salida contiene algunos espacios vacíos, reservados. En el futuro, estos espacios pueden contener un número de versión de FLAC y otra información relativa a características que aún no existen. Algunos campos están limitados a sólo ciertos patrones de bits, mientras que otros patrones son inválidos. Ésto contribuye a la integridad de la salida. Todos los números incluidos en la salida son números enteros y están en formato big-endian. La salida de FLAC comienza con el marcador `fLaC`, seguido de uno o más bloques de metadatos. El metadato STREAMINFO es obligatorio y puede haber hasta 128 bloques de metadatos diferentes. Actualmente, están definidos los siguientes tipos de metadatos:

El término Big Endian significa que el byte de orden superior (el "big end" o extremo más grande) de un número o una cadena es almacenado en la memoria en la dirección más baja (la que viene primero). Por ejemplo, dado el número de 4 bytes $b_3b_2b_1b_0$, si el byte más significativo b_3 está almacenado en la dirección A, entonces el byte menos significativo b_0 será almacenado en la dirección $A + 3$.

- STREAMINFO. Este bloque de metadatos contiene información acerca del stream de salida completo, como la frecuencia de muestreo, el número de canales de audio, y el número total de muestras. STREAMINFO debe ser el primer bloque de metadatos.

- APPLICATION. Un bloque de metadatos para el uso de aplicaciones de terceros. El único campo obligatorio es un identificador de 32 bits. Cualquier persona puede solicitar al equipo de desarrollo FLAC tal ID. El resto de un bloque APPLICATION es especificado por la aplicación registrada.

- PADDING. Este es un tipo especial de bloque de metadatos que permite una cantidad de relleno arbitraria. El contenido de tal bloque carece de significado. Este tipo de metadatos se utiliza en casos donde el usuario planea editar los metadatos después de la codificación. El usuario puede dar instrucciones al codificador para reservar un bloque PADDING de tamaño suficiente para su posterior inclusión de metadatos.

- SEEKTABLE. Un bloque de metadatos con una tabla para almacenar puntos de búsqueda. Puede haber un único SEEKTABLE en un stream de salida, pero la tabla puede tener cualquier número de puntos de búsqueda. La búsqueda de un punto particular en un archivo de audio comprimido es especialmente útil. Es posible saltar a cualquier muestra dada en un stream de FLAC sin una tabla de búsqueda, pero el tiempo que toma el salto es impredecible debido a que la tasa de bits puede variar ampliamente dentro de stream. La inclusión de puntos de búsqueda en la secuencia de salida reduce considerablemente el tiempo de salto. Cada punto de búsqueda ocupa 18 bytes, por lo que un 1 % de resolución en un stream agrega $100 \times 18 = 1800$ bytes a la salida.

- VORBIS-COMMENT. Este bloque implementa la especificación del comentario de Vorbis. Contiene una lista de pares nombre/valor legibles por humanos, codificada en UTF-8. Los metadatos VORBIS-COMMENT son el único mecanismo con la etiqueta de "soportado oficialmente" en FLAC. Puede haber sólo un bloque VORBIS-COMMENT en un stream de salida.

- CUESHEET. Este útil tipo de bloque de metadatos sirve para los datos que pueden usarse en una hoja cue. Soporta pistas y puntos de índice, que son compatibles con el estándar de disco CD de audio digital, así como otros metadatos CD-DA. El bloque CUESHEET es especialmente útil para realizar copias de seguridad de discos CD-DA, pero parece que su uso normal es como mecanismo de cueing (intercalar pistas) de propósito general para la reproducción de audio.

Los bloques de metadatos son seguidos por los frames donde cada frame comienza con un encabezado. El encabezado del frame comienza con un código de sincronización, y contiene la información adicional necesaria para el decodificador, tal como la velocidad de muestreo, el número de bits por muestra, el número de canales de audio, un número frame/muestra, y un CRC de 8 bits de la cabecera del frame mismo.

El código de sincronización de 14 bits 11111111111110 comienza cada frame. Este código es importante porque el decodificador puede necesitar iniciar la decodificación dentro del stream comprimido y debe tener, por lo tanto, una forma de localizar el comienzo de un frame. Desafortunadamente, los códigos de Rice dentro de un frame caracterizan patrones de bits que arbitrariamente pueden verse como un código de sincronización. En consecuencia, cuando el decodificador encuentra el patrón de bits de una sincronización, tiene que leer el resto de la cabecera y verificar su CRC para asegurarse de que ésta es una cabecera de frame.

La tasa de muestreo, el número de bits por muestra, y el número de canales de audio son los mismos para el trabajo de compresión completo, pero aparecen en la cabecera de cada frame porque el decodificador puede necesitar comenzar la decodificación en medio de un stream y quizás no tenga la posibilidad de leer el bloque de metadatos STREAMINFO localizado al inicio del stream de salida.

El número frame/muestra en la cabecera es, o bien el número de la primera muestra en el frame (en casos donde las tramas tienen tamaños diferentes), o bien el número del frame mismo (en casos donde los frames tienen tamaños fijos). El código de sincronización, CRC, y el número frame/muestra permite la resincronización del decodificador en caso de errores. También permite la búsqueda incluso si ningún punto de búsqueda ha sido especificado.

Debido a que aparecen los mismos parámetros en cada encabezado del frame, el encabezado es simplemente superpuesto y ésto afecta adversamente al rendimiento de la compresión. Para mantener las cabeceras pequeñas, al menos en los casos en que se utilizan los parámetros comunes, FLAC emplea tablas de búsqueda para los valores más comúnmente utilizados de ciertos parámetros. Por ejemplo, la tasa de muestreo se almacena en la cabecera del frame como un código de 4 bits. Ocho de los 16 códigos posibles (códigos de 4 a 11) especifican las frecuencias de muestreo de uso común, de 8, 16, 22,05, 24, 32, 44,1, 48, y 96 kHz. Tres códigos más orientan al decodificador para encontrar la tasa de muestreo en el extremo de la cabecera del frame. Ellos son los siguientes:

- Código 1100, encontrar tasa de muestreo de 8 bits (en kHz) en el extremo de la cabecera.
- Código 1101, encontrar tasa de muestreo de 16 bits (en Hz) en el extremo de la cabecera.
- Código 1110, encontrar tasa de muestreo de 16 bits (en decenas de Hz) en el extremo de la cabecera.

El código 1111 no es válido (para reducir la posibilidad de perder el tiempo de sincronización), y los códigos $0-3$ son reservados. El mismo método es utilizazo para especificar el tamaño del bloque y los bits por muestra.

Los subframes (codificados con Rice) siguen al encabezado. Cada subframe tiene su propia cabecera (con los atributos del subframe, tales como el método de predicción y el orden, y parámetros de codificación residuales), y existe un subframe para cada canal de audio. Tenga en cuenta que los subframes no se intercalan. El decodificador, por lo tanto, necesita reservar búferes lo suficiente grandes como para leer y almacenar los subframes de un frame, y después volver sobre los búferes y decodificar un código de Rice de cada subframe, sucesivamente.

Los subframes son seguidos por un relleno de bits a cero, si es necesario, para completar el último byte del último subframe (recordemos que los códigos de Rice son de tamaño variable, de modo que la longitud total de un subframe puede no ser un múltiplo entero de ocho bits).

El frame pie de página contiene un CRC de 16 bits del frame codificado completo, para una detección de errores robusta. Si el decodificador de referencia detecta un error de CRC, genera un bloque silencioso. Otros decodificadores pueden mostrar un mensaje de error.

```
(* Polinomio cúbico de Lagrange uniforme para predicción de orden 4 en FLAC *)
Clear[Q,t]; t0=0; t1=1; t2=2; t3=3;
Q[t_] := Plus @@ {
((t-t1)(t-t2)(t-t3))/((t0-t1)(t0-t2)(t0-t3))P4,
((t-t0)(t-t2)(t-t3))/((t1-t0)(t1-t2)(t1-t3))P3,
((t-t0)(t-t1)(t-t3))/((t2-t0)(t2-t1)(t2-t3))P2,
((t-t0)(t-t1)(t-t2))/((t3-t0)(t3-t1)(t3-t2))P1}
```

Figura 7.30: Código para un polinomio de Lagrange

7.10.1. Polinomio de predicción de cuarto orden

La Sección 7.9 desarrolla las expresiones para predictores lineales de órdenes de 0 a 3 (véase también la Figura 7.27). La extensión de estos conceptos a un predictor lineal de orden 4 es sencilla. Comenzamos con los cuatro puntos $P_4 = (t-4, s_4)$, $P_3 = (t-3, s_3)$, $P_2 = (t-2, s_2)$, y $P_1 = (t-1, s_1)$ y construimos un polinomio de grado 3 que pase a través de esos puntos (los puntos se seleccionan de manera que su coordenada x corresponda al tiempo y su coordenada y son muestras de audio). Una elección natural es la interpolación polinómica cúbica no uniforme de Lagrange $Q_{3nu}(t) = \sum_{i=0}^{3} P_{i+1} L_i^3(t)$ cuyos coeficientes vienen dados por (véase, por ejemplo, [Salomon 99] p. 204, Ecuaciones 4.17 y 4.18)

$$L_i^3(t) = \frac{\Pi_{j\neq i}^3 (t-t_j)}{\Pi_{j\neq i}^3 (t_i - t_j)}, \quad \text{para } 0 \leq i \leq 3.$$

El código en *Mathematica* de la Figura 7.30 efectúa los cálculos y produce:

$$Q(t) = -\frac{1}{6}(t-1)(t-2)(t-3)P_4 + \frac{1}{2}t(t-2)(t-3)P_3 - \frac{1}{2}t(t-1)(t-3)P_2 + \frac{1}{6}t(t-1)(t-2)P_1.$$

Es fácil verificar que $Q(0) = P_4$, $Q(1) = P_3$, $Q(2) = P_2$, y $Q(3) = P_1$. Extrapolando a $t = 4$ se obtiene $Q(4) = 4P_1 - 6P_2 + 4P_3 - P_4$, y cuando ésto se traduce a muestras de audio el resultado es:

$$\hat{s}_4(t) = 4s(t-1) - 6s(t-2) + 4s(t-3) - s(t-4) \tag{7.7}$$

[compárese con la Ecuación (7.5)]. Cuando esta predicción se resta de la muestra de audio actual $s(t)$, el residuo es:

$$e_4(t) = s(t) - \hat{s}_4(t) = s(t) - 4s(t-1) + 6s(t-2) - 4s(t-3) + s(t-4). \tag{7.8}$$

Ésta es una sencilla expresión aritmética que involucra sólo cuatro adiciones y sustracciones.

◇ **Ejercicio 7.7 (sol. en pág. 1106):** Compruébese el rendimiento de la predicción de orden 4 desarrollada aquí. selecciónense cuatro ítems correlacionados y compárese la predicción de la Ecuación (7.7) con el valor real del siguiente ítem correlacionado.

7.10.2. Codificación predictiva lineal (LPC)

Dado un conjunto de valores correlacionados $\{a_i\}$, la codificación predictiva lineal (LPC) es un sofisticado método para predecir cualquier elemento del conjunto a_i a partir de sus predecesores inmediatos a_{i-1} a a_{i-n}, donde n es un parámetro. La idea es (véase también la Sección 4.26) tratar varios conjuntos de coeficientes c_j y seleccionar el conjunto que minimiza la diferencia:

$$d_i = \left[a_i - \sum_{j=1}^{n} c_j a_{i-j} \right]^2. \tag{7.9}$$

Una referencia para la LPC es [Rabiner y Schafer 78].

En primer lugar, algunos conocimientos matemáticos. En estadística y probabilidad tratamos con variables aleatorias y definimos cantidades útiles, tales como la media y la varianza. Dada una variable aleatoria V que toma valores v_i, denotamos la probabilidad de que V tenga un valor específico v mediante $P(V=v)$. La media aritmética (o promedio) de V es $\left(\sum_i^n v_i\right)/n$, pero los estadísticos también definen una cantidad relacionada E llamada esperanza (o valor esperado) de V. La esperanza $E[V]$ de la variable aleatoria V es la suma de todos los valores v_i de V, cada uno multiplicado por su probabilidad. En consecuencia,

$$E[V]=\sum_i v_i P(V=v_i).$$

El promedio y la esperanza de una variable aleatoria son a menudo, pero no siempre, iguales. En muchos casos, la probabilidad de un valor iguala su frecuencia de aparición. Cuando ésto es cierto, entonces el promedio y la esperanza son iguales. De lo contrario, son diferentes.

La autocorrelación $R_V(d)$ de una variable aleatoria V es la correlación de V con una copia de sí mismo, desplazada d posiciones. Por ejemplo, dado un array $a=(a_1,a_2,\ldots,a_n)$ de n valores, construimos los dos arreglos $x=(a_1,a_2,\ldots,a_{n-1})$ e $y=(a_2,a_3,\ldots a_n)$ de $n-1$ valores cada uno, y calculamos el coeficiente de correlación de Pearson R de x e y. Ésto se convierte en la autocorrelación $R_a(1)$ de un array a con un desplazamiento de 1 posición. Aquellos no familiarizados con la correlación pueden consultar cualquier texto sobre probabilidad y/o estadística. El breve documento [corr.pdf 02] también puede ser de ayuda. Lo que es importante para el propósito de esta discusión es que (bajo ciertos supuestos) la autocorrelación y la esperanza de una variable aleatoria están relacionados mediante $R_V(k)=E[V_iV_{i+k}]$.

Ahora, de vuelta a la codificación de predicción lineal. Con el fin de encontrar el conjunto de coeficientes $\{c_j\}$ que minimiza la Ecuación (7.9), tenemos que diferenciar el valor esperado de d_i con respecto a un coeficiente c_p, fijando la derivada a cero, y hacer ésto para todos los posibles valores de p, desde 1 hasta n. El resultado es:

$$-2\left[\left(a_i-\sum_{j=1}^{n}c_j a_{i-j}\right)a_{i-p}\right]=0,\quad \text{para } 1\le p\le n,$$

ó

$$\sum_{j=1}^{n}c_j E[a_{i-j}a_{i-p}]=E[a_i a_{i-p}]\quad \text{para } 1\le p\le n,$$

Reemplazando las esperanzas, con los coeficientes de autocorrelación se obtiene el sistema de n ecuaciones lineales:

$$\begin{bmatrix} R(0) & R(1) & R(2) & \ldots & R(n-1)\\ R(1) & R(0) & R(1) & \ldots & R(n-2)\\ R(2) & R(1) & R(0) & \ldots & R(n-3)\\ \vdots & \vdots & \vdots & & \vdots\\ R(n-1) & R(n-2) & R(n-3) & \ldots & R(0)\end{bmatrix}\begin{bmatrix} c_1\\ c_2\\ c_3\\ \vdots\\ c_n\end{bmatrix}=\begin{bmatrix} R(1)\\ R(2)\\ R(3)\\ \vdots\\ R(n)\end{bmatrix},$$

con los n coeficientes c_j como incógnitas. Ésto se puede escribir de forma compacta como $\mathbf{R}C=P$ y puede ser fácilmente resuelto mediante la eliminación Gaussiana o invirtiendo la matriz $\mathbf{R}$. La cuestión es que la inversión de la matriz requiere en general $O\left(n^3\right)$ operaciones, pero el nuestro no es un caso general, porque nuestra matriz $\mathbf{R}$ es especial. Es fácil ver que cada diagonal de $\mathbf{R}$ está formada por elementos idénticos. Dicha matriz se denomina matriz de Toeplitz, debido a su creador, Otto Toeplitz, y puede ser invertida mediante cierto número de algoritmos especializados y eficientes. Además, nuestra $\mathbf{R}$ también es simétrica.

Otto Toeplitz (1881–1940) procedía de una familia judía de la que salieron varios profesores de matemáticas. Tanto su padre, Emil Toeplitz, como su abuelo, Julio Toeplitz, enseñaban matemáticas en un instituto y también publicaron artículos matemáticos. Otto fue criado en Breslau y asistió a un instituto en esa ciudad. Sus antecedentes familiares hicieron que fuera natural que él también estudiara matemáticas.

Durante su larga y productiva carrera, Toeplitz estudió geometría algebraica, ecuaciones integrales y teoría espectral. Trabajó también en los procesos de sumatorios, espacios de dimensión infinita y funciones sobre ellos. En la década de 1930, desarrolló una teoría general de espacios de dimensión infinita y criticó el trabajo de Banach por ser demasiado abstracto.

Los operadores de Toeplitz y las matrices de Toeplitz llevan su nombre.

FLAC emplea un método recursivo y eficaz, conocido como algoritmo de Levinson-Durbin, para resolver este sistema de n ecuaciones. Este algoritmo fue propuesto por primera vez por Norman Levinson en 1947 [Levinson 47], mejorado por J. Durbin en 1960 [Durbin 60] y mejorado de nuevo por varios investigadores. En su forma actual, requiere sólo $3n^2$ multiplicaciones. Una descripción de este algoritmo está fuera del alcance de este libro, pero puede encontrarse, junto con una derivación, en [Parsons 87].

El autor desea dar las gracias a Josh Coalson, por su ayuda con esta sección.

7.11. WavPack

(Esta sección ha sido escrita por David Bryant, desarrollador de WavPack.)

WavPack [WavPack 06] es un algoritmo de compresión de audio multiplataforma completamente abierto y un software que soporta tres modos de compresión: sin pérdidas, con pérdidas de alta calidad , y un modo único de compresión híbrido. Maneja muestras de audio enteras de hasta a 32 bits de ancho y también datos en punto flotante IEEE de 32 bits [IEEE754 85]. El stream de entrada es dividido por WavPack en bloques que pueden ser mono o estéreo y son generalmente de 0,5 segundos de duración (pero la longitud es flexible en la realidad). Los bloques pueden ser combinados en secuencia por el codificador para manejar streams de audio multicanal. Todas las tasas de muestreo de audio están soportadas por WavPack en todos sus modos.

WavPack utiliza por defecto el modo sin pérdidas. En este modo, las muestras de audio son simplemente comprimidas en su resolución completa y no se descarta información alguna. WavPack generalmente proporciona un compromiso razonable entre la razón de compresión y la velocidad de codificación/decodificación. Sin embargo, para aplicaciones específicas, puede ser más deseable un compromiso alternativo. Por esta razón, WavPack incorpora un modo "rápido" opcional que es veloz y supone sólo una pequeña penalización en el rendimiento de la compresión y un modo "alto" que tiene como objetivo la compresión máxima (a un ritmo algo más lento).

El modo con pérdidas no emplea ninguna codificación de subbanda o de enmascaramiento de ruido psicoacústico. En su lugar, se basa en la cuantificación variable en el dominio del tiempo combinado con el modelado del ruido suave. Este modo puede operar desde tasas de bits tan bajas como 2,22 bits por muestra completamente sin pérdidas y ofrece una flexibilidad y un rendimiento considerablemente mayor que el similar, pero mucho más simple, ADPCM (Sección 7.6). Ésto hace del modo con pérdidas de calidad el ideal para codificación de audio de alta calidad donde el almacenamiento de datos o los requisitos de ancho de banda del modo sin pérdidas podría ser prohibitivo.

Por último, el modo híbrido combina los modos con y sin pérdidas en una sola operación. En lugar de un único archivo de salida generado a partir de una fuente, se generan dos archivos. El primero, con la extensión `wv`, es un pequeño archivo con pérdidas que puede ser reproducido independientemente. El segundo es un archivo de "rectificación" (WVC) que, cuando se combina con el primero, proporciona la restauración sin pérdidas del audio original. Los dos archivos juntos tienen esencialmente el mismo

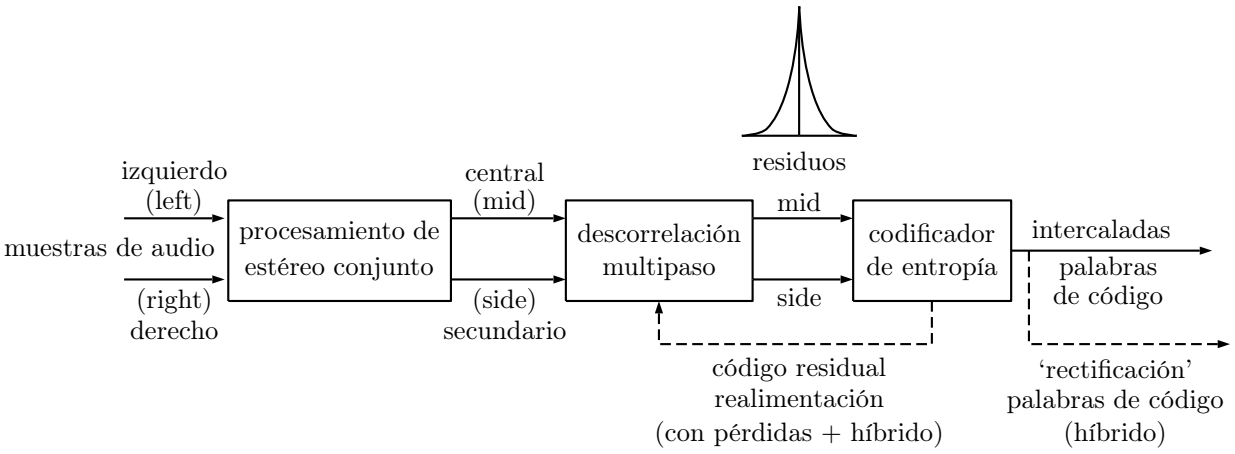


Figura 7.31: El codificador de WavPack.

tamaño que la salida producida por el modo sin pérdidas puro, por lo que el modo híbrido genera una pequeña sobrecarga adicional.

El hardware de decodificación eficiente fue uno de los objetivos de diseño de WavPack, y esto pesaba (junto con consideraciones de patentes) en las decisiones de diseño. Como resultado, incluso el modo de WavPack más exigente será decodificado fácilmente en un CD con calidad audio en tiempo real en un procesador ARM7 corriendo a 75 MHz (y los modos por defecto utilizan considerablemente menos que éso).

La Figura 7.31 es un diagrama de bloques del codificador WavPack. Sus partes principales son: el procesamiento de estéreo conjunto (eliminación de las correlaciones entre canales), la descorrelación multipaso (eliminación de correlaciones intra-canal entre muestras de audio vecinas), y el codificador de entropía.

Descorrelación

Los pasos de la descorrelación son prácticamente idénticos en los tres modos de WavPack. El primer paso en el proceso de descorrelación es la de convertir los canales estéreo izquierdo y derecho en los canales diferencia estándar y promedio (también referidos como secundario y central) mediante $s(k) = l(k) - r(k)$ y $s'(k) = \text{int}((l(k) + r(k))/2)$. Observe que la división entera (denotada int) por 2 descarta el bit menos significativo de la suma $l(k) + r(k)$, pero este bit puede ser reconstruido a partir de la diferencia, porque una suma $A + B$ y una diferencia $A - B$ tienen el mismo bit menos significativo.

El segundo paso de la descorrelación es la predicción. WavPack realiza múltiples pasadas de filtros predictivos lineales de clave única muy simples, que emplean el método signo-signo de LMS para la adaptación. (El método signo-signo de LMS [adaptiv9 06] es una variación del método estándar de las medias cuadráticas mínimas.) Tener sólo un factor de peso ajustable por filtro elimina los problemas de inestabilidad y la falta de convergencia que pueden ocurrir cuando se actualizan múltiples claves desde un único resultado. La complejidad general de cálculo del proceso está controlado sencillamente por el número total de pasadas de filtrado realizadas. El caso por defecto efectúa cinco pasadas, ya que ésto proporciona un buen compromiso de compresión vs. velocidad. En el modo "rápido", sólo realiza dos pasadas, mientras que el modo "alto" incorpora el máximo permitido de 16 pasadas.

Realmente, hay 13 variaciones del filtro según el valor de la muestra (o la función lineal de dos

valores de muestra) sea utilizado como valor de entrada $u(k)$ para el filtro de predicción. Estas variaciones son identificadas por un parámetro denominado "term" ("término" o "condición"). En el caso de un solo canal, el filtrado depende del valor de este parámetro como sigue:

$$u(k) = \begin{cases} 1 \leq \text{term} \leq 8, & s(k - \text{term}), \\ \text{term} = 17, & 2s(k-1) - s(k-2), \\ \text{term} = 18, & (3s(k-1) - s(k-2))/2{,}0, \end{cases}$$

donde $s(k)$ es la muestra a predecir y la historia de la muestra reciente es $s(k-1), s(k-2), \ldots, s(k-8)$.

Para el caso de dos canales (que puede ser izquierdo y derecho o central y secundario) hay otras tres variaciones de filtro disponibles que utilizan información multicanal:

$$\begin{aligned} &\text{si } (\text{term} = -1), u(k) = s'(k),\ u'(k) = s(k-1), \\ &\text{si } (\text{term} = -2), u(k) = s'(k-1),\ u'(k) = s(k), \\ &\text{si } (\text{term} = -3), u(k) = s'(k-1),\ u'(k) = s(k-1). \end{aligned}$$

Una vez que $u(k)$ ha sido determinado, un valor de error (residual o residuo) $e(k)$ es calculado como la diferencia $e(k) = s(k) - \omega(k)\, u(k)$, donde $\omega(k)$ es el valor del peso del filtro actual. El peso varía generalmente entre $+1$ para muestras altamente correlacionadas y 0 para muestras completamente no correlacionadas. También puede convertirse en negativo si hay una correlación negativa entre los valores de muestra. Existiría una correlación negativa si $(-1 \times u(k))$ fue mejor que $u(k)$ como un predictor de $s(k)$. Ésto sucede a menudo con muestras con grandes cantidades de altas frecuencias.

Finalmente, el peso es actualizado para la siguiente muestra basándose en los signos del filtro de entrada y del residuo $\omega(k+1) = \omega(k) + d \cdot \text{sgn}(u(k)) \cdot \text{sgn}(e(k))$, donde el parámetro d es el tamaño de paso de la adaptación, un pequeño entero positivo. (La función `sgn` usada por WavPack devuelve $+1$, 0, ó -1.) La regla para adaptar $\omega(k)$ implica que si, o bien $u(k)$, o bien $e(k)$ es cero, el peso no es actualizado. En los casos en que la entrada al filtro proceda del mismo canal que la muestra que se está siendo predicha (i.e., donde "term" es positivo), la magnitud del peso se autolimita y, por lo tanto, se le permite exceder en ± 1. Sin embargo, en los casos multicanal (donde "term" es negativo), el peso podría crecer indefinidamente, y en consecuencia se recorta a ± 1 tras cada adaptación.

He aquí un ejemplo de un paso de adaptación. Asumiendo que la entrada $u(k)$ al predictor es 105, la muestra $s(k)$ para ser predicha es 100, el peso del filtro actual $\omega(k)$ es 0,8, y el tamaño de paso actual d es 0,01, primero calculamos el residuo $e(k) = s(k) - \omega(k) \cdot u(k) = 100 - 105 \cdot 0{,}8 = 16$ y luego actualizamos el peso:

$$\begin{aligned} \omega(k+1) =& \omega(k) + s \cdot \text{sgn}(e(k)) \cdot \text{sgn}(u(k)) \\ =& 0{,}8 + 0{,}01 \cdot \text{sgn}(100) \cdot \text{sgn}(16) \\ =& 0{,}8 + 0{,}01 \cdot 1 \cdot 1 = 0{,}81. \end{aligned}$$

Podemos ver que si el peso original hubiera sido más positivo (mayor que 0,8), la predicción habría sido más cercana a $s(k)$ y el residuo habría sido más pequeño. Por consiguiente, incrementamos el peso en esta iteración.

La magnitud del tamaño de paso es un compromiso entre una adaptación rápida y un funcionamiento ruidoso en torno al valor óptimo. En otras palabras, si el tamaño de paso es grande, el filtro se ajustará rápidamente a los cambios bruscos en la señal, pero el peso saltará de un lado al otro cuando está cerca del valor correcto. Un tamaño de paso demasiado pequeño hará que el filtro se ajuste demasiado lentamente a los cambios rápidos en el audio.

En el modo sin pérdidas, los resultados de la descorrelación (los residuos $e(k)$) se pasan simplemente por el codificador de entropía para la conversión exacta. Por el contrario, en los modos híbridos con pérdidas, los valores descorrelacionados pueden codificarse inexactamente basados en un error máximo

permisible predefinido. En esta situación, el valor real codificado es devuelto desde el codificador de entropía y este valor se utiliza para actualizar el peso del filtro y generar el valor de la muestra reconstruida. Ésto es necesario debido a que durante la posterior decodificación, los valores exactos de las muestras no estarán disponibles, y debemos hacer funcionar el codificador con sólo la información que tengamos en ese momento.

Implementación

Todos los datos de muestra son enviados al descorrelacionador como enteros con signo de 32 bits, incluso si la fuente es más pequeña. Ésto se hace para permitir rutinas comunes, pero también se requiere desde un punto de vista aritmético porque, por ejemplo, una clave de datos de 16 bits podría saturar algunos pasos si se restringe el almacenamiento a 16 bits. Las rutinas están diseñadas de tal manera que no producirán desbordamiento en los datos de muestra hasta los 25 bits significativos. Ésto permite el estándar de audio de 24 bits y una mantisa de 25 bits en datos IEEE de 32 bits. Cualquier valor con mantisa mayor que ésta debe ser manejada por canales de datos adicionales.

Los pesos del filtro se representan en enteros con signo de 16 bits con 10 bits de fracción (por ejemplo $1024 = 000001\underbrace{0\ldots00}_{10}{}_2$ representa 1,0), y el ajuste del tamaño de paso puede variar de 1 a 7 unidades de esa resolución. Para valores de entrada que se ajustan en enteros con signo de 16 bits, es posible llevar a cabo la multiplicación del peso redondeado con una sola operación de $16\,\text{bits} \times 16\,\text{bits} = 32\,\text{bits}$ mediante `predicción = muestra * peso + 512) >> 10;`

Puesto que tanto "muestra" como "peso", están en formato de 16 bits sin signo, el signo del producto se ajustará a 32 bits. Añadimos 512 para redondear y luego desplazar el resultado a la derecha 10 lugares para compensar los 10 bits de fracción en la representación del peso. Esta operación está implementada muy eficazmente en los procesadores modernos.

Para aquellos casos en los que la muestra no se ajusta a enteros de 16 bits, necesitamos más de 32 bits en el resultado intermedio, y se proporcionan varios métodos alternativos en el código fuente. Si una operación de multiplicación se provee con 40 bits de resolución de producto (como en el ColdFire EMAC o varios chips DSP de Texas Instruments), entonces la operación anterior funcionará (aunque pueden requerirse algunos escalados de las entradas). Si sólo se proporcionan resultados de 32 bits, es posible llevar a cabo operaciones separadas para los 16 bits inferiores y superiores de la muestra de entrada y luego combinar los resultados con desplazamientos y una adición. Ésto se ha implementado de manera eficiente con las extensiones MMX al conjunto de instrucciones de `x86`. Finalmente, algunos procesadores proporcionan asistencia por hardware para la aritmética de punto flotante de doble precisión que puede fácilmente efectuar la operación con la precisión requerida.

Como se mencionó anteriormente, se usan múltiples pasadas del filtro de descorrelación. Para los modos de operación estándar (rápido, por defecto, y alto) el tamaño de paso es siempre 2 (que representa aproximadamente 0,002) y se emplean los siguientes tiempos (terms), en el orden que se indica:

$$\begin{aligned}\text{modo rápido terms} &= \{17, 17\}\,,\\ \text{modo por defecto terms} &= \{18, 18, 2, 3, -2\}\,,\\ \text{modo alto terms} &= \{18, 18, 2, 3, -2, 18, 2, 4, 7, 5, 3, 6, 8, -1, 18, 2\}\,.\end{aligned}$$

Para la operación de un solo canal, las pasadas con term negativo (i.e., pasadas con correlación multicanal) son omitidas. Tenga en cuenta, sin embargo, que el decodificador no se fija a cualquiera de estas configuraciones debido a que el número de pasadas de descorrelación, así como otra información tal como qué términos y tamaño de paso utilizar en cada pasada, es enviada al decodificador como información adicional al principio de cada bloque.

Resto a codificar	cuando $m = 6$	cuando $m = 7$	cuando $m = 8$	cuando $m = 9$	cuando $m = 10$	cuando $m = 11$
0	00	00	000	000	000	000
1	01	010	001	001	001	001
2	100	011	010	010	010	010
3	101	100	011	011	011	011
4	110	101	100	100	100	100
5	111	110	101	101	101	1010
6		111	110	110	1100	1011
7			111	1110	1101	1100
8				1111	1110	1101
9					1111	1110
10						1111

Tabla 7.32: Códigos binarios ajustados para seis valores de m.

Para el procesamiento sin pérdidas, es posible llevar a cabo cada pasada de descorrelación en un bucle separado que actúa sobre un array de muestras. Ésto permite implementar el proceso en pequeñas rutinas, altamente optimizadas. Desafortunadamente, esta técnica no es posible en el modo con pérdidas, porque los resultados de la codificación de entropía son inexactos y deben tenerse en cuenta para cada muestra. Sin embargo, en el lado de la decodificación, esta técnica puede ser utilizada para ambos modos, con y sin pérdidas (ya que son esencialmente idénticos desde el punto de vista del descorrelacionador).

Generalmente, las operaciones no son paralelizables porque la salida de una pasada es la entrada a la siguiente pasada. Sin embargo, en el modo bicanal, los dos canales pueden ejecutarse en paralelo para los valores de term positivos (y de nuevo ésto ha sido implementado en las instrucciones MMX).

Codificador de entropía

El paso de descorrelación genera residuos $e(k)$ que son números con signo (generalmente pequeños). Recordemos que la descorrelación es un proceso de múltiples pasos, donde las diferencias $e(k)$ calculadas por un paso se convierten en la entrada del siguiente paso. Debido a esta característica de WavPack, nos referimos a la salidas finales $e(k)$ del proceso de descorrelación como residuos. Los residuos se comprimen mediante el codificador de entropía. La secuencia de los residuos se caracteriza mejor como una distribución geométrica bilateral (*two-sided geometric distribution* o TSGD) centrada en cero (véase la Figura 2.9 para la distribución geométrica unilateral estándar). Los códigos de Golomb (Sección 2.5) proporcionan un método sencillo para codificar de manera óptima distribuciones geométricas unilaterales similares. Este código depende de un único parámetro entero positivo m. Para la codificación de Golomb de cualquier número entero no negativo n, primero dividimos n en una magnitud $\texttt{mag} = \text{int}\,(n/m)$ y un resto $\texttt{rem} = n \bmod m$, luego codificamos la magnitud como prefijo unario (i.e., `mag` 1's seguidos por un solo 0) y seguimos con una representación del resto en código binario ajustado. Si m es una potencia entera de 2 ($m = 2^k$), entonces el resto simplemente es codificado como el valor binario de rem usando k bits. Si m no es una potencia entera de 2, entonces el resto se codifica en k o $k + 1$ bits donde $2^k < m < 2^{k+1}$. Debido a que los valores de la parte baja del rango son más probables que los valores en la parte alta, las palabras de código más cortas están reservadas para los valores más pequeños.

La Tabla 7.32 muestra los códigos binarios ajustados para valores de m de 6 a 11.

Los códigos de Rice son una simplificación común de este método, donde m es una potencia de 2. Ésto elimina la necesidad de los códigos binarios ajustados y elimina la división requerida por

los códigos de Golomb generales (porque la división por una potencia de 2 se implementa como un desplazamiento). Los códigos de Rice son comúnmente utilizados por los algoritmos de compresión sin pérdidas de audio para codificar los errores de predicción. Sin embargo, los códigos de Rice son menos eficientes cuando el m óptimo está en la mitad de dos potencias consecutivas de 2. También son menos adecuados para la codificación con pérdidas debido a las discontinuidades resultantes por grandes saltos en el valor de m. Por estas razones, los códigos de Rice no fueron elegidos.

Antes de codificar un residuo $e(k)$, se convierte a un valor no negativo por medio de: `if` $e(k) < 0,\ e(k) = -(e(k)+1)$. El bit de signo original de $e(k)$ es agregado al final del valor codificado. Observe que el bit de signo siempre se escribe en el stream comprimido, incluso cuando el valor a codificar es cero, porque -1 también se codifica como cero. Ésto es menos eficiente para valores codificados muy pequeños, donde la probabilidad de aparición del valor cero es significativamente más alto que los valores vecinos. Sin embargo, sabemos por experiencia que un archivo de audio puede tener largas rachas (runs) de muestras de audio cero consecutivas (silencio), pero relativamente pocas muestras de audio cero aisladas. Por consiguiente, nuestro método elegido es ligeramente más eficiente cuando el cero no es significativamente más común que sus vecinos. Más tarde mostramos que WavPack codifica los runs de residuos cero con un código de Elias.

El método más simple para la determinación de m es dividir los residuos en bloques y determinar el valor de m que codifica el bloque en el menor número de bits. Los algoritmos de compresión que emplean este método tienen que enviar el valor de m al decodificador como información adicional. Es incluído en el stream comprimido antes de codificar las muestras. Manteniendo la filosofía general de WavPack, en su lugar implementamos un método que ajuste dinámicamente m para cada muestra de audio.

La discusión de la página 71 (y especialmente la Ecuación (2.2)) muestra que el mejor valor de m es la mediana de los valores de las muestras recientes. Por lo tanto, intentamos adaptar m directamente a partir del residuo actual $e(k)$ usando la sencilla expresión:

$$\begin{aligned} \texttt{if}\ (e(k) \geq m(k))\ \texttt{then}\, m(k+1) &= m(k) + \operatorname{int}\left[\frac{m(k)+127}{128}\right], \\ \texttt{else}\, m(k+1) &= m(k) - \operatorname{int}\left[\frac{m(k)+126}{128}\right]. \end{aligned} \tag{7.10}$$

La idea es que la media acual $m(k)$ será incrementada en una pequeña cantidad cuando se produzca un residuo $e(k)$ por encima de ella y será decrementada en una cantidad casi idéntica cuando el residuo se produzca por debajo de ella. Los distintos desplazamientos (126 y 127) para los dos casos fueron seleccionados de manera que el valor de la mediana puede moverse hacia arriba desde 1, pero nunca baja de 1. Ésto funciona muy bien para los valores grandes, pero debido a que la mediana es simplemente un entero, tiende a ser "saltarín" con los valores pequeños, ya que debe moverse al menos una unidad en cada paso. Por esta razón, la implementación real añade cuatro bits de fracción a la mediana que son simplemente ignorados cuando el valor es usado en una comparación. De esta manera, se puede mover suavemente por los valores más pequeños.

Es interesante observar que el valor inicial de la mediana no es crucial. Sin tener en cuenta su valor inicial, m se desplaza hacia la mediana de los valores que se presentan. En WavPack, m se inicializa al comienzo de cada bloque con el valor con que se quedó en el bloque anterior, pero incluso si se ha iniciado a 1 seguiría trabajando (pero tomaría un tiempo alcanzar la mediana).

Existe un problema al utilizar un valor adaptativo de la mediana de esta manera. A veces, llega un residuo, que es tan grande en comparación con la mediana que genera un código de Golomb con un vasto número de 1's. Por ejemplo, si m es igual a 1 y se produce un residuo de 24 000, entonces el código de Golomb para la muestra daría lugar a `mag` $= \operatorname{int}(24\,000/1) = 24\,000$ y `rem` $= 24\,000 \bmod 1 = 0$, y por tanto, comenzaría con 24 000 1's, el equivalente a 3 000 bytes de todo 1's. Para evitar tal situación y evitar códigos de Golomb ridículamente largos, WavPack sólo genera códigos de Golomb con 15 ó

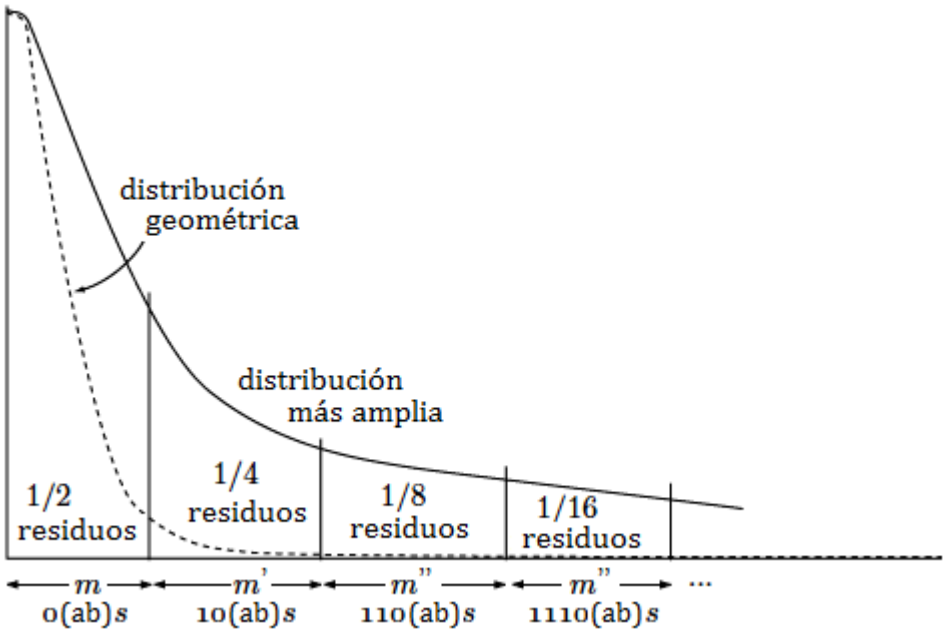


Figura 7.33: Áreas medias para la codificación de Golomb recursiva.

menos 1's consecutivos. Si un código de Golomb para un residuo $e\,(k)$ requiere k 1's consecutivos, donde k es mayor o igual que 16, entonces WavPack codifica $e\,(k)$ generando 16 1's, seguidos por el código gamma de Elias de k (código C_1 de la Tabla 2.6). Cuando los residuos se distribuyen geométricamente, esta situación es sumamente rara, por lo que no afecta a la eficiencia, pero sí previene los casos extraordinarios y muy poco frecuentes en los que se reduce drásticamente la compresión.

Recordemos que cada valor codificado es seguido por el bit de signo del residuo original $e\,(k)$. En consecuencia, una muestra de audio cero es ineficazmente codificada como los dos bits 00. Debido a ésto, WavPack emplea un método especial para rachas (*runs*) de residuos cero consecutivas. Un run de k ceros se codifica en el mismo código gamma de Elias precedido por un único 1 para distinguirlo de un código regular (ésto se hace sólo cuando las medianas son muy pequeñas a evitar el desperdicio de un bit para cada muestra.)

Una fuente de eficiencia inferior a la óptima con el método descrito hasta ahora es que a veces los datos no se distribuyen de una manera exactamente geométrica. La decadencia de los valores más altos puede ser más rápida o más lenta que exponencial, ya sea porque el descorrelacionador es no es perfecto, ya sea porque los datos de audio tienen características de distribución fuera de lo habitual. En estos casos, puede ocurrir que la mediana estimada dinámicamente no sea un buen valor para m, o quizás ningún valor de m puede producir códigos óptimos. Para estos casos, se ha añadido un mayor refinamiento que he llamado codificación de Golomb recursiva.

En este método, calculamos la primera mediana como en la Ecuación (7.10) y la denotamos por m. Sin embargo, para valores por encima de la mediana restamos m del valor y calculamos una nueva mediana, llamada m', que representa el punto $3/4$ de la población de muestras. Finalmente, calculamos una tercera mediana (m'') para los valores de las muestras que se encuentran por encima de la segunda mediana (m'), y usamos ésto para todas las particiones restantes (Figura 7.33). Las tres primeras áreas de codificación aún representan $1/2$, $1/4$, y $1/8$ de los residuos, pero no abarcan necesariamente el mismo número de valores posibles como sucedería con un solo m. No obstante, ésto es no es un problema para el decodificador, ya que conoce todas las medianas y, por lo tanto, todos las amplitudes de tamaños.

En la práctica, la codificación de Golomb recursiva funciona así. Si el residuo que está siendo codificado se encuentra en el primer área, bien. Si no, entonces se resta m del residuo y se supone que empezamos todo de nuevo. En otras palabras, primero dividimos los residuos en dos grupos iguales (inferior a m y superior a m). Luego, restamos m del grupo "superior a m" (por lo que comienza de nuevo en cero) y dividimos aquellos en dos grupos. A continuación, hacemos ésto una vez más. De esta manera, está garantizado que las primeras áreas tengan la distribución adecuada porque calculamos las medianas segunda y tercera directamente en lugar de asumir que son iguales en anchura a la primera. La Tabla 7.34 muestra algunas de las primeras áreas donde (ab) representa la codificación binaria ajustada del resto (residuo módulo m) y S representa el bit de signo.

rango	prob.	codificación
$0 \leq \text{residuo} < m$	$1/2$	$0\,(ab)\,S$
$0 \leq \text{residuo} < m + m'$	$1/4$	$10\,(ab)\,S$
$m + m' \leq \text{residuo} < m + m' + m''$	$1/8$	$110\,(ab)\,S$
$m + m' + m'' \leq \text{residuo} < m + m' + 2m''$	$1/16$	$1110\,(ab)\,S$
...		

Tabla 7.34: Probabilidades para la codificación de Golomb recursiva.

Codificación con pérdidas e híbrida

La codificación de Golomb recursiva permite que los arbitrarios enteros de la TSGD sean simple y eficientemente codificados, incluso cuando la distribución no es perfectamente exponencial y aun cuando contenga runs de ceros. Para la implementación de la codificación con pérdidas, es posible transmitir simplemente la magnitud unaria y el bit de signo, ignorando del total de datos restante. En el lado de la decodificación, el valor central de un rango de magnitud especificado es elegido como valor de salida. Ésto produce un promedio de tres bits por residuo como se ilustra la Tabla 7.35. La tabla se basa en la suma infinita

$$1 + \sum_{n=1}^{\infty} \frac{n}{2^n},$$

donde el 1 representa el bit de signo y cada término en la suma es una probabilidad multiplicada por el número de bits para la probabilidad. La suma infinita suma no más de dos bits, y el total, después de añadir el bit de signo, es de tres bits.

	prob.	bits enviados	# de bits	(# de bits) × prob.
	$1/2$	$0S$	2	$2/2$
	$1/4$	$10S$	3	$3/4$
	$1/8$	$110S$	4	$4/8$
	$1/16$	$1110S$	5	$5/16$
	$1/32$	$11110S$	6	$6/32$
	$1/64$	$111110S$	7	$7/64$
	...	...	...	...
total	1			3

Tabla 7.35: Tres bits por residuo.

Ésto funciona bien. Sin embargo, se desea una tasa de bits mínima más baja y por tanto se emplea un nuevo esquema que divide las muestras en 5:2 en lugar de a la mitad 1:1. En otras palabras, se

símbolo	frec.	$\log_2(1/\text{frec.})$
0	$5/7$	0,485427
1	$2/7$	1,807355

(a)

secuencia de entrada	frec.	# de bits ideal	secuencia de salida
00	$25/49$	0,970854	0
01	$10/49$	2,292782	10
1	$14/49$	1,807355	11

(b)

Tabla 7.36: Conversión óptima de magnitudes unarias.

encuentra una "mediana" que tiene cinco residuos por debajo por cada dos que están por encima. Ésto se logra fácilmente modificando el procedimiento anterior así:

$$\begin{aligned}\texttt{if } (s(k) > m(k)) \texttt{ then } m(k+1) &= m(k) + 5 \cdot \text{int}\left[\frac{m(k)+127}{128}\right], \\ \texttt{else } m(k+1) &= m(k) - 2 \cdot \text{int}\left[\frac{m(k)+125}{128}\right].\end{aligned}$$

La idea es que la mediana es impulsada hacia arriba hasta cinco unidades cada vez que el valor es más alto, e impulsada hacia abajo dos unidades cuando es inferior. Si hay dos subidas por cada cinco bajadas, la mediana permanece estacionaria. Debido a que los movimientos de la mediana bajan un mínimo de dos unidades, consideramos que 1 ó 2 sea el mínimo (no va a ir por debajo de éste).

Ahora, la primera región contiene $5/7$ de los residuos, el segundo tiene un $10/49$, el tercero $20/343$, y así sucesivamente. Usando estas zonas medias agrandadas, la tasa de bits mínima se cae de 3 a 2,4 bits por residual, como se muestra mediante la suma infinita siguiente:

$$1 + \sum_{n=1}^{\infty} \frac{5n \times 2^{n-1}}{7^n}.$$

Hay una pérdida de eficiencia con estos códigos unarios porque el número de 1's y 0's en el stream de datos ya no es el mismo. Hay un único 0 para cada residuo, pero puesto que hay sólo 1,4 bits totales por residuo (sin contar el bit de signo), el promedio debe ser solamente de 0,4 1's por residuo. En otras palabras, la probabilidad para los 0's es de $5/7$ y la probabilidad para los 1's es $2/7$. Utilizando el método $\log_2$ para determinar el número ideal de bits necesario para representar esos símbolos, obtenemos los números que aparecen en la Tabla 7.36a y derivamos un esquema de codificación multibit óptimo (Tabla 7.36b).

La Tabla 7.37 muestra que el número medio de bits de magnitud unaria transmitidos se ha reducido por la conversión a $(73/84) \times 1{,}4 = 1{,}21666 \approx 1{,}22$ bits por muestra. Utilizamos el número 1,4 porque sólo los datos de magnitud unaria están sujetos a esta conversión; la adición de nuevo del bit de signo sin procesar nos lleva a cerca de 2,22 bits por muestra. Ésto representa para el codificador la ejecución con "rotundas" pérdidas (i.e., ningún resto es enviado) sin embargo, el método es tan eficiente que se utiliza para todos los modos. En el caso sin pérdidas, la representación binaria completa del resto es insertada entre los datos de magnitud y los bits de signo de terminación.

En la práctica, normalmente no deseamos que el codificador se ejecute su tasa de bits con pérdidas mínimas hasta su perfección. En lugar de eso, generalmente nos gustaría llegar a algún lugar entre el modo medio completo con pérdidas y el modo sin pérdidas para que podamos enviar algunos (pero no todos) de los restantes valores. Específicamente, se desea tener un error máximo permitido que pueda ajustarse durante la codificación para darnos una relación señal-a-ruido específica o una tasa de bits constante. El envío de sólo los datos suficientes para llegar a un límite de error establecido para cada

	secuencia de entrada	frec.	bits de entrada	entrada neta*	secuencia de salida	bits de salida	salida neta*
	00	$^{25}/_{49}$	2	$^{50}/_{49}$	0	1	$^{25}/_{49}$
	01	$^{10}/_{49}$	2	$^{20}/_{49}$	10	2	$^{20}/_{49}$
	1	$^{14}/_{49}$	1	$^{14}/_{49}$	11	2	$^{28}/_{49}$
totales		1		$^{84}/_{49}$			$^{73}/_{49}$

*Los valores "netos" son la frecuencia multiplicada por el número de bits.

Tabla 7.37: Ahorros netos para la conversión unaria.

muestra (y no más) es óptimo para este tipo de cuantificación porque el ruido percibido es equivalente al nivel de RMS de los valores de error.

Para lograr ésto, tenemos que hacer que el codificador calcule el rango de posibles valores de muestra que satisfacen la información de la magnitud que se acaba de enviar. Si este rango es menor que el límite de error deseado, entonces hemos terminado y ningún dato adicional es enviado para esa muestra. De lo contrario, se envían bits consecutivos al stream de datos, cada uno estrechando el rango posible a la mitad (i.e., 0 = mitad inferior y 1 = mitad superior) hasta que el tamaño del rango es menor que el límite de error. Este proceso binario es realizado al unísono por el decodificador, para permanecer sincronizado con el stream de bits.

En este punto, debería quedar claro cómo el modo híbrido sin pérdidas divide el stream de bits en una parte con pérdidas y una parte de rectificación. Una vez que el límite de error especificado ha sido alcanzado, la posición numérica del valor de la muestra en el rango restante es codificado usando el código binario ajustado descrito anteriormente, pero estos datos se dirigen hacia el stream de "rectificación" en lugar de al stream de bits principal. Una vez más, en la decodificación, el proceso es revertido exactamente, de éste modo el decodificador puede permanecer sincronizado con los dos streams de bits.

Con respecto al codificador de entropía, la sobrecarga asociada con tener dos archivos en lugar de uno es insignificante. El único costo es la diferencia resultante de la codificación de la porción con pérdidas del resto con el método de división binaria en lugar del método binario ajustado. El resto del costo procede de parte del descorrelacionador donde debe enfrentarse con una señal más ruidosa.

Más información profundizando en la teoría y la implementación de WavPack puede encontrarse en [WavPack 06].

7.12. Audio (de) monkey

Audio monkey es un algoritmo de compresión de audio sin pérdidas rápido, eficiente y libre, y una implementación (para el sistema operativo Windows) de Matt Ashland [monkeyaudio 06]. Este método también tiene las siguientes características importantes:

- Detección de errores. La implementación incorpora CRCs en el stream comprimido para detectar prácticamente todos los errores.
- Soporte de etiquetas. El algoritmo inserta etiquetas en el stream comprimido. Ésto hace que sea fácil para el usuario gestionar y catalogar una colección entera de archivos de audio comprimidos con monkey.
- Soporte externo. Audio monkey también puede ser usado como un front-end en un codificador de línea de comandos. La herramienta GUI actual permite utilizar diferentes codificadores de

línea de comandos mediante la creación de un sencillo script XML que describe cómo pasar parámetros y qué valores de retorno tener en cuenta.

- El código fuente está disponible gratuitamente. Los desarrolladores de software pueden usarlo (modificado o no) en otros programas sin restricciones.

El primer paso en audio monkey es transformar los canales estéreo izquierdo y derecho en los canales X (central, mitad, o promedio) e Y (secundario, lateral, o diferencia) de acuerdo con $X = (L + R)/2$ e $Y = L - R$. La intercorrelación entre L y R implica que el canal central X es similar a ambos, L y R, mientras que el canal secundario Y está formado por números pequeños. Observe que los elementos (coeficientes de transformación) de X e Y son números enteros, pero ya no son muestras de audio. Además, la transformada es reversible a pesar de que la división por 2 causa la pérdida del bit menos significativo (LSB) de X. Este bit puede ser reconstruido por el decodificador, ya que es idéntico al LSB de Y. (Observe que los bits menos significativos de cualquier suma $A + B$ y diferencia $A - B$ son iguales).

El siguiente paso es la predicción. Este paso aprovecha la intracorrelación entre elementos consecutivos en los canales central y secundario. Audio monkey emplea un método novedoso para predicción lineal que consiste en un predictor de primer orden fijo seguido por múltiples filtros de desplazamiento adaptativos. Existen diversos grados de filtrados de red neuronal, dependiendo del nivel de compresión —hasta tres filtros de redes neuronales consecutivos con ventanas de 1024 muestras en la compresión más alta—.

En su último paso, el audio monkey codifica los residuos con un codificador de tipo rango (Sección 2.15.1).

El stream comprimido es organizado en frames, donde cada frame tiene un CRC de 31 bits (un CRC estándar de 32 bits con el bit más significativo truncado). El codificador también calcula una suma de comprobación (*checksum*) MD5 para el archivo completo. El decodificador comienza verificando este checksum. Un fallo en la verificación indica un error, y el decodificador localiza el error verificando los CRCs de los frames individuales.

> El mejor compresor de audio sin pérdidas de todos... escrito por el mejor diablillo (monkey) de todos. ¡Gran trabajo, Matt!
>
> —Desde `http://www.ashlands.net/Links.htm`.

7.13. Codificación de audio sin pérdidas MPEG-4 (ALS)

La codificación de audio sin pérdidas MPEG-4 (ALS o *Audio Lossless Coding*) es la última adición a la familia de codificadores de audio MPEG-4. ALS puede introducir muestras de audio de punto flotante y se basa en una combinación de predicción lineal (tanto a corto como a largo plazo), codificación multicanal y una eficiente codificación de los residuos de audio mediante códigos de Rice y códigos de bloque (estos últimos son también conocidos como códigos de bloque de Gilbert-Moore (*Block Gilbert-Moore Codes* o BGMC [Gilbert y Moore 59] y [Reznik 04]). Debido a esta organización, ALS no se restringe a la codificación de señales de audio, y puede comprimir eficientemente y sin pérdidas otros tipos señales correlacionadas unidimensionales de tamaño fijo, tales como datos médicos (ECG-EKG y EEG) y sísmicos. Debido a su naturaleza sin pérdidas y su alta ganancia en compresión, sus desarrolladores previeron las siguientes aplicaciones de la compresión de audio para la ALS:

- Archivado (archivos de audio en radiodifusión, estudios de grabación, sellos discográficos, y bibliotecas)
- Operaciones de estudio (almacenamiento, trabajo en colaboración, y transferencia digital)

- Formatos de disco de alta resolución (CD, DVD, y formatos futuros)
- Distribución de archivos de audio por internet.
- Tiendas online de música (descarga de música comprada)
- Reproductores portátiles de música (una aplicación especialmente popular)

En la actualidad, no todas estas aplicaciones emplean ALS, porque muchos creen que la compresión de audio debe ser siempre con pérdidas y no tiene sentido preservar cada bit del audio, porque el oído no puede distinguir entre la reconstrucción del sonido con pérdidas y sin pérdidas. Existe un debate en curso acerca de la ventaja de los métodos de compresión de audio sin pérdidas (*lossy*) sobre aquellos con pérdidas (*lossless*), y si la compresión de audio sin pérdidas es realmente necesaria. Una búsqueda en Internet le retornará muchas opiniones fuertes, que van desde "Ningún otro amigo de la compresión sin pérdidas (*lossless-phile*) ¿Cuando se darán cuenta de que no tiene sentido la compresión de audio sin pérdidas?" a "Pasé todo el verano pasado convirtiendo mi colección de 200 Gb de audio a un formato con pérdidas, y ahora tengo que hacerlo de nuevo sin pérdidas, debido a la baja calidad de los mp3." Aquí hay dos opiniones más:

"Al tratar de codificar sin pérdida alguna obtuve archivos enormes. Por ejemplo, con sólo 50 minutos de música, las invenciones y sinfonías de Bach con Kenneth Gilbert generó un archivo de alrededor de 350 Mb con Mono. Al parecer, el sonido del clavicémbalo [clave] es tan rico en armónicos que crea ondas de sonido muy complejas (con un indicador de entropía muy alta —mis disculpas—, estoy traduciendo un término griego). Es por éso que muchos archivos mp3 de solos de clavicémbalos, no suenan tan agradables en mp3 (mientras que otros instrumentos con un sonido más 'redondo' como la flauta, suena muy bien)."

"Para mí realmente el clavicémbalo en mp3 suena mejor que otros instrumentos simplemente porque es rico en armónicos, la decodificación a mp3 elimina una gran cantidad de altas frecuencias; el clavicémbalo tiene muchos, por lo que aún suena bien; otros instrumentos, por ejemplo el piano, suena mucho peor, la voz humana también es muy vulnerable a la reducción de demasiadas altas frecuencias. Además, usted tiene que saber cómo hacer un buen mp3. Algunos mp3 a tasas de bits de 256K/s son horribles, aunque a veces un mp3 a 160K/s suena realmente bien. Por supuesto, tales bajas tasas de bits son casi siempre demasiado bajas para la música orquestal; en este caso sólo la compresión sin pérdidas es suficiente (si usted tiene un reproductor de CD decente y un buen software de extracción, por supuesto)."

La principal referencia para MPEG-4 ALS es [mpeg-4.als 06], un sitio Web mantenido por Tilman Liebchen, un desarrollador de este algoritmo y el editor estándar de MPEG-4 ALS. MPEG-4 ALS (en esta sección nos referiremos a ella simplemente como ALS) soporta cualquier tasa de muestreo de audio, muestras de audio de hasta 32 bits (incluyendo muestras en formato de punto flotante de 32 bits), y hasta 2^{16} canales de audio. Las pruebas realizadas por el equipo de desarrollo de ALS se describen en [Liebchen et al. 05] e indican que ALS, incluso en sus modos más simples, supera a FLAC.

Primero, un poco de historia. En julio de 2002, el comité MPEG publicó una convocatoria de propuestas para un algoritmo de codificación de audio sin pérdidas. En diciembre de 2002, siete organizaciones habían presentado codificadores que cumplían los requisitos básicos. Éstos fueron evaluados a principios de 2003 en términos de eficiencia de compresión, complejidad y flexibilidad. En marzo de 2003, el codificador propuesto por investigadores de la Universidad Técnica de Berlín fue seleccionado como el primer borrador de trabajo. Los siguientes dos años fueron testigos de un mayor desarrollo, seguido de su implementación y pruebas. Este trabajo culminó en las especificaciones técnicas de ALS, que se finalizaron en julio de 2005, y fueron adoptadas por el comité de MPEG y más tarde por la ISO. La designación formal de ALS es el estándar internacional ISO/IEC 14496-3:2005/Amd 2:2006.

La organización del codificador y del decodificador ALS se muestra en la Figura 7.38. Los datos de audio de origen se dividen en frames, cada frame se divide en canales de audio, y cada canal puede

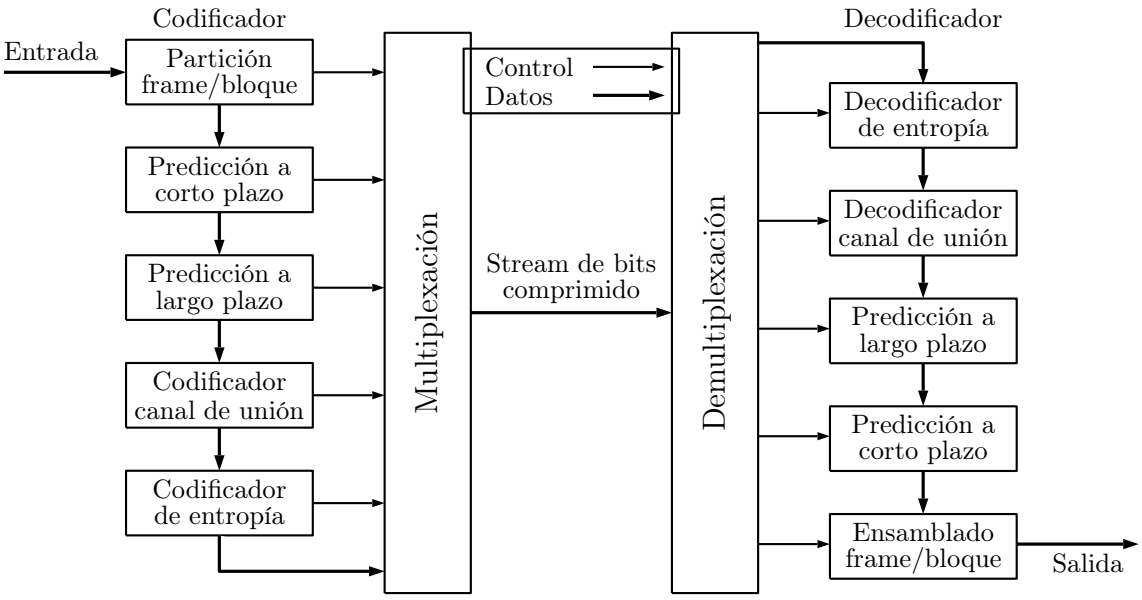


Figura 7.38: Organización del codificador y decodificador ALS.

dividirse en bloques. El codificador puede modificar los tamaños de bloque si ésto mejora la compresión. Los canales de audio pueden ser unidos ya sea restando pares de canales adyacentes, ya sea mediante la codificación multicanal, con el fin de reducir la redundancia entre canales. Las muestras de audio en cada bloque se predicen mediante una combinación de predictores de corto y largo plazo para generar los residuos. Los residuos son codificados por códigos de Rice y códigos de bloque de Gilbert-Moore que son escritos en el stream de salida. Este stream también incluye etiquetas que permiten el acceso aleatorio del audio en virtualmente cualquier punto. Otra característica opcional útil es una suma de comprobación CRC para una mejor integridad de la salida.

Predicción. La predicción lineal se utiliza comúnmente en codificadores de audio sin pérdidas y sus principios se han mencionado en otra parte de este libro (véanse las Secciones 4.26, 7.9 y 7.10.1). Un predictor lineal de respuesta de impulso finito (*finite-impulse-response* o FIR) de orden K predice la muestra de audio actual $x\,(n)$ a partir de sus K predecesores inmediatos, calculando la suma lineal

$$\hat{x}\,(n) = \sum_{k=1}^{K} h_k x\,(n-k)\,,$$

donde los h_k son los coeficientes a determinar. El residuo $e\,(n)$ se calcula como la diferencia $x\,(n) - \hat{x}\,(n)$ y después es codificada. Si la predicción se hace correctamente, los residuos están correlacionados y son también números pequeños, ideal para posteriormente codificarlos mediante un código de tamaño variable.

El decodificador obtiene $e\,(n)$ leyendo su código y decodificándolo. Luego calcula $\hat{x}\,(n)$ a partir de los coeficientes h_k. En consecuencia, el decodificador necesita conocer estos coeficientes y ésto puede conseguirse de tres maneras como sigue:

1. Los coeficientes son siempre iguales. Ésto se denomina predicción de n-ésimo orden y se discute en la Sección 7.9 para $n = 0, 1, 2$, y 3 y en la Sección 7.10.1, para $n = 4$.

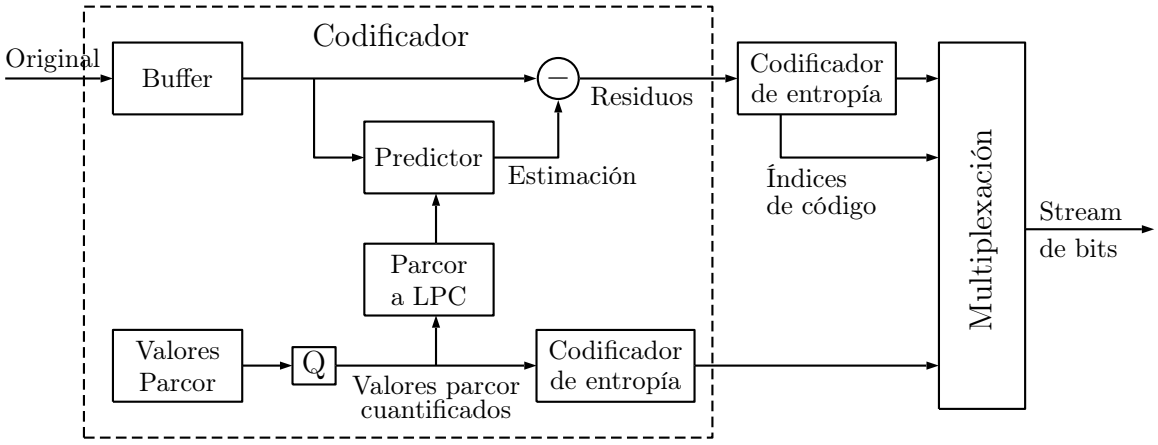


Figura 7.39: Predicción adaptativa hacia delante en ALS.

2. Los coeficientes se calculan basándose en las K muestras de audio que preceden $x(n)$ (ésto se denomina predicción hacia delante) y también pueden depender de algunos residuos anteriores a $e(n)$ (predicción hacia atrás). Estas muestras y los residuos son conocidos por el decodificador y éste puede calcular los coeficientes al unísono con el codificador.

3. Los coeficientes son calculados por el codificador basándose en todas las muestras de audio en el bloque actual. Cuando el decodificador se pone a decodificar la muestra de audio $x(n)$, tiene sólo las muestras anteriores a ella y no las siguientes $x(n)$. Por consiguiente, los coeficientes tienen que ser transmitidos en el stream de bits, como información adicional, para el uso del decodificador. Naturalmente, tienen que ser codificados eficientemente.

ALS adopta la predicción adaptativa hacia delante como en el punto 3 con una predicción de orden K de hasta 1023. La Figura 3.39 es un diagrama de bloques de los pasos de predicción del codificador ALS. Un bloque entero de muestras de audio es leído en un buffer, y el codificador ejecuta el algoritmo de Levinson-Durbin (como también lo hace FLAC, Sección 7.10). Este algoritmo (véase, por ejemplo, [Parsons 87] para una derivación) es recursivo, donde cada iteración incrementa el orden del filtro. El orden final K es conocido en la última iteración. El algoritmo produce un conjunto de coeficientes de filtro h_i que minimiza la varianza de los residuos para el bloque.

El algoritmo de Levinson-Durbin es adaptativo porque calcula un conjunto diferente de coeficientes de filtro y un orden K diferente para cada bloque, dependiendo de las variables estadísticas de las muestras de audio y del tamaño del bloque. El resultado es un conjunto de coeficientes de filtro que no sólo producen residuos más pequeños, sino que ellos mismos son pequeños, fáciles de comprimir, y por lo tanto minimizan la cantidad de información adicional en el stream de bits.

[El valor del predictor de orden K es importante. Un K grande reduce la varianza del error de predicción y produce una tasa de bits R_e más pequeña para los residuos codificados. Por otro lado, un orden de predicción más grande implica más coeficientes de filtro y consecuentemente una mayor tasa de bits R_c de coeficientes codificados (la información adicional). En consecuencia, un aspecto importante del algoritmo de Levinson-Durbin es determinar el valor de K que minimice la tasa de bits total $R(K) = R_e(K) + R_c(K)$. El algoritmo estima tanto $R_e(K)$ como $R_c(K)$ en cada iteración y se detiene cuando su suma ya no disminuye.]

Mientras calcula los coeficientes de filtro h_i en una iteración, el algoritmo de Levinson-Durbin también calcula un conjunto de k_i constantes referenciadas como coeficientes de correlación parcial

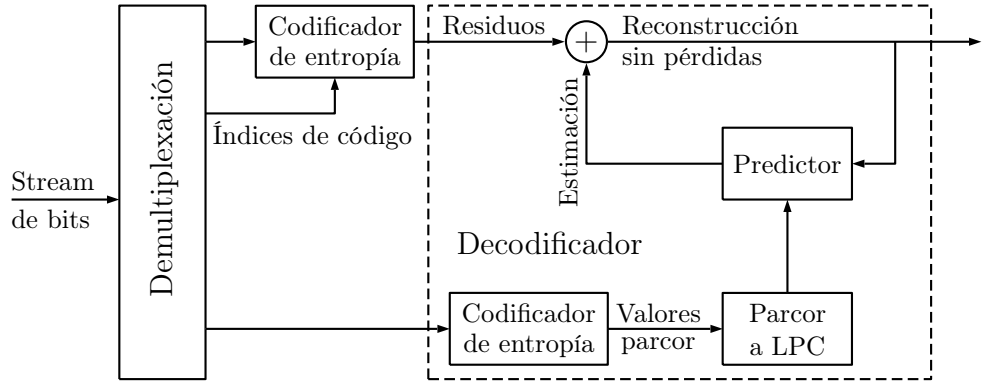


Figura 7.40: Decodificación de las predicciones en ALS.

(*parcor*) (también conocidos como coeficientes de reflexión, véase [Rabiner y Schafer 78]). Estos coeficientes tienen una propiedad interesante. Siempre que sus magnitudes son menores que 1, producen un filtro estable. Por lo tanto, es preferible cuantificar los coeficientes parcor k_i y utilizar los parcors cuantificados para calcular los coeficientes deseados h_i. Los parcors cuantificados son comprimidos y multiplexados en el stream de bits para ser utilizados posteriormente por el decodificador.

Los parcors se utilizan ahora para calcular un conjunto de coeficientes de filtro h_i que son a su vez usados para predecir todas las muestras de audio en el bloque. Las estimaciones predichas son restadas de las muestras reales, y los residuos resultantes son codificados y también multiplexados en el stream de bits.

El decodificador ALS (Figura 7.40) es mucho más simple, ya que no tiene que calcular ningún coeficiente. Para cada bloque, desmultiplexa los parcors codificados del stream de bits, los decodifica, y los convierte en coeficientes de filtro h_i que se utilizan después para calcular las estimaciones $\hat{x}(n)$ para el bloque entero. Cada residuo codificado es leído entonces desde el stream de bits, se decodifica, y se convierte en una muestra de audio. La complejidad de los cálculos en el decodificador depende sobre todo del orden K de la predicción y este orden varía de un bloque a otro.

Cuantificación. ALS cuantifica los parcors r_i, no los coeficientes de filtro h_i, porque es conocido (véase, por ejemplo, [Kleijn y Paliwal 95]) que estos últimos son muy sensibles incluso a pequeños errores de cuantificación. Con el fin de terminar con un filtro estable, los parcors son cuantificados en el intervalo $[-1,1]$ y el método de cuantificación tiene en cuenta dos hechos importantes: (1) Los parcors son menos sensibles a los errores de cuantificación que los coeficientes de filtro, pero los parcors cercanos a $+1$ ó -1 son sensibles. (2) Los dos primeros parcors r_1 y r_2, están normalmente cercanos a -1 y $+1$, respectivamente, mientras que los restantes parcors son normalmente más pequeños. En consecuencia, el principal problema de la cuantificación es con respecto a los dos primeros parcors y se resuelve mediante la sencilla expresión de compansión (companding):

$$C(r) = -1 + \sqrt{2(r+1)}.$$

La Figura 7.41 muestra el comportamiento de este compander. La Función $C(r)$ está cercana a la vertical en $r = -1$, lo que implica que es muy sensible a pequeños valores de r en torno a -1. Por lo tanto, $C(r)$ es un compander ideal para r_1. Su complemento, $-C(-r)$, está cercano a la vertical en $r = +1$ y por tanto, es un candidato ideal para compandir r_2. Con el fin de simplificar los cálculos, en realidad se utiliza $+C(-r_2)$ para compandir r_2, lo que lleva a un valor compandido de alrededor de -1. Después de la compansión, ambos parcors son cuantificados en los valores de siete bits (un bit de signo y seis bits de magnitud) a_1 y a_2 usando un cuantificador simple y uniforme. Los resultados

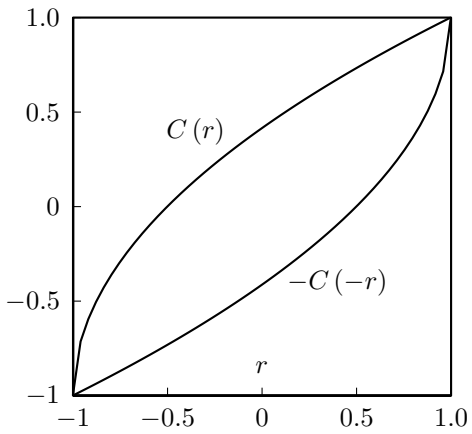


Figura 7.41: Funciones de compansión (*compander*) $C\,(r)$ y $-C\,(-r)$.

finales son:

$$a_1 = \left[64\left(-1+\sqrt{2\,(r_1+1)}\right)\right], \quad a_2 = \left[64\left(-1+\sqrt{2\,(-r_2+1)}\right)\right].$$

El restantes parcors r_i $(i > 2)$ no son compandidos, pero se cuantifican en siete bits mediante el cuantificador uniforme $a_i = [64r_i]$. Los parcors cuantificados de 7 bits a_i son con signo, lo que los coloca en el intervalo $[-64, +63]$. Este intervalo está centrado en cero, pero el promedio de los a_is puede ser distinto de cero. Para reducir su tamaño aún más, los a_is son re-centrados en torno a sus valores más probables y luego son codificados con códigos de Rice. El resultado es que cada parcor recentrado y cuantificado puede codificarse con aproximadamente cuatro bits, sin ninguna degradación notable del audio reconstruido.

La conversión de parcor a LPC mostrada en las Figuras 7.39 y 7.40 utiliza cálculos con enteros y, por lo tanto, puede ser reconstruida a su precisión completa tanto por el codificador como por el decodificador.

Tamaño del bloque de conmutación. Como se mencionó anteriormente, la entrada se divide en frames de tamaño fijo. El tamaño del frame pueden ser seleccionados por el usuario en función de la tasa de muestras de audio. Por ejemplo, si el usuario prefiere 43 ms de audio en un frame, entonces el tamaño del frame debe establecerse en 2 048 muestras si la frecuencia de muestreo es de 48 kHz, ó 4 096 muestras si la tasa de muestreo es de 96 kHz. Inicialmente, cada frame es un bloque, pero un codificador sofisticado puede subdividir un frame en bloques con el fin de adaptar la predicción lineal a las variaciones en la entrada de audio. Éste se conoce como bloque de conmutación. En general, los bloques largos con altos órdenes de predicción son preferibles para los segmentos estacionarios de audio, mientras que los bloques cortos con órdenes inferiores son adecuados para segmentos de audio transitorios.

Un frame de tamaño N puede subdividirse en bloques de tamaño $N/2$, $N/4$, $N/8$, $N/16$, y $N/32$, y la única restricción es que en cada paso de subdivisión un bloque de tamaño n debe ser dividido en dos bloques de tamaño $n/2$ cada uno. La Figura 7.42 muestra algunos ejemplos de subdivisiones de bloques válidas (a–d) e inválidas (e, f).

La ALS estándar no especifica cómo determina un codificador el bloque de conmutación. Los métodos pueden variar desde ningún bloque de conmutación, hasta la conmutación mediante la estimación de las características de la señal de audio, o hasta complejos algoritmos que ensayan muchas descomposiciones de bloques y seleccionan la mejor. El esquema de subdivisión de bloques (BS) seleccionado por el codificador tiene que ser transmitido al decodificador como información adicional, pero que no

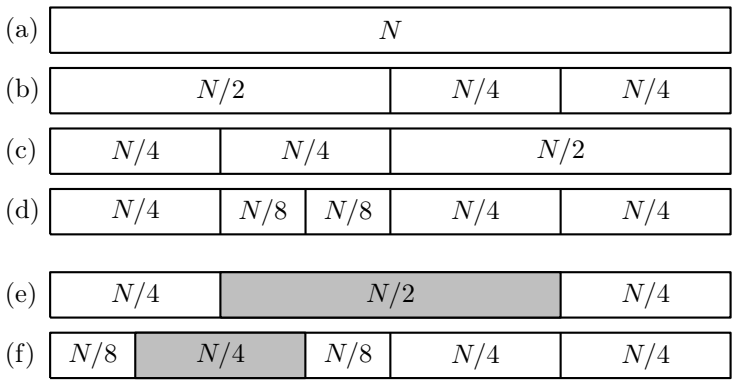


Figura 7.42: Ejemplos de subdivisión de bloques.

aumenta el trabajo del decodificador, ya que el decodificador aún tiene que procesar el mismo número de muestras de audio por frame, independientemente de cómo se haya subdividido el frame. Por consiguiente, el cambio de bloque puede ser empleado por un codificador sofisticado para aumentar significativamente la compresión, sin degradar el rendimiento del decodificador.

Debido a la regla que gobierna el bloque de conmutación, la información adicional BS enviada al decodificador es muy pequeña y no excede de 31 bits. La idea es comenzar con un bit que especifica si el frame de tamaño N ha sido dividido en dos bloques de tamaño $N/2$ cada uno. Si este bit es 1, es seguido por dos bits que indican si cualquiera de estos bloques ha sido dividido en dos bloques de tamaño $N/4$ cada uno. Si cualquiera de los dos bits es 1, va seguido de cuatro bits que indican cuál de los cuatro bloques de tamaño $N/4$, en su caso, ha sido dividido en dos bloques de tamaño $N/8$. Los cuatro bits pueden ser seguidos por ocho bits (por los bloques de tamaño $N/16$) y después por otros 16 bits (para los bloques de tamaño $N/32$), hasta un total de no más de 31 bits.

Como ejemplo, la estructura de bloques de la Figura 7.42c se describe con la BS de 31 bits 1 | 10 | 0100 | 00000000 | 0000000000000000. La representación más simple de la BS es enviar siempre los 31 bits al decodificador, pero puede conseguirse una pequeña ganancia precediendo BS con dos bits de "recuento" que especifican su longitud. Así es cómo puede efectuarse ésto. Nuestra BS consta de cinco campos de los cuales el primero es un único bit y debe existir siempre. Los cuatro campos restantes son necesarios sólo si el primer campo es 1. En consecuencia, son suficientes dos bits de recuento precediendo al primer bit de BS para especificar cuántos de los otros cuatro campos siguen al primer campo. Por ejemplo, $xx0$ significa cualesquier dos bits de recuento seguidos de un primer campo de 0, lo que indica ningún bloque de conmutación, mientras que 101 | 10 | 0100 indica dos campos (dos bits y cuatro bits) tras el primer bit a 1 (el campo de recuento es $10_2 = 2$).

Acceso aleatorio. Un algoritmo de compresión de audio práctico y amigable debe permitir al usuario saltar adelante y atrás hasta cualquier punto deseado en el stream de bits y decodificar el audio desde ese punto. Esta característica, conocida como acceso aleatorio o búsqueda, se implementa en ALS por medio de frames de acceso aleatorio. Normalmente, las primeras K muestras de audio de un frame son predichas mediante las muestras que preceden al frame, lo que impide al decodificador saltar al comienzo de un frame y decodificarlo sin primero efectuar la decodificación del frame anterior. Un frame de acceso aleatorio es un frame especial que puede decodificarse sin necesidad de decodificar el frame anterior. Suponiendo que un frame corresponde a 50 ms de audio, si cada décimo frame es generado por el codificador como un frame de acceso aleatorio, el decodificador puede saltar a puntos en el audio con una precisión de 500 ms, más que aceptable para aplicaciones de audición, edición, y transmisión (streaming) de audio normal. El tamaño de un frame y la distancia entre los frames de acceso aleatorio son parámetros controlados por el usuario. El último puede ser 1 a 255 frames.

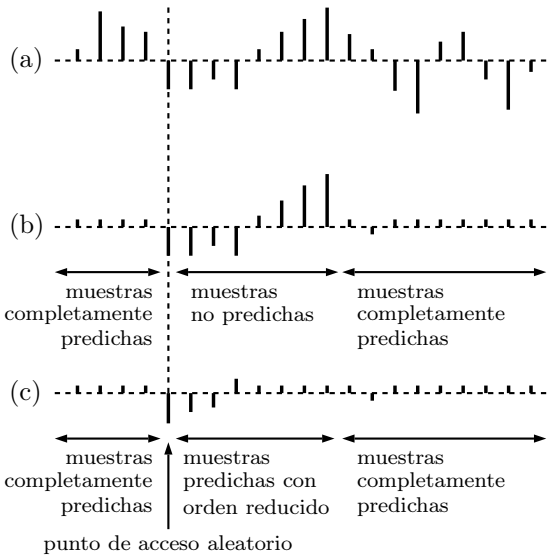


Figura 7.43: Predicción progresiva en ALS.

El único problema con el acceso aleatorio en ALS es la predicción de las primeras K muestras de audio de un frame de acceso aleatorio. Un predictor lineal de K-ésimo orden normalmente predice las primeras K muestras de un frame usando tantas muestras como sea necesario del frame previo, pero en un frame de acceso aleatorio, el predictor ALS utiliza sólo muestras del frame actual y tiene que predecir, por ejemplo, el cuarto frame a partir de las tres primeras muestras del frame. Ésto se denomina predicción progresiva. La primera muestra no se puede predecir. La segunda muestra se predice a partir de la primera (predicción de primer orden), la tercera muestra se predice a partir de las dos primeras (segundo orden), y así sucesivamente hasta que la muestra $K+1$ se predice a partir de sus K predecesoras en el mismo frame. La Figura 7.43 resume este proceso. La parte (a) muestra una secuencia típica de muestras de audio. La parte (b) muestra el resultado de la predicción continua (no progresiva), donde las primeras K muestras de un frame de acceso aleatorio no han sido predichas (por tanto, los residuos son iguales a las muestras originales). La parte (c) ilustra los resultados de predicción progresiva. Muestra cómo el primer residuo del frame de acceso aleatorio es idéntico a la primera muestra, pero los residuos sucesivos son cada vez más pequeños.

Predicción a largo plazo. Un píxel en una imagen digital está correlacionada con sus vecinos cercanos y una muestra de audio es correlacionada de forma similar con sus muestras vecinas, pero hay una diferencia entre la correlación en las imágenes y en el sonido. Muchos instrumentos musicales generan sonido que consta de una frecuencia fundamental y armónicos más altos que son múltiplos de esa frecuencia. El concepto de armónicos es fácil de entender cuando consideramos un instrumento de cuerda (pero también es válido para instrumentos de viento y de percusión). Una cuerda está atada al instrumento en ambos extremos. Cuando es punteada, la cuerda tiende a vibrar como se muestra en la Figura 7.44a donde toma la forma de una media onda. La frecuencia fundamental (o primeros armónicos) está por lo tanto asociada con una longitud de onda que es el doble de la longitud de la cuerda. Sin embargo, parte del tiempo, la cuerda también vibra como en la parte (b) de la figura, donde su centro es estacionario. Ésto crea la segunda frecuencia de armónicos, cuya longitud de onda asociada es igual a la longitud de la cuerda. Si la cuerda vibra como en la parte (c) de la figura, genera los terceros armónicos, con una longitud de onda que es igual a dos tercios de la cuerda. En general, la longitud de onda λ de los n-ésimos armónicos está relacionada con la longitud L de la

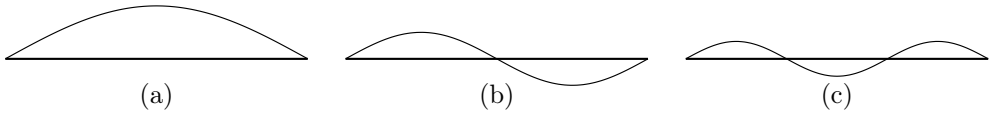


Figura 7.44: Frecuencias fundamentales y harmónicos.

cuerda mediante $\lambda = (2/n)\, L$, lo que significa que la frecuencia f de los n-ésimos armónicos viene dado por $f = s/\lambda = (s/2L)\, n$, donde s es la velocidad de la onda (observe que la velocidad depende del medio de la cuerda, pero no de la longitud de onda o de la frecuencia). Por consiguiente, las frecuencias de los armónicos más altos son múltiplos enteros de la frecuencia fundamental.

Normalmente no oímos los tonos armónicos como independientes, mayormente debido a que tienen amplitudes cada vez menores que la frecuencia fundamental, pero también porque nuestro sistema oído–cerebro se acostumbra a ellos desde una edad temprana y los combina con la frecuencia fundamental. Sin embargo, los armónicos más altos están normalmente presentes en el sonido generado por la voz o por los instrumentos musicales, y contribuyen a la riqueza del audio que escuchamos. Un tono "puro", con sólo una frecuencia fundamental y sin ninguna de las frecuencias más altas, puede ser generada mediante dispositivos electrónicos, pero suena artificial, tenue, y sin interés.

Por consiguiente, los armónicos existen en muchos streams de audio y contribuyen a la correlación entre las muestras de audio. Sin embargo, es fácil ver que esta correlación es de largo alcance. La frecuencia de la nota musical Do (*middle C*) es de 261 Hz. A una velocidad de muestreo de 48 kHz, una onda sinusoidal completa de Do corresponde a $48\,000/261 = 184$ muestras de audio, y a una frecuencia de muestreo de 192 kHz, la onda sinusoidal completa corresponde a $192\,000/261 = 736$ muestras de audio. En consecuencia, puede existir alguna correlación entre dos muestras de audio separadas por cientos de muestras, y ALS trata de aprovechar esta correlación.

ALS emplea un predictor a largo plazo (LTP) que comienza con un residuo de $e\,(n)$ y calcula un nuevo residuo $\tilde{e}\,(n)$ restando de $e\,(n)$ las contribuciones de cinco residuos $e\,(n-r+j)$ distantes pero relacionados. El parámetro r es el retraso. Éste tiene que ser determinado por el codificador dependiendo de la frecuencia actual de la entrada de audio y de la tasa de muestreo. El índice j varía sobre de cinco valores. La regla del cálculo es:

$$\tilde{e}\,(n) = e\,(n) - \left(\sum_{j=-2}^{2} \gamma_{r+j} \cdot e\,(n-r+j) \right),$$

donde los cinco coeficientes γ son los valores de ganancia cuantificados. Tanto r como los cinco valores de ganancia tienen que ser determinados por el codificador (usando un algoritmo sin especificar) y enviado al decodificador como información adicional. El residuo de largo plazo resultante $\tilde{e}\,(n)$ se codifica en lugar del residuo de corto plazo $e\,(n)$. [Nótese que también se utiliza $\tilde{e}\,(n)$ en vez de $e\,(n)$ para la predicción multicanal.]

El decodificador calcula un residuo de corto plazo $e\,(n)$ a partir de un residuo de largo plazo $\tilde{e}\,(n)$ por medio de:

$$e\,(n) = \tilde{e}\,(n) + \left(\sum_{j=-2}^{2} \gamma_{r+j} \cdot e\,(n-r+j) \right),$$

y luego emplea $e\,(n)$ y LPC para determinar la muestra de audio siguiente.

Codificación de conjuntos de canales. ALS puede soportar hasta 2^{16} canales de audio, por lo que es importante aprovechar cualquier posible correlación y dependencia entre canales. Dos esquemas de codificación multicanal están implementados como parte de ALS y pueden ser utilizadas.

1. **Codificación de diferencia**. A menudo, se sabe o sospecha que dos canales de audio $x_1(n)$ y $x_2(n)$ están correlacionados. Para sacar provecho de ésto, muchos métodos de compresión calculan la diferencia $d(n) = x_1(n) - x_2(n)$ y codifican el par $[d(n), x_1(n)]$ o el par $[d(n), x_2(n)]$. Un enfoque mucho mejor es utilizar LPC para calcular y codificar los residuos para los tres pares $00 = [x_1(n), x_2(n)]$, $01 = [d(n), x_1(n)]$, y $10 = [d(n), x_2(n)]$ y seleccionar el más pequeño de los tres residuos. Si ésto se realiza en una muestra mediante muestras básicas, entonces el decodificador tiene que ser informado de que de los tres pares fueron seleccionados para la muestra de audio i. Esta información requiere dos bits, pero sólo tres de las cuatro combinaciones de 2 bits son utilizadas. Por éso, esta información adicional debe ser comprimida. Podemos considerarla una secuencia de trits (dígitos ternarios), un trit para cada muestra de audio. Al final de un bloque, el codificador intenta comprimir la secuencia para el bloque ya sea mediante run length, ya sea contando el número de ocurrencias de 00, 01 y 10, y asignándoles tres códigos de Huffman apropiados.

2. **Codificación multicanal**. Un volcán en una remota isla del mar del Sur comienza a mostrar signos de una erupción inminente. Los geólogos van rápidamente al lugar y colocan muchos sismógrafos en y alrededor de la montaña. Las señales generadas por los sismógrafos están correlacionados y sirven como un ejemplo de muchos canales de muestras correlacionadas. Otro ejemplo de muchos canales correlacionados son las señales biomédicas obtenidas de un paciente por varios dispositivos de medición.

   El principio utilizado por ALS para codificar muchos canales correlacionados es seleccionar un canal como referencia r y codificarlo independientemente prediciendo sus muestras, calculando los residuos $e^r(n)$, y codificándolos. Cualquier otro canal c es codificado mediante el cálculo de los residuos $e^c(n)$ y luego haciendo cada residuo más pequeño restando del mismo una combinación lineal de tres residuos correspondientes (tres derivaciones) $e^r(n-1)$, $e^r(n)$, y $e^r(n+1)$ del canal de referencia. El cálculo es:

$$\hat{e}^c(n) = e^c(n) - \left(\sum_{j=-1}^{1} \gamma_j \cdot e^r(n+j) \right),$$

   donde los tres coeficientes de ganancia γ_j se calculan resolviendo el sistema de ecuaciones

$$\gamma = \mathbf{X}^{-1} \cdot \mathbf{y}, \tag{7.11}$$

   donde

$$\begin{aligned}
\gamma &= (\gamma_{-1}, \gamma_0, \gamma_{+1})^T, \\
\mathbf{X} &= \begin{bmatrix} {e^r_{-1}}^T \cdot e^r_{-1} & {e^r_{-1}}^T \cdot e^r_0 & {e^r_{-1}}^T \cdot e^r_{+1} \\ {e^r_{-1}}^T \cdot e^r_0 & {e^r_0}^T \cdot e^r_0 & {e^r_0}^T \cdot e^r_{+1} \\ {e^r_{-1}}^T \cdot e^r_{+1} & {e^r_{-1}}^T \cdot e^r_{+1} & {e^r_{+1}}^T \cdot e^r_{+1} \end{bmatrix}, \\
\mathbf{y} &= \left(e^{cT} \cdot e^r_{-1}, e^{cT} \cdot e^r_0, e^{cT} \cdot e^r_{+1} \right), \\
e^c &= (e^c(0), e^c(1), e^c(2), \ldots, e^c(N-1))^T, \\
e^r_{-1} &= (e^r(0), e^r(1), e^r(2), \ldots, e^r(N-1))^T, \\
e^r_{+1} &= (e^r(1), e^r(2), e^r(3), \ldots, e^r(N))^T.
\end{aligned}$$

   (Donde N es el tamaño del frame y T denota una transposición.) Los residuos resultantes $\hat{e}^c(n)$ son posteriormente codificados y escritos en el stream de bits.

El decodificador reconstruye la señal residual original aplicando la operación inversa:

$$e^c(n) = \hat{e}^c(n) + \left(\sum_{j=-1}^{1} \gamma_j \cdot e^r(n+j) \right).$$

También hay un modo de codificación multicanal que emplea seis derivaciones para la predicción de largo plazo. Los residuos $e^c(n)$ de una codificación de canal son calculados y entonces se hacen más pequeños restando una combinación lineal de los seis residuos del canal de referencia. Éstos incluyen los tres residuos $e^r(n-1)$, $e^r(n)$, y $e^r(n+1)$ que corresponden a $e^c(n)$ y otro conjunto de tres residuos consecutivos a una distancia r de n, donde el parámetro de retardo r es calculado por el codificador mediante una correlación cruzada entre el canal codificado y el canal de referencia. El cálculo es:

$$\hat{e}^c(n) = e^c(n) - \left(\sum_{j=-1}^{1} \gamma_j \cdot e^r(n+j) + \sum_{j=-1}^{1} \gamma_{r+j} \cdot e^r(n+r+j) \right).$$

Los seis parámetros γ de ganancia pueden obtenerse minimizando la energía de los seis residuos restados, de forma similar a la ecuación (7.11). El decodificador, como de costumbre, reconstruye los residuos originales de acuerdo con el cálculo similar:

$$e^c(n) = \hat{e}^c(n) - \left(\sum_{j=-1}^{1} \gamma_j \cdot e^r(n+j) + \sum_{j=-1}^{1} \gamma_{r+j} \cdot e^r(n+r+j) \right).$$

Una vez que la señal residual reconstruida $e^c(n)$ es obtenida por el decodificador, se utiliza por el predictor lineal de corto y/o largo plazo para obtener la muestra de audio reconstruida.

Codificación de los residuos. Dos modos, simple y avanzado, están disponibles para el codificador ALS para codificar los residuos $e(n)$. En el modo simple, sólo se utilizan códigos de Rice. El codificador tiene que determinar el parámetro de Rice, y un codificador sofisticado selecciona el mejor parámetro calculando la distribución de los residuos. Una opción en el modo sencillo consiste en utilizar el mismo parámetro de Rice para todos los residuos en un bloque. Una alternativa es particionar un bloque en cuatro partes y usar un parámetro de Rice diferente para codificar los residuos en cada parte. Los parámetros de Rice tienen que ser enviados al decodificador como información adicional tal como muestra la Figura 7.39.

El modo avanzado determina un valor $e_{\text{máx}}$ y codifica cada residuo $e(n)$ de acuerdo con su tamaño relativo a $e_{\text{máx}}$ como muestra la Figura 7.45. Los residuos que satisfacen $|e(n)| < e_{\text{máx}}$ se encuentran en la región central de la distribución de residuos y se codifican mediante una versión especial de códigos de bloque de Gilbert-Moore (BGMC, [Gilbert y Moore 59] y [Reznik 04]). Cada residuo se divide en una parte más significativa que es codificada con BGMC y una parte menos significativa que es escrita en formato raw (de longitud fija) en el stream de bits. La referencia [Reznik 04] tiene más información sobre la forma en que el codificador puede seleccionar valores razonables para $e_{\text{máx}}$ y sobre el número de bits menos significativos. Los residuos que satisfacen $|e(n)| \geq e_{\text{máx}}$ están localizados en las extremos de la distribución de residuos. Primero son recentrados mediante $e_t(n) = e(n) - e_{\text{máx}}$ y luego codificados en códigos de Rice como en el modo simple.

Muestras de audio en punto (o coma) flotante. Se ha mencionado anteriormente que ALS soporta señales de audio en formato de punto flotante, específicamente, en formato IEEE de 32 bits [IEEE754 85]. El codificador separa la secuencia $\mathbf{X}$ de muestras en punto flotante de un bloque en la

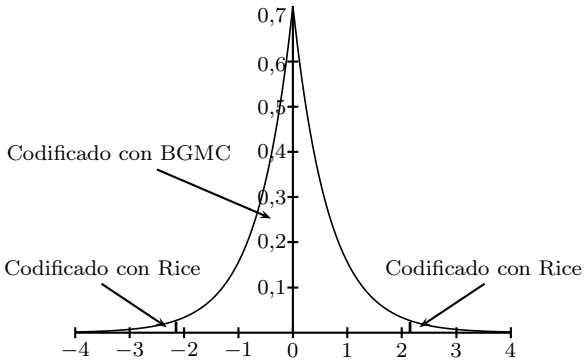


Figura 7.45: Codificación avanzada de residuos.

suma de una secuencia entera **Y** y una secuencia **Z** de pequeños números en punto flotante. La secuencia **Y** de valores enteros se comprime en la forma habitual, mientras la secuencia **Z** es comprimida mediante un algoritmo denominado utilidad Lempel-Ziv enmascarado, una variante de LZW.

La secuencia original **X** de muestras de audio en punto flotante es separada en la forma $\mathbf{X} = A \otimes \mathbf{Y} + \mathbf{Z}$, donde **X**, **Y**, y **Z** son vectores, $\otimes$ denota la multiplicación con redondeo y A es un multiplicador común, un número en coma flotante en el intervalo $[1, 2)$. El principal problema es seleccionar un multiplicador común que reduzca al mínimo los valores en punto flotante en el vector **Z**. El estándar ALS propone un enfoque para este problema, pero no garantiza que este enfoque es el mejor. Para una discusión detallada de este punto como así como la utilidad Lempel-Ziv enmascarado, véase [Liebchen et al. 05].

7.14. Reproductores de audio MPEG-1/2

Los aspectos de la compresión de vídeo de los estándares MPEG-1 y MPEG-2 se describen en la Sección 6.5. Sus principios de compresión de audio se discuten aquí. Se recomienda a los lectores leer las Secciones 7.2 y 7.3 antes de intentar hacer frente a este material. Algunas referencias de este sección son: [Brandenburg y Stoll 94], [Pan 95], [Rao y Hwang 96], y [Shlien 94].

El nombre formal de MPEG-1 es *the international standard for moving picture video compression* (*el estándar internacional para la compresión de imágenes de vídeo en movimiento*), *IS 11172*. Consta de cinco partes, de las cuales la parte 3 [ISO/IEC 93] es la definición del algoritmo de compresión de audio. Al igual que otros estándares elaborados por la ITU y la ISO, el documento que describe MPEG-1 tiene secciones de carácter *normativo* y otras de carácter *informativo*. Una sección normativa forma parte de la especificación estándar. Está dirigida a implementadores, está escrita en un lenguaje preciso, y debe ser seguida estrictamente en la implementación del estándar en las plataformas informáticas actuales. Una sección informativa, por otra parte, ilustra conceptos que se tratan en otro lugar, explica las razones que han llevado a elegir ciertas opciones y decisiones, y contiene material de referencia. Un ejemplo de una sección normativa son las tablas de los distintos parámetros y de los códigos de Huffman utilizados en el audio MPEG. Un ejemplo de una sección informativa es el algoritmo utilizado por el audio MPEG para implementar un modelo psicoacústico. El modelo MPEG no requiere ningún algoritmo en particular, y un codificador MPEG puede utilizar cualquier método para implementar el modelo. Esta sección informativa se limita a describir las distintas alternativas.

Los formatos MPEG-1 y MPEG-2 (o, en breve, MPEG-1/2) de audio estándar especifican tres métodos de compresión llamados *layers* (*capas*) y designados I, II, y III. Las tres capas son parte del estándar MPEG-1. Una película comprimida con MPEG-1 utiliza una sola capa, y el número de capa

es especificado en el stream comprimido. Cualquiera de las capas puede ser usada para comprimir un archivo de audio sin video. Los módulos funcionales de las capas inferiores también son utilizados por las capas superiores, pero las capas superiores tienen características adicionales que producen una mejor compresión. Un aspecto interesante del diseño del estándar es que las capas forman una jerarquía en el sentido de que un decodificador de la capa III también puede decodificar archivos de audio comprimidos por las capas I o II.

El resultado de tener tres capas fue una creciente popularidad de la capa III. El codificador es extremadamente complejo, pero produce una compresión excelente, y ésto, combinado con el hecho de que el decodificador es mucho más simple, produjo al final de la década de 1990 una explosión de lo que popularmente se conoce como archivos de sonido `mp3`. Es fácil obtener legal y libremente un decodificador layer III y mucha música ya está codificada en layer III. Hasta ahora, este ha sido un gran éxito de la parte de audio del proyecto MPEG.

El estándar de audio MPEG [ISO/IEC 93] comienza con la descripción de la normativa del formato del stream comprimido para cada una de las tres capas. Sigue con una descripción de la normativa del decodificador; la descripción del codificador (es diferente para las tres capas) y de los dos modelos psicoacústicos siguientes, y es informativa; cualquier codificador que genere un stream comprimido correcto es un codificador MPEG válido. Existen apéndices (anexos) que discuten temas relacionados, tales como la protección contra errores.

En contraste con el video MPEG, donde están disponibles muchas fuentes de información, existe relativamente poca literatura técnica sobre el audio MPEG. Además de las referencias en otras partes de esta sección, se remite al lector al consorcio MPEG [MPEG 00]. Este sitio contiene una lista de otros recursos, y se actualiza cada cierto tiempo. Otro recurso es la Sociedad de Ingeniería de Audio (*Audio Engineering Society* o AES). La mayor parte de las ideas y técnicas utilizadas en el estándar de audio MPEG (y también en otros métodos de compresión de audio) fueron publicadas originalmente en las muchas actas del congreso de esta organización. Desafortunadamente, éstos no están disponibles libremente y tienen que ser adquiridos desde la AES.

La historia del audio MPEG-1 comienza en diciembre de 1988, cuando un grupo de expertos se reunió por primera vez en Hannover, Alemania, para discutir el tema. Durante 1989, fueron propuestos al menos 14 algoritmos por estos expertos. Ellos se unieron finalmente en cuatro grupos, conocidos hoy como ASPEC, ATAC, MUSICAM y SB-ADPCM. Éstos fueron probados y comparados en 1990, con resultados que llevaron a la idea de la adopción de las tres capas. Cada una de las tres capas es un método de compresión diferente, y que aumenta en complejidad (y en rendimiento) de la capa I a la capa III. El primer borrador del estándar de audio ISO MPEG-1 (estándar 11172, parte 3) estuvo listo en diciembre de 1990. Las tres capas fueron implementadas y probadas durante el primer semestre de 1991, y las especificaciones de las dos primeras capas se congelaron en junio de 1991. Layer III pasó por modificaciones y pruebas durante 1991, y el estándar de audio MPEG completo quedó listo y fue enviado por el grupo de expertos de la ISO en diciembre de 1991, para su aprobación. A la ISO/IEC le tomó casi un año examinar el estándar, y finalmente fue aprobado en noviembre de 1992.

Cuando una película es digitalizada, la parte de audio a menudo puede consistir en dos pistas de sonido (sonido estéreo), cada una muestreada a 44,1 kHz con muestras de 16 bits. La velocidad de datos de audio es, por lo tanto, de $2 \times 44\,100 \times 16 = 1\,411\,200$ $^{\text{bits}}/_{\text{seg}}$; cerca de 1,5 $^{\text{Mbits}}/_{\text{seg}}$. Además a la tasa de muestreo de 44,1 kHz, el estándar MPEG permite velocidades de muestreo de 32 kHz y 48 kHz. Una característica importante del audio MPEG es su razón de compresión, que el estándar especifica ¡por adelantado! El estándar pide para un stream comprimido con una de las diversas tasas de bits (Tabla 7.46) rangos entre 32 y 224 kbps por canal de audio (normalmente hay dos canales, para un sonido estéreo). Dependiendo de la velocidad de muestreo original, estas tasas de bits se traducen a factores de compresión de 2,7 (bajo) a 24 (¡impresionante!) La razón para la especificación de las tasas de bits del stream comprimido es que el stream también incluye datos comprimidos para las partes de vídeo y del sistema. (Estas partes se mencionaron en la Sección 6.5 y no son las mismas que las tres capas de audio.)

	Tasa de bits (*Bitrate*) en Kbps		
Índice	Capa (*Layer*) I	Capa (*Layer*) II	Capa (*Layer*) III
0000	formato libre	formato libre	formato libre
0001	32	32	32
0010	64	48	40
0011	96	56	48
0100	128	64	56
0101	160	80	64
0110	192	96	80
0111	224	112	96
1000	256	128	112
1001	288	160	128
1010	320	192	160
1011	352	224	192
1100	384	256	224
1101	416	320	256
1110	448	384	320
1111	prohibido	prohibido	prohibido

Tabla 7.46: Tasas de bits en las tres capas.

El principio de la compresión de audio MPEG es la cuantificación. Los valores que se cuantifican, sin embargo, no son las muestras de audio, sino números (llamados señales) tomados en el dominio de la frecuencia del sonido (ésto se discute en el párrafo siguiente). El hecho de que la razón de compresión (o equivalentemente, la tasa de bits) es conocida por el codificador que significa éste sabe en todo momento cuántos bits puede asignar a las señales cuantificadas. En consecuencia, el *algoritmo de distribución (o asignación) de bits* (adaptativo) es una parte importante del codificador. Este algoritmo utiliza la tasa de bits conocida y el espectro de frecuencias de las muestras de audio más recientes para determinar el tamaño de las señales cuantificadas de modo que el ruido de cuantificación (la diferencia entre una señal original y una cuantificada) sea inaudible (es decir, esté por debajo del *umbral de enmascaramiento*, un concepto que se discute en la Sección 7.3).

Los modelos psicoacústicos utilizan la frecuencia del sonido que está siendo comprimido, pero el stream de entrada está formado por muestras de audio, no por frecuencias de sonido. Las frecuencias tienen que ser calculadas a partir de las muestras. Por esta razón la primera etapa de la codificación de audio MPEG es una transformada discreta de Fourier, donde un conjunto de 512 muestras de audio consecutivas es transformado al dominio de la frecuencia. Puesto que el número de frecuencias puede ser enorme, se agrupan en 32 subbandas de frecuencias de igual anchura (la capa III utiliza un número diferente pero el mismo principio). Para cada subbanda, se obtiene un número que indica la intensidad del sonido en el rango de frecuencias de esa subbanda. Estos números (llamados *señales*) son entonces cuantificados. La tosquedad de la cuantificación en cada subbanda es determinada por el umbral de enmascaramiento en la subbanda y por el número de bits todavía disponible para el codificador. El umbral de enmascaramiento es calculado para cada subbanda usando un modelo psicoacústico.

MPEG utiliza dos modelos psicoacústicos para implementar el enmascaramiento de frecuencia y el enmascaramiento temporal. Cada modelo describe cómo los sonidos fuertes enmascara otros sonidos que se producen cercanos en la frecuencia o en el tiempo. El modelo particiona el rango de frecuencias en 24 bandas críticas y especifica cómo aplicar efectos de enmascaramiento dentro de cada banda. Los efectos de enmascaramiento dependen, por supuesto, en las frecuencias y amplitudes de los tonos. Cuando el sonido se descomprime y reproduce, el usuario (oyente) puede seleccionar cualquier amplitud

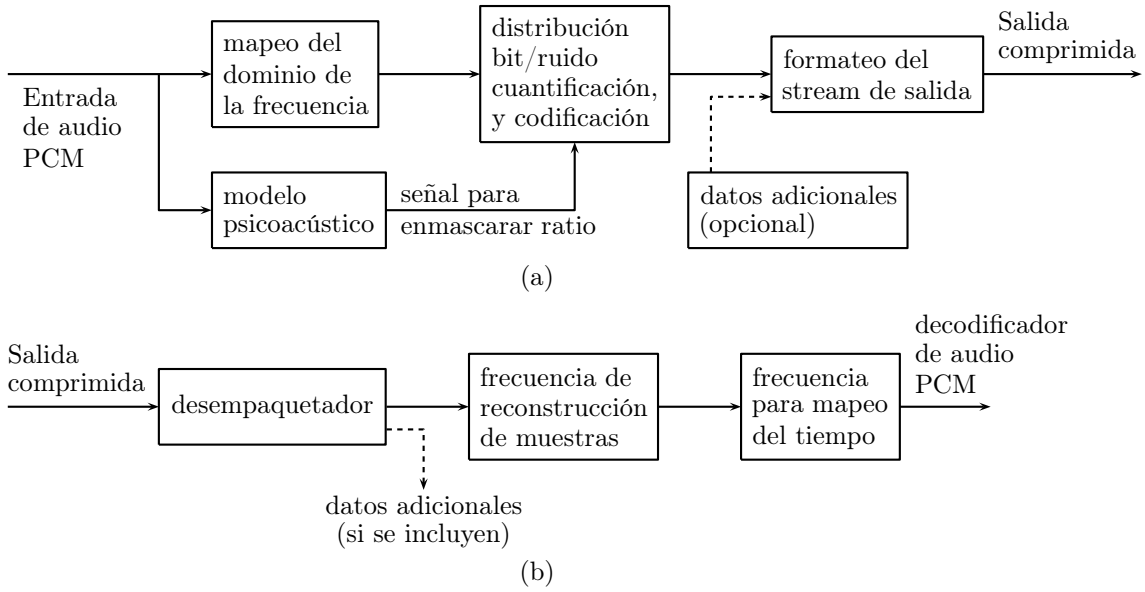


Figura 7.47: Audio MPEG: (a) Codificador y (b) Decodificador.

de reproducción, por lo que el modelo psicoacústico tiene que ser diseñado para el peor caso. Los efectos de enmascaramiento también dependen de la naturaleza de la fuente del sonido que se comprime. La fuente puede ser similar al tono o al ruido. Los dos modelos psicoacústicos empleados por MPEG se basan en el trabajo experimental realizado por investigadores durante muchos años.

El decodificador debe ser rápido, ya que puede tener que decodificar toda la película (video y audio) en tiempo real, por lo que debe ser simple. Como resultado, no utiliza ningún modelo psicoacústico o algoritmo de distribución de bits. El stream comprimido, por lo tanto, debe contener toda la información que el decodificador necesita para descuantificar las señales. Esta información (el tamaño de las señales cuantificadas) debe ser escrita por el codificador en el stream comprimido, y constituye sobrecarga que debe ser sustraída del número restante de bits disponibles.

La Figura 7.47 es un diagrama de bloques de los componentes principales del codificador de audio MPEG y del decodificador. Los datos auxiliares son definibles por el usuario y normalmente consisten en información relacionada con aplicaciones específicas. Estos datos son opcionales.

7.14.1. Codificación en el dominio de la frecuencia

El primer paso en la codificación de las muestras de audio es transformarlas desde el dominio del tiempo al dominio de la frecuencia. Ésto se efectúa mediante un banco de *filtros polifásicos* que transforman las muestras en 32 subbandas de frecuencias de igual anchura. Los filtros fueron diseñados para proporcionar un funcionamiento rápido combinado con buenas resoluciones de tiempo y frecuencia. Como resultado, su diseño incluyó tres compromisos.

El primer compromiso es la igualdad en las anchuras de las 32 bandas de frecuencia. Ésto simplifica los filtros, pero está en contraste con el comportamiento del sistema auditivo humano, cuya sensibilidad es dependiente de la frecuencia. Idealmente, los filtros deben dividir la entrada en bandas críticas, discutidas en la Sección 7.3. Estas bandas están formadas de tal manera que la percepción del volumen de un sonido dado y su audibilidad en presencia de otro sonido, que lo enmascara, son consistentes dentro de una banda crítica, pero diferentes al atravesar estas bandas. Desafortunadamente, cada una de las subbandas de baja frecuencia se solapa con varias bandas críticas, con el resultado de que el

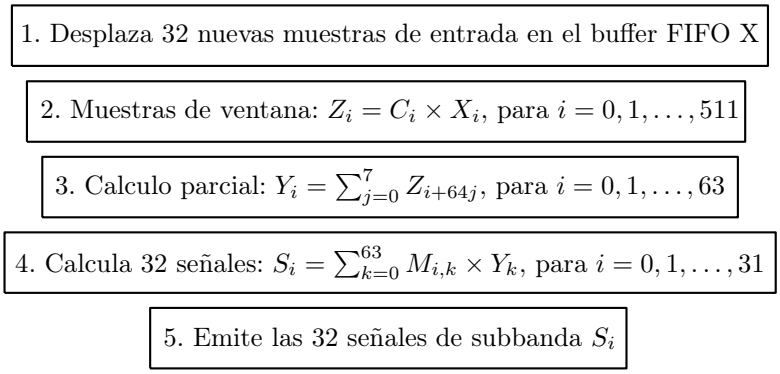


Figura 7.48: Banco de filtros polifase.

algoritmo de distribución de bits no puede optimizar el número de bits asignados a la señal cuantificada en dichas subbandas. Cuando varias bandas críticas están cubiertas por una subbanda X, el algoritmo de distribución de bits selecciona la banda crítica con el enmascaramiento de ruido menor y utiliza esa banda crítica para calcular el número de bits asignados a las señales cuantificadas en la subbanda X.

El segundo compromiso involucra el banco de filtros inverso, el usado por el decodificador. La transformación original de tiempo a frecuencia implica la pérdida de información (incluso antes de cualquier cuantificación). El banco de filtros inverso recibe, por lo tanto, datos que son ligeramente erróneos, y los utiliza para efectuar la transformación inversa de frecuencia a tiempo, produciendo más distorsiones. Por consiguiente, el diseño de los dos bancos de filtros (para las transformaciones directa e inversa) tiene que usar compromisos para minimizar esta pérdida de información.

El tercer compromiso tiene que ver con los filtros individuales. Los filtros adyacentes deben idealmente pasar por diferentes rangos de frecuencia. En la práctica, tienen una frecuencia de solapamiento considerable. El sonido de una sola frecuencia, pura, puede, por lo tanto, penetrar a través de dos filtros y producir señales (que son más adelante cuantificadas) en dos de las 32 subbandas en vez de en sólo una subbanda.

El banco de filtros polifásico utiliza (además de otras estructuras de datos intermedios) un buffer X con capacidad para 512 muestras de entrada. El buffer es una cola FIFO y siempre contiene las 512 muestras de entrada más recientes. La Figura 7.48 muestra los cinco pasos principales del algoritmo de filtrado polifásico.

Las 32 muestras de audio siguientes leídas desde la entrada son desplazadas en el buffer. En consecuencia, el buffer siempre tiene las 512 muestras de audio más recientes. Las señales $S_t\,[i]$ para las 32 subbandas se calculan mediante:

$$S_t\,[i] = \sum_{k=0}^{63} \sum_{j=0}^{7} M_{i,k}\,(C\,[k+64j] \times X\,[k+64j])\,,\ i = 0, \ldots, 31. \tag{7.12}$$

La notación $S_t\,[i]$ representa la señal de la subbanda i en el tiempo t. El vector C contiene 512 coeficientes de la ventana de análisis y está completamente especificado por el estándar. Algunos de los coeficientes de filtro de la subbanda C_i se muestran en la Tabla 7.49a. M es la matriz de análisis definida por:

$$M_{i,k} = \cos\left(\frac{(2i+1)\,(k-16)\,\pi}{64}\right),\ i = 0, \ldots, 31,\ k = 0, \ldots, 63. \tag{7.13}$$

Observe que la expresión entre paréntesis en la Ecuación (7.12) no depende de i, mientras que $M_{i,k}$, en la Ecuación (7.13) no depende de j. (Esta matriz es una variante de la MDCT, que es una

i	C_i	i	C_i	i	C_i
0	0,000000000	252	−0,035435200	506	−0,000000477
1	−0,000000477	253	−0,035586357	507	−0,000000477
2	−0,000000477	254	−0,035694122	508	−0,000000477
3	−0,000000477	255	−0,035758972	509	−0,000000477
4	−0,000000477	256	−0,035780907	510	−0,000000477
5	−0,000000477	257	−0,035758972	511	−0,000000477
⋮		⋮			

(a)

i	D_i	i	D_i	i	D_i
0	0,000000000	252	−1,133926392	506	0,000015259
1	−0,000015259	253	−1,138763428	507	0,000015259
2	−0,000015259	254	−1,142211914	508	0,000015259
3	−0,000015259	255	−1,144287109	509	0,000015259
4	−0,000015259	256	−1,144989014	510	0,000015259
5	−0,000015259	257	−1,144287109	511	0,000015259
⋮		⋮			

(b)

Tabla 7.49: Coeficientes de la ventana de (a) análisis y (b) síntesis.

versión modificada de la bien conocida matriz de la DCT.) Esta característica es un compromiso que genera un menor número de operaciones aritméticas. De hecho, las 32 señales de $S_t\,[i]$ se calculan con sólo $512 + 32 \times 64 = 2560$ multiplicaciones y $64 \times 7 + 32 \times 63 = 2464$ adiciones, que llegan a cerca de 80 multiplicaciones y 80 sumas por señal. Otro punto a destacar es la decimación de las muestras (Sección 5.7). El banco de filtros completo produce 32 señales de salida para 32 muestras de entrada. Puesto que cada uno de los 32 filtros produce 32 señales, su salida debe ser decimada, manteniendo sólo una señal por filtro.

La Figura 7.51a,b ilustra gráficamente las operaciones efectuadas por el codificador y el decodificador durante la etapa de filtrado polifásico. La parte (a) de la figura muestra cómo el buffer X contiene 64 segmentos de 32 muestras de audio cada uno. El buffer es desplazado un segmento a la derecha antes de que el siguiente segmento de 32 nuevas muestras sea leído de la secuencia de entrada e introducido por el lado izquierdo. Después de multiplicar el buffer X por los coeficientes de la ventana C, los productos se mueven al Z vector. El contenido de este vector es dividido en segmentos de 64 números cada uno, y los segmentos son agregados. El resultado es el vector Y, el cual se multiplica por la matriz de MDCT, para producir el vector final de 32 señales de subbanda.

La parte (b) de la figura ilustra las operaciones efecuadas por el decodificador. Un grupo de 32 señales de subbanda es multiplicado por la matriz $N_{i,k}$ de la IMDCT, para producir el vector V, formado por dos segmentos de 32 valores cada uno. Los dos segmentos son desplazados en el buffer FIFO V por la izquierda. El buffer V tiene espacio para los últimos 16 vectores V (i.e., 16×64, ó 1024, valores). Se crea un nuevo vector U de 512 entradas a partir de 32 segmentos alternos del buffer V, tal como se muestra. El vector U es multiplicado por los 512 coficientes D_i de la ventana de síntesis (similares a los coeficientes C_i de la ventana de análisis utilizada por el codificador), para crear el vector W. Algunos de los coeficientes D_i se muestran en la Tabla 7.49b. Este vector se divide en 16 segmentos de 32 valores de cada uno y los segmentos se suman. El resultado son 32 muestras de audio reconstruidas. La Figura 7.50 es un diagrama de flujo que ilustra este proceso. La matriz de síntesis

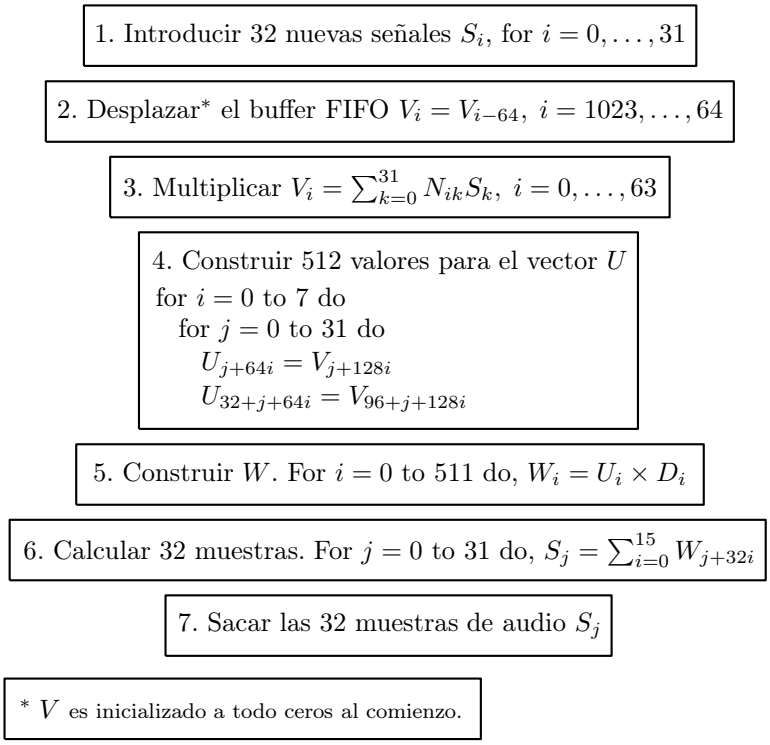


Figura 7.50: Reconstrucción de las muestras de audio.

$N_{i,k}$ de la IMDCT viene dada por:

$$N_{i,k} = \cos\left(\frac{(2k+1)\,(i+16)\,\pi}{64}\right),\ i = 0, \ldots, 63,\ k = 0, \ldots, 31.$$

Las señales de subbanda calculadas por la etapa de filtrado del codificador son recogidas y empaquetadas en *frames* que contienen 384 señales (en la capa I) ó 1152 señales (en las capas II y III) cada uno. Las señales en un frame son posteriormente escaladas y cuantificadas de acuerdo con el modelo psicoacústico utilizado por el codificador y el algoritmo de distribución de bits. Los valores cuantificados, junto con los factores de escala y la información de cuantificación (el número de niveles de cuantificación en cada subbanda) son escritos en el stream comprimido (la capa III también utiliza códigos de Huffman para codificar los valores cuantificados aún más).

7.14.2. Formato de los datos comprimidos

En la capa I, cada frame está formado por 12 señales por subbanda, para un total de $12 \times 32 = 384$ señales. En las capas II y III, un frame contiene 36 señales por subbanda, para un total de 1152 señales. Las señales en el frame son cuantificadas (éste es el paso importante cuando se logra compresión) y escritas en el stream comprimido junto con otra información.

Cada frame escrito en la salida comienza con una cabecera 32 bits cuyo formato es idéntico para las tres capas. La cabecera contiene un código de sincronización (12 bits a 1) y 20 bits correspondientes a parámetros de codificación mencionados más abajo. Si se utiliza la protección contra errores, la cabecera es seguida inmediatamente por una palabra de entrada de 16 bits CRC (Sección 3.28) que

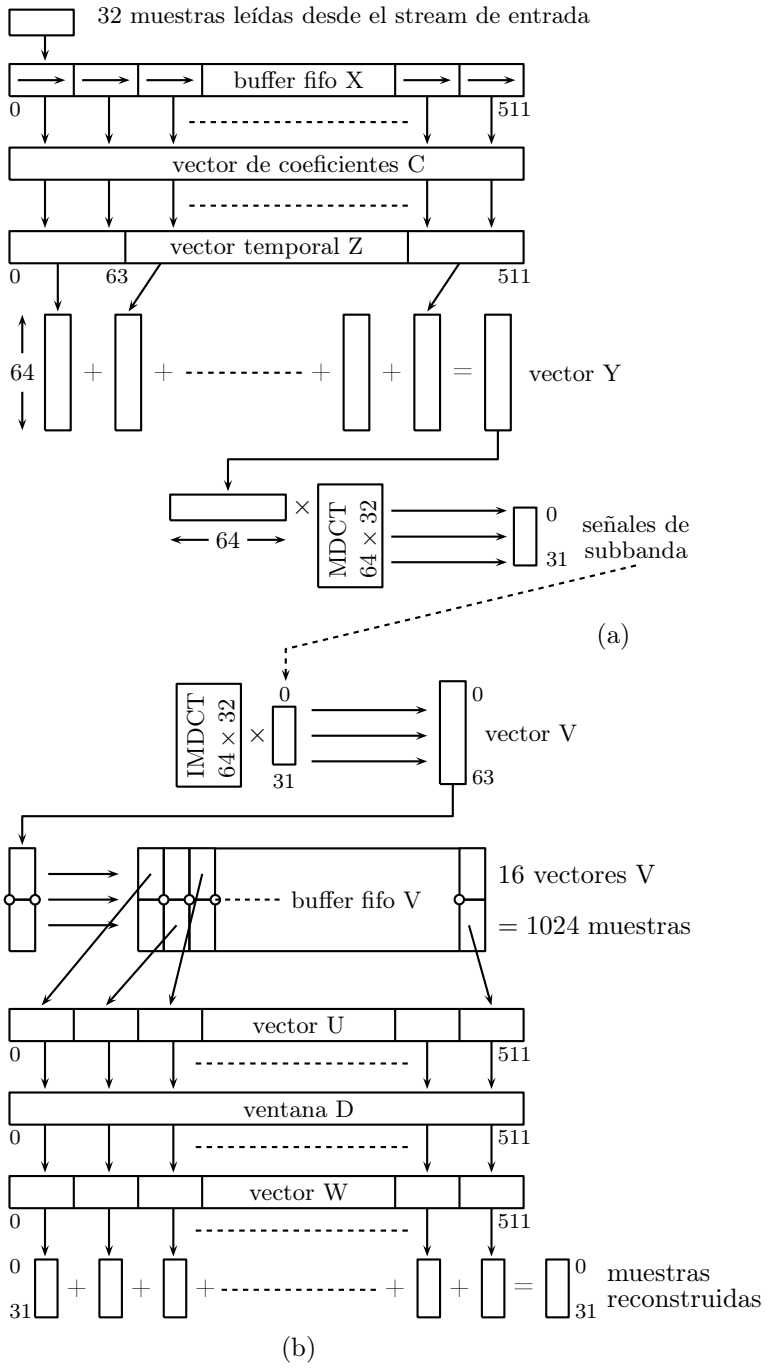


Figura 7.51: Audio MPEG: (a) Codificador y (b) Decodificador.

usa el polinomio generador $CRC_{16}(x) = x^{16} + x^{15} + x^2 + 1$. A continuación viene un frame de las señales cuantificadas, seguido por un bloque (opcional) de datos auxiliares. Los formatos de los dos últimos ítems dependen de la capa.

En la capa I, el código CRC de 16 bits es calculado a partir de los últimos 16 bits (bits 16–31) de la cabecera, y de la información de distribución de bits. En la capa II, el código CRC se calcula a partir de estos bits más la información de selección del factor de escala. En la capa III, el código CRC es calculado a partir de los últimos 16 bits de la cabecera y también, ya sea desde de los bits 0–135 de datos de audio (en modo monocanal), ya sea desde los bits 0–255 de datos de audio (en los otros tres modos, véase el campo 8 más abajo).

El código de sincronización es utilizado por el decodificador para comprobar que lo que se está leyendo es, en efecto, la cabecera. Este código es una cadena de 12 bits a 1, por lo que el formato completo del stream comprimido ha tenido que ser diseñado para evitar una ocurrencia accidental de una cadena de doce 1's en cualquier otro lugar.

Los 20 bits restantes de la cabecera se dividen en 12 campos de la siguiente manera:

- *Campo 1.* Un bit ID cuyo valor es 1 (ésto indica que se utiliza MPEG). Un valor 0 está reservado y no se utiliza actualmente.
- *Campo 2.* Dos bits para indicar el número de capa. Los valores válidos son: 11 para capa I, 10 para capa II, 01 para capa III. El valor 00 está reservado.
- *Campo 3.* Un bit indicador de protección de errores. Un valor 0 indica que se ha añadido redundancia al stream comprimido para ayudar en la detección de errores.
- *Campo 4.* Cuatro bits para indicar la tasa de bits o *bitrate* (Tabla 7.46). Un índice cero indica una tasa de bits "fija", donde un frame puede contener un *slot* adicional, que depende del bit de relleno (campo 6).
- *Campo 5.* Dos bits para indicar una de tres frecuencias de muestreo. Los tres valores son: 00 para 44,1 kHz, 01 para 48 kHz, y 10 para 32 kHz. El valor 11 está reservado.
- *Campo 6.* Un bit para indicar si se utiliza el relleno. El relleno puede añadir un *slot* (los *slots* se discuten en la Sección 7.14.3) al stream comprimido después de un cierto número de frames, para asegurarse de que el tamaño total de los frames es igual o ligeramente menor que la suma:

$$\sum_{\text{primer frame}}^{\text{frame actual}} \frac{\text{tamaño del frame} \times \text{tasa de bits}}{\text{frecuencia de muestreo}},$$

donde el tamaño del frame es de 384 señales para la capa I y 1152 señales para las capas II y III. El siguiente algoritmo puede ser utilizado por el codificador para determinar si es necesario relleno o no.

```
for primer frame de audio:
  rest:=0;
  relleno:=no;
for cada frame de audio subsiguiente:
  if capa=I
    then dif:=(12 × bitrate) modulo (frecuencia-de-muestreo)
    else dif:=(144 × bitrate) modulo (frecuencia-de-muestreo);
  rest:=rest − dif;
if rest<0 then
  relleno:=yes;
  rest:=rest+(frecuencia-de-muestreo)
else relleno:=no;
```

Este algoritmo tiene una interpretación simple. Un frame se divide en N ó $N + 1$ slots, donde N depende de la capa. Para la capa I, N viene dada por:

$$N = 12 \times \frac{\text{tasa de bits}}{\text{frecuencia de muestreo}}.$$

Para las capas II y III, viene dada por:

$$N = 144 \times \frac{\text{tasa de bits}}{\text{frecuencia de muestreo}}.$$

Si ésto no produce un número entero, el resultado es truncado y se usa el relleno.

El relleno también se menciona en la Sección 7.14.3

- *Campo 7.* Un bit para uso privado del codificador. Este bit no será utilizado por la norma ISO/IEC en el futuro.

- *Campo 8.* Un campo de dos bits para el modo estéreo. Los valores son: 00 para estéreo, 01 para estéreo conjunto (estéreo de intensidad y/o estéreo ms), 10 para canal dual, y 11 para monocanal.

  La información estéreo es codificada en uno de los cuatro modos: estéreo, canal dual, estéreo conjunto y estéreo ms. En los dos primeros modos, las muestras de los dos canales estéreo son comprimidos y escritos en la salida. El codificador no comprueba cualesquiera correlaciones entre los dos. El modo estéreo se utiliza para comprimir los canales estéreo izquierdo y derecho, mientras que el modo canal dual se utiliza para comprimir diferentes conjuntos de muestras de audio, tales como una emisión bilingüe. El modo estéreo conjunto explota las redundancias entre los canales izquierdo y derecho, ya que muchas veces son idénticos, similares, o difieren en un lapso de tiempo reducido. El modo estéreo ms ("ms" proviene de "middle-side") es un caso especial de estéreo conjunto, donde dos señales, un valor medio (*middle*) M_i y un valor lateral o secundario (side) S_i, son codificados en lugar de los canales de audio izquierdo y derecho L_i y R_i. Los valores middle-side son calculados mediante la siguiente suma y diferencia:

$$L_i = \frac{M_i + S_i}{\sqrt{2}}, \text{ y } R_i = \frac{M_i - S_i}{\sqrt{2}}.$$

- *Campo 9.* Un campo de dos bits para el modo de extensión. Ésto se utiliza en el modo estéreo conjunto. En las capas I y II los bits indican que las subbandas están en estéreo de intensidad. Todas las demás subbandas están codificadas en modo "estéreo". Los cuatro valores son:

  - 00 para subbandas en estéreo de intensidad 4–31, límite = 4.
  - 01 para subbandas en estéreo de intensidad 8–31, límite = 8.
  - 10 para subbandas en estéreo de intensidad 12–31, límite = 12.
  - 11 para subbandas en estéreo de intensidad 16–31, límite = 16.

  En la capa III estos bits indican qué tipo de método de codificación estéreo conjunta se aplica. Los valores son:

  - 00 para estéreo de intensidad *off*, estéreo ms *off*.
  - 01 para estéreo de intensidad *on*, estéreo ms *off*.
  - 10 para estéreo de intensidad *off*, estéreo ms *on*.
  - 11 para estéreo de intensidad *on*, estéreo ms *on*.

Índice	factor de escala	Índice	factor de escala	Índice	factor de escala
0	2,00000000000000	21	0,01562500000000	42	0,00012207031250
1	1,58740105196820	22	0,01240157071850	43	0,00009688727124
2	1,25992104989487	23	0,00984313320230	44	0,00007689947814
3	1,00000000000000	24	0,00781250000000	45	0,00006103515625
4	0,79370052598410	25	0,00620078535925	46	0,00004844363562
5	0,62996052494744	26	0,00492156660115	47	0,00003844973907
6	0,50000000000000	27	0,00390625000000	48	0,00003051757813
7	0,39685026299205	28	0,00310039267963	49	0,00002422181781
8	0,31498026247372	29	0,00246078330058	50	0,00001922486954
9	0,25000000000000	30	0,00195312500000	51	0,00001525878906
10	0,19842513149602	31	0,00155019633981	52	0,00001211090890
11	0,15749013123686	32	0,00123039165029	53	0,00000961243477
12	0,12500000000000	33	0,00097656250000	54	0,00000762939453
13	0,09921256574801	34	0,00077509816991	55	0,00000605545445
14	0,07874506561843	35	0,00061519582514	56	0,00000480621738
15	0,06250000000000	36	0,00048828125000	57	0,00000381469727
16	0,04960628287401	37	0,00038754908495	58	0,00000302772723
17	0,03937253280921	38	0,00030759791257	59	0,00000240310869
18	0,03125000000000	39	0,00024414062500	60	0,00000190734863
19	0,02480314143700	40	0,00019377454248	61	0,00000151386361
20	0,01968626640461	41	0,00015379895629	62	0,00000120155435

Tabla 7.52: Factores de escala de las capas I y II.

- *Campo 10.* Bit de derechos de autor (*copyright*). Si el stream comprimido está protegido por copyright, este bit debería ser 1.
- *Campo 11.* Un bit que indica original/copia. Un valor 1 indica un stream comprimido original.
- *Campo 12.* Un campo de 2 bits de énfasis. Ésto indica el tipo de de énfasis que se ha utilizado. Los valores son: 00 para ninguno, 01 para 50/15 microsegundos, 10 está reservado, y 11 indica énfasis de CCITT J.17.

Capa I: Las 12 señales de cada subbanda son escaladas de tal manera que la más grande se convierte en 1 (o muy cercana a 1, pero no mayor que 1). El modelo psicoacústico y el algoritmo de distribución de bits son invocados para determinar el número de bits asignados a la cuantificación de cada señal escalada en cada subbanda (o, equivalentemente, el número de niveles de cuantificación). Las señales escaladas son entonces cuantificadas. El número de niveles de cuantificación, los factores de escala y las 384 señales cuantificadas se colocan entonces en sus áreas en el frame, que es escrito en la salida. Cada elemento de asignación de bits es de cuatro bits, cada factor de escala es de seis bits, y cada muestra cuantificada ocupa entre dos y 15 bits en el frame.

El número l de niveles de cuantificación y el número q de bits por valor cuantificado están relacionados mediante $2^q = l$. El algoritmo de distribución de bits utiliza tablas para determinar q para cada una de las 32 subbandas. Los $32\,(q-1)$ valores se escriben entonces, como números de 4 bits, en el frame, para uso del decodificador. En consecuencia, el valor 0000 de 4 bits leído por el decodificador desde el frame para la subbanda s indica al decodificador que las 12 señales de s han sido cuantificadas toscamente a un bit cada una, mientras que el valor 1110 significa que las señales de 16 bits han sido bien cuantificadas a 15 bits cada una. El valor 1111 no se utiliza, para evitar un conflicto con el código de sincronización. Muchas veces, el codificador decide cuantificar todas las 12 señales de una subbanda

s a cero, y ésto es indicado en el frame mediante un código diferente. En tal caso, el valor de entrada de 4 bits del frame para s es ignorado por el decodificador.

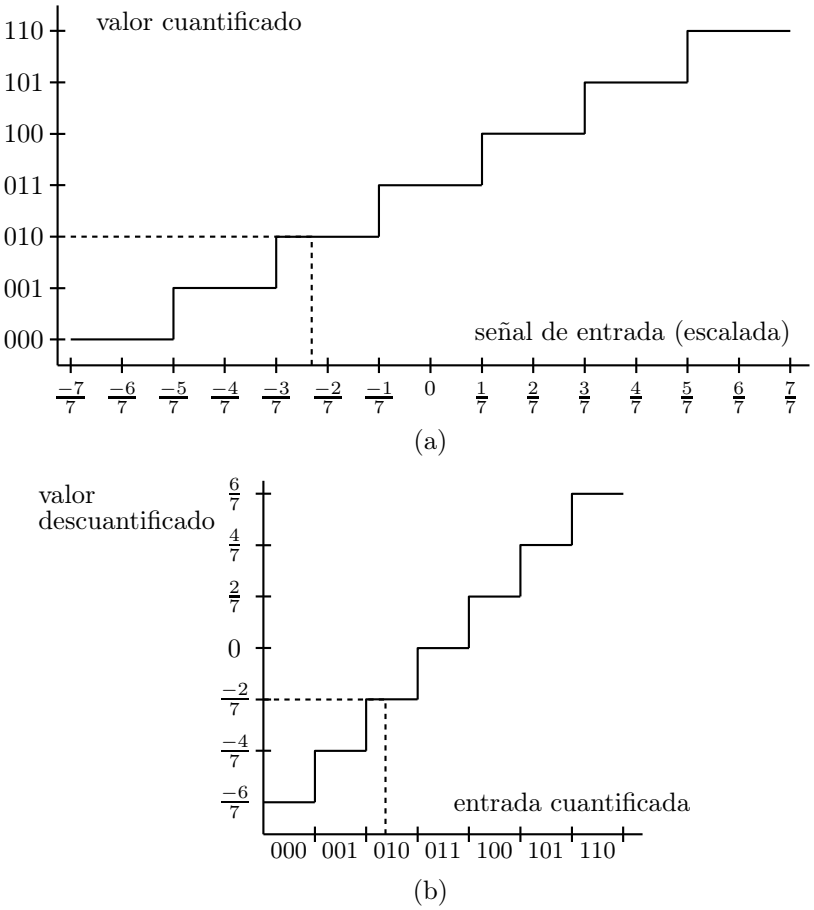


Figura 7.53: Ejemplos de (a) Cuantificación y (b) Descuantificación.

◇ **Ejercicio 7.8 (sol. en pág. 1106):** Explíquese por qué el codificador puede decidir cuantificar 12 señales consecutivas de la subbanda s a cero.

El decodificador multiplica los valores de la señal descuantificada por los factores de escala que se encuentran en el frame. Hay un factor de escala para las 12 señales de una subbanda. Este factor de escala es seleccionado por el codificador a partir de una tabla de 63 factores de escala especificados por el estándar (Tabla 7.52). Los factores de escala de la tabla aumentan en un factor de $\sqrt[3]{2}$.

◇ **Ejercicio 7.9 (sol. en pág. 1107):** ¿Cómo se traduce este incremento en un aumento del nivel de decibelios?

La cuantificación se efectúa con la simple regla siguiente: Si el algoritmo de distribución de bits asigna b bits a cada valor cuantificado, entonces el número n de niveles de cuantificación es determinado mediante $b = \log_2(n+1)$, ó $2^b = n+1$. El valor $b = 3$, por ejemplo, produce $n = 7$ valores de cuantificación. La Figura 7.53a,b muestra tales ejemplos de cuantificación. Las señales de entrada que están siendo cuantificadas ya han sido escaladas al intervalo $[-1, +1]$, y la cuantificación es *midtread*. Por ejemplo, todos los valores de entrada en el rango $[-1/7, -3/7]$ se cuantifican en 010 (las líneas

bits asignados	código de 4 bits	número de niveles	SNR (dB)
0	0000	0	0,00
2	0001	3	7,00
3	0010	7	16,00
4	0011	15	25,28
5	0100	31	31,59
6	0101	63	37,75
7	0110	127	43,84
8	0111	255	49,89
9	1000	511	55,93
10	1001	1023	61,96
11	1010	2047	67,98
12	1011	4095	74,01
13	1100	8191	80,03
14	1101	16383	86,05
15	1110	32767	92,01
inválido	1111		

Tabla 7.54: Distribución de bits y cuantificación en la Capa I.

discontinuas en la figura). La descuantificación es también sencilla. El valor cuantificado 010 es siempre descuantificado a $-2/7$. Observe que los valores cuantificados varían de 0 a $n-1$. El valor de $n = 11\ldots1$ nunca se utiliza, con el fin de evitar un conflicto con el código de sincronización.

La Tabla 7.54 muestra que el algoritmo de distribución de bits para la capa I puede seleccionar el tamaño de las señales cuantificadas para que estén entre 0 y 15 bits. Para cada uno de estos tamaños, la tabla muestra el código de asignación de 4 bits que el codificador escribe en el frame para uso del decodificador, el número q de niveles de cuantificación (entre 0 y $2^{15} - 1 = 32\,767$, y la relación señal a ruido en decibelios.

En la práctica, el codificador escala una señal S_i en un factor de escala scf que es determinado a partir de la Tabla 7.52, y cuantifica el resultado mediante el cálculo:

$$S_{qi} = \left(A \left[\frac{S_i}{scf} \right] + B \right) \Big|_N ,$$

donde A y B son las constantes mostradas en la Tabla 7.55 (las tres entradas marcadas con asteriscos son utilizadas por la capa II, pero no por la capa I) y N es el número de bits necesarios para codificar el número de niveles de cuantificación. (La barra vertical con subíndice N significa: Tomar los N bits más significativos.) Con el fin de evitar conflictos con el código de sincronización, el bit más significativo del valor cuantificado S_{qi} es invertido antes de que esa cantidad sea escrita en la salida.

Formato de la capa II: Ésto es una extensión del método básico descrito para la capa I. Cada frame consta ahora de 36 señales por subbanda, y tanto los datos de distribución de bits como la información del factor de escala están codificados más eficientemente. Además, la cuantificación puede ser mucho más fina y puede tener hasta $2^{16} - 1 = 65\,535$ niveles. Un frame se divide en tres partes, numeradas 0, 1, y 2. Cada parte se asemeja a un frame de la capa I y contiene 12 señales por subbanda. Los datos de distribución de bits son comunes a las tres partes, pero la información para los factores de escala se organizan de manera diferente. Puede ser común a las tres partes, puede aplicarse a sólo dos de las tres partes, o puede ser especificada para cada parte por separado. Esta información se compone de una información de selección del factor de escala (sfsi) de 2 bits. Este número indica si se escriben uno, dos, o tres factores de escala por subbanda en el frame, y cómo se aplican.

número de niveles	A	B
3	0,750000000	−0,250000000
5*	0,625000000	−0,375000000
7	0,875000000	−0,125000000
9*	0,562500000	−0,437500000
15	0,937500000	−0,062500000
31	0,968750000	−0,031250000
63	0,984375000	−0,015625000
127	0,992187500	−0,007812500
255	0,996093750	−0,003906250
511	0,998046875	−0,001953125
1023	0,999023438	−0,000976563
2047	0,999511719	−0,000488281
4095	0,999755859	−0,000244141
8191	0,999877930	−0,000122070
16383	0,999938965	−0,000061035
32767	0,999969482	−0,000030518
65535*	0,999984741	−0,000015259

Tabla 7.55: Coeficientes de cuantificación in las capas I y II.

La sección de distribución de bits del frame es codificada de forma más eficiente limitando la elección de niveles de cuantificación a las subbandas más altas y a las tasas de bits más bajas. En lugar de los cuatro bits por subbanda utilizados por la capa I para especificar la opción de distribución de bits, el número de bits utilizados por la capa II varía de 0 a 4, dependiendo del número de subbanda. El estándar MPEG contiene una tabla que busca el codificador mediante el número de subbanda y la tasa de bits para encontrar el número de bits.

La cuantificación es similar a la capa I, excepto que la capa II puede a veces empaquetar tres valores cuantificados consecutivos de una palabra de código único. Ésto puede hacerse cuando el número de niveles de cuantificación es una potencia de 2, y reduce el número de bits perdidos.

7.14.3. Codificación: Capas I y II

El estándar MPEG especifica una tabla de factores de escala. Para cada subbanda, el codificador compara la mayor de las 12 señales con los valores de la tabla, busca el siguiente valor más grande, y utiliza el índice de la tabla de ese valor para determinar el factor de escala para las 12 señales. En la capa II, el codificador determina tres factores de escala para cada subbanda, una para cada una de las tres partes. Calcula la diferencia entre las dos primeras y las dos últimas y utiliza las dos diferencias para decidir si codifica uno, dos, o los tres factores de escala. Este proceso se describe en la Sección 7.14.4.

La información de distribución de bits en la capa II utiliza los bits 2–4. La *información de selección del factor de escala* (*scale factor select information* o sfsi) es de dos bits. El factor de escala en sí utiliza seis bits. Cada señal cuantificada utiliza los bits 2–16, y puede haber datos auxiliares.

El estándar describe dos modelos psicoacústicos. Cada uno produce una cantidad llamada *relación señal a máscara* (*signal to mask ratio* o SMR) para cada subbanda. El algoritmo de distribución de bits utiliza esta SMR y la SNR de la Tabla 7.54 para calcular la *relación máscara a ruido* (*mask to noise ratio* o MNR) como la diferencia:

$$\text{MNR} = \text{SMR} - \text{SNR dB}.$$

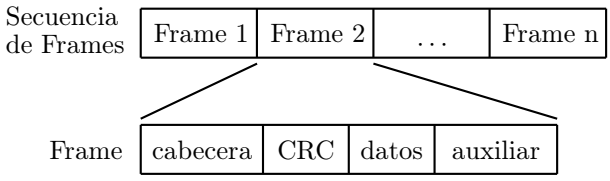


Figura 7.56: Formato de salida comprimido.

La MNR indica la discrepancia entre el error de la wavelet y la medición perceptiva, y la idea es que las señales de subbanda pueden ser comprimidas tanto como la MNR. Como resultado, cada iteración del bucle de distribución de bits determina la MNR mínima de todas las subbandas. El principio básico de la distribución de bits es reducir al mínimo la MNR sobre un fotograma usando no más que el número de bits B_f disponibles para el frame.

En cada iteración, el algoritmo calcula el número de bits B_f disponibles para codificar un frame. Éste se calcula a partir de la frecuencia de muestreo (número de muestras de entrada por segundo, normalmente $2\times44\,100$) y la tasa de bits (número de bits escritos en la salida comprimida por segundo). El cálculo es:

$$^{\text{frames}}/_{\text{seg.}} = \frac{^{\text{muestras}}/_{\text{segundo}}}{^{\text{muestras}}/_{\text{frame}}}, \quad B_f = \frac{^{\text{bits}}/_{\text{segundo}}}{^{\text{frames}}/_{\text{segundo}}} = \frac{\text{tasa de bits} \times \text{muestras por frame}}{\text{frecuencia de muestreo}}.$$

En consecuencia, B_f se mide en $^{\text{bits}}/_{\text{frame}}$. El encabezado de la trama ocupa 32 bits, y el CRC, si se usa, requiere 16 bits. Los datos de distribución de bits son cuatro bits por subbanda. Si se utilizan los datos auxiliares, su tamaño también es determinado. Estas cantidades se restan del valor de B_f calculado anteriormente.

El paso principal del algoritmo de asignación de bits es maximizar la MNR mínima para todas las subbandas mediante la asignación de los bits restantes a los factores de escala y las señales de cuantificación. En la capa I, los factores de escala toman seis bits por subbanda, pero en la capa II hay varias maneras de codificarlas (Sección 7.14.4).

El paso principal de distribución de bits es un bucle. Comienza con la asignación de bits cero a cada una de las subbandas. Si el algoritmo asigna bits cero a una subbanda particular, entonces no será necesario ningún bit para los factores de escala y las señales cuantificadas. De lo contrario, el número de bits asignados depende del número de capa y del número de factores de escala (1, 2, ó 3) codificados para la subbanda. El algoritmo calcula la SNR y la MNR para cada subbanda y busca la subbanda con el MNR más bajo cuya asignación de bits aún no alcanzado su límite máximo. La distribución de bits para esa subbanda es incrementada en un nivel, y el número de bits adicionales necesarios es substraído del balance de bits disponibles. Ésto se repite hasta que el balance de bits disponibles llegue a cero o hasta que todas las subbandas han alcanzado su límite máximo.

El formato de salida comprimido se muestra en la Figura 7.56. Cada frame se compone de entre dos y cuatro partes. El frame se organiza en secciones o *slots*, donde el tamaño de cada sección es de 32 bits en la capa I, y 8 bits en las capas II y III. En consecuencia, el número de slots es $B_f/32$ ó $B_f/8$. Si el último slot no está completo, se rellena con ceros.

La relación entre la tasa de muestreo, el tamaño del frame, y el número de slots es fácil de visualizar. Los parámetros típicos de capa I son: (1) una velocidad de muestreo de 48 000 $^{\text{muestras}}/_{\text{seg.}}$ (2) una tasa de bits de 64 000 $^{\text{bits}}/_{\text{seg.}}$, y (3) 384 señales cuantificadas por frame. El decodificador debe decodificar $^{48\,000}/_{384} = 125$ frames por segundo. Por consiguiente, cada trama tiene que ser decodificada en 8 ms. Para emitir 125 frames en 64 000 bits, cada frame debe tener $B_f = 512$ bits disponibles para codificarlo. El número de slots por frame es, por lo tanto, $^{512}/_{32} = 16$.

Un ejemplo similar asume: (1) una velocidad de muestreo de 32 000 $^{\text{muestras}}/_{\text{seg.}}$, (2) una tasa de bits de 64 000 $^{\text{bits}}/_{\text{seg.}}$, y (3) 384 señales cuantificadas por frame. debe decodificar $^{32\,000}/_{384} = 83{,}33$

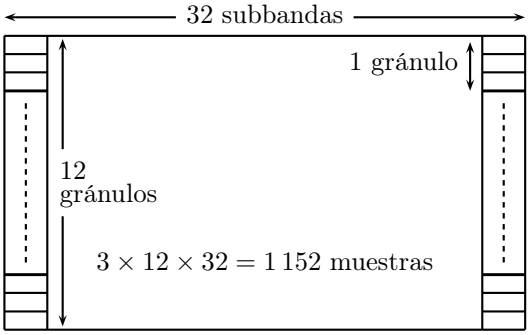


Figura 7.57: Organización de las señales de subbanda de la capa II.

frames por segundo. En consecuencia, cada frame tiene que ser decodificado en 12 ms. Para emitir 83,33 frames en 64 000 bits, cada frame debe tener $B_f = 768$ bits disponibles para codificarlo. Por lo tanto, el número de slots por frame es $^{768}/_{32} = 24$.

◇ **Ejercicio 7.10 (sol. en pág. 1107):** Realícese el mismo cálculo para una tasa de muestreo de 44 100 $^{\text{muestras}}/_{\text{seg.}}$.

7.14.4. Codificación: Capa II

Un frame en la capa II consta de 36 señales de subbanda, organizadas en 12 gránulos como se muestra en la Figura 7.57. Para cada subbanda, se calculan tres factores de escala $scf1$, $scf2$, y $scf3$; para cada grupo de 12 señales, un factor de escala. Ésto se efectúa en seis pasos de la siguiente manera:

- *Paso 1:* Se determina el máximo de los valores absolutos de estas 12 señales.
- *Paso 2:* Este máximo se compara con los valores en la columna "factor de escala" de la Tabla 7.52, y se observa la entrada más pequeña que es mayor que la máxima.
- *Paso 3:* El valor en la columna "Índice" de esta entrada se denota scf_i.
- *Paso 4:* Tras repetir los pasos 1–3 para $i = 1, 2, 3$, se calculan las dos diferencias $D_1 = scf1 - scf2$ y $D_2 = scf2 - scf3$.
- *Paso 5:* Se determinan dos valores de "clase", para D_1 y D_2, mediante:

$$\text{Clase}_i = \begin{cases} 1, & \text{si } D_i \leq -3, \\ 2, & \text{si } -3 < D_i < 0, \\ 3, & \text{si } D_i = 0, \\ 4, & \text{si } 0 < D_i < 3, \\ 5, & \text{si } D_i \geq 3. \end{cases}$$

- *Paso 6:* Dependiendo de las dos clases, se determinan tres factores de escala a partir de la Tabla 7.58, columna "s. fact. usado". Los valores de 1, 2 y 3 de esta columna soportan los factores de escala primero, segundo, y tercero, respectivamente, dentro de un frame. El valor 4 supone el máximo de los tres factores de escala. La columna titulada "patr. trans." indica aquellos factores de escala que son realmente escritos en el stream comprimido.

clase 1	clase 2	s. fact usado	patrón trans.	s. fact. selecc.	clase 1	clase 2	s. fact usado	patrón trans.	s. fact. selecc.
1	1	123	123	0	3	4	333	3	2
1	2	122	12	3	3	5	113	13	1
1	3	122	12	3	4	1	222	2	2
1	4	133	13	3	4	2	222	2	2
1	5	123	123	0	4	3	222	2	2
2	1	113	13	1	4	4	333	3	2
2	2	111	1	2	4	5	123	123	0
2	3	111	1	2	5	1	123	123	0
2	4	444	4	2	5	2	122	12	3
2	5	113	13	1	5	3	122	12	3
3	1	111	1	2	5	4	133	13	3
3	2	111	1	2	5	5	123	123	0
3	3	111	1	2					

Tabla 7.58: Capa II. Patrones de transmisión del factor de escala.

$scsfi$	# de factores de escala codificados	decodificación de factores de escala
0 (00)	3	$scf1$, $scf2$, $scf3$
1 (01)	2	primero es $scf1$ y $scf2$ segundo es $scf3$
2 (10)	1	un factor de escala
3 (11)	2	primero es $scf1$ segundo es $scf2$ y $scf3$

Tabla 7.59: Capa II. Selección de información del factor de escala.

La Tabla 7.59 muestra los cuatro valores de la *información de selección del factor de escala* (*scale factor select information* o scfsi) de 2 bits.

Como ejemplo, supongamos que las dos diferencias D_1 y D_2 entre los tres factores de escala A, B, y C de las tres partes de una cierta subbanda se clasifican como $(1, 1)$. La Tabla 7.58 muestra que el factor de transmisión es 123 y la información de selección es 0. La regla superior en la Tabla 7.59 muestra que un scfsi de 0 significa que cada uno de los tres factores de escala es codificado por separado en el stream de salida como un número de 6 bits. (Esta regla es utilizada tanto por el codificador como por el decodificador.) Los tres factores de escala ocupan, en este caso, 18 bits, por lo que hay no hay ahorros en comparación con la codificación de la capa I.

A continuación suponemos que las dos diferencias se clasifican como $(1, 3)$. La Tabla 7.58 muestra que el factor de escala es 122 y la información de selección es 3. La regla inferior en la Tabla 7.59 muestra que un scfsi de 3 significa que los sólo los factores de escala A y B necesitan ser codificados (ya que $B = C$), ocupando sólo 12 bits. El decodificador asigna los primeros seis bits como el valor A, y los siguientes seis bits como los valores de ambos, B y C. En consecuencia, la redundancia en los factores de escala ha sido traducida en un pequeño ahorro.

◇ **Ejercicio 7.11 (sol. en pág. 1107):** ¿Cómo son codificados los tres factores de escala cuando las dos diferencias son clasificadas como $(3, 2)$?

La cuantificación en la capa II es similar a la de la capa I con la diferencia de que se utilizan las dos constantes C y D de la Tabla 7.60 en vez de las constantes A y B. Otra diferencia es el uso de la agrupación. La Tabla 7.60 muestra que se requiere la agrupación cuando el número de niveles de cuantificación es 3, 5, ó 9. En cualquiera de estos casos, se combinan tres señales en una palabra de

número de pasos	C	D	agrupar	muestras/código	bits/código
3	1.3333333333	0.5000000000	sí	3	5
5	1.6000000000	0.5000000000	sí	3	7
7	1.1428571428	0.2500000000	no	1	3
9	1.7777777777	0.5000000000	sí	3	10
15	1.0666666666	0.1250000000	no	1	4
31	1.0322580645	0.0625000000	no	1	5
63	1.0158730158	0.0312500000	no	1	6
127	1.0078740157	0.0156250000	no	1	7
255	1.0039215686	0.0078125000	no	1	8
511	1.0019569471	0.0039062500	no	1	9
1023	1.0009775171	0.0019531250	no	1	10
2047	1.0004885197	0.0009765625	no	1	11
4095	1.0002442002	0.0004882812	no	1	12
8191	1.0001220852	0.0002441406	no	1	13
16383	1.0000610388	0.0001220703	no	1	14
32767	1.0000305185	0.0000610351	no	1	15
65535	1.0000152590	0.0000305175	no	1	16

Tabla 7.60: Clases de cuantificación para la capa II.

código, que es luego cuantificada. El decodificador reconstruye las tres señales $s\,(1)$, $s\,(2)$, y $s\,(3)$ a partir de la palabra de código ω mediante:

$$\begin{aligned}&\omega = 0,\\ &\text{for } i = 0 \text{ to } 2,\\ &\quad s\,(i) = \omega \bmod \text{nos},\\ &\quad \omega = \omega \div \text{nos},\end{aligned}$$

donde "mod" es la función módulo, "$\div$" representa la división entera, y "nos" denota el número de pasos de cuantificación.

La Figura 7.61a,b muestra el formato de un frame en ambas capas, I y II. En la capa I (parte (a) de la figura), hay 384 señales por frame. Asumiendo una tasa de muestreo de 48 000 $^{\text{muestras}}/_{\text{seg.}}$ y una tasa de bits de 64 000 $^{\text{bits}}/_{\text{seg.}}$, dicho marco debe ser decodificado completamente en 80 ms, para una tasa de decodificación de 125 $^{\text{frames}}/_{\text{seg.}}$.

◇ **Ejercicio 7.12 (sol. en pág. 1107):** ¿Cuál es una tasa de frames típica para la capa II?

7.14.5. Modelos psicoacústicos

La tarea de un modelo psicoacústico es hacer posible que el codificador decida fácilmente la cantidad de ruido de cuantificación permitido en cada subbanda. Esta información es utilizada posteriormente por el algoritmo de distribución de bits, junto con el número de bits disponibles, para determinar el número de niveles de cuantificación para cada subbanda. El estándar de audio MPEG especifica dos modelos psicoacústicos. Cualquier modelo puede usarse con cualquier capa, pero solamente el modelo II genera la información específica necesaria para la capa III. En la práctica, el modelo I es el único utilizado en capas I y II. La capa III puede usar cualquier modelo, pero logra mejores resultados cuando se utiliza el modelo II.

El estándar MPEG permite una gran libertad en la forma en que se implementan los modelos. La complejidad del modelo que se implementa realmente en un codificador de audio MPEG dado

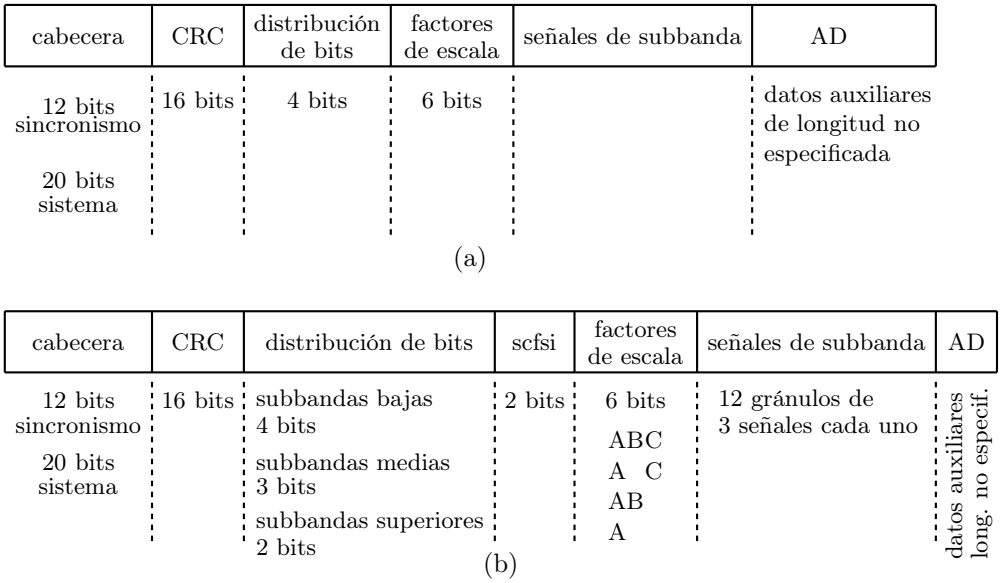


Figura 7.61: Organización de los frames de salida de las capas I y II.

depende del factor de compresión deseado. Para aplicaciones de consumo, donde los grandes factores de compresión no son críticos, el modelo psicoacústico puede ser completamente eliminado. En tal caso, el algoritmo de asignación de bits no utiliza una SMR (relación señal a máscara). Simplemente asigna bits a la subbanda con la SNR (relación señal a ruido) mínima.

Una descripción completa de los modelos está fuera del alcance de este libro y puede localizarse en el texto del estándar de audio MPEG [ISO/IEC 93], pp. 109–139. Los pasos principales de los dos modelos son los siguientes:

1. Se utiliza una transformada de Fourier para convertir las muestras de audio originales a su dominio de frecuencia. Esta transformación es independiente y distinta de aquélla de los filtros polifásicos porque los modelos necesitan una resolución de frecuencia más fina con el fin de determinar con precisión el umbral de enmascaramiento.

2. Las frecuencias resultantes están agrupadas en bandas críticas, no en las mismas 32 subbandas usadas por la parte principal del codificador.

3. Los valores espectrales de las bandas críticas se separan en componentes tonal (tipo sinusoidal) y monotonal (similar al ruido).

4. Antes de que los umbrales de enmascaramiento del ruido para las diferentes bandas críticas puedan ser determinados, el modelo aplica una función de enmascaramiento a las señales en las distintas bandas críticas. Esta función ha sido determinada empíricamente, por experimentación.

5. El modelo calcula un umbral de enmascaramiento para cada subbanda.

6. Para cada subbanda, se calcula la SMR (relación señal a máscara). Ésta es la energía de la señal en la subbanda dividida por el umbral de enmascaramiento mínimo para la subbanda. El conjunto de 32 SMRs, uno por cada subbanda, constituye la salida del modelo.

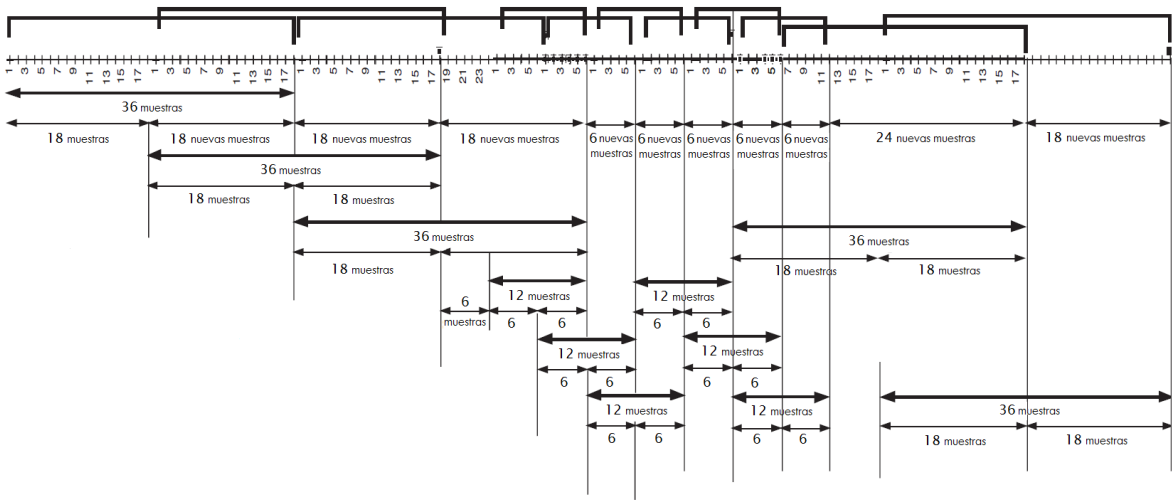


Figura 7.62: Superposición de ventanas MDCT.

7.14.6. Codificación: Capa III

La capa III emplea un algoritmo mucho más refinado y complejo que las dos primeras capas. Ésto se refleja en los factores de compresión, que son mucho más altos. La diferencia entre la capa III y las capas I y II comienza en el primer paso, el filtrado. Se utiliza el mismo banco de filtros polifásico (Tabla 7.49a), pero que es seguido por una versión modificada de la transformada discreta del coseno. La MDCT corrige algunos de los errores introducidos por los filtros polifásicos y subdivide las subbandas para acercarlas a la bandas críticas. El decodificador de la capa III debe usar la MDCT inversa, por lo que tiene que trabajar duro. La MDCT puede realizarse ya sea con un corto bloque de 12 muestras (produciendo seis coeficientes de transformada), ya sea con un largo bloque de 36 muestras (generando 18 coeficientes de transformada). Independientemente del tamaño de bloque elegido, los bloques consecutivos transformados mediante la MDCT adquieren una considerable superposición, como muestra la Figura 7.62. En esta figura, los bloques se muestran por encima de la línea gruesa, y los grupos resultantes de 18 ó 6 coeficientes están situados por debajo de la línea. Los bloques largos producen un mejor espectro de frecuencias de sonido estacionario (sonido donde las muestras adyacentes no difieren en mucho), mientras que los bloques cortos son preferibles cuando el sonido varía a menudo.

La MDCT utiliza n muestras de entrada x_k (donde n es uno de 36 ó 12) para obtener $n/2$ (i.e., 18 ó 6) coeficientes de transformada S_i. La transformada y su inversa vienen dados por:

$$S_i = \sum_{k=0}^{n-1} x_k \cos\left(\frac{\pi}{2n}\left[2k+1+\frac{n}{2}\right](2i+1)\right),\ i = 0, 1, \ldots, \frac{n}{2}-1, \tag{7.14}$$

$$x_k = \sum_{k=0}^{n/2-1} S_i \cos\left(\frac{\pi}{2n}\left[2k+1+\frac{n}{2}\right](2i+1)\right),\ i = 0, 1, \ldots, n-1. \tag{7.15}$$

El tamaño de un bloque corto es un tercio del de un bloque largo, por lo que pueden mezclarse. Cuando un frame es construido, la MDCT puede utilizar todos los bloques largos, todos los bloques cortos (tres veces más), o bloques largos para las dos subbandas de menor frecuencia y bloques cortos para las 30 subbandas restantes. Ésto es un compromiso donde los bloques largos proporcionan una

resolución en frecuencia más fina para las frecuencias más bajas, donde es más útil, y los bloques cortos mantienen una mejor resolución en tiempo para las frecuencias altas.

Dado que la MDCT proporciona una mejor resolución en la frecuencia, tiene que dar lugar a una resolución más pobre en el tiempo debido al principio de incertidumbre (Sección 5.3). Lo que sucede en la práctica es que la cuantificación de los coeficientes MDCT provoca errores que se propagan con el tiempo y causan distorsiones audibles que se manifiestan como preecos (léase: "pre-ecos").

El modelo psicoacústico utilizado por la capa III tiene características adicionales que detectan condiciones de preecos. En tales casos, la capa III utiliza un complejo algoritmo de distribución de bits que se apropia de bits del repositorio de bits disponibles con el fin de aumentar temporalmente el número de niveles de cuantificación y, por lo tanto, reducir los preecos. La capa III también puede cambiar a bloques MDCT cortos, lo que reduce el tamaño de la ventana de tiempo, si se "sospecha" que las condiciones son favorables para los preecos.

(El modelo psicoacústico de la capa III calcula una cantidad llamada "entropía psicoacústica" (PE), y el codificador de capa III "sospecha" que las condiciones son favorables para los preecos si PE > 1800.)

Los coeficientes MDCT pasan a través de algún tipo de procesamiento para eliminar los artefactos causados por la superposición de frecuencia de las 32 subbandas. Ésto se conoce como reducción de aliasing. sólo los bloques largos se envían al procedimiento de *reducción de aliasing*. La MDCT utiliza 36 muestras de entrada para calcular 18 coeficientes, y el procedimiento de reducción de aliasing utiliza una operación mariposa (*butterfly*) entre dos conjuntos de 18 coeficientes. Esta operación se ilustra gráficamente en la Figura 7.63a con un código en lenguaje C listado en la Figura 7.63b. El índice i en la figura es la distancia desde la última línea del bloque anterior a la primera línea del bloque actual. Se calculan ocho mariposas, con diferentes valores de los pesos cs_i y ca_i que vienen dados por:

$$cs_i = \frac{1}{\sqrt{1+c_i^2}}, \quad ca_i = \frac{c_i}{\sqrt{1+c_i^2}}, \quad i = 0, 1, \ldots, 7.$$

Los ocho valores c_i especificados por el estándar son $-0{,}6$, $-0{,}535$, $-0{,}33$, $-0{,}185$, $-0{,}095$, $-0{,}041$, $-0{,}0142$, y $-0{,}0037$. Las Figuras 7.63c,d muestran los detalles de una sola mariposa para el codificador y el decodificador, respectivamente.

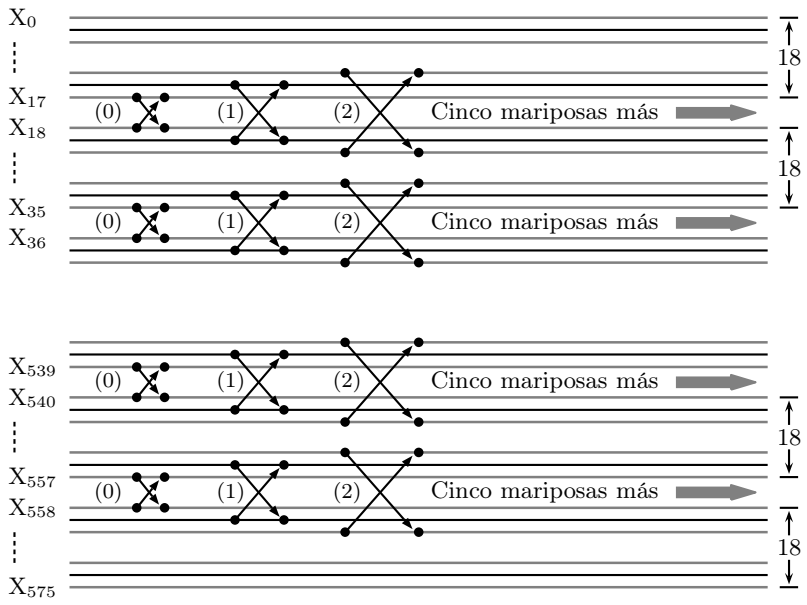
La cuantificación en la capa III es no uniforme. El cuantificador eleva los valores a cuantificar a la potencia $3/4$ antes de la cuantización. Ésto proporciona una SNR más consistente. El decodificador revierte esta operación descuantificando un valor y elevándolo a la potencia $4/3$. La cuantificación se lleva a cabo mediante:

$$is\,(i) = \text{nint}\left[\left(\frac{xr\,(i)}{\text{quant}}\right)^{3/4} - 0{,}0946\right], \tag{7.16}$$

donde $xr\,(i)$ es el valor absoluto de la señal de la subbanda i, "quant" es la cuantificación del tamaño de paso, "nint" es la función "número entero más cercano", y $is\,(i)$ es el valor cuantificado. Como en las capas I y II, la cuantificación es *midtread*, i.e., los valores en torno a 0 se cuantifican a 0 y el cuantificador es simétrico alrededor del 0.

En las capas I y II, cada subbanda puede tener su propio factor de escala. La capa III utiliza *bandas* de factores de escala. Estas bandas cubren varios coeficientes MDCT cada una, y sus anchos están cercanos a las anchuras de las bandas críticas. Existe un algoritmo de distribución de ruido que selecciona los valores para los factores de escala.

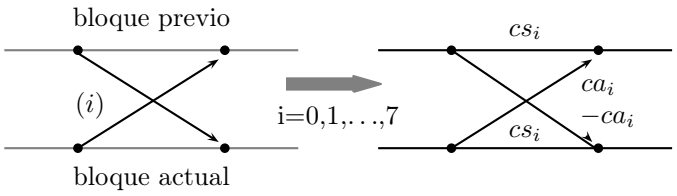
La capa III utiliza los códigos de Huffman para comprimir aún más los valores cuantificados. El codificador produce 18 coeficientes MDCT por subbanda. Ordena los 576 coeficientes resultantes ($= 18 \times 32$) por orden creciente de frecuencia (para bloques cortos, hay tres conjuntos de coeficientes dentro de cada frecuencia). Nótese que los 576 coeficientes MDCT corresponden a 1152 muestras de audio transformadas. El conjunto de coeficientes ordenados se divide en tres regiones, y cada región se codifica con un conjunto diferente de códigos de Huffman. Ésto se debe a que los valores en cada región



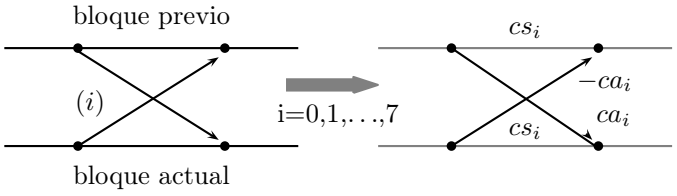
(a)

```
for(sb=1; sb < 32; sb++)
  for(i=0; i < 8; i++) {
    xar[18 * sb - 1 - i] = xr[18 * sb - 1 - i]cs[i] - xr[18 * sb + i]ca[i]
    xar[18 * sb + i] = xr[18 * sb + i]cs[i] + xr[18 * sb - 1 - i]ca[i]
                        }
```

(b)



(c)



(d)

Figura 7.63: Reducción de aliasing en mariposa en la capa III.

Campo nombre	Canal único	Doble canal
Datos principales	9	9
bits privados =	5	3
scfsi:	4	8
parte2 longitud3 =	12×2	12×4
valores grandes =	9×2	9×4
ganancia global =	8×2	8×4
compresión factor de escala =	4×2	4×4
preflag:	1×2	1×4
escalado de factor de escala:	1×2	1×4
selección de tabla count1 =	1×2	1×4
ventana de intercambio de flag =	1×2	1×4
información adicional =	22×2	22×4
(dependiente de la ventana de intercambio de flag)		
Total (bits)	136	256

Tabla 7.64: Tamaño de infor. adicional para audio MPEG de la capa III.

tienen una distribución estadística diferente. Los valores de para las frecuencias más altas tienden a ser pequeños y tienen rachas (*runs*) de ceros, mientras que los valores para las frecuencias más bajas tienden a ser grandes. Las tablas de códigos son proporcionadas por el estándar (las 32 tablas de las páginas 54–61 de [ISO/IEC 93]). La división de los valores cuantificados en tres regiones también ayuda a controlar la propagación de errores.

Comenzando con los valores de frecuencia más altos, donde hay muchos ceros, el codificador selecciona la primera región como el run de ceros continuo de la frecuencia más alta. Es raro, pero posible, no tener ningún run de ceros. El run se limita a un número par de ceros. Este run no tiene que ser codificado, ya que su valor se puede deducir de los tamaños de las otras dos regiones. Su tamaño, sin embargo, debe ser uniforme, ya que las otras dos regiones codifican sus valores en grupos de numeración par.

La segunda región consta de un run continuo de los tres valores -1, 0, y 1. Ésta se denomina región "count1". Cada código de Huffman para esta región codifica cuatro valores consecutivos, por lo que el número de códigos debe ser $3^4 = 81$. La longitud de esta región debe, por supuesto, ser un múltiplo de 4.

La tercera región, conocida como la región "de valores grandes", se compone de los valores restantes. Opcionalmente, es dividido en tres subregiones, cada una con su propia tabla de códigos de Huffman. Cada código de Huffman codifica dos valores.

La tabla de códigos Huffman más grande especificada por el estándar tiene 16×16 códigos. Valores mayores se codifican utilizando un mecanismo de códigos de escape.

Un frame F en la capa III se organiza de la siguiente manera: Comienza con la cabecera de 32 bits habitual, que es seguida por el (opcional) CRC de 16 bits. Ésto es seguido por 136 bits (para un único canal) y 256 bits (para doble canal) de información adicional. El tamaño de la información adicional para el MPEG/Audio de la capa III, para canal único y dual, se muestra en la Tabla 7.64. La última parte del frame constituye los datos principales. La información adicional es seguida por un segmento de los datos principales (la información adicional contiene, entre otros datos, la longitud del segmento) pero los datos en este segmento ¡no tienen por qué ser los del frame F! El segmento puede contener datos de varios frames debido al uso por el codificador de un *repositorio de bits*.

El concepto del repositorio de bits ya ha sido mencionado. El codificador puede prestar los bits de este repositorio cuando decide aumentar el número de niveles de cuantificación porque sospecha que pueden producirse preecos. El codificador también puede donar bits al repositorio cuando necesita

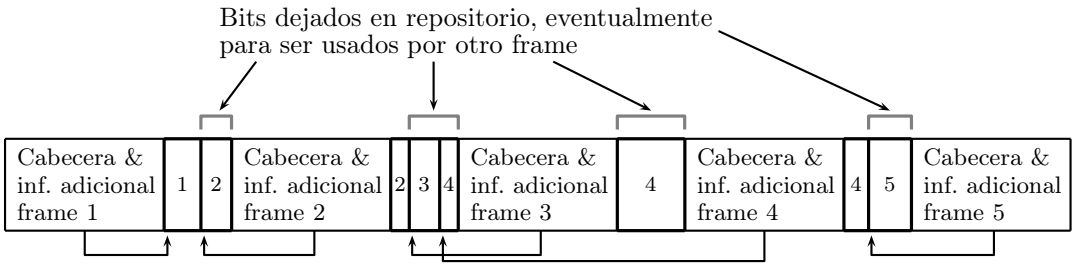


Figura 7.65: Stream comprimido en la capa III.

menos que el número medio de bits para codificar un frame. Los préstamos, sin embargo, sólo pueden hacerse a partir de donaciones previas; el repositorio no puede tener un número negativo de bits.

La información adicional de un frame incluye un puntero de 9 bits que apunta al inicio de los datos principales para el frame, y el concepto completo de datos principales, segmentos de tamaño fijo, punteros, y repositorios de bits se ilustra en la Figura 7.65. En este diagrama, el frame 1 necesitaba sólo la mitad de su asignación de bits, por lo que dejó la otra mitad en el repositorio, donde fue eventualmente utilizado por el frame 2. El frame 2 necesitaba un poco de espacio adicional en su "propio" segmento, dejando el resto del segmento en el repositorio. Ésto fue eventualmente utilizado por los frames 3 y 4. El frame 3 no necesitaba nada de su propio segmento, por lo que el segmento entero se dejó en el repositorio, y fue finalmente utilizado por el frame 4. Ese frame también necesitaba parte de su propio segmento, y el resto fue utilizado por el frame 5.

La distribución de bits en la capa III es similar a la de las otras capas, pero incluye la complejidad añadida de la distribución del ruido. El codificador (Figura 7.66) calcula la distribución de bits, efectúa la cuantificación real de las señales de subbanda, las codifica mediante códigos de Huffman, y cuenta el número total de bits generados en este proceso. Ésta es la asignación de bits del bucle interior. El algoritmo de distribución de ruido (también llamado procedimiento de análisis mediante síntesis) se convierte en un bucle externo, donde el codificador calcula el ruido de cuantificación (i.e. descuantifica y reconstruye las señales de subbanda y calcula las diferencias entre cada señal y su contraparte reconstituida). Si encuentra que ciertas bandas de factor de escala tienen más ruido de lo que el modelo psicoacústico permite, el codificador aumenta el número de niveles de cuantificación para estas bandas, y repite el proceso. Este proceso termina cuando cualquiera de las tres condiciones siguientes es verdadera:

1. Todas las bandas de factor de escala tienen el ruido permitido o menos.

2. La siguiente iteración podría requerir una recuantificación de TODAS las bandas de factor de escala.

3. La siguiente iteración podría necesitar más bits de los que están disponibles en el repositorio de bits.

◇ **Ejercicio 7.13 (sol. en pág. 1107):** La capa III es extremadamente compleja y dura de implementar. En vista de ésto, ¿cómo es que hay tantos programas gratuitos o de bajo costo disponibles para reproducir archivos de audio `.mp3` en todas las plataformas informáticas?

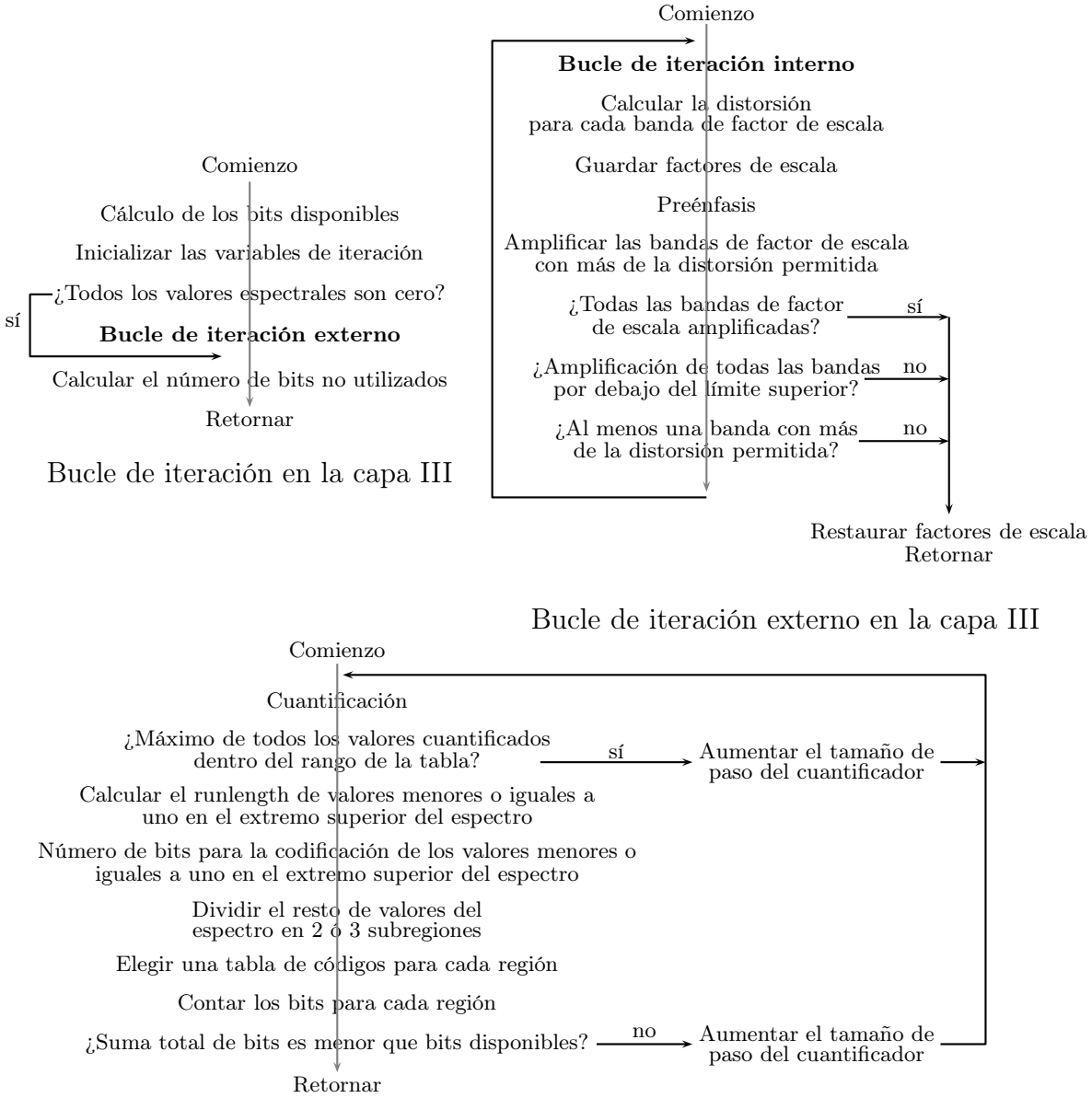


Figura 7.66: Iteración de bucle en la capa III.

7.15. Codificación de Audio Avanzada (AAC)

El desarrollo, por Philips y Sony, de los discos compactos familiares (CD), comenzó alrededor de 1974. En junio de 1980, las dos compañías acordaron un estándar de CD común y en 1981 este estándar fue aprobado por el comité de Disco de Audio Digital (*Digital Audio Disc*). (Nótese la ortografía `disc`, en lugar de `disk`; la última palabra identifica un disco magnético.) El estándar, que ha sido utilizado desde entonces, incluye especificaciones de hardware y de software del formato de la señal, el material y las dimensiones del disco, los códigos correctores de errores, y muchas otras características. Lo que nos parece inesperado e inusual, sin embargo, es que nuestros CDs de música graban el audio en un formato no comprimido. Las muestras de audio sin tratar (*raw*), típicamente 44 100 muestras de 16 bits por segundo de audio, son escritas en el disco con un código de corrección de errores sofisticado, pero sin ninguna intención de comprimirlos.

Los CD actuales tienen una capacidad de unos 80 minutos de sonido. Ésto puede alojar muchas canciones, y es suficiente para un concierto o dos y para la mayoría de las sinfonías (aunque la tercera sinfonía de Mahler, de unos 100 minutos, es una notable excepción que viene a la mente). Una ópera, sin embargo, requiere de dos o más CDs (el ciclo del Anillo de Wagner, que se abarca 18 horas, necesita 14–16 CDs), como también lo hacen las obras completas de muchos compositores. En consecuencia, la compresión puede ser muy útil. Actualmente, el siempre popular formato mp3 logra rutinariamente factores de compresión de 18–20 y puede reducir una biblioteca de música de 100 CDs a sólo cinco o seis CDs, y eventualmente a un único DVD.

Teniendo en cuenta todo esto, es natural preguntarse acerca de la falta de compresión de audio en el estándar CD original. La mejor explicación es que los ingenieros simplemente no creían que el audio se podía comprimir con pérdidas y eficientemente, a quizás al 20 % de su tamaño tamaño, sin pérdidas de calidad apreciables. Un término común usado por los expertos en audio en la década de 1970 y 80 fue "orejas de oro", que se refiere a los audiófilos que afirman que pueden distinguir entre la música en vivo y la música grabada, sobre todo música que se reproduce desde un archivo en formato comprimido. En consecuencia, cuando el grupo de expertos en imágenes en movimiento (*Moving Pictures Experts Group* o MPEG) entró en funciones, se concentró en la compresión de vídeo. Fue sólo en 1988 que varios pioneros del audio formaron el grupo de audio junto a MPEG (véase también la página 791) y comenzó el proceso que nos proporcionó las tres capas MPEG-1 y MPEG-2, seguidas por el estándar de compresión de audio avanzado (AAC).

El estándar de compresión AAC fue originalmente parte del proyecto MPEG-2, y fue posteriormente aumentado y ampliado como parte de MPEG-4. Algunas referencias importantes para este eficiente y complejo método son: [Bosi y Goldberg 03], [Brandenburg 99], [Pereira y Ebrahimi 02], e [ISO/IEC 03] (este último es la especificación formal del estándar y no es muy legible). AAC es el resultado de un esfuerzo internacional intensivo que involucra compañías y particulares. El proyecto se inició en noviembre de 1994, cuando una serie de propuestas fueron presentadas al comité de MPEG-2. El comité surgió con una estructura preliminar de AAC de un conjunto de módulos que constituyen partes independientes del sistema global. Cada módulo ha sido desarrollado, implementado y probado por separado antes de convertirse en parte del producto final. Los módulos principales son los siguientes:

- Control de ganancia.
- Banco de filtros.
- Predicción.
- Cuantificación y codificación.
- Codificación sin ruido de Huffman.

- Multiplexación de cadenas de bits (*bitstreams*).
- Conformación de ruido temporal (*Temporal noise shaping* o TNS).
- Codificación de estéreo Medio/secundario (Mid/side o M/S).
- Codificación de estéreo de intensidad.

Estos módulos se muestran en las Figuras 7.72 y 7.73 y la mayoría de ellos se describen en esta sección.

Hoy en día, prácticamente todos los productores y consumidores de audio están de acuerdo en que la compresión de audio (tanto con pérdidas como sin pérdidas) es efectiva y muy útil (nótese la popularidad de los reproductores de mp3). Incluso más, aquéllos familiarizados con los comités de MPEG y la forma en que operan están de acuerdo en que el proceso de publicación de los algoritmos MPEG, la selección de candidatos prometedores, y la implementación y prueba de los mismos, ha dado lugar a mejores métodos de compresión de audio, alentando al mismo tiempo la compatibilidad entre muchos tipos diferentes de equipos de audio.

Otro aspecto del éxito de MPEG es que los métodos principales utilizados actualmente para la compresión de audio provienen de esta organización y no de los desarrolladores comerciales que tratan de bloquear a los usuarios, los algoritmos y el software propietario. La principal razón del éxito del audio MPEG es su proceso de desarrollo. Todo el proceso de selección, las pruebas y la adopción de un estándar de compresión es abierto y competitivo. El comité de audio de MPEG incluye a investigadores académicos y representantes de la industria. Se reúnen periódicamente para establecer objetivos para el siguiente método, para publicar esos objetivos, para recibir, estudiar, y discutir ideas de muchos colaboradores, y para implementar y probar las ideas más prometedoras.

El primer éxito del grupo de audio de MPEG llegó en 1991 con MPEG-1. Este primer estándar de compresión de vídeo ya incorpora las tres capas tratadas en la Sección 7.14. MPEG-2 se finalizó en 1997. Añadió algunas mejoras menores a las tres capas, pero su principal contribución a la compresión de audio fue la primera versión del estándar [ISO/IEC 03] de compresión de audio avanzada (*Advanced Audio Compression* o AAC).

[La capa 1 de MPEG-1/2 raramente se utiliza. La capa 2 se usa en el audio digital de radiodifusión (*Digital Audio Broadcasting* o DAB) en Europa, en el audio para el vídeo, y en sistemas de distribución de difusión. La capa 3 es, por supuesto, ampliamente utilizada —bajo el nombre mp3, derivada de su extensión de archivo— en productos de consumo, así como en los codificadores de difusión.]

Por consiguiente, el nombre completo de mp3 es MPEG-1 y MPEG-2 de capa 3. Un nombre un poco más corto es MPEG-1/2 de capa 3. Los usuarios y lectores han observado durante mucho tiempo que el término "capas" es confuso y desafortunado. Tal vez "niveles" o "versiones" habría sido una mejor elección. También hay a menudo una confusión de MPEG-2 con el nivel 2.

MPEG-3 fue pensado para la compresión de HDTV, pero se encontró que era redundante (era muy similar a MPEG-2) y se fusionó con MPEG-2 (Sección 6.5).

El siguiente diseño de MPEG, MPEG-4, se completó en 1999 (Sección 6.6). Incorporaba varias mejoras en AAC y añadió el módulo AAC-LD (de *Low Delay*, baja latencia, Sección 7.15.3).

El trabajo sobre MPEG-7 se encuentra actualmente en marcha, pero muchas almas curiosas preguntan una cuestión natural: ¿Qué pasa con los MPEGs 5 y 6? La literatura oficial de MPEG no dice nada acerca de este salto en la numeración, pero parece (al menos ésto es lo que un participante en las reuniones sobre MPEG recuerda) que alguien propuso continuar la secuencia entera 1, 2, y 4, con 8 (porque se trata de las primeras potencias de 2 y se escriben en binario como $10\ldots0_2$). En respuesta, alguien propuso (tal vez como una broma) seleccionar el 7 en lugar del 8 porque MPEG-7 es supuestamente tan diferente como sus predecesores y porque 7 es un número de la suerte.

¿Por qué el 7 es considerado un número de la suerte? Nadie puede decirlo con certeza, pero aquí hay siete ejemplos que justifican este título:

- Los pitagóricos llamaban 7 al número perfecto. Llamaron a 3 y 4 (el triángulo y el cuadrado) las figuras perfectas.
- Los antiguos conocían siete planetas, el sol, la luna, Mercurio, Venus, Marte, Júpiter y Saturno. Similarmente, hubo siete maravillas en el mundo antiguo.
- En las antiguas escrituras y la tradición, el número 7 tiene un papel importante. La Biblia menciona este número muchas veces (los 7 días de la semana, los 7 años sabáticos, los 7 años de hambre, los 7 años de abundancia, los 7 años ocupados en la construcción del Templo del Rey Salomón). Los árabes tenían siete templos sagrados. En los misterios persas existían siete amplias cavernas a través de las cuales los aspirantes tenían que pasar. Los godos tenían siete deidades, al igual que los romanos, de cuyos nombres se derivan nuestros días de la semana. Esta preocupación por el 7 ha encontrado su camino hacia los tiempos modernos en forma de Blancanieves y los siete enanitos, y otras obras de la literatura.
- Los siete pecados capitales (introducidos por San Gregorio el Grande alrededor del 600) son: el orgullo, la envidia, la ira, la avaricia, la tristeza, la gula y la lujuria.
- El cuerpo de bomberos de Trumpton (Pugh, Pugh, Barney, Magrew, Cuthburt, Dibble y Grub). Trumpton fue un programa de televisión para niños en stop-motion emitido en 1967.
- Hasta el siglo 15, se creía que el mundo tenía siete mares. El mar Rojo, el Mediterráneo, el golfo Pérsico, el mar Negro, el mar Adriático, el mar Caspio, y el océano Índico.
- En la cultura china, el séptimo día de la primera luna del año lunar se conoce como el Día del ser humano y se celebra como el cumpleaños universal de todos los seres humanos.

Los científicos, ingenieros y expertos que contribuyeron a las tres capas de MPEG-1 y MPEG-2 se han topado con un enfoque práctico y eficaz de la compresión de audio, la llamada codificación perceptual. La idea es identificar las partes de un stream de audio que no son totalmente percibidas por el sistema oído/cerebro y cuantificarlas o incluso eliminarlas por completo. Lo que en realidad se cuantifica son los coeficientes de frecuencia (a menudo conocidos como coeficientes espectrales) y no las propias muestras de audio. Este es el principio de enmascaramiento acústico que hizo posible, primero mp3 y posteriormente, AAC.

El oído humano (Sección 7.3) es muy sensible, pero no es un instrumento de precisión. Su sensibilidad depende en gran medida de la frecuencia del sonido (así como de otros factores como la edad y el entorno). La Figura 7.67 muestra el umbral normal del oído (una curva determinada por muchos experimentos sensitivos) y cómo depende de la frecuencia (véase también la Figura 7.5a). Observe que la escala de frecuencia es logarítmica debido a su amplio intervalo (de 20 Hz a 20 kHz). Cualquier sonido por debajo de este umbral es inaudible, y la figura muestra que el umbral es alto tanto en frecuencias bajas como altas. Para que un sonido a estas frecuencias pueda ser audible, tiene que ser fuerte; su amplitud tiene que superar el umbral. El umbral es más bajo en el rango de frecuencias de 3–5 kHz, lo que indica que el oído es muy sensible a esta distancia y puede detectar sonidos muy tenues.

La cuestión es que el umbral normal puede ser momentáneamente perturbado por sonidos fuertes. La figura muestra cómo un sonido fuerte con una frecuencia de 300 Hz eleva umbral normal de las frecuencias de 80 Hz hasta aproximadamente 1 kHz. El resultado es que el sonido etiquetado “x”, a 150 Hz, que está por encima del umbral normal, ahora se encuentra por debajo del nuevo umbral perturbado, y por lo tanto está enmascarado; no puede ser oído, y sus muestras de audio pueden eliminarse (véase también la Figura 7.5b).

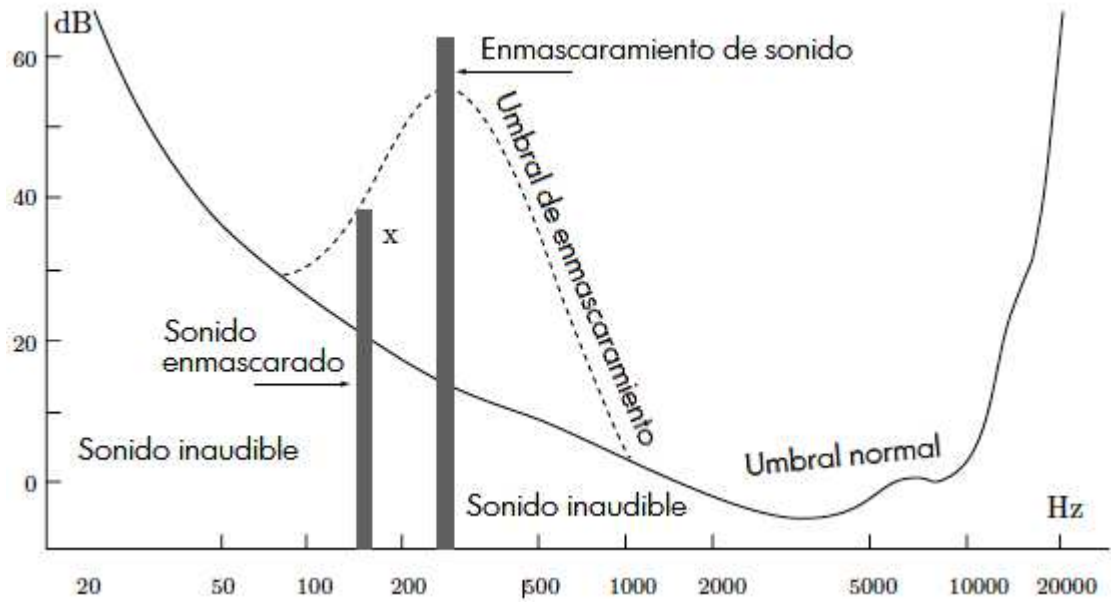


Figura 7.67: Enmascaramiento de la frecuencia de audio.

El tiempo es también un factor de enmascaramiento acústico. Cualquier perturbación del umbral normal es temporal y decae normalmente en menos de un segundo. La Figura 7.68 muestra un sonido fuerte, de 60 dB (a frecuencia f) que tiene una duración de 5 ms. Un nuevo umbral comienza a 60 dB y decae casi por completo en aproximadamente 500 ms. Los sonidos en o alrededor de la frecuencia f que ocurren por debajo de este umbral son inaudibles. El sonido "x" a 30 dB es inaudible, ya que se produce sólo 5 ms después del enmascaramiento de sonido, pero el mismo sonido de 30 dB "y" que ocurre 10 ms más tarde es totalmente audible. El oído se ve abrumado por el fuerte sonido hasta tal punto que no puede percibir otros sonidos durante un tiempo.

La Figura 7.69 ilustra el concepto de enmascaramiento acústico en frecuencia y tiempo a la vez. Un enmascaramiento de sonido (en forma de rectángulo) crea una superficie que se extiende sobre las frecuencias vecinas y sobre el tiempo. Cualesquier sonidos con amplitudes y frecuencias bajo esta

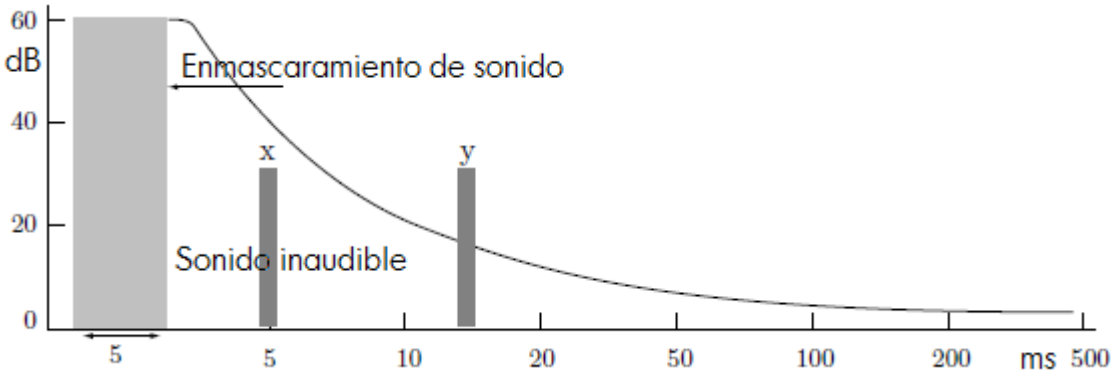


Figura 7.68: Enmascaramiento de audio en el tiempo.

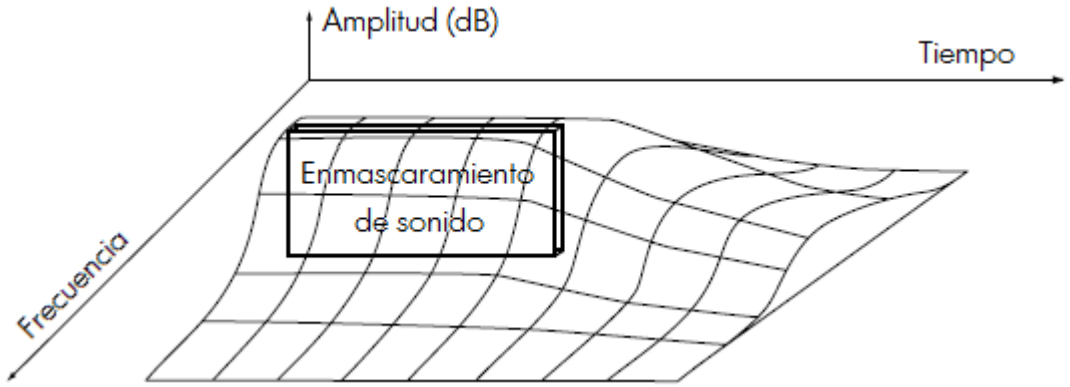


Figura 7.69: Enmascaramiento de audio en frecuencia y tiempo..

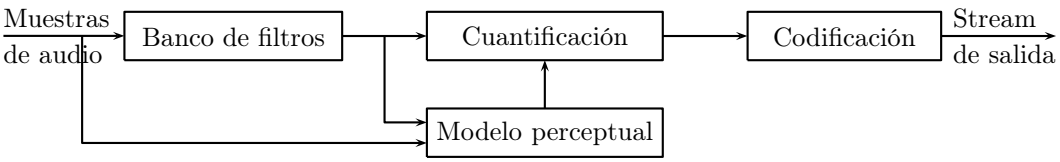


Figura 7.70: Diagrama de bloques de un codificador perceptual.

superficie están enmascarados.

En base a este comportamiento del oído, un algoritmo de compresión de audio con pérdidas en tiempo/frecuencia (T/F) trabaja siguiendo estos pasos (Figura 7.70):

- El codificador repasa las muestras de audio en el stream de entrada y las agrupa en rangos (ventanas) de muestras solapadas. Cada ventana es filtrada a continuación para convertir sus muestras de audio en coeficientes de frecuencia. Cada coeficiente corresponde a un pequeño intervalo de frecuencias (una banda de frecuencia) e indica la intensidad del sonido en la banda.
- Utiliza los coeficientes de frecuencia para identificar los sonidos de enmascaramiento.
- Un modelo de percepción (o psicoacústico), estima entonces la altura de la curva de máscara en cada banda de frecuencia. Esta altura está determinada por el umbral normal y por perturbaciones del mismo, causadas por el enmascaramiento de sonidos en diversas frecuencias y en distintos momentos en el pasado.
- Cuando la curva de umbral actualizada está lista, se utiliza para determinar los coeficientes de frecuencia que no son importantes. Estos coeficientes son cuantificados ahora.
- Los coeficientes cuantificados son codificados mediante un código de tamaño variable.
- Los códigos se escriben en el stream de salida, junto con otra información adicional que necesita el decodificador.

Un punto importante es que el enmascaramiento de audio depende de la frecuencia del sonido, pero los datos a ser comprimidos están en forma de muestras de audio. En consecuencia, la primera tarea del codificador es tomar un conjunto de muestras de audio y calcular las frecuencias de las ondas sonoras generadas por ellas. (En la práctica, se identifican los intervalos de frecuencias, conocidos

como bandas de frecuencias, y se calcula un coeficiente de frecuencia (o espectral) por banda. Las bandas no deben exceder la anchura de las bandas críticas del oído discutidas en la Sección 7.3.) Ésto se efectúa por medio de un banco de filtros polifásicos. La forma en que ésto se lleva a cabo en la capa 3 (mp3) se discute en la Sección 7.14.1, y en particular en la Figura 7.48. En mp3, las 512 muestras de audio más recientes se almacenan en una memoria intermedia (*buffer*) y se utilizan para calcular los coeficientes espectrales para 32 bandas de frecuencia. Cada coeficiente espectral indica la amplitud del audio (específicamente, la parte del audio descrita por las 512 muestras) en una banda de frecuencia. Las siguientes 32 muestras de audio se desplazan entonces en el buffer y se repite el proceso. AAC realiza un cálculo similar, pero usa un buffer con 2048 muestras de audio, calcula 1024 coeficientes espectrales, cada uno indicando la amplitud del audio en una banda de frecuencia de 23,4 Hz de ancho, y desplaza las siguientes 1024 muestras en el buffer.

[Los coeficientes espectrales se utilizan más tarde por el decodificador para reconstruir las muestras de audio, pero estas muestras son diferentes de las originales porque los coeficientes espectrales son cuantificados para lograr la compresión.]

El siguiente componente de mp3 es un modelo psicoacústico (Sección 7.14.5). Éste determina la altura del umbral de enmascaramiento para cada banda de frecuencia. Sin embargo, el modelo no se especifica en detalle y la idea es que cada aplicación selecciona su propio modelo y el éxito de una implementación particular depende de la complejidad del modelo y de su semejanza con el comportamiento real del sistema oído/cerebro. La especificación del estándar AAC emplea similarmente un modelo psicoacústico sin especificar su funcionamiento preciso. (Véase el diagrama de flujo en la página 144 de [ISO/IEC 03].)

Para cada banda de frecuencias, el codificador compara el coeficiente espectral para el umbral de la banda. Si el coeficiente espectral es menor que el umbral, el coeficiente es cuantificado. Ésta es la principal fuente de compresión en las tres capas de MPEG-1/2 así como en AAC, y es con pérdidas. La cantidad de cuantificación depende de cuánto más pequeño es el coeficiente que el umbral. La cantidad de cuantización es crucial. Si un coeficiente se cuantifica demasiado, el decodificador podría generar muestras de audio muy diferentes del original, lo que produce un ruido perceptible y una mala calidad del audio. Por otra parte, el resultado de una cuantificación insuficiente (demasiado buena) es la baja compresión que no alcanza la tasa de bits de destino especificada por el usuario.

AAC emplea varias herramientas que mejoran la cuantificación de la siguiente manera:

- El modelado de ruido temporal (*Temporal Noise Shaping* o TNS) es un algoritmo sofisticado que minimiza el efecto de la propagación temporal. Ésto mejora sobre todo la cuantificación (y por lo tanto la compresión) de las señales de voz.
- Un módulo de predicción mejora el rendimiento del cuantificador en los casos donde el audio original presenta patrones, tales como una alta tonalidad (sonido que se asemeja a una sinusoide).
- El modelado de ruido perceptual (*Perceptual noise shaping* o PNS) proporciona un control más preciso de la cuantificación, ahorrando bits y mejorando la compresión.

El siguiente paso es la codificación. Los coeficientes de frecuencia cuantificados son codificados. Este paso mejora la compresión porque los coeficientes son reemplazados por códigos de tamaño variable, pero esta compresión adicional es sin pérdidas (*lossless*) —la literatura sobre AAC se refiere a ella como "silenciosa (*noiseless*)"—. Tanto mp3 como AAC utilizan códigos de Huffman. El primer proceso de codificación se describe en la Sección 7.14.6. La última utiliza 12 tablas de códigos de Huffman. Para seleccionar una tabla de códigos, el codificador selecciona un grupo de dos o cuatro coeficientes de frecuencia cuantificados consecutivos y selecciona una tabla de códigos en función del tamaño del grupo y del mayor valor absoluto de los coeficientes en el grupo. Se proporcionan códigos de Huffman para los coeficientes de hasta ± 16, pero también existe un mecanismo de escape que permite la codificación de coeficientes de hasta $\pm 8\,191$.

El último paso es recoger los códigos de Huffman, agregar flags o banderas, códigos de control y otra información necesaria para el decodificador, y multiplexar todo esto para crear un stream de salida final.

Dos características inusuales de mp3 son el repositorio de bits y el modo estéreo conjunto. El usuario indica la calidad de compresión mediante la especificación de una tasa de bits. Una tasa de bits de 64 $^{\text{Kb}}/_{\text{seg.}}$, por ejemplo, dirige al codificador a comprimir cada segundo de audio en sólo 64 K bits. Para un sonido estéreo, una tasa de muestreo de 44 100 muestras por segundo genera 88 200 muestras (equivalente a 176 400 bytes ó 1 411 200 bits) por segundo. La compresión de 1 411 200 bits en 65 536 bits implica un factor de compresión de aproximadamente 21,5. Ésto es impresionante, pero puede dar lugar a la degradación del sonido reconstruido. En consecuencia, mp3 mantiene un repositorio de bits. Cada vez que un conjunto de coeficientes es cuantificado, codificado, y emitido, el codificador estima cuántos bits se permiten para el conjunto según la tasa de bits especificada por el usuario y cuántos bits se utilizan realmente. Todos los bits "no utilizados" se añaden al repositorio de bits para ser usados en conjuntos futuros si fuera necesario. Si se utilizan demasiados bits para la compresión del conjunto actual, los bits adicionales se substraen del depósito. Si el depósito no tiene suficientes bits, el codificador repite su operación con una cuantificación más gruesa, lo que ahorra bits, pero produce un sonido mal reconstruido. El modo estéreo conjunto se beneficia de la correlación entre los canales de audio estéreo izquierdo y derecho.

Por consiguiente, AAC comparte las características básicas de mp3, con las siguientes mejoras:

- Un banco de filtros más grande y mejorado que calcula 1 024 coeficientes espectrales para cada ventana de 2 048 muestras de audio, lo que genera bandas de frecuencia estrechas y una resolución de frecuencia mucho más fina que la de mp3.
- El modelado de ruido temporal (TNS), un nuevo algoritmo que minimiza el efecto del enmascaramiento temporal. Ésto es especialmente útil para la compresión de señales de voz.
- Un módulo de predicción mejora la cuantificación para el audio periódico o el audio con patrones.
- El modelado de ruido perceptual (PNS) proporciona un control de la cuantificación que conduce a una mejor compresión.

En abril de 2003, Apple Computer atrajo la atención del público a AAC con el anuncio de que sus productos iTunes e iPod soportarían el formato de audio AAC MPEG-4 (los viejos iPods requerían una actualización del firmware). Los clientes pueden descargar la música desde el repositorio de música de iTunes (*iTunes Music Store*) en una versión protegida del formato (que utiliza las extensiones de archivo `m4a` y `m4v`). Por esta razón, AAC se ha vuelto a asociar con el hardware y software de Apple, por lo que muchos creen erróneamente que AAC procede de "Apple Audio Codec" en lugar de "Advanced Audio Coding".

> MP4 también se refiere a la teoría de perturbaciones de cuarto orden de Mϕller-Plesset.

7.15.1. Detalles de la codificación avanzada de audio (AAC)

Comenzamos con los detalles de la AAC tal como se definen en MPEG-2. La Sección 7.15.2 discute las características añadidas a AAC por MPEG-4. Las Figuras 7.72 y 7.73 son diagramas de bloques del codificador y decodificador AAC, respectivamente.

AAC soporta tres *perfiles*. Éstos constituyen las distintas configuraciones del algoritmo básico, y ofrecen diferentes soluciones de compromiso entre la eficiencia de compresión y la complejidad de

Índice	Perfil
0	Principal
1	Baja complejidad (*Low Complexity* o LC)
2	Tasa de muestreo escalable (*Scalable Sampling Rate* o SSR)
3	Reservado

Tabla 7.71: Los tres perfiles de AAC.

codificación. Un índice de perfil de 2 bits es escrito en el stream comprimido (Tabla 7.71) para indicar el perfil al decodificador.

Índice 0. Perfil principal. En este perfil, AAC ofrece la mejor compresión para cualquier frecuencia de muestreo y tasa de bits. Se emplean todos los módulos, rutinas y herramientas de la especificación AAC (con excepción de la herramienta de control de ganancia). Éste es el perfil normal y tiene sentido cuando están disponibles memoria y potencia de procesamiento suficientes (como es habitual en las computadoras actuales).

Índice 1. Perfil de baja complejidad (*Low Complexity* o LC). La herramienta de control de ganancia y de predicción no se utilizan. Además, el orden de TNS está limitado a 12. Por lo demás, este perfil es un subconjunto del perfil principal, lo que significa que un stream comprimido generado por LC puede ser decodificado por el perfil principal.

Índice 2. Tasa de muestreo escalable (*Scalable Sampling Rate* o SSR). Este perfil está diseñado para ser simplificado (para reducir la complejidad) cuando los datos de audio originales constan de frecuencias limitadas (ancho de banda de audio reducido). Se requiere la herramienta de control de ganancia en SSR y sólo es usado por SSR (pero no se utiliza en la más baja de las cuatro subbandas PQF). La predicción y los canales de acoplamiento no se utilizan. Asimismo, el orden de TNS y el ancho de banda están limitados. Un stream comprimido generado por LC puede ser decodificado por SSR, pero el audio resultante se limitará a (aproximadamente) 5 kHz debido a que la banda más baja del primer banco de filtros no se utiliza. SSR es más simple que los otros perfiles.

El Índice 3 está reservado para un perfil futuro.

Control de ganancia. Esta herramienta es un preprocesador y se utiliza sólo en el perfil de SSR. Se compone de módulos para filtrado, detección de ganancia, y modificación de ganancia. El control de ganancia comienza filtrando la entrada (muestras de audio) en cuatro bandas de frecuencia de 6 kHz de ancho. Este filtrado se efectúa mediante un banco de filtros de cuadratura polifásicos (*polyphase quadrature filters* o PQF). Los coeficientes de frecuencia en cada banda de frecuencia son examinados por los detectores de ganancia, en busca de variaciones rápidas de energía (i.e., coeficientes consecutivos con tamaños muy diferentes). Los modificadores de ganancia modificadores utilizan estos resultados para modificar los coeficientes a fin de comprimir las dinámicas del audio de entrada.

En el decodificador AAC, el control de ganancia realiza las mismas operaciones en orden inverso; se convierte en un postprocesador. La modificación se revierte, para restaurar la dinámica original de la señal de audio, y se utiliza el banco de filtros PQF inverso para generar muestras de audio.

Los coeficientes de las cuatro bandas de frecuencia PQF se calculan mediante:

$$h_i = \frac{1}{4}\cos\left[\frac{(2i+1)(2n+5)\pi}{16}\right]Q(n), \quad \text{para } 0 \le n \le 95, \text{ y } 0 \le i \le 3,$$

donde los primeros 48 valores de $Q(n)$ se muestran en la Tabla 7.74 y los 48 valores restantes vienen dados por $Q(n) = Q(95-n)$ para $48 \le n \le 95$.

La Figura 7.75 muestra los componentes principales del codificador de control de ganancia (parte a) y del decodificador (parte b).

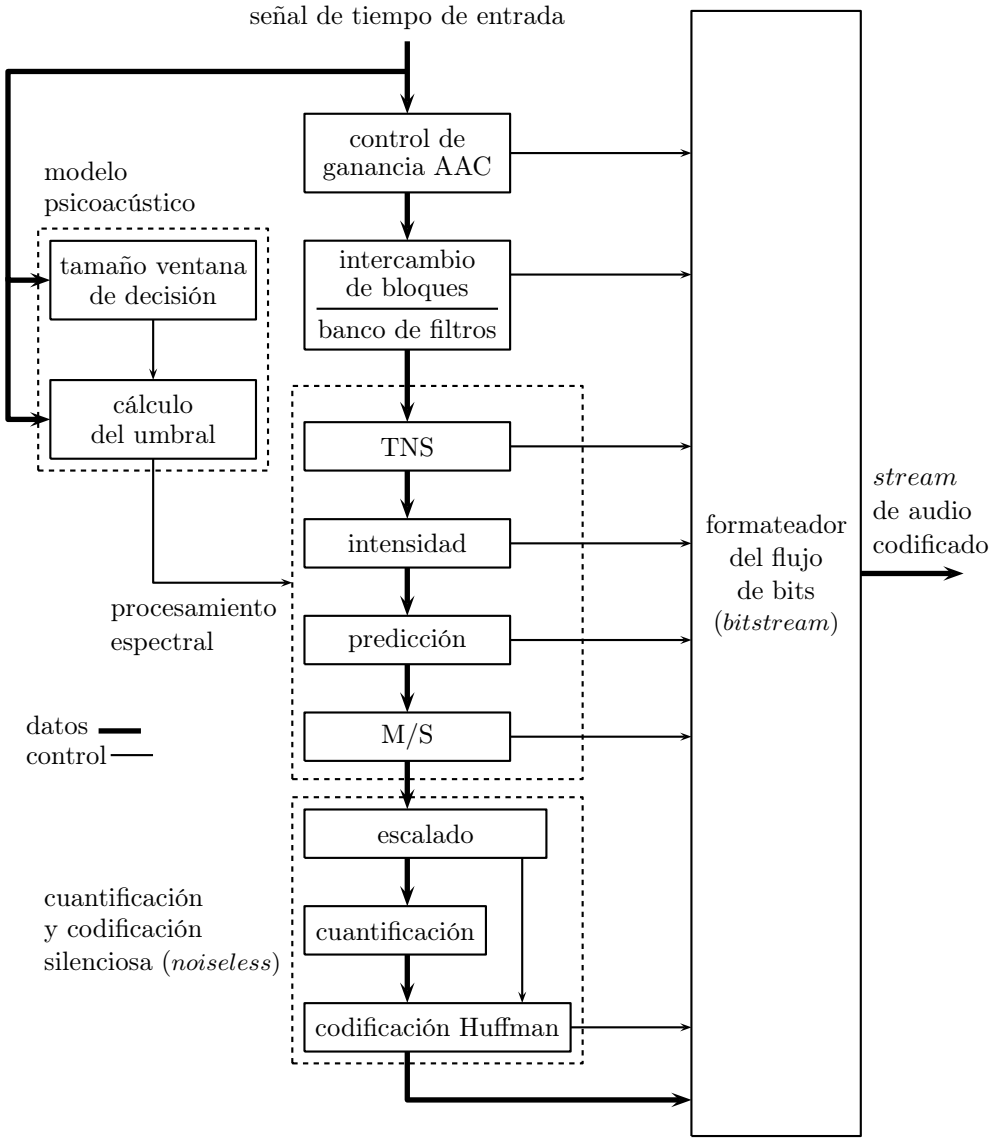


Figura 7.72: Codificador AAC.

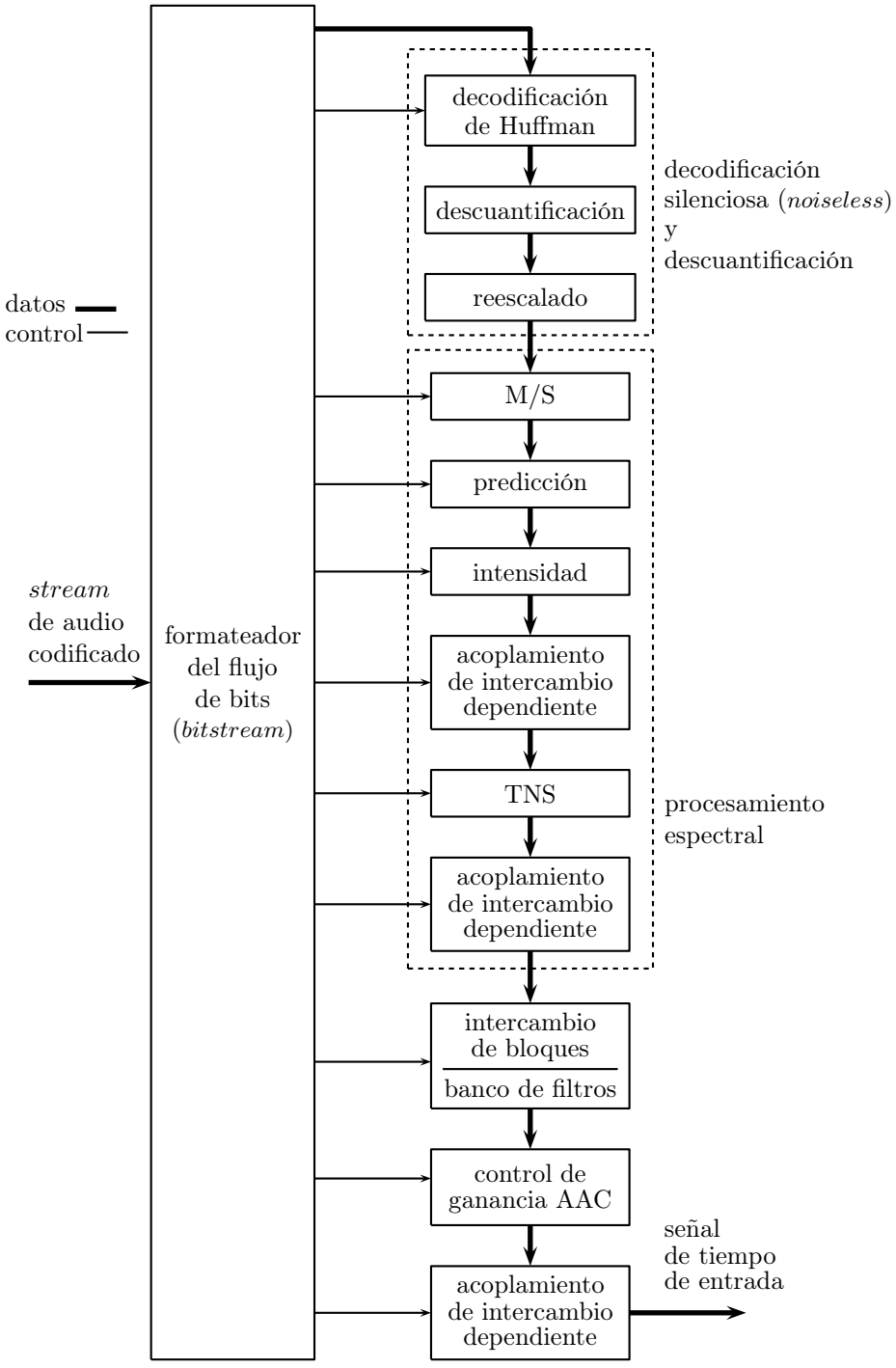


Figura 7.73: Decodificador AAC.

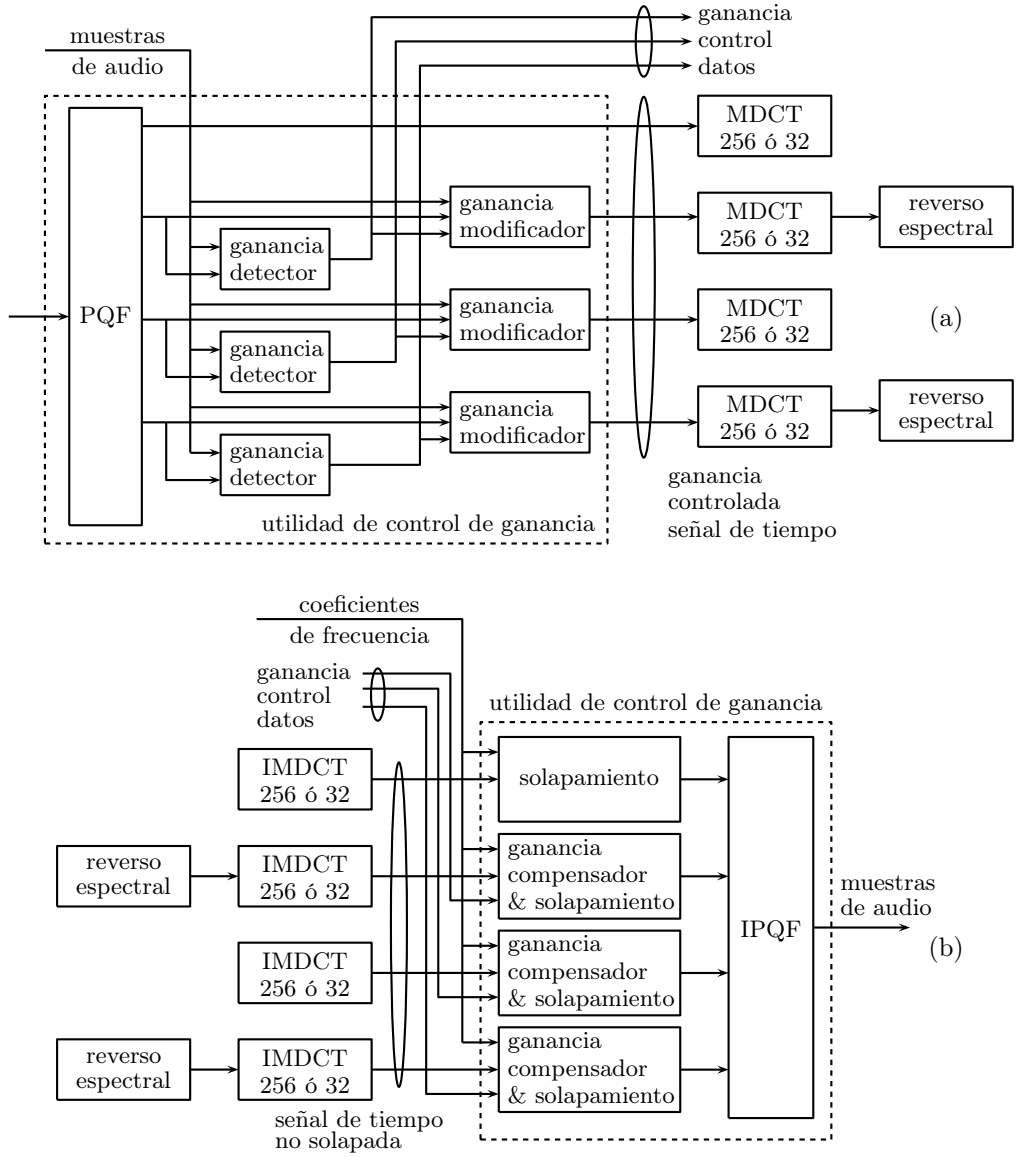


Figura 7.75: Control de ganancia de AAC (a) Codificador y (b) Decodificador.

j	$Q(j)$	j	$Q(j)$	j	$Q(j)$
0	9,7655291007575512 E-05	16	9,8770514991715300 E-03	32	3,3313658300882690 E-02
1	1,3809589379038567 E-04	17	6,1562567291327357 E-03	33	−1,4691563058190206 E-02
2	9,8400749256623534 E-05	18	−4,1793946063629710 E-04	34	−7,2307890475334147 E-02
3	−8,6671544782335723 E-05	19	−9,2128743097707640 E-03	35	−1,2993222541703875 E-01
4	−4,6217998911921346 E-04	20	−1,8830775873369020 E-02	36	−1,7551641029040532 E-01
5	−1,0211814095158174 E-03	21	−2,7226498457701823 E-02	37	−1,9626543957670528 E-01
6	−1,6772149340010668 E-03	22	−3,2022840857588906 E-02	38	−1,8073330670215029 E-01
7	−2,2533338951411081 E-03	23	−3,0996332527754609 E-02	39	−1,2097653136035738 E-01
8	−2,4987888343213967 E-03	24	−2,2656858741499447 E-02	40	−1,4377370758549035 E-02
9	−2,1390815966761882 E-03	25	−6,8031113858963354 E-03	41	1,3522730742860303 E-01
10	−9,5595397454597772 E-03	26	1,5085400948280744 E-02	42	3,1737852699301633 E-01
11	1,1172111530118943 E-03	27	3,9750993388272739 E-02	43	5,1590021798482233 E-01
12	3,9091309127348584 E-03	28	6,2445363629436743 E-02	44	7,1080020379761377 E-01
13	6,9635703420118673 E-03	29	7,7622327748721326 E-02	45	8,8090632488444798 E-01
14	9,5595442159478339 E-03	30	7,9968338496132926 E-02	46	1,0068321641150089 E+00
15	1,0815766540021360 E-02	31	6,5615493068475583 E-02	47	1,0737914947736096 E+00

Tabla 7.74: Los primeros 48 coeficientes $Q(n)$.

Filtrado. El codificador AAC transforma las muestras de audio en coeficientes de frecuencia mediante una transformada DCT modificada (MDCT). El proceso es similar al empleado por las tres capas de MPEG-1 y MPEG-2, pero con una resolución más alta y con mejoras. En cualquier momento dado, el codificador transforma un conjunto de muestras de audio consecutivas referidas a una ventana. Hay dos tipos de ventanas, largas y cortas. Una ventana larga se compone de 2 048 muestras consecutivas y es transformada para producir 1 024 coeficientes de frecuencia (una transformada larga). Una ventana corta tiene 256 muestras de longitud y genera 128 coeficientes (una transformada corta). Una vez que los coeficientes de una ventana han sido calculados, el codificador desplaza las muestras de audio en el buffer en la mitad del tamaño de la ventana. En consecuencia, las ventanas se solapan en un 50 % de su tamaño.

Las ventanas largas producen muchos coeficientes, por lo que cada coeficiente corresponde a una estrecha banda de frecuencias. Ésto conduce a una cuantificación exacta y por lo tanto a una pérdida más significativa de los datos. Los datos perdidos en la cuantificación corresponden a aquellas partes del audio que no son percibidas por el sistema oído/cerebro. En consecuencia, las ventanas largas tienen sentido en aquellas partes del audio que son ya sean tonales, ya sean de baja frecuencia. Por otro lado, el audio atonal o de alta frecuencia varía rápidamente, por lo que las partes del audio de entrada se prestan a sí mismas a una mejor compresión con ventanas cortas.

Al codificador AAC le corresponde analizar la entrada, identificar sus partes estacionarias y transitorias, y utilizar este conocimiento para decidir cuándo y con qué frecuencia debe cambiar el filtrado de las ventanas. Con el fin de suavizar la transición entre las ventanas largas y cortas, AAC define tres tipos de ventanas largas como sigue:

- `LONG_WINDOW`. Éste es el tipo normal de ventanas largas; pueden utilizarse muchas ventanas largas consecutivas, que se superponen en 1 024 muestras de audio y cada una produce 1 024 coeficientes de frecuencia.

- `LONG_START_WINDOW`. Cuando el codificador decide cambiar de una ventana larga a una corta, se utiliza una ventana de este tipo. Ésta tiene un solapamiento de 1 024 muestras de audio con la ventana larga que le precede y de 128 muestras con la ventana corta que le sigue. Produce 1 024 coeficientes de frecuencia.

Frecuencia	Espec. & tablas
$f \geq 92\,017$	96 000
$92\,017 > f \geq 75\,132$	88 200
$75\,132 > f \geq 55\,426$	64 000
$55\,426 > f \geq 46\,009$	48 000
$46\,009 > f \geq 37\,566$	44 100
$37\,566 > f \geq 27\,713$	32 000
$27\,713 > f \geq 23\,004$	24 000
$23\,004 > f \geq 18\,783$	22 050
$18\,783 > f \geq 13\,856$	16 000
$13\,856 > f \geq 11\,502$	12 000
$11\,502 > f \geq 9\,391$	11 025
$9\,391 > f$	8 000

Tabla 7.76: 12 conjuntos de especificaciones de frecuencia.

- `LONG_STOP_WINDOW`. Este tipo de ventana larga se utiliza cuando el codificador decide pasar de un conjunto de ventanas cortas (que siempre vienen en grupos de ocho) a una larga ventana. Este tipo de ventana tiene un solapamiento de 128 muestras de audio con la ventana corta que le precede y una superposición de 1 024 muestras con la ventana larga que le sigue. También produce 1 024 coeficientes de frecuencia.

La anchura de las bandas de frecuencia depende de la tasa de muestreo (número de muestras de audio por segundo y por canal de audio), el número de canales (dos para estéreo), y el número de bandas de frecuencia. La literatura sobre AAC proporciona especificaciones y tablas para las siguientes 12 tasas de muestreo (en Hz): 96 000, 88 200, 64 000, 48 000, 44 100, 32 000, 24 000, 22 050, 16 000, 12 000, 11 025 y 8 000. Cualesquiera otras frecuencias de muestreo deben utilizar uno de los 12 conjuntos de especificaciones y tablas mostradas en la Tabla 7.76. Para una tasa de muestreo de 48 000 Hz y dos canales estéreo, cada canal se muestrea a 24 000 Hz. Una ventana larga genera 1 024 coeficientes de frecuencia, por lo que cada coeficiente corresponde a una banda de frecuencia de $^{24\,000}/_{1\,024} \approx 23{,}4$ kHz.

La Figura 7.77a muestra una secuencia típica de tres ventanas de longitud, que abarca un total de 4 096 muestras de audio. La parte (b) de la figura muestra una `LONG_START_WINDOW`, seguida por ocho ventanas cortas, seguidas por una `LONG_STOP_WINDOW`.

La MDCT calculada por el codificador se especifica mediante [compárese con la Ecuación (7.14)]

$$X_{i,k} = 2\sum_{n=0}^{N-1} z_{i,n} \cos\left[\frac{2\pi}{N}(n+n_0)(k+{}^1/_2)\right], \quad \text{para } 0 \leq k < {}^N/_2,$$

donde $z_{i,n}$ es una muestra de audio, i es el índice del bloque (véase "agrupación y entrelazado" más abajo), n es la muestra de audio en la ventana actual, k es el índice de los coeficientes de frecuencia X, N es el tamaño de la ventana (2 048 ó 256), y $n_0 = ({}^N/_2 + 1)/2$. Cada coeficiente de frecuencia $X_{i,k}$ se calcula mediante un bucle que cubre la ventana actual por completo (ventana i) y se repite en todas las N muestras de audio en esa ventana. Sin embargo, el índice k de los coeficientes de frecuencia varían en el intervalo $[0, {}^N/_2)$, por lo que sólo han sido computados 1 024 ó 128 coeficientes.

La MDCT inversa (IMDCT) calculada por el decodificador viene dada por [compárese con la Ecuación (7.15)]

$$x_{i,n} = \frac{2}{N}\sum_{k=0}^{\frac{N}{2}-1} espec\,[i]\,[k] \cos\left[\frac{2\pi}{N}(n+n_0)(k+{}^1/_2)\right], \quad \text{para } 0 \leq n < N,$$

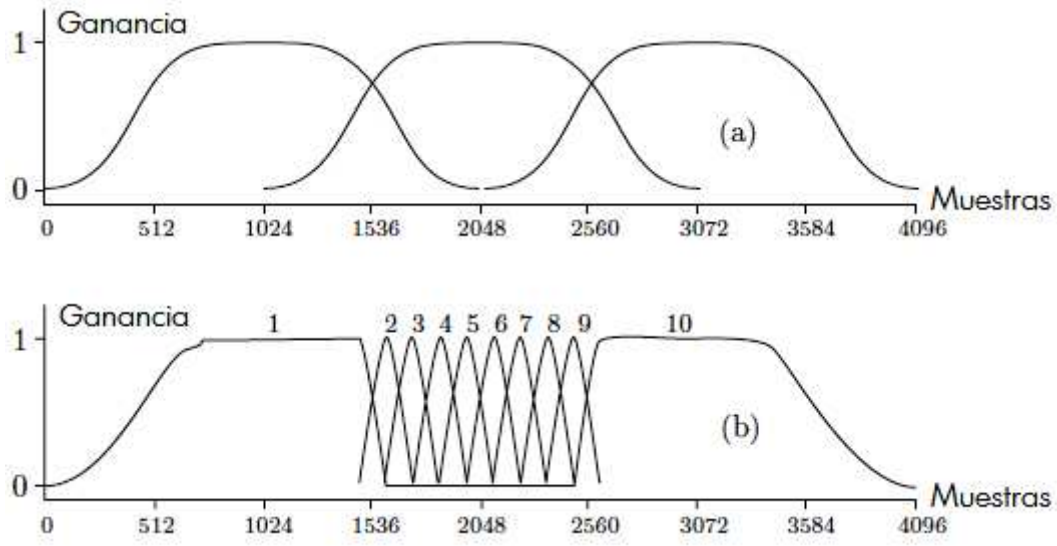


Figura 7.77: Ventanas superpuestas largas y cortas para filtrado AAC.

donde i es el índice del bloque (i.e., ventana), n es el índice de la muestra de audio en la ventana actual, k es el índice de los coeficientes de frecuencia *espec*, N es el tamaño de la ventana (2 048 ó 256), y $n_0 = ({}^{N}/_{2} + 1)/2$. Un vistazo de cerca a este cálculo muestra que se calculan los valores de N de $x_{i,n}$, y cada uno se obtiene como una suma de ${}^{N}/_{2}$ coeficientes de frecuencia.

La **cuantificación** en AAC es, al igual que muchas otras características de AAC, una mejora sobre las tres capas. La regla de cuantificación es [compárese con la Ecuación (7.16)]

$$is\,(i) = \text{sign}\,(x\,(i))\,\text{nint}\left[\left(\frac{|x\,(i)|}{2^{\text{sf}/4}}\right)^{3/4} + 0{,}4054\right], \tag{7.17}$$

donde "sign" es el bit de signo del coeficiente $x\,(i)$, "nint" procede de "*nearest integer*" que significa "entero más cercano", y sf se refiere al factor de escala (que es el tamaño de paso del cuantificador). Además, los valores cuantificados están limitados a un máximo de 8 191. La idea es aplicar una cuantificación gruesa a los coeficientes grandes, porque los coeficientes grandes pueden perder más bits con menos efecto en la reconstrucción del audio. En consecuencia, el código en *Mathematica*:

```
lst = {1., 10., 100., 1000., 10000.};
Table[lst[[i]] - lst[[i]]^0.75, {i, 1, 5}]
```

produce 0, 4,37659, 68,3772, 822,172 y 9,000, lo que ilustra cómo son cuantificados los coeficientes más pequeños. Los cinco elementos de la lista `lst` se cuantifican a 1, 5,62341, 31,6228, 177,828 y 1,000. La no linealidad específica de la potencia-constante ${}^{3}/_{4}$ no es mágica, y se obtuvo como resultado de muchas pruebas e intentos de poner a punto el codificador. Una potencia-constante de 0,5, por ejemplo, habría dado lugar a la más pronunciada cuantificación 0, 6,83772, 90, 968,377, y 9900. Similarmente, la "mágica" constante 0,4054 es el resultado de la experimentación (en las primeras versiones de AAC y mp3, esta constante era −0,0946).

La regla de cuantificación de la Ecuación (7.17) involucra los factores de escala. Cada coeficiente de frecuencia es escalado, durante la cuantificación, en una cantidad relacionada con su factor de escala. Los factores de escala son números enteros sin signo de 8 bits. Cuando los 1 024 coeficientes de una ventana larga son cuantificados, se agrupan en bandas de factor de escala, donde cada banda es un

#	desde	#	desde	#	desde	#	desde
0	0	25	216	1	4	26	240
2	8	27	264	3	12	28	292
4	16	29	320	5	20	30	352
6	24	31	384	7	28	32	416
8	32	33	448	9	36	34	480
10	40	35	512	11	48	36	544
12	56	37	576	13	64	38	608
14	72	39	640	15	80	40	672
16	88	41	704	17	96	42	736
18	108	43	768	19	120	44	800
20	132	45	832	21	144	46	864
22	160	47	896	23	176	48	928
24	196					hasta	1023

Tabla 7.78: Bandas de factor de escala para ventanas largas a 44,1 y 48 kHz.

conjunto de coeficientes espectrales que se escalan en cierto factor de escala. El número de coeficientes en una banda es un múltiplo de 4, con un máximo de 32. Esta restricción hace que sea posible establecer cuatro coeficientes consecutivos en código de Huffman. La Tabla 7.78 muestra las 49 bandas de factor de escala para los tres tipos de ventanas largas y tasas de muestreo de 44,1 y 48 kHz. Por ejemplo, la banda de factor de escala 20, comienza en el coeficiente de frecuencia 132 y finaliza en el coeficiente 143, para un total de 12 coeficientes. La última banda de factor de escala incluye los 96 coeficientes, de 928 a 1 023. El estándar AAC especifica otras bandas de factor de escala para las ventanas cortas y otras tasas de muestreo.

El parámetro AAC `global_gain` es un número entero de 8 bits sin signo con el valor del factor de escala de la primera banda. Este valor se calcula mediante el modelo psicoacústico dependiendo del enmascaramiento de los sonidos que se encuentran en la ventana actual. Los factores de escala de las otras bandas están determinados por los valores calculados en incrementos de 1,5 dB. Los factores de escala se comprimen diferencialmente mediante el cálculo de la diferencia $\text{factorDeEscala}\,(i) - \text{factorDeEscala}\,(i-1)$ y reemplazándolo con un código de Huffman. El hecho de que los factores de escala de bandas consecutivas aumenten en incrementos de 1,5 dB implica que la diferencia entre factores de escala consecutivos no puede superar 120. La Tabla 7.79 muestra algunos códigos de Huffman para comprimir las diferencias de factores de escala. Las longitudes de los códigos varían desde un solo bit a 19 bits.

Codificación silenciosa. Una vez que los coeficientes de frecuencia han sido cuantificados, son sustituidos por códigos de Huffman. Ésto aumenta la compresión, sin embargo es sin pérdidas, de ahí el nombre "silenciosa". Existen 12 libros de códigos de Huffman (Tabla 7.80), aunque se trata de una pseudotabla, para la codificación de rachas (*runs*) de coeficientes cero. Un bloque de 1 024 coeficientes cuantificados se divide en secciones, donde cada sección contiene una o varias bandas de factores de escala. Se utiliza el mismo libro de códigos de Huffman para codificar todos los coeficientes de una sección, pero diferentes secciones pueden utilizar tablas de códigos distintas. La longitud de cada sección (en unidades de bandas de factor de escala) y el índice del libro de códigos de Huffman utilizado para codificar la sección deben ser incluidos en el stream de salida como información adicional para el decodificador.

La idea del seccionamiento es emplear un algoritmo adaptativo con el fin de minimizar el número total de bits de los códigos de Huffman usados para codificar un bloque. En consecuencia, con el fin de determinar las secciones para un bloque, el codificador tiene que ejecutar un algoritmo de mezcla complejo y codicioso, que comience probando el mayor número de secciones (donde cada sección

índice	longitud	código	índice	longitud	código
0	18	3ffe8	61	4	a
1	18	3ffe6	62	4	c
2	18	3ffe7	63	5	1b
3	18	3ffe5	64	6	39
4	19	7fff5	65	6	3b
5	19	7fff1	66	7	78
6	19	7ffed	67	7	7a
7	19	7fff6	68	8	f7
8	19	7ffee	69	8	f9
9	19	7ffef	70	9	1f6
10	19	7fff0	71	9	1f9
⋮			⋮		
57	5	1a	118	19	7ffec
58	4	b	119	19	7fff4
59	3	4	120	19	7fff3
60	1	0			

Tabla 7.79: Códigos Huffman para diferencias de factores de escala.

	Tamaño de tupla	Valor abs. Máx.	Signo
0		0	
1	4	1	sí
2	4	1	sí
3	4	2	no
4	4	2	no
5	2	4	sí
6	2	4	sí
7	2	7	no
8	2	7	no
9	2	12	no
10	2	12	no
11	2	16 (ESC)	no

Tabla 7.80: 12 libros de códigos de Huffman.

es una banda de factor de escala) y que utilice el libro de códigos de Huffman con el índice más pequeño posible. El algoritmo procede tentativamente mezclando secciones. Dos secciones fusionadas permanecen unidas si se reduce el número total de bits. Si las dos secciones que se están siendo tentativamente mezcladas utilizan libros de códigos Huffman diferentes, la sección resultante debe utilizar el libro de códigos con el índice más alto.

El libro de códigos de Huffman 0 es especial. Se trata de un mecanismo de escape que se utiliza para las secciones donde todos los coeficientes cuantificados son cero.

Agrupación y entrelazado. Una ventana larga produce 1 024 coeficientes y una corta produce sólo 128 coeficientes. Sin embargo, las ventanas cortas siempre vienen en grupos de ocho, por lo que también producen 1 024 coeficientes, organizados como una matriz de 8×128. Con el fin de aumentar aún más la eficiencia de la codificación, las ocho ventanas cortas pueden agruparse de tal manera que los coeficientes dentro de un grupo comparten factores de escala (tienen solamente un conjunto de factores de escala). Cada grupo es un conjunto de ventanas adyacentes. Por ejemplo, las tres primeras ventanas pueden convertirse en el grupo 0, la siguiente ventana puede convertirse en el grupo 1 (un grupo de ventana única), las siguientes dos ventanas pueden formar el grupo 2, y las dos últimas ventanas pueden constituir el grupo 3. Cada ventana consta ahora de dos índices, el índice de grupo y el índice de la ventana dentro del grupo. los índices comienzan en cero. En consecuencia, la ventana [0] [2] es la tercera ventana (índice 2) del primer grupo (índice 0). Cada ventana tiene varias bandas de factor de escala, y cada banda es un conjunto de coeficientes. Por tanto, un coeficiente particular, `c` es identificado mediante cuatro índices, grupo (`g`), ventana dentro del grupo (`w`), banda de factor de escala dentro de la ventana (`b`), y coeficiente dentro de una banda (`k`). Por consiguiente, `c` [`g`] [`w`] [`b`] [`k`] es un arreglo (*array*) de cuatro dimensiones donde `k` es el índice más rápido.

El entrelazado es el proceso de intercambio del orden de las bandas de factor de escala y las ventanas; un coeficiente `c` [`g`] [`w`] [`b`] [`k`]se convierte en `c` [`g`] [`b`] [`w`] [`k`]. El resultado del entrelazado es que los coeficientes de frecuencia que pertenecen a la misma banda de factor de escala, pero a distintos tipos de bloques (i.e., los coeficientes que deben ser cuantificados con los mismos factores de escala), se ponen juntos en una banda de factor de escala (más grande). Ésto tiene la ventaja de combinar todas las secciones de ceros debido a la limitación de la banda dentro de cada grupo.

Una vez que todas las secciones y los grupos han sido determinados y el entrelazado está completo, los coeficientes se codifican reemplazando cada coeficiente con un código de Huffman tomado desde uno de los libros de códigos. La Tabla 7.80 lista todos los 12 libros de códigos de Huffman. Para cada libro, la tabla muestra su índice (0–11), el tamaño máximo de tupla (pueden codificarse hasta dos o cuatro coeficientes de frecuencia mediante un código de Huffman), el valor absoluto máximo de los coeficientes que pueden ser codificados mediante el libro de códigos, y la información: con signo/sin signo. Un libro de códigos con signo codifica sólo los coeficientes positivos. Un libro de códigos sin signo proporciona códigos para los valores sin signo de los coeficientes de frecuencia. Por lo tanto, si el coeficiente 5 va a ser codificado mediante tal libro de códigos con el código de Huffman de 10 bits `3f0`, entonces su signo (0) es escrito en el stream comprimido después de este código. Similarmente, el coeficiente de -5 es codificado con el mismo código de Huffman de 10 bits y posteriormente es seguido por un bit de signo 1. Si dos o cuatro coeficientes se codifican mediante un solo código de Huffman, entonces sus bits de signo siguen a este código en el stream comprimido. Existen dos libros de códigos para cada valor absoluto máximo, cada uno para una probabilidad de distribución diferente. Siempre se elige el mejor ajuste.

El primer paso en la codificación de la siguiente tupla de coeficientes de frecuencia es seleccionar un libro de códigos de Huffman (el índice se escribe en el stream comprimido para el uso del decodificador). Conociendo el índice del libro de códigos (1–11), la Tabla 7.80 especifica el tamaño de tupla (dos o cuatro coeficientes consecutivos a ser codificados). Si los coeficientes no exceden el tamaño máximo permitido por el libro de códigos (1, 2, 4, 7, 12, ó 16, dependiendo del libro de códigos), el codificador utiliza los valores de los coeficientes en la tupla para calcular un índice para el libro de códigos. El

```
if (unsigned) {
    mod = lav + 1;
    off = 0;
}
else {
    mod = 2*lav + 1;
    off = lav;
}
if (dim == 4) {
    w = INT(idx/(mod*mod*mod)) - off;
    idx -= (w+off)*(mod*mod*mod)
    x = INT(idx/(mod*mod)) - off;
    idx -= (x+off)*(mod*mod)
    y = INT(idx/mod) - off;
    idx -= (y+off)*mod
    z = idx - off;
}
else {
    y = INT(idx/mod) - off;
    idx -= (y+off)*mod
    z = idx - off;
}
```

Figura 7.81: Decodificación de coeficientes de frecuencia.

libro de códigos es posteriormente accedido para proporcionar el código de Huffman para la n-tupla de coeficientes. El código de Huffman es escrito en el stream de salida, y si el libro de códigos es sin signo (índices del libro de códigos 3, 4, ó 7–11), el código de Huffman es seguido por dos o cuatro bits de signo de los coeficientes. La decodificación de un código de Huffman se lleva a cabo revirtiendo los pasos, excepto que el índice del libro de códigos de Huffman es leído por el decodificador desde el stream comprimido. Los pasos de la decodificación se muestran, como código en C, en la Figura 7.81. La figura utiliza las siguientes convenciones:

- `unsigned` es el valor booleano en la columna 4 de la Tabla 7.80.
- `dim` es el tamaño tupla del libro de códigos, listado en la segunda columna de la Tabla 7.80.
- `lav` es el valor absoluto máximo en la columna 3 de la Tabla 7.80.
- `idx` es el índice de la palabra de código.

La figura muestra cómo se utilizan `unsigned`, `dim`, `lav`, e `idx` para calcular ya sean los cuatro coeficientes `w`, `x`, `y`, y `z`, ya sean los dos coeficientes `y` y `z`. Si se usó un libro de códigos sin signo, los dos o cuatro bits de signo se leen del stream comprimido y se adjuntan a los coeficientes recién calculados.

Si los coeficientes exceden el tamaño máximo permitido por el libro de códigos, cada uno es codificado con una secuencia de escape. Un código de Huffman que representa una secuencia de escape se construye siguiendo esta regla: Se comienza con N 1's seguidos por un solo 0. Se sigue con un valor binario (llamado `escapeword`) de $N+4$ bits, y se interpreta la secuencia completa de $N+1+(N+4)$ bits como el número $2^{N+4}+$ `escapeword`. En consecuencia, la secuencia de escape 01111 corresponde a $N=0$ y un `escapeword` de 4 bits, de 15. Por consiguiente, su valor es $2^{0+4}+15=31$. La secuencia de escape 1011111 corresponde a $N=1$ y un `escapeword` de 5 bits, de 31. Su valor, por lo tanto, es

índice	longitud	palabra de código	índice	longitud	palabra de código
0	11	7f8	41	5	14
1	9	1f1	42	7	65
2	11	7fd	43	5	16
3	10	3f5	44	7	6d
4	7	68	45	9	1e9
5	10	3f0	46	7	63
6	11	7f7	47	9	1e4
7	9	1ec	48	7	6b
8	11	7f5	49	5	13
9	10	3f1	50	7	71
10	7	72	51	9	1e3
⋮	⋮	⋮	⋮	⋮	⋮
38	7	62	79	9	1f7
39	5	12	80	11	7f4
40	1	0			

Tabla 7.82: Libro 1 de códigos de Huffman (Parcial).

$2^{1+4} + 31 = 63$. Las secuencias de escape están limitadas a 21 bits, por lo que la secuencia de escape más grande comienza con ocho 1's, seguidos por un solo 0, seguido por $8 + 4 = 12$ 1's. Corresponde a $N = 8$ y un `escapeword` de $8+4$ bits, de $2^{12}-1$. Su valor es, por consiguiente, $2^{8+4}+\left(2^{12}-1\right) = 8\,191$.

La Tabla 7.82 muestra parte del primer libro de códigos Huffman. El campo "longitud" es la longitud del código correspondiente, en bits.

Modelado de ruido temporal (*Temporal Noise Shaping* o TNS). Este módulo de AAC aborda el problema de los preecos (léase: "pre-ecos"), un problema que se crea cuando la señal de audio a ser comprimida es transitoria y varía rápidamente. El problema es la falta de correspondencia entre el enmascaramiento umbral del oído (como se predijo mediante el modelo psicoacústico) y el ruido de cuantificación (la diferencia entre un coeficiente de frecuencia y su valor cuantificado). El codificador AAC (así como las tres capas de MPEG-1 y MPEG-2) genera un ruido de cuantificación que se distribuye uniformemente en cada ventana de filtros, mientras que el umbral de enmascaramiento varía significativamente, incluso durante el corto período de tiempo que corresponde a una ventana.

Una ventana de filtros larga consta de 2 048 muestras de audio, por lo que representa un tiempo muy corto. Una frecuencia de muestreo típica es de 44 100 $^{\text{muestras}}/_{\text{segundo}}$, por lo que 2 048 muestras corresponden a 0,46 segundos, un tiempo muy corto. Si el sonido original (señal de audio) no varía mucho durante este período de tiempo, el codificador AAC determinará el umbral de enmascaramiento correcto y llevará a cabo la cuantificación correcta. Sin embargo, si la señal de audio varía considerablemente durante este periodo, la cuantificación es incorrecta, debido a que ruido de cuantificación se distribuye uniformemente en cada ventana de filtros. El módulo de TNS alivia el problema de preecos mediante la remodelación y el control de la estructura del ruido de cuantificación en el tiempo (dentro de cada ventana transformada). Ésto se efectúa aplicando un proceso de filtrado a las partes de los datos de frecuencia de cada canal.

Quizás la mejor manera de entender el principio de funcionamiento de TNS es considerar la siguiente dualidad. La mayoría de las señales de audio cuentan con muestras de audio que están correlacionadas, similarmente a los píxeles adyacentes en una imagen. Por lo tanto, este tipo de audio puede ser comprimido, ya sea mediante la predicción de muestras de audio, ya sea transformando las muestras de audio en coeficientes de frecuencia y cuantificando estos últimos. En los sonidos transitorios, las muestras de audio no están correlacionadas, lo que sugiere la siguiente aproximación opuesta (o dual)

Muestras	bandas sf	# de predictores	Frecuencia máx.
96 000	33	512	24 000,00
88 200	33	512	22 050,00
64 000	38	664	20 750,00
48 000	40	672	15 750,00
44 100	40	672	14 470,31
32 000	40	672	10 500,00
24 000	41	652	7 640,63
22 050	41	652	7 019,82
16 000	37	664	5 187,50
12 000	37	664	3 890,63
11 025	37	664	3 574,51
8 000	34	664	2 593,75

Tabla 7.83: Límite superior de frecuencia para la predicción.

a la compresión de este tipo de sonidos: O bien transformar las muestras de audio en coeficientes de frecuencia, y predecir los siguientes n coeficientes a partir de sus predecesores inmediatos, o bien codificar las muestras de audio directamente (i.e., mediante códigos de tamaño variable o algún codificador de entropía).

Predicción. Esta herramienta mejora la compresión prediciendo un coeficiente de frecuencia a partir de los coeficientes correspondientes en las dos ventanas precedentes, y cuantificando y codificando la diferencia de predicción. La predicción tiene sentido para los ítems que están correlacionados. En AAC, se utilizan las ventanas cortas cuando el codificador decide que el sonido que está siendo comprimido es no estacionario. Por consiguiente, la predicción se usa sólo en los 1 024 coeficientes de frecuencia de las ventanas largas.

La predicción AAC es de segundo orden, similar a como se muestra en la Figura 7.27b, pero es adaptativa. El algoritmo de predicción es por lo tanto complejo y requiere muchos cálculos. El codificador AAC es a menudo complejo y sus requisitos de velocidad y memoria normalmente no son cruciales. El decodificador, sin embargo, debe ser rápido. Es por ésto que AAC tiene características que limitan la complejidad de la predicción. Las características principales son: (1) La predicción se utiliza sólo en el perfil principal (Tabla 7.71). (2) En cada ventana larga, la predicción se efectúa en aquellos coeficientes de frecuencia que corresponden a las bajas frecuencias, pero se detiene a una cierta frecuencia máxima que depende de la tasa de muestreo (Tabla 7.83). (3) Además, el algoritmo de predicción depende de las variables que se almacenan internamente como números en punto flotante de 16 bits (llamados f.p. truncados, en el estándar de punto flotante IEEE) en lugar de números en punto flotante de 32 bits. Ésto conlleva un ahorro sustancial en el espacio de memoria.

La tabla indica, por ejemplo, que a una frecuencia de muestreo de 48 kHz, la predicción puede ser utilizada en las primeras 40 bandas de factor de escala (bandas de 0 a 39) y se detiene en el primer coeficiente de frecuencia que corresponde a una frecuencia mayor que o igual a 15,75 kHz. [La columna 3 de la Tabla 7.83, (el número de predictores), no se discute aquí.]

A veces, la predicción no funciona; el error de predicción es mayor que el coeficiente que está siendo predicho. Por lo tanto, la predicción se efectúa dos veces para cada banda de factor de escala en la ventana actual. En primer lugar, se predicen los coeficientes en la banda y los errores de la predicción son cuantificados y codificados. A continuación, los coeficientes mismos son cuantificados y codificados sin predicción. El codificador selecciona el método que produce menos bits y escribe los bits en el stream de salida, precedido por un flag que indica si se utilizó o no la predicción para esa banda de factor de escala.

La regla de cuantificación de la Ecuación (7.17) genera resultados intermedios que son números

reales, luego convierte el resultado final al entero más cercano (nint). Palabras de distintas longitudes y circuitos ALU en ordenadores diferentes pueden, por tanto causar resultados numéricos que difieren en algo, y la posibilidad de que ésto ocurra aumenta cuando se utiliza la predicción. Por ello, el mecanismo de predicción AAC utiliza reglas para restablecer la predicción. Los predictores, tanto del codificador como del decodificador (que pueden ejecutarse en equipos diferentes) son periódicamente restablecidos a estados iniciales bien definidos, lo que ayuda a evitar las diferencias entre ellos.

7.15.2. Extensiones MPEG-4 para AAC

Las principales características y el funcionamiento de la AAC se han resuelto como parte de MPEG-2. El proyecto MPEG-4 añade a AAC varios algoritmos que contribuyen a la compresión eficiente y ofrecen nuevas características extendidas. Las mejoras principales son: (1) sustitución de ruido perceptual (*perceptual noise substitution* o PNS), (2) predicción a largo plazo (*long-term prediction* o LTP), (3) cuantificación vectorial entrelazada de ponderación en el dominio de la transformada (*transform-domain weighted interleave vector quantization* o TwinVQ) que codifica el núcleo, (4) retraso largo AAC (*long-delay AAC* o AAC-LD), (5) módulo de resistencia a errores (*error-resilience* o ER), (6) codificación de audio escalable, y (7) modo de escalabilidad de grano fino (*fine-grain scalability* o FGS). Algunas de estas características se discuten aquí.

Substitución de ruido perceptual. Un micrófono convierte las ondas sonoras del aire en una señal eléctrica que varía con el tiempo; una forma de onda. AAC es normalmente un método de compresión con pérdida. Cuantifica los coeficientes de frecuencia para lograr la mayor parte de su eficiencia de compresión (la compresión restante se consigue mediante la codificación Huffman de los coeficientes, y es sin pérdidas). Si se omite la etapa de cuantificación, se convierte en AAC sin pérdidas (a pesar de que pierde la mayor parte de su capacidad de compresión) y el decodificador AAC reconstruye la forma de onda de sonido original (excepto pequeños cambios debidos a la naturaleza limitada de la aritmética de la máquina). Si la etapa de cuantificación no se omite, el decodificador genera una forma de onda que se asemeja a la original.

La substitución de ruido perceptual (PNS) es un nuevo enfoque de la compresión de audio donde parte del sonido original se convierte por el codificador PNS a un número, consiguiendo así la compresión. Este número, sin embargo, no es suficiente para que el decodificador reconstruya la forma de onda de audio original. En su lugar, el decodificador genera audio que es percibido por el oído como el sonido original, aunque corresponde a una forma de onda diferente. PNS es especialmente adecuado para el audio *noiselike* (similar a ruido), donde se consiguen altos factores de compresión y un sonido reconstruido de intermedia a buena calidad.

PNS se basa en el hecho de que lo que el oído percibe en presencia del audio noiselike no depende tanto de la forma de onda precisa del audio como de la forma en que su frecuencia varía con el tiempo (su buena estructura espectral). PNS se incluye opcionalmente en el codificador AAC para analizar los coeficientes de frecuencia antes de que se cuantifiquen, con el fin de determinar las partes noiselike de la entrada. Una vez que una banda de factor de escala de coeficientes de frecuencia ha sido identificada como noiselike, no es cuantificada y codificada con Huffman, sino que en su lugar se envía al codificador PNS para ser procesada. El codificador PNS simplemente escribe un flag en el stream de salida para indicar que esta banda de factor de escala está procesada mediante PNS. El flag es seguido por la potencia total de ruido del conjunto de coeficientes (la suma de los cuadrados de los coeficientes, adecuadamente codificados con Huffman). Cuando el decodificador AAC lee el stream comprimido e identifica el flag, invoca al decodificador PNS. Este decodificador introduce la potencia total P y prueba varios conjuntos de números pseudoaleatorios hasta que encuentra un conjunto cuya potencia total es igual a P. Los números en este conjunto son posteriormente enviados al decodificador AAC, como si fueran coeficientes de frecuencia descuantificados.

Es fácil ver por qué PNS logra una excelente compresión para las partes noiselike del audio de entrada. Lo que es más difícil de entender es cómo se decodifica un conjunto de números aleatorios

para generar un sonido noiselike que el oído no pueda distinguir del sonido original.

La **predicción a largo plazo** es una técnica que descubre y explota una redundancia especial en las muestras de audio originales: la redundancia que está relacionada con una (normalmente invisible) periodicidad en algunas partes del audio. El codificador AAC funciona como de costumbre; prepara las ventanas de los coeficientes de frecuencia C, y cuantifica y codifica cada ventana. Sin embargo, antes de que los códigos de Huffman sean emitidos en el stream de salida, el codificador AAC invoca al módulo LTP, que realiza los siguientes pasos:

- Actúa temporalmente como un decodificador AAC. Decodifica la ventana y reconstruye las muestras de audio.
- Compara estas muestras con las muestras originales.
- La comparación produce parámetros de retardo y de ganancia que se utilizan posteriormente para predecir la ventana actual a partir de sus predecesores.
- Las muestras predichas se filtran para convertirse en coeficientes de frecuencia.
- Estos coeficientes se restan de los coeficientes del conjunto C para convertirse en un conjunto de residuos.
- Cualquier residuo de los coeficientes de conjunto C (lo que resulta en el menor número de bits) es codificado mediante Huffman y se envía al stream de salida.

Cuantificación vectorial Twin (VQ). Este algoritmo de cuantificación fue desarrollado como parte de MPEG-4 para su uso en el codificador de audio escalable. (MPEG-4 ofrece una selección de varios codificadores de audio como AAC, escalable, HILN, ALS, CELP y HVXC. Los dos últimos son codificadores de voz y ALS es sin pérdidas.) Debido al éxito de TwinVQ, es a veces usado en AAC como una alternativa a la cuantificación AAC original.

TwinVQ es un proceso de dos pasos. El primer paso (normalización espectral) reescala los coeficientes de frecuencia a un rango de amplitudes deseado y calcula varios parámetros de la señal que son utilizados posteriormente por el decodificador. La segunda etapa entrelaza los coeficientes normalizados, los organiza en los llamados subvectores, y aplica una cuantificación vectorial ponderada a los subvectores. (La cuantificación vectorial se discute en la Sección 4.14.)

7.15.3. AAC-LD (Low Delay —Baja latencia)

A la gente le gusta comunicarse. Este hecho se hace evidente si tenemos en cuenta el éxito de los muchos inventos que permiten y mejoran las comunicaciones. Todas las tecnologías, como el telégrafo, el teléfono, la radio, y el correo electrónico (email) y la telefonía por internet han tenido éxito. Un aspecto importante y obvio de la comunicación es una conversación. A nadie le gusta hablar con las paredes. Cuando la gente habla espera una respuesta rápida. Incluso un pequeño retraso en una respuesta es molesto, y un largo retraso puede matar una conversación. La alta velocidad de las ondas electromagnéticas y de los electrones en los alambres evita retrasos en las llamadas locales y en las conversaciones de radio bilaterales. Sin embargo, las llamadas telefónicas de larga distancia que se enrutan a través de uno o más satélites de comunicaciones pueden provocar un retraso considerable (cerca a un segundo) entre el hablante y el oyente.

Cuando las futuras naves espaciales tripuladas vayan más allá de la órbita de la luna (cerca de 1,2–1,5 segundos luz de distancia), el retraso causado por la velocidad finita de la luz podría ser molesto. A la distancia de Marte (unos cuatro minutos de luz) una conversación normal entre la tripulación y la Tierra sería imposible.

> Hubo un coro de "adioses", y la pantalla de visualización se quedó en blanco. Qué extraño pensar, se dijo Poole, que todo esto había sucedido hace más de una hora; ahora, su familia se habría dispersado de nuevo y sus miembros estarían a millas de casa. Pero de alguna manera este lapso de tiempo, aunque podría ser frustrante, también era una bendición disfrazada. Como todo hombre de su edad, Poole daba por sentado que podía hablar al instante, a cualquier persona en la Tierra, cuando quisiera. Ahora que ésto ya no era cierto, el impacto psicológico fue profundo. Se había mudado a una nueva dimensión de la lejanía, y casi todos los vínculos emocionales se habían extendido más allá del punto de fluencia.
>
> —Arthur C. Clarke, *2001: Una odisea del espacio*, (1968)

Diversos estudios sobre los efectos de los retrasos conversacionales en las personas han llegado a la conclusión de que retrasos de hasta 0,1 segundos son imperceptibles, retrasos de hasta 0,25 segundos son aceptables, y retrasos más largos son normalmente molestos. En general se acepta que una demora de 0,15 segundos es el máximo para una "buena" interactividad.

Hay, sin embargo, el problema de los ecos y es especialmente pronunciada en la telefonía por internet y los chats por ordenador. En un chat por ordenador, una persona A habla en un micrófono conectado a su ordenador, el programa de chat digitaliza el sonido y lo envía al equipo receptor, en el que se convierte en una tensión analógica y alimenta a un altavoz. Éste provoca un retraso (si el chat incluye también vídeo, los datos tienen que ser comprimidos antes de transmitirlos y son descomprimidos en el destino, causando un retardo más largo). En el receptor, la persona B escucha el mensaje, pero su micrófono también lo oye e inmediatamente lo transmite de vuelta a A, que lo escucha como un débil eco. El uso de auriculares en lugar de altavoces elimina este problema, pero en las llamadas de conferencia, es natural de utilizar altavoces.

La presencia de ecos reduce la calidad de las conversaciones a distancia y es, por supuesto, indeseable. Los ingenieros de audio e investigadores de telecomunicaciones han experimentado con el efecto combinado de los retardos y los ecos en las respuestas de voluntarios de ensayo, y han llegado a conclusiones y recomendaciones (véase [G131 06]).

Es por esta razón que MPEG-4 ha añadido un módulo de baja latencia a AAC. Este módulo puede manejar tanto el habla como la música con alta compresión, reconstrucción de alta calidad, y retardos cortos. Una ventaja de AAC-LD es la escalabilidad. Cuando el usuario permite altas tasas de bits (es decir, baja compresión), la calidad del audio reconstruido por este módulo se hace mayor y puede convertirse fácilmente en indistinguible de aquél sin pérdidas.

El retardo total causado por AAC es una suma de varios retrasos provocados por altas tasas de muestreo (muchas muestras de audio por segundo), filtrado (los bancos de filtros están diseñados para alta resolución), el cambio de ventana, y la manipulación del repositorio de bits. El cambio de ventana ocasiona retrasos, ya que requiere un proceso de consulta con anticipación para determinar las propiedades de los datos futuros (muestras de audio que no han sido aún examinadas) Ésta es una razón por la que es tan difícil de implementar un codificador AAC de buena calidad y por la que muchos AAC codificadores "baratos" producen una baja compresión.

Cada una de estas fuentes de retardo ha sido examinada y modificada en AAC-LD. Las principales características de esta variante son las siguientes:

- Se limita la velocidad de muestreo a sólo 48 kHz y utiliza tamaños de frame de 480 ó 512 muestras. Ésto reduce el tamaño de la ventana a la mitad de su longitud normal.
- Las ventanas no se cambian. Los artefactos de preeco se reducen mediante TNS.
- La "forma de la ventana" del banco de filtros de frecuencia es ahora adaptativa. En AAC, el forma es una curva sinusoidal ancha, pero AAC-LD cambia dinámicamente una ventana a una forma que tiene un solapamiento entre las bandas por su parte inferior (Figura 7.84).

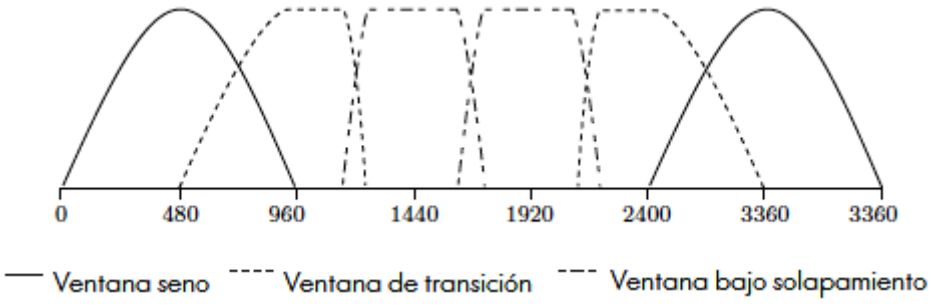


Figura 7.84: Formas de ventana en AAC-LD.

- El repositorio de bits, o bien se ha eliminado por completo, o bien su uso está limitado.

Estas características reducen de manera significativa los retardos a muy por debajo de 0,1 segundos, mientras que la disminución en la compresión relativa a AAC es sólo moderada. La tasa de bits se incrementa en alrededor de 8 kbits por canal estéreo; no es muy significativa.

Se llevó a cabo una batería de pruebas para comparar el rendimiento de mp3 (un solo canal) a AAC-LD. Los resultados indican que AAC-LD supera a mp3 en la mitad de las pruebas, mientras que sigue siendo tan bueno como mp3 la otra mitad.

7.15.4. Tests AAC

El testeo de los métodos de compresión de audio es especialmente importante y complejo. Un algoritmo para la compresión de texto es fácil de testear, pero los métodos (especialmente los con pérdidas o *lossily*) para comprimir imágenes o audio son difíciles de probar debido a que estos tipos de datos (especialmente de audio) están sujetos a opinión. Un probador humano tiene que ser expuesto a una imagen original o a una secuencia de audio (antes de comprimirla) y a los mismos datos después de su compresión y descompresión. Este tipo doble test es ciego (al probador no se le dice qué datos son originales y cuáles son reconstruidos) y tiene que ser repetido varias veces (ya que incluso una persona que decide al azar logrará un éxito del 50 % de media en la identificación de los datos) y con varios probadores voluntarios, tanto con experiencia como sin experiencia (porque todos perciben imágenes, palabras y música de diferentes maneras). Se llevaron a cabo extensos y exitosos tests para AAC en la BBC en Inglaterra, la CBC y el centro de investigaciones en comunicaciones (CRC o Comunications Research Centre) en Canadá, y la NHK de Japón.

Una vez que el grupo MPEG-4 decidió los algoritmos que componen AAC, fueron implementados y probados. El objetivo de todo el proyecto era crear un método de compresión de audio capaz de comprimir audio a una tasa de bits de 64 kbps y generar sonido reconstruido indistinguible del original. No todos los sonidos son iguales. Ciertos sonidos se prestan a una compresión eficiente, mientras que otros son casi aleatorios y desafían cualquier intento de ser comprimidos. La misma secuencia de sonido puede tener segmentos fáciles y difíciles, por lo que la esencia de las pruebas era identificar qué sonidos representan un reto para el codificador AAC y utilizarlos en las pruebas.

Una definición rigurosa del término "indistinguibles" se proporciona por la ITU-R en la publicación TG-10 [ITU/TG10 91].

Los tests probaron que AAC se comporta al menos tan bien como otros compresores de audio para tasas de bits bajas (en el rango de 16 a 64 kbps) y es definitivamente superior a otros compresores con tasas de bits superiores a 64 kbps.

Se realizaron dos series de tests en 1996–97. El primero en 1996 en la BBC en Inglaterra y en la corporación de radiodifusión de Japón [NHK 06]. El segundo tuvo lugar en 1997 en el centro de

#	Nombre	Descripción
1	Cast	Castañuelas distribuidas por el frente, ruido en sonido envolvente
2	Clarinet	Clarinete al frente, ambiente de antecámara teatro, lluvia en surround
3	Eliot	Voz femenina y masculina en un restaurante, música de cámara
4	Glock	Carillón (*Glockenspiel*) y timbales
5	Harp	Clavicémbalo (*Harpsichord*)
6	Manc	Orquesta, cuerdas, címbalos, tambores, cuernos
7	Pipe	Diapasón
8	Station	Voz masculina con efectos de locomotora a vapor
9	Thal	Piano al frente izquierda, saxo al frente derecha, voz femenina en el centro
10	Tria	Triángulo

Tabla 7.85: Tests AAC en 1996 (después de [Bosi y Goldberg 03]).

investigación de comunicaciones [CRC 98] en Canadá.

La primera serie de tests utilizó un grupo de 23 oyentes expertos fiables en la BBC y 16 oyentes expertos fiables en la NHK. Los test compararon 10 muestras de audio, cada una comprimida en AAC y en MPEG-2 de capa 2. Los tests fueron del tipo conocido como triple estímulo/referencia oculta/doble ciego, que se especifica en [ITU-R/BS1116 97]. Se escogió inicialmente un conjunto de 94 muestras de audio y posteriormente se redujo a 10 pasajes críticos juzgados . Se enumeran en la Tabla 7.85. Los resultados (que se analizan en profundidad en la página 360 de [Bosi y Goldberg 03]) indican la superioridad de AAC en comparación con la capa 2.

El segundo conjunto de pruebas ha sido bien documentado y parece haber sido muy extenso. Las pruebas empezaron con una selección de material de audio apropiada. Un panel de tres expertos se encargaron de esta tarea. Durante varios meses se recogieron y se escucharon 80 sonidos obtenidos a partir de: (1) los test previos, (2) los archivos CRC, y (3) las colecciones privadas de los propios expertos. Además, todos aquellos que contribuyeron a los test de las implementaciones de AAC suministraron fragmentos de audio que a su juicio eran difíciles de comprimir.

Cada uno de los 80 ítems fue posteriormente comprimido a 96 kbps y descomprimido por cada uno de los 17 codificadores AAC utilizados en los tests, resultando en 1 360 secuencias de audio. El panel de expertos escuchó todas las 1 360 secuencias y redujo su número a 340 ($= 20 \times 17$) que a su juicio resistirían la compresión y presentarían a los codificadores AAC considerables desafíos.

La etapa de selección final reduce el número de elementos de audio a prueba de 20 a sólo ocho, y estos ocho elementos fueron los utilizados en los test. Incluyen sonidos complejos como arpegios, música de un contrabajo arqueado, una trompeta con sordina, una canción de Susan Vega, y una mezcla de música y lluvia.

El primer mp3 registrado fue la canción *El restaurante de Tom* (*Tom's Diner*) por Susan Vega [suzannevega 06], escrita por ella (sobre el restaurante de Tom en la ciudad de Nueva York) en 1983. Como resultado, a ella se la conoce a veces como la madre del mp3.

En 1993, Karlheinz Brandenburg dirige un equipo de programadores/ingenieros que implementaron un mp3 para Fraunhofer-Gesellschaft y patentaron su software. Como resultado de ello, Brandenburg es a veces llamado el padre del mp3. La historia de mp3 está bien documentada. Dos de las muchas fuentes disponibles son: [h2g2 06] y [MPThree 06].

Brandenburg utilizó la canción *Tom's Diner* como su primer ítem de test de audio para mp3, porque la claridad vocal inusual de esta pieza le permitió percibir con precisión cualquier degradación del audio tras haberlo comprimido y descomprimido.

"Estaba listo para poner a punto mi algoritmo de compresión", recuerda Brandenburg.

> "En algún lugar en el pasillo, en una radio sonaba *Tom's Diner*. Yo estaba electrificada. Sabía que iba a ser casi imposible de comprimir esta cálida voz a capella."

Debido a que la canción depende de matices muy sutiles de inflexión de Vega, el algoritmo tendría que ser muy, muy bueno para identificar las partes más importantes del audio y desechar el resto. Así Brandenburg testeó cada refinamiento de su implementación con esta canción. Acabó escuchando la canción miles de veces, y el resultado fue una rápida, eficiente y robusta implementación de mp3.

Una vez que los objetos de prueba (los ítems de audio) estaban listos, se eligieron los sujetos de prueba (las personas haciendo juicios). Se seleccionaron un total de 24 oyentes, incluyendo siete músicos (intérpretes, compositores, estudiantes), seis ingenieros de audio, tres profesionales de audio, tres sintonizadores de piano, dos programadores, y tres personas elegidas al azar.

Los jueces concluyeron de que la calidad de las piezas de prueba de audio en AAC (a 96 kbps) era comparable a la producida por la capa 2 a 192 kbps, y por la capa 3 a 128 kbps. Basándose en estas pruebas, el CRC anunció que AAC había pasado el objetivo establecido por la ITU de "calidad indistinguible" del sonido comprimido.

Como resultado de estos tests, el audio comprimido por AAC a 128 kbps es actualmente considerado la mejor opción para la compresión de audio con pérdidas y es la elección de muchos usuarios, incluyendo los exigentes amantes de la música clásica y los críticos.

7.16. Dolby AC-3

Dolby Laboratories desarrolla y entrega productos y tecnologías que hacen del entretenimiento una experiencia más realista e inmersiva. Durante cuatro décadas, Dolby ha estado a la vanguardia de la definición de la alta calidad del audio y del sonido envolvente en el cine, la radiodifusión (*broadcast*), los sistemas de audio para el hogar, los automóviles, los DVDs, los auriculares, los juegos, los televisores y las computadoras personales. Tiene la sede central en San Francisco, con sede europea en Inglaterra; la empresa cuenta con oficinas de enlace de la industria del entretenimiento en Nueva York y Los Angeles, y oficinas de enlace de licencias en Londres, Shanghai, Beijing, Hong Kong y Tokio. (Cita de [Dolby 06].)

Dolby Laboratories fue fundada por Ray Dolby, que comenzó su carrera en la escuela secundaria, cuando se fue a trabajar a tiempo parcial para Ampex Corporation en Redwood City, California. Aunque todavía en la universidad, se unió al pequeño equipo de ingenieros de Ampex dedicado a inventar la primera grabadora de vídeo práctica del mundo, que se introdujo en 1956; su objetivo era la electrónica.

Después de graduarse en la Universidad de Stanford en 1957, a Dolby se le adjudicó una beca universitaria (*Marshall Fellowship*) para la Universidad de Cambridge en Inglaterra. Después de seis años en Cambridge destacó con un Ph.D. (*Philosophiae Doctor*) en física, Dolby trabajó en la India por dos años como asesor de las Naciones Unidas de la Organización de Instrumentos Científicos Central. Regresó a Inglaterra en 1965 para fundar su propia empresa, Dolby Laboratories, Inc. en Londres. Siempre como corporación de Estados Unidos, la compañía trasladó su sede a San Francisco en 1976. (Cita de [Dolby 06].)

AC-3, también conocido como Dolby Digital, procede de "codificador de audio de tercera generación de Dolby". AC-3 es un codificador de audio perceptivo basado en los mismos principios que las tres capas MPEG-1/2 y AAC. Esta breve sección se concentra en las características especiales de AC-3 y lo que lo distingue de otros codificadores perceptivos. La información detallada puede encontrarse en

[Bosi y Goldberg 03]. AC-3 fue aprobado por el comité de sistemas de televisión avanzada (*Advanced Television Systems Committee* o ATSC) de Estados Unidos en 1994. La especificación formal puede encontrarse en [ATSC 06 y 10] y también pueden estar disponibles a partir de este autor.

AC-3 fue desarrollado para combinar tasas de bits bajas con una excelente calidad del sonido reconstruido. Se usa en varias aplicaciones de vídeo importantes tales como la siguientes:

- HDTV (en Norteamérica). La televisión de alta definición (*high-definition television* o HDTV) es un estándar para televisión de alta resolución. Actualmente, la HDTV se está haciendo muy popular y sus productos están reemplazando poco a poco los formatos de televisión más antiguos existentes, como NTSC y PAL.
- DVD-video. Éste es un estándar desarrollado por el *DVD Forum* para el almacenamiento de películas digitales de larga duración en DVDs. El estándar permite el uso de tanto MPEG-1 como MPEG-2 para la compresión de vídeo, pero este último es más común. Los reproductores comerciales de DVD-Video son actualmente muy populares. Estos dispositivos se conectan a un televisor como un reproductor de cintas de vídeo. El formato de vídeo digital incluye un sistema de codificación de contenidos (*Content Scrambling System* o CSS) para impedir la copia no autorizada de DVDs.
- DVB. El proyecto de vídeo digital de radiodifusión (*Digital Video Broadcasting* o DVB) es un consorcio liderado por la industria de más de 270 organismos de radiodifusión, fabricantes, operadores de redes, desarrolladores de software, cuerpos reguladores, y otros en más de 35 países que se comprometieron a diseñar estándares globales para la distribución global de la televisión digital y los servicios de datos. Los servicios que utilizan los estándares DVB están disponibles en todos los continentes con más de 110 millones de receptores DVB desplegados. (Cita de [DVB 06].)
- Varias transmisiones por cable digital y por satélite.

Lo que distingue a AC-3 de otros codificadores perceptuales es que fue originalmente diseñado para soportar varios canales de audio. Un equipo de audio antiguo utiliza un solo canal de audio (llamado mono o monofónico). El año 1957 marcó el debut del disco fonográfico estéreo de larga duración (*long-play* o LP), que popularizó el sonido estéreo (más preciso, estereofónico). El estéreo requiere dos canales de audio, izquierdo y derecho.

El formato cuadrafónico (*quad*) apareció en la década de 1970. Consta de una matriz de codificación de cuatro canales de datos de audio inmersa en un registro de dos canales. Permite una grabación de dos canales para albergar cuatro canales de audio que se reproducen en cuatro altavoces. Quad nunca llegó a ser muy popular debido al costo de los nuevos amplificadores, receptores y altavoces adicionales.

A mediados de 1970, Dolby Labs dio a conocer un nuevo proceso de sonido envolvente que era fácil adaptar para el uso doméstico. El desarrollo del HiFi estéreo VCR y la radiodifusión de televisión estéreo en la década de 1980 proporcionó una oportunidad adicional para que el sonido envolvente se hiciera popular. Una razón importante de la popularidad del sonido envolvente es la capacidad de agregar procesadores Dolby surround a los receptores estéreo existentes; no hay necesidad de comprar el equipo completo nuevo.

El método Dolby Surround implica la codificación de cuatro canales de audio —frontal izquierdo, central, frontal derecho, y trasero envolvente— en una señal de dos canales. Un circuito decodificador separa entonces los cuatro canales y los envía a los altavoces adecuados: izquierdo (*left*), derecho (*right*), trasero y central fantasma (el canal central se deriva de los canales frontales L/R). El resultado es un entorno de escucha equilibrada donde los principales sonidos provienen de los canales izquierdo y derecho, el audio vocal o los diálogos emanan del canal central fantasma, y el ambiente o la información de efectos proviene de detrás del oyente.

código	tasa de bits	código	tasa de bits
2	32	20	160
4	40	22	192
6	48	24	224
8	56	26	256
10	64	28	320
12	80	30	384
14	96	32	448
16	112	34	512
18	128	36	640

Tabla 7.86: Tasas de bits (kbps) en AC-3.

cód.	config.	#	canal	altavoces	cód.	config.	#	canal	altavoces
000	$1+1$	2	C1, C2		100	2/1	3	L, R, S	
001	1/0	1	C		101	3/1	4	L, C, R, S	
010	2/0	2	L, R		110	2/2	4	L, R, SL, SR	
110	3/0	3	L, C, R		111	3/2	5	L, C, R, SL, SR	

Tabla 7.87: Configuraciones de canal en AC-3.

En consecuencia, el número de canales de audio utilizados en los equipos de sonido ha aumentado sostenidamente a lo largo de los años, y sólo fue natural para Dolby llegar a un método de compresión de audio que soporta hasta cinco canales. El rango de tasas de bits es de 32 a 640 kbps, dependiendo del número de canales (Tabla 7.86, en donde "código" es el código de la tasa de bits enviada al decodificador). Más exactamente, AC-3 soporta de uno a 5.1 canales de audio. La inusual designación 5.1 significa cinco canales más un canal especial de efectos de baja frecuencia (*low-frequency-effects* o LFE) (referido como canal 0.1) que proporciona una manera económica y práctica de lograr un impacto de graves extra. Las ocho configuraciones de canal compatibles con AC-3 se muestran en la Tabla 7.87, donde "cód." es un código enviado al decodificador y "config." es la notación utilizada por Dolby para indicar la configuración del canal.

Además del audio comprimido, el stream de salida de AC-3 puede incluir datos secundarios como identificadores de idioma, avisos de copyright, marcas de tiempo, y otra información de control. AC-3 también es compatible con las llamadas funciones de escucha que incluyen: mezcla de la reproducción (para reducir el número de canales, para aquellos oyentes que carecen de equipos avanzados de escucha), normalización del diálogo, compatibilidad con Dolby surround, streams de bits para personas con discapacidad auditiva, y control de rango dinámico. La última característica es especialmente útil. la persona que codifica el audio con AC-3 puede especificar los parámetros de control del rango para segmentos individuales de audio. El decodificador AC-3 utiliza estos parámetros para ajustar el nivel del audio decodificado en un bloque mediante bloques base (donde un bloque se compone de unos pocos ms de audio, dependiendo de la frecuencia de muestreo), garantizando de este modo que el audio que se está reproduciendo siempre tendrá el rango dinámico completo.

El primer paso del codificador AC-3 es agrupar las muestras de audio (que pueden ser de hasta 24 bits cada una) en bloques de 512 muestras cada una. Cada bloque es evaluado para determinar si el

audio es tonal o transitorio. Si el audio es transitorio, el tamaño de bloque se acorta a 256 muestras de audio. Cada bloque es mapeado sobre el dominio de la frecuencia mediante una DCT modificada. El número de coeficientes de frecuencia calculados para el bloque es la mitad del tamaño del bloque. Seis bloques conforman una ventana y el tamaño de los bloques de la ventana se puede cambiar entre largo y corto, similar a la forma en que se hace en AAC.

Una vez que los coeficientes de frecuencia han sido calculados, el codificador los utiliza en un proceso denominado rematrización (*rematrixing*) para llevar a cabo la codificación multicanal. Los coeficientes de frecuencia resultantes de la MDCT son números reales y se almacenan en formato de punto flotante, con una mantisa y un exponente. El primero incluye los dígitos significativos del coeficiente, mientras que el último indica su tamaño. Así, los dos números de punto flotante $0{,}12345678 \times 2^{-10}$ y $0{,}12345678 \times 2^{+10}$ tienen mantisas idénticas que constituyen ocho dígitos significativos, pero tienen tamaños marcadamente diferentes indicados por sus distintos exponentes.

La compresión se consigue mediante la codificación de los exponentes y la cuantificación de las mantisas (los cantidad de cuantificación depende de la rutina de distribución de bits). La primera compresión es sin pérdidas, mientras que la segunda es con pérdidas. La rutina de distribución de bits aplica un modelo psicoacústico para determinar la precisión adecuada necesaria para las mantisas.

El último paso del codificador es combinar (multiplexar) los exponentes codificados y las mantisas cuantificadas con información de control, tal como flags de conmutación de bloque, parámetros de acoplamiento, flags de rematrización, bits de rango del exponente, flags de tramado o difuminado, y parámetros de asignación de bits para crear el stream de salida final.

Thomas Dolby (Thomas Morgan Robertson, nació el 14 de octubre de 1958, en Londres) es un músico británico. Su padre era un profesor de historia griega y en su juventud vivió en varios países de la Europa mediterránea como Grecia, Italia y Francia. El apodo de "Dolby" viene del nombre Dolby Laboratories, y se lo pusieron sus amigos impresionados por sus retoques de estudio. Según se dice, Dolby Laboratories estaban muy disgustados con que Robertson usara el nombre de la empresa como su propio nombre artístico y le demandó, tratando de impedirle utilizar el nombre de Dolby completo. Eventualmente, lograron restringirlo usando la palabra Dolby en cualquier otro contexto que no tuviera el nombre de Thomas. (Véase [Thomas Dolby 06]).

Me suena bien.

—Jason James Richter (como Jesse), en *Liberad a Willy* (1993)

Capítulo 8

Otros métodos

En los capítulos anteriores se discuteron los principales tipos de métodos de compresión: RLE, métodos estadísticos, y los métodos basados en diccionarios. Hay métodos de compresión de datos que no son tan fáciles de clasificar y no pertenecen claramente a cualquiera de los tipos discutidos hasta ahora. Aquí se describen algunos de tales métodos.

- El método de Burrows-Wheeler (Sección 8.1) comienza con una cadena S de n símbolos y la codifica (i.e., permuta) en otra cadena L que satisface dos condiciones: (1) Cualquier área de L tiende a tener una concentración de sólo unos pocos símbolos. (2) Es posible reconstruir la cadena original S a partir de L.
- La técnica *symbol ranking* (o de *clasificación de símbolos*) (Sección 8.2) utiliza el contexto para clasificar símbolos más bien que asignarlos probabilidades.
- ACB es un método nuevo, basado en un diccionario asociativo (Sección 8.3). Tiene características que lo relacionan con los métodos tradicionales basados en diccionarios, así como al método symbol ranking.
- La Sección 8.4 es una descripción de un método de similitud de contextos basados en el orden. Este método utiliza el contexto de un símbolo en una forma que recuerda a ACB. También asigna rangos a los símbolos, y esta característica hace referencia al método de Burrows–Wheeler y también a la clasificación de símbolos.
- El caso especial de las *cadenas binarias dispersas* se discute en la Sección 8.5. tales cadenas se pueden comprimir de manera muy eficiente debido al gran número de ceros consecutivos que contienen.
- Los métodos de compresión que se basan en palabras en lugar de en símbolos individuales son el objetivo de la Sección 8.6.
- La compresión de una imagen textual es el tema de la Sección 8.7. Cuando un documento impreso tiene que ser guardado en el ordenador, tiene que ser escaneado en primer lugar; un proceso que lo convierte en una imagen que típicamente contiene millones de píxeles. El complejo método aquí descrito se ha desarrollado para este tipo de datos, que forma un tipo especial de imagen: una *imagen textual*. Tal imagen está hecha de píxeles, pero la mayoría de los píxeles se agrupan para formar caracteres, y el número de grupos diferentes no es grande.
- El método FHM (por Fibonacci, Huffman, y Markov) es un método inusual, de propósito especial, para la compresión de curvas.

- La codificación dinámica de Markov utiliza máquinas de estado finito para estimar la probabilidad de símbolos y la codificación aritmética codificarlos realmente. Éste es un método de compresión de alfabetos de dos símbolos (binario).

- Sequitur, Sección 8.10, es un método especialmente adecuado para la compresión de texto semiestructurado. Se basa en gramáticas libres de contexto.

- La Sección 8.11 es una descripción detallada de *edgebreaker*, un método muy original para comprimir la información de conectividad de una malla triangular. Este método y sus diversas extensiones puede convertirse en el estándar para la compresión de superficies poligonales, uno de los tipos de superficies más comunes utilizadas en los gráficos por ordenador. Edgebreaker es un ejemplo de método de *compresión geométrica*.

- Las Secciones 8.12 y 8.12.1 describen dos algoritmos —SCSU y BOCU-1— para la compresión de los documentos basados en Unicode.

- La Sección 8.13 es un breve resumen de los métodos de compresión utilizados por el popular Formato de Documento Portátil (PDF) de Adobe.

- La Sección 8.14 (escrita por Giovanni Motta), cubre una variante poco conocida de la compresión de datos, a saber, cómo comprimir las diferencias entre dos archivos. Todos tenemos teléfonos móviles, y damos estos pequeños dispositivos útiles por sentado. Sin embargo, un teléfono móvil es un dispositivo complejo que está dirigido por un ordenador y tiene un sistema operativo. Naturalmente, el software en el interior del teléfono tiene que ser actualizado de vez en cuando, pero el envío de un archivo ejecutable grande para muchos miles de teléfonos es un importante quebradero de cabeza de la comunicación. Es mejor aislar las diferencias entre el viejo y el nuevo software, comprimir estas diferencias, y enviar el archivo comprimido a todos los teléfonos.

- La Sección 8.15 sobre la compresión de datos hiperespectrales (escrito conjuntamente con Giovanni Motta) trata el tema de los datos hiperespectrales. Tales datos son similares a una imagen digital, pero cada "píxel" se compone de muchos componentes (de cientos a miles).

8.1. El método de Burrows-Wheeler

La mayoría de los métodos de compresión operan en *modo continuo* (*streaming*) en el que el codificador introduce uno o varios bytes, los procesa, y continúa hasta detectar el indicador fin-de-archivo. El método de Burrows-Wheeler (BW), descrito en esta sección [Burrows y Wheeler 94], trabaja en *modo bloque*, donde el *stream*[1] de entrada se lee bloque por bloque, y cada bloque es codificado por separado como una cadena. El método se conoce, por lo tanto, como *clasificación de bloques*. El método de BW es de propósito general; funciona bien en imágenes, sonido y texto, y puede lograr tasas de compresión muy altas (1 bit por byte o incluso mejor).

La idea principal del método BW es comenzar con una cadena S de n símbolos y desorganizarla, produciendo otra cadena L que satisface dos condiciones:

[1] Una cadena o secuencia de datos continuos.

1. Cualquier región de L tenderá a tener una concentración de sólo unos pocos símbolos. Otra forma de decir ésto es: si un símbolo s se encuentra en una posición determinada en L, entonces es probable encontrar en las inmediaciones otras apariciones de s. Esta propiedad significa que L puede ser comprimida fácil y eficientemente con el método mover–al–frente (Sección 1.5), tal vez en combinación con RLE. Ésto también significa que el método BW funcionará bien sólo si n es grande (de por lo menos varios miles de símbolos por cadena).

2. Es posible reconstruir la cadena original `S` a partir de `L` (puede ser necesario un conjunto de datos pequeño para la reconstrucción, además de `L`, pero no muchos).

El término matemático para revolver los símbolos es *permutación*, y es fácil demostrar que una cadena de símbolos n tiene $n!$ (se pronuncia "n factorial") permutaciones. Es un número muy grande, incluso para valores relativamente pequeños de n, por lo que la permutación particular utilizada por BW tiene que ser cuidadosamente seleccionada. El codificador BW procede con los siguientes pasos:

1. La cadena L es creada —por el codificador— como una permutación de S. También genera una información adicional —denotada por I—, que será utizada más tarde por el decodificador en el paso 3.

2. El codificador comprime L e I, y escribe los resultados en la cadena de salida. Este paso comienza típicamente con RLE, continúa con la codificación mover–al–frente, y finalmente aplica la codificación de Huffman.

3. El decodificador lee la cadena de salida y la decodifica aplicando los mismos métodos que en el anterior paso 2, pero en orden inverso. El resultado es la cadena L y la variable I.

4. Ambas —L e I—, son utilizadas por el decodificador para reconstruir la cadena original S.

> Odio hacer sumas. No hay mayor error que llamar a la aritmética una ciencia exacta. Hay permutaciones y aberraciones perceptibles a la mente completamente nobles como la mía; sutiles variaciones que los contables ordinarios no descubren; leyes ocultas de los números que requieren una mente como la mía, para percibirlas. Por ejemplo, si usted realiza una suma de abajo arriba, y luego de arriba abajo, el resultado es siempre diferente.
>
> —Mrs. La Touche, *La Matemática Gaceta*, v. 12 (1924)

El primer paso es comprender cómo se crea la cadena L a partir de S, y qué información se necesita se almacenar en I, para la posterior reconstrucción. Usamos la ya familiar cadena `swiss␣miss` para ilustrar este proceso.

Dada una cadena de entrada de n símbolos, el codificador construye una matriz de $n \times n$, donde almacena la cadena S en la fila superior, seguida por $n-1$ copias de S, cada una desplazada cíclicamente (rotada) un símbolo a la izquierda (Figura 8.1a). La matriz se ordena lexicográficamente por filas (véase la Sección 3.4 sobre el orden lexicográfico), produciendo la matriz ordenada de la Figura 8.1b). Obsérvese que cada fila y cada columna de cada una de las dos matrices es una permutación de S y, por lo tanto, contiene todos los n símbolos de S. La permutación de L seleccionada por el codificador es la **última columna** de la matriz ordenada. En nuestro ejemplo, ésta es la cadena `swm␣siisss`. La única información necesaria para reconstruir eventualmente S a partir de L es el número de la fila de la cadena original en la matriz ordenada, que en nuestro ejemplo es 8 (las filas y las columnas se numeran comenzando desde 0). Éste número se almacena en I.

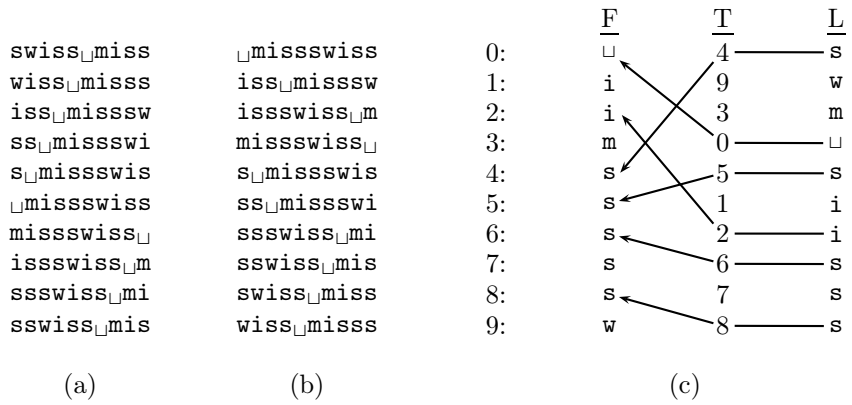


Figura 8.1: Principios sobre la compresión de Burrows–Wheeler.

Es fácil ver por qué `L` contiene concentraciones de símbolos idénticos. Supongamos que las palabras: `bail, fail, hail, jail, mail, nail, pail, rail, sail, tail` y `wail` aparecen en algún lugar en `S`. Después de la ordenación, todas las permutaciones que comienzan con `il` aparecerán juntas. Todas ellas contribuyen con una `a` en L, de modo que L tendrá una concentración de `a`es. Además, todas las permutaciones que comienzan con `ail` va a aparecer juntas, contribuyendo a una concentración de las letras `bfhjmnprstw` en una región de L.

Ahora podemos caracterizar el método de BW diciendo que utiliza la ordenación para agrupar juntos los símbolos en función de sus contextos. Sin embargo, el método considera el contexto en un sólo lado de cada símbolo.

◇ **Ejercicio 8.1 (sol. en pág. 1107):** La última columna —L— de la matriz ordenada contiene concentraciones de caracteres idénticos, por lo que L es fácil de comprimir. Sin embargo, la primera columna —F— de la misma matriz es aún más fácil de comprimir, ya que contiene rachas, no sólo concentraciones, de caracteres idénticos. ¿Por qué selecciona la columna L y no la F?

Nótese también que el codificador en realidad no tiene que construir los dos matrices de $n \times n$ (o incluso una de ellas) en la memoria. Los detalles prácticos del codificador se discuten en la Sección 8.1.2, así como la compresión de L e I; pero vamos a ver primero cómo funciona el decodificador.

El decodificador lee un flujo comprimido, lo descomprime utilizando Huffman y mover–al–frente (y quizás también RLE), y luego reconstruye cadena `S` a partir de la `L` descomprimida en tres pasos:

1. La primera columna de la matriz ordenada (columna F de la Figura 8.1c) se construye a partir de L. Éste es un proceso sencillo, ya que F y L contienen los mismos símbolos (ambas son permutaciones de S) y F está ordenada. El decodificador simplemente ordena la cadena L para obtener F.

2. Mientras ordena L, el decodificador prepara una matriz auxiliar T que muestra las relaciones entre los elementos de L y F (Figura 8.1c). El primer elemento de T es 4, lo que implica que el primer símbolo de L (la letra "`s`") se encuentra en la posición 4 de F. El segundo elemento de T es de 9, lo que implica que el segundo símbolo de L (la letra "`w`") se encuentra en la posición 9 de F, y así sucesivamente. El contenido de T en nuestro ejemplo es $(4, 9, 3, 0, 5, 1, 2, 6, 7, 8)$.

3. La cadena F ya no es necesaria. El decodificador utiliza L, I, y T para reconstruir S de acuerdo con:

$$\begin{gathered} \mathrm{S}\left[n-1-i\right] \leftarrow \mathrm{L}\left[\mathrm{T}^{i}\left[\mathrm{I}\right]\right], \text{ para } i = 0, 1, \ldots, n-1, \\ \text{donde } \mathrm{T}^{0}\left[j\right] = j, \text{ y } \mathrm{T}^{i+1}\left[j\right] = \mathrm{T}\left[\mathrm{T}^{i}\left[j\right]\right]. \end{gathered} \tag{8.1}$$

Estos son los dos primeros pasos en esta reconstrucción:

$$\mathtt{S}\,[10-1-0]=\mathtt{L}\left[\mathtt{T}^0\,[\mathtt{I}]\right]=\mathtt{L}\left[\mathtt{T}^0\,[8]\right]=\mathtt{L}\,[8]=\mathtt{s},$$
$$\mathtt{S}\,[10-1-0]=\mathtt{L}\left[\mathtt{T}^1\,[\mathtt{I}]\right]=\mathtt{L}\left[\mathtt{T}\left[\mathtt{T}^0\,[\mathtt{I}]\right]\right]=\mathtt{L}\,[\mathtt{T}\,[8]]=\mathtt{L}\,[7]=\mathtt{s}.$$

◇ **Ejercicio 8.2 (sol. en pág. 1107):** Complétese esta reconstrucción.

Antes de llegar a los detalles de la compresión, puede ser interesante comprender por qué la Ecuación (8.1) reconstruye S a partir de L. Los siguientes argumentos explican por qué funciona este proceso:

1. T está construida de tal manera que $\mathrm{F}\,[\mathrm{T}\,[i]]=\mathrm{L}\,[i]$ para $i=0,\ldots,n$.
2. Un vistazo a la matriz ordenada de la Figura 8.1b, muestra que en cada fila i, el símbolo $\mathrm{L}\,[i]$ precede al símbolo $\mathrm{F}\,[i]$ en la cadena original S (la palabra *precede* tiene que ser entendida como *precede cíclicamente*). Específicamente, en la fila I (8 en nuestro ejemplo), L [I] precede cíclicamente a F [I], pero F [I] es el primer símbolo de S, por lo que L [I] es el *último* símbolo de S. La reconstrucción comienza con L [I] y se reconstruye S de derecha a izquierda.
3. $\mathrm{L}\,[i]$ precede a $\mathrm{F}\,[i]$ en S para $i=0,\ldots,n-1$. Por lo tanto, $\mathrm{L}\,[\mathrm{T}\,[i]]$ precede a $\mathrm{F}\,[\mathrm{T}\,[i]]$, pero $\mathrm{F}\,[\mathrm{T}\,[i]]=\mathrm{L}\,[i]$. La conclusión es que $\mathrm{L}\,[\mathrm{T}\,[i]]$ precede a $\mathrm{L}\,[i]$ en S.
4. Por consiguiente, la reconstrucción comienza con $\mathrm{L}\,[\mathrm{I}]=\mathrm{L}\,[8]=\mathtt{s}$ (el último símbolo de S) y procede con $\mathrm{L}\,[\mathrm{T}\,[\mathrm{I}]]=\mathrm{L}\,[\mathrm{T}\,[8]]=\mathrm{L}\,[7]=\mathtt{s}$ (el situado junto al último símbolo de S). Ésta es la razón por la cual, la Ecuación (8.1) describe correctamente la reconstrucción.

8.1.1. La compresión de L

La compresión de L se basa en su atributo principal, es a saber, contiene concentraciones (aunque no necesariamente *runs*) de símbolos idénticos. La utilización de RLE tiene sentido, pero sólo como un primer paso en un proceso de compresión de múltiples etapas. El paso principal en la compresión de L debe utilizar el método mover–al–frente (Sección 1.5). Este método se aplica a nuestro ejemplo $\mathrm{L}=\mathtt{sw}\sqcup\mathtt{msiisss}$ como sigue:

1. Se inicializa A con una lista que contiene nuestro alfabeto $\mathrm{A}=(\sqcup,\mathtt{i},\mathtt{m},\mathtt{s},\mathtt{w})$.
2. Para $i=0,\ldots,n-1$, se codifica el símbolo L_i como el número de símbolos que le preceden en A, y luego se mueve el símbolo L_i hasta el comienzo de A.
3. Se combinan los códigos del paso 2 en una lista C, que se comprime aún más utilizando Huffman o la codificación aritmética.

Los resultados se resumen en la Figura 8.2a. La lista final de los códigos es el elemento de 10 del array $\mathrm{C}=(3,4,4,3,3,4,0,1,0,0)$, que ilustra cómo cualquier concentración de símbolos idénticos produce códigos pequeños. A la primera aparición de i se le asigna el código 4, pero a la segunda ocurrencia se le asigna el código 0. Las dos primeras apariciones de s obtienen el código 3, pero la siguiente obtiene el código 1.

Es interesante comparar los códigos en C, que son enteros en el rango $[0,n-1]$, con los códigos obtenidos sin el paso adicional de "mover al frente". Es fácil codificar L utilizando los tres pasos anteriores, pero sin mover símbolo L_i hasta el comienzo de A. El resultado es C'=$(3,4,2,0,3,1,1,3,3,3)$, una lista de números enteros *en el mismo rango* $[0,n-1]$. Ésta es la razón por la cual la aplicación de mover–al–frente no es suficiente. Las listas C y C′ contienen elementos del mismo intervalo; pero en promedio, los elementos de C son más pequeños. Por consiguiente, además deben ser codificados

L	A	Código	C	A	L
s	␣imsw	3	3	␣imsw	s
w	s␣imw	4	4	s␣imw	w
m	ws␣im	4	4	ws␣im	m
␣	mws␣i	3	3	mws␣i	␣
s	␣mwsi	3	3	␣mwsi	s
i	s␣mwi	4	4	s␣mwi	i
i	is␣mw	0	0	is␣mw	i
s	is␣mw	1	1	is␣mw	s
s	si␣mw	0	0	si␣mw	s
s	si␣mw	0	0	si␣mw	s
	(a)			(b)	

Figura 8.2: Codificación/Decodificación de L mediante mover–al–frente.

utilizando la codificación de Huffman o algún otro método estadístico. Los códigos de Huffman para C pueden asignarse suponiendo que el código 0 tiene la probabilidad más alta, y el código $n-1$, la probabilidad más pequeña.

En nuestro ejemplo, un posible conjunto de códigos de Huffman es: 0—0, 1—10, 2—110, 3—1110, 4—1111. La aplicación de este conjunto a C produce los 29 bits: 1110|1111|1111|1110|1110|1111|0|10|0|. (Su aplicación a C′ produce los 32 bits: 1110|1111|110|0|1110|10|10|1110|1110|1110.) Nuestra cadena original de 10 caracteres `swiss␣miss` se ha codificado utilizando 2,9 bits por carácter, un resultado muy bueno. Cabe señalar que el método de Burrows-Wheeler puede lograr fácilmente una mejor compresión que eso cuando se aplica a cadenas más largas (miles de símbolos).

◇ **Ejercicio 8.3 (sol. en pág. 1108):** Dada la cadena S = `sssssssssh` calcúlese la cadena L y su compresión usando mover–al–frente.

La decodificación de C se realiza con el proceso inverso de mover–al–frente. Asumimos que la relación del alfabeto A está disponible para el decodificador (o bien es la lista de todos los bytes posibles, o bien ha sido escrito por el codificador en la cadena de salida). La Figura 8.2b muestra los detalles de la decodificación de C = $(3,4,4,3,3,4,0,1,0,0)$. El primer código es 3, por lo que el primer símbolo de la nueva construcción de L es el *cuarto* elemento en A, o "`s`". Este símbolo se mueve entonces al principio de A, y el proceso continúa.

8.1.2. Consejos de implementación

Puesto que el método de Burrows–Wheeler es eficiente sólo para cadenas largas (de al menos miles de símbolos), cualquier aplicación práctica debe permitir valores grandes de n. El valor máximo de n debe ser tan grande como para que dos matrices de $n \times n$ no puedan ser alojadas en la memoria disponible (al menos no confortablemente), y todas las operaciones del codificador (la preparación de las permutaciones y su ordenación) debe hacerse con arrays unidimensionales de tamaño n. En principio, es suficiente tener en memoria sólo, la cadena original S y la matriz auxiliar T. [Manber y Myers 93] y [McCreight 76] discuten las estructuras de datos usadas en esta implementación.

La cadena S contiene los datos originales, pero sorprendentemente, también contiene todas las permutaciones necesarias. Ya que las únicas permutaciones que necesitamos generar son rotaciones, podemos generar la permutación i de la matriz 8.1a mediante el escaneo de S —desde la posición i hasta el final—, y a continuación, seguir cíclicamente desde el inicio de S hasta la posición $i-1$. La permutación 5, por ejemplo, puede generarse escaneando la subcadena $(5,9)$ de S (`␣miss`), seguida por la subcadena $(0,4)$ de S (`swiss`). El resultado es `␣missswiss`. El primer paso en una

implementación práctica, por lo tanto, sería escribir un procedimiento que acepte un parámetro i y escanee la permutación correspondiente.

Cualquier método utilizado para ordenar las permutaciones tiene que compararlas. La comparación de dos permutaciones puede hacerse mediante la búsqueda de las mismas en S, sin tener que mover símbolos o crear nuevas matrices.

Una vez que el algoritmo de ordenación determina que la permutación i debería estar en la posición j en la matriz ordenada (Figura 8.1b), actualiza el valor de T $[i]$ a j. En nuestro ejemplo, la ordenación finaliza con T $= (5, 2, 7, 6, 4, 3, 8, 9, 0, 1)$.

◇ **Ejercicio 8.4 (sol. en pág. 1108):** Muéstrese cómo se utiliza T para crear la salida principal del codificador, L e I.

La implementación del decodificador es sencilla, porque no hay necesidad de crear matrices de $n \times n$. El decodificador recibe bits que son códigos de Huffman. Los utiliza para crear los códigos de C, descomprimir cada uno de ellos al mismo tiempo que se crea —con el procedimiento inverso de mover–al–frente— el siguiente símbolo de L. Cuando L está lista, el decodificador la ordena y guarda el resultado en F, generando un array T en el proceso. Después de eso, reconstruye S a partir de L, T, e I. Por consiguiente, el decodificador necesita al menos tres estructuras en cualquier momento: las dos cadenas L y F (que constan típicamente de un byte por símbolo), y la matriz T (con al menos dos bytes por puntero, para permitir valores grandes de n).

> Describimos una ordenación de bloques, un algoritmo de compresión de datos sin pérdidas, y nuestra aplicación de dicho algoritmo. Comparamos el rendimiento de nuestra implementación con los compresores de datos disponibles más frecuentes, ejecutándolos en el mismo hardware.
>
> M. Burrows y D. J. Wheeler, 10 de mayo, 1994.

8.2. Clasificación de símbolos

Al igual que tantas otras ideas en el ámbito de la información y de los datos, la idea de la compresión de texto mediante la clasificación de símbolos es debida a Claude Shannon, el creador de la teoría de la información. En su clásica obra sobre el contenido de información del texto en inglés [Shannon 51] él describe un método para determinar experimentalmente la entropía de tales textos. En un experimento típico, un pasaje de texto tiene que ser predicho, carácter por carácter, por una persona (el examinado). En una versión del método, el examinado predice el siguiente carácter y luego el examinador le dice si la predicción fue correcta o, si no lo era, cuál es el siguiente carácter. En otra versión, el examinado tiene que continuar la predicción hasta que obtenga la respuesta correcta. El examinador entonces utiliza el número de respuestas incorrectas para estimar la entropía del texto.

Como se vio, en la última versión de la prueba, los examinados humanos fueron capaces predecir el siguiente carácter en una conjetura alrededor del 79 % del tiempo y rara vez necesitaron más de 3−4 conjeturas. La Tabla 8.3 muestra la distribución de turnos según lo publicó Shannon.

El hecho de que esta probabilidad sea tan sesgada implica una baja entropía (la conclusión de Shannon era que la entropía del texto en inglés está en el rango de $0{,}6 - 1{,}3$ bits por letra), que a su vez implica la posibilidad de una muy buena compresión.

# de conjeturas	1	2	3	4	5	> 5
Probabilidad	79 %	8 %	3 %	2 %	2 %	5 %

Tabla 8.3: Probabilidades de conjeturas del texto en inglés.

El método de clasificación de símbolos de esta sección [Fenwick 96] se basa en la última versión de la prueba de Shannon. El método utiliza el contexto C del símbolo actual S (los N símbolos que preceden a S) para preparar una lista de símbolos que son propensos a seguir a C. La lista está ordenada del más probable al menos probable. La posición de S en esta lista (la numeración de la posición comienza en 0) es escrita entonces por el codificador, después de haber sido debidamente codificada, en la secuencia de salida. Si el programa funciona como un humano examinado, podemos esperar que el 79 % de los símbolos codificados recaigan en 0 (primera posición en la lista de clasificación), creando rachas (*runs*) de ceros, que pueden ser fácilmente comprimidas mediante RLE.

Los diversos métodos basados en el contexto descritos en otra parte de este libro, especialmente PPM, utilizan el contexto para estimar las probabilidades de los símbolos. Tienen que generar y sacar símbolos de escape al cambiar los contextos. Por el contrario, la clasificación de símbolos no estima las probabilidades y no utiliza símbolos de escape. La ausencia de escapes parece ser la característica principal que contribuye al excelente rendimiento del método. A continuación se presenta un bosquejo de los principales pasos del algoritmo de codificación.

- *Paso 0*: El *índice de clasificación* (un entero contando la posición de S en la lista de clasificación) es puesto a 0.
- *Paso 1*: Se utiliza un diccionario de tipo LZ77, con un buffer de búsqueda que contiene texto que ya ha sido introducido y codificado, y con un buffer de preanálisis contiene texto nuevo sin procesar. El texto más reciente en el búfer de búsqueda se convierte en el *contexto actual* C. El símbolo de más a la izquierda, R, en el búfer de preanálisis (inmediatamente a la derecha de C) es el *símbolo actual*. El buffer de búsqueda se explora de derecha a izquierda (del texto reciente al viejo) para encontrar cadenas que coincidan con C. Este proceso es muy similar al descrito en la Sección 3.16 (compresión LZP). Se selecciona el emparejamiento más largo (si hay varios de ellos más largos, se selecciona el más reciente). La longitud del emparejamiento, N, se convierte en el *orden actual*. El símbolo P que sigue a la cadena coincidente (i.e., inmediatamente a la derecha de la misma) es examinado. Éste es el símbolo que ocupó el primer lugar por el algoritmo. Si P es idéntica a R, la búsqueda ha terminado y el algoritmo genera el índice de clasificación (que es actualmente 0).
- *Paso 2*: Si P es diferente de R, el índice de clasificación se incrementa en 1, P es declarado *excluido*, y los otros emparejamientos de orden N, si los hay, son examinados de la misma manera. Asume que Q es el símbolo que sigue a tal emparejamiento. Si Q está en la lista de símbolos excluidos, entonces no tiene sentido examinarlo, y la búsqueda continúa con el siguiente emparejamiento. Si Q no ha sido excluido, se le compara con R. Si son idénticos, la búsqueda ha terminado, y el algoritmo de codificación genera el índice de clasificación. De lo contrario el índice de clasificación es incrementado en 1, y Q es excluido.
- *Paso 3*: Si ninguno de los emparejamientos de orden N es seguido por un símbolo idéntico a R, el orden del emparejamiento se decrementa en 1, y el buffer de búsqueda es escaneado de nuevo de derecha a izquierda (desde el texto más reciente al más viejo) para encontrar las cadenas de tamaño $N-1$ que coincidan con C. Para cada fracaso en este análisis, el índice de clasificación es incrementado en 1, y Q es excluido.
- *Paso 4*: Cuando el orden del emparejamiento recorre todo el camino hacia abajo hasta 0, el símbolo R es comparado con los símbolos en una lista que contiene el alfabeto entero, de nuevo

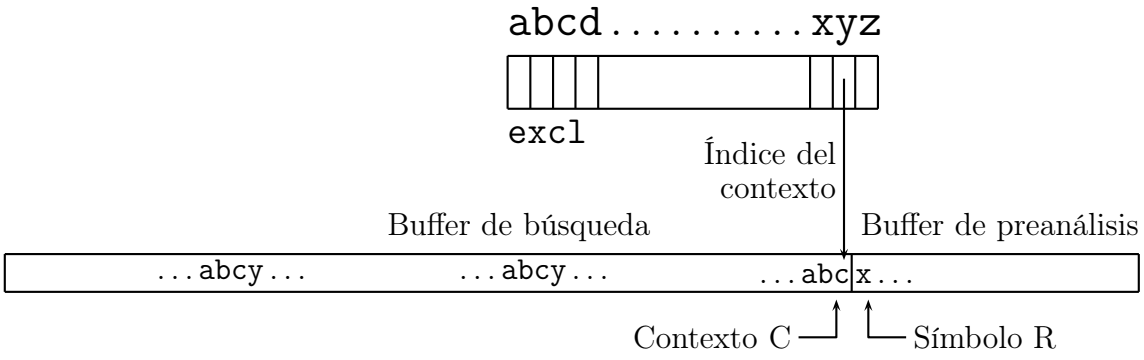


Figura 8.4: Mecanismos de exclusión.

usando exclusiones e incrementando el índice de clasificación. Si el algoritmo llega a este paso, encontrará R en esta lista, y emitirá el valor de salida actual del índice de clasificación (que por tanto será normalmente un número largo).

Aquí se discuten algunos detalles de la implementación.

1. Implementación de la exclusión. Cuando una cadena S que coincide con C es encontrada, el símbolo P inmediatamente a la derecha de S es comparado con R. Si P y R son diferentes, P debe ser declarado excluido. Ésto significa que las futuras ocurrencias de P deben ser ignoradas. La primera implementación de la exclusión que viene a la mente es una lista a la que se agregan los símbolos excluidos. La búsqueda en dicha lista, sin embargo, es tiempo consumido, y es posible hacerlo mucho mejor.

   El método aquí descrito utiliza un array `excl` indexado por los símbolos del alfabeto. Si el alfabeto está formado, por ejemplo, por sólo 26 letras, el array tendrá 26 ubicaciones indexadas desde la `a` hasta la `z`. La Figura 8.4 muestra una sencilla aplicación que requiere sólo un paso para determinar si un símbolo dado está excluido. Supongamos que el contexto actual C es la cadena "...`abc`". Sabemos que la `c` permanecerá en el contexto, incluso si el algoritmo tiene que recorrer todo el camino hasta el orden 1. El algoritmo, por lo tanto, prepara un puntero a `c` (que se llama *índice del contexto*). Supongamos que el análisis detecta otra cadena `abc`, seguida por `y`, y la compara con el contexto actual. Coinciden, pero van seguidas por símbolos diferentes. La decisión es excluir `y`, y ésto se efectúa ajustando el elemento `excl`[y] del array al índice del contexto (i.e., para apuntar a `c`). Mientras el algoritmo escanea posibles coincidencias para el mismo contexto C, el índice del contexto seguirá siendo el mismo. Si posteriormente encuentra otra cadena coincidente `abc`, también seguida por `y`, el algoritmo compara `excl`[y] con el índice del contexto, y encuentra que son iguales, por lo que sabe que `y` ya ha sido excluido. Cuando se cambia al siguiente contexto actual no hay necesidad de inicializar o modificar los punteros en el array `excl`.

2. Se ha mencionado anteriormente que el escaneo y la búsqueda de coincidencias con el contexto actual C se efectúa mediante un método similar al utilizado por LZP. El lector debe revisar la Sección 3.16 antes de seguir leyendo. Recordemos que N (el orden) es inicialmente desconocido. El algoritmo tiene que explorar el buffer de búsqueda y encontrar el emparejamiento más largo con el contexto actual. Una vez hecho esto, la longitud N del emparejamiento se convierte en el orden actual. Por consiguiente el proceso se comienza mediante una operación hash de los dos símbolos de más a la derecha del contexto actual C y los utiliza para localizar una posible coincidencia.

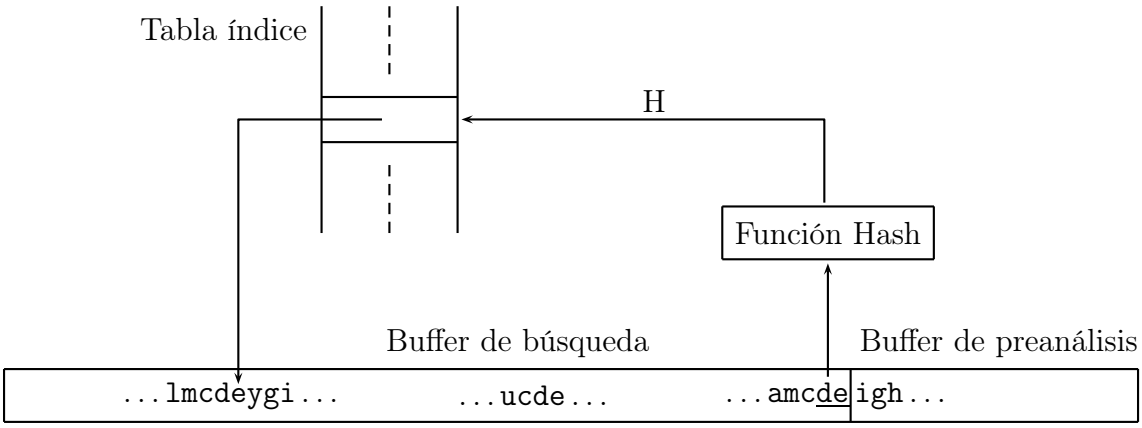


Figura 8.5: Métodos de búsqueda de cadenas y comparación.

La Figura 8.5 muestra el contexto actual "...`amcde`". Suponemos que ya ha ido emparejado con alguna cadena de longitud 3 (i.e., una cadena "...`cde`"), y tratamos de hacerla coincidir con una cadena más larga. Se realiza una operación hash con los dos símbolos "`de`" para producir un puntero a la cadena "`lmcde`". El problema es comparar el contexto actual con "`lmcde`" y encontrar si, y en qué medida, coinciden. Ésto se efectúa mediante las tres reglas siguientes:

a) *Regla 1*: Se comparan los símbolos que preceden (i.e., a la izquierda de) `cde` en las dos cadenas. En nuestro ejemplo son ambos `m`, por lo que el emparejamiento es ahora de tamaño 4. Se repite esta regla hasta que falla. Se determina el orden N del emparejamiento. Una vez conocido el orden, el algoritmo puede necesitar decrementarlo posteriormente y comparar las cadenas más cortas. En tal caso, esta regla tiene que ser modificada. En lugar de comparar los símbolos *precedentes* de las cadenas, lo que deberían compararse son los símbolos de más a la izquierda de las dos cadenas.

b) *Regla 2*: (Todavía no estamos seguros de si las dos cadenas son idénticas.) Se comparan los símbolos centrales de las dos cadenas. En nuestro caso, puesto que las cadenas tienen una longitud de 4, ésto sería, o bien `c`, o bien `d`. Si la comparación falla, las cadenas son diferentes. De lo contrario, se utiliza la Regla 3.

c) *Regla 3*: Se comparan las cadenas símbolo por símbolo para determinar si finalmente son idénticas.

Parece innecesariamente engorroso tener que pasar por las tres reglas cuando sólo la tercera es realmente necesaria. Sin embargo, las dos primeras reglas son sencillas, e identifican el 90 % de los casos en los que las dos cadenas son diferentes. La Regla 3, que es lento, tiene que ser aplicada sólo si las dos primeras reglas no han identificado las cadenas como diferentes.

3. Si el algoritmo de codificación tiene que decrementar el orden en todo el recorrido hacia abajo hasta 1, se enfrenta a un problema especial. Ya no puede efectuar operaciones hash de dos símbolos. La búsqueda para cadenas coincidentes de orden 1 (i.e., símbolos individuales) requiere, por lo tanto, un método diferente que se ilustra mediante la Figura 8.6. Se muestran dos listas enlazadas, una enlazando las ocurrencias de `s` y la otra enlazando las ocurrencias de `i`. Observe cómo sólo están vinculadas ciertas ocurrencias de `s`, mientras que otras son ignoradas. La regla es saltar una ocurrencia de `s` que es seguida por un símbolo que ya se ha visto. En consecuencia,

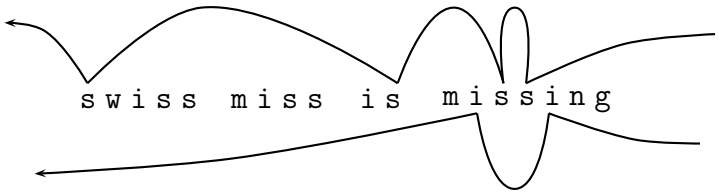


Figura 8.6: Búsqueda de contexto de orden 1.

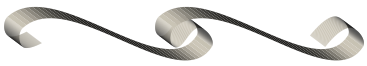
las primeras ocurrencias, `si`, `ss`, `s␣`, y `sw` son enlazadas, mientras que otras apariciones de `s` son omitidas.

La lista que enlaza estas ocurrencias de `s` comienza vacía y se construye gradualmente, a medida que se introduce más texto y es desplazado en el buffer de búsqueda. Cuando se crea un nuevo contexto con `s` como su símbolo de más a la derecha, la lista es actualizada. Ésto se hace localizando el símbolo a la derecha del nuevo `s`, digamos `a`, explorando la lista en busca de un enlace `sa`, eliminándolo si lo encuentra (no puede existir más de uno), y enlazándolo al `s` actual a la lista.

Esta lista hace que sea fácil buscar y localizar todas las ocurrencias de orden 1 del contexto `s` que van seguidas por símbolos diferentes (i.e., con exclusiones).

Dicha lista debe ser construida y actualizada para cada símbolo del alfabeto. Si el algoritmo es implementado para manejar símbolos de 8 bits, entonces se necesitan 256 de tales listas y tienen que ser actualizadas.

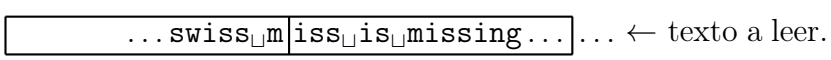
Los detalles de implementación anteriores muestran la complejidad de este método. Es lento, pero que produce una excelente compresión.



8.3. ACB

No hay muchos detalles disponibles de la implementación real del ACB, un método de compresión de texto original y altamente eficiente de George Buyanovsky. La única documentación disponible en la actualidad está en ruso [Buyanovsky 94] y no está actualizada. (Una interpretación informal en inglés, debida a Leonid Broukhis, está disponible en `http://www.cbloom.com/news/leoacb.html`.) El nombre ACB proviene de Associative Coder (de) Buyanovsky (código asociativo (de) Buyanovsky). Comenzamos con un ejemplo y seguimos con algunas características y una variante. Los detalles precisos del algoritmo ACB, sin embargo, son aún desconocidos. El lector también debe consultar [Buyanovsky 94], [Fenwick 96], y [Lambert 99].

Supongamos que el texto "...`swiss␣miss␣is␣missing`..." forma parte del flujo de datos de entrada. El método utiliza un buffer LZ77 deslizante donde asumimos que los siete primeros símbolos ya han sido introducidos y están ahora en el buffer de búsqueda. El buffer de preanálisis comienza con la cadena "`iss␣is`...".



...s|wiss␣m	1	...swiss␣|m	
...sw|iss␣m	2	...swi|ss␣m	
...swi|ss␣m	3	...s|wiss␣m	
...swis|s␣m	4	...swis|s␣m	
...swiss|␣m	5	...swiss|␣m	
...swiss␣|m	6	...sw|iss␣m	
(a)		(b)	

Tabla 8.7: Seis contextos y contenidos.

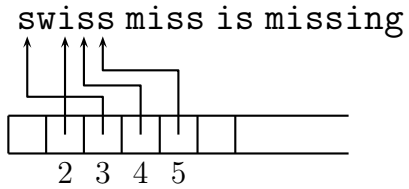


Figura 8.8: Organización del diccionario.

A medida que el texto es introducido y codificado, todos los contextos se van colocando en un diccionario, cada uno con el texto que le sigue. Este texto se llama cadena de *contenido* del contexto. Las seis entradas (contexto|contenido) que corresponden a los siete símbolos de más a la derecha en el buffer de búsqueda se muestran en la Tabla 8.7a. El diccionario es luego ordenado por los contextos, *de derecha a izquierda*, como muestra la Tabla 8.7b. Tanto los contextos como los contenidos son ilimitados. Se asume que son tan largos como sea posible, pero pueden incluir solamente símbolos del buffer de búsqueda, ya que el buffer de preanálisis es desconocido para el decodificador. De esta manera ambos, codificador y decodificador, pueden crear y actualizar sus diccionarios al unísono.

(De Internet.) ACB — Codificador asociativo de Buyanovsky. El uso del algoritmo ACB asegura un coeficiente de compresión récord.

8.3.1. El codificador

El contexto actual ...swiss␣m es emparejado por el codificador con las entradas del diccionario. Las entradas 2 y 3 ofrecen la mejor coincidencia (la comparación se efectúa de derecha a izquierda). Asumimos arbitrariamente que el algoritmo de emparejamiento selecciona la entrada 2 (obviamente, el algoritmo no toma decisiones arbitrarias y es la misma para el codificador y el decodificador). El contenido actual iss... también es comparado con el diccionario. La mejor opción es la entrada 6. Los cuatro símbolos iss␣ coinciden, por lo que la salida es $(6-2, 4, \texttt{i})$, un triplete que comprime los *cinco* símbolos iss␣i. El primer elemento del triplete es la distancia d entre el mejor contenido y el mejor contexto emparejado (puede ser negativa). El segundo elemento es el número l de símbolos coincidentes (que esperamos grande, pero también puede ser cero). El tercer elemento es el primer símbolo no coincidente en el buffer de preanálisis (según el espíritu de LZ77). Los cinco símbolos comprimidos son añadidos a los campos "contenido" de *todas* las entradas del diccionario (Tabla 8.9a) y también son desplazados en el buffer de búsqueda. Estos símbolos también generan cinco entradas para ser agregadas al diccionario, que se muestran, reordenadas, en la Tabla 8.9b.

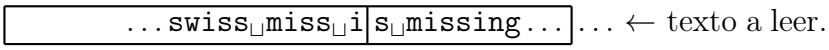
El nuevo buffer de deslizamiento es:

...s|wiss␣ miss␣i	1 ...swiss␣miss␣|i
...sw|iss␣ miss␣i	2 ...swiss␣|miss␣i
...swi|ss␣ miss␣i	3 ...swiss␣mi|ss␣i
...swis|s␣ miss␣i	4 ...swi|ss␣ miss␣i
...swiss|␣miss␣i	5 ...swiss␣m|iss␣i
...swiss␣|miss␣i	6 ...s|wiss␣ miss␣i
...swiss␣m|iss␣i	7 ...swiss␣mis|s␣i
...swiss␣mi|ss␣i	8 ...swis|s␣ miss␣i
...swiss␣mis|s␣i	9 ...swiss␣miss|␣i
...swiss␣miss|␣i	10 ...swiss|␣miss␣i
...swiss␣miss␣|i	11 ...sw|iss␣ miss␣i
(a)	(b)

Tabla 8.9: Once contextos y sus contenidos.

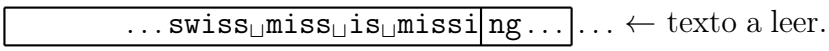
...s|wiss␣miss␣is␣missi	1 ...swiss␣miss␣is␣|missi
...sw|iss␣miss␣is␣missi	2 ...swiss␣miss␣|is␣missi
...swi|ss␣miss␣is␣missi	3 ...swiss␣|miss␣is␣missi
...swis|s␣miss␣is␣missi	4 ...swiss␣miss␣i|s␣missi
...swiss|␣miss␣is␣missi	5 ...swiss␣miss␣is␣mi|ssi
...swiss␣|miss␣is␣missi	6 ...swiss␣mi|ss␣is␣missi
...swiss␣m|iss␣is␣missi	7 ...swi|ss␣miss␣is␣missi
...swiss␣mi|ss␣is␣missi	8 ...swiss␣miss␣is␣m|issi
...swiss␣mis|s␣is␣missi	9 ...swiss␣m|iss␣is␣missi
...swiss␣miss|␣is␣missi	10 ...s|wiss␣miss␣is␣missi
...swiss␣miss␣|is␣missi	11 ...swiss␣miss␣is|␣missi
...swiss␣miss␣i|s␣missi	12 ...swiss␣miss␣is␣mis|si
...swiss␣miss␣is|␣missi	13 ...swiss␣mis|s␣is␣missi
...swiss␣miss␣is␣|missi	14 ...swis|s␣miss␣is␣missi
...swiss␣miss␣is␣m|issi	15 ...swiss␣miss␣is␣miss|i
...swiss␣miss␣is␣mi|ssi	16 ...swiss␣miss|␣is␣missi
...swiss␣miss␣is␣mis|si	17 ...swiss|␣miss␣is␣missi
...swiss␣miss␣is␣miss|i	18 ...sw|iss␣miss␣is␣missi
(a)	(b)

Tabla 8.10: Dieciocho contextos y sus contenidos.



La mejor coincidencia de contexto la ofrecen las entradas 2 y 3 (arbitrariamente asumimos que el algoritmo de emparejamiento selecciona la entrada 3). El contenido que encaja más es la entrada 8. Los seis símbolos `s␣miss` coinciden, por lo que la salida es $(8-3, 6, \texttt{i})$, un triplete que comprime siete símbolos. Los siete símbolos son agregados al campo "contenido" de cada entrada del diccionario y también son desplazados en el buffer de búsqueda. Se añaden siete nuevas entradas al diccionario, que se muestran en la Tabla 8.10a (desordenadas) y 8.10b (ordenadas).

El nuevo buffer de deslizamiento es:



(Observe que cada diccionario ordenado es una permutación de los símbolos de texto en el buffer de búsqueda. Esta característica del ACB se parece al método de Burrows-Wheeler, Sección 8.1.)

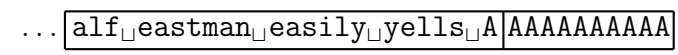
La mejor coincidencia de contexto son ahora las entradas 6 ó 7 (suponemos que es seleccionado el 6), pero no hay ninguna coincidencia de contenido, ya que ningún contenido comienza con una `n`. Ningún símbolos coincide, por lo que la salida es $(0, 0, \texttt{n})$, un triplete que comprime el símbolo único `n` (en realidad genera expansión). Este símbolo debe ahora ser añadido al diccionario y también desplazado en el buffer de búsqueda. (Fin del ejemplo.)

◇ **Ejercicio 8.5 (sol. en pág. 1108):** ¿Por qué este triplete tiene como primer elemento un cero?

8.3.2. El decodificador

El decodificador ACB construye y actualiza el diccionario al unísono con el codificador. En cada paso, el codificador y el decodificador tienen el mismo diccionario (mismos contextos y contenidos). La diferencia entre ellos es que el decodificador no tiene los datos del buffer de preanálisis. Sin embargo, el decodificador tiene los datos en el buffer de búsqueda, y los utiliza para encontrar la mejor coincidencia de contexto para, por ejemplo, la entrada de diccionario `t`. Esto se efectúa antes de que el decodificador introduzca cualquier cosa. Posteriormente, introduce un triplete $(\texttt{d}, \texttt{l}, \texttt{x})$ y agrega la distancia `d` a `t` para encontrar el contenido que mejor empareje con `c`. El decodificador simplemente copia `l` símbolos de la parte del contenido de entrada `c`, anexa el símbolo `x`, y envía la cadena resultante al flujo de datos descomprimidos. Esta cadena también se utiliza para actualizar el diccionario.

Observe que la parte del contenido de entrada `c` puede tener menos de símbolos que `l`. En este caso, la decodificación se hace algo más complicada y se asemeja al ejemplo de LZ77 (de la Sección 3.3)



(El autor está en deuda con Donna Klaasen por señalar ésto.)

Una versión modificada de ACB escribe pares (distancia, longitud del emparejamiento) en el *stream* comprimido en lugar de tripletes. Cuando la longitud `l` del emparejamiento es cero, escribe el código del símbolo en bruto (típicamente en ASCII u 8 bits), en vez de un par. Cada salida —un par o un código en bruto— ahora debe ser precedida por un flag o bandera que indique su tipo.

El diccionario puede ser organizado como una lista de punteros al buffer de búsqueda. La Figura 8.8 muestra cómo la entrada 4 del diccionario apunta a la segunda `s` de `swiss`. Siguiendo este puntero,

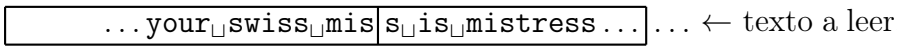
es fácil localizar tanto el contexto de la entrada 4 (el buffer búsqueda a la izquierda del puntero, el texto pasado) como su contenido (la parte del buffer de búsqueda a la derecha del puntero, el texto futuro).

Parte del excelente rendimiento de ACB se atribuye a la forma en que codifica las distancias `d` y las longitudes del emparejamiento `l`, que son su principal salida. Desafortunadamente, los detalles son desconocidos.

Es evidente que ACB está algo relacionado tanto con LZ77 como con LZ78. Lo que no es inmediatamente obvio es que ACB también está relacionado con el método de clasificación de símbolos (Sección 8.2). La distancia `d` entre las entradas, mejor contenido y mejor contexto, puede considerarse una medida de clasificación. En este sentido ACB es un método de compresión *phrase-ranking* (de *clasificación de expresiones*).

8.3.3. Una variación

Aquí tenemos una variación del método ACB básico que es más lento, requiere una ordenación adicional para cada emparejamiento, pero es más eficiente. Asumimos que la cadena



está en los búferes de búsqueda y de preanálisis. Denotamos esta cadena por `S`. Parte del diccionario actual (ordenado según el contexto, como de costumbre) se muestra en la Tabla 8.11a, donde las ocho primeras entradas y las cinco últimas proceden del buffer de búsqueda actual `your␣swiss␣mis`, y se supone que las diez entradas centrales proceden de los datos más antiguos.

Todas las entradas del diccionario cuyos campos de contexto concuerden con el buffer de búsqueda en al menos `k` símbolos —donde `k` es un parámetro, fijado a 9 en nuestro ejemplo— son seleccionados y se convierten en la *lista asociativa*, que se muestra en la Tabla 8.11b. Observe que estas entradas concuerdan con el buffer de búsqueda en diez símbolos, pero asumimos que `k` ha sido fijado a 9. Todas las entradas en la lista asociativa tienen contextos de `k` símbolos idénticos y representan entradas del diccionario con contextos similares al buffer de búsqueda (de ahí el nombre "asociativo").

La lista asociativa está ahora ordenada en orden ascendente según el **contenido**, generando la Tabla 8.12a. Ahora es obvio que `S` puede colocarse entre las entradas 4 y 5 de la lista ordenada (Tabla 8.12b).

Puesto que cada una de estas tres líneas está ordenada, podemos olvidarnos temporalmente de que están formados por caracteres, y simplemente considerarlos tres cadenas de bits ordenadas que pueden escribirse como en la Tabla 8.13a. Los bits `xx...x` son la parte donde las tres líneas concuerdan (la cadena `swiss␣mis` | `s␣is␣`), y los bits `zz...z` son una coincidencia completa entre la entrada 4 y el buffer de preanálisis (la cadena `mist`). Todo lo que el codificador tiene que emitir es el índice 4, el bit subrayado (que denotamos por `b` y que puede, por supuesto, ser cero), y la longitud `l` de la cadena `zz...z`. En consecuencia, la salida del codificador es el triplete (`4`, `b`, `l`).

En nuestro ejemplo `S` concuerda mejor con la entrada que le precede. En algunos casos puede concordar mejor con la entrada siguiente, como en la Tabla 8.13b (donde el bit `b` es mostrado como un cero).

◇ **Ejercicio 8.6 (sol. en pág. 1108):** Muéstrese por qué la configuración de la Tabla 8.13c es imposible.

El decodificador mantiene el diccionario sincronizado con el codificador, por lo que puede crear la misma lista asociativa, ordenarla y utilizar las partes idénticas (la intersección) de las entradas 4 y 5 para identificar la cadena `xx...x`. Después utiliza `l` para identificar la parte `zz...z` en la

```
          ...your␣|swiss␣mis
    ...your␣swiss␣|mis
  ...your␣swiss␣mi|s
       ...your␣swi|ss␣mis
   ...your␣swiss␣m|is
             ...yo|ur␣swiss␣mis
           ...your|␣swiss␣mis
         ...your␣s|wiss␣mis
      ...young␣mis|creant...
    ...unusual␣mis|fortune...
      ...plain␣mis|ery...
   ...no␣swiss␣mis|spelled␣it␣so...
   ...no␣swiss␣mis|s␣is␣mistaken...
   ...or␣swiss␣mis|read␣it␣to...
...your␣swiss␣mis|s␣is␣missing...
 ...his␣swiss␣mis|s␣is␣here...
  ...my␣swiss␣mis|s␣is␣trouble...
    ...always␣mis|placed␣it...
     ...your␣swis|s␣mis
    ...your␣swiss|␣mis
           ...you|r␣swiss␣mis
      ...your␣sw|iss␣mis
            ...y|our␣swiss␣mis
```

(a)

```
swiss␣mis|spelled␣it␣so...
swiss␣mis|s␣is␣s␣mistaken...
swiss␣mis|read␣it␣to...
swiss␣mis|s␣is␣missing...
swiss␣mis|s␣is␣here...
swiss␣mis|s␣is␣trouble...
```

(b)

Tabla 8.11: (a) Diccionario ordenado. (b) Lista asociativa.

```
1 swiss␣mis|read␣it␣to...
2 swiss␣mis|s␣is␣here...
3 swiss␣mis|s␣is␣missing...
4 swiss␣mis|s␣is␣s␣mistaken...
5 swiss␣mis|s␣is␣trouble...
6 swiss␣mis|spelled␣it␣so...
```

(a)

```
4. swiss mi|s is mistaken...
S. swiss mi|s is mistress...
5. swiss mi|s is trouble...
```

(b)

Tabla 8.12: (a) Lista asociativa ordenada. (b) Tres líneas.

```
4. xx...x0zz...z0A
S. xx...x0zz...z1B
5. xx...x1CC...
```

(a)

```
4. xx...x0CC...
S. xx...x1zz...z0B
5. xx...x1zz...z1A
```

(b)

```
4. xx...x0CC...
S. xx...x1zz...z1B
5. xx...x1zz...z0A
```

(c)

Tabla 8.13: (a, b) Dos posibilidades, y (c) Una imposibilidad, de tres líneas.

```
                zz . . . . . z
S. xx . . . x0|01011110001011|0A
S. xx . . . x0|01011110001011|1B
5. xx . . . x1 CC . . .
```

Tabla 8.14: Un ejemplo.

entrada 4 y genera la cadena de bits `xx...xb̃zz...zb`(donde b̃ es el complemento de `b`) como la salida descomprimida del triplete (`4`, `b`, `1`).

Esta variante puede ser mejorada (produciendo una mejor compresión pero más lenta) si en lugar de `1`, el codificador genera el número `q` de bits b̃ en la parte `zz...z`. Ésto mejora la compresión, ya que $q \leq 1$. El decodificador entonces comienza a copiar los bits desde la sección `zz...z` de la entrada 4 hasta que encuentra la $(q+1)$-ésima ocurrencia de b̃, que ignora. Ejemplo: si $b = 1$ y la sección `zz...z` es `01011110001011` (precedida por b̃ = `0`, y seguida por `b` = `1`), entonces `q` = `6`. Las tres líneas se muestran en la Tabla 8.14. Es fácil ver cómo el decodificador puede crear la sección de 14 bits `zz...z` copiando los bits de la entrada 4 hasta que encuentra el séptimo `0`, que ignora. La salida del codificador es, pues, el triplete (codificado) $(4, 1, 6)$ en vez de $(4, 1, 14)$. Escribir el valor 6 (codificado) en lugar de 14 en el *stream* comprimido mejora algo el rendimiento de la compresión global.

Otra mejora posible es eliminar las entradas idénticas en la lista asociativa ordenada. Esta técnica puede ser llamada *exclusión de frase*, en analogía con las técnicas de exclusión de PPM y el método de clasificación de símbolos. En nuestro ejemplo, Tabla 8.12a, no existen entradas idénticas, pero si hubiera habido alguna, la exclusión habría reducido el número de entradas a menos de 6.

◇ **Ejercicio 8.7 (sol. en pág. 1108):** ¿Cómo mejoraria ésto la compresión?

La principal fortaleza del ACB se deriva de la forma en que opera. Selecciona las entradas de diccionario con contextos que son similares al contexto actual (el buffer de búsqueda), luego ordena las entradas seleccionadas por el contenido y selecciona el mejor emparejamiento del contenido. Esto es lento y también requiere un diccionario enorme (un diccionario pequeño no proporcionaría buenas correspondencias) pero produce una excelente compresión basada en el contexto sin necesidad de símbolos de escape o cualquier otra estratagema "artificial".

8.3.4. Archivos de contexto

Una característica interesante del ACB es su capacidad para crear y utilizar *archivos de contexto*. Cuando un archivo `abc.ext` es comprimido, el usuario puede especificar la creación de un archivo de contexto llamado, por ejemplo, `abc.ctx`. Este archivo contiene el diccionario final generado durante la compresión de `abc.ext`. Posteriormente, el usuario puede comprimir otro archivo `lmn.xyz` pidiendo al ACB que utilice `abc.ctx` como archivo de contexto. El archivo `lmn.xyz` será comprimido utilizando el diccionario de `abc.ext`. Después de ésto, ACB reemplazará el contenido de `abc.ctx`. En lugar del diccionario original, ahora contendrá el diccionario de `lmn.xyz` (que no fue usado en la compresión real de `lmn.xyz`). Si el usuario desea mantener el contenido original de `abc.ctx`, sus atributos pueden establecerse a "sólo lectura". Los archivos de contexto pueden ser muy útiles, como ilustran los siguientes ejemplos.

1. Un escritor envía por correo electrónico (*e-mail*) un gran manuscrito a un editor. Debido a su tamaño, el archivo manuscrito debe ser enviado comprimido. La primera vez que se hace ésto, el escritor pide al ACB la creación de un archivo de contexto, entonces envía al editor por e-mail tanto el manuscrito comprimido como el archivo de contexto. Los dos archivos deben enviarse por correo electrónico, por lo que la compresión no sirve de mucho esta primera vez.

El editor descomprime el manuscrito usando el archivo de contexto, lo lee y responde con las modificaciones propuestas para el manuscrito. El escritor modifica el manuscrito, lo comprime de nuevo con el mismo archivo de contexto, y lo envía de nuevo por e-mail, pero esta vez sin el archivo de contexto. El archivo de contexto del escritor ha sido actualizado, por lo que el escritor no puede utilizarlo para descomprimir lo que acaba de enviar por correo electrónico (pero en ese momento no lo necesita). El editor todavía tiene el archivo de contexto original, por lo que puede descomprimir la segunda versión del manuscrito, proceso durante el cual ACB crea un nuevo archivo de contexto para que el editor lo utilice la próxima vez.

2. La colección completa de las historias de detectives de un autor famoso debe ser comprimida y se guarda como un fichero. Dado que todos los archivos son historias de detectives y son todos del mismo autor, es lógico suponer que tienen estilos similares de escritura y por tanto contextos similares. Se selecciona una historia para servir como "entrenamiento" de los archivos. Se comprime y se crea un archivo de contexto. Este archivo de contexto se guarda de forma permanente y se utiliza para comprimir y descomprimir todos los otros archivos en el fichero.

3. Un autor de *shareware*[2] escribe una aplicación `abc.exe` que es utilizada (y pagada) por mucha gente. El autor decide hacer la versión 2 disponible. Comienza comprimiendo la versión anterior, mientras crea un archivo de contexto `abc.ctx`. El archivo resultante comprimido no es necesario y se elimina inmediatamente. El autor luego utiliza `abc.ctx` como archivo de contexto para comprimir su versión 2, y posteriormente elimina `abc.ctx`. El resultado es un archivo comprimido (i.e., pequeño), que contiene la versión 2, y se coloca en internet, para ser descargado por los usuarios de la versión 1. Cualquier persona que tenga una versión se puede descargar el resultado y descomprimirlo. Todo lo que necesitan es comprimir su versión 1 con el fin de obtener un archivo de contexto, y después utilizar ese archivo de contexto para descomprimir lo que haya descargado.

> Este algoritmo... es sencillo para implementaciones en software y hardware.
>
> —George Buyanovsky

8.4. Semejanza de contextos basados en el orden

La idea de la similitud de contextos es un "pariente" del método de clasificación de símbolos de la Sección 8.2 y del método de Burrows-Wheeler (Sección 8.1). En contraste con el método de Burrows-Wheeler, el método de semejanza de contextos de esta sección es adaptativo.

El método utiliza la similitud del contexto para ordenar los contextos vistos previamente por orden lexicográfico inverso. Basándose en la secuencia ordenada de contextos, se asigna un *rango* al símbolo siguiente. Los rangos se escriben en el *stream* comprimido y posteriormente son utilizados por el decodificador para reconstruir los datos originales. El *stream* comprimido también incluye cada uno de los distintos símbolos de entrada en bruto (formato *raw*), y estos datos también son utilizados por el decodificador.

El codificador: El codificador lee la entrada símbolo a símbolo y mantiene una lista ordenada de pares (contexto, símbolo). Cuando el símbolo siguiente es introducido, el codificador inserta un par nuevo en el lugar que le corresponde en la lista, y utiliza la lista para asignar un rango al símbolo. El rango es escrito por el codificador en el *stream* comprimido y a veces es seguido por el mismo símbolo

[2] Un autor que ofrece una aplicación *shareware*, lo hace gratuitamente para el usuario, pero la aplicación no será funcionalmente completa. Para ello, el usuario deberá pagar al autor.

#	contexto	símbolo
0	λ	`b`
1	`ba`	`c`
2	`bacacaba`	x
3	`baca`	`c`
4	`bacaca`	`b`
5	`b`	`a`
6	`bacacab`	`a`
7	`bac`	`a`
8	`bacac`	`a`

Tabla 8.15: La lista ordenada para `bacacaba`.

en formato raw. El funcionamiento del codificador se ilustra mejor con un ejemplo. Supongamos que la cadena introducida hasta ahora es `bacacaba`, y que el siguiente símbolo (aún no leído por el codificador) se denota por x. La lista actual se muestra en la Tabla 8.15, donde λ representa la cadena vacía.

Cada una de las nueve entradas en la lista se compone de un contexto y el símbolo que siguió al contexto en el *stream* de entrada (excepto la entrada 2, donde la entrada es aún desconocida). La lista está ordenada por los contextos, pero en orden inverso. Se asume, por definición, que la cadena vacía, es menor que cualquier otra cadena. Ésta es seguida por todos los contextos que van a terminar con una "`a`" (hay cuatro de ellos), y se clasifican de acuerdo con el segundo símbolo de la derecha, luego con el tercer símbolo, y así sucesivamente. Éstos son seguidos por todos los contextos que terminan con "`b`", luego por los que terminan con "`c`", y así sucesivamente. El contexto actual `bacacaba` pasa a ser el número 2 en esta lista. Una vez que el decodificador ha decodificado los ocho primeros símbolos, tendrá disponible la misma lista de nueve entradas.

El codificador ahora clasifica a los contextos en la lista de acuerdo con cuán similares son al contexto 2. Es evidente que el contexto 1 es el más parecido a 2, ya que comparten dos símbolos. Por consiguiente, la clasificación se inicia con el contexto 1. El contexto 1 es comparado entonces con los siete contextos restantes: 0 y 3–8. Este contexto termina con una "`a`", por lo que es similar a los contextos 3 y 4. Seleccionamos el contexto 3 como el más parecido al 1, ya que es más corto que el 4. Esta regla de selección del contexto más corto es arbitraria. Simplemente garantiza que el decodificador será capaz de construir el mismo rango. La clasificación hasta el momento es $1 \to 3$. El contexto 3 es ahora comparado con los otros seis contextos. Es claro que es más similar al contexto 4, ya que comparten el último símbolo. La clasificación hasta ahora es, por tanto, $1 \to 3 \to 4$. El contexto 4 es ahora comparado con los otros cinco contextos. Estos contextos no comparten ningún sufijo con el 4, por lo que el más corto, el contexto 0, es seleccionado como el más similar. La clasificación hasta el momento es $1 \to 3 \to 4 \to 0$. El contexto 0 es ahora comparado con los cuatro contextos restantes. No comparte ningún sufijo con ellos, por lo que el más corto de los cuatro, el contexto 5, es seleccionado. La clasificación hasta ahora es $1 \to 3 \to 4 \to 0 \to 5$.

◇ **Ejercicio 8.8 (sol. en pág. 1108):** Continúese este proceso.

Como la respuesta a este ejercicio muestra, la clasificación final de los contextos es:

$$\begin{array}{cccccccccc} 1 & \to & 3 & \to & 4 & \to & 0 & \to & 5 & \to & 6 & \to & 7 & \to & 8 \\ \texttt{c} & & \texttt{c} & & \texttt{b} & & \texttt{b} & & \texttt{a} & & \texttt{a} & & \texttt{a} & & \texttt{a} \end{array} . \tag{8.2}$$

Esta clasificación de los contextos es ahora utilizado por el codificador para asignar un rango al siguiente símbolo x. El codificador introduce x y lo compara, de izquierda a derecha, con los símbolos mostrados en la Ecuación (8.2). El rango de x es uno más que el número de símbolos distintos que se han encontrado en la comparación. Por consiguiente, si x es el símbolo "`c`", que se encuentra

(a)

0	λ	b
1	b	x

(b)

0	λ	b
1	ba	x
2	b	a

(c)

0	λ	b
1	ba	c
2	b	a
3	bac	x

(d)

0	λ	b
1	ba	c
2	baca	x
3	b	a
4	bac	a

(e)

0	λ	b
1	ba	c
2	baca	c
3	b	a
4	bac	a
5	bacac	x

(f)

0	λ	b
1	ba	c
2	baca	c
3	bacaca	x
4	b	a
5	bac	a
6	bacac	a

Tabla 8.16: Construcción de las listas ordenadas para `bacacaba`.

inmediatamente en la Ecuación (8.2), no existen símbolos distintos, y a "`c`" se le asigna un rango 1. Si x es "`b`", el codificador se encuentra con sólo un símbolo distinto, a saber, "`c`", antes de llegar a "`b`", por lo que el rango de "`b`" se convierte en 2. Si x es "`a`", su rango es 3, y si x es un símbolo diferente, su rango es 4 [uno más que el número de símbolos distintos en la Ecuación (8.2)]. El codificador escribe el rango en el *stream* comprimido, y si el rango es 4, el codificador también escribe el símbolo real en formato raw, siguiendo al rango.

A continuación mostramos los primeros pasos en la codificación de la cadena de entrada `bacacaba`. El primer símbolo "`b`" se escribe en la salida sin cambios (formato raw). No se le asigna ningún rango. El codificador (y también el decodificador, al unísono) construye la tabla de una entrada (λ `b`). El segundo símbolo, "`a`", es introducido. Su contexto es "`b`" y el codificador inserta la entrada (`b a`) en la lista. La nueva lista se muestra en la Tabla 8.16a con x denotando el nuevo símbolo introducido, ya que este símbolo no es conocido todavía por el decodificador. Los siguientes cinco pasos se resumen en la Tabla 8.16b–f.

La Ecuación: (8.3) muestra los distintos rangos de contexto:

$$
\begin{array}{lll}
\begin{array}{c}0\\ \texttt{b}\end{array}, &
\begin{array}{ccc}0&\to&2\\ \texttt{b}&&\texttt{a}\end{array}, &
\begin{array}{ccccc}0&\to&2&\to&1\\ \texttt{b}&&\texttt{a}&&\texttt{c}\end{array},\\[1ex]
\multicolumn{2}{l}{\begin{array}{ccccccc}1&\to&0&\to&3&\to&4\\ \texttt{c}&&\texttt{a}&&\texttt{a}&&\texttt{a}\end{array},} &
\begin{array}{ccccccccc}4&\to&0&\to&3&\to&1&\to&2\\ \texttt{a}&&\texttt{b}&&\texttt{a}&&\texttt{c}&&\texttt{c}\end{array},\\[1ex]
\multicolumn{3}{l}{\begin{array}{ccccccccccc}2&\to&1&\to&0&\to&4&\to&5&\to&6\\ \texttt{c}&&\texttt{c}&&\texttt{b}&&\texttt{a}&&\texttt{a}&&\texttt{a}\end{array}.}
\end{array}
\qquad (8.3)
$$

Con esta información, es fácil construir manualmente el *stream* comprimido. El primer símbolo, "`b`", es emitido en formato raw. Al segundo símbolo, "`a`", se le asigna el rango 1 y también es emitido siguiendo a su rango. A la primera "`c`" se le asigna el rango 2 y también es emitido (cada símbolo de entrada distinto es sacado en formato raw, tras su rango, la primera vez que es leído y procesado). A la segunda "`a`" se le asigna el rango 2, porque hay un símbolo distinto ("`b`") que le precede en la lista de rangos de contexto. A la segunda "`c` se le asigna el rango 1.

◇ **Ejercicio 8.9 (sol. en pág. 1108):** Complétese el *stream* de salida.

◇ **Ejercicio 8.10 (sol. en pág. 1109):** Practique sus conocimientos del codificador en la corta cadena de entrada `ubladiu`. Muéstrense los contextos ordenados y el rango de contexto después de cada símbolo de entrada. Muéstrese también la salida producida por el codificador.

El decodificador: Una vez que se comprende el funcionamiento del codificador, es evidente que el decodificador puede imitar al codificador. Puede construir y mantener la tabla de contextos ordenada

```
Introduzca el primer elemento. Éste es un símbolo en bruto (raw). Emitirlo.
while not fin-de-archivo
  Introduzca el siguiente elemento. Éste es el rango de un símbolo.
  If este rango es > el nro total de símbolos distintos vistos hasta el momento
    then  Introduzca el siguiente elemento. Éste es un símbolo raw. Emitirlo.
    else  Trasladar el rango a un símbolo usando el rango de contexto actual.
          Emitir este símbolo.
  endif
  La cadena que haya sido emitida hasta el momento es el contexto actual.
    Insértela en la tabla de contextos ordenados.
endwhile
```

Figura 8.17: El algoritmo de decodificación.

$S[1\ldots0] =$	λ	λ|y
$S[1\ldots7] =$	yabreca	yabreca|b
$S[1\ldots2] =$	ya	ya|b
$S[1\ldots8] =$	yabrecab	yabrecab|r
$S[1\ldots3] =$	yab	yab|r
$S[1\ldots6] =$	yabrec	yabrec|a
$S[1\ldots5] =$	yabre	yabre|c
$S[1\ldots9] =$	yabrecabr	yabrecabr|$s_{10}\uparrow\downarrow$
$S[1\ldots4] =$	yabr	yabr|e
$S[1\ldots1] =$	y	y|a
	(a)	(b)

Tabla 8.18: (a) Lista ordenada para yabrecabr. (b) Inserción del prefijo siguiente.

a medida que lee el *stream* comprimido, y utilizar la tabla para regenerar los datos originales. El algoritmo de decodificación se muestra en la Figura 8.17.

La estructura de datos: Las primeras versiones del método de clasificación del contexto utilizaban un árbol binario de decisión para almacenar los distintos contextos. Ésto era lento, por lo que la longitud de los contextos tuvo que ser limitada a ocho símbolos. La nueva versión, descrita en [Yokoo 99a], utiliza un *lista de prefijos* como estructura de datos, y es lo suficientemente rápida como para permitir contextos de longitud ilimitada. Denotamos los símbolos de entrada por s_i. Sea $S[1\ldots n]$ la cadena $s_1 s_2 \ldots s_n$ de n símbolos. Utilizamos la notación $S[i\ldots j]$ para denotar la subcadena $s_i \ldots s_j$. Si $i > j$, entonces $S[i\ldots j]$ es la cadena vacía λ.

Como ejemplo, consideremos la cadena de 9 símbolos $S[1\ldots9]$ = yabrecabr. La Tabla 8.18a muestra los diez prefijos de esta cadena (incluyendo el prefijo vacío) ordenados en orden lexicográfico inverso. La Tabla 8.18b considera los prefijos, los contextos y muestra los diez pares (contexto, símbolo). Esta tabla ilustra cómo insertar el prefijo siguiente, que consiste en el siguiente símbolo de entrada s_{10} adicionado al contexto actual yabrecabr. Si s_{10} no es cualquiera de los símbolos del extremo derecho de los prefijos (i.e., si no es ninguno de abcery), entonces s_{10} determina la posición del prefijo siguiente. Por ejemplo, si s_{10} es "x", entonces el prefijo yabrecabrx debe insertarse entre yabr e y. Si, por el contrario, s_{10} es uno de abcery, comparamos yabrecabrs_{10} con los prefijos que le preceden y lo seguimos en la tabla, hasta encontrar la primera coincidencia.

Por ejemplo, si s_{10} es "e", entonces, comparando yabrecabre con los prefijos posteriores (yabr e

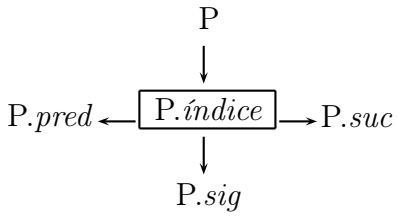


Figura 8.19: Representación del nodo $S\,[1 \ldots \mathrm{P}.\mathit{índice}]$.

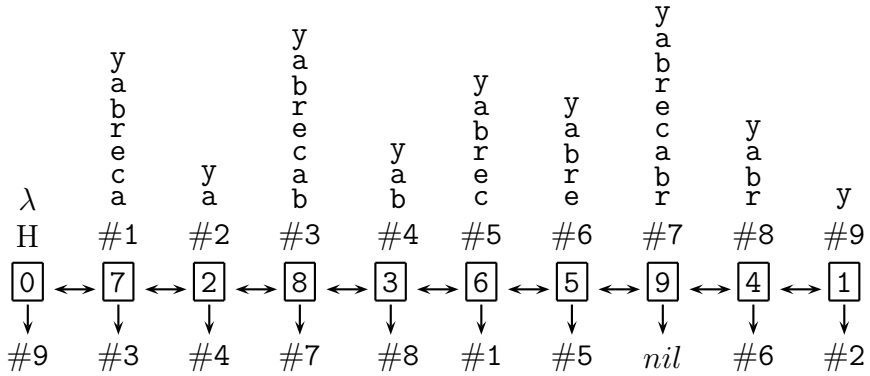


Figura 8.20: Lista de prefijos para `yabrecabr`.

`y`) no se encuentra una coincidencia, pero comparado con los prefijos le preceden concuerda con `yabre` en un solo paso. En tal caso (una coincidencia encontrada mientras se busca hacia arriba), la regla es que `yabrecabre` debe convertirse en el predecesor de `yabre`. Del mismo modo, si s_{10} es "`a`", entonces comparando `yabrecabra` con los prefijos precedentes se encuentra una coincidencia con `ya`, por lo `yabrecabra` debe convertirse en el predecesor de `ya`. Si s_{10} es "`r`", entonces comparando `yabrecabrr` con los prefijos posteriores empareja con `yabr` y la regla en este caso (una coincidencia encontrada mientras se busca hacia abajo) es que `yabrecabrr` debe convertirse en el sucesor de `yabr`.

Una vez que se ha captado ésto, la estructura de datos de la lista de prefijos es fácil de comprender. Es una lista doblemente enlazada donde cada nodo está asociado con una entrada prefijo $S\,[1 \ldots i]$ y contiene el entero i y tres punteros. Dos punteros (*pred* y *suc*) apuntan a los nodos predecesores y sucesores, y el tercero (siguiente o *sig*) apunta al nodo asociado con el prefijo $S\,[1 \ldots i+1]$. La Figura 8.19 muestra los punteros de un nodo general que representa a la subcadena $S\,[1 \ldots \mathrm{P}.\mathit{índice}]$. Si un nodo se corresponde con la cadena de entrada completa $S\,[1 \ldots n]$, entonces su campo *sig* se establece en el puntero nulo *nil*. La lista de prefijos se inicializa con un nodo especial H que representa la cadena vacía. Algunas operaciones sobre listas se simplifican si la lista termina con otro nodo especial T. La Figura 8.20 muestra la lista de prefijos para `yabrecabr`.

Así es como se inserta el siguiente prefijo en la lista de prefijos. Asumimos que la lista ya contiene todos los prefijos de la cadena $S\,[1 \ldots i]$ y que el siguiente prefijo $S\,[1 \ldots i+1]$ debe ser insertado ahora. Si el nuevo símbolo de entrada s_{i+1} precede o sucede a todos los símbolos vistos hasta el momento, entonces el nodo que representa $S\,[1 \ldots i+1]$ debe insertarse al comienzo de la lista (i.e., a la derecha del nodo especial H) o al final de la lista (i.e., a la izquierda del nodo especial T), respectivamente. De lo contrario, si s_{i+1} no está incluido en $S\,[1 \ldots i]$, entonces existe una posición única Q que satisface:

$$S\,[Q.\mathit{índice}] < s_{i+1} < S\,[Q.\mathit{suc}.\mathit{índice}]\,. \tag{8.4}$$

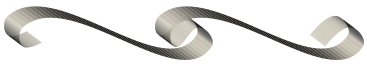
El nuevo nodo para $S\,[1 \ldots i+1]$ debe ser insertado entre los dos nodos apuntados por Q y $Q.\mathit{suc}$.

Si el símbolo s_{i+1} ya ha aparecido en $S[1 \ldots i]$, entonces las desigualdades en la Ecuación (8.4) pueden convertirse en igualdades. En este caso, el inmediato predecesor o sucesor de $S[1 \ldots i+1]$ tiene el mismo último símbolo que s_{i+1}. Si ésto es cierto para el sucesor inmediato (i.e., si el sucesor inmediato $S[1 \ldots j+1]$ de $S[1 \ldots i+1]$ satisface $s_{j+1} = s_{i+1}$), entonces $S[1 \ldots j]$ precede a $S[1 \ldots i]$. El nodo correspondiente a $S[1 \ldots j]$ debe ser el primer nodo que satisfaga $s_{j+1} = s_{i+1}$ al recorrer la lista desde el nodo actual hasta el principio. Podemos comprobar si el siguiente símbolo coincide con s_{i+1} siguiendo el puntero sig.

Inversamente, si el último símbolo s_{j+1} del sucesor inmediato $S[1 \ldots j+1]$ de $S[1 \ldots i+1]$ es igual a s_{i+1}, entonces el nodo para $S[1 \ldots j]$ debe ser el primero en satisfacer $s_{j+1} = s_{i+1}$ cuando la lista es recorrida desde el nodo actual hasta el último nodo. En cualquier caso se parte del nodo actual y se busca hacia delante y hacia atrás, buscando el nodo de $S[1 \ldots j]$ comparando los últimos símbolos, s_{j+1} con s_{i+1}. Una vez que el nodo de $S[1 \ldots j]$ ha sido localizado, el nodo para $S[1 \ldots j+1]$ puede alcanzarse en una etapa siguiendo el puntero sig. El nuevo nodo de $S[1 \ldots i+1]$ debe insertarse adyacente al nodo de $S[1 \ldots j+1]$.

[Yokoo 99a] contiene un análisis de períodos de complejidad y detalles acerca de esta estructura.

El método de ordenación de contexto fue publicado por primera vez en [Yokoo 96] con el análisis y la evaluación añadida en [Yokoo 97]. Ha sido desarrollado y mejorado aún más por su desarrollador, Hidetoshi Yokoo. Ha sido comunicado al autor en [Yokoo 99b].



8.5. Cadenas dispersas

Independientemente de lo que los datos de entrada representen —texto, binarios, imágenes, o cualquier otra cosa— podemos pensar en el *stream* de entrada como una cadena de bits. Si la mayoría de los bits son ceros, la cadena es *dispersa*. Las cadenas dispersas pueden ser comprimidas de manera muy eficiente, y esta sección describe métodos desarrollados específicamente para esta tarea. Antes de llegar a los métodos individuales puede ser útil convencer al lector de que las cadenas dispersas no son un concepto teórico sino que ocurren comúnmente en la práctica. He aquí algunos ejemplos.

1. Un dibujo. Imagine un dibujo, técnico o artístico, hecho con un lápiz negro sobre papel blanco. Si el dibujo no es muy complejo, la mayor parte del mismo permanece blanco. Cuando tal dibujo es escaneado y digitalizado, la mayoría de los píxeles resultantes son blancos, y el porcentaje de los negros es pequeño. El mapa de bits (*bitmap*) resultante es un ejemplo de una cadena dispersa.

2. Un índice de un bitmap para una base de datos grande. Imagine una gran base de datos de documentos de texto. Un mapa de bits para tal base de datos es un conjunto de cadenas de bits (o vectores de bits) que hace que sea fácil identificar todos los documentos en los que aparece una determinada palabra ω. Para aplicar un bitmap, primero hay que preparar una lista de todas las palabras ω_j distintas en todos los documentos. Suponemos que hay W de tales palabras. El siguiente paso es revisar cada documento d_i y preparar una cadena de bits D_i, de W bits de longitud, que contenga un 1 en la posición j si la palabra ω_j aparece en el documento d_i. El mapa de bits es el conjunto de todas esas cadenas de bits. Dependiendo de la base de datos, tales cadenas de bits pueden ser dispersas.

   (La indexación de grandes bases de datos es una operación importante, ya que una base de datos informatizada debe facilitar la búsqueda. El método tradicional de indexación consiste en preparar una *concordancia*. Originalmente, la palabra concordancia se refería cualquier índice global de la Biblia, pero hoy en día existen concordancias de las obras completas de Shakespeare,

Wordsworth, y muchas otras. Una concordancia computarizada se llama *archivo invertido*. Este archivo incluye una entrada para cada término en los documentos que constituyen la base de datos. Una entrada es una lista de punteros a todas las ocurrencias del término, de forma similar a un índice de un libro. Un puntero puede ser el número de un capítulo, de una sección, de una página, un par de líneas de la página, de una frase, o incluso de una palabra. Un archivo invertido donde punteros apuntan a palabras individuales se considera de *grano fino*. Un archivo invertido donde se señala, por ejemplo, un capítulo se considera de *grano grueso*. Un archivo invertido de grano fino puede parecer preferible, pero debe usar punteros grandes, y como consecuencia, puede llegar a ser tan grande que puede tener que ser almacenado en forma comprimida.)

3. Las cadenas dispersas también han sido mencionadas en la sección 4.8.3, en relación con JPEG.

Los métodos aquí descritos (excepto la compresión de prefijos, Sección 8.5.5) se deben a [Fraenkel y Kein 85]. La Sección 2.5 discute los códigos de Golomb e ilustra su aplicación a las cadenas dispersas.

8.5.1. Creando bits OR

Este método comienza con una cadena dispersa L_1 de tamaño n_1 bits. En el primer paso, L_1 se divide en k subcadenas de igual tamaño. En cada subcadena se efectúa la operación lógica OR sobre todos los bits[3], y los resultados (un bit por subcadena) se convierten en la cadena L_2, que será comprimida en el paso 2. Todas las subcadenas de ceros de L_1 ahora han sido eliminadas. Aquí tenemos un ejemplo de una cadena dispersa de 64 bits, que dividimos en 16 subcadenas de tamaño 4 cada una:

$$\begin{aligned} L_1 =& 0000 \mid 0000 \mid 0000 \mid 0100 \mid 0000 \mid 0000 \mid 0000 \mid 1000 \mid 0000 \\ & \mid 0000 \mid 0000 \mid 0000 \mid 0010 \mid 0000 \mid 0000 \mid 0000. \end{aligned}$$

Después de realizar la operación OR sobre cada subcadena de 4 bits obtenemos la cadena de 16 bits $L_2 = 0001 \mid 0001 \mid 0000 \mid 1000$.

En el paso 2, el mismo proceso se aplica a L_2, y el resultado es la cadena de 4 bits $L_3 = 1101$, que es lo suficientemente corta como para no necesitar más etapas de compresión. Después de la eliminación de todas las subcadenas de ceros en L_1 y L_2, terminamos con las tres cortas cadenas:

$$L_1 = 0100 \mid 1000 \mid 0010, \quad L_2 = 0001 \mid 0001 \mid 1000, \quad L_3 = 1101.$$

El *stream* de salida se compone de siete subcadenas de 4 bits ¡en vez de las 16 originales! (Son necesarios algunos números más, para indicar la longitud de cada subcadena.)

El decodificador funciona de forma diferente (este es un método de compresión asimétrico). Comienza con L_3 y considera cada uno de sus bits a 1, un puntero a una subcadena de L_2; y cada uno de sus bits a 0, un puntero a una subcadena de todo ceros que no está almacenada en L_2. De esta manera, la cadena L_2 puede ser reconstruida a partir de L_3, y la cadena L_1, a su vez, puede ser reconstruida a partir de L_2. La Figura 8.21 ilustra este proceso. Las subcadenas mostradas entre corchetes son las que no están contenidas en el stream comprimido.

◇ **Ejercicio 8.11 (sol. en pág. 1109):** Este método se vuelve altamente ineficiente para las cadenas que no son dispersas, y puede fácilmente dar lugar a expansión. Analícese el peor caso, donde cada grupo de L_1 es distinto de cero.

[3] Una operación OR sobre dos bits produce un 1 cuando cualquiera de los dos bits (o ambos) es 1, y 0 en el otro caso.

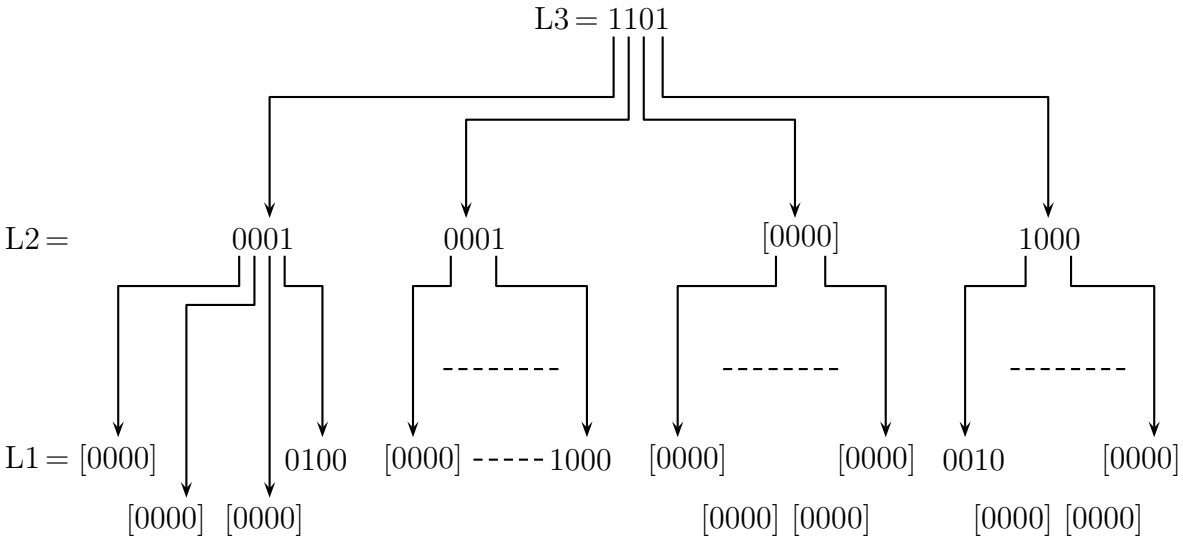


Figura 8.21: Reconstrucción de L_1 a partir de L_3.

8.5.2. Códigos de tamaño variable

Comenzamos con un *stream* de entrada que es una cadena dispersa L de n bits. La dividimos en grupos de l bits cada uno, y asignamos a cada grupo un código de tamaño variable. Puesto que un grupo de l bits puede tener uno de 2^l valores, necesitamos 2^l códigos. Como L es dispersa, la mayor parte de los grupos estará formada por l ceros, lo que implica que el código de tamaño variable asignado al grupo de l ceros (el grupo de ceros) debe ser el más corto (quizás de sólo un bit). Los otros $2^l - 1$ códigos de tamaño variable pueden ser asignados arbitrariamente, o de acuerdo con las frecuencias de ocurrencia de los grupos. La última opción requiere un paso adicional en el stream de entrada para calcular las frecuencias. En el caso ideal, donde todos los grupos son de ceros, y cada uno se codifica como un bit, el stream de salida estará formado por n/l bits, obteniendo una razón de compresión de $1/l$. Ésto muestra que, en principio, la razón de compresión puede mejorarse incrementando l, pero en la práctica, un l grande significa muchos códigos, que, a su vez, aumentan el tamaño del código y disminuyen la relación de compresión para una cadena "promedio" L.

Puede desarrollarse un mejor enfoque una vez que nos damos cuenta de que un stream de entrada disperso debe contener **runs** de grupos de ceros.[4] Un run de grupos de ceros puede estar formado por uno, dos, o hasta n/l de tales grupos. Es posible asignar códigos de tamaño variable para los runs de grupos de ceros, así como a los grupos no cero (con al menos un elemento distinto de cero), y la Tabla 8.22 ilustra este enfoque para el caso de 16 grupos. El problema es que hay $2^l - 1$ grupos no cero y n/l posibles run lengths[5] de grupos de ceros. Normalmente, n es grande y l es pequeño, por lo que n/l es grande. Si aumentamos l, entonces n/l se hace más pequeño, pero $2^l - 1$ se hace más grande. Por consiguiente, siempre acabamos con muchos códigos, lo que implica códigos largos.

Un enfoque más prometedor es dividir los run lengths (que son números enteros entre 1 y n/l) en clases, asignando un código C_i de tamaño variable a cada clase i, y asignando un código de dos partes a cada run length. Imagine una racha (*run*) de r grupos de ceros, donde r pasa a pertenecer a la clase i, y pasa a ser el tercero en esta clase. Cuando en el flujo de datos de entrada se encuentra una racha de r grupos de ceros, el código de r se escribe en el flujo de datos de salida. Tal código consta de

[4] Varios bloques de ceros seguidos.
[5] Longitudes de *run*. Número de grupos de ceros en el run de ceros.

Tamaño del run length	Run de ceros	Grupo no cero	
1	0000	1	0001
2	0000 0000	2	0010
3	0000 0000 0000	3	0011
⋮	⋮ ⋮	⋮	
16	0000 0000 ... 0000	15	1111
	(a)		(b)

Tabla 8.22: (a) n/l Run lengths. (b) $2^l - 1$ Grupos no cero.

Run length	Código	$r - 2^{i-1}$	$i - 1$	Código de Huffman | m
1	C_1	$1 - 2^{1-1} = 0$	0	00010
2	C_2	$2 - 2^{2-1} = 0$	1	00011 | 0
3	C_2	$3 - 2^{2-1} = 1$	1	0010 | 1
4	C_3	$4 - 2^{3-1} = 0$	2	0011 | 00
5	C_3	$5 - 2^{3-1} = 1$	2	010 | 01
6	C_3	$6 - 2^{3-1} = 2$	2	011 | 10
7	C_3	$7 - 2^{3-1} = 3$	2	1 | 11
8	C_4	$8 - 2^{4-1} = 0$	3	00001 | 000
9	C_4	$9 - 2^{4-1} = 1$	3	000001 | 001
⋮				⋮
15	C_4	$15 - 2^{4-1} = 7$	3	000000000001 | 111

Tabla 8.23: $\log_2(n/l)$ Clases de Run Lengths.

dos partes: La primera es el código C_i de la clase, y el segundo es 2, la posición de r en su clase (las posiciones están numeradas a partir de cero). La experiencia en el diseño de algoritmos y números binarios sugiere la siguiente definición de clases: Un run length de r grupos de ceros está en la clase i si $2^{i-1} \le r < 2^i$, donde $i = 1, 2, \ldots, [\log_2(n/l)]$. Esta definición implica que la posición de r en su clase es $m = r - 2^{i-1}$, un número que puede escribirse en $(i-1)$ bits. La Tabla 8.23 muestra las cuatro clases para el caso $n/l = 16$ (16 grupos). Observe cómo los números m son escritos como números de $(i-1)$ bits, por lo que para $i = 1$ (la primera clase), m no es necesario. Los códigos de Huffman C_i de tamaño variable mostrados en la tabla tienen únicamente un propósito ilustrativo y se basan en la suposición (arbitraria) de que los run lengths más comunes son 5, 6, y 7.

Es fácil ver en la tabla que un run de 16 grupos de ceros (que corresponde a una secuencia de entrada de todo ceros) no pertenece a ninguna de las clases. Por lo tanto, debe ser asignado a un código de tamaño variable especial. El número total de códigos de tamaño variable requeridos en este enfoque es, por lo tanto, $2^l - 1$ (para los grupos no cero) más $[\log_2(n/l)]$ (para los run lengths grupos de ceros) más 1 para el caso especial en el que todos los grupos son iguales a cero. Un ejemplo típico es una cadena de entrada de 1 megabit (i.e., $n = 2^{20}$). Asumiendo $l = 8$, el número de los códigos es $2^8 - 1 + \log_2\left(2^{20}/8\right) + 1 = 256 - 1 + 17 + 1 = 273$. Con $l = 4$ el número de códigos es $2^4 - 1 + \log_2\left(2^{20}/4\right) + 1 = 16 - 1 + 18 + 1 = 34$; mucho más pequeño, pero se requieren más códigos para codificar la misma cadena de entrada.

Grupos no cero								Clases	
1	111111	5	01111	9	00111	13	00011	C_1	110
2	111110	6	01110	10	00110	14	00010	C_2	10
3	111101	7	01101	11	00101	15	00001	C_3	010
4	111100	8	01100	12	00100	*16*	00000	C_4	1110

Tabla 8.24: Veinte códigos.

El funcionamiento del decodificador es sencillo. Lee el siguiente código de tamaño variable, que representa, o bien un grupo no cero de l bits, o bien una serie (*run*) de r grupos de ceros, o bien una cadena de entrada de todo ceros. En el primer caso, el decodificador crea el grupo no cero. En el segundo caso, el código indica al decodificador el número de clase i. El decodificador entonces lee los $i-1$ bits siguientes bits para obtener el valor de m, y calcula $r = m + 2^{i-1}$ como el tamaño del run length de grupos de ceros. Después, el decodificador crea un run de r grupos de ceros. En el tercer caso el decodificador crea una cadena de n bits a cero.

Ejemplo: Una cadena de entrada de tamaño $n = 64$ dividida en 16 grupos de $l = 4$ bits cada uno. El número de códigos necesarios es de $2^4 - 1 + \log_2(64/4) + 1 = 16 - 1 + 4 + 1 = 20$. Asumimos arbitrariamente que cada uno de los 15 grupos no cero se produce con una probabilidad de $1/40$, y que la probabilidad de ocurrencia de los runs en las cuatro clases son $6/40$, $8/40$, $6/40$, y $4/40$, respectivamente. La probabilidad de ocurrencia de un run de 16 grupos se supone que es $1/40$. La Tabla 8.24 muestra los posibles códigos para cada grupo no cero y para cada clase de runs de grupos de ceros. El código de 16 grupos de ceros es 00000 (se corresponde con el 16 en cursiva).

◇ **Ejercicio 8.12 (sol. en pág. 1109):** Codificar la cadena de entrada:

0000 | 0000 | 0000 | 0100 | 0000 | 0000 | 0000 | 1000 | 0000 | 0000 | 0000 | 0000 | 0010 | 0000 | 0000 | 0000

utilizando los códigos de la Tabla 8.24.

8.5.3. Códigos de tamaño variable en base 2

Las clases definidas como en la sección anterior requieren un conjunto de $\left(2^l - 1\right) + [\log_2(n/l)] + 1$ códigos. El método es eficaz, pero lento, ya que el código para un *run* de grupos de ceros involucra tanto un código de clase C_i como la cantidad m. En esta sección, nos fijamos en una manera de manejar los runs de grupos de ceros definiendo códigos R_1, R_2, R_4, R_8,... para run lengths de $1, 2, 4, 8, \ldots, 2^i$ grupos de ceros. La representación binaria del número 17, e.g., es 10001, por lo que un run de 17 grupos de ceros sería codificado como R_{16} seguido por R_1. Puesto que los run lengths pueden ser de 1 a n/l, el número de códigos R_i requeridos es de $1 + (n/l)$, más que antes. La experiencia muestra, sin embargo, que los run largos son raros, por lo que los códigos de Huffman asignados a R_i deberían ser cortos para valores pequeños de i, y largos para grandes íes. Además de los códigos de R_i, aún son necesarios $2^l - 1$ códigos para los grupos no cero. En el caso $n = 64$, $L = 4$ tenemos 15 códigos para los grupos no cero y 7 códigos para R_1 a R_{64}. Un ejemplo está ilustrado en la Tabla 8.25 donde todos los códigos de grupos no cero son de 5 bits de longitud y comienzan con 0, mientras que los siete códigos de R_i comienzan con 1 y son de tamaño variable.

Los códigos de R_i no tienen que corresponder a potencias de 2. Ellos pueden estar basados en 3, 4, ó incluso números enteros más grandes. Echemos un vistazo rápido a los códigos de R_i octales (basados en el 8). Ellos son denotados mediante R_1, R_8, R_{64},.... Para codificar un run de 17 grupos de ceros ($17 = 2 \times 8^1 + 1 \times 8^0 = 21_8$) necesitamos **dos** copias del código para R_8, seguido por el código para R_1. El número de códigos R_i es menor, pero algunos tendrán que aparecer varias (hasta 7) veces.

La regla general es: Supongamos que los códigos de R_i se basan en el número B. Si identificamos un run de R grupos de ceros, primero tenemos que expresar R en la base B, a continuación, creamos

Grupos no cero		Run lengths	
1 0 0001	9 0 1001	R_1	1 1
2 0 0010	10 0 1010	R_2	1 01
3 0 0011	11 0 1011	R_4	1 001
4 0 0100	12 0 1100	R_8	1 00011
5 0 0101	13 0 1101	R_{16}	1 00010
6 0 0110	14 0 1110	R_{32}	1 00001
7 0 0111	15 0 1111	R_{64}	1 00000
8 0 1000			

Tabla 8.25: Códigos para R_i en base 2.

copias de los códigos R_i de acuerdo con los dígitos de dicho número. Si $R = d_3d_2d_1$ en base B, entonces la salida codificada para el run length R debería estar formada por d_1 copias de R_1, seguidas por d_2 copias de R_2 y por d_3 copias de R_3.

◇ **Ejercicio 8.13 (sol. en pág. 1110):** Codifíquese la cadena de entrada de 64 bits del Ejercicio 8.12 utilizando los códigos de Tabla 8.25.

8.5.4. Códigos de tamaño variable basados en Fibonacci

Los códigos R_i utilizados en el apartado anterior se basan en potencias de 2, ya que cualquier número entero positivo puede ser expresado en esta base utilizando sólo los dígitos 0 y 1. Resulta que los conocidos números de Fibonacci también tienen esta propiedad. Cualquier número entero positivo R puede expresarse como $R = b_1F_1 + b_2F_2 + b_3F_3 + b_4F_5 + \cdots$ (que es b_4F_5, no b_4F_4), donde los F_i son los números Fibonacci $1, 2, 3, 5, 8, 13, \ldots$ y los b_i son dígitos binarios. Los números de Fibonacci crecen más lentamente que las potencias de 2, lo que significa que son necesarios más de ellos para expresar un run length de R de grupos de ceros dado. Sin embargo, esta representación tiene la interesante propiedad de que la cadena $b_1b_2\ldots$ no contiene 1's adyacentes ([Knuth 73], ex. 34, p. 85). Si la representación de R en esta base está formada por d dígitos, realmente serían necesarios, a lo más $\lceil d/2 \rceil$ códigos F_i para codificar un run length de R grupos de ceros. Como ejemplo, el entero 33 es igual a la suma $1 + 3 + 8 + 21$, por lo que se expresa en la base de Fibonacci como el número de 7 bits 1010101. Por consiguiente, un run length de 33 grupos de ceros se codifica, en este método, como los cuatro códigos: F_1, F_3, F_8, y F_{21}.

La Tabla 8.26 es un ejemplo de los códigos de Fibonacci para el run length de grupos de ceros. Observe que con siete códigos de Fibonacci se pueden expresar sólo runs de hasta $1 + 2 + 3 + 5 + 8 + 13 + 21 = 53$ grupos. Como queremos un máximo de 64 grupos, necesitamos un código más. La Tabla 8.26 por tanto, tiene ocho códigos, frente a los siete de la Tabla 8.25.

◇ **Ejercicio 8.14 (sol. en pág. 1110):** Codifíquese la cadena de entrada de 64 bits del Ejercicio 8.12 utilizando los códigos de la Tabla 8.26.

Esta sección y la anterior sugieren que puede utilizarse cualquier sistema de numeración para construir los códigos de los runs de grupos de ceros. Sin embargo, son preferibles los sistemas de numeración basados en dígitos binarios, ya que ciertos códigos pueden ser omitidos en tal caso, y ningún código tiene que ser duplicado. Otra posibilidad es el uso de sistemas de numeración donde ciertas combinaciones de dígitos son imposibles. He aquí un ejemplo, también basado en los números de Fibonacci.

La bien conocida relación de recurrencia que satisfacen estos números es $F_i = F_{i-1} + F_{i-2}$. Lo que puede escribirse:

$$F_{i+2} = F_{i+1} + F_i = (F_i + F_{i-1}) + F_i = 2F_i + F_{i-1}.$$

Grupos no cero				Run lengths	
1	0 0001	9	0 1001	F_1	1 1
2	0 0010	10	0 1010	F_2	1 01
3	0 0011	11	0 1011	F_3	1 001
4	0 0100	12	0 1100	F_5	1 00011
5	0 0101	13	0 1101	F_8	1 00010
6	0 0110	14	0 1110	F_{13}	1 00001
7	0 0111	15	0 1111	F_{21}	1 00000
8	0 1000			F_{34}	1 000000

Tabla 8.26: Códigos para F_i basados en Fibonacci.

Los números producidos por esta relación también puede servir como base para un sistema de numeración que tiene dos propiedades interesantes: (1) Cualquier número entero positivo se puede expresar utilizando sólo los dígitos 0, 1, y 2. (2) Cualquier dígito 2 es seguido por un 0.

Los primeros números producidos mediante esta relación son: 1, 3, 7, 17, 41, 99, 239, 577, 1393 y 3363. Es fácil verificar los siguientes ejemplos:

1. $7000_{10} = 2001002001$ (ya que $7000_{10} = 2 \times 3363 + 239 + 2 \times 17 + 1$.
2. $168_{10} = 111111$.
3. $230_{10} = 201201$.
4. $271_{10} = 1001201$.

En consecuencia, un run de 230 grupos de ceros puede ser comprimido generando dos copias del código de Huffman de 99, seguido por el código de Huffman de 17, por dos copias del código de 7, y por el código de 1. Otra posibilidad es asignar a cada uno de los números de la base 1, 3, 7, etc, dos códigos de Huffman: uno para dos copias, y el otro para una copia.

8.5.5. Compresión mediante prefijos

El principio del método de compresión de prefijos [Salomon 00] es asignar una dirección a cada bit 1 en la cadena dispersa, dividir cada dirección en un prefijo y un sufijo, a continuación, seleccionar todos los bits a 1 cuyas direcciones tienen el mismo prefijo y enviarlas al archivo comprimido escribiendo el prefijo común, seguido por todos los sufijos diferentes. La compresión se logrará si acabamos con muchos bits a 1 que tengan el mismo prefijo. Con el fin de que varios bits a 1 tengan el mismo prefijo, sus direcciones deben estar cercanas. Sin embargo, en una cadena larga y dispersa, los bits a 1 pueden estar muy alejados. La compresión de prefijos trata de llevar juntos los bits a 1 que están separados en la cadena, dividiendo ésta en segmentos de igual tamaño que son luego colocados uno debajo del otro, creando una *matriz* dispersa de bits. Es fácil ver que los bits a 1 que están muy separados en la cadena original pueden acercarse en esta matriz. Como ejemplo, consideremos una cadena binaria de 2^{20} bits (un megabit). La distancia máxima entre los bits de la cadena original es de aproximadamente un millón, pero cuando la cadena se reorganiza como una matriz de dimensiones $2^{10} \times 2^{10} = 1024 \times 1024$, la distancia máxima entre los bits es de sólo unos mil.

Nuestro problema es asignar direcciones a los elementos de matriz de tal manera que (1) la dirección de un elemento de la matriz sea un número único y (2) los elementos cercanos sean asignados a direcciones que no difieran en mucho. La manera habitual de hacer referencia a los elementos de una matriz es mediante la fila y la columna. Podemos crear una dirección de un solo número concatenando los números fila y columna de un elemento, pero ésto no es satisfactorio. Consideremos, por ejemplo,

los dos elementos de la matriz en las posiciones (1 400) y (2 400). Ciertamente son vecinos cercanos, pero sus números son 1 400 y 2 400, ¡no muy cercanos!

Por consiguiente, utilizamos un método diferente. Pensamos en la matriz como una imagen digital donde cada bit se convierte en un píxel (blanco para un 0 y negro para un 1) y requerimos que esta imagen sea de tamaño $2^n \times 2^n$ para algún entero n. Ésto normalmente requiere la ampliación de la cadena original con bits a 0 hasta que su tamaño se convierta en una potencia de 2 (2^{2n}). El tamaño original de la cadena, por lo tanto, debe ser escrito, en bruto (formato *raw*), en el archivo comprimido para su uso por el descompresor. Si la cadena es dispersa, la imagen correspondiente tiene pocos píxeles negros. A esos píxeles se les asignan direcciones usando el concepto de *quadtree*.

Para comprender cómo se hace ésto, vamos a suponer que nos dan una imagen de tamaño $2^n \times 2^n$. La dividimos en cuatro cuadrantes y etiquetados $\left(\begin{smallmatrix} 0 & 1 \\ 2 & 3 \end{smallmatrix}\right)$. Observe que éstos son números de 2 bits. Cada cuadrante se divide posteriormente en cuatro subcuadrantes etiquetados de la misma manera. Por lo tanto, cada subcuadrante obtiene un número de cuatro bits (dos dígitos). Este proceso continúa de forma recursiva, y a medida que los subsubcuadrantes se hacen más pequeños, sus números se hacen más largos. Cuando este esquema de numeración se lleva a píxeles individuales, el número de un píxel resulta ser de $2n$ bits de longitud. La Figura 4.157, duplicada aquí, muestra la numeración de los píxeles en una imagen de $2^3 \times 2^3 = 8 \times 8$ y también una sencilla imagen formada por 18 píxeles negros. Cada número de píxel es de seis bits (tres dígitos) de largo, y abarcan desde 000 hasta 333. La cadena original utilizada para crear la imagen es:

100001000100010000100100000111100010010001000100110000000011000000

(donde los seis ceros a la derecha, o algunos de ellos, han sido añadidos para conseguir que el tamaño de la cadena sea una potencia de dos).

000	001	010	011	100	101	110	111
002	003	012	013	102	103	112	113
020	021	030	031	120	121	130	131
022	023	032	033	122	123	132	133
200	201	210	211	300	301	310	311
202	203	212	213	302	303	312	313
220	221	230	231	320	321	330	331
222	223	232	233	322	323	332	333

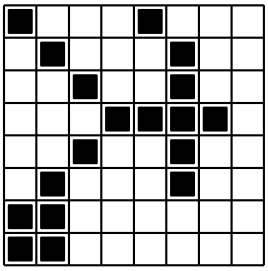


Figura 4.157: Ejemplo de compresión mediante prefijos.

El primer paso es usar los métodos de quadtree para averiguar los números id de tres dígitos de los 18 píxeles negros. Ellos son: 000, 101, 003, 103, 030, 121, 033, 122, 123, 132, 210, 301, 203, 303, 220, 221, 222, y 223.

El siguiente paso es seleccionar un valor de *prefijo*. Para nuestro ejemplo seleccionamos $P = 2$, una elección que se justifica a continuación. El código de un píxel se divide ahora en P dígitos de prefijo seguidos por $3 - P$ dígitos de sufijo. El último paso se centra sobre la secuencia de píxeles negros y selecciona todos los píxeles con el mismo prefijo. El primer prefijo es 00, por lo que se seleccionan todos los píxeles que comienzan por 00. Ellos son 000 y 003. Se eliminan de la secuencia original y se comprimen escribiendo el token 00|1|0|3 en el *stream* de salida. La primera parte de este token es un prefijo (00), la segunda parte es un recuento (1), y el resto son los sufijos de los dos píxeles con prefijo 00. Observe que un recuento de 1 implica dos píxeles. El recuento es siempre uno menos que el número de píxeles que están siendo contados. Dieciséis píxeles permanecen ahora en la secuencia original, y el primero de ellos tiene prefijo 10. Los dos píxeles con este prefijo, a saber, 101 y 103, son

eliminados y se comprimen escribiendo el token 10|1|1|3 en el *stream* de salida. Ésto continúa hasta que la secuencia original se convierte en vacía. El resultado final es la cadena de nueve tokens:

$$00|1|0|3 \quad 10|1|1|3 \quad 03|1|0|3 \quad 12|2|1|2|3 \quad 13|0|2 \quad 21|0|0 \quad 30|1|1|3 \quad 20|0|3 \quad 22|3|0|1|2|3,$$

o en binario (cada dígito, dos bits),

$$0000010011\ 0100010111\ 0011010011\ 011010011011\ 01110010$$
$$10010000\ 1100010111\ 10000011\ 10101100011011$$

(sin los espacios). Tal cadena puede decodificarse de manera única, ya que cada símbolo comienza con un prefijo de dos dígitos, seguido por un recuento c de un dígito, seguido por $(c+1)$ sufijos de 1 dígitos. Precediendo a los tokens, el archivo comprimido debe contener, en formato raw, el valor de n, el tamaño original de la cadena dispersa, y el valor de P que se utilizó en la compresión. Una vez comprendido ésto, es obvio que la descompresión es trivial. El descompresor lee los valores de n y P, y estos dos números son todo lo que necesita para leer y decodificar todos los tokens de forma inequívoca.

◇ **Ejercicio 8.15 (sol. en pág. 1110):** ¿Pueden los píxeles adyacentes tener prefijos diferentes?

Observe que los resultados de nuestro ejemplo produce expansión debido a que nuestra cadena binaria es corta y, por lo tanto, no dispersa. Una cadena dispersa tiene que ser de al menos decenas de miles de bits.

En general, el prefijo es de P dígitos (ó $2P$ bits) de longitud, y el recuento y cada sufijo son de $n-P$ dígitos cada uno. El número máximo de sufijos en un token es, por lo tanto, 4^{n-P} y el tamaño máximo de un símbolo es de $P+(n-P)+4^{n-P}\,(n-P)$ dígitos. Cada token corresponde a un prefijo diferente. Un prefijo tiene P dígitos, cada uno entre 0 y 3, por lo que el número máximo de tokens es de 4^P. Por consiguiente, toda la cadena comprimida ocupa como máximo

$$4^P\left[P+(n-P)+4^{n-P}\,(n-P)\right]=n\cdot 4^P+4^n\,(n-P)$$

dígitos. Para encontrar el valor óptimo de P diferenciamos esta expresión con respecto a P,

$$\frac{d}{dP}\left[n\cdot 4^P+4^n\,(n-P)\right]=n\cdot 4^P\ln 4-4^n,$$

e igualamos la derivada a cero. La solución es:

$$4^P=\frac{4^n}{n\cdot\ln 4},\quad \text{ó } P=\log_4\left[\frac{4^n}{n\cdot\ln 4}\right]=\frac{1}{2}\log_2\left[\frac{4^n}{n\cdot\ln 4}\right].$$

Para $n=3$ esto produce:

$$P\approx\frac{1}{2}\log_2\left[\frac{4^3}{3\times 1{,}386}\right]=\frac{\log_2 15{,}388}{2}=\frac{3{,}944}{2}=1{,}97.$$

Por esta razón se ha seleccionado $P=2$ en nuestro ejemplo. Un programa de compresión práctico debería contener una tabla con los valores de P calculados previamente para todos los valores previstos de n. La Tabla 8.27 muestra dichos valores para $n=1,2,\ldots,12$.

Este método para calcular el valor óptimo de P se basa en el peor de los casos. Utiliza el número máximo de sufijos en un token, pero muchos tokens tienen sólo unos pocos sufijos. También usa el número máximo de prefijos, pero en una cadena dispersa muchos prefijos pueden no producirse. Los experimentos indican que para grandes cadenas dispersas (correspondientes a valores de n de 9–12), la mejor compresión se obtiene si el valor de P seleccionado es uno menos que el valor indicado en

n:	1	2	3	4	5	6	7	8	9	10	11	12
P:	0,76	1,26	1,97	2,76	3,60	4,47	5,36	6,26	7,18	8,10	9,03	9,97

Tabla 8.27: Dependencia de P de n.

```
Clear[t]; t=Log[4]; (* log natural *)
Table[{n,N[0.5 Log[2,4^n/(n t)],3]}, {n,1,12}]//TableForm
```

Código en Mathematica para la Tabla 8.27.

la Tabla 8.27. Sin embargo, para cadenas dispersas cortas, los valores de la Tabla 8.27 son óptimos. Seleccionar, por ejemplo, $P = 1$ en lugar de $P = 2$ para nuestro corto ejemplo (con $n = 3$) resulta en los cuatro tokens:

$$0|3|00|03|30|33 \quad 1|5|01|03|21|22|23|32 \quad 2|5|10|03|20|21|22|23 \quad 3|1|01|03,$$

que requieren un total de 96 bits, más que los 90 bits necesarios para la elección $P = 2$.

En nuestro ejemplo el campo de recuento (`contador`) puede llegar hasta 3, lo que significa que un token de salida, cuyo formato es `prefijo` | `Contador` | `sufijos`, puede comprimirse a un máximo de cuatro píxeles. Una mejor elección puede ser codificar el campo `contador` para que su longitud pueda variar. Incluso el sencillo código unario podría producir buenos resultados, pero una mejor opción puede ser un código de Huffman donde los recuentos pequeños son asignados a códigos cortos. Tales códigos pueden ser construidos basándose en la distribución de valores para el campo `contador` determinados por varias cadenas dispersas de "entrenamiento". Si el campo `contador` está codificado, entonces un token puede tener cualquier tamaño. El compresor, por lo tanto, tiene que generar el token actual en un corto array de bytes y escribir los bytes "completos" en el archivo comprimido, moviendo el último byte, que está normalmente sólo parcialmente lleno, al inicio del array antes de generar el siguiente token. El byte último se escribe en el archivo comprimido con ceros a la derecha.

◇ **Ejercicio 8.16 (sol. en pág. 1110):** ¿Tiene sentido codificar también los campos `prefijo` y `sufijo`?

Uno de los principios de este método es acercar los bits individuales de la cadena dispersa entre sí reorganizando la cadena unidimensional en un array bidimensional. En consecuencia, parece razonable tratar de mejorar el método reorganizando la cadena en un array tridimensional, un cubo, o una caja rectangular, acercando los bits aún más. Si el tamaño de la cadena es 2^{3n}, cada dimensión del array será de tamaño 2^n. Para $n = 7$, la distancia máxima entre los bits de la cadena es de $2^{37} = 2^{21} \approx 2$ millones, mientras que la distancia entre ellos en el cubo tridimensional es sólo $2^7 = 128$, ¡una reducción considerable!

El cubo se puede dividir mediante un *octree*, donde cada uno de los ocho octantes se identifica por un número en el rango 0–7 (un número de 3 bits). Cuando esta partición se lleva a cabo de forma recursiva hasta los píxeles individuales, cada píxel se identifica por un número de n dígitos, donde cada dígito está en el rango 0–7.

◇ **Ejercicio 8.17 (sol. en pág. 1110):** ¿Es posible mejorar aún más el método reorganizando la cadena original en un hipercubo de cuatro dimensiones?

```
procedure RowCol(ind,R1,R2,C1,C2: integer);
case ind of
0: R2:=(R1+R2)÷2; C2:=(C1+C2)÷2;
1: R2:=(R1+R2)÷2; C1:=((C1+C2)÷2)+1;
2: R1:=((R1+R2)÷2)+1; C2:=(C1+C2)÷2;
3: R1:=((R1+R2)÷2)+1; C1:=((C1+C2)÷2)+1;
endcase;
if ind≤n then RowCol(ind+1,R1,R2,C1,C2);
end RowCol;

main program
integer ind, R1, R2, C1, C2;
integer array id[10];
bit array M[2^n,2^n];
ind:=0; R1:=0; R2:=2^n − 1; C1:=0; C2:=2^n − 1;
RowCol(ind, R1, R2, C1, C2);
M[R1,C1]:=1;
end;
```

Figura 8.28: Procedimiento recursivo `RowCol`.

David Salomon tiene un artículo sobre un problema algo más esotérico: la compresión de cadenas dispersas. Si bien ésto no es un algoritmo de propósito general que se ajusta a todo, sí muestra cómo abordar un problema más especializado, pero no fuera de lo común: "Prefix Compression of Sparse Binary Strings" `http://www.acm.org/crossroads/xrds6-3/prefix.html`. (Del *Dr. Dobbs Journal*, marzo de 2000.)

Decodificación: El decodificador comienza leyendo el valor de n y construye de una matriz M de $2^n \times 2^n$ ceros. Suponemos que las filas y columnas de M están numeradas de 0 a $2^n - 1$. A continuación, lee el siguiente token en el archivo comprimido y reconstruye la direcciones (las llamaremos *números id*) de los píxeles (los bits a 1) incluidos en el token. Esta tarea es sencilla y no requiere algoritmos especiales. A continuación, el decodificador tiene que convertir cada id en números de fila y columna. Aquí se describe un algoritmo recursivo para ésto. Una vez que la fila y la columna de un píxel son conocidos, el decodificador establece esa ubicación en la matriz M a 1. La última tarea es la de "desdoblar" M en una única cadena de bits, y eliminar los ceros adicionados artificialmente, si hay alguno, del final.

Nuestro algoritmo recursivo supone que los dígitos individuales del número id son almacenados en un array `id`, con el dígito más significativo almacenado en la posición 0. El algoritmo comienza estableciendo de las dos variables `R1` y `R2` a 0 y $2^n - 1$, respectivamente. Ellas denotan el rango de los números de fila en M. Cada paso del algoritmo examina el siguiente dígito menos significativo del número id en el array `id` y actualiza `R1` o `R2` de tal forma que la distancia entre ellos se reduce a la mitad. Después de n pasos recursivos, convergen a un valor común que es el número de fila del píxel actual. Dos variables más, `C1` y `C2`, son inicializadas y actualizadas de manera similar, para convertirse en el número de columna del píxel actual en M. La Figura 8.28 es un algoritmo en pseudocódigo de un procedimiento recursivo `RowCol` que ejecuta este algoritmo. También se muestra un programa principal (`main program`) que efectúa la inicialización e invoca a `RowCol`. Tenga en cuenta que el símbolo ÷ significa división entera, y que el algoritmo puede ser fácilmente modificado para el caso en que la fila y la columna comiencen en 1.

8.6. Compresión de texto basada en palabras

Todos los métodos de compresión de datos mencionados en este libro operan con alfabetos pequeños. Un alfabeto típico puede estar formado por los dos dígitos binarios, los dieciséis píxeles de 4 bits, los códigos ASCII de 7 bits, o los bytes de 8 bits. En esta sección, se considera la aplicación de métodos conocidos a grandes alfabetos que consisten en *palabras*.

No está claro cómo definir una palabra en los casos donde el flujo de datos de entrada se compone de los píxeles de una imagen, por lo que limitamos nuestra discusión a secuencias de texto. En tal secuencia una palabra se define como una cadena máxima de cualesquier caracteres alfanuméricos (letras y dígitos) u otros caracteres (signos de puntuación y espacios). Denotamos por `A` al alfabeto de todas las palabras alfanuméricas y por `P`, aquel formado por todas las otras palabras. Una consecuencia de esta definición es que en cualquier secuencia de texto —ya sea el código fuente de un programa de ordenador, una obra de ficción, o un menú de restaurante— las palabras de `A` y `P` son estrictamente *alternantes*. Un sencillo ejemplo es la línea de código fuente en lenguaje C

$$␣␣\texttt{for}␣(␣\texttt{short}␣\texttt{i}␣=␣\texttt{0};␣\texttt{i}␣<␣\texttt{npoints};␣\texttt{i++}␣)\bullet$$

donde • indica el carácter fin–de–línea (CR, LF, o ambos). Esta línea puede ser dividida en la secuencia de 15 palabras alternas:

”␣␣” ”for” ”␣(␣” ”short” ”␣” ”i” ”␣=␣” ”0” ”;␣” ”i” ”␣<␣” ”npoints” ”;␣” ”i” ”++␣)•”.

Evidentemente, el tamaño de una palabra del alfabeto puede ser muy grande y con fines prácticos puede considerarse infinito. Ésto implica que un método que requiere el almacenamiento completo del alfabeto en la memoria no puede ser modificado para tratar las palabras como unidades básicas (símbolos) de compresión.

◇ **Ejercicio 8.18 (sol. en pág. 1111):** ¿Cuál es un ejemplo de tal método?

Un punto importante a tener en cuenta es que los *streams* de entrada cortos tienden a tener un pequeño número de palabras distintas, por lo que cuando un método de compresión existente es modificado para operar con palabras, debe tenerse cuidado para asegurar que sigue funcionando de manera eficiente con pequeñas cantidades de datos.

Cualquier método de compresión basado en las frecuencias de los símbolos puede ser modificado para comprimir palabras efectuando una pasada extra, donde la frecuencia de aparición de todas las palabras de la entrada es contada. Sin embargo, esta modificación no es práctica por las siguientes razones:

1. Un método de dos pasos es inherentemente lento.

2. La información reunida por el primer paso tiene que ser incluida en el stream comprimido, debido a que el decodificador la necesita. Ésto disminuye la compresión incluso si esa información se incluye en forma comprimida.

Por consiguiente, tiene más sentido quedarse con las versiones *adaptativas* de los métodos existentes. Tales versiones deben comenzar con una base de datos vacía (diccionario o recuentos de frecuencia) y deben añadir palabras a la misma, a medida que aparecen en el flujo de entrada. Cuando una nueva palabra es introducida, deben emitirse los códigos ASCII puros (formato raw) sin comprimir de los caracteres individuales de la palabra, precedidos por un código de escape. De hecho, es incluso mejor usar algún esquema sencillo de compresión para comprimir los códigos ASCII. Dicha versión también debería tomar ventaja de la naturaleza alternante de las palabras en el stream de entrada.

```
repeat
  introducir una palabra alfanumérica W;
  if W está en el A-árbol then
    emitir código de W;
    incrementar recuento de W;
  else
    emitir un A-escape;
    emitir W (quizás codificado);
    añadir W al A-árbol con un recuento de 1;
    Incrementar el contador de escapes
  endif;
  reorganizar el A-árbol si el necesario;
  introducir una ‘‘otra’’ palabra P;
  if P está en el P-árbol then
  ...
  ... código similar al anterior
  ...
until fin-de-archivo.
```

Figura 8.29: Algoritmo de Huffman adaptativo basado en palabras.

8.6.1. Codificación de Huffman adaptativa basada en palabras

Ésta es una modificación de la codificación de Huffman basada en caracteres (Sección 2.9). Se mantienen dos árboles de Huffman, para los dos alfabetos `A` y `P`, y el algoritmo alterna entre ellos. La Figura 8.29 muestra los pasos principales del algoritmo.

Los principales problemas de este método son los siguientes:

1. En qué formato emitir las nuevas palabras. Una palabra nueva puede escribirse en el stream de salida, siguiendo al código de escape, utilizando los códigos ASCII de sus caracteres. Sin embargo, puesto que una palabra normalmente consta de varios caracteres, una mejor idea es codificarla utilizando el método de Huffman adaptativo basado en caracteres original. Por consiguiente, el algoritmo de Huffman adaptativo basado en palabras “contiene” un algoritmo de Huffman adaptativo basado en caracteres que es invocado de cuando en cuando. Este punto es crítico, puesto que una secuencia de datos de entrada corta normalmente contiene un alto porcentaje de palabras nuevas. La escritura de sus códigos *raw* en el stream de salida puede degradar el rendimiento global de la compresión considerablemente.

2. ¿Qué hacer cuando el codificador se queda sin memoria debido a los grandes árboles de Huffman? Una buena solución es eliminar nodos del árbol (y reacomodar el árbol después de tales eliminaciones, así sigue siendo un árbol de Huffman) en lugar de eliminar el árbol completo. Los mejores nodos a eliminar son aquellos cuyos recuentos son tan bajos que sus códigos de Huffman son más largos de los códigos que se les asignarían si fueran vistos por primera vez. Si hay sólo unos pocos (o ninguno) de esos nodos, entonces los nodos con recuentos de frecuencia bajas deberían suprimirse.

   La experiencia demuestra que la codificación de Huffman adaptativa basada en palabras produce una mejor compresión que la versión basada en caracteres, pero es más lenta, ya que los árboles de Huffman tienden a hacerse grandes, enlenteciendo las operaciones de búsqueda y actualización.

3. La primera palabra de la secuencia de entrada puede ser alfanumérica o de otro tipo. Por consiguiente, el stream comprimido debe comenzar con un flag que indique el tipo de esta palabra.

```
S:=cadena vacía;
repeat
  if actualEsAlfa then introducir palabra alfanumérica W
                  else introducir palabra no alfanumérica W;
  endif;
  if W es una palabra nueva then
  if S no es la cadena vacía then emitir nº de cadena de S; endif;
  emitir un escape seguido por el texto de W;
  S:=cadena vacía;
  else
  if comienzoEsAlfa then buscar S,W en el A-diccionario
                    else buscar S,W en el P-diccionario;
    endif;
    if S,W fue encontrado then S:=S,W
    else
      emitir el número de cadena de S;
      añadir S al A- o al P-diccionario;
      comienzoEsAlfa:=actualEsAlfa;
      S:=W;
    endif;
  endif;
  actualEsAlfa:=not actualEsAlfa;
until fin-de-archivo.
```

Figura 8.30: LZW basado en palabras.

8.6.2. LZW basado en palabras

LZW basado en palabras es una modificación del método LZW basado en caracteres (Sección 3.12). El número de palabras en el stream de entrada no se conoce de antemano y puede ser también muy grande. Como resultado, el diccionario de LZW no se puede inicializarse para todas las palabras posibles, como se hace en el método LZW basado en caracteres original. La idea principal es empezar con un diccionario vacío (en realidad dos diccionarios: un `A`-diccionario y un `P`-diccionario) y utilizar códigos de escape.

> ¡Cuide su fraseología!
>
> —Paul Ford como alcalde Shinn en *The Music Man* (1962)

Cada frase añadida al diccionario consta de dos cadenas, una de `A` y la otra de `P`. Todas las frases donde la primera cadena es de `A` se añaden al `A`-diccionario. Todas aquellas en las que la primera cadena es de `P` se añaden al `P`-diccionario. La ventaja de tener dos diccionarios es que las frases pueden ser numeradas desde 1 en cada diccionario, lo que mantiene los números de frase pequeños. Tenga en cuenta que dos frases diferentes en los dos diccionarios pueden tener el mismo número, ya que el decodificador sabe si la frase siguiente a decodificar proviene del `A`- o del `P`-diccionario. La Figura 8.30 es un algoritmo general, donde la notación "`S,W`" significa que la cadena `W` es agregada a la cadena `S`.

Observe la línea "`emitir un escape seguido por el texto de W;`". En lugar de escribir el código en bruto (formato raw) de `W` en el stream de salida, de nuevo es posible utilizar LZW (basado en caracteres) para codificarlo.

```
prevW:=escape;
repeat
  introducir la siguiente palabra de puntuación WP y emitir su texto;
  introducir la siguiente palabra alfanumérica WA;
  if WA es nueva then
    emitir un escape;
    emitir WA aritméticamente codificada en caracteres;
    añadir AW a la lista de palabras;
    establecer la frecuencia del par (prevW,WA) en 1;
    incrementar la frecuencia del par (prevW,escape) en 1;
  else
    emitir WA aritméticamente codificada;
    incrementar la frecuencia del par (prevW,WA);
  endif;
  prevW:=WA;
until fin-de-archivo.
```

Figura 8.31: Predictor de orden 1 basado en palabras.

8.6.3. Predicción de orden 1 basado en palabras

La gramática inglesa impone correlaciones evidentes entre palabras consecutivas. Es común, por ejemplo, encontrar los pares de palabras "`the boy`" o "`the beauty`" en el texto inglés, pero raramente un par como "`the went`". Ésto refleja las reglas sintácticas básicas que rigen la estructura de una oración, y deben existir normas similares también en otros idiomas. Un algoritmo de compresión que utiliza la predicción de orden 1 puede, por lo tanto, tener mucho éxito cuando aplicado a una secuencia de datos de entrada obedece a reglas sintácticas estrictas. Tal algoritmo [Horspool y Cormack 92] debe mantener una estructura de datos apropiada para las frecuencias de todos los pares de palabras alfanuméricas vistas hasta el momento. Supongamos que el texto "$\ldots P_i A_i P_j$" ha sido introducido recientemente, y que la siguiente palabra es A_j. El algoritmo debe obtener la frecuencia del par (A_i, A_j) de la estructura de datos, calcular su probabilidad, enviarla a un codificador aritmético junto con A_j, y actualizar el recuento de (A_i, A_j). Tenga en cuenta que no hay correlaciones evidentes entre palabras de puntuación consecutivas, pero puede haber algunas correlaciones entre un par (P_i, A_i) o (A_i, P_j). Un ejemplo es una palabra de puntuacion que contiene un punto, que suele indicar el final de una frase, lo que indica que la palabra alfanumérica siguiente probablemente comience con una letra mayúscula, y sea un artículo. La Figura 8.31 es un algoritmo básico en pseudocódigo, que implementa estas ideas; trata de descubrir correlaciones sólo entre palabras alfanuméricas.

Dado que este método utiliza un codificador aritmético para codificar palabras, es natural extenderlo para utilizar el mismo codificador aritmético, aplicado a caracteres individuales, para codificar el texto sin procesar (en formato raw) de nuevas palabras.

> Cuando uso una palabra, significa exactamente lo que yo quiero que signifique —ni más ni menos—.
>
> —Humpty Dumpty

8.7. Compresión de imágenes de texto (textuales)

Todos los métodos descritos hasta ahora asumen que el stream de entrada es un archivo de ordenador o reside en la memoria. La vida, sin embargo, no siempre es tan simple, y a veces los datos a ser comprimidos están formados por un documento impreso que incluye texto, quizás en varias columnas y líneas (horizontales y verticales). El método aquí descrito no puede enfrentarse muy bien con las imágenes, así que asumimos que los documentos de entrada no incluyen ninguna imagen. El documento puede estar en varios idiomas y tipos de letra, y puede contener notas musicales u otras notaciones en vez de texto plano. También puede estar escrito a mano, pero el método descrito aquí funciona mejor con material impreso, ya que la escritura tiene normalmente demasiada variación. Ejemplos de tales datos son: (1) libros raros y documentos históricos originales importantes que se están deteriorando debido a la vejez o al mal manejo, (2) fichas de catálogo de la biblioteca antigua a punto de ser descartadas debido a la automatización, y (3) manuscritos mecanografiados que son de interés para los estudiosos. En muchos de estos casos, es importante conservar *toda* la información sobre el documento, no sólo el texto. Ésto incluye las fuentes originales, notas al margen, y borrones diversos, huellas dactilares y otras manchas.

Antes de cualquier procesamiento por ordenador, el documento tiene, por supuesto, que ser escaneado y convertido en píxeles negros y blancos. Tal documento escaneado se llama *imagen textual*, puesto que es texto descrito mediante píxeles. En la discusión que sigue, esta colección de píxeles se llama *entrada* o *imagen original*. La resolución del escaneado debe ser lo más alta posible, y esto plantea la cuestión de la compresión. Incluso a la baja resolución de 300 dpi, una página de $8{,}5 \times 11$" con márgenes de 1 pulgada por todos los lados, tiene un área de impresión de $6{,}5 \times 9 = 58{,}5$ pulgadas cuadradas, lo que se traduce en $58{,}5 \times 300^2 = 5{,}265$ millones de píxeles. A 600 dpi (resolución media) tal página se convierte en cerca de 21 mil millones de píxeles. La compresión tiene aún más sentido si el documento contiene una gran cantidad de espacio en blanco, ya que en tal caso la mayor parte de los píxeles serán blancos, lo que produce una compresión excelente.

Un enfoque a este problema es el reconocimiento óptico de caracteres (OCR u *optical character recognition*). El software OCR existente utiliza algoritmos sofisticados para reconocer la forma de los caracteres impresos y emite sus códigos ASCII. Si se usa el OCR, el archivo comprimido debe incluir los códigos ASCII, cada uno con un par de coordenadas (x, y) especificando su posición en la página.

◇ **Ejercicio 8.19 (sol. en pág. 1111):** Las coordenadas (x, y) pueden especificar la posición de un carácter con respecto a un origen, posiblemente, en la esquina superior izquierda o en la esquina inferior izquierda de la página. ¿Cuál puede ser una mejor opción para las coordenadas?

El OCR puede ser una buena solución en los casos donde todo el texto está en una fuente (tipo de letra), y no hay necesidad de preservar las manchas, borrones, y la forma precisa de los caracteres mal impresos. Esto tiene sentido para documentos tales como viejos manuales técnicos que no estén lo bastante obsoletos y puedan ser necesarios en el futuro. Sin embargo, si el documento contiene varias fuentes, el software OCR a menudo hace un trabajo deficiente. Asimismo, no puede manejar los acentos, las imágenes, las manchas, las notas musicales, jeroglíficos, o cualquier cosa que no sea texto.

(Debe mencionarse que las últimas versiones de algunos paquetes de OCR, tales como Xerox Techbridge, *permiten* manejar acentos. La aplicación Adobe Acrobat Capture va incluso más lejos. Introduce una página escaneada y la convierte a formato PDF. Internamente representa la página como una colección de pictogramas reconocidos y mapas de bits no reconocidos. Como resultado, es capaz de producir un archivo PDF que, cuando es impreso o visualizado, puede ser casi indistinguible del original.)

Puede usarse la compresión de faxes (Sección 2.13), pero no produce el mejor resultado, ya que está basado en RLE y no presta ninguna atención al texto mismo. Un documento donde las letras se repiten continuamente y otro en el que ningún carácter aparece dos veces puede terminar siendo comprimido en la misma medida.

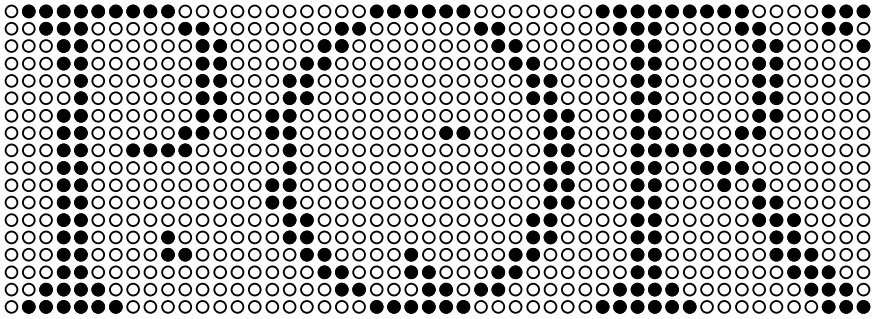


Figura 8.32: Huellas y motas.

El método aquí descrito [Witten et al. 92] es complejo y requiere varios pasos, pero es general, conserva el documento completo, y produce una compresión excelente. Los factores de compresión de 25 no son infrecuentes. El método también puede ser fácilmente modificado para incluir la compresión con pérdida como una opción, en cuyo caso puede producir una compresión con factores de 100 [Witten et al. 94]. Una referencia adicional es [Constantinescu y Arps 97].

El principio del método consiste en separar los píxeles que representan el texto del resto del documento. El texto es luego comprimido con un método que cuenta las frecuencias de los símbolos y los asigna probabilidades, mientras que el resto del documento —que típicamente está compuesto de píxeles aleatorios y puede ser considerado "ruido"— es comprimido por otro método, más adecuado. Este es un resumen del método (el lector puede consultar [Witten et al. 94] para los detalles).

El codificador empieza identificando las líneas de texto. A continuación, escanea cada línea, identificando los límites de los caracteres individuales. El codificador no intenta realmente reconocer los caracteres. Trata cada conjunto conectado de píxeles como un carácter, llamado *marca o huella* (*mark*). A menudo, una huella es un carácter de texto, pero también puede ser parte de un carácter. La letra `i`, por ejemplo, se compone de dos partes inconexas, el pie y el punto, por lo que cada uno se convierte en una huella. Algo parecido a ö, que se convierte en tres huellas. De esta manera, el algoritmo no necesita saber nada acerca de los lenguajes, las fuentes o los acentos utilizados en el texto. El método funciona incluso si el texto se compone de caracteres "exóticos", notas musicales, o jeroglíficos. La Figura 8.32 muestra ejemplos de tres huellas y algunas *motas* (*specks*). Un ser humano puede reconocer fácilmente las huellas como las letras PQR, pero el software podría tener dificultades en ésto, sobre todo porque faltan algunos píxeles.

Las huellas muy pequeñas (menos que el tamaño de un punto) se dejan en la entrada y no son procesadas de nuevo. Cada huella que sobrepasa un cierto tamaño es comparada con una biblioteca de huellas encontradas previamente (llamados símbolos). Si la huella es idéntica a uno de los símbolos, sus píxeles se eliminan de la imagen textual original (la entrada). Si la huella es "lo suficientemente cercana" a uno de los símbolos de la biblioteca, entonces la diferencia entre la huella y el símbolo es ignorada en la entrada (se convierte en parte de lo que se llama el *residuo*) y el resto es eliminado. Si la huella es lo suficientemente diferente de todos los símbolos de la biblioteca, se añade a la biblioteca como un nuevo símbolo y todos sus píxeles son eliminados de la entrada. En cada uno de estos casos el codificador genera el triplete:[6]

$$(\#\ de\ símbolos\ en\ la\ librería, x, y)\,,$$

que es comprimido posteriormente. Las cantidades x e y son las distancias horizontales y verticales (medidas en píxeles) entre la esquina inferior izquierda de la huella y la esquina inferior derecha de su

[6]El símbolo # indica "número de".

predecesor; son los *desplazamientos* (*offsets*). La primera huella en una línea de impresión normalmente tiene un gran desplazamiento x negativo, ya que está localizado a la izquierda de su predecesor.

El caso donde una huella está "suficientemente cercana" a un símbolo de la biblioteca es importante. En la práctica, ésto significa que la huella y el símbolo describen el mismo carácter pero existen pequeñas diferencias entre ellos debido a una impresión pobre o mala, o a la alineación del documento durante el escaneo. Los píxeles que constituyen la diferencia normalmente están, por lo tanto, en forma de un *halo* alrededor de la huella. El residuo se compone, por consiguiente de halos (que son reconocibles o casi reconocibles como caracteres "fantasma") y motas, y manchas que son demasiado pequeñas para ser incluidas en la biblioteca. Considerado como una imagen, el residuo es por lo tanto bastante aleatorio (y, por lo tanto, poco compresible), porque no satisface la condición de que "los vecinos más cercanos de un píxel deben tener el mismo valor que el píxel en sí."

Huella: Una muestra visible realizada en una superficie.

Cuando la entrada completa (el documento escaneado) ha sido escaneada de esta manera, el codificador selecciona todos los símbolos de la biblioteca que emparejaban únicamente con una huella y devuelve sus píxeles a la entrada. Ellos se convierten en parte del residuo. (Este paso es omitido cuando la opción de compresión con pérdida es utilizada.) El codificador entonces se queda con la biblioteca de símbolos, la cadena de tripletes de símbolos, y el residuo. Cada uno de éstos es comprimido por separado.

El decodificador primero descomprime la biblioteca y la lista de tripletes. Ésto es rápido y normalmente produce texto que puede ser mostrado inmediatamente e interpretado por el usuario. Luego, el residuo es descomprimido, añadiendo píxeles a la visualización y llevándola a su forma original. Este proceso sugiere una manera de aplicar la compresión con pérdida: Ignorar únicamente el residuo (y omitir el paso de arriba de devolver los símbolos ya utilizados al residuo). Ésto acelera tanto la compresión como la descompresión, y mejora significativamente el rendimiento de la compresión. Los experimentos muestran que el residuo, aunque compuesto por relativamente pocos píxeles, puede ocupar hasta el 75 % del *stream* comprimido, ya que es aleatorio y, por lo tanto, se comprime mal.

El algoritmo real es muy complejo, ya que tiene que identificar las huellas y decidir si una huella es lo suficientemente cercana a cualquier símbolo de la biblioteca. Aquí, sin embargo, discutiremos sólo la forma en que se comprimen la biblioteca, los tripletes, y el residuo.

El número de símbolos en la biblioteca es codificado en primer lugar, utilizando uno de los códigos de prefijos de la Sección 2.3.1. Cada símbolo se codifica en dos partes: La primera codifica la altura y la profundidad del símbolo (utilizando el mismo código que para el número de símbolos); la segunda codifica los píxeles de los símbolos usando el método de compresión de imágenes binivel basado en el contexto de la Sección 4.18.

Los tripletes son codificados en tres partes. La primera parte es la lista de números de símbolos, que es codificada en una versión modificada de PPM (Sección 2.18). El método original de PPM fue diseñado para un alfabeto cuyo tamaño se conoce de antemano, pero el número de huellas en un documento es desconocido de antemano y puede ser muy grande (especialmente si el documento consta de más de una página). La segunda parte es la lista de x desplazamientos, y la tercera parte, la lista de y desplazamientos. Ellos son codificados mediante la codificación aritmética adaptativa.

Los desplazamientos x e y son las distancias horizontales y verticales (medidas en píxeles) entre la esquina inferior derecha de una huella y la esquina inferior izquierda de la siguiente. En una página impresa con esmero, como la presente, todos los caracteres en una línea, excepto aquellos con trazos

descendentes, están alineados verticalmente en sus partes inferiores, lo que significa que más desplazamientos y serán cero o números pequeños. Si se utiliza una fuente proporcional, entonces el espacio horizontal entre los caracteres de una palabra son también idénticos, lo que resulta en desplazamientos de x que son también pequeños números. El primer carácter de una palabra tiene un desplazamiento x, cuyo tamaño es el espacio entre palabras. En una página impresa cuidadosamente todos los espacios entre palabras en una línea deben ser iguales, aunque aquellos en otras líneas puedan ser diferentes.

Todo esto significa que muchos valores de x aparecerán varias veces en la lista de valores de x, y lo mismo para los valores de y. Lo que es más, si un valor de x de, por ejemplo, 3 se encuentra asociado con el símbolo s, hay una buena probabilidad de que otras apariciones del mismo s tengan un desplazamiento x de 3. Este argumento sugiere el uso de un método de compresión adaptativo para comprimir las listas de desplazamientos x e y.

El método que realmente se utiliza, introduce el siguiente triplete (s, x, y) y comprueba si el símbolo s ha sido visto en el pasado seguido por los mismos desplazamientos x. Si es así, entonces el desplazamiento x es codificado con una probabilidad

$$\frac{\text{el número de veces que este } x \text{ ha sido asociado con este } s}{\text{el número de veces que este } s \text{ ha sido visto}},$$

y el recuento (s, x) se incrementa en 1. Si x no ha sido visto asociado con este s, entonces el algoritmo genera un código de escape, y asigna a x la probabilidad

$$\frac{\text{el número de veces que este } x \text{ ha sido visto}}{\text{el número total de desplazamientos de } x \text{ vistos hasta ahora}},$$

(sin tener en cuenta los símbolos asociados). Si este valor de x en particular nunca ha sido visto en el pasado, el algoritmo produce un segundo código de escape y codifica x con el mismo código de prefijos utilizado para el número de símbolos de la biblioteca (y para la altura y la anchura de un símbolo). El valor de y del triplete se codifica de la misma manera.

La compresión del residuo presenta un problema especial, puesto que visto como una imagen, el residuo es aleatorio y, por tanto incompresible. Sin embargo, visto como texto, el residuo consiste en su mayoría en halos alrededor de caracteres (de hecho, la mayor parte del mismo puede ser legible, o casi legible), lo que sugiere el siguiente enfoque:

El codificador comprime la biblioteca y los tripletes, y los escribe en el stream comprimido, seguido por el residuo comprimido. El decodificador lee la biblioteca y los tripletes, y los utiliza para decodificar el *texto reconstruido*. Sólo entonces lee y descomprime el residuo. En consecuencia, tanto el codificador como el decodificador tiene acceso al texto reconstruido cuando codifican y decodifican el residuo, ¡y este hecho es explotado para comprimir el residuo! Se emplea método de compresión de imágenes binivel basado en el contexto de la Sección 4.18, pero con un giro.

Se utiliza una tabla de tamaño 2^{17} para acumular los recuentos de frecuencia de contextos de 17 bits (en la práctica, se organiza como un árbol binario de búsqueda o una tabla hash). Los píxeles son residuos escaneados fila por fila. El primer paso en la codificación de un píxel en la posición (r, c) del residuo es ir a la misma posición (r, c) *en el texto reconstruido* (que es, por supuesto, una imagen hecha de píxeles) y utilizar el contexto de 13 píxeles que se muestra en la Figura 8.33a para generar un índice de 13 bits. El segundo paso es usar el contexto de cuatro píxeles de la Figura 8.33b *en los píxeles del residuo* añadiendo cuatro bits más para este índice. El índice final de 17 bits se utiliza después para calcular una probabilidad para el píxel actual, basado en el valor del píxel (0 ó 1) y en los recuentos que se encuentran en la tabla para este índice de 17 bits. La cuestión es que la parte de 13 bits del índice se basa en píxeles que *siguen* al píxel actual en el texto reconstruido. Tal contexto es normalmente imposible (se llama *clarividente*) pero se puede utilizar en este caso, porque el texto reconstruido es conocido para el decodificador. Ésta es una variación interesante sobre el tema de la compresión de datos aleatorios.

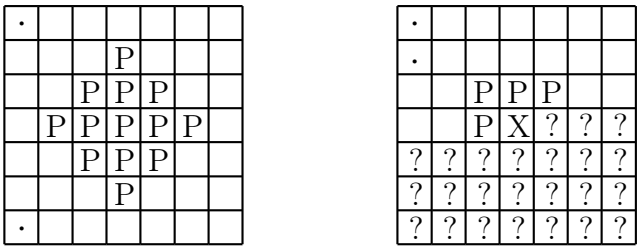


Figura 8.33: (a) Contexto clarividente. (b) Contexto secundario.

Incluso con este método inteligente, el residuo todavía ocupa una gran parte (hasta el 75 %) del stream comprimido. Después de algunos experimentos, los desarrolladores se dieron cuenta de que ¡no es necesario comprimir el residuo en absoluto! En vez de éso, la imagen original (la entrada) puede ser comprimida y descomprimida usando el método anterior, y esto proporciona mejores resultados, a pesar de que la imagen original es menos dispersa que el residuo, ya que (la imagen original) no es tan aleatoria como el residuo.

Este enfoque, de la compresión de la imagen original en lugar del residuo también significa que el residuo no es necesario en absoluto. El codificador no necesita reservar espacio de memoria para ello y crea realmente los píxeles, lo que acelera la codificación.

Por lo tanto, el stream comprimido consta de dos partes: la biblioteca de símbolos y los tripletes (que se decodifican para formar el texto reconstruido), seguidos de la entrada completa en formato comprimido. El decodificador descomprime la primera parte, la muestra de modo que el usuario puede leerla inmediatamente, después la utiliza para descomprimir la segunda parte. Otra ventaja de este método es que a medida que los píxeles de la imagen original son descomprimidos y se vuelven conocidos, pueden mostrarse, reemplazando los píxeles del texto reconstruido y, por lo tanto, mejorando la visualización de la imagen descomprimida por el usuario en *tiempo real*. El decodificador es, por lo tanto, capaz de mostrar una imagen aproximada muy rápidamente, y luego limpiarla fila por fila, mientras el usuario observa, hasta que la imagen final es visualizada.

Como se ha mencionado, los detalles de este método son complejos, ya que implican un proceso de reconocimiento de patrones, además de algoritmos de codificación y decodificación. Aquí se presentan algunas de las complejidades involucradas.

La identificación y extracción de una huella del documento se realiza mediante el escaneo de la entrada de izquierda a derecha y de arriba abajo. El primer píxel no blanco encontrado es el píxel superior izquierdo de una huella. Este píxel se utiliza para rastrear todo el contorno de la huella, un proceso complejo que implica un algoritmo similar a los utilizados en los gráficos por ordenador para rellenar un área. El punto principal es que la huella puede tener un contorno interior, así como uno exterior (pensemos en las letras `O`, `P`, `Q` y `Φ`), y puede haber otras huellas anidadas en su interior.

◇ **Ejercicio 8.20 (sol. en pág. 1111):** Parece que ninguna letra del alfabeto latino consta de dos partes anidadas. ¿Cuáles son ejemplos de huellas que contengan huellas de otras, más pequeñas anidadas dentro ellas?

El rastreo de los límites de una huella implica también la cuestión de la conectividad. ¿Cuándo se consideran los píxeles conectados? La Figura 8.34 ilustra los conceptos de 4- y 8-conectividad y deja claro que debe utilizarse el último método, porque el primero puede perder segmentos de letras que normalmente se consideran conectados.

La comparación de una huella con los símbolos de la biblioteca es el siguiente problema complejo. ¿Cuando una huela es considerada "suficientemente cercana" a un símbolo de la biblioteca? No

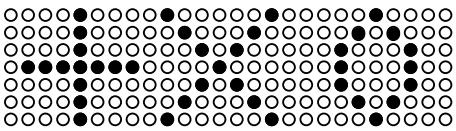


Figura 8.34: 4- y 8-Conectividad.

es suficiente simplemente ignorar las pequeñas áreas de píxeles donde los dos difieren, ya que ésto puede conducir a la identificación, por ejemplo, de una `e` como una `c`. Se necesita un método más sofisticado, basado técnicas de reconocimiento de patrones. También es importante acelerar el proceso de comparación de una huella con un símbolo, ya que una huella tiene que ser comparada con todos los símbolos de la biblioteca antes de considerarse uno nuevo. Es necesario un algoritmo que averigüe rápidamente si la huella y el símbolo son demasiado diferentes. Este algoritmo puede utilizar pistas como las grandes diferencias en la altura y en la anchura de ambos, o en sus áreas totales o perímetros, o en el número de píxeles negros de cada uno.

Cuando se determina que una huella es suficientemente diferente de todos los símbolos existentes en la biblioteca, se añade a la biblioteca y se convierte en un símbolo. Pueden ser encontradas otras huellas en el futuro, que estén lo suficientemente cercanas a este símbolo y terminar siendo asociadas con ella. Ellas deben ser "recordadas" por el codificador. Por consiguiente, el codificador mantiene una lista adjunta a cada símbolo de la biblioteca, que contiene las huellas asociadas con el símbolo. Cuando la entrada completa ha sido escaneada, la biblioteca contiene la primera versión de cada símbolo, junto con una lista de huellas que son similares al mismo. Para lograr una mejor compresión, cada símbolo es reemplazado ahora por un promedio de todas las huellas en su lista. Un píxel en este promedio se establece en negro si es negro en más de la mitad de las huellas en la lista. El promedio de símbolos no sólo se traduce en una mejor compresión, sino también mejora el aspecto del texto reconstruido, haciendo posible el uso de la compresión con pérdida más a menudo. En principio, cualquier cambio en un símbolo debe dar lugar a modificaciones en el resto, pero ya sabemos que en la práctica el residuo no tiene que ser mantenido en absoluto.

Todas estas complejidades convierten las imágenes textuales en un ejemplo complejo e interesante de un método de compresión de datos de propósito especial y muestra cuánto se puede ganar de un enfoque sistemático en el que cada idea es implementada y experimentada antes de ser rechazada, aceptada o mejorada.

8.8. Codificación dinámica de Markov

La codificación dinámica de Markov es un método de compresión estadístico de dos etapas adaptativo debido a G. V. Cormack y R. N. Horspool [Cormack y Horspool 87] (véase también [Yu 96] para una implementación). La etapa 1 emplea una máquina de estado finito para estimar la probabilidad de el siguiente símbolo. La etapa 2 es un codificador aritmético que lleva a cabo la compresión real. Recordemos que el método PPM (Sección 2.18) funciona de manera similar. Los autómatas finitos (también llamados máquinas de estados finitos) se discuten en muchos textos.

> Tres siglos después de Hobbes, los autómatas se multiplican con una agilidad que ninguna visión formada en el siglo XVII podría haber predicho.
>
> —George Dyson, *Darwin entre las máquinas* (1997)

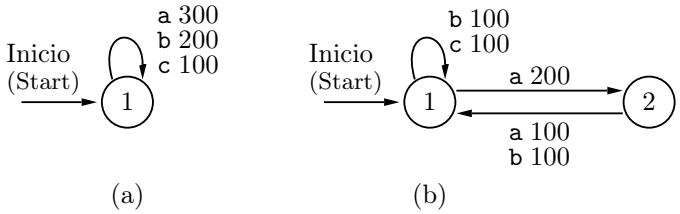


Figura 8.35: Modelos de estados finitos para la compresión de datos.

Una máquina de estados finitos puede ser utilizado en la compresión de datos como un modelo, para calcular las probabilidades de los símbolos de entrada. La Figura 8.35a muestra el modelo más simple, una máquina de un solo estado. El alfabeto se compone de los tres símbolos `a`, `b`, y `c`. Supongamos que el flujo de entrada es la cadena de 600 símbolos `aaabbcaaabbc`... Cada vez que un símbolo es introducido, la máquina emite su probabilidad estimada, actualiza su recuento, y permanece en su (único) estado. Cada una de las tres probabilidades se fija inicialmente en $1/3$ y es actualizado muy rápidamente a su valor correcto ($300/600$ para `a`, $200/600$ para `b`, y $100/600$ para `c`) debido a la regularidad de esta entrada en particular. Asumiendo que la máquina utiliza siempre las probabilidades correctas, la entropía (número de bits por símbolo de entrada) de este primer ejemplo es:

$$-\frac{300}{600}\log_2\left(\frac{300}{600}\right)-\frac{200}{600}\log_2\left(\frac{200}{600}\right)-\frac{100}{600}\log_2\left(\frac{100}{600}\right)\approx 1{,}46.$$

Estado: modo o condición de ser (un estado de preparación)

La Figura 8.35b ilustra un modelo de dos estados. Cuando una `a` es un introducida en el estado 1, éste actualiza su recuento, y la máquina cambia al estado 2. Cuando se introduce una `a` o una `b` en el estado 2, éste actualiza sus recuentos y vuelve al estado 1 (este modelo se detendrá prematuramente si en el estado 2 se introduce una `c`). Cada sexteto de símbolos leídos de la secuencia de entrada cambia entre los dos estados cuatro veces de la siguiente manera:

$$a \xrightarrow{2} a \xrightarrow{1} a \xrightarrow{2} b \xrightarrow{1} b \xrightarrow{1} c \xrightarrow{1} .$$

El estado 1 acumula 200 recuentos de `a`, 100 recuentos de `b`, y 100 recuentos de `c`. El estado 2 acumula 100 recuentos de `a` y 100 recuentos de `b`. En consecuencia, el estado 1 maneja 400 de los 600 símbolos, y el estado 2, los restantes 200. La probabilidad de la máquina de estar en el estado 1 es, por lo tanto, $400/600 = 4/6$, y la de estar en el estado 2 es $2/6$.

La entropía de este modelo se calcula por separado para cada estado, y la entropía total es la suma de las entropías individuales de los dos estados, ponderada por las probabilidades de los estados. El estado 1 tiene tres elementos, "`a`", "`b`" y "`c`", que salen del mismo con probabilidades $2/6$, $1/6$ y $1/6$, respectivamente. El estado 2 tiene dos elementos, "`a`" y "`b`", que salen de él, cada uno con una probabilidad de $1/6$. Por lo tanto, las entropías de los dos estados son:

$$\begin{aligned}-\frac{100}{600}\log_2\left(\frac{100}{600}\right)-\frac{200}{600}\log_2\left(\frac{200}{600}\right)-\frac{100}{600}\log_2\left(\frac{100}{600}\right)&\approx 1{,}3899 \quad \text{(estado 1)}\,,\\ -\frac{100}{600}\log_2\left(\frac{100}{600}\right)-\frac{100}{600}\log_2\left(\frac{100}{600}\right)&\approx 0{,}8616 \quad \text{(estado 2)}\,.\end{aligned}$$

Por tanto, la entropía total es de $1{,}3899 \times 4/6 + 0{,}8616 \times 2/6 = 1{,}21$.

Asumiendo que el codificador aritmético funciona en o cercano a la entropía, este modelo de dos estados codifica un símbolo en 1,21 bits, en comparación con los 1,46 bits/símbolo del modelo previo, de un estado. Así es cómo, una máquina de estados finitos con los estados correctos puede usarse para producir buenas estimaciones de probabilidad (buenas predicciones) para la compresión de datos.

La pregunta lógica en este punto es, dada una secuencia de entrada (*stream*) en particular, ¿cómo encontramos la máquina de estados finitos particular, que caracterizará con la entropía más pequeña a esa cadena? Un método de fuerza simple y bruta es tratar todas las posibles máquinas de estados finitos, pasar el stream de entrada a través de cada uno de ellos, y medir los resultados. Este enfoque es poco práctico, ya que existen n^{na} máquinas de n estados para un alfabeto de tamaño a. Incluso para los alfabetos más pequeños, con dos símbolos, este número crece exponencialmente: una máquina de 1 estado, 16 máquinas de 2 estados, 729 máquinas de 3 estados, y así sucesivamente.

Es evidente que se necesita un enfoque inteligente, donde el algoritmo pueda comenzar con una simple máquina de un solo estado, y adaptarla a los datos de entrada en particular mediante la adición de estados a medida que aumentan su longitud, basándose en los recuentos acumulados en cualquier paso. Éste es el enfoque adoptado por el algoritmo DMC.

8.8.1. El algoritmo DMC

Este algoritmo fue desarrollado originalmente para datos binarios (i.e, un alfabeto de dos símbolos). Los ejemplos más comunes de datos binarios son los archivos en código máquina (ejecutable), imágenes (tanto monocromáticas como a color), y el sonido. Cada estado de la máquina de estados finitos DMC (o modelo DMC; en esta sección las palabras "máquina" y "modelo" se usan indistintamente) lee un bit de la cadena (*stream*) de entrada, le asigna una probabilidad basada en los recuentos del pasado, y cambia a uno de los otros dos estados en función de si el bit de entrada era 1 ó 0. El algoritmo comienza con una máquina pequeña (quizás tan simple como un único estado) y le agrega estados basándose en la entrada. En consecuencia, es adaptativo. Tan pronto como un nuevo estado es añadido a la máquina, ésta empieza a contar los bits y los usa para calcular las probabilidades para 0 y 1. Incluso en esta simple forma, la máquina puede crecer mucho y llenar rápidamente toda la memoria disponible. Una de las ventajas de tratar con datos binarios es que el codificador aritmético puede ser muy eficiente si tiene que lidiar con sólo dos símbolos.

En su forma original, el algoritmo de DMC no comprime muy bien el texto. Recuérdese que la compresión se lleva a cabo mediante la reducción de la redundancia, y que la redundancia de un archivo de texto se presenta en los caracteres del texto, no en los bits individuales. Es posible extender DMC para manejar los 128 caracteres ASCII implementando una máquina de estados finitos con estados más complejos. Un estado en tal máquina debe introducir un carácter ASCII, asignarle una probabilidad basada en el recuento de los caracteres del pasado, y cambiar a uno de 128 estados según el carácter que fue introducido. Tal máquina crecerá consumiendo aún más espacio en memoria que la versión binaria.

El algoritmo DMC consta de dos partes: la primera se ocupa de calcular probabilidades, y la segunda se refiere a la adición de nuevos estados a la máquina existente. La primera parte calcula las probabilidades contando, para cada estado S, cuántos ceros y unos han sido introducidos en ese estado. Supongamos que en el pasado la máquina estuvo en el estado S varias veces, y se introdujo un 0 s_0 veces y un 1 s_1 veces mientras permanecía en este estado (i.e., partió del estado S s_0 veces con la salida 0, y s_1 veces con la salida 1; Figura 8.36a). La forma más sencilla de asignar probabilidades a los dos bits es mediante la siguiente definición:

La probabilidad de que un 0 sea introducido, mientras permanece en el estado S es $\frac{s_0}{s_0+s_1}$,

La probabilidad de que un 1 sea introducido, mientras permanece en el estado S es $\frac{s_1}{s_0+s_1}$.

Pero esto, por supuesto, plantea el *problema de la probabilidad cero*, puesto que cualquiera de s_0 o s_1 puede ser cero. La solución adoptada por DMC es asignar probabilidades que son siempre distintas

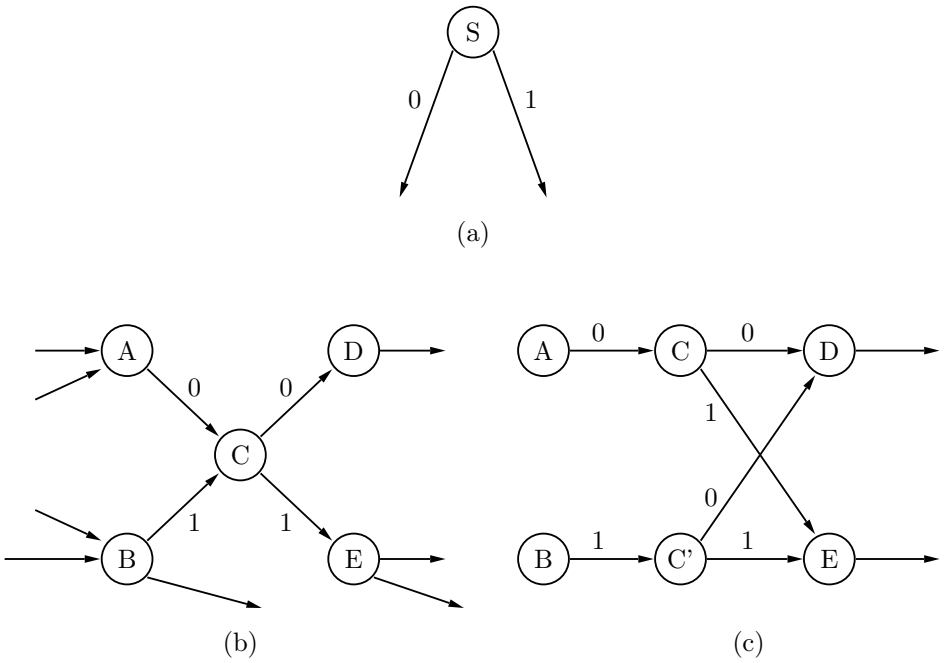


Figura 8.36: Los principios de DMC.

de cero y que dependerán de un parámetro entero positivo c. Las definiciones son las siguientes:

La probabilidad de que un 0 sea introducido, mientras permanece en el estado S es $\frac{s_0+c}{s_0+s_1+2c}$,

La probabilidad de que un 1 sea introducido, mientras permanece en el estado S es $\frac{s_1+c}{s_0+s_1+2c}$.

La asignación de valores pequeños a c implica que los pequeños valores de s_0 y s_1 afectarán a las probabilidades significativamente. Ésto se hace cuando el usuario siente que las distribuciones de los dos bits en los datos pueden ser "aprendidas" rápidamente por el modelo. Si los datos son tales que necesitan más tiempo para adaptarse a las distribuciones de bits correctas, valores más grandes de c pueden llevar a una mejor compresión. La experiencia demuestra que para streams de entrada muy grandes, el valor preciso de c no marca mucha diferencia.

◇ **Ejercicio 8.21 (sol. en pág. 1111):** ¿Por qué aparece c en el numerador, pero $2c$ en el denominador de las dos probabilidades?

La segunda parte del algoritmo DMC se refiere a cómo añadir un nuevo estado a la máquina. Considere la posibilidad de los cinco estados mostrados en la Figura 8.36b, que puede ser parte de un modelo DMC de estados finitos grande. Cuando un 0 es introducido, mientras está en el estado A, o cuando es introducido un 1 mientras está en el estado B, la máquina pasa al estado C. El siguiente bit de entrada cambia a D o E. Cuando cambia a D, e.g., se pierde parte de la información, ya que la máquina no "recuerda" si había llegado hasta allí desde A o desde B. Esta información puede ser importante si los bits de entrada están correlacionados (i.e., si las probabilidades de ciertos patrones de bits son muy diferentes de las de los otros patrones). Si la máquina está actualmente en el estado A, llegará a D si se introduce 00. Si se encuentra en el estado B, llegará a D si se introduce 10. Si las probabilidades de los patrones de entrada 00 y 10 son muy diferentes, el modelo puede calcular las mejores probabilidades (puede producir mejores predicciones) si conocía si llegó a D desde A o desde B.

La idea central de DMC es comparar los recuentos de las transiciones $A \to C$ y $B \to C$, y si son significativamente diferentes, crear una copia del estado C, llamarlo C', y colocar la copia

de forma que $A \to C \to (D, E)$, pero $B \to C' \to (D, E)$ (Figura 8.36c). Este proceso de copia se denomina *clonación*. La máquina se vuelve más compleja, pero ahora puede mantener mejores recuentos (recuentos que dependen de las correlaciones específicas entre A y D, A y E, B y D, y B y E) y, como resultado, calcular mejores probabilidades. Incluso la adición de un estado puede mejorar la estimación de probabilidad significativamente, ya que puede "sacar a la luz" correlaciones entre un estado anterior A y un estado posterior D. En general, mientras más estados se agregan mediante clonación, más fácil es para el modelo "aprender" acerca de las correlaciones (incluso las de largo alcance) entre los bits de entrada.

Una vez que el nuevo estado C' es creado, los recuentos originales del estado C deben dividirse entre C y C'. Idealmente, deben ser divididos en proporción a los recuentos de las transiciones $A \to C \to (D, E)$ y $B \to C \to (D, E)$, pero estos recuentos no están disponibles (de hecho, la clonación se lleva a cabo con precisión con el fin de tener estos recuentos en el futuro). Lo siguiente mejor es dividir los nuevos recuentos en proporción a los recuentos de las transiciones $A \to C$ y $B \to C$.

Un punto interesante es que la clonación innecesaria no hace mucho daño. Aumenta el tamaño de la máquina de estados finitos en un estado, pero las probabilidades calculadas no se empeoran. (Dado que la máquina tiene ahora un estado más, cada estado será visitado con menos frecuencia, lo que dará lugar a menores recuentos y, por lo tanto, se amplifican las pequeñas fluctuaciones en la distribución de los bits de entrada, pero ésto es una desventaja menor.)

Todo esto sugiere que la clonación se realizará lo más pronto posible, por lo que precisamos para decidir de una(s) regla(s) exacta(s) para la clonación. Un vistazo a la Figura 8.36b muestra que la clonación debe ser efectuada sólo cuando ambas transiciones $A \to C$ y $B \to C$ tienen altos recuentos. Si ambas tienen bajos recuentos, "no hay suficientes estadísticas" para justificar la clonación. Si A tiene un recuento alto, y B tiene un recuento bajo, no se gana mucho con la clonación de C, ya que B no es muy activa. Ésto sugiere que la clonación debe hacerse cuando tanto A como B tengan recuentos elevados y uno de ellos tenga un recuento mucho más alto que el otro. Por lo tanto, el algoritmo DMC utiliza dos parámetros $C1$ y $C2$, y la siguiente regla:

> Si el estado actual es A y el siguiente estado es C, entonces C es un candidato para los clonación, y debe ser clonado si el recuento para la transición $A \to C$ es mayor que $C1$ **y** el recuento total de todas las otras transiciones $X \to C$ es mayor que $C2$ (X representa todos los estados semilla C, excepto el estado actual A).

La elección de los valores de $C1$ y $C2$ es crítico. Los valores pequeños producen una clonación rápida de los estados. Ésto implica una mejor compresión, ya que el modelo de "aprende" las correlaciones en los datos más rápido, pero también utiliza más memoria, lo que aumenta la probabilidad de encontrarse sin memoria durante la ejecución mientras aún quedan muchos datos para ser comprimidos. Los valores grandes tienen el efecto contrario. Por tanto, cualquier aplicación práctica debe permitir que el usuario especifique los valores de los dos parámetros. También tiene sentido empezar con valores pequeños y aumentarlos gradualmente a medida que la compresión avanza. Ésto permite que el modelo "aprenda" rápido al principio, y también retrasa el momento en que la ejecución del algoritmo se queda sin memoria.

◇ **Ejercicio 8.22 (sol. en pág. 1111):** La Figura 8.37 muestra parte de un modelo DMC de estados finitos. El estado A cambia a D cuando se introduce un 1, por lo que D es un candidato para la clonación cuando A es el estado actual. Suponiendo que el algoritmo ha decidido clonar D, muéstrense los estados tras la clonación.

La Figura 8.38 es un ejemplo sencillo que muestra seis pasos de adición de estados a un modelo DMC hipotético. El modelo comienza con el estado único 0 cuyas salidas vuelven de nuevo y se convierten en sus entradas (son *reflexivas*). En 8.38b un nuevo estado, el estado 1, es añadido a la salida 1 del estado 0. Usamos la notación $0, 1 \to 1$ (se lee: desde el estado 0, con la salida 1, se va nuevo estado 1) para indicar esta operación. En 8.38c la operación $0, 0 \to 2$ añade un nuevo estado 2.

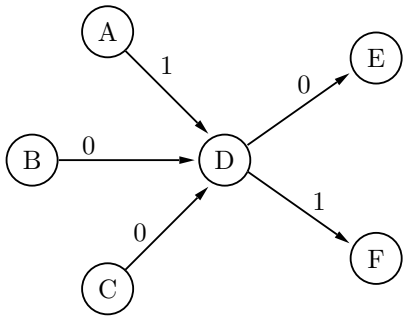


Figura 8.37: El estado D es un candidato.

En 8.38d,e,f, los estados 3, 4 y 5 se añaden mediante las operaciones 1, 1 → 3, 2, 1 → 4; y 0, 0 → 5. La Figura 8.38f, por ejemplo, fue construida añadiendo estado 5 a la salida 0 del estado 0. Las dos salidas del estado 5 se determinan examinando la salida 0 del estado 0. Puesto que esta salida iba al estado 2, la nueva salida 0 del estado 5 va al estado 2. Además, dado que esta salida iba al estado 2, la nueva salida 1 del estado 5 se convierte en un copia de la salida 1 del estado 2, por lo que lleva al estado 4.

◇ **Ejercicio 8.23 (sol. en pág. 1111):** Dibújese el modelo DMC tras la operación 1, 1 → 6.

8.8.2. Inicio y parada DMC

Cuando comienza el algoritmo de DMC, sólo necesita tener un estado que vuelva a sí mismo cuando se introduzca ya sea 0, ya sea 1, como se muestra en la Figura 8.39a. Este estado es clonado muchas veces y puede crecer muy rápido para convertirse en una máquina de estados finitos compleja con muchos miles de estados. Esta forma de iniciar el algoritmo DMC funciona bien para entradas binarias. Sin embargo, si la entrada consiste en símbolos no binarios, una máquina inicial apropiada, una que tome ventaja de las posibles correlaciones entre los bits individuales de un símbolo, puede proporcionar una compresión mucho mejor. El árbol de la Figura 8.39b es una buena opción para símbolos de 4 bits, debido a que cada nivel corresponde a uno de los cuatro bits. Si existe, por ejemplo, una gran probabilidad de que un símbolo $01xx$ tenga la forma $011x$ (i.e., un 01 al inicio de un símbolo seguido por otro 1), el modelo lo descubrirá muy rápidamente y clonará el estado marcado en la figura. Un árbol binario completo similar, pero con 255 estados en lugar de 15, puede ser apropiado como modelo inicial en aquellos casos donde los datos consisten en símbolos de 8 bits. Máquinas iniciales más complejas pueden incluso tomar ventaja de las correlaciones entre el último bit de un símbolo de entrada y el primer bit del símbolo siguiente. Uno de tales modelos, una malla (*braid*) diseñada para símbolos de entrada de 3 bits, se muestra en la Figura 8.39c.

Cualquier aplicación práctica de DMC debería abordar la cuestión del uso de la memoria. El número de estados puede crecer muy rápidamente y llenar cualquier cantidad de disponible memoria. La solución más sencilla es continuar, una vez que la memoria está llena, sin clonación. Una mejor solución es descartar el modelo existente y empezar de nuevo. Ésto tiene la ventaja de que los nuevos estados a ser clonados se basarán en nuevas correlaciones descubiertas en los datos, y las viejas correlaciones serán olvidadas. Una solución incluso mejor es mantener siempre los k símbolos de entrada más recientes en una cola circular y usarlos para construir un pequeño modelo inicial cuando el viejo es desechado. Cuando el algoritmo se reanuda, este pequeño modelo inicial tomará ventaja de las correlaciones descubiertas recientemente, por lo que el algoritmo de no tendrá que "volver a aprender" los datos desde cero. Este método minimiza la pérdida de compresión resultante de descartar el viejo modelo.

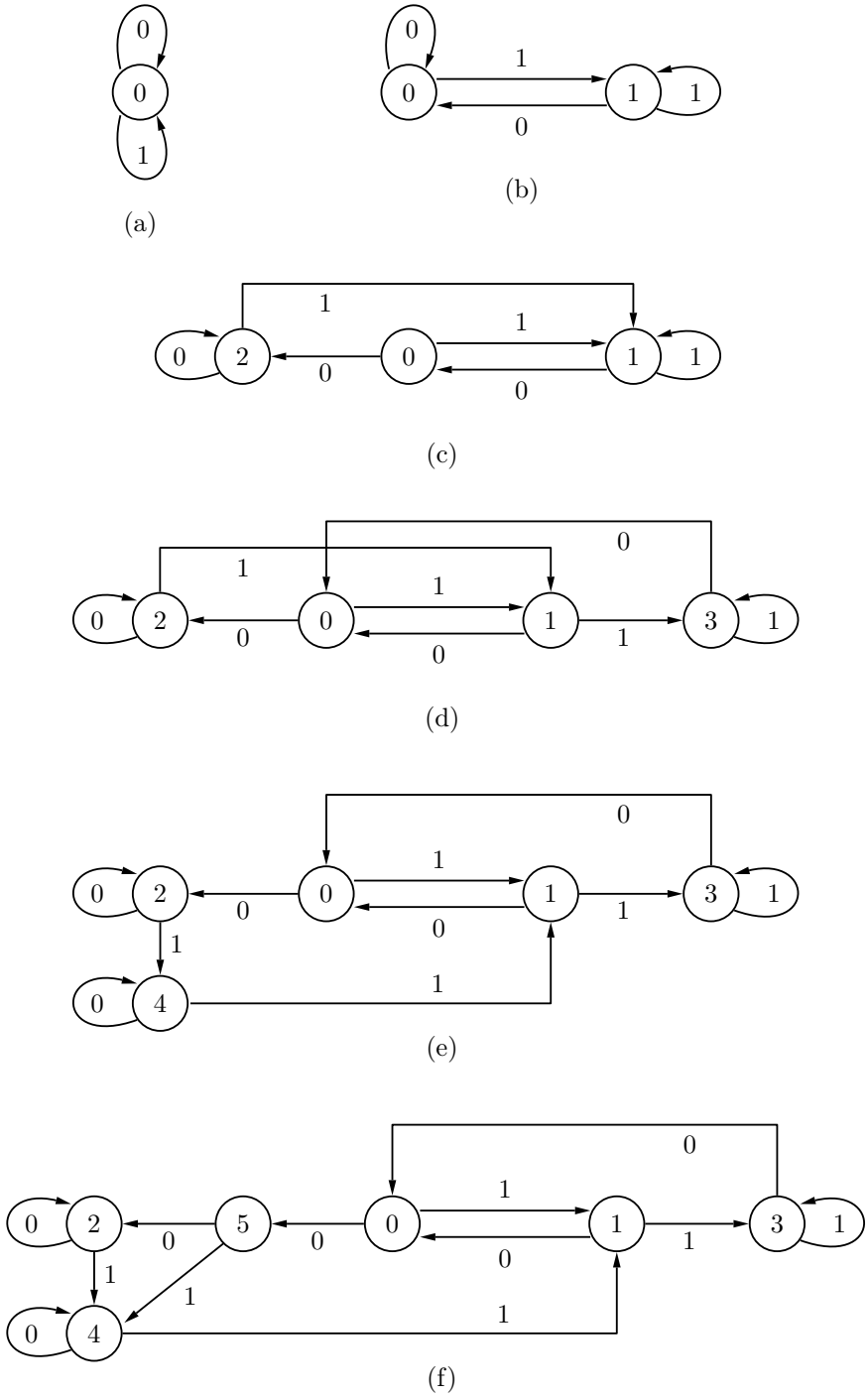


Figura 8.38: Los primeros seis estados.

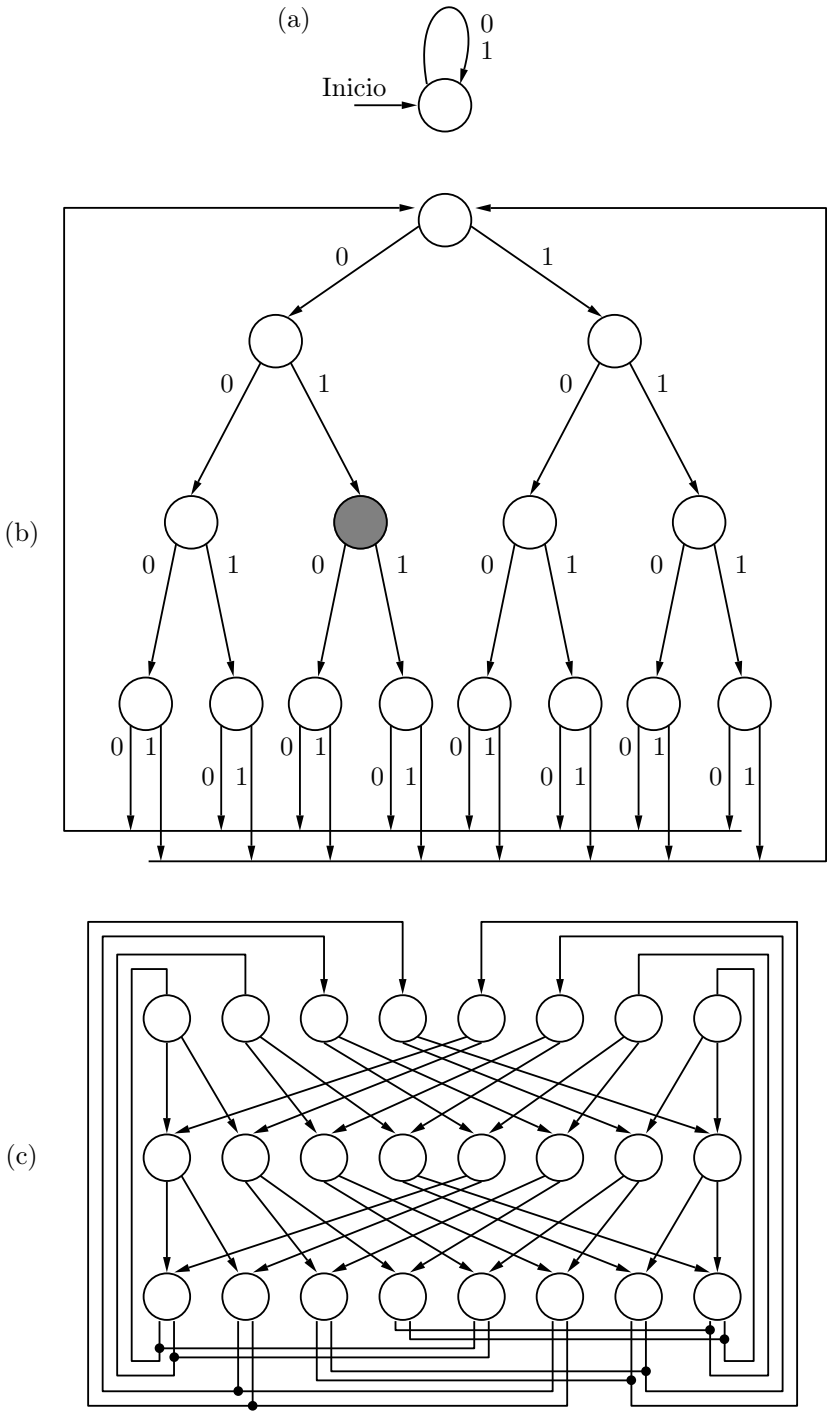


Figura 8.39: Modelos DMC iniciales.

◇ **Ejercicio 8.24 (sol. en pág. 1111):** ¿Cómo afecta la pérdida de compresión según el valor de k?

El principio fundamental de DMC, la regla de la clonación, se basa en la intuición, no en la teoría. En consecuencia, la principal justificación de DMC es ¡que funciona! Produce una excelente compresión, comparable a la alcanzada por PPM, si bien es más rápido.

8.9. Curva de compresión FHM

El nombre FHM es un acrónimo de Fibonacci, Huffman, y Markov. Este método es una modificación del método de *codificación de cadena multianillo* (véase [Freeman 61] y [Wong y Koplowitz 92]), y utiliza números de Fibonacci [Vorobev 83] para construir cuadrados de tamaños específicos alrededor del punto actual, de tal manera que el número total de alternativas en cualquier punto en la compresión es exactamente 256.

El método está diseñado para comprimir curvas, y es especialmente adecuado para la compresión de firmas digitales. Dicha firma se obtiene, en un punto de venta o durante la entrega de paquetes, con un lápiz sobre una tableta gráfica especial que interrumpe la curva de la firma en una gran secuencia de puntos. Para cada punto $\mathbf{P}_i$ tal tableta registra sus coordenadas, el tiempo que tardó el usuario en mover el lápiz de $\mathbf{P}_i$ a $\mathbf{P}_{i-1}$, y el ángulo y la presión del lápiz en $\mathbf{P}_i$. Esta información se mantiene entonces en un archivo y puede ser utilizada más tarde por un sofisticado algoritmo para comparar con firmas futuras. Puesto que la secuencia puede ser grande (cinco ítems por punto y cientos de puntos), debe ser archivada en formato comprimido.

En la presente discusión, se considera sólo la compresión de las coordenadas. La curva resultante comprimida es muy similar a la curva original, pero cualquier instante, ángulo e información de la presión se pierde. El método se basa en las dos ideas siguientes:

1. Una línea recta está totalmente definida por sus dos extremos, por lo que cualquier punto interior puede ser ignorado. En regiones en las que la curva que está siendo comprimida tiene una pequeña curvatura (es decir, que está cercana a una línea recta) ciertos puntos que han sido digitalizados por la tableta pueden ser ignorados sin afectar a la forma de la curva.

2. En cualquier instante del proceso de compresión, la curva se coloca dentro de una cuadrícula centrada en el *punto de anclaje* actual A (la red tiene el mismo tamaño de coordenadas usado por la tableta). El siguiente punto de anclaje B se selecciona de tal manera que: (1) el segmento recto AB es tan largo como sea posible, (2) la curva en la región AB se asemeja a una línea recta, y (3) B puede ser elegido de entre los 256 puntos de la cuadrícula. Todos los puntos que han sido digitalizados originalmente por la tableta entre A y B son eliminados, y B se convierte en el punto de anclaje actual. Nótese que, en general B no es uno de los puntos digitalizados originalmente. En consecuencia, el método reemplaza el conjunto original de puntos con el conjunto de puntos de anclaje, y sustituye la curva por el conjunto de segmentos rectos que unen los puntos de anclaje. En las regiones de gran curvatura los puntos de anclaje están muy juntos. El primer punto de anclaje es el primer punto digitalizado por la tableta.

La Figura 8.40a muestra una red S_1 de 5×5 centrada en el punto de anclaje actual X. Existen 16 puntos en la circunferencia de S_1, y ocho de ellos están marcados con círculos. Seleccionamos el primer punto Y que pertenece a la curva, pero que se encuentra fuera de la red, y conectamos los puntos X e Y con un segmento recto (la flecha en la figura). Seleccionamos los dos puntos marcados más próximos a la flecha y construimos el triángulo que se muestra en trazos. No hay puntos en el interior de este triángulo, lo que significa que la parte de la curva en el interior de S_1 está cercana a una línea recta. Antes de seleccionar el siguiente punto de anclaje, probamos la siguiente red más grande, S_2 (Figura 8.40b). Esta red cuenta con 32 puntos en su circunferencia, y 16 de ellos están marcados. Al igual que antes, seleccionamos el primer punto Z localizado en la curva, pero fuera de S_2, y conectamos los

puntos X y Z con un segmento recto (la flecha en la figura). Seleccionamos los dos puntos marcados más próximos a la flecha y construimos el triángulo (en trazos). Esta vez existe un punto (Y) en la curva que está fuera del triángulo. Ésto significa que la parte de la curva en el interior de S_2 no está suficientemente cercana a una línea recta. Por lo tanto, volvemos a S_1 y seleccionamos el punto P (el punto marcado más cercano a la flecha) como siguiente punto de anclaje. Por consiguiente, la distancia entre el siguiente anclaje y la curva verdadera es menos de una cuadrícula unidad. El punto P es codificado, las rejillas son centradas en P, y el proceso continúa.

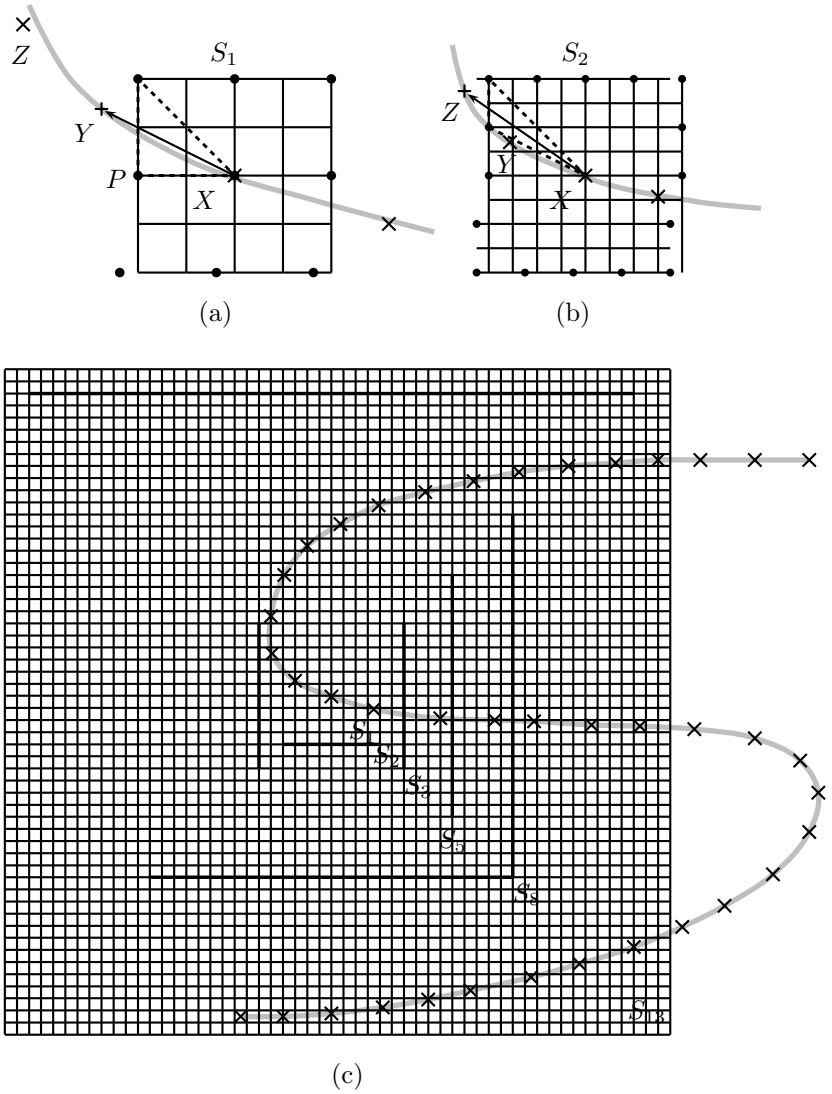


Figura 8.40: Compresión FHM de una curva.

La Figura 8.40c muestra las seis redes utilizadas por este método. Éstas se denotan por S_1, S_2, S_3, S_5, S_8, y S_{13}, con 8, 16, 24, 40, 64, y 104 puntos marcados, respectivamente. En consecuencia, el número total de puntos marcados es 256.

◇ **Ejercicio 8.25 (sol. en pág. 1112):** ¿De dónde proceden los números de Fibonacci entran en

juego?

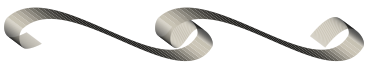
Dado que el siguiente punto de anclaje P puede ser uno de 256 puntos, se puede escribir en el stream comprimido en ocho bits. Tanto el codificador como el decodificador deben tener una tabla o una regla que les diga dónde se encuentran cada uno de los 256 puntos con respecto al punto de anclaje actual. La experiencia demuestra que algunos de los 256 puntos se seleccionan con más frecuencia que otros, lo que sugiere un código de Huffman para codificarlos. Los 104 puntos del borde de S_{13}, por ejemplo, se seleccionan sólo cuando la curva tiene una región larga (26 unidades de coordenadas o más) donde la curva está cercana a una línea recta. Por tanto, estos puntos deberían ser asignados a códigos de Huffman largos. Una forma práctica de determinar la frecuencia de ocurrencia de cada uno de los 256 puntos es *entrenar* el algoritmo con muchas firmas reales. Una vez la tabla de 256 códigos de Huffman ha sido determinada, es construida tanto en el codificador como en el decodificador.

Ahora para Markov. Una cadena de Markov (o un modelo de Markov) es una secuencia de valores donde cada valor depende de uno de sus predecesores, no necesariamente del predecesor inmediato, pero no en cualquier otro valor (en un modelo de Markov de orden k, un valor depende de k de sus vecinos del pasado). El trabajo con firmas reales muestra que el número de reversiones de dirección (o casi reversiones) es pequeño en comparación con el tamaño de la firma. Ésto implica que si un segmento L_i entre dos puntos de anclaje va en una dirección determinada, hay una buena probabilidad de que el siguiente segmento L_{i+1} apuntará en una dirección algo distinta, pero no muy diferente. En consecuencia, el segmento L_{i+1} depende del segmento L_i pero no de L_{i-1} o segmentos anteriores. Por tanto, la secuencia de segmentos L_i es una cadena de Markov, un hecho que sugiere el uso de una tabla de códigos de Huffman diferente para cada segmento. Un segmento va de un punto de anclaje al siguiente. Puesto que hay 256 puntos de anclaje diferentes, un segmento puede apuntar en una de 256 direcciones. Denotamos las direcciones por $D_0, D_1, \ldots, D_{255}$. Ahora necesitamos construir 256 tablas con 256 códigos de Huffman cada una; un total de ¡$2^{16} = 65\,536$ códigos!

Para calcular la tabla de códigos de Huffman para una dirección D_i tenemos que analizar muchas firmas y contar cuántas veces un segmento que apunta en la dirección D_i está precedido por un segmento que apunta en la dirección D_0, cuántas veces está precedido por un segmento que apunta en la dirección D_1, y así sucesivamente. En general, el código j en la tabla de códigos de Huffman para la dirección D_i depende de la probabilidad condicional $P(D_j \mid D_i)$ de que un segmento apuntando en la dirección D_i esté precedido por un segmento que apunta en la dirección D_j. La asignación de 2^{16} lugares para las tablas no es inusual en las aplicaciones actuales (2006), y la cuestión principal es que el decodificador puede imitar todas las operaciones del codificador.

◇ **Ejercicio 8.26 (sol. en pág. 1112):** Estímese la probabilidad condicional $P(D_i \mid D_i)$.

◇ **Ejercicio 8.27 (sol. en pág. 1112):** Estímese la razón de compresión de este método.



8.10. Sequitur

Sequitur se basa en el concepto de gramáticas independientes del contexto, así que empezamos con una breve revisión de este campo. Un lenguaje (natural) comienza con un pequeño número de bloques de construcción (letras y signos de puntuación) y los utiliza para construir palabras y oraciones. Una frase es una secuencia finita (una cadena) de símbolos que obedece ciertas reglas gramaticales, y el número de frases válidas es, a efectos prácticos, ilimitado. Similarmente, un lenguaje formal utiliza un pequeño número de símbolos (llamados *símbolos terminales*) con los cuales se pueden construir

secuencias válidas. Cualquier secuencia válida es finita, el número de secuencias válidas es normalmente es ilimitado, y las secuencias se construyen acorde a ciertas reglas (a veces llamadas *reglas de producción*).

Las reglas pueden usarse para construir secuencias válidas y también para determinar si un secuencia dada es válida. Una regla de producción consta de un símbolo no terminal a la izquierda y una serie de símbolos terminales y no terminales a la derecha. El símbolo no terminal en el lado izquierdo se convierte en el nombre de la cadena a la derecha. En general, el lado derecho puede contener varias cadenas alternativas, pero las reglas generadas por sequitur tienen una única cadena.

Todo el campo de las lenguas y gramáticas formales se basa en el trabajo pionero de Noam Chomsky en la década de 1950 [Chomsky 56]. La notación BNF, que se utiliza para describir la sintaxis de lenguajes de programación, se basa en el concepto de reglas de producción, así como también en los L-sistemas [Salomon 99].

Usamos letras minúsculas para denotar los símbolos terminales y las letras mayúsculas para los no terminales. Supongamos que se dan las siguientes reglas de producción: `A` $\rightarrow$ `ab`, `B` $\rightarrow$ `Ac`, `C` $\rightarrow$ `BdA`. Con estas reglas se pueden generar las cadenas válidas: `ab` (una aplicación del no terminal `A`), `abc` (una aplicación de `B`), `abcdab` (una aplicación de `C`), así como muchas otras. Alternativamente, podemos verificar que la cadena `abcdab` es válida, ya que podemos escribirla como `AcdA`, reescribir ésta como `BdA`, y reemplazar ésta con `C`. Es evidente que la producción reglas reduce la redundancia de la secuencia original, por lo que pueden servir como base para un método de compresión.

En una gramática libre de contexto, las normas de producción no dependen del contexto de un símbolo. Existen también gramáticas sensibles al contexto.

Sequitur (del latín, "se sigue") se basa en el concepto de gramáticas libres de contexto. Se considera el stream de entrada una secuencia válida en algún lenguaje formal. Se lee la entrada símbolo a símbolo y se usan las sentencias repetidas en los datos de entrada para crear un conjunto de reglas de producción libres de contexto. Cada repetición se traduce en una regla, y es reemplazada por el nombre de la regla (un símbolo no terminal), lo que resulta en una representación más corta. Generalmente, un conjunto de reglas de producción puede ser utilizada para generar muchas secuencias válidas, pero las normas de producción producidas por sequitur no son generales. Pueden ser usadas sólo para reconstruir los datos originales. Las reglas de producción en sí mismas no son mucho más pequeñas que los datos originales, por lo que sequitur tiene que efectuar un paso más, donde comprime las reglas de producción. Las reglas de producción comprimidas se convierten en el stream comprimido y el decodificador de sequitur aplica las reglas (tras su descompresión) para reconstruir los datos originales.

Si la entrada es un texto típico de un lenguaje natural, la regla de nivel superior se vuelve muy larga, típicamente 10–20 % el tamaño de la entrada, y las otras reglas son cortas, con típicamente 2–3 símbolos cada una.

La Figura 8.41 muestra tres ejemplos de secuencias y gramáticas de entrada cortas. La secuencia de entrada `S` a la izquierda de la Figura 8.41a ya es una gramática (de una regla). Sin embargo, contiene la cadena `bc` repetida, por lo que esta sentencia se convierte en una regla de producción cuyo nombre es el símbolo no terminal `A`. El resultado es una gramática de dos reglas, donde la primera regla es la secuencia de entrada con su redundancia eliminada, y la segunda regla, que reemplaza el *digrama* `bc` con el único símbolo no terminal `A`, es corta. (El lector debería revisar la discusión sobre la codificación del digrama en la Sección 1.3.)

La Figura 8.41b es un ejemplo de una gramática donde la regla `A` incluye a la regla `B`. La entrada `S` es considerada una gramática de una regla. Tiene redundancia, por lo que cada ocurrencia de `abcdbc` es sustituida por `A`. La regla `A` todavía tiene redundancia, debido a la repetición de la cadena `bc`, lo que justifica la introducción de otra regla `B`.

Sequitur construye sus gramáticas mediante la observación de dos principios (o la aplicación de dos *restricciones*), que denotamos por $p1$ y $p2$. La restricción $p1$ manifiesta: Ninguna pareja de símbolos adyacentes aparecerá más de una vez en la gramática (ésto puede ser dicho de otro modo como: Cada digrama en la gramática es único). La restricción $p2$ dice: Cada regla debe utilizarse más de una vez.

Entrada	Gramática
S → abcdbc	S → aAdA A → bc
(a)	
S → abcdbcabcdbc	S → AA A → aBdB B → bc
(b)	
S → abcdbcabcdbc	S → AA A → abcdbc
	S → CC A → bc B → aA C → BdA
(c)	

Figura 8.41: Tres secuencias de entrada y gramáticas.

Ésto asegura que las reglas sean útiles. Una regla que ocurre sólo una vez es inútil y debería suprimirse.

La secuencia de la Figura 8.41a contiene el digrama `bc` dos veces, por lo que $p1$ requiere la creación de una nueva regla (la regla `A`). Una vez hecho esto, el digrama `bc` se produce sólo una vez, en el interior de la regla `A`. La Figura 8.41c muestra cómo se pueden violar las dos restricciones. La primera gramática de la Figura 8.41c contiene dos apariciones de `bc`, violando así $p1$. La segunda gramática contiene regla `B`, que se utiliza sólo una vez. Es fácil ver cómo la eliminación de `B` reduce el tamaño de la gramática.

◇ **Ejercicio 8.28 (sol. en pág. 1112):** ¡Muéstrese ésto!

El codificador sequitur construye las reglas de la gramática, mientras impone las dos restricciones en todo momento. Si se viola la restricción $p1$, el codificador genera una nueva regla de producción. Cuando se viola $p2$, se elimina la regla inútil. El codificador comienza estableciendo la regla `S` para el primer símbolo de entrada. A continuación, entra en un bucle donde son introducidos y anexados nuevos símbolos a `S`. Cada vez que un nuevo símbolo se anexa a `S`, el símbolo y su predecesor se convierten en el digrama actual. Si el digrama actual ya aparece en la gramática, entonces $p1$ ha sido violada, y el codificador genera una nueva regla con el digrama actual del lado derecho y con un nuevo símbolo no terminal del lado izquierdo. Las dos ocurrencias del digrama son sustituidas por el no terminal.

La Figura 8.42 ilustra el funcionamiento del codificador con la secuencia de entrada `abcdbcabcd`. La columna de la izquierda muestra el nuevo símbolo que se introduce o una acción que está siendo tomada para hacer cumplir una de las dos restricciones. Las siguientes dos columnas de la izquierda, muestran la entrada hasta el momento y la gramática creada hasta ahora. Las dos últimas columnas muestran los digramas duplicados y cualquier regla infrautilizada. La última línea de la Figura 8.42a muestra cómo la introducción del símbolo `c` crea una repetición del digrama `bc`, iniciando así la generación de una nueva regla `A`. Observe que la aparición de un nuevo digrama duplicado no siempre genera una nueva regla. Si, por ejemplo, se crea el nuevo digrama duplicado `xy`, pero se encuentra que está en el lado derecho de una regla existente `A` → `xy`, entonces `xy` se sustituye por `A` y no hay necesidad de generar una nueva regla. Ésto se ilustra en la Figura 8.42b, donde se encuentra una tercera ocurrencia de `bc`. No se genera ninguna regla nueva y la regla existente `A` se adjunta a `S`. Ésto crea un nuevo digrama duplicado `aA` en `S`, por lo que se genera una nueva regla `B` → `aA` en la Figura

8.42c. Por último, la Figura 8.42d muestra cómo la restricción $p2$ resulta en una nueva regla (la regla `C`), cuya parte derecha consta de tres símbolos. Así es como se pueden generar reglas que son más largas que un digrama.

◇ **Ejercicio 8.29 (sol. en pág. 1112):** ¿Por qué se elimina la regla `B` en la Figura 8.42d?

Un detalle más, a saber, la utilización de la regla, aún tiene que ser discutido. Cuando se genera una nueva regla `X`, el codificador genera también un contador asociado a `X`, e inicializa el contador al número de veces que se utiliza `X` (una nueva regla se utiliza normalmente dos veces cuando que se genera por primera vez). Cada vez que `X` es utilizada en otra regla `Y`, el codificador incrementa el contador de `X` en 1. Cuando `Y` es eliminado, el contador de `X` se disminuye en 1. Si el contador de `X` llega a 1, regla `X` es eliminada.

Como se mencionó anteriormente, la gramática (el conjunto de reglas de producción) no es mucho más pequeña que los datos originales, por lo que tiene que ser comprimida. Un ejemplo es el archivo `book1` del Calgary Corpus (Tabla Intr.3), que tiene 768 771 bytes de longitud. El codificador sequitur genera para este archivo una gramática donde la primera regla (la regla `S`) tiene 131 416 símbolos y cada una de las otras 27 364 reglas tiene 1 967 símbolos en promedio. En consecuencia, el tamaño de la gramática es de 185 253 símbolos, o alrededor del 24 % el tamaño de los datos originales; no muy impresionante. Está también la cuestión de los nombres de los símbolos no terminales. En el caso de `book1` hay 27 365 reglas (incluyendo la regla `S`), por lo que se necesitan 27 365 nombres para los símbolos no terminales.

El método descrito aquí para la compresión de la gramática consta de dos partes. La primera parte utiliza la codificación aritmética para comprimir los símbolos gramaticales individuales. La segunda parte proporciona una manera elegante de manejar los nombres de los muchos símbolos no terminales.

La parte 1 emplea la codificación aritmética adaptativa (Sección 2.15), con un modelo de orden-0. La restricción $p1$ asegura que ningún digrama aparece dos veces en la gramática, por lo que no hay ninguna ventaja en el uso de modelos de orden superior para estimar las probabilidades de aparición de los símbolos mientras se comprime una gramática. Ésta es también la razón por la cual los métodos de compresión que utilizan modelos de orden superior, como PPM (Sección 2.18), no harían un mejor trabajo en este caso. La aplicación de este método para la gramática de `book1` produce un rendimiento en la compresión de 3,49 bpc; no muy bueno.

La parte 2 elimina la necesidad de comprimir los nombres de los símbolos no terminales. El número de símbolos terminales (normalmente letras y signos de puntuación) es relativamente pequeño. Éste suele ser el conjunto de 128 caracteres ASCII. El número de no terminales, por el contrario, puede ser enorme (27 365 para `book1`). La solución es no asignar nombres explícitos a los no terminales. El codificador envía (i.e., escribe en el stream comprimido) la secuencia de símbolos de entrada, y cada vez que se genera una regla, envía suficiente información para que el decodificador pueda reconstruir la regla. La regla `S` representa la entrada completa, por lo que este método es equivalente a enviar la regla `S` y otras reglas, a medida que se van generando.

Cuando el codificador encuentra un no terminal al enviar la regla S, éste se maneja de una de tres maneras, dependiendo del número de veces que se ha encontrado en el pasado. La primera vez que un no terminal es localizado en `S`, su contenido (i.e., el lado derecho de la regla) es enviado. El decodificador ni siquiera sabe que está recibiendo los símbolos que constituyen una regla. La segunda vez que se localiza el no terminal, se envía un par (puntero, contador). El puntero es el desplazamiento del no terminal desde el principio de la regla `S`, y el contador es la longitud de la regla. (Ésto es similar a los símbolos generados por LZ77, Sección 3.3.) El decodificador utiliza el par para formar una nueva regla que puede ser usada posteriormente. El nombre de la regla es simplemente su número de serie, y las reglas están numeradas tanto para el codificador como para el decodificador de la misma manera. En la tercera y siguientes apariciones del no terminal, el número de serie es enviado por el codificador e identificado por el decodificador.

Nuevo símbolo o acción	la cadena hasta ahora	gramática resultante	digramas duplicados	reglas sin uso
a	a	S → a		
b	ab	S → ab		
c	abc	S → abc		
d	abcd	S → abcd		
b	abcdb	S → abcdb		
c	abcdbc	S → abcdbc	bc	
aplicar $p1$		S → aAdA A → bc		
	(a)			
a	abcdbca	S → aAdAa A → bc		
b	abcdbcab	S → aAdAab A → bc		
c	abcdbcabc	S → aAdAabc A → bc	bc	
aplicar $p1$		S → aAdAaA A → bc	aA	
	(b)			
aplicar $p1$	abcdbcabc	S → BdAB A → bc B → aA		
	(c)			
d	abcdbcabcd	S → BdABd A → bc B → aA	Bd	
aplicar $p1$		S → CAC A → bc B → aA C → Bd		B
aplicar $p2$		S → CAC A → bc C → aAd		
	(d)			

Figura 8.42: Ejemplo detallado del codificador Sequitur (Tras [Nevill-Manning y Witten 97]).

De esta manera, las primeras dos veces que se encuentra una regla, su nombre (i.e., el número de serie) no se envía, una característica que mejora en gran medida la compresión. Además, en lugar de enviar la gramática al decodificador regla por regla, se envía una regla sólo cuando es necesario para la primera vez. Usando la codificación aritmética, junto con la parte 2 para comprimir `book1` se obtienen unos rendimientos de compresión de 2,82 bpc.

◇ **Ejercicio 8.30 (sol. en pág. 1112):** Muéstrese la información enviada al decodificador por la cadena de entrada `abcdbcabcdbc` (Figura 8.41b).

La información detallada sobre la aplicación real de sequitur puede encontrarse en [Nevill-Manning 96].

Una de las ventajas de sequitur es que cada regla se utiliza más de una vez. Ésto está en contraste con algunos métodos basados en diccionarios que añaden (al diccionario) cadenas que nunca pueden ocurrir en el futuro y por lo tanto nunca podrán ser utilizadas.

Una vez comprendidos los principios de sequitur, es fácil ver que funciona mejor cuando los datos a ser comprimidos constan de cadenas idénticas que son adyacentes. En general, las cadenas idénticas no son adyacentes en el stream de entrada, pero existe un tipo de datos, a saber, *texto semiestructurado*, donde las cadenas idénticas están muchas veces también adyacentes. El texto semiestructurado se define como datos que son legibles por humanos y también adecuados para el procesamiento de la máquina. Un ejemplo común es HTML. Un archivo HTML se compone de texto con etiquetas de marcado embebidas. Hay un pequeño número de etiquetas diferentes, y tienen que ajustarse a determinadas normas. Por lo tanto, las etiquetas pueden ser consideradas altamente estructuradas, en contraste con el texto, que es no estructurado y libre. El archivo HTML completo es por tanto semiestructurado. Otros ejemplos de texto semiestructurado son los formularios, los mensajes de correo electrónico (*emails*), y las bases de datos. Cuando un formulario es almacenado en un ordenador, algunos campos son fijos (estos son las partes altamente estructuradas) y otras partes tienen que ser rellenadas por el usuario (con texto no estructurado, libre). Un email incluye varias partes fijas, además del texto del mensaje. Lo mismo es cierto para una base de datos. Sequitur fue utilizado por los desarrolladores para comprimir dos grandes bases de datos genealógicas [Nevill-Manning y Witten 97], resultando en ratios de compresión de 11–13 %.

8.11. Compresión de mallas de triángulo: Edgebreaker

Las superficies poligonales se utilizan comúnmente en los gráficos por ordenador. Tales superficies se componen de polígonos planos, por lo que son muy sencillas de construir, de guardar en la memoria, y de *renderizar*[7]. Una superficie normalmente es renderizada simulando la luz reflejada de la misma (aunque algunas superficies son renderizadas simulando la luz que emiten o transmiten). Puesto que una superficie poligonal está compuesta por polígonos planos, parece angulosa y poco natural cuando es representada. Sin embargo, existen métodos sencillos (tales como el sombreado de Gouraud y de Phong, véase, e.g., [Salomon 99]) que suaviza la reflexión sobre los polígonos, resultando una superficie lisa y curvada de aspecto realista . Este hecho, combinado con la sencillez de la superficie poligonal, ha hecho que este tipo de superficie sea muy común.

Cualquier polígono plano puede ser utilizado en una superficie poligonal, pero los triángulos son comunes, ya que un triángulo es siempre plano (otros polígonos deben ser testeados en planicidad antes de que puedan ser incluidos en dicha superficie). Ésta es la razón por la que una superficie poligonal es normalmente una malla de triángulos. Dicha malla está completamente representada por las coordenadas de sus vértices (las esquinas de los triángulos) y por su información de conectividad (los bordes que conectan los vértices). Puesto que las superficies poligonales son tan comunes, la compresión de una malla de triángulos es un problema práctico. El algoritmo edgebreaker que se describe aquí [Rossignac 98] es un método eficiente que puede comprimir la información de conectividad de una

[7] "Vestir" la superficie poligonal, cubrirla con alguna imagen.

malla de triángulos a alrededor de dos bits por triángulo; un resultado impresionante (la lista de las coordenadas de los vértices se comprime por separado).

Matemáticamente, la información de conectividad es un grafo llamado *grafo de incidencia*. Por lo tanto, edgebreaker es un ejemplo de *compresor geométrico*. Puede comprimir ciertos tipos de geometrías. Para la mayoría de las mallas de triángulos, el número de triángulos es más o menos dos veces el número de vértices. Como resultado, el grafo de incidencia es típicamente dos veces tan grande como la lista de las coordenadas de los vértices. La lista de las coordenadas de los vértices también es fácil de comprimir, ya que consta de tripletes de números (enteros o reales), pero no está claro inmediatamente cómo comprimir eficientemente la información geométrica incluida en el grafo incidencia. Es por lo que edgebreaker se concentra en la compresión de la información de conectividad.

El codificador edgebreaker codifica la información de conectividad de una malla de triángulos recorriendola triángulo por triángulo, y asignando uno de cinco códigos a cada triángulo. El código expresa la relación topológica entre el triángulo actual y el límite de la malla restante. El código se agrega a una lista historial de códigos de triángulo, y el triángulo actual es eliminado. La eliminación de un triángulo puede cambiar la topología de la malla restante. En particular, si la malla restante está separada en dos regiones con sólo un vértice común, entonces cada región se comprime por separado. En consecuencia, el método es recursivo, y utiliza una pila para salvar un borde de cada una de las regiones en espera de ser codificadas. Cuando el último triángulo de una región es eliminado, el codificador efectúa una operación pop sobre la pila (sacando el último elemento que entró en la misma), y comienza la codificación de otra región. Si la pila está vacía (no hay más regiones para comprimir), el algoritmo termina. El decodificador utiliza la lista de códigos de triángulo para reconstruir la conectividad, tras lo cual, usa la lista de coordenadas de vértices para asignar las coordenadas originales a cada vértice.

El algoritmo de codificación: Asumimos que la malla de triángulos original es homeomórfica a la mitad de una esfera, es decir, que es una región única, y tiene una frontera o límite `B` que es una curva poligonal cerrada sin auto-intersecciones. El límite se llama bucle (*loop*). Se selecciona un borde del límite y se denota por g (ésta es la puerta (*gate*) actual). Puesto que la puerta está en el límite, es un borde de un solo triángulo. Dependiendo de la topología local de g y su triángulo, al triángulo se le asigna uno de los cinco códigos: `C`, `L`, `E`, `R`, o `S`. El triángulo es entonces eliminado (lo que cambia el límite `B`) y la siguiente puerta es seleccionada entre uno de los bordes sobre el límite. La siguiente puerta es un borde del nuevo triángulo actual, adyacente al previo.

El término *homeomorfismo* es un concepto topológico que se refiere a la equivalencia topológica intrínseca. Dos objetos son homeomórficos si pueden ser deformados cada uno en el otro, mediante un mapeo invertible continuo. El homeomorfismo ignora los detalles geométricos de los objetos y también el espacio en el que están inmersos, por lo que la deformación puede llevarse a cabo en un espacio de dimensiones superiores. Ejemplos de objetos homeomórficos son: (1) imágenes espejo, (2) una banda de Möbius con un número par de medias torsiones y otra banda de Möbius con un número impar de medias torsiones, (3) un donut y un anillo.

La Figura 8.43e muestra cómo la eliminación de un triángulo (el marcado con X) puede convertir una malla simple en dos regiones que comparten un vértice pero ningún borde en común. El límite de la nueva malla ahora se corta a sí mismo, por lo que el codificador la divide en dos regiones, cada una con un límite simple sin auto-intersección. Un borde en la frontera de una región es introducido (operación *push*) en la pila, para ser usado más tarde, y el codificador continúa con los triángulos del resto de la región. Si la malla original es grande, pueden formarse muchas regiones durante la codificación. Después de haber asignado un código al último triángulo en una malla y de ser eliminado (el código del último triángulo en una malla es siempre `E`), el codificador saca un elemento de la pila (operación *pop*), y comienza en otra región. Una vez determinado un código para un triángulo, éste es añadido a una historia de compresión `H`. Esta historia es una cadena cuyos elementos son los cinco símbolos

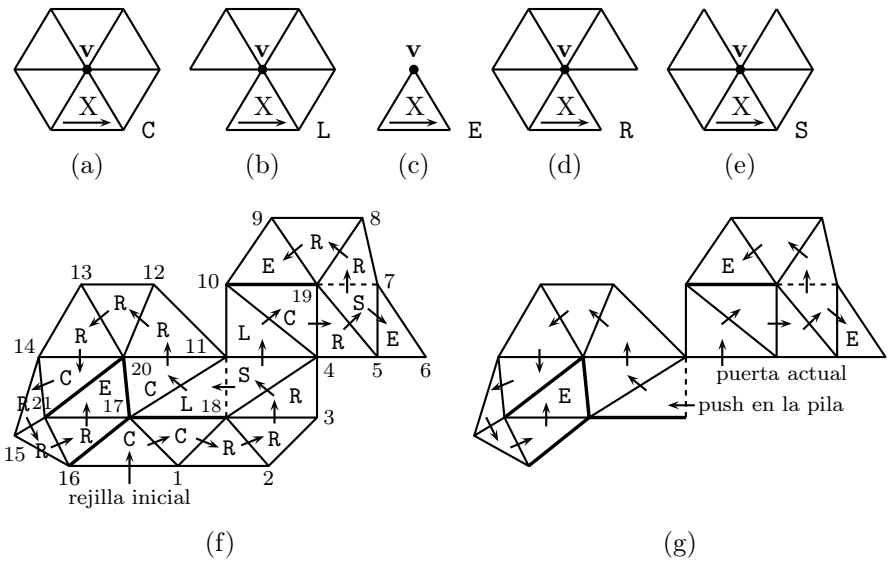


Figura 8.43: Los cinco códigos asignados a los triángulos.

`C`, `L`, `E`, `R`, y `S`, donde cada símbolo es codificado mediante un código prefijo. Sorprendentemente, esta historia es todo lo que el decodificador necesita para reconstruir la conectividad de la malla original.

A continuación, mostramos cómo se asignan los cinco códigos a los triángulos. Sea **v** el tercer vértice del triángulo actual (el triángulo cuyo borde exterior es la puerta actual g, marcada con una flecha). Si **v** todavía no ha sido visitado, al triángulo se le asigna el código `C`. El vértice **v** de la Figura 8.43a no ha sido visitado aún, ya que ninguno de los triángulos que le rodean ha sido eliminado. Por tanto, al triángulo actual (marcado con X) se le asigna el código `C`. A un triángulo se le asigna un código `L` (de *left*, izquierda) si su tercer vértice (el que no está unido a la puerta) es exterior (localizado en el límite) y precede inmediatamente a g (Figura 8.43b). Similarmente, a un triángulo se le asigna un código `R` (de *right*, derecha) si su tercer vértice es exterior y sigue inmediatamente a g (Figura 8.43d). Si el tercer vértice **v** es exterior pero no precede o sigue inmediatamente a g, al triángulo se le asigna el código `S` (Figura 8.43e). Finalmente, si el triángulo es el último en su región (i.e., si el vértice **v** precede y sigue inmediatamente a g), al triángulo se le asigna el código `E` (Figura 8.43c).

La Figura 8.43f muestra un ejemplo de una malla con 24 triángulos. La puerta inicial (seleccionada arbitrariamente) está señalada. Las flechas indican el orden en el que se visitan los triángulos. Los vértices están numerados en el orden en que se añaden a la lista **P** de coordenadas de vértices. El decodificador reconstruye la malla en el mismo orden que el codificador, por lo que sabe cómo asignar a los vértices los mismos números de serie. El decodificador utiliza entonces la lista **P** de coordenadas de vértices para asignar las coordenadas reales a todos los vértices. El tercer conjunto de coordenadas en **P**, por ejemplo, se asigna al vértice etiquetado 3.

Inicialmente, toda la malla es una región. Su límite o frontera es una curva poligonal de 16 segmentos que no se intersecta a sí misma. Durante la codificación, a medida que el codificador elimina más y más triángulos, la malla degenera en tres regiones, por lo que dos bordes (los que se muestran en líneas de trazos) son introducidos en la pila y son extraídos posteriormente. La Figura 8.43g muestra las dos primeras regiones y los últimos triángulos de las tres regiones. Como el codificador se mueve de triángulo en triángulo, todos los bordes interiores (excepto los cinco bordes que se muestran en líneas gruesas) se convierten en puertas. El codificador produce una lista **P** de las coordenadas de todos los

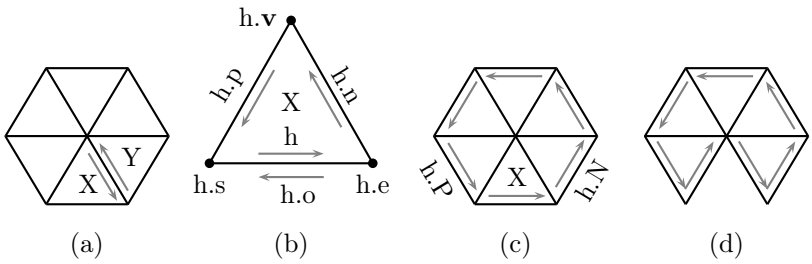


Figura 8.44: Manipulación de los *half-edges*.

21 vértices, así como la historia de la compresión:

$$\texttt{H} = \texttt{CCRRRSLCRSERRELCRRRCRRRE}.$$

Observe que hay tres triángulos con un código `E`. Éllos son los últimos triángulos visitados en las tres regiones. Además, cada triángulo `C` corresponde a un vértice interior (de los cuales hay cinco en nuestro ejemplo).

La lista **P** de las coordenadas de los vértices se inicializa con las coordenadas de los vértices del límite de la malla (si la malla tiene una frontera; una malla que es homeomórfica a una esfera no tiene una curva de contorno). En nuestro ejemplo, éstos son los vértices 1–16. Durante el ciclo principal, mientras el codificador examina y elimina los triángulos, añade un vértice (interior) a **P** para cada triángulo con un código `C`. En nuestro ejemplo, éstos son los cinco vértices interiores 17–21.

Hay que especificar unos pocos términos más, así como alguna notación, antes de poder enumerar los pasos detallados del algoritmo de codificación. El término *half-edge* (borde-mitad) es un concepto topológico útil. Éste es un borde dirigido junto con uno de los dos triángulos que inciden sobre el mismo. La Figura 8.44a muestra dos triángulos X e Y con un borde en común. El borde en común junto con el triángulo X es un half-edge, y el mismo borde en común, junto con el triángulo Y es otro half-edge que apunta en la dirección opuesta. Un borde exterior está asociado con único half-edge. Los siguientes términos están asociados con un half-edge `h` (Figura 8.44b,c):

1. El inicio del vértice h se denota h.s (abreviatura de *h.start*).
2. El final del vértice h se denota h.e (abreviatura de *h.end*).
3. El tercer vértice de X (el que no está ligado a h) se denota h.v (abreviatura de *h.vertex*).
4. El half-edge que sigue a h en el triángulo X se denota h.n (abreviatura de *h.next*).
5. El half-edge que precede a h en el triángulo X se denota h.p (abreviatura de *h.previous*)
6. El half-edge que es opuesto a h se denota h.o (abreviatura de *h.opposite*).
7. El half-edge que sigue a h por la frontera de la malla se denota h.N (abreviatura de *h.Next*).
8. El half-edge que precede a h en el límite se denota por h.P (abreviatura de *h.Previous*).

La Figura 8.44c muestra una malla sencilla compuesta por seis triángulos. Los seis half-edges se muestran siguiendo la frontera de la malla. A medida que el codificador visita los triángulos y los elimina, la frontera de la malla cambia. Cuando se elimina un triángulo, un half-edge que se usaba como parte de la frontera desaparece, y dos half-edges que antes eran internos ahora se situan en la frontera. Ésto se ilustra en la Figura 8.44d. Para ayudar al codificador a manipular los half-edges, están

vinculados en una lista. Ésta es una lista doblemente enlazada cíclica donde cada half-edge apunta tanto a su sucesor como a su predecesor.

El algoritmo presentado aquí utiliza la sencilla notación para indicar operaciones en la lista de half-edges. La notación h.x = y indica que el campo h.x del half-edge h debe establecerse para apuntar a y. El algoritmo también utiliza dos tipos de flags binarios. Los flags **v**.m marcan cada vértice **v** visitado previamente. Se utilizan para distinguir entre los triángulos C y S sin tener que atravesar la frontera. Los flags h.m macan cada half-edge h ubicado en el límite de la porción restante de la malla. Estos flags se usan durante las operaciones S para simplificar el proceso de búsqueda del half-edge b tal que g.**v** es b.**e**. El último elemento de notación es la barra vertical, que se utiliza para denotar la concatenación. En consecuencia, H = H | C indica la concatenación del código C con la historia hasta el momento, y **P** = **P** | **v** indica la operación de añadir el vértice **v** a la lista de coordenadas de vértices **P**.

El algoritmo de codificación edgebreaker ahora puede ser descrito en detalle. Comienza con una etapa de inicialización donde: (1) se selecciona la primera puerta (*gate*) y un puntero a la misma es introducido en la pila, (2) todos los half-edges a lo largo de la frontera son identificados, marcados, y vinculados a una lista doblemente enlazada cíclica, y (3) se inicializa la lista **P** de las coordenadas de los vértices. **P** es inicializada a las coordenadas de todos los vértices en la frontera de la malla (si la malla tiene una frontera) comenzando desde el extremo final del vértice de la puerta.

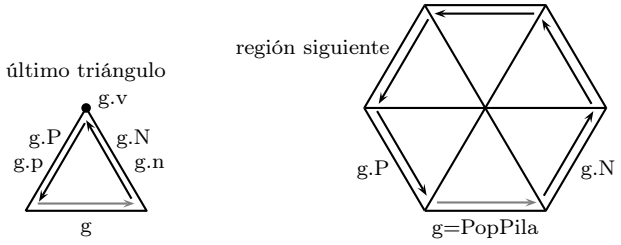
El bucle principal itera sobre los triángulos, comenzando con el triángulo asociado con la primera puerta. A cada triángulo se le asigna un código y es eliminado. Dependiendo del código, el codificador ejecuta uno de cinco casos (recordemos que `B` representa el límite o frontera de la región actual). El bucle puede detenerse sólo cuando a un triángulo se le asigna un código `E` (i.e., es el último en su región). La rutina que maneja el caso `E` intenta sacar de la pila un puntero a la puerta de la siguiente región. Si la pila está vacía, todas las regiones que constituyen la malla han sido codificadas, por lo que la rutina detiene el bucle de codificación. Este es el bucle principal:

```
if not g.v.m then case C% v no marcado
else if g.p==g.P        % borde izquierdo de X está en B
     then if g.n==g.N then case E else case L endif;
     else if g.n==g.N then case R else case S endif;
     endif;
endif;
```

Aquí se muestran los detalles de los cinco casos, junto con los diagramas que ilustran cada caso.

- **Caso C:** La Figura 8.46 muestra una sencilla región con seis half-edges en su frontera, vinculados en una lista. El triángulo inferior obtiene el código `C`, es eliminado, y la lista pierde un half-edge y gana dos nuevos.
- **Caso L:** La Figura 8.47 es un sencillo ejemplo de este caso. El borde izquierdo del triángulo inferior es parte de la frontera de la región. El triángulo es eliminado, y el límite es actualizado.
- **Caso R:** La Figura 8.48 muestra un ejemplo de este caso. El borde derecho del triángulo actual es parte de la frontera de la región. El triángulo es eliminado, y el límite es actualizado.
- **Caso S:** La Figura 8.49 muestra un sencillo ejemplo. El triángulo superior ha desaparecido, por lo que la eliminación del triángulo inferior convierte la malla original en dos regiones. La de la izquierda es introducida en la pila, y el codificador continúa con la región de la derecha.
- **Caso E:** La Figura 8.45 muestra el último triángulo de una región. Sólo es necesario desmarcar sus tres bordes. A continuación, se intenta obtener de la pila la puerta a la siguiente región. Si la pila está vacía (no existen más regiones), el codificador se detiene.

```
H=H|E;                    % añadir E a la historia
g.m=0; g.n.m=0; g.p.m=0; % desmarcar bordes
if PilaVacía then stop
else PopPila; g=PilaTop; % comenzar en la región siguiente
endif
```



(a) (b)

Figura 8.45: Manejo del caso E.

Compresión de la Historia: La cadena H de la historia consta de sólo cinco tipos de símbolos, por lo que pueden ser comprimidos fácil y eficientemente con códigos de prefijos. Un método eficiente es un trabajo de compresión de dos pasadas, donde la primera pasada cuenta la frecuencia de cada símbolo y la segunda pasada efectúa la compresión real. Entre las pasadas, se calcula un conjunto de cinco códigos de Huffman y se escribe en el inicio del stream comprimido. Tal método, sin embargo, es lento. Se puede obtener un resultado más rápido (y sólo ligeramente menos eficiente) seleccionando el siguiente conjunto fijo de cinco códigos de prefijos: Asignar un código de 1 bit a C, y los códigos de 3 bits a cada uno de los otros cuatro símbolos. Una elección posible es el conjunto 0, 100, 101, 110, y 111 para codificar C, S, R, L, y E, respectivamente.

El promedio del tamaño del código no es difícil de estimar. El número total de bits utilizados para codificar la historia H es $c = |\mathtt{T}| - |\mathtt{C}| = |\mathtt{C}| + 3\,(|\mathtt{S}| + |\mathtt{L}| + |\mathtt{R}| + |\mathtt{E}|)$. Denotamos por T al conjunto de triángulos en la malla, por $\mathtt{V_i}$ al conjunto de vértices interiores de la malla, y por $\mathtt{V_e}$ al conjunto de vértices exteriores. El tamaño de $\mathtt{V_i}$ es el tamaño de $\mathtt{C}\,|\mathtt{C}| = |\mathtt{V_i}|$. También utilizamos La ecuación de Euler para mallas simples, $t - e + v = 1$, donde t es el número de triángulos, e es el número de bordes (aristas), y v es el número de vértices. Todo ésto se combina para producir:

$$|\mathtt{S}| + |\mathtt{L}| + |\mathtt{R}| + |\mathtt{E}| = |\mathtt{T}| - |\mathtt{C}| = |\mathtt{T}| - |\mathtt{V_i}|\,,$$

y $c = |\mathtt{V_i}| + 3\,(|\mathtt{T}| - |\mathtt{V_i}|)$ o $c = 2\,|\mathtt{T}| + (|\mathtt{T}| - 2\,|\mathtt{V_i}|)$. Puesto que $|\mathtt{T}| - 2\,|\mathtt{V_i}| = |\mathtt{V_e}| - 2$, obtenemos $c = 2\,|\mathtt{T}| + |\mathtt{V_e}| - 2$.

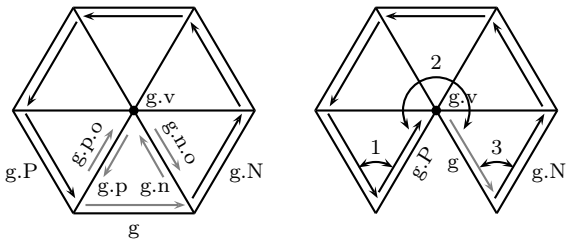
El resultado es que para mallas simples, con una frontera inicial sencilla y corta, tenemos $|\mathtt{V_e}| \ll |\mathtt{V_i}|$ (la malla es interiormente pesada), por lo que $c \approx 2\,|\mathtt{T}|$. La longitud de la historia es de dos bits por triángulo.

Una malla pequeña puede tener un número relativamente grande de bordes exteriores y, por lo tanto, puede ser exteriormente pesada. La mayoría de los triángulos, de tal malla obtienen un código R, por lo que los cinco códigos vistos más arriba deben ser cambiados. El código de R debe ser de 1 bit de longitud, y los otros cuatro códigos deben ser de tres bits cada uno. Este conjunto de códigos produce $c = 3\,|\mathtt{T}| - 2\,|\mathtt{R}|$. Si la mayoría de los triángulos tienen un código R, entonces $|\mathtt{T}| \approx |\mathtt{R}|$ y $c \approx 1$; aún más impresionante.

Concluimos de que el conjunto de cinco códigos de prefijos debe ser seleccionado de acuerdo con la razón $|\mathtt{V_e}| \,/\, |\mathtt{V_i}|$.

El decodificador: Edgebreaker es un método de compresión asimétrico. El funcionamiento del decodificador es muy diferente al del codificador y requiere dos pasadas: preprocesamiento y generación.

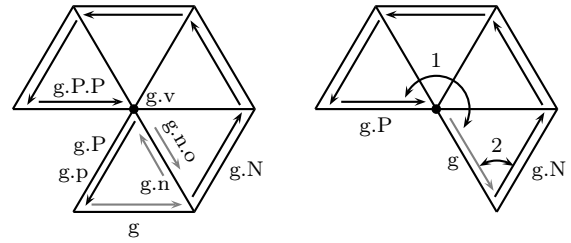
```
H = H|C                          % añadir C a la historia
P = P|g.v;                       % añadir v a P
g.m = 0;  g.p.o.m = 1;           % actualizar flags
g.n.o.m = 1;  g.v.m = 1;
g.p.o.P = g.P;  g.P.N = g.p.o;   % fijar enlace 1
g.p.o.N = g.n.o;  g.n.o.P = g.p.o;  % fijar enlace 2
g.n.o.N = g.N;  g.N.P = g.n.o;   % fijar enlace 3
g = g.n.o;  PilaTop = g;         % avanzar puerta (g)
```



(a) (b)

Figura 8.46: Manejo del caso C.

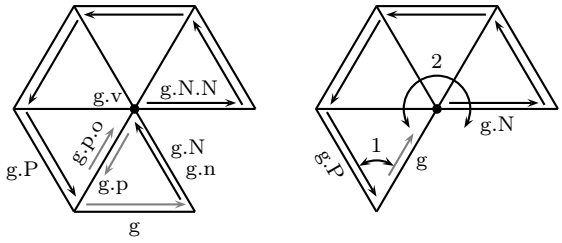
```
H = H|L                          % añadir L a la historia
g.m = 0;  g.P.m = 0;  g.n.o.m = 1;  % actualizar flags
g.P.P.n = g.n.o;  g.n.o.P = g.P.P;   % fijar enlace 1
g.n.o.N = g.N;  g.N.P = g.n.o;   % fijar enlace 2
g = g.n.o;  PilaTop = g;         % avanzar puerta (g)
```



(a) (b)

Figura 8.47: Manejo del caso L.

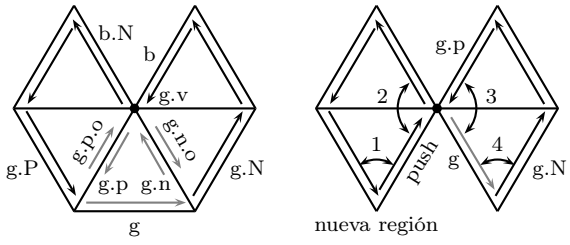
```
H = H|R                          % añadir R a la historia
g.m = 0;  g.N.m = 0;  g.p.o.m = 1;  % actualizar flags
g.N.N.P = g.p.o;  g.p.o.N = g.N.N;      % fijar enlace 1
g.p.o.P = g.P;  g.P.N = g.p.o;          % fijar enlace 2
g = g.p.o;  PilaTop = g;          % avanzar puerta (g)
```



(a) (b)

Figura 8.48: Manejo del caso R.

```
H = H|S                          % añadir S a la historia
g.m = 0;  g.n.o.m = 1;  g.p.o.m = 1;  % actualizar flags
b = g.n                          % candidato inicial para b
while not b.m do b = b.o.p;
                          % convertir v en el b marcado
g.P.N = g.p.o;  g.p.o.P = g.P;          % fijar enlace 1
g.p.o.N = b.N;  b.N.P = g.p.o;          % fijar enlace 2
b.N = g.n.o;  g.n.o.P = b;              % fijar enlace 3
g.n.o.N = g.N;  g.N.P = g.n.o;          % fijar enlace 4
PilaTop = g.p.o;  PushPila;      % salvar región nueva
g = g.n.o;  PilaTop = g;          % avanzar puerta (g)
```



(a) (b)

Figura 8.49: Manejo del caso S.

La pasada de preprocesamiento determina el número de triángulos, bordes, y vértices, así como los desplazamientos para los códigos S. La pasada de generación crea los triángulos en el orden en el que fueron eliminados por el codificador. Este paso determina las etiquetas de los tres vértices de cada triángulo y los almacena en una tabla.

La *etapa de preprocesamiento* lee los códigos de la historia H y efectúa ciertas acciones para cada uno de los cinco tipos de códigos. Utiliza las siguientes variables y estructuras de datos:

1. El contador de triángulos t. Éste es incrementado para cada código, por lo que sigue el número total de triángulos.

2. El valor de $|\mathtt{S}| - |\mathtt{E}|$ es registrado en d. Una vez que se ha comprendido totalmente el funcionamiento del codificador, debería quedar claro que los códigos S y E actúan como pares de corchetes balanceados y que el último código en H es siempre E. Después de leer éste último E, el valor de d se hace negativo.

3. El contador de vértices c es incrementado cada vez que se encuentra un código C en la historia H. El valor actual de c se convierte en la etiqueta del siguiente vértice g.**v**.

4. La variable e registra el valor de $3\,|\mathtt{E}| + |\mathtt{L}| + |\mathtt{R}| - |\mathtt{C}| - |\mathtt{S}|$. Su valor final es $|\mathtt{V_e}|$. Cuando se lee un código S, el valor actual de e es introducido por el decodificador en una pila.

5. El número de códigos S es registrado por la variable s. Cuando e es introducido en la pila, s se usa para establecer una conexión entre e y el correspondiente S.

6. Una pila en la que se introducen los pares (e, s) cuando se encuentra un código S. El último par es sacado de la pila cuando se lee un código E de H. Se utiliza para calcular el desplazamiento.

7. Una tabla O de los desplazamientos.

Todas las variables se inicializan a cero. La pila y la tabla O empiezan vacías.

Las operaciones realizadas en la etapa de preprocesamiento para cada tipo de código son las siguientes:

código S : $e{-}=1;\ s{+}=1;\ \mathtt{push}\,(e,s)\,;\ d{+}=1;$
código E : $e{+}=3;\ (e',s')=\mathtt{popPila};\ O\,[s']=e=e'-2;\ d{-}=1;\ \mathtt{if}\ d<0,\ \mathtt{stop}.$
código C : $e{-}=1;\ c{+}=1;$
código L : $e{+}=1;$
código R : $e{+}=1;$

Además, t es incrementada para leer cada código. Al final de este paso, t contiene el número total $|\mathtt{T}|$ de triángulos , c es el número $|\mathtt{V_i}|$ de vértices interiores, e se establece en el número de $|\mathtt{V_e}|$ de vértices exteriores y la tabla O contiene los desplazamientos, ordenados en el orden en que los códigos S se producen en H.

Deben explicarse dos puntos en relación con estas operaciones. El primero es por qué el valor final de e es $|\mathtt{V_e}|$. El cálculo de e utiliza H para averiguar cuántos bordes se añadieron a la frontera B o fueron eliminados de la misma por el codificador. Este número es luego utilizado para determinar el tamaño inicial de B. Recordemos que el codificador elimina dos bordes de B y añade un borde la misma durante el procesamiento de un código R o un código L. El procesamiento de un código E por el codificador implica la eliminación de tres bordes.

◇ **Ejercicio 8.31 (sol. en pág. 1112):** ¿Cómo cambia el número de aristas (bordes) cuando el codificador procesa un código C o un código S?

	C	C	R	R	R	S	L	C	R	S	E	R	R	E	L	C	R	R	R	C	R	R	R	E
t	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
d	0	0	0	0	0	1	1	1	1	2	1	1	1	0	0	0	0	0	0	0	0	0	0	−1
c	1	2	2	2	2	2	2	3	3	3	3	3	3	3	3	4	4	4	4	5	5	5	5	5
e	−1	−2	−1	0	1	0	1	0	1	0	3	4	5	8	9	8	9	10	11	10	11	12	13	16
s	0	0	0	0	0	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
e',s'						0,1	0,1	0,1	0,1	0,2	0,1	0,1	0,1											
$O\,[s']$											1		6											

Tabla 8.50: Un ejemplo de paso de preprocesamiento.

Estos cambios de recuento de bordes implican que el número inicial de bordes (y por lo tanto también el número inicial de vértices) en la frontera es $3\,|\mathtt{E}| + |\mathtt{L}| + |\mathtt{R}| - |\mathtt{C}| - |\mathtt{S}|$. Es por ésto por lo que e registra este número.

El segundo punto son los desplazamientos. Sabemos que los códigos `S` y `E` actúan como corchetes emparejados en `H`. Cualquier subcadena en `H` que comienza con una `S` y termina con la `E` correspondiente contiene la información de conectividad para una región de la malla original. También sabemos que e registra el valor de $3\,|\mathtt{E}| + |\mathtt{L}| + |\mathtt{R}| - |\mathtt{C}| - |\mathtt{S}|$ para la subcadena de `H` que ya ha sido leída y procesada. Por consiguiente, la diferencia entre los valores de e en la operación de `E` y en la operación de `S` correspondiente es el número de vértices en la frontera de esa región. Restamos 2 de este valor a fin de no contar los dos vértices de la puerta g como parte del desplazamiento.

Para cada código `S`, el valor de e es introducido en la pila. Cuando el correspondiente código `E` es leído de `H`, se extrae un elemento de la pila y es restado del valor actual de e.

El paso de preprocesamiento se ilustra mediante la aplicación a la historia

$$\mathtt{H} = \mathtt{CCRRRSLCRSERRELCRRRCRRRE}$$

que ha sido generada por el codificador a partir de la malla de la Figura 8.43f. La Tabla 8.50 muestra los valores de todas las variables involucradas, después de haber leído cada código de `H`. El par (e', s') es el contenido de la parte superior de la pila del decodificador. Los dos desplazamientos, 1 y 6, se almacenan en la tabla `O`. El valor final de e es 16 (el número de vértices exteriores o, equivalentemente, el número de vértices en `B`). El valor final de c es 5. Éste es el número de vértices interiores. El valor final de t es, por supuesto, 24, el número de triángulos.

La *etapa de generación* comienza con una fase de inicialización donde se efectúa siguiente:

1. Comienza con una tabla vacía `TV` de vértices de triángulos, donde cada entrada contendrá las etiquetas de los tres vértices de un triángulo.

2. Inicializa un contador de vértices c a $|\mathtt{V_e}|$, por lo que las referencias a los vértices exteriores preceden a las referencias de los interiores.

3. Construye una lista circular doblemente enlazada de bordes $|\mathtt{V_e}|$, donde cada nodo corresponde a un borde `G` y contiene un puntero `G.P` al nodo precedente, un puntero `G.N` al nodo sucesor, y una etiqueta entera `G.e` que identifica el vértice final del borde `G`. Las etiquetas son enteros crecientes de 1 a $|\mathtt{V_e}|$.

4. Crea una pila de referencias a los bordes y la inicializa con una única entrada que hace referencia al primer borde `G` (la puerta) en el límite `B`. Observe que se utilizan letras mayúsculas para los bordes para distinguirlas de los half-edges usados por el codificador.

5. Establece un contador de triángulos $t = 0$ y un contador $s = 0$ de operaciones `S`.

Luego, realiza una pasada leyendo la historia H símbolo a símbolo. Para cada símbolo determina las etiquetas de los tres vértices del triángulo actual X y las almacena en la TV $[t]$. También actualiza B, G, y la pila, si es necesario. Una vez que la puerta actual, G, es conocida, determina dos de los tres vértices del triángulo actual. Éstos son: G.P.e y G.e. La determinación del tercer vértice depende del código actual leído de H. Aquí se muestran las acciones para cada uno de los cinco códigos (la notación x ++ significa devolver el valor actual de x, luego incrementarlo; mientras que ++ x significa incrementar x, y luego retornar su nuevo valor):

código C : TV$[++\,t] = (\texttt{G.P.e}, \texttt{G.e}, ++\,c)$;
Nuevo borde A; A.e $= c$;
G.P.N = A; A.P = G.P;
A.N = G; G.P = A;
código R : TV$[++\,t] = (\texttt{G.P.e}, \texttt{G.e}, \texttt{G.N.e})$;
G.P.N = G.N; G.N.P = G.P;
G = G.N;
código L : TV$[++\,t] = (\texttt{G.P.e}, \texttt{G.e}, \texttt{G.P.P.e})$;
G.P = G.P.P; G.P.P.N = G;
código E : TV$[++\,t] = (\texttt{G.P.e}, \texttt{G.e}, \texttt{G.N.e})$;
G = PopPila;
código S : D = G.N; repetir D = D.N; O$[++\,s]$ veces;
TV$[++\,t] = (\texttt{G.P.e}, \texttt{G.e}, \texttt{D.e})$;
Nuevo borde A; A.e = D.e;
G.P.N = A; A.P = G.P;
PopPila; Push A;
A.N = D.N; D.N.P = A;
G.P = D; D.N = G;
Push G;

Estas acciones se han sido ilustradas aquí para la historia

H = CCRRRSLCRSERRELCRRRCRRRE,

originalmente generada por el codificador a partir de la malla de la Figura 8.43f. El paso de preprocesamiento computa $|\mathtt{V_e}| = 16$, así que comenzamos con una frontera inicial de 16 bordes y 16 vértices que etiquetamos de 1 a 16. El primer borde (el que está asociado con el vértice 1) es nuestra puerta inicial G. La variable c se establece en 16. El primer código C encontrado en H establece la entrada TV $[1]$ en las tres etiquetas 16 (G.P.e), 1 (G.e), y 17 (el resultado de $++\,c$). El lector debería utilizar la Figura 8.43f para verificar que estos son, de hecho, los vértices del primer triángulo procesado y eliminado por el codificador. Un nuevo borde, A, es también creado, y 17 se almacena como su etiqueta. El nuevo borde A es insertado delante de G actualizando los punteros de la siguiente manera: G.P.N = A, A.P = G.P, A.N = G y G.P = A. El segundo código C crea el triángulo $(17, 1, 18)$ y se inserta otro borde nuevo, con la etiqueta 18, delante de G. (El lector debe verificar de nuevo, que $(17, 1, 18)$ son los vértices del segundo triángulo procesado y eliminado por el codificador.) El primer código R crea el triángulo $(18, 1, 2)$, elimina la puerta G de la frontera, y declara el borde etiquetado 2 como la puerta actual.

◇ **Ejercicio 8.32 (sol. en pág. 1112):** Muéstrese el resultado de procesar el segundo y tercer código R.

El primer código `S` salta los seis vértices (seis, porque `O`[1] = 6) 5, 6, 7, 8, 9, y 10. Después, determina que el triángulo 6 tiene como vértices (18, 4, 11), y divide la frontera en dos regiones(11, 12, 13, 14, 15, 16, 17, 18) y (4, 5, 6, 7, 8, 9, 10). La parte inferior de la pila apunta al borde (8, 11), el primer borde en la primera región. La parte superior de la pila apunta al borde (11, 4), el primer borde en la segunda región. El código `L` crea el triángulo (11, 4, 10) y elimina el último borde de la segunda región. En este punto, el borde `G` actual es borde (10, 4).

[Rossignac 98] tiene más detalles de este interesante y original método, incluyendo extensiones de edgebreaker para mallas con agujeros en ellos, y para mallas sin fronteras o límites (mallas que son homeomórficas a una esfera).

8.12. SCSU: Compresión Unicode

El código ASCII es viejo, habiendo sido diseñado en la década de 1960. Con el advenimiento de los láseres de bajo costo y las impresoras de inyección de tinta y las pantallas de alta resolución, se ha hecho posible visualizar e imprimir caracteres de cualquier tamaño y forma. Como resultado, el código ASCII, con sus 128 caracteres, ya no satisface las necesidades de la informática moderna. Comenzando en 1991, el consorcio de Unicode (cuyos miembros incluyen a grandes corporaciones informáticas, productores de software, proveedores de bases de datos, centros de investigación, organismos internacionales, varios grupos de usuarios e individuos interesados) ha propuesto y diseñado un nuevo esquema de codificación de caracteres que satisface las demandas de hardware y software actuales. Existe información disponible sobre Unicode en [Unicode 03].

El estándar Unicode asigna un número, denominado punto de código (*code point*) a cada carácter (elemento de código). Un punto de código se muestra en hexadecimal con una "U+" que lo precede. En consecuencia, el punto de código U+`0041` es el número $\texttt{0041}_{16} = 65_{10}$. Representa el carácter "A" en el estándar Unicode.

El estándar Unicode también asigna un nombre único a cada carácter. Al elemento de código U+`0041`, por ejemplo, se le asigna el nombre "LETRA A MAYÚSCULA LATINA", y a U+`0A1B`, se le asigna el nombre "LETRA CHA GURMUKHI".

Una característica importante del estándar Unicode es la forma en que relaciona los grupos de códigos. Un grupo de caracteres relacionados se conoce como *script*, y a tales caracteres se les asignan códigos consecutivos; se convierten en una zona o región contigua de Unicode. Si los caracteres están en orden en el script original (como, A–Z en el alfabeto latino y de α a ω en el griego), entonces sus Unicodes reflejan ese orden. Los tamaños de las regiones varían mucho, dependiendo del script.

La mayor parte de los puntos de código Unicode son números de 16 bits (2 bytes). Existen 64K (ó 65 536) de estos códigos, pero Unicode reserva 2 048 de los códigos de 16 bits para extender este sistema a códigos de 32 bits (así añade alrededor de 1,4 millones de pares de códigos suplentes). La mayoría de los caracteres de uso común encajan en los primeros 64K puntos de código, una región del espacio de códigos que se denomina plano multilingüe básico (BMP o *basic multilingual plane*). Hay alrededor de 6 700 puntos de código no asignados para una futura expansión del BMP, además de más de 870 000 puntos de código adicionales no utilizados en las otras regiones del espacio de código. Más caracteres están en estudio para su inclusión en futuras versiones del estándar.

El Unicode comienza con el conjunto de 128 códigos ASCII U+`0000` a U+`007F` y continúa con el griego, cirílico, hebreo, árabe, índico y otros scripts. Éstos son seguidos por símbolos y signos de puntuación, signos diacríticos, símbolos matemáticos, símbolos técnicos, flechas, *dingbats*[8], y así sucesivamente. El espacio de códigos continúa con el Hiragana y el Katakana (ambos para transcribir el japonés), y el Bopomofo (para transcribir el chino). Estos unificados ideogramas de Asia oriental (*ideogramas Han)* son seguidos por el conjunto completo del Hangul moderno (para transcipción del coreano). Hacia el final del BMP hay un conjunto de puntos de código reservados para uso privado,

[8] Ornamento carácter o espaciador usado en tipografía.

seguido por una serie de caracteres de compatibilidad. Los caracteres de compatibilidad son variantes de caracteres que se codifican sólo para permitir la transcodificación de estándares anteriores y aplicaciones obsoletas que van a usarlos.

El estándar Unicode también reserva puntos de código para uso privado. Cualquiera puede asignar estos códigos privados para sus propios caracteres y símbolos o utilizarlos con fuentes especiales. Hay 6 400 puntos de código de uso privado en el BMP y otros 131 068 puntos de código complementarios de uso privado en otros lugares del espacio de códigos.

La versión 3.2 de Unicode especifica códigos para 95 221 caracteres de alfabetos del mundo, conjuntos de ideogramas, y colecciones de símbolos. La versión actual[9] es 5 y sus especificaciones detalladas deben publicarse a finales de 2006. (La documentación de la última versión del estándar Unicode siempre se encuentra en el vínculo estable: `http://www.unicode.org/versions/latest/`.)

El método descrito en esta sección se debe a [Wolf et al. 00]. Es un esquema de compresión estándar para Unicode, abreviado SCSU (acrónimo de *standard compression scheme for Unicode*). Comprime cadenas de puntos de código. Al igual que cualquier método de compresión, funciona eliminando la redundancia de los datos originales. La redundancia en Unicode deriva del hecho de que el texto típico en Unicode tiende a tener caracteres que se encuentran en la misma región del espacio de códigos Unicode. En consecuencia, un texto que utiliza el conjunto básico de caracteres latinos está formado sobre todo por puntos de código de la forma U+00xx. Éstos pueden ser comprimidos a un byte cada uno. Un texto en árabe tiende a utilizar sólo caracteres árabes, que comienzan en U+0600. Tal texto puede ser comprimido especificando una dirección de inicio y luego convirtiendo cada punto de código a su distancia (o desplazamiento) desde esa dirección. Ésto introduce el concepto de ventana. La distancia debe ser de un solo byte, porque una distancia de 2 bytes reemplazando a 2 bytes de código no produce compresión. Este tipo de compresión se denomina modo byte-único. Un desplazamiento de 1 byte indica un tamaño de ventana de 256, pero ya veremos por qué el método utiliza ventanas de la mitad de ese tamaño. En la práctica, puede haber complicaciones, como demuestran los siguientes tres ejemplos:

1. Una cadena de texto en un script (secuencia de comandos) determinado puede tener marcas de puntuación embebidas en él, y éstos tienen puntos de código en una región diferente. Una marca de puntuacion única dentro de una cadena puede ser escrita en formato *raw* (tal cual es, sin codificarlo) en el stream comprimido y también requiere una etiqueta especial para indicar un código raw. El resultado es una ligera expansión.

2. El script puede incluir cientos o incluso miles de caracteres. En un cierto punto, el siguiente carácter a ser comprimido puede estar demasiado lejos de la dirección de inicio, por lo que debe especificarse una nueva dirección de inicio (una nueva ventana) sólo para el siguiente carácter. Ésto se hace mediante una etiqueta "cambio-sin bloqueo".

3. Del mismo modo, en un cierto instante, los caracteres que están siendo comprimidos pueden estar todos en un ventana diferente, por lo que tiene que insertarse una etiqueta "cambio-con bloqueo", para indicar la dirección de inicio de la nueva ventana.

Como resultado, el método emplea etiquetas, lo que implica que una etiqueta tiene que ser de al menos algunos bits, lo que plantea la cuestión de cómo distingue el decodificador las etiquetas de los caracteres comprimidos. La solución es limitar el tamaño de la ventana a 128. Hay 8 ventanas estáticas y 8 ventanas dinámicas (Las Tablas 8.51 y 8.52, respectivamente, donde CJK es un acrónimo de Chino, Japonés y Coreano). Las posiciones de inicio de este último se pueden cambiar por etiquetas.

SCSU emplea las siguientes convenciones:

1. Cada etiqueta es un byte en el intervalo $[0\times 00, 0\times 1F]$, excepto que los códigos ASCII para CR ($0\times 0D$), LF ($0\times 0A$) y TAB (o HT 0×09) no se utilizan para las etiquetas. Por tanto,

[9] En el momento de la traducción de este libro al castellano, la última versión de Unicode es la 6.2.0.

n	Inicio	Área principal
0	0000	Entrecomillado en el modo byte-único
1	0080	Latín1 suplementario
2	0100	Latín extendido-A
3	0300	Combinación de marcas diacríticas
4	2000	Marcas de puntuación general
5	2080	Símbolos de moneda
6	2100	Símbolos de letras y formas de números
7	3000	Símbolos y puntuación CJK (*Chino, Japonés, koreano*)

Tabla 8.51: Ventanas estáticas.

n	Inicio	Área principal
0	0080	Latín1 suplementario
1	00C0	Latín1 supl. + Latín extendido-A
2	0400	Cirílico
3	0600	Arábigo
4	0900	Devanagari
5	3040	Hiragana
6	30A0	Katakana
7	FF00	ASCII de ancho completo

Tabla 8.52: Ventanas dinámicas.

puede haber $32 - 3 = 29$ etiquetas (pero ya veremos que hay más valores reservados para las etiquetas en el modo Unicode). Las etiquetas se utilizan para indicar un cambio a otra ventana, un reposicionamiento de una ventana, o un salto a un modo no comprimido (*raw*) llamado modo Unicode.

2. Un código en el rango, de U+0020 a U+007F se comprime eliminando sus ocho ceros más significativos. Se convierte en un solo byte.

3. Cualquier otro código se comprime a un byte en el rango, de 0×80 a $0 \times FF$. Éstos indican desplazamientos en el rango, de 0×00 a $0 \times EF$ (0 a 127) en la ventana actual.

Ejemplo: La cadena 041C, 043E, 0441, 002D, 043A, 0562 y 000D se comprime a los bytes 12, 9C, BE, C1, 2D, BA, 1A, 02, E2, y 0E. La etiqueta byte 12 indica la ventana, de 0x0400 a 0x047F. El código 041C es un desplazamiento de 1C desde el inicio de esa ventana, por lo que se comprime al byte $1C + 80 = 9C$. El código 043E está en el desplazamiento 3E, por lo que se comprime a $3E + 80 = BE$. El código 0441 se comprime de manera similar a C1. El código 002D (el código ASCII de un guión) se comprime (sin etiquetas) a sus 8 bits menos significativos, 2D. (Ésto es menos de 0x80, por lo que el decodificador no se confunde.) El código 043A es comprimido en la ventana actual a BA, pero la compresión del código 0562 debe hacerse en la ventana [0x0500, 0x057F], por lo que debe ir precedido de la etiqueta 1A (seguido por el índice 02) que selecciona esta ventana. El desplazamiento del código 0562 en la nueva ventana es 62, por lo que se comprime al byte E2. Por último, el código 000D (CR) se comprime a sus ocho bits menos significativos 0E sin una etiqueta adicional.

La etiqueta 12 se llama SC2. Indica un desplazamiento bloqueado para la ventana dinámica 2, que comienza por defecto en 0x0400. La etiqueta 1A se denomina SD2 e indica un reposicionamiento de la ventana 2. El byte 02 que sigue 1A es un índice a la Tabla 8.53 y cambia la posición de comienzo de la ventana a $2x80_{16} = 100_{16}$, por lo que la ventana se mueve desde el original, 0x0400 a 0x0500.

X	Desplazamiento [X]	Comentarios
00	reservado	para uso interno
01–67	$X \times 80$	half-blocks de U+0080 a U+3380
68–A7	$X \times 80 + \texttt{AC00}$	half-blocks de U+E000 a U+FF80
A8–F8		reservado para uso futuro
F9	00C0	caracteres Latín1 + medio o extendido-A
FA	0250	extensiones IPA (*International Phonetic Alphabet**)
FB	0370	Griego
FC	0530	Armenio
FD	3040	Hiragana
FE	30A0	Katakana
FF	FF60	Katakana de media anchura

*Alfabeto fonético internacional

Tabla 8.53: Desplazamientos de ventanas.

Comenzamos con los detalles del modo byte-único. Este modo es el elegido por defecto cuando comienza el codificador SCSU. Cada código de 16 bits es comprimido en este modo a un solo byte. Las etiquetas se necesitan de vez en cuando, y pueden ir seguidas por hasta dos argumentos, cada uno de un byte. Este modo continúa hasta que se encuentra una de las siguientes etiquetas: (1) fin-de-entrada, (2) SCU, o (3) SQU. En este modo se utilizan seis tipos de etiquetas (para un total de 27 etiquetas diferentes) de la siguiente manera:

1. SQU (0E). Esta etiqueta (denominada *quote Unicode*) es un cambio temporal (sin bloqueo) al modo Unicode. Esta etiqueta es seguida por los dos bytes de un código raw.

2. SCU (0F). Esta etiqueta (*change to Unicode*) es un cambio permanente (con bloqueo) al modo Unicode. Ésto se utiliza para una cadena de caracteres consecutivos que pertenecen a distintos scripts y están por lo tanto en ventanas diferentes.

3. SQn (01–08). Esta etiqueta (*quote from window n*) es un cambio sin bloqueo a la ventana n. Invoca (i.e., escribe en formato raw) el código siguiente, por lo que no hay compresión. El valor de n (entre 0 y 7) es determinado por la etiqueta (entre 1 y 8). Esta etiqueta debe ir seguida por un byte usado como un desplazamiento a la ventana seleccionada. Si el byte está en el intervalo, de 00 a 7F, debe seleccionarse la ventana estática n. Esta etiqueta estima un código, luego vuelve al modo byte-único. Por ejemplo, SQ3 seguido por 14 selecciona desplazamiento de 14 en la ventana estática 3, por lo que estima el código $\texttt{0300} + \texttt{14} = \texttt{0314}$. Otro ejemplo es SQ4 seguido de 8A. Éste selecciona el desplazamiento $\texttt{8A} - \texttt{80} = \texttt{0A}$ en la ventana dinámica 4 (que normalmente comienza en 0900, pero podría ser reposicionada), por lo que estima el código $\texttt{0900} + \texttt{0A} = \texttt{090A}$.

4. SCN (10–17). Esta etiqueta (*change to window n*) es un cambio con bloqueo a la ventana n.

5. SDN (18–1F). Esta etiqueta (*define window n*) reposiciona la ventana n y la convierte en la ventana actual. La etiqueta es seguida por un índice de un byte a la Tabla 8.53 que indica la nueva dirección de inicio de la ventana n.

6. SDX (0B). Esta es la etiqueta de "*define extended*". Es seguida por dos bytes denotados H y L. Los tres bits más significativos de H determinan la ventana estática a ser seleccionada, y los restantes 13 bits de H y L se convierten en un entero N que determina la dirección de inicio de la ventana como $\texttt{10000} + \texttt{80} \times N$ (hexadecimal).

La etiqueta SQ0 es importante. Se utiliza como un flag para los puntos de código cuyo byte más significativo puede ser confundido con una etiqueta (i.e., que está en el rango, de `00` a `1F`). Cuando se encuentra tal byte, el decodificador debe ser capaz de decir si se trata de una etiqueta o el inicio de un código raw. Como resultado, cuando el codificador introduce un código que comienza con semejante byte, lo escribe en la salida en formato raw (marcado), precedido por una etiqueta SQ0.

Luego viene el modo Unicode. Cada carácter se escribe en este modo en formato raw, por lo que no se produce compresión (hay incluso una ligera expansión debida a las etiquetas requeridas). Una vez que este modo es seleccionado mediante una etiqueta SCU, permanece efectiva hasta el final de la entrada o hasta que se encuentre una etiqueta que seleccione una ventana activa. Se utilizan cuatro tipos de etiquetas en este modo de la siguiente manera:

1. UQU (`F0`). Esta etiqueta identifica un carácter Unicode. Los dos bytes siguientes a la etiqueta son escritos en la salida en formato raw.

2. UCN (`E0`–`E7`). Esta etiqueta es un cambio con bloqueo al modo byte-único y también selecciona la ventana n.

3. UDN (`E8`–`EF`). Define la ventana n. Esta etiqueta es seguida por un índice de un solo byte. Selecciona la ventana n y la reposiciona de acuerdo con las posiciones de inicio de la Tabla 8.53.

4. UDX (`F1`). Define la ventana extendida. Esta etiqueta (similar a SDX) es seguida por dos bytes denotados H y L. Los tres bits más significativos de H determinan la ventana dinámica a ser seleccionada, y los restantes 13 bits de H y L se convierten en un entero N que determina la dirección de inicio de la ventana mediante $\texttt{10000} + \texttt{80} \times N$ (hexadecimal).

Los cuatro tipos de etiquetas requieren 18 valores, pero casi todos los 29 valores de etiquetas posibles son utilizados por el modo byte-único. Como resultado, el modo Unicode usa valores de variables que son puntos de código válidos. El byte `E0`, por ejemplo, es la etiqueta UC0, pero también es la mitad más significativa de un punto de código válido (de hecho, es la mitad más significativa de 256 puntos de código válidos). Por tanto, el codificador reserva estos 18 valores (más unos pocos más para uso futuro) para etiquetas. Cuando el codificador encuentra cualquier carácter cuyo código comienza con uno de esos valores, el carácter se escribe en formato raw (precedido por una etiqueta UQU). Estos casos no son comunes, porque los valores reservados se han tomado de la zona de uso privado de Unicode, y esta área se utiliza muy poco.

SCSU también especifica formas de comprimir sustitutos de Unicode. Con códigos de 16 bits, se tienen 65 536 códigos. Sin embargo, $\texttt{800}_{16} = 2048_{10}$ códigos de 16 bits han sido reservados para una extensión de Unicode a códigos de 32 bits. Los $\texttt{400}_{16}$ códigos, de U+`D800` a U+`DBFF`, están reservados como sustitutos altos, y los $\texttt{400}_{16}$ códigos, de U+`DC00` a U+`DFFF`, están reservados como sustitutos bajos. Ésto permite unos $\texttt{400} \times \texttt{400} = \texttt{100 000}_{16}$ códigos adicionales de 32 bits. Un código de 32 bits se conoce como par suplente y puede ser codificado en SCSU en una de varias maneras, tres de las cuales se muestran aquí:

1. En el modo Unicode, en formato raw (cuatro bytes).

2. En el modo byte-único, con cada mitad citada. En consecuencia, SQU, H1, L1, SQU, H2, L2.

3. También en el modo byte-único, como un solo byte, estableciendo una ventana dinámica en la posición adecuada con una etiqueta SDX o UDX.

La secuencia de dos códigos U+`FEFF` (o su homólogo invertido U+`FFFE`) se produce muy rara vez en archivos de texto, por lo que sirve como una firma, para identificar archivos de texto en Unicode. Esta secuencia es conocida como un *byte de marca de orden* (BOM o *byte order mark*). SCSU recomienda varias formas de comprimir esta firma, y un codificador puede seleccionar cualquiera de ellas

8.12.1. BOCU-1: Compresión Unicode

El acrónimo BOCU (*binary-ordered compression for Unicode*) procede de compresión binaria ordenada para Unicode. BOCU es un sencillo método de compresión para archivos de tipo Unicode [BOCU 01]. Su característica principal es preservar el orden binario de los puntos de código que están siendo comprimidos. En consecuencia, si los dos puntos de código x e y se comprimen en a y b y si $x < y$, entonces $a < b$.

El método básico BOCU se basa en la diferenciación (Sección 1.3.1). El punto de código previo se resta del punto de código actual para producir un valor diferencia. Los puntos de código consecutivos en un documento normalmente son similares, por lo que la mayoría de las diferencias son pequeñas y en forma en un solo byte. Sin embargo, debido a que un punto de código en Unicode 2.0 (publicado en 1996) está en el rango, de U+`000000` a U+`10FFFF` (códigos de 21 bits), las diferencias pueden, en principio, ser cualquier número en el intervalo [−`10FFFF`, `10FFFF`] y pueden requerir hasta tres bytes cada uno. Este método básico se mejora de dos maneras.

La primera mejora aumenta la compresión en pequeños alfabetos. En Unicode, la mayoría del los alfabetos pequeños comienzan en un límite de 128 bytes, aunque el tamaño del alfabeto puede ser de más de 128 símbolos. Ésto sugiere que no se calcule la diferencia entre los valores de código actual y previo, sino entre el valor de código actual y el valor ubicado en la mitad del segmento de 128 bytes donde se encuentra el valor de código anterior. Específicamente, la diferencia se calcula restando un *valor base* desde el punto de código actual. El valor base se obtiene a partir del punto de código anterior de la siguiente manera: Si el valor del código anterior está en el intervalo de `xxxx00` a `xxxx7F` (i.e., sus siete bits menos significativos son, de 0 a 127), el valor base se establece en `xxxx40` (los siete LSBs son 64), y si el punto de código anterior está en el rango, de `xxxx80` a `xxxxFF` (i.e., sus siete bits menos significativos son, de 128 a 255), el valor base se establece en `xxxxC0` (los siete LSBs son 192). De esta manera, si el punto de código actual está dentro de 128 posiciones del valor base, la diferencia está en el rango [−128, +127], lo que hace que encaje en un byte.

La segunda mejora tiene que ver con los símbolos remotos. Un documento en un alfabeto no latino (donde los puntos de código son muy diferentes de los códigos ASCII) puede utilizar espacios entre las palabras. El punto de código de un espacio es el código ASCII 20_{16}, por lo que cualquier par de puntos de código que incluya un espacio genera una gran diferencia. Por eso BOCU calcula una diferencia como primer cómputo de los valores base de los tres puntos de código previos, y luego resta el valor base más pequeño del punto de código actual.

BOCU-1 es la versión de BOCU que se utiliza comúnmente en la práctica [BOCU 02]. Esta difiere del método original BOCU en el uso de un conjunto diferente de rangos de valores de byte y en la codificación de los caracteres de control ASCII, de U+`0000` a U+`0020` con los valores de byte, de 0_{16} a 20_{16}, respectivamente. Estas características hacen que BOCU-1 sea adecuado para la compresión de archivos de entrada que sean tipos de medios MIME[10] (texto).

> Il faut avoir beaucoup étudié pour savoir peu (es necesario estudiar mucho para saber un poco).
>
> —Montesquieu (Charles de Secondat), *Pensées diverses*

8.13. Formato de Documento Portátil (PDF)

Las primeras computadoras digitales se desarrollaron en la década de 1940, durante y después de la Segunda Guerra Mundial, como una herramienta rápida y flexible usada sobre todo para romper

[10] MIME es un acrónimo de *Multipurpose Internet Mail Extensions*; en español, extensiones multipropósito de correo de internet. Se utilizan para el intercambio de todo tipo de archivos (texto, audio, video, etc.) a través de internet, de forma transparente para el usuario.

(descifrar) códigos. Ya en la década de 1950, había una gran variedad de modelos de computadoras, fabricadas por diversos fabricantes como Philco, Fairchild, NCR y RCA. No es de extrañar que estos primeros ordenadores no siguieran ningún estándar de organización de datos y fueran incompatibles. La introducción del código ASCII, en 1967 (su última actualización en 1986), hizo mucho para normalizar los datos digitales. Por primera vez, se hizo relativamente fácil transferir datos entre ordenadores — primero en cintas magnéticas y discos, y posteriormente a través de redes—. Con el advenimiento de las aplicaciones multimedia en la década de 1980 se hizo posible representar, almacenar y editar cualquier tipo de datos, texto, imágenes, vídeo y audio en formato digital en un ordenador. Los usuarios inmediatamente sintieron la necesidad de un estándar que hiciera que facilitara crear, editar, imprimir, y transferir documentos multimedia entre ordenadores.

En 1991, John Warnock, cofundador de Adobe Inc., llegó con un esbozo de un estándar que permitiría a los usuarios de computadoras de cualquier lugar hacer precisamente éso. Sus ideas se convirtieron en la base del estándar del formato de documento portátil (PDF o *portable document format*), que actualmente es muy popular. PDF fue introducido por primera vez al público en 1992 en la conferencia de COMDEX, y se hizo comercialmente disponible, bajo el nombre comercial de *Acrobat*, en 1993. Las ventajas de esta técnica se notaron inmediatamente por los usuarios de computadoras comerciales y privados, y ya en 1994 el gobierno de Estados Unidos lo adoptó para distribuir los formularios de impuestos para el público. La aparición del software libre *Acrobat reader* en 1995 hizo mucho para aumentar la popularidad del PDF, y aún más fue hecho por las extensiones del estándar básico. Características tales como el color, los plugins para navegadores de Internet, y los códigos de caracteres de doble byte, han sido añadidas gradualmente a PDF y en 1999 PDF se convirtió en un estándar ANSI. Hoy en día, existen varias versiones de PDF como PDF/A (para archivado, publicado por la ISO en 2005), PDF/E (para ingeniería) y PDF/X (para impresión). Una referencia general para PDF es [adobepdf 06].

Una prueba sorprendente de la inmensa popularidad de PDF se produjo en el 2000, cuando un libro de Stephen King, *Riding the Bullet* (*Viaje a las tinieblas*), fue descargado 400 000 veces en las primeras 24 horas de estar disponible como un libro electrónico (*e-book*).

Las principales características del formato PDF son las siguientes:

- Cualquier aplicación puede enviar su salida al software de Acrobat que lo convierte a PDF. En consecuencia, es fácil crear documentos en este formato.
- Una vez que un documento PDF ha sido creado, puede ser enviado a otra plataforma informática donde se puede leer, editar e imprimir.
- El software de Acrobat permite editar documentos PDF mediante la adición de marcadores y comentarios, marcas de dibujo, medidas de dimensiones, retoques del texto, y recorte de páginas.
- Integridad. Un documento PDF se ve exactamente igual que el original. Todos los detalles de texto, dibujos, gráficos en color y fotografías se preservan totalmente.
- Seguridad. Es posible cifrar un documento PDF con una contraseña para que sólo personas autorizadas puedan leerlo. También es posible hacer el documento de sólo lectura, requieriendo una contraseña para copiar sus datos o modificarlo. Las firmas digitales protegidas por contraseña también pueden ser incluidas.
- Búsqueda fácil. En un documento PDF se pueden buscar palabras o frases. Es también posible añadir marcadores y saltar a cualquier marcador de forma rápida y sencilla.
- Formato abierto. Los detalles del formato PDF están disponibles en [adobepdf 06].
- Tamaño de archivo pequeño. PDF utiliza algoritmos de compresión para comprimir el texto y las imágenes en un documento. Esta importante característica es la favorita especial de los usuarios

de archivos PDF que aman el pequeño espacio de almacenamiento ocupado por documentos de gran tamaño y el poco tiempo que lleva transferirlos entre ordenadores.

El archivo de salida generado por un compresor es binario, pero PDF puede codificar opcionalmente tal archivo en ASCII base-85, un método desarrollado por Adobe Inc., por lo que parece un archivo de texto y consta de códigos ASCII de 7 bits. Este proceso produce cinco códigos ASCII para cada cuatro bytes de datos binarios y por lo tanto provoca expansión (en un factor de $^5/_4 = 1{,}25$). Dado un grupo de cuatro bytes b_1, b_2, b_3, y b_4, el codificador genera cinco bytes de salida, de c_1 a c_5 a partir de la relación:

$$b_1 \times 256^3 + b_2 \times 256^2 + b_3 \times 256^1 + b_4 = c_1 \times 85^4 + c_2 \times 85^3 + c_3 \times 85^2 + c_4 \times 85^1 + c_5.$$

Esta relación se puede interpretar de la siguiente manera. Un byte consta de ocho bits, por lo que tiene valores en el rango, de 0 a 255. Los cuatro bytes de entrada se consideran los dígitos de un entero en base 256 que se convierte a un entero en base 85. Cada uno de los cinco bytes c_i resultantes, tiene por lo tanto valores en el rango, de 0 a 84, y posteriormente se convierte en un carácter ASCII sumando 33. Los caracteres resultantes tienen códigos que abarcan, desde 33 (carácter ASCII !) hasta 117 (carácter ASCII `u`). Si todos los cinco bytes de salida son cero, se representan mediante el código ASCII 122 (carácter `z`). El último grupo de bytes introducido puede tener sólo uno, dos, o tres bytes, y es completado a cuatro bytes añadiendo bytes 0 cuando sea necesario.

En comparación con las propiedades anteriores, las características de compresión de PDF parecen sencillas. Los algoritmos utilizados por PDF para comprimir los diversos tipos de datos son, o bien métodos conocidos, o bien métodos o versiones de aquéllos ampliamente utilizados. PDF emplea LZW (Sección 3.12) para comprimir texto, gráficos e imágenes. A partir de PDF versión 1.2, Flate (una variante de Deflate, Sección 3.23) es también usado para los mismos tipos de datos. JPEG (Sección 4.8) se utiliza para comprimir imágen en color y en escala de grises. Comenzando con PDF 1.5, JPEG 2000 (Sección 5.19) también es usado con el mismo propósito. Las imágenes monocromáticas (binivel) también pueden ser comprimidas en PDF mediante compresión de fax (grupo 3 o grupo 4, Sección 2.13) o la codificación run-length (Sección 1.2). A partir de PDF 1.4, JBIG2 (Sección 4.12) es también utilizado para el mismo propósito. El software de Acrobat normalmente selecciona el método de compresión apropiado de forma automática, pero los usuarios pueden, en principio, activar o desactivar cualquier algoritmo de compresión. La desactivación de JPEG y JBIG2 puede ser importante en ciertas aplicaciones porque estos métodos se utilizan normalmente para la compresión con pérdida y deben ser evitados en aplicaciones en las que la pérdida de datos es prohibitiva.

La codificación run-length utilizada por PDF produce secuencias de bytes donde el primer byte de una secuencia especifica una longitud y puede ser seguida por hasta 128 bytes de datos. Si la longitud está en el rango $[0, 127]$, especifica "ningún *run*". En este caso, el byte de longitud es seguido por longitud + 1 (i.e., entre uno y 128) bytes de datos. Si la longitud está en el rango $[129, 255]$, implica que el único byte que le sigue constituye un run de (257 − longitud) bytes idénticos (i.e., un run de longitud, de dos a 128 bytes). Una longitud de 128 especifica fin-de-datos (EOD o *end-of-data*).

En el mejor de los casos (los datos constan de un run largo), este método produce una secuencia de dos bytes para cada conjunto de 128 bytes idénticos, lo que es equivalente a un factor de compresión de 64. En el peor de los casos (ningún run), cada secuencia de 128 bytes de datos es codificado en 129 bytes, provocando de este modo una ligera expansión.

8.14. Diferenciación de archivos

(Esta sección está escrita de forma conjunta con Giovanni Motta.)

El término *diferenciación de archivos* se refiere a cualquier método que localiza y comprime las diferencias entre dos archivos. Imagine un archivo `A` con dos copias que son mantenidas por dos

usuarios. Cuando una copia es actualizada por un usuario, debe ser enviada al otro usuario, para mantener las dos copias idénticas. En lugar de enviar una copia de A, que puede ser grande, puede enviarse un archivo mucho más pequeño que contenga sólo las diferencias, en formato comprimido, y usarse en el extremo receptor para actualizar la copia de A.

En la literatura profesional, este tipo de compresión es a menudo llamada "diferenciación". Utilizamos "diferenciación de archivos" porque el término "diferenciación" se utiliza en este libro (Sección 1.3.1) para describir un método de compresión completamente diferente.

Diferenciación: provocar que difieran, para que sean diferentes.

Más formalmente, la diferenciación de archivos se refiere a la compresión de un conjunto de datos *objetivo* dando un conjunto de datos de *referencia*. Las aplicaciones de esta técnica de compresión incluyen la distribución y actualizaciones de software (o *parches*), la revisión de sistemas de control, la compresión de copias de seguridad de archivos, y el almacenamiento en archivos de varias versiones de los datos.

Cuando se utiliza para la compresión de archivos de copia de seguridad o en una revisión de sistemas de control, se pueden almacenar múltiples versiones del mismo archivo de forma compacta mediante el almacenamiento de la versión más reciente y las diferencias con la versión inmediatamente precedente. Si es necesario, la diferencias se pueden utilizar para recuperar versiones anteriores. En estas aplicaciones, la diferencia (a veces llamada *reverso delta*) se aplica para recuperar una versión más antigua. Lo contrario sucede en el caso de un sistema de distribución de software, donde la diferencia (*patch* o *parche*) es aplicada con el fin de actualizar el software de una versión original (más vieja).

Si los archivos de referencia y objetivo son lo suficientemente similares, la diferenciación de archivos es capaz de generar archivos comprimidos que son órdenes de magnitud más pequeñas de lo que se puede lograr con técnicas de compresión ordinarias.

La diferenciación de archivos es particularmente útil para la distribución de actualizaciones de software a través Internet. Aún más relevante es su aplicación al parcheado y actualización de los dispositivos móviles inalámbricos como los teléfonos celulares y PDAs, donde la capacidad del enlace inalámbrico es limitado.

Yo no pinto cosas. Yo sólo pinto las diferencias entre las cosas. —Henri Matisse

8.14.1. Utilidad diff de UNIX

La primera aproximación a la diferenciación de archivos utiliza una combinación de operaciones que añaden (`APPEND`), eliminan (`DELETE`) y cambian (`CHANGE`) las líneas de texto para transformar un archivo en otro. Una popular herramienta de UNIX, `diff`, se basa en este paradigma [Hunt 76]. Dadas dos archivos de texto, `diff` genera un conjunto mínimo de cambios en forma de comandos. Los comandos, aplicados en secuencia, transforman el primer archivo en el segundo. Un compresor sin pérdidas (`gzip`, por ejemplo) puede ser utilizado para reducir aún más el tamaño de los comandos que `diff` emite. `diff` genera comandos en un formato legible por humanos. Opcionalmente, puede generar lotes de comandos que pueden alimentar directamente un editor de texto como `ed`.

La Figura 8.54 muestra la salida de `diff` aplicada a los archivos de texto `OLD` y `NEW`. El modificador opcional –`e` se utiliza para obtener una salida que sea compatible con el editor de texto `ed`. La salida está formada por los números de línea, los comandos y los parámetros. El punto "." finaliza una secuencia de parámetros. El comando `a` se utiliza para añadir nuevas líneas de texto, `d` borra una o más líneas, y `c` reemplaza el contenido de una o más líneas. Con el archivo `OLD` abierto en el editor `ed`,

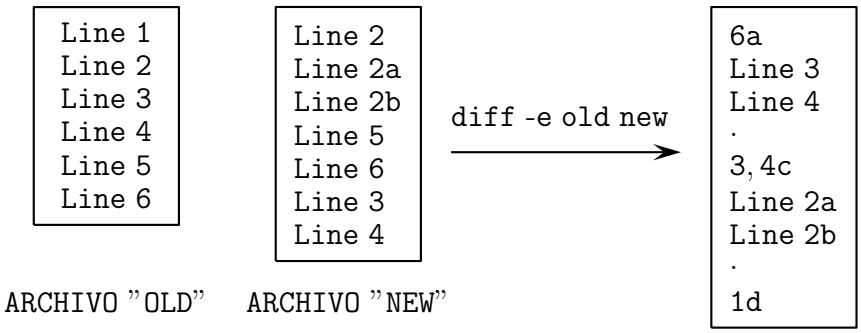


Figura 8.54: Comando `diff` de UNIX.

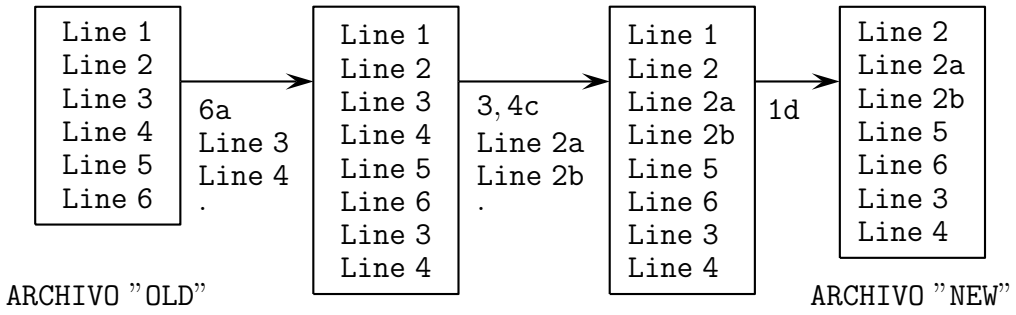


Figura 8.55: Parcheado con el editor `ed`.

el comando `6a` posiciona el cursor en la sexta línea y agrega las líneas de texto "`Line 3`" y "`Line 4`". El siguiente comando, `3, 4c` cambia la tercera y la cuarta línea en "`Line 2a`" y "`Line 2b`", respectivamente. Finalmente, `1d` elimina la primera línea y completa la transformación. El efecto individual de cada comando se puede ver en la Figura 8.55.

Los comandos emitidos por `diff` añaden, eliminan y cambian líneas de texto completas, por lo que `diff` falla al captar las diferencias basadas en caracteres. Una limitación incluso mayor que impide a `diff` lograr una compresión más alta es su incapacidad para copiar patrones que están fuera de orden o se repiten varias veces en el archivo objetivo. Por ejemplo, en la Figura 8.55 podemos ver cómo "`Line 3`" y "`Line 4`" se anexan al final del archivo con su texto y son enviados explícitamente como un parámetro del comando `6a`. `diff` no utiliza el hecho de que dos líneas idénticas ya están presentes en el archivo original y pueden ser copiados al final del mismo con un comando que tiene una representación más compacta.

El núcleo del algoritmo usado por `diff` se basa en la solución de los problemas de subsucesiones comunes más largos [Cormen et al. 01]. Se introducen optimizaciones específicas para reducir el uso de memoria y lograr un buen rendimiento en archivos de texto normales. Variantes y mejoras en el algoritmo original han sido descritas en [Myers 86].

8.14.2. Diferenciación de archivos: VCDIFF

El método VCDIFF descrito aquí se debe a [Korn et al. 02]. Comenzamos con un archivo fuente. El archivo es copiado, y la copia es modificada para convertirse en un archivo objetivo. La tarea es utilizar ambos archivos para codificar las diferencias entre ellos en forma comprimida, de tal manera que cualquiera que tenga el archivo origen y las diferencias codificadas será capaz de reconstruir el

archivo objetivo. El principio consiste en anexar el archivo objetivo al archivo de origen, aplicar al archivo combinado LZ77 o un método similar, y comenzar a escribir el archivo comprimido cuando se alcance el archivo objetivo (i.e., cuando el inicio del archivo objetivo alcance el límite entre el buffer de lectura anticipada y el buffer de búsqueda). Incluso sin ningún detalle, es inmediatamente claro que la compresión es un caso especial de la diferenciación de archivos. Si no existe el archivo origen, entonces la diferenciación de archivos se reduce a una compresión LZ77 del archivo objetivo.

Cuando el proceso LZ77 alcanza al archivo objetivo, éste se comprime de manera muy eficiente debido a que el archivo de origen ya ha sido completamente leído en el buffer de búsqueda. Hay que recordar que el archivo objetivo es una versión modificada del archivo origen. En consecuencia, muchas partes del archivo objetivo provienen del archivo origen y por lo tanto se encuentran en el buffer de búsqueda, lo que se traduce en una excelente compresión. Si el archivo de origen es demasiado grande para caber completamente en el buffer de búsqueda, ambos archivos, origen y destino, tienen que ser segmentados, y la diferencia entre cada par de segmentos debe ser comprimida por separado.

Los desarrolladores de VCDIFF proponen una variante del viejo algoritmo LZ77 que comprime las diferencias entre los archivos origen y objetivo, y crea un archivo comprimido "delta" con tres tipos de instrucciones: `ADD`, `RUN` y `COPY`. Imaginamos el archivo fuente, con bytes denotados por $S_0, S_1, \ldots, S_{s-1}$, seguido inmediatamente por el archivo objetivo, con bytes $T_0, T_1, \ldots, T_{t-1}$. Ambos archivos se almacenan en un buffer U, por lo que el índice de buffer de S_i es i y el índice de buffer de T_j es $s + j$. El codificador VCDIFF tiene un trabajo no trivial y no está descrito en [Korn et al. 02]. La tarea del codificador es escanear U, encontrar coincidencias entre las cadenas de origen y de objetivo, y crear un archivo delta con instrucciones delta. Cualquier codificador que cree un archivo delta válido se considera un codificador VCDIFF compatible. El decodificador VCDIFF, por otro lado, es sencillo y emplea instrucciones delta para generar el archivo objetivo de la siguiente manera:

- `ADD` tiene dos argumentos, una longitud x y una secuencia de x bytes. Los bytes son anexados por el decodificador al archivo objetivo que está siendo generado.
- `RUN` es un caso especial de `ADD` donde los bytes x son idénticos. Tiene dos argumentos, una longitud x y un solo byte b. El decodificador ejecuta `RUN` añadiendo x ocurrencias de b al archivo objetivo.
- `COPY` también tiene dos argumentos, una longitud x y un índice p en el buffer U. El decodificador localiza la subcadena de longitud x que comienza en el índice p en U y agrega una copia al archivo objetivo.

El siguiente ejemplo está tomado de [Korn et al. 02]. Asume que los archivos origen y objetivo son las cadenas:

$$a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l\ m\ n\ o\ p$$
$$a\ b\ c\ d\ w\ x\ y\ z\ e\ f\ g\ h\ e\ f\ g\ h\ e\ f\ g\ h\ e\ f\ g\ h\ z\ z\ z\ z$$

Por tanto, las diferencias entre ellos pueden expresarse mediante las cinco instrucciones delta:

$$\begin{aligned}&\texttt{COPY4}, 0\\ &\quad\texttt{ADD4}, w\ x\ y\ z\\ &\texttt{COPY4}, 4\\ &\texttt{COPY12}, 24\\ &\quad\texttt{RUN4}, z\end{aligned}$$

que son fácilmente decodificadas. El decodificador coloca el archivo origen en su buffer, luego lee y ejecuta las instrucciones delta para crear el archivo objetivo en el mismo buffer, inmediatamente tras

el archivo origen. La primera instrucción indica al decodificador que debe copiar los cuatro bytes a partir del índice 0 del buffer (es decir, los primeros cuatro símbolos fuente) al objetivo. La segunda instrucción le dice que añada los cuatro bytes $w\ x\ y\ z$ al objetivo. La tercera instrucción se refiere a los cuatro bytes que comienzan en el índice 4 del buffer (i.e., la cadena $e\ f\ g\ h$). Éstos son anexados al archivo objetivo. En este punto, el buffer del decodificador consta de los 28 bytes:

$$a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l\ m\ n\ o\ p\ |\ a\ b\ c\ d\ w\ x\ y\ z\ e\ f\ g\ h$$

por lo que la cuarta instrucción, que se refiere a los 12 bytes que comienzan en el índice 24, es especial. El decodificador copia primero los cuatro bytes $e\ f\ g\ h$ que comienzan en el índice 24 (ésto aumenta el tamaño del buffer a 32 bytes), después copia los cuatro bytes desde el índice 28 al 31, entonces repite ésto una vez más hasta copiar un total de 12 bytes. (Observe cómo el archivo objetivo creado parcialmente es usado por estas instrucciones para añadir datos al mismo.) La última instrucción añade los cuatro bytes z en el búfer, completando así el archivo objetivo.

Un archivo delta comienza con una cabecera que contiene varios detalles sobre el resto del archivo. La cabecera es seguida por ventanas, donde cada ventana está formada por las instrucciones delta para cada segmento de los archivos origen y objetivo. Si los archivos son lo suficientemente pequeños para caber en un buffer, entonces el archivo delta tiene sólo una ventana. Cada ventana comienza con varios ítems que especifican el formato de las instrucciones delta, seguidos por las instrucciones delta es sí mismas. Para una mejor compresión, las instrucciones se codifican en tres arreglos o arrays, como se muestra en el resto de esta sección.

Una instrucción `COPY` tiene dos argumentos, de los cuales el segundo es un índice. Estos índices son las ubicaciones de las coincidencias, por lo que a menudo están correlacionados. Recordemos que una coincidencia o emparejamiento (*match*) es la misma cadena que se encuentra en los archivos origen y objetivo. Por lo tanto, si se produce una coincidencia en el índice p del buffer, lo más probable es que la siguiente ocurrencia se encuentre en el índice $p + e$ del buffer, donde e es un número positivo. Por esta razón VCDIFF escribe los índices de emparejamientos consecutivos en un array especial (`addr`), separado de sus instrucciones, y en formato relativo (cada índice es escrito con respecto a su predecesor).

El número de posibles instrucciones delta es enorme, pero la experiencia indica que, la mayor parte del tiempo, se utiliza un pequeño número de esas instrucciones, mientras que el resto rara vez son utilizadas. Como resultado, se ha especificado una "tabla de códigos de instrucción" especial con 256 entradas para codificar eficientemente las instrucciones delta. Cada entrada de la tabla corresponde a una instrucción delta de uso común o a un par de instrucciones consecutivas, y el archivo delta en sí tiene un array (`inst`) con índices a la tabla en lugar de instrucciones reales. La tabla es fija y está integrada tanto en el codificador como en el decodificador, por lo que no necesita ser escrita en el archivo delta (aplicaciones especiales pueden requerir tablas diferentes, y ésas tienen que ser escritas en el archivo). La tabla de códigos de instrucciones aumenta la compresión del archivo delta, pero requiere un codificador sofisticado. Si el codificador tiene que usar una cierta instrucción delta que no está en la tabla, tiene que cambiar su estrategia y utilizar en su lugar dos (o más) instrucciones de la tabla.

Cada entrada en la tabla de códigos de instrucciones consta de dos tripletes (o, equivalentemente, seis campos): *inst1*, *size1*, *modo1*; e *inst2*, *size2,* y *modo2*. Cada triplete indica una instrucción delta, un tamaño, y un modo. Si el campo "inst" es 0, entonces el triplete indica que no hay instrucción delta. El campo "size" es el tamaño de los datos asociados con la instrucción (las instrucciones `ADD` y `RUN` tienen datos, pero estos datos están almacenados en un array separado). El campo "modo" se usa sólo para las instrucciones `COPY`, que tienen varios modos.

En resumen, las instrucciones delta se escriben en el archivo delta en tres arreglos. El primer array (`data`) tiene los valores de los datos para las instrucciones `ADD` y `RUN`. Esta información no se comprime. El segundo array (`inst`) son las propias instrucciones, codificadas como punteros a la tabla de códigos

de instrucciones. Esta matriz es una secuencia de tripletes (índice, size1, size2) donde "índice" es un índice de la tabla de códigos de instrucciones (indicando una o dos instrucciones delta) y los dos tamaños (opcionales) anticipan los tamaños incluidos en la tabla. El tercer array (`addr`) contiene los índices de las instrucciones copiar, codificados en formato relativo.

8.14.3. Zdelta

Zdelta es un algoritmo de diferenciación de archivos desarrollado por Dimitre Trendafilov, Nasir Memon y Torsten Suel [Trendafilov et al. 02]. Zdelta adapta la librería de compresión `zlib` al problema de la compresión de archivos diferencial. Similarmente a VCDIFF, zdelta representa el archivo objetivo mediante la combinación de las copias de tanto el archivo de referencia como el objetivo. Un codificador de Huffman es utilizado para comprimir aún más esta representación.

Las copias de los archivos referencia y objetivo se encuentran con el uso de dos tablas hash. Si el archivo de referencia es suficientemente pequeño, la tabla hash correspondiente es construida por completo previamente mediante operaciones hash de cada secuencia de tres caracteres consecutivos (3-gramas) en el archivo de referencia. En la tabla hash, cada 3-grama está asociado a la posición de su primer carácter en el archivo. Los archivos de referencia grandes requieren el uso de una ventana que refrene el uso de la memoria. La tabla hash para el archivo objetivo se construye durante la compresión. Al igual que en `zlib`, las entradas hash nunca se eliminan de forma individual. La tabla hash se vacía periódicamente para obtener espacio para las nuevas entradas.

Un comando `COPY` tiene cuatro parámetros: la longitud de la copia, el desplazamiento desde uno de varios punteros, el puntero en sí, y la dirección del desplazamiento. Una de las diferencias más importantes con VCDIFF es el uso de múltiples punteros para especificar la ubicación de la copia. El uso de múltiples punteros permite una codificación más compacta. Zdelta mantiene y actualiza de forma independiente uno o más punteros en el archivo de referencia y un puntero implícito en el archivo objetivo. El puntero implícito en el archivo objetivo siempre apunta al comienzo de la sección que está a punto de ser comprimida. Los desplazamientos pueden ser especificados desde cualquier puntero, pero se consigue una codificación más compacta prefiriendo siempre el puntero que genera el menor desplazamiento.

La posición de cada puntero representa una estimación (predicción) de la ubicación del siguiente emparejamiento. El desplazamiento (i.e., la más exacta de esas estimaciones) puede interpretarse como el error de predicción.

Después de cada emparejamiento, los punteros en el buffer de referencia se mueven de acuerdo a una estrategia predeterminada que tiene como objetivo la predicción de la posición de la siguiente copia. En [Trendafilov et al. 02], los experimentos muestran que los mejores resultados se han conseguido manteniendo sólo dos triples en el archivo de referencia y uno en el objetivo. La adición de más punteros complica la codificación y conlleva sólo una ganancia de compresión muy modesta. Tras un emparejamiento, sólo se actualiza un puntero en el archivo de referencia. Si el desplazamiento del emparejamiento actual es menor que una cantidad dada, el puntero que se utiliza en ese desplazamiento (la coincidencia más cercana) es trasladado al final de la copia. Ésto se hace para anticipar la localización de la siguiente copia en el caso de archivos similares. De lo contrario, el otro puntero se mueve al final de la copia. Esta estrategia toma en cuenta la posibilidad de un emparejamiento aislado. Si zdelta no puede encontrar una coincidencia de al menos tres caracteres, se emite un carácter como un literal y el proceso de búsqueda de coincidencias se reinicia a partir del siguiente carácter en el archivo.

La implementación descrita en [Trendafilov et al. 02] permite emparejamientos de hasta 1026 caracteres y desplazamientos en el rango $[0, 32766]$. Zdelta intenta reutilizar tanto como sea posible la biblioteca `zlib`. Por desgracia, el codificador Huffman utilizado en `zlib` sólo admite códigos para longitudes de $[0, 255]$ caracteres. Zdelta supera esta limitación representando la longitud l de un emparejamiento como $L = (l + 3) + 256 \times c$ y la codificación L y c por separado. Cuando se usan tres

c (longitudes)	$-ptr_{\text{objetivo}}$	$+ptr_{\text{ref(1)}}$	$-ptr_{\text{ref(1)}}$	$+ptr_{\text{ref(2)}}$	$-ptr_{\text{ref(2)}}$
0 (3–258)	código 1	código 2	código 3	código 4	código 5
1 (259–514)	código 6	código 7	código 8	código 9	código 10
2 (515–770)	código 11	código 12	código 13	código 14	código 15
3 (771–1026)	código 16	código 17	código 18	código 19	código 20

Tabla 8.56: Codificación para los flags de zdelta.

punteros, es posible codificar en una única palabra de código —llamada *flag zdelta*— el parámetro c, el puntero, y el signo del desplazamiento.

La codificación utilizada en [Trendafilov et al. 02] aparece en la Tabla 8.56 y usa 20 códigos diferentes, ya que existen cuatro valores posibles para el parámetro c y cinco combinaciones de puntero y dirección de desplazamiento. La dirección del desplazamiento puede ser positiva o negativa para punteros en el buffer de referencia, pero solamente negativa para el puntero implícito en buffer del objetivo. Zdelta utiliza tres árboles de Huffman distintos: uno para los desplazamientos, uno para los literales y las longitudes, y otro para los flags.

Se utiliza una estrategia codiciosa para buscar emparejamientos. Todas las coincidencias encontradas se comparan acorde a su longitud, y emparejamientos más largos son preferibles a los más cortos. Si un desplazamiento de emparejamiento es más grande que una constante dada, su longitud es penalizada por lo que puede seleccionarse una coincidencia un poco más corta pero más cercana en su lugar. Los emparejamientos más cercanos son siempre preferibles porque su desplazamiento puede ser codificado de forma más compacta.

El mejor emparejamiento no se emite de inmediato. Su longitud se guarda en una variable l_{prev} y se compara con la mejor coincidencia comenzando en el siguiente carácter. El más largo de los dos emparejamientos es emitido. Si se selecciona el segundo de los dos, el carácter adicional es emitido como un literal. Esta estrategia se ha tomado de `zlib`, y se sabe que logra una buena compresión, ya que mitiga la avidez de la selección del emparejamiento mediante el aplazamiento de la elección de la coincidencia más larga.

Una tabla hash con compartimentos superpoblados ralentiza el tiempo de ejecución sin mejora sustancial en la compresión. La existencia de compartimentos extremadamente grandes puede ser debido a una función hash defectuosa (y, por lo tanto, a un error de diseño) o a muchas repeticiones del mismo patrón en los archivos. Puesto que zdelta se basa en la función hash que utiliza `zlib`, es probable que un compartimento grande esté causado por muchas repeticiones del mismo patrón. Ésto sucede por ejemplo cuando un archivo contiene muchas repeticiones consecutivas del mismo carácter. Zdelta evita este problema limitando el número de elementos buscados en un compartimento hash dado a 1 024.

Zdelta reutiliza muchas de las funciones de `zlib`. Las principales diferencias son:

- Una tabla hash adicional mantiene los punteros al conjunto de datos de referencia.
- Se utiliza un árbol adicional Huffman para codificar los flags de zdelta.
- La compresión se lleva a cabo en una sola etapa, con el archivo de referencia totalmente disponible desde el principio.

8.14.4. Exediff

Un algoritmo de compresión de archivos diferencial basado en las operaciones `COPY` y `ADD` trabaja bien con archivos de texto porque la inserción y la eliminación de material nuevo son operaciones que se realizan típicamente en el texto. Por otra parte, en los archivos de texto, los cambios tienden a estar

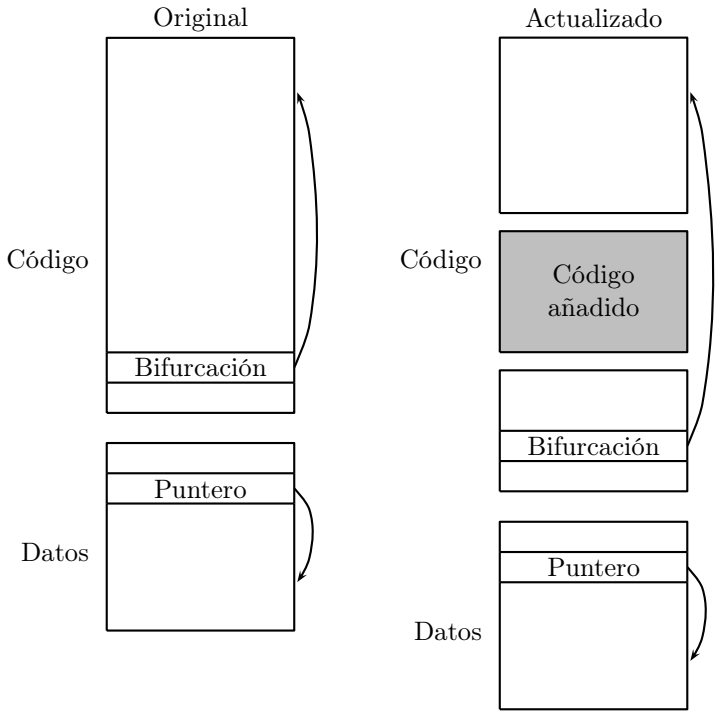


Figura 8.57: Cambios en archivos ejecutables tras la adición de nuevo código.

localizados. El material puede ser añadido, eliminado o movido, pero en general la mayor parte del texto sigue siendo el mismo. La situación es muy diferente con el código ejecutable (archivos binarios). El código ejecutable usa referencias relativas o absolutas a objetos (funciones, variables, etc), por lo que un cambio en el código fuente, tal como la adición de un nuevo objeto, o su reubicación, afectará a las referencias en las secciones del archivo que no se hayan cambiado explícitamente. Por ejemplo, si se añade nuevo código a un archivo binario, como en la Figura 8.57, todos los desplazamientos relativos que atraviesan la sección insertada probablemente cambiarán. Lo mismo ocurrirá con las referencias absolutas tras el punto de inserción. Como consecuencia de ello, un pequeño cambio en el programa de origen es probable que produzca cambios repartidos a lo largo de todo el archivo ejecutable.

Cuando se habla de compresión diferencial de código ejecutable, es útil hacer una distinción entre los dos tipos de diferencias entre archivos que se producen habitualmente:

- *Cambios principales.* Cambios que son causados por modificaciones del código fuente tales como la adición de una nueva función, la supresión de un objeto, o un cambio en la estructura de un bucle de control.
- *Cambios secundarios.* Los que hacen referencia a los cambios en los valores asumidos por los punteros, los desplazamientos, y las referencias introducidas como consecuencia de un cambio primario.

Debido a los cambios secundarios, secciones del archivo que no han sido explícitamente modificadas pueden no coincidir. Si asumimos que el compresor no tiene acceso al original ni a los programas fuente del objetivo, depende de un enfoque de fuerza bruta para desensamblar los archivos de referencia y objetivo. Los objetos correspondientes pueden coincidir, las referencias que han cambiado de una versión a otra pueden ser ajustadas, y, finalmente, puede utilizarse un algoritmo de diferenciación de archivos como VCDIFF o zdelta.

Esta solución es muy eficaz y permite la compresión de dos a cinco veces mejor que con los algoritmos de diferenciación de archivos más comunes. Sin embargo, tiene la desventaja de ser compleja y dependiente de la plataforma (o, dependiendo de la implementación, incluso del compilador y del enlazador). Por otra parte, desensamblar un archivo binario puede ser una tarea extremadamente dura, que se complica por instrucciones de longitud variable, optimizaciones del compilador, la intervención humana, y estructuras de datos mezcladas con las instrucciones.

Una solución más interesante fue propuesta por Brenda Baker, Udi Manber y Robert Muth en [Baker et al. 99]. Se utilizan dos algoritmos llamados *exediff* y *exepatch*. El primero genera un parche, dado un original y un código ejecutable actualizado. Éste último aplica el parche al código original con el fin de obtener la actualización.

Exediff utiliza una transformación con pérdidas para reducir el efecto de los cambios secundarios en el código ejecutable. Itera sobre dos operaciones denominadas *pre-matching* (pre-localización) y *value* (valor) hasta que el tamaño del parche converge a un mínimo y no puede reducirse más. Pre-matching se basa en una solución al algoritmo de subsecuencia común más larga (*Longest Common Subsequence algorithm* o LCS) descrita en [Cormen et al. 01]. La transformación con pérdidas se basa en el conocimiento detallado de la arquitectura, ya que implica la localización de las instrucciones y los punteros en el archivo binario. Exediff ha sido desarrollado y testeado en un microprocesador Alpha de 64 bits, sin embargo, el algoritmo se puede adaptar a otras arquitecturas. La implementación en otras arquitecturas puede ser enormemente complicada por la presencia de instrucciones de longitud variable. Sin embargo, exediff no asume el acceso al código fuente.

La transformación con pérdidas descrita en [Baker et al. 99] supone que el código que está siendo comprimido ha sido desarrollado por un DEC UNIX Alfa. Los ejecutables del DEC UNIX Alpha se almacenan en un formato de archivo objeto llamado ECOFF y se componen de tres secciones:

1. *Segmento de texto*. Que contiene las instrucciones y constantes de sólo lectura
2. *Segmento de datos*. Que contiene estructuras de datos sin inicializar.
3. *Segmento Bss*. Que contiene estructuras de datos inicializadas a cero.

La recuperación *pre-matching* y *value* se aplica sólo al segmento de datos y al código ejecutable en el segmento de texto. En [Baker et al. 99], se asume que todas las instrucciones en el segmento de texto tienen 64 bits de longitud y se componen de un *código de operando*, tres *registros*, y un valor *inmediato*: *opcode*, reg_1, reg_2, reg_3, e *inmediato*. los valores para reg_i indican el índice de un registro de propósito general o de un registro de propósito especial.

La transformación localiza las instrucciones máquina, inspecciona los tres registros, de reg_1 a reg_3, y sustituye el índice de los registros de propósito general con un valor predeterminado llamado ϵ. Los índices de los registros de propósito especial se dejan sin modificar. El operando *inmediato* es reemplazado por un flag que indica si su valor es negativo o no: Si *inmediato* es no negativo, se establece en POS; de lo contrario, se establece en NEG. Se efectúan operaciones similares sobre los punteros en el segmento de datos. Un puntero de 64 bits a una instrucción en el segmento de texto es sustituido por la palabra clave "$TEXT$" seguida por la operación hash de los tres primeros opcodes en el destino del puntero. Un puntero a un byte b en el segmento de datos es sustituido por la palabra clave "$DATA$", seguida por el valor de b. Punteros a otros objetos son sustituidos por la palabra clave "$OTHER$".

Mediante la sustitución de los ítems propensos a cambios secundarios con valores hash, flags y palabras clave, la transformada intenta eliminar el efecto de los cambios secundarios mientras preserva la información valiosa que puede ser usada en la realización de la búsqueda de emparejamiento del algoritmo de Subsecuencia Común más Larga. LCS alinea los archivos transformados en un intento de encontrar los cambios principales. Las subsecuencias comunes corresponden a los emparejamientos entre el original transformado y el archivo de actualización transformado. Las coincidencias se mantendrán en el mismo orden en los dos archivos y nunca se cruzarán.

LCS particiona los dos archivos en secciones como sigue:

1. *Emparejados e iguales.* Estas secciones representan código ejecutable que no ha sido cambiado entre el original y la actualización.

2. *Emparejados pero no iguales.* Tras la transformación con pérdidas, las secciones coincidentes en el original y en la actualización. Estas secciones coincidentes probablemente representan cambios secundarios. Algunas de las coincidencias serán falsas y serán eliminadas mediante el proceso iterativo.

3. *Secciones no emparejadas.* Representan código que ha cambiado entre las dos versiones. Las secciones no emparejadas corresponden a cambios primarios.

Las secciones coincidentes son reconstruidas por exepatch con el uso de comandos `COPY`. Las secciones no emparejadas consistirán en material insertado y se traducirán a instrucciones `ADD`. Las secciones emparejadas pero desiguales (debido a los cambios secundarios) serán copias, pero necesitarán un paso más con el fin de reflejar fielmente el contenido del archivo de actualización.

Exediff utiliza la recuperación *value* para predecir el contenido con pérdidas de los punteros transformados, registros, e immediatos. Exepatch realiza una predicción similar, por lo que la recuperación *value* debe basarse en información común para ambos, codificador y decodificador. Los valores que pueden ser predichos correctamente que denominan ítems de emparejamiento irrecuperables (*Unrecoverable Matching Items*) y se almacenan explícitamente en el parche. Los valores se recuperan en secuencia, comenzando desde los pequeños desplazamientos hasta los grandes en el archivo de actualización. Los valores recuperados se utilizan en las recuperaciones sucesivas para condicionar un efecto en cascada.

La recuperación *value* se basa en un conjunto de heurísticas encadenadas que dependen del conocimiento del dominio y de las características específicas del tipo de ítem que está siendo predicho (registro, puntero, valor inmediato, etc.. . .). Las heurísticas se aplican en una secuencia predeterminada conocida tanto por exediff como por exepatch.

En [Baker et al. 99] se describen los siguientes esquemas de recuperación:

1. *Valor emparejado.* El valor presenta una coincidencia total y no necesita una predicción adicional.

2. *Dirección trasladada.* El valor representa la dirección de un objeto dentro de un archivo y esa dirección puede calcularse mediante el uso del desplazamiento relativo de dos coincidencias. Por dirección entendemos cualquier tipo de referencia como un desplazamiento, el índice de una tabla, una posición del archivo, etc.

3. *Valor igual.* Un valor idéntico ha sido recuperado con éxito en un emparejamiento previo. El valor recuperado previamente puede ser reutilizado en el emparejamiento actual.

4. *Valor cercano.* El valor puede ser calculado a partir de un valor numéricamente cercano recuperado en un emparejamiento.

La naturaleza con pérdidas de la transformación utilizada en *pre-matching* puede generar emparejamientos espurios. Estas falsas coincidencias aumentan el tamaño del parche porque hacen que la recuperación *value* fracase. Los valores que no pueden ser recuperados tienen que ser enviados explícitamente en el parche como ítems de emparejamiento irrecuperables. Este problema se resuelve en [Baker et al. 99] mediante un método que efectúa un *pre-matching* iterativo, calcula el "beneficio" de cada coincidencia, elimina los emparejamientos que comprometen la compresión, y repite el proceso hasta que no se pueda conseguir una reducción adicional del tamaño del parche.

8.14.5. BSDiff

BSDiff es un algoritmo que aborda el problema de la compresión por diferenciación de archivos de código ejecutable manteniendo al mismo tiempo un enfoque independiente de la plataforma.

BSDiff fue creado por Colin Percival mientras trabajaba en FreeBSD Update como complemento a su investigación doctoral. Durante este período, Colin escribió varias versiones de BSDiff (hasta la versión 4.0) antes de darse cuenta, cuatro meses más tarde, de que era posible mejorar el algoritmo de emparejamiento de BSDiff y escribir una tesis doctoral sobre él y los problemas de coincidencia relacionados [Percival 06]. La versión del algoritmo descrito en la tesis (BSDiff 6) se basa en los mismos principios que BSDiff 4 pero utiliza un algoritmo de emparejamiento más sofisticado. A menos que se especifique lo contrario, en lo siguiente nos centramos en la descripción de BSDiff 4, ya BSDiff 6 aún no está disponible para el público.[11]

Aunque originalmente diseñado para resolver el problema de la actualización de seguridad binaria para FreeBSD de UNIX [Percival 03a], el más reciente lanzamiento de BSDiff 4 (versión 4.3) ha sido portado a varias plataformas y está disponible para FreeBSD, NetBSD, OpenBSD, Darwin, Debian, Gentoo, OS X, y Windows. BDFiff es utilizado actualmente por FreeBSD y OS X para distribuir actualizaciones de seguridad de binarios y, con algunas modificaciones, por el proyecto Mozilla para acelerar la descarga de las actualizaciones de Firefox.

El algoritmo que BSDiff 4 [Percival 03b] utiliza se basa en las siguientes observaciones:

1. Las regiones de código ejecutable afectadas por cambios secundarios (i.e., no involucradas en una modificación del programa fuente; Véase 8.14.4) presentarán diferencias dispersas. Los cambios secundarios constituyen sólo una pequeña parte del código compilado, y las referencias son probablemente cambios en sólo uno o dos bytes.

2. Puesto que los datos y el código se suelen desplazar en bloques, habrá muchas referencias afectadas por cambios secundarios que necesitan ser ajustados en la misma cantidad.

Cuando las secciones de los archivos original y de actualización hayan sido contrastados entre sí mediante un algoritmo de emparejamiento aproximado, las posiciones en las que las secciones emparejadas no coinciden serán dispersas y las diferencias de byte entre estas posiciones serán altamente comprimibles.

Bsdiff 4 escanea ambos archivos, el original y el de actualización, y construye un índice con una tabla hash o con un método similar. Luego, usando el índice, encuentra un conjunto de regiones que concuerdan exactamente. Estas regiones se extienden, ampliándose hacia delante y hacia atrás permitiendo discrepancias. El conjunto de las regiones aproximadamente emparejadas se corresponderá toscamente con los cambios secundarios y con las secciones de código no modificadas. Las regiones, en los archivos de actualización para los que no existe una coincidencia aproximada se corresponden con cambios primarios. Con este conjunto de coincidencias aproximadas, BSDiff 4 crea un archivo de revisión (parche) que consiste en:

1. Una sección de *control*, que contiene instrucciones `ADD` e `INSERT`. Cada instrucción `ADD` especifica un desplazamiento en el archivo original y el número de bytes del archivo original que se copia. Una instrucción `INSERT` especifica el número de bytes que tienen que ser insertados en el archivo de actualización. Los bytes insertados son almacenados juntos en otra sección.

2. Una sección de *diferencias*, que contiene las diferencias de byte entre emparejamientos aproximados.

3. Una sección que contiene los bytes que no formaban parte de los emparejamientos aproximados. Esos bytes serán insertados en el archivo de actualización mediante las instrucciones `INSERT` de la sección de control.

[11] Entiéndase: en el momento de escribir esta sección en el libro original, en inglés.

La concatenación de estas tres secciones es ligeramente más grande que el archivo de actualización. Sin embargo, la sección de control y las diferencias byte a byte son altamente comprimibles, por lo Bsdiff 4 utiliza `bzip2` para reducir eficazmente el tamaño del parche. `bzip2` es llamado de forma independiente para cada sección. Las tres secciones presentan estadísticas diferentes, y la compresión independiente a menudo proporciona una mejora sustancial sobre la compresión de todas las secciones juntas. Por la misma razón, los bytes que se agregan mediante la instrucción `INSERT` no son intercalados con los códigos de instrucciones en la sección de control sino que se agrupan en una sección separada.

> Birds of a feather compress better together[a].
>
> —Colin Percival, *Matching with Mismatches and Assorted Applications*[b], (2006)

[a] Aves de mismo plumaje casan mejor juntas.
[b] Emparejando con discordancias y diversas aplicaciones.

Un programa decodificador llamado *bspatch* descomprime el archivo de revisión (el parche) y aplica de forma secuencial las instrucciones contenidas en la sección de control. Para cada instrucción `ADD`, bspatch copia bytes del archivo original al de actualización. Las copias representan secciones que han sido emparejadas aproximadamente, por lo que bspatch debe recuperar una cantidad igual de bytes de la sección de diferencias del parche y añadirlos byte a byte a la sección copiada. Para cada `INSERT`, bspatch se limitará a recuperar el número especificado de bytes de la tercera sección del parche y los inserta en el archivo de actualización.

BSDiff 4 y exediff logran una compresión similar sobre un conjunto de archivos de referencia [Percival 03b]. Este resultado es singular ya que BSDiff 4 es independiente de la plataforma. Se consigue una compresión aún mejor mediante BSDiff 6, la versión descrita en [Percival 06]. Aunque confían en los mismos principios básicos de codificación de las ubicaciones de las regiones emparejadas aproximadamente y sus diferencias dispersas, BSDiff 6 utiliza un algoritmo de emparejamiento diferente y más sofisticado. En el mismo conjunto de archivos de referencia, BSDiff 6 mejora la compresión de BSDiff 4 en aproximadamente, de un 10 % a un 20 %.

El autor (Giovanni Motta) desea agradecer a Colin Percival sus comentarios sobre esta sección.

8.15. Compresión de datos hiperespectrales

(Esta sección está escrita conjuntamente con Giovanni Motta.)

Una imagen digital está compuesta de píxeles. En una imagen monocromática (binivel), un píxel puede tener uno de dos colores, por lo que se representa mediante un bit. En una imagen en color, un píxel representado por k bits puede tener uno de 2^k colores. Un valor típico es $k = 24$, donde un píxel puede tener aproximadamente 16,78 millones de colores. Los monitores de visualización de bajo costo actuales para computadoras personales pueden mostrar este número de colores. Aparentemente un número tan grande de colores en una sola imagen sería suficiente para cualquier propósito, pero la vida no es tan simple. Existe un gran campo (y creciente) de aplicaciones que requieren imágenes donde cada píxel está representado por cientos o incluso miles de bits. Un conjunto de datos tan grande, ya no se conoce como imagen, sino que se denomina *datos hiperespectrales*. A continuación se presentan algunos ejemplos de aplicaciones que requieren datos hiperespectrales.

1. *Satélites espía* (denominados oficialmente satélites de reconocimiento o *recon sats*). No es suficiente tener una cámara de alta resolución montada en un satélite, tomando fotografías, y transmitiéndolas a la Tierra. El enemigo puede ocultar fácilmente un tanque cubriéndolo con ramas y hojas. El enemigo puede inducirnos a error construyendo tanques, aviones y vehículos blindados de caucho inflable y plástico. Cuando una cámara toma una foto, registra la luz visible reflejada por los objetos

que ve. Con el fin de distinguir entre la luz reflejada de acero y la luz reflejada del caucho (o entre la luz reflejada de ramas de árboles normales y la luz reflejada de ramas de árboles colocadas sobre un tanque), la cámara en un satélite espía tiene que mirar al suelo con algo más que la luz visible. Tiene que medir la radiación reflejada desde cada punto del suelo en muchas longitudes de onda.

Una longitud de onda es un número real, no es un entero, por lo que en principio existe un número infinito de longitudes de onda, incluso en una parte muy estrecha del espectro electromagnético. Naturalmente, una cámara práctica no puede medir la intensidad de la reflexión de la radiación en la precisa longitud de onda de 423,708235 nm. Puede medir la radiación en, digamos, el estrecho rango de frecuencias, de 420–430 nm, y tal rango se denomina *banda*. Una cámara espía típica consta de un conjunto de sensores sensibles que pueden medir y registrar la radiación en tal vez 250 bandas de frecuencia, normalmente localizadas entre 400 nm y 2 400 nm. Ésto incluye el rango visible (400–700 nm) y parte del rango infrarrojo del espectro electromagnético y puede contener información que es mucho más útil que sólo la luz visible. (El rango infrarrojo es muy amplio e incluye longitudes de onda de hasta 1 mm.)

Como ejemplo, el sensor AVIRIS (*airborne visible/infrared imaging spectrometer* o espectrómetro de imagen visible/infrarroja aerotransportado) consta de tres sensores de 64 bandas de frecuencias cada una, más un cuarto sensor con 32 bandas, haciendo un total de 224 bandas. Otros sensores tienen diferentes números de bandas cuyo rango oscila de cientos a miles.

El ojo no puede percibir la radiación infrarroja (véase la discusión sobre la visión humana en página 348), por lo que esa imagen debe ser pintada con colores artificiales. Una persona entrenada mirando una imagen de alta resolución con colores artificiales puede decir de inmediato la diferencia entre un tanque de caucho y un tanque de acero, entre ramas de árboles reales y ramas que ocultan un lanzacohetes. Si se requiere un análisis más detallado, pueden visualizarse y pintarse artificialmente, en un momento determinado, sólo unas pocas bandas .

En consecuencia, cada "píxel" en la imagen tomada por una cámara consta de 250 números, cada uno un entero de al menos 16 bits. La resolución de la cámara (el tamaño de un píxel en el suelo) debe ser alta. La resolución de los satélites espías modernos se mantiene en secreto, pero se conjetura que está en torno a unos 10 cm. Como resultado, la cantidad de datos recogidos por un satélite espía para un kilómetro cuadrado es de $10\,000^2 \times 250 \times 2 = 5 \times 10^{10}$ bytes; enorme, pero ¡fundamental para la vigilancia del campo de batalla con éxito!

2. *Detección remota*. Este término se refiere a la utilización de un sensor para recoger remotamente datos acerca de un determinado entorno. Los datos pueden ser ópticos (intensidades de reflexión en muchas longitudes de onda), acústicos (la intensidad de ondas sonoras en varias frecuencias), o concentraciones de sustancias químicas. Los siguientes son ejemplos típicos de sensores remotos:

- El radar se utiliza para medir el rango y la velocidad de objetivos bien definidos, tales como una aeronave u objetivos confusos tales como una nube de vapor de agua.
- Los altímetros y radares (ambos microondas y láser) montados en satélites se utilizan para mapear toda la Tierra (a una alta precisión), así como características en el fondo del mar (a una precisión mucho menor).
- El Lidar, los radiómetros, y los fotómetros miden las concentraciones de diversas sustancias químicas. Esta aplicación proporciona información importante sobre las concentraciones de sustancias químicas en la atmósfera.
- Las compañías de petróleo y minerales están constantemente buscando nuevos yacimientos. Ellas montan sensores en satélites de observación terrestre y miden la luz solar reflejada por grandes regiones de la Tierra en un intento de localizar recursos naturales.
- Los oceanógrafos usan el sonar para "otear" bajo el agua profunda en búsqueda de objetos interesantes, especies marinas y cambios en el fondo del mar. Ésta es una aplicación hiperespectral,

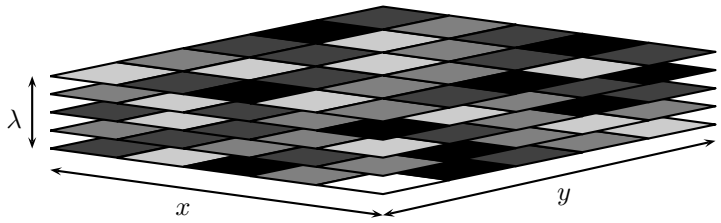


Figura 8.58: Organización de datos hiperespectrales.

porque en el agua penetran distintas frecuencias de sonido y son reflejadas por los objetos de diferentes maneras.

- Los geólogos utilizan sismómetros para "ver" el interior de la tierra, ya sea para localizar recursos, ya sea para estudiar formaciones geológicas. El principio consiste en crear ondas acústicas subterráneas con una explosión, y tiene sentido generar varias explosiones con distintas potencias y medir las ondas acústicas creadas por cada una.
- Imágenes médicas. Además de los rayos X y la resonancia magnética, la medicina moderna puede ver el interior de nuestros cuerpos mediante la ecografía. Las ondas sonoras de alta frecuencia (típicamente de 2 a 10 MHz) se dirigen dentro de un área de interés en el cuerpo y son reflejadas por los tejidos de formas diferentes para producir una imagen en un monitor de visualización. Este tipo de imágenes se utiliza a menudo para visualizar el feto durante el embarazo.
- Muchos científicos están interesados en cierta información, tal como la salud de los cultivos en muchas áreas o la propagación de plancton en los océanos. Tal información puede formar parte de un estudio de amplio rango sobre los cambios climáticos y el calentamiento global.

En todos estos casos, el resultado es una gran cantidad de datos organizados en tres dimensiones. Dos dimensiones son espaciales (cada píxel tiene una ubicación y es identificado mediante un par de coordenadas (x, y)), y la tercera dimensión es espectral (las bandas de un píxel, normalmente indexadas por λ o por t). Una imagen tiene una única resolución, pero los datos hiperespectrales tienen tanto una resolución espacial (el tamaño de un píxel) como una resolución espectral (el número de bandas espectrales de un píxel). Datos con dos bandas espectrales por píxel se denominan banda dual. Datos con tres o varias (quizás 8–10) bandas se denominan multiespectrales, y datos con más bandas, hiperespectrales. La Figura 8.58 muestra una organización típica de los datos hiperespectrales. cada plano es una banda, y consta de filas y columnas de píxeles. Las unidades más pequeñas se denominan píxeles a pesar de que no siempre son unidades visuales.

Podemos interpretar cada plano de la Figura 8.58 como una imagen (píxeles mostrados en relación espacial entre sí) y cada columna como un espectro (variaciones entre los píxeles vistas como una función de la longitud de onda). Una columna también puede ser considerada un punto en un espacio n-dimensional.

Antes de hablar acerca de la compresión de los datos hiperespectrales, he aquí algunas palabras sobre su procesamiento. La mayoría de los algoritmos para el procesamiento de datos hiperespectrales se refieren a detección de anomalías o la identificación de materiales, pero el primer paso en el procesamiento es eliminar los efectos de la variación de la iluminación a partir de un bloque de datos. Un satélite puede recopilar datos sobre una región determinada durante varias horas, y la luz reflejada desde el suelo durante este tiempo varía a medida que el sol se mueve a través del cielo. Esta variación debe ser eliminada antes de poder efectuar cualquier otro procesamiento.

Debido al enorme tamaño de un bloque de datos hiperespectrales, la compresión es importante. Una visión general de varios métodos para la compresión de datos hiperespectrales puede encontrarse

en [Motta et al. 06]. Los datos pueden ser comprimidos, ya sea en la fuente (el satélite, sismómetro, u otro sensor) o en el receptor (llamado *sink*, el ordenador al que se envían y donde son almacenados y procesados). Un satélite puede estar fuera de contacto con las estaciones terrestres varias veces durante cada órbita, por lo que pueden gastar estos tiempos comprimiendo datos a bordo. Cuando la comunicación se reanuda, el satélite puede parar la compresión y empezar a enviar los datos comprimidos a la Tierra. Un sismómetro, por otra parte, debe ser portátil, ligero y de bajo costo, por lo que tiene sentido que sólo transmita sus datos sin procesar (en formato *raw*) tan pronto como sean recogidos, a un equipo remoto en el que serán almacenados, comprimidos, y procesados.

La Figura 8.58 muestra que la compresión de los datos hiperespectrales es similar a la compresión de imágenes con la característica añadida de la correlación inter-banda. Una imagen puede ser considerada datos hiperespectrales con una banda, y la compresión de una imagen explotar la correlación de los píxeles en esta banda (correlación intra-banda). Cuando existen varias bandas, un método de compresión también debe aprovechar la correlación de píxeles entre bandas vecinas (correlación inter-banda). Es bien sabido que en una imagen en color existe una correlación entre los planos de color. La diferencia entre una imagen y los datos hiperespectrales es que las bandas en el último tipo son estrechas, por lo que la correlación es mayor en el dominio espectral que en el dominio espacial.

Más a menudo que no, la compresión debe ser sin pérdidas, ya que algunas aplicaciones (especialmente de espionaje y médicas) pueden depender crucialmente de los valores de unos pocos píxeles individuales. De hecho, los ítems de datos importantes pueden a veces ser más pequeños que un píxel. La compresión con pérdida de los datos hiperespectrales puede tener sentido en los casos en que se envían estos datos para una evaluación inicial. A veces, un comprador potencial puede querer examinar un bloque de datos antes de comprarlo, y puede que no le importe mirar datos comprimidos con pérdidas. Alternativamente, algunos clientes pueden ser atraídos por el bajo costo de los datos hiperespectrales comprimidos con pérdidas. Cuando se elige la compresión con pérdida para los datos hiperespectrales, es importante desarrollar una medida de la distorsión adecuada que garantice la calidad de los datos comprimidos. El principio rector es que la aplicación de cualquier algoritmo de procesamiento a datos con pérdidas y datos sin pérdidas produce resultados que difieren sólo en los detalles, no en su características principales.

Dado un bloque de datos hiperespectrales para ser comprimido, denotamos un píxel genérico antes de la compresión por $B(x,y,t)$ y el mismo píxel tras la compresión con pérdida y la descompresión por $\hat{B}(x,y,t)$. La medida de distorsión más simple es la distorsión absoluta máxima (MAD o *maximum absolute distortion*). Ésta garantiza que el valor absoluto de la diferencia $B(x,y,t) - \hat{B}(x,y,t)$ no exceda de un cierto parámetro distancia d. La desventaja de esta medida de distorsión es que los píxeles pequeños pueden volverse relativamente más distorsionados que los grandes; MAD no tiene en cuenta los valores absolutos de los píxeles individuales.

Una mejor medida de la distorsión es el porcentaje de distorsión absoluta máxima (PMAD o *percentage maximum absolute distortion*), donde la diferencia absoluta $\left|B(x,y,t) - \hat{B}(x,y,t)\right|$ siempre se mantiene en o por debajo de $p \times B(x,,y,t)$, donde el parámetro porcentaje p está en el intervalo $[0,1]$. Otras medidas también son posibles.

Dada una medida de distorsión adecuada, la Figura 8.59a es un diagrama de bloques de los principales componentes de un codificador hiperespectral con pérdidas de datos. El compresor en sí mismo puede ser cualquier algoritmo estándar —tal como una transformación, cuantificación vectorial, o predicción— que tiene que ser extendido para tomar ventaja de las correlaciones tridimensionales en los datos. La característica más importante del codificador es la retroalimentación de la información desde el compresor hasta la medida de la distorsión. El último componente examina la información y decide si necesita una mayor compresión. La etapa de preprocesamiento es opcional. Involucra un proceso sencillo, tal como el reordenamiento de bandas, que mejora la compresión principal que le sigue. El preprocesamiento debe ser reversible, puesto que el decodificador (Figura 8.59b) tiene que llevar a cabo un procesamiento posterior. Observe cómo se envía la información adicional al decodificador

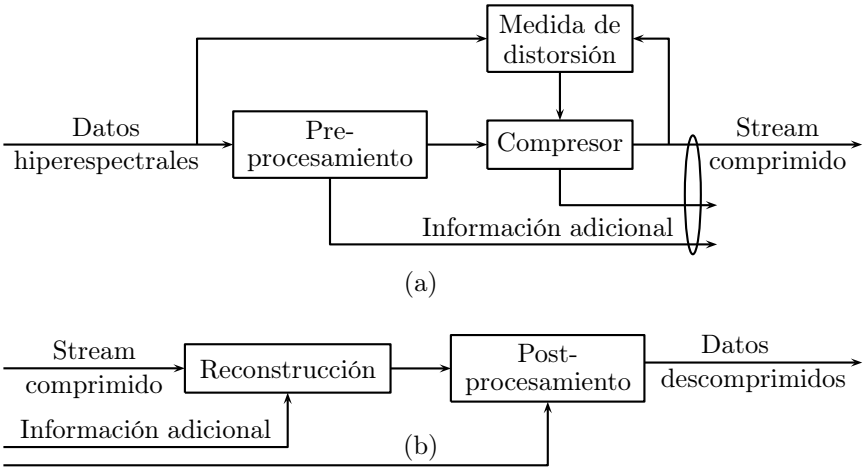


Figura 8.59: Organización de un codificador de datos hiperespectrales.

entrelazándola con los datos comprimidos en el stream de salida.

El resto de esta sección trata sobre las extensiones de varias técnicas de compresión estándar a datos hiperespectrales.

8.15.1. Métodos predictivos

En un método predictivo, el codificador predice los ítems de datos actuales x a partir de varios de sus predecesores. La predicción $\hat{x}$ se resta de x para formar un residuo e, que es luego codificado con un codificador de entropía. Si la predicción se realiza correctamente y si los elementos de datos individuales están correlacionados, los residuos $e = x - \hat{x}$ serán normalmente enteros pequeños que son fáciles de codificar mediante códigos de tamaño variable. Observe que sólo pueden ser usados en la predicción los predecesores de x, debido a que el decodificador debe realizar la misma predicción y tiene acceso sólo a aquellos elementos de datos que ya han sido decodificados (los predecesores).

Muchos métodos de compresión de audio sin pérdidas utilizan este enfoque. *Shorten* (Sección 7.9) emplea predictores lineales de onden n en su entrada (una secuencia unidimensional de muestras de audio) para predecir la muestra actual $s(t)$. Los primeros cuatro predictores lineales, correspondientes a n valores entre 0 y 3, vienen dados por la Ecuación (7.5), duplicada aquí.

$$\begin{aligned}
\hat{s}_0(t) &= 0,\\
\hat{s}_1(t) &= s(t-1),\\
\hat{s}_2(t) &= 2s(t-1) - s(t-2),\\
\hat{s}_3(t) &= 3s(t-1) - 3s(t-2) + s(t-3).
\end{aligned} \tag{7.5}$$

Cuando se aplica un método de predicción a la compresión de imágenes, éste predice un píxel mediante el cálculo de una suma ponderada de sus vecinos cercanos, asignando más peso a los vecinos cercanos. La Figura 4.70 muestra cómo JPEG-LS (Sección 4.9) predice un píxel x a partir de los tres vecinos c, b, y d por encima de él y del vecino a de su izquierda. Estos vecinos serán conocidos por el decodificador cuando éste obtenga la decodificación de x. El método de ponderación del árbol de contexto (Sección 4.27) predice de manera similar un píxel X a partir de los mismos cuatro vecinos (Figura 4.140) mediante una suma ponderada. Si consideramos una imagen como un arreglo

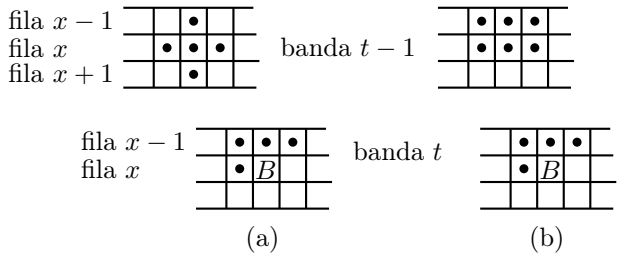


Figura 8.60: Dos predicciones alternativas de datos hiperespectrales.

(array) bidimensional de píxeles dispuestos en filas y columnas, podemos expresar tal predicción con la notación:

$$\hat{x}\,[i,j] = a\,x\,[i-1,j-1] + b\,x\,[i-1,j] + c\,x\,[i-1,j+1] + d\,x\,[i,j-1]\,, \tag{8.5}$$

donde los cuatro pesos, a a d, deben sumar 1.

Ahora está claro cómo la predicción lineal puede extenderse a datos hiperespectrales. Un bloque de datos es considerado una caja rectangular tridimensional, donde cada píxel $B\,[x,y,t]$ tiene tres coordenadas y es predicho mediante los píxeles por encima del mismo, a su izquierda, y cercanos a él en la banda precedente (banda $t-1$). Si el codificador escanea los datos hiperespectrales banda por banda, entonces la Ecuación (8.5) se puede extender a:

$$\begin{aligned}\hat{B}\,[x,y,t] = (&B\,[x-1,y-1,t] + B\,[x,y-1,t] + B\,[x+1,y-1,t] + B\,[x-1,y,t]\\ &+ B\,[x,y-1,t-1] + B\,[x-1,y,t-1]\\ &+ B\,[x,y,t-1] + B\,[x+1,y,t-1] + B\,[x,y+1,t-1])/9\end{aligned}$$

(Figura 8.60a, donde cada píxel recibe un peso de $1/9$). Tenga en cuenta que cualquier píxel en la banda $t-1$ es conocido por el decodificador y, por lo tanto, puede ser utilizado para la predicción. Una alternativa es que el codificador escanee los datos fila por fila, y para cada fila, banda por banda. En tal caso, no todos los píxeles de la banda $t-1$ están disponibles para la predicción, y la Ecuación (8.5) puede extenderse a:

$$\begin{aligned}\hat{B}\,[x,y,t] = (&B\,[x-1,y-1,t] + B\,[x,y-1,t] + B\,[x+1,y-1,t] + B\,[x-1,y,t]\\ &+ B\,[x-1,y-1,t-1] + +B\,[x,y-1,t-1] + B\,[x-1,y-1,t-1]\\ &+ B\,[x-1,y,t-1] + B\,[x,y,t-1] + B\,[x+1,y,t-1])/10\end{aligned}$$

(Figura 8.60b, donde cada píxel recibe un peso de $1/10$). También es posible asignar diferentes pesos a los píxeles utilizados en la predicción. Los pesos deben sumar la unidad y debe reflejar la distancia de un píxel de predicción al píxel predicho $B\,[x,y,t]$.

Un ejemplo práctico de este enfoque a la compresión de datos hiperespectrales es el algoritmo tridimensional de modulación de pulsos codificados diferencial y adaptativo (ADPCM o *adaptive differential pulse code modulation*) descrito en [Roger y Cavenor 96]. Los dos autores han tratado de 25 configuraciones de píxeles para la predicción y han seleccionado experimentalmente las cinco que figuran en la Tabla 8.61.

Los prefijos SA, SE, y SS se refieren a predictores espaciales, espectrales, y mixtos, respectivamente. Los tres últimos predictores utilizan parámetros que se han obtenido mediante la minimización de la varianza de los errores de predicción en cada fila de píxeles. Suponiendo que hay n píxeles en cada fila, la varianza de la fila x es la suma:

$$\sum_{y=1}^{n} \Big(\hat{B}\,[x,y,t] - B\,[x,y,t]\Big)^2\,.$$

Nombre	Píxeles utilizados
SA-2RC	$(B[x-1,y,t]+B[x,y-1,t])/2$
SS-1	$B[x,y-1,t]+B[x,y,t-1]-B[x,y-1,t-1]$
SE-01B	$a+b\,B[x,y,t-1]$
SE-02B	$a+b\,B[x,y,t-1]+c\,B[x,y,t-2]$
SS-01	$a+b\,B[x,y-1,t]+c\,B[x,y,t-1]+d\,B[x,y-1,t-1]$

Tabla 8.61: Cinco configuraciones de predicción para datos hiperespectrales.

Esta cantidad depende de los valores de a, b, c, y d, y el conjunto de cuatro parámetros que genere la suma más pequeña (a medida que son computados por la minimización de mínimos cuadrados) es eligido para predicción de píxel de la fila x. Este conjunto también es enviado al decodificador como información adicional, y se calcula otro conjunto para la siguiente fila.

Los residuos resultantes de esta predicción son codificados mediante códigos de Rice. Este tipo de código fue seleccionado experimentalmente por los desarrolladores de ADPCM como el que produce los mejores resultados.

8.15.2. DCT tridimensional

La Sección 4.6 es una discusión detallada de la transformada discreta del coseno (DCT). En ella se explican los orígenes de esta importante transformación y muestra cómo se puede aplicar a la compresión de imágenes digitales. Incluso un rápido vistazo a la Ecuación (4.15) muestra que la DCT es bidimensional, lo que sugiere inmediatamente la posibilidad de ampliarla a tres dimensiones y aplicarla a la compresión de los datos hiperespectrales. La extensión es sencilla. La DCT directa en tres dimensiones (3DCT) se convierte en:

$$G_{ijk}=\sqrt{\frac{2^3}{n^3}}C_iC_jC_k\sum_{x=0}^{n-1}\sum_{y=0}^{n-1}\sum_{z=0}^{n-1}p_{xyz}\cos\left[\frac{(2x+1)\,i\pi}{2n}\right]\cos\left[\frac{(2y+1)\,j\pi}{2n}\right]\cos\left[\frac{(2z+1)\,k\pi}{2n}\right], \tag{8.6}$$

para $0\le i,j,k\le n-1$ y para

$$C_f=\begin{cases}\frac{1}{\sqrt{2}}, & f=0\\ 1\ , & f>0\end{cases}.$$

Y la DCT tridimensional inversa es:

$$p_{xyz}=\sqrt{\frac{2^3}{n^3}}\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1}C_iC_jC_kG_{ijk}\cos\left[\frac{(2x+1)\,i\pi}{2n}\right]\cos\left[\frac{(2y+1)\,j\pi}{2n}\right]\cos\left[\frac{(2z+1)\,k\pi}{2n}\right], \tag{8.7}$$

para $0\le x,y,z\le n-1$.

A aquéllos familiarizados con los principios de JPEG (Sección 4.8) les será más fácil visualizar su extensión a datos tridimensionales. Simplemente se divide un gran conjunto de datos hiperespectrales en cubos de $8\times8\times8$ píxeles cada uno, se aplica la 3DCT a cada cubo, se recogen los coeficientes de transformada resultantes en una secuencia en zigzag, se cuantifican, y se codifican los resultados con un codificador de entropía, como el código de Huffman. Ésto funciona, pero hay dos puntos que deben ser tomados en consideración.

```
(* DCT 3D para datos hiperespectrales *)
Clear[Pixl, DCT];
Cr[i_]:=If[i==0,1/Sqrt[2],1];
DCT[i_,j_,k_]:=(Sqrt[2]/32) Cr[i]Cr[j]Cr[k]Sum[Pixl[[x+1,y+1,z+1]]
Cos[(2x+1)i Pi/8]Cos[(2y+1)j Pi/8]Cos[(2z+1)k Pi/8],
{x, 0, 3}, {y, 0, 3}, {z, 0, 3}];
Pixl = Table[Random[Integer, {30, 60}],{4},{4},{4}];
Table[Round[DCT[m,n,p]],{m,0,3},{n,0,3},{p,0,3}];
MatrixForm[%]
```

Figura 8.62: DCT tridimensional aplicada a datos correlacionados.

El primer punto tiene que ver con la correlación entre las bandas. Dependiendo de la naturaleza de los datos hiperespectrales, el usuario puede conocer o sospechar que la correlación inter-banda es más débil que la correlación intra-banda. En tal caso, los cubos que están siendo transformados y cuantificados pueden contener un menor número de bandas que las dimensiones espaciales y convertirse en cajas rectangulares. Tales cajas pueden, por ejemplo, tener ocho dimensiones espaciales pero sólo cuatro dimensiones espectrales. Las Ecuaciones (8.6) y (8.7) deben modificarse en consecuencia. La suma en z debe ir de 0 a 3, los cosenos deben tener, ya sea 2×8, ya sea 2×4 en su denominadores, y la constante debe ser $\sqrt{2^3/8\times8\times4} \approx 0{,}177$.

El segundo punto tiene que ver con la secuencia en zigzag. La Figura 8.62 muestra un sencillo código en *Mathematica* que calcula la 3DCT de un bloque de $4 \times 4 \times 4$ números enteros aleatorios. La constante se convierte en $\sqrt{2^3/4\times4\times4} \approx 0{,}3536$.

Se realizaron tres tests, con los datos aleatorios de los intervalos $[30, 60]$, $[30, 160]$, y $[30, 260]$. La constante ha sido establecida de tal manera que el coeficiente DC sea la media de los ítems de datos aleatorios. Estos conjuntos de datos aleatorios exhiben cada vez menos correlación a medida que aumentan las diferencias. Los cubos de $4 \times 4 \times 4$ de coeficientes de transformada resultantes (redondeados al entero más cercano y con barras sobre los números de signo negativo) se muestran en la Figura 8.63.

$$\begin{bmatrix} 46 & 0 & 0 & \overline{2} \\ \overline{3} & 1 & \overline{2} & 0 \\ 2 & 0 & 0 & 0 \\ 0 & \overline{1} & 0 & 3 \end{bmatrix} \quad \begin{bmatrix} \overline{1} & 0 & 1 & \overline{1} \\ 2 & \overline{2} & 0 & 1 \\ 1 & 0 & 1 & 0 \\ \overline{1} & 1 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 0 & \overline{1} \\ 0 & 0 & 0 & 0 \\ \overline{1} & \overline{2} & 0 & 1 \\ 1 & 2 & 0 & \overline{1} \end{bmatrix} \quad \begin{bmatrix} \overline{1} & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & \overline{1} \\ 0 & 2 & 2 & \overline{1} \end{bmatrix}.$$

$$\begin{bmatrix} 95 & 8 & 0 & \overline{9} \\ 4 & 0 & 5 & 0 \\ \overline{3} & \overline{1} & 1 & 3 \\ \overline{2} & 4 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} \overline{6} & 3 & 5 & \overline{4} \\ 3 & \overline{2} & 1 & 10 \\ 1 & \overline{9} & \overline{6} & \overline{4} \\ \overline{6} & \overline{5} & 1 & 7 \end{bmatrix} \quad \begin{bmatrix} \overline{2} & 1 & 6 & 1 \\ \overline{6} & 4 & \overline{2} & 0 \\ 5 & \overline{1} & \overline{1} & \overline{3} \\ 6 & \overline{3} & 4 & \overline{2} \end{bmatrix} \quad \begin{bmatrix} \overline{4} & 2 & \overline{2} & 1 \\ \overline{3} & 13 & \overline{5} & \overline{1} \\ \overline{2} & 4 & 4 & \overline{12} \\ 1 & 10 & 4 & 0 \end{bmatrix}.$$

$$\begin{bmatrix} 137 & \overline{5} & \overline{14} & 8 \\ \overline{21} & 10 & 1 & 17 \\ 3 & \overline{8} & \overline{6} & \overline{13} \\ 4 & \overline{4} & 0 & \overline{10} \end{bmatrix} \quad \begin{bmatrix} \overline{11} & 10 & \overline{10} & 3 \\ 4 & 14 & 5 & 3 \\ \overline{6} & 3 & 8 & \overline{18} \\ \overline{6} & 8 & 2 & 12 \end{bmatrix} \quad \begin{bmatrix} 14 & 16 & 2 & 4 \\ 7 & \overline{11} & 12 & 8 \\ \overline{11} & \overline{3} & 9 & \overline{5} \\ 1 & 2 & 12 & \overline{4} \end{bmatrix} \quad \begin{bmatrix} 3 & 8 & \overline{9} & \overline{9} \\ \overline{1} & 7 & 11 & 18 \\ \overline{4} & \overline{7} & 1 & 8 \\ \overline{7} & 3 & 3 & \overline{1} \end{bmatrix}.$$

Figura 8.63: Coeficientes de la DCT tridimensional.

En los tres tests, es obvio que los coeficientes de transformada resultantes se hacen más grandes a medida que los números aleatorios se desvían de su promedio y, en consecuencia, están menos correlacionados. Además, la secuencia en zigzag para la 3DCT es más compleja que en la DCT bidimensional.

Los coeficientes en torno a la esquina superior izquierda de cada uno de los cuatro planos son grandes y deberían ser recogidos en primer lugar. Además, los coeficientes tienden a ser más pequeños al pasar de un plano a otro lejos del coeficiente DC. Como resultado, una secuencia tal como la que se muestra aquí puede tener sentido:

$$\begin{bmatrix} 1 & 2 & 6 & 7 \\ 3 & 5 & 8 & 23 \\ 4 & 9 & 22 & 53 \\ 10 & 21 & 54 & 61 \end{bmatrix} \quad \begin{bmatrix} 11 & 12 & 16 & 24 \\ 13 & 15 & 25 & 30 \\ 14 & 26 & 29 & 56 \\ 27 & 28 & 55 & 62 \end{bmatrix} \quad \begin{bmatrix} 17 & 18 & 33 & 34 \\ 19 & 32 & 35 & 40 \\ 31 & 36 & 39 & 57 \\ 37 & 38 & 58 & 63 \end{bmatrix} \quad \begin{bmatrix} 20 & 41 & 45 & 46 \\ 42 & 44 & 47 & 52 \\ 43 & 48 & 51 & 59 \\ 49 & 50 & 60 & 64 \end{bmatrix}.$$

Otra posibilidad es el patrón en zigzag tradicional, donde cada movimiento se repite en todas las bandas:

$$\begin{bmatrix} 1 & 5 & 21 & 25 \\ 9 & 17 & 29 & 49 \\ 13 & 33 & 45 & 53 \\ 37 & 41 & 57 & 61 \end{bmatrix} \quad \begin{bmatrix} 2 & 6 & 22 & 26 \\ 10 & 18 & 30 & 50 \\ 14 & 34 & 46 & 54 \\ 38 & 42 & 58 & 62 \end{bmatrix} \quad \begin{bmatrix} 3 & 7 & 23 & 27 \\ 11 & 19 & 31 & 51 \\ 15 & 35 & 47 & 55 \\ 39 & 43 & 59 & 63 \end{bmatrix} \quad \begin{bmatrix} 4 & 8 & 24 & 28 \\ 12 & 20 & 32 & 52 \\ 16 & 36 & 48 & 56 \\ 40 & 44 & 60 & 64 \end{bmatrix}.$$

Este enfoque a la compresión de datos hiperespectrales es una extensión directa de los conceptos inherentes a JPEG. Una aplicación similar de esta transformación ha sido propuesta por [Li y Furht 03] para la compresión de imágenes bidimensionales. Una implementación VLSI de bajo consumo de la 3DCT ha sido propuesta por [Saponara et al. 03], y un rápido algoritmo para esta transformación ha sido desarrollada por [Boussakta y Alshibami 04].

Otras variaciones en este enfoque también son posibles y han sido probadas. Glen Abousleman ha desarrollado una combinación de 3DCT seguida de una cuantificación de codificación de enrejado (TCQ o *trellis-coded-quantization*) específicamente para comprimir imágenes hiperespectrales tomadas por satélites. Los datos hiperespectrales se dividen en cubos de $8 \times 8 \times 8$ píxeles cada uno, la 3DCT es aplicada a cada cubo, lo que resulta en un cubo de coeficientes de transformada. Los coeficientes ubicados en la misma posición en cada cubo (coeficientes semejantes o *like-coefficients*) son recolectados en una secuencia y cada secuencia es entonces codificada usando la TCQ. El número de secuencias es de $8^3 = 512$, y cada secuencia consta de $n/512$ píxeles, donde n es el tamaño total de los datos hiperespectrales. Detalles del método TCQ y su diseño de libro de códigos pueden encontrarse en [Abousleman 06].

8.15.3. Cuantificación vectorial

Al igual que la transformada y los métodos predictivos, la cuantificación vectorial también puede ser aplicada a la compresión de datos hiperespectrales. Puesto que los datos hiperespectrales muestran una fuerte correlación en el dominio espectral, una elección natural es considerar cada píxel como un vector al que se aplica una cuantificación. Una vez que el vector ha sido cuantificado y sustituido por un índice, se puede utilizar un método de compresión que explote la correlación espacial para reducir aún más el tamaño de esta representación.

Dos temas principales impiden una aplicación directa de esta idea. A saber:

- La cuantificación vectorial es un método con pérdidas. Los datos hiperespectrales son recolectados con un gran costo, por lo que es necesario preservar su contenido de información completo. Las aplicaciones que utilizan estos datos deben confiar en la calidad más alta posible. Las pérdidas introducidas por un cuantificador vectorial tienen naturaleza estadística. La información que es estadísticamente rara es descartada en favor de los vectores que aparecen a menudo. Sin embargo, aplicaciones tales como la *detección de objetivos* tienen como propósito la localización

de objetos con una firma espectral rara y sucede a menudo que el objeto que el algoritmo quiere localizar tiene dimensiones físicas de sub-píxel. Por ejemplo, si un tanque militar en un campo de batalla es representado por un sensor en el que un píxel cubre un área de 20×20 metros, el tanque ocupará sólo una fracción de un píxel. La firma espectral del tanque será (linealmente) combinada con la firma de tierra. Puesto que el suelo es estadísticamente dominante, un cuantificador vectorial lo codificará bien, pero perderá la firma del tanque.

- Un píxel hiperespectral puede tener desde cientos a miles de componentes. Se espera que la dimensión de cada píxel crezca con la siguiente generación de sensores. El tamaño de los datos crece linealmente con la resolución espectral y cuadráticamente con la resolución espacial, por lo que que el aumento de la resolución espectral es la primera opción a la hora de diseñar un nuevo generador de imágenes. El diseño de un cuantificador vectorial de vectores dimensionalmente altos es muy exigente y puede convertirse rápidamente en computacionalmente imposible. Además, los cuantificadores vectoriales se convierten más y más ineficientes a medida que el vector de dimensión aumenta [Kendall 61].

Un cuantificador vectorial puede utilizarse para realizar una compresión sin pérdidas si el error de cuantificación es codificado en el flujo de bits (*bitstream*) junto con los índices de cuantificación. Sin embargo, puesto que el error de cuantificación es un vector que tiene el mismo tamaño que la entrada, el primer problema puede ser resuelto con este método solamente si el vector que es descompuesto en el índice de cuantificación y el error pueden codificarse de forma más compacta que la entrada original.

Un cuantificador vectorial puede simplificarse estructurando el libro de códigos. Los métodos tradicionales utilizan *árboles*, *particionamiento*, *codificación residual* y *trellises* (teselados). Un cuantificador vectorial estructurado sacrifica algo de compresión (o de calidad, para un ratio de compresión dado) a favor de la velocidad. Un libro de códigos estructurado puede ser más fácil para el diseño y la búsqueda.

Dado que se espera que las firmas espectrales aumenten en el futuro, un cuantificador vectorial particionado ofrece la ventaja de la escalabilidad. En un cuantificador vectorial particionado, los vectores de entrada se subdividen en un número fijo de sub-vectores, cada uno cuantificado de forma independiente. Los vectores más largos se pueden acomodar fácilmente al incrementar el número de particiones, ya que la complejidad de diseño y búsqueda crece linealmente con el número de particiones.

Los sub-vectores se cuantifican de forma independiente, por lo que es claro que el particionamiento no puede ser mejor que la cuantificación del vector completo. Un cuantificador vectorial particionado no toma ventaja de la correlación que existe entre las particiones de un mismo vector. También está la cuestión de decidir cuán amplia debe ser cada partición. El particionamiento de un vector de firma espectral en particiones del mismo tamaño sería muy inferior al óptimo debido al hecho de que los componentes del vector tienen estadísticas muy diferentes, con media, varianza y rango variando salvajemente.

El cuantificador vectorial de particionado localmente óptimo (LPVQ o *Locally Optimal Partitioned Vector Quantizer*) introducido por Giovanni Motta, Francesco Rizzo, y James Storer [Motta et al. 06] aborda estas dos cuestiones explotando la correlación entre los índices de cuantificación y determinando en tiempo de diseño la partición (localmente) óptima de las firmas espectrales para un conjunto de datos dado.

La Figura 8.64 muestra el codificador LPVQ. Una firma espectral de entrada $B(x, y)$ es particionada en P sub-vectores disjuntos que tienen límites $b_0, b_1, \ldots, b_{P-1}, b_P$. subvectores se cuantifican de forma independiente por $VQ_1, VQ_2, \ldots, VQ_P$. Cada salida del cuantificador un índice $J_i(x, y)$. Los índices se reconstruyen mediante los cuantificadores inversos VQ_i^{-1}, y los sub-vectores comprimidos con pérdidas se restan para determinar el error de cuantificación $E(x, y) = B(x, y) - \hat{B}(x, y)$. Finalmente, un codificador de entropía elimina las redundancias de los índices de cuantificación y del error de cuantificación.

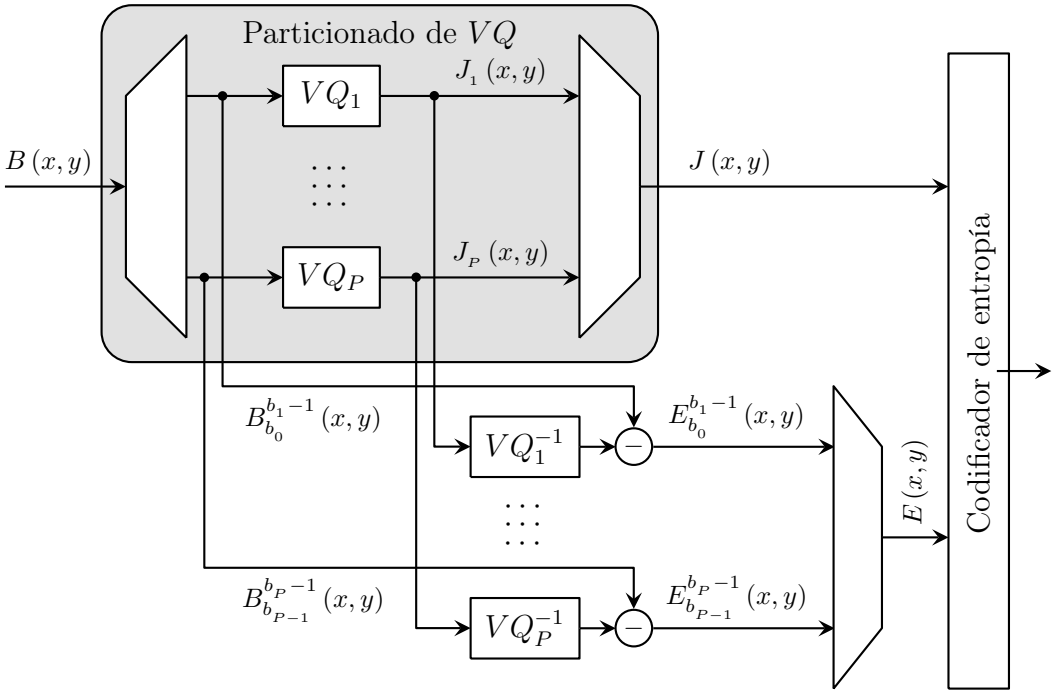


Figura 8.64: Cuantificador vectorial de particionado localmente óptimo (Codificador).

Si la aplicación permite el uso de un compresor casi sin pérdidas (*near-lossless*), puede introducirse un pequeño y controlado error de cuantificación antes de que el residuo pase por el codificador de entropía. Una función interesante de LPVQ es la posibilidad de delimitar firmemente el error en un píxel-a-píxel básico. En [Motta et al. 06] se describen cierto número de experimentos con varias métricas de error.

Si los vectores en los P libros de códigos son ordenados según su energía, los planos de índices LPVQ parecerán P imágenes en escala de grises. Los planos de índices conservan muchas características de los datos originales y naturaleza de imagen-semejante sugiere una codificación inspirada en JPEG-LS (Sección 4.9), el estándar de codificación sin pérdidas ISO/JPEG de imágenes naturales. La naturaleza tridimensional de los planos de índices LPVQ, sin embargo, permite el uso de un contexto causal tridimensional, que JPEG-LS no proporciona. Las estadísticas de los planos de índices muestran que, incluso después de la cuantificación, la correlación entre los planos es más fuerte que la correlación espacial.

La codificación de entropía del error residual se lleva a cabo mediante un codificador aritmético. Cada componente espectral y cada índice de cuantificación tiene un modelo estadístico diferente. Este método se basa en la asunción de que los errores para los diferentes componentes espectrales y las distintas clases de cuantificación tienen distribuciones de probabilidad ligeramente diferentes.

Los P libros de códigos están diseñados utilizando una adaptación del algoritmo Linde-Buzo-Gray (o LBG, Sección 4.14) [Linde, Buzo, y Gray 80]. El algoritmo de particionamiento generaliza LBG observando que, una vez los límites de la partición permanecen fijos, las medidas de distorsión que sean aditivas con respecto a los componentes del vector —como el error cuadrático medio (*Mean Squared Error* o MSE), por ejemplo— pueden ser minimizadas de forma independiente para cada partición aplicando condiciones de optimización en los centroides (el centroide de una figura geométrica es el centro de simetría) y en las celdas. Similarmente, cuando los centroides y las celdas se mantienen fijos,

los límites de las particiones (localmente óptimas) pueden determinarse ávidamente. De este modo, el algoritmo de diseño comienza a partir de particiones de igual tamaño e itera las condiciones de optimización para los centroides, las celdas y los límites. Este diseño converge a centroides localmente óptimos y límites vectoriales.

Además de la compresión competitiva, LPVQ tiene la ventaja de que los índices de cuantificación retienen información importante acerca de la escena original. Los índices de cuantificación constituyen sólo una pequeña fracción del *bitstream*, pero pueden ser utilizados para navegar por la imagen y seleccionar las regiones de interés.

Una característica adicional del compresor LPVQ es producir opcionalmente un error de cuantificación estrechamente vinculado que es controlado en un píxel-a-píxel básico. Debido a la estructura jerárquica de los datos comprimidos mediante este algoritmo, es posible llevar a cabo una clasificación de píxel-puros y una detección de objetivos directamente en el dominio comprimido, con un considerable incremento en velocidad y ahorros de memoria. En [Motta et al. 06] se describe un algoritmo competente.

En la comedia, de hecho, fueron descubiertos y
empleados una mayor variedad de métodos que en la tragedia.

—T.S. Eliot, *El bosque sagrado* (1920)

Parte II

Complementos al libro

Apéndice A

El código ASCII

La Tabla A.1 muestra los 128 códigos ASCII. Cada código se muestra tanto en octal (la columna del extremo izquierdo y la fila superior) como en hexadecimal (la columna del extremo derecho y, o bien la primera fila, o bien la última fila). Los números octales están escritos en cursiva, precedidos por una comilla(*'*). Los números hexadecimales están escritos en una fuente de ancho fijo, precedidos por una comilla doble ("). Para encontrar el código de un carácter, se sustituye el código de la fila superior o inferior para la `x`. El código octal de "A", por ejemplo, se obtiene sustituyendo el *'1* de la fila superior en la *x* del elemento *'10x* de la columna de la izquierda. Así, el código octal es 101, o el binario 1000001. Del mismo modo, el código hexadecimal de "A" es una combinación de **''**`4x` y *'1*, i.e., es **''**`41`.

◇ **Ejercicio A.1 (sol. en pág. 1112):** ¿Cuál es el código hexadecimal de "I"?

Los primeros 32 códigos (0–31), así como el último (`DEL`) son caracteres de control. `SP` (código 32) representa un espacio en blanco. Los otros códigos son para las letras, dígitos, y los signos de puntuación

A.1. Características ASCII

1. Los primeros 32 códigos, y también el último código, son los caracteres de control. Aquéllos son caracteres que se utilizan en operaciones de entrada/salida y en las comunicaciones entre computadoras, y no tienen un grafo asociado, i.e., no se pueden imprimir. Se describen en la siguiente subsección.

2. Los códigos ASCII son arbitrarios. El código de "A" es $\texttt{41}_{16}$, pero no hay ninguna razón especial para haberle asignado ese valor en particular, y casi habría servido también cualquier otro valor. La única regla para la asignación de los códigos es que el código de "B" debe seguir, numéricamente, al código de "A". Por consiguiente, "B" tiene el código $\texttt{42}_{16}$, "C" el $\texttt{43}_{16}$, y así sucesivamente. Lo mismo es cierto para las letras minúsculas y para los diez dígitos.

   Hay también una sencilla relación entre los códigos de las letras mayúsculas y las minúsculas. El código de "a" ($\texttt{61}_{16} = 01\mathbf{1}0\,0001_2$) se obtiene a partir del código de "A" ($\texttt{41}_{16} = 01\mathbf{0}0\,0001_2$) cambiando el sexto bit a 1.

3. El bit de paridad en la Tabla A.1 es siempre 0. El código ASCII no especifica el valor del bit de paridad, y se puede utilizar cualquier valor. Las diferentes computadoras utilizan el código ASCII con paridad par, paridad impar, sin paridad, o incluso con una paridad fija de 1.

↗	*'0*	*'1*	*'2*	*'3*	*'4*	*'5*	*'6*	*'7*	↖
'00x	NULL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	''0x
'01x	BS	HT	LF	VT	FF	CR	SO	SI	
'02x	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	''1x
'03x	CAN	EM	SUB	ESC	FS	GS	RS	US	
'04x	SP	!	"	#	$	%	&	'	''2x
'05x	(	)	*	+	,	-	.	/	
'06x	0	1	2	3	4	5	6	7	''3x
'07x	8	9	:	;	<	=	>	?	
'10x	@	A	B	C	D	E	F	G	''4x
'11x	H	I	J	K	L	M	N	O	
'12x	P	Q	R	S	T	U	V	W	''5x
'13x	X	Y	Z	[	\	]	^	_	
'14x	‘	a	b	c	d	e	f	g	''6x
'15x	h	i	j	k	l	m	n	o	
'16x	p	q	r	s	t	u	v	w	''7x
'17x	x	y	z	{	|	}	~	DEL	
⇑ Octal	''8	''9	''A	''B	''C	''D	''E	''F	↙ ⇑ Hexa

Tabla A.1: El código ASCII.

4. El código del carácter de control DEL está compuesto por todo 1's (a excepción de la paridad que está, como siempre, sin especificar). Ésta es una tradición de los viejos tiempos de la informática (y también de la telegrafía), cuando la cintas de papel perforado eran un medio importante de entrada/salida. Cuando se perforaba información sobre una cinta de papel, siempre que el usuario se diera cuenta de un error, podía eliminar el carácter equivocado, pulsando la tecla DEL en el teclado. Ésto funcionaba haciendo retroceder la cinta un espacio y perforando un marco con todo 1's en la parte superior de los agujeros del carácter erróneo. Cuando se leía la cinta, el lector de cintas simplemente omitía cualquier marco con todo a unos.

> Una estúpida pregunta ASCII, produce una estúpida ANSI.
>
> Anónimo.

A.2. Los caracteres de control ASCII

NUL (Null —nulo–): Ningún carácter. Utilizado como relleno de espacio en un dispositivo de E/S cuando no hay caracteres.

`SOH` (Start of heading —comienzo de cabecera—): Indica el inicio de un cabecera en un dispositivo de E/S. La cabecera puede incluir información relativa a la totalidad del registro que le sigue.

`STX` (Start of text —comienzo del texto—): Indica el inicio del bloque de texto en una E/S en serie.

`ETX` (End of text —fin del texto—): Indica el final de un bloque en una E/S en serie. Complementa el `STX`.

`EOT` (End of transmission —fin de la transmisión—): Indica el final de toda transmisión de E/S en serie.

`ENQ` (Enquiry —petición—): Una señal de petición normalmente se envía desde un ordenador a un dispositivo de E/S antes de iniciar una transferencia de E/S, para verificar que el dispositivo está preparado para aceptar o enviar datos.

`ACK` (Acknowledge —reconocimiento—): Una respuesta afirmativa a una ENQ.

`BEL` (Bell —campana—): Hace que el dispositivo de E/S haga sonar una campana o un timbre o una alarma, para llamar la atención del operador.

`BS` (Backspace —retroceso—): Un comando para que el dispositivo de E/S haga retroceder un carácter. No todos los dispositivos de E/S pueden responder a un `BS`. Un teclado es un ejemplo sencillo de un dispositivo de entrada que no puede volver al carácter anterior. Una vez que se ha presionado una nueva tecla, el teclado pierde la anterior.

`HT` (Horizontal tab —tabulador horizontal—): Se envía a un dispositivo de salida para indicar un movimiento horizontal hasta la siguiente tabulación.

`LF` (Line feed —salto de línea—): Un importante código de control. Indica que el dispositivo de salida se mueva *verticalmente*, hasta el principio de la siguiente línea.

`VT` (Vertical tab —tabulador vertical—): Pide a un dispositivo de salida moverse verticalmente a la siguiente tabulación vertical.

`FF` (Form feed —avance de página—): Pide al dispositivo de salida mover el soporte de salida verticalmente hasta el inicio de la página siguiente; algunos dispositivos de salida, tales como una cinta o un plotter, no tienen ninguna página y para ellos el carácter `FF` carece de sentido.

`CR` (Carriage return —retorno de carro—): Pide a un dispositivo de salida moverse *horizontalmente*, hasta el comienzo de la línea.

`SO` (Shift out): Indica que los códigos de los caracteres que siguen (hasta detectar un `SI`), no están en el juego de caracteres estándar.

`SI` (Shift in): Termina una cadena no estándar de texto.

`DLE` (Data link escape —escape de enlace de datos—): Cambia el significado del carácter que le sigue inmediatamente.

`DC1--DC4` (Device controls —controladores de dispositivo—): Caracteres especiales para el envío de comandos a los dispositivos de E/S. Su significado no está predefinido.

`NAK` (Negative acknowledge —reconocimiento negativo—): Una respuesta negativa a una petición.

`SYN` (Synchronous idle —síncrono inactivo—): Enviado por un transmisor de serie síncrono cuando no hay datos para enviar.

`ETB` (End transmission block —bloque final de transmisión—): Indica el final de un bloque de datos en la transmisión en serie. Se utiliza para dividir los datos en bloques.

`CAN` (Cancel —cancelar—): Indica al dispositivo receptor que cancele (haga caso omiso) del bloque recibido previamente debido a un error en la transmisión.

`EM` (End of medium —fin del medio—): Enviado por un dispositivo de E/S cuando se ha detectado el final de su medio. El medio puede ser una cinta, papel, tarjetas, o cualquier otra cosa utilizada para registrar y almacenar información.

`SUB` (Substitute —suplente—): Cuando el dispositivo receptor, bajo ciertas condiciones, ha recibido incorrectamente un carácter (tiene un bit de paridad erróneo), lo sustituye por este carácter.

`ESC` (Escape —cambio de código—): Altera el significado del carácter inmediatamente posterior. Ésto se utiliza para extender el conjunto de caracteres. Así, `ESC` seguido por una "X" puede significar algo especial para un determinado programa.

Los cuatro separadores siguientes no tienen un significado predefinido en ASCII, excepto que `FS` es el separador más general (separa grandes grupos) y `US`, el menos general:

`FS` (File separator — separador de archivo—),

`GS` (Group separator —separador de grupo—),

`RS` (Record separator — separador de registros—),

`US` (Unit separator —separador de unidades—).

`SP` (Space —espacio—): Éste es el familiar blanco o espacio entre palabras. No tiene impresión y es, por lo tanto, considerado como un carácter de control en lugar de un signo de puntuacion.

`DEL` (Delete —borrar—): Se envía inmediatamente después de haber enviado un carácter erróneo. `DEL` indica la eliminación del carácter anterior (véase el punto 4 de la sección anterior).

Apéndice B

Conceptos básicos sobre probabilidad

El concepto clave discutido aquí es el de *variable aleatoria discreta*. Tal variable X se define considerando los resultados de una secuencia de eventos aleatorios E_i. Asociamos un número real x_i a cada evento E_i, y definimos X como el conjunto de todos los diferentes $x_i s$ (una variable aleatoria también puede ser continua, pero este caso no se discute aquí).

Un ejemplo sencillo es el conjunto de resultados al lanzar un dado. Los resultados son números entre 1 y 6, obtenidos con igual probabilidad. Un ejemplo más interesante es el resultado de tirar dos dados. El resultado es un número entre 2 y 12, pero éstos se obtienen con distintas probabilidades. Un resultado de 2 sólo puede obtenerse si cada dado cae en 1; por consiguiente, tiene una baja probabilidad. Un resultado de 12, similarmente, se obtiene también con una baja probabilidad. Un resultado de 6, sin embargo, se obtiene cuando los dos dados caen en $(1, 5)$, $(2, 4)$, ó $(3, 3)$, por lo que es mucho más común. Un ejemplo práctico son los píxeles de una imagen. En una imagen con 8 bits por píxel, e.g., el valor de un píxel está entre 0 y 255, pero no todos los valores se producen con la misma probabilidad. Imagine dos imágenes, **A**, con una gran cantidad de rojo y **B**, en su mayor parte azul. Las distribuciones de los píxeles de esas imágenes forman diferentes variables aleatorias, porque un píxel con, digamos, el valor 112 se puede producir en **A** con una probabilidad de 0,01 y en **B**, con una probabilidad de 0,02.

◇ **Ejercicio B.1 (sol. en pág. 1113):** Considérese el caso de lanzar tres monedas, donde una moneda puede caer[1] en cara (H) o cruz (T). Establézcase nuestra variable aleatoria X en el número de monedas que caen sobre la cara. Muéstrense todos los valores de X y sus probabilidades.

Es evidente que podemos escribir $X = (x_1, p_1, x_2 p_2 \ldots, x_n, p_n)$, donde los x_i son los distintos valores que toma X y p_i es la probabilidad de x_i. El promedio (o *media*) de X es $\overline{X} = (1/n) \sum x_i$.

El valor esperado, o *esperanza*, de la variable X se denota $E(X)$ y se define como $E(X) = \sum p_i x_i = \sum x_i P(X = x_i)$. Ésta es la suma de los posibles valores x_i de X, cada uno ponderado por su probabilidad p_i. La notación $P(X = x_i)$ debe ser leída "la probabilidad de que X tome el valor x_i". Si todas las probabilidades p_i son iguales ($p_i = 1/n$), entonces $E(X) = \overline{X}$. En el caso de tres monedas, el número de caras esperado es $E(X) = \overline{X} = (3+2+2+1+2+1+1+0)/8 = 3/2$. No esperamos, por supuesto, que cualquier evento individual (lanzamiento) produzca 1,5 caras, pero éesta será el número obtenido en promedio.

◇ **Ejercicio B.2 (sol. en pág. 1113):** ¿Cuál es el valor esperado de la variable aleatoria X que toma sólo un valor v?

A continuación consideramos las dos variables aleatorias $X = (0, 20)$ e $Y = (10, 10)$. En cualquier caso, el valor esperado de la variable es 10. Sin embargo, la variable Y siempre es igual a 10, mientras

[1] H viene de Head (cabeza) y T de tail (cola).

que X nunca toma este valor. Ésto indica que la esperanza de una variable aleatoria no da suficiente información al respecto, y deberíamos considerar en qué medida los valores individuales x_i forman parte de la esperanza matemática $E(X)$ (cuán lejos los valores individuales se desvían del valor esperado). Por lo tanto, deseamos obtener una expresión para la *varianza* de una variable aleatoria. Intuitivamente, parece que Y debería tener una varianza cero, mientras que la varianza de X debería ser distinta de cero.

El primer método que viene a la mente es el cálculo de la suma de las diferencias $[x_i - E(X)]$. Esta definición de la varianza es simple, pero también contradictoria, ya que produce un cero en el caso de X. Mejores resultados se obtienen cuando se utilizan diferencias absolutas $|x_i - E(X)|$, en lugar de las diferencias.

Para comprender por qué incluso esta definición no es satisfactoria, considere el caso de $n-1$ valores x_i, quizá los resultados de $n-1$ mediciones de longitudes, que están todos en el rango $[0, 10]$. La esperanza está también en este rango. Ahora realizamos la próxima medición y obtenemos el valor $x_n = 1000$. Tiene sentido exigir que este valor único, que está tan lejos de los otros como de la esperanza, deba dar más peso al cálculo de la varianza. Es por éso por lo que se utilizan los cuadrados de las diferencias, en lugar de las propias diferencias, en la definición de la varianza. Si la expectativa es 10, entonces un valor $x_i = 5$ contribuye con $(5-10)^2 = 25$, pero un valor $x_i = 110$ proporciona la contribución mucho mayor de $(110-10)^2 = 10000$.

Ésta es la razón por la que varianza de la variable aleatoria X se define como (véase también la página 4.21.3):

$$\mathrm{Var}(X) = \sum p_i [x_i - E(X)]^2 .$$

Si todos los p_i son idénticos, entonces $p_i = 1/n$ y la varianza se convierte en:

$$\mathrm{Var}(X) = \frac{1}{n} \sum [x_i - E(X)]^2 .$$

La desviación estándar es otra medida estadística útil. Se define como la raíz cuadrada de la varianza.

La varianza es una unidad natural para medir cómo varía una variable, pero la desviación estándar también es útil porque una variable aleatoria X tiene una dimensión, tal como la altura o el peso. La dimensión de la varianza es el cuadrado de la dimensión de la variable (el cuadrado de la altura o el peso al cuadrado), por lo que la varianza y la variable aleatoria no pueden ser utilizados juntos en los cálculos (por ejemplo, no se puede sumar o restar). La desviación estándar, sin embargo, tiene la dimensión de la variable.

> La computadora informó aquí que tres espacios representaron el ochenta y uno por ciento de la varianza.
>
> —Michael Crichton, *El hombre terminal*

La desviación estándar es una medida del "radio" de la variable. Si X tiene una Distribución de Gauss (Sección B.3.1), el 68 % de sus valores estarán dentro de un estándar desviación de la media, y el 95 % estará dentro de dos desviaciones estándar.

◇ **Ejercicio B.3 (sol. en pág. 1113):** La definición anterior implica $\mathrm{Var}(X) = E\left[(X - E(X))^2\right]$. Demuéstre que $\mathrm{Var}(X) = E\left(X^2\right) - [E(X)]^2$.

Las observaciones bivariables son aquellas en las que se toman los valores de dos variables aleatorias, X e Y. Estas observaciones se conocen como "apareadas", y el emparejamiento puede ocurrir en varias situaciones. Los siguientes son algunos ejemplos:

1. Cuando hay dos variables diferentes para cada caso (e.g., la edad y la talla del zapato, la altura y el peso, el sexo y el coeficiente intelectual (IQ), la mortalidad infantil de in país y de la educación media).

2. Cuando la misma variable se mide para cada caso en dos instantes diferentes (e.g., el nivel de lectura antes del entrenamiento y el nivel de lectura después, el IQ a los 3 años y el IQ a los 6 años).

3. Cuando las mismas o distintas variables se miden a partir de casos relacionados (e.g., los logros educativos de padre e hijo, la altura y la altura del marido y de la esposa; la ansiedad de la madre y de la seguridad del niño).

4. Cuando las mismas o distintas variables se miden en casos no relacionados al mismo tiempo (por ejemplo, la tasa de desempleo en la ciudad A y en la ciudad B en un mes determinado).

Las dos variables pueden ser independientes o dependientes. Imaginemos dos personas que viven en áreas diferentes, cuyos números de teléfono (menos los códigos de área) son los mismos. Intuitivamente sentimos que las alturas de estas personas son independientes. Por otro lado, la ingresos de dos personas que tienen un número similar de teléfono dentro de la misma área puede ser dependiente (aunque tal vez sólo en un pequeño grado), porque tales personas pueden vivir cerca unas de otras, por lo que existe la posibilidad de que interactúen o incluso trabajen juntas.

La cuestión es cómo medir la relación entre dos variables aleatorias. Es fácil definir la independencia. Dos variables aleatorias X e Y son independientes si

$$P\left(X=a, Y=b\right)=P\left(X=a\right)P\left(Y=b\right), \quad \text{para cualquier } a, b.$$

La *covarianza* de dos variables aleatorias es una medida de la relación lineal entre ellas. La covarianza sólo indica la dirección de la relación lineal, no su fuerza. Se define como:

$$COV_{xy}=\frac{\sum\left(x_i-\overline{X}\right)\left(y_i-\overline{Y}\right)}{n-1}.$$

El siguiente sencillo código en Matlab calcula la matriz de covarianza de una matriz cuadrada donde cada columna es una variable:

```
function xy=covarmat(x)
[m,n]=size(x);
xc=x-repmat(sum(x)/m,m,1);% resta del promedio
xy=xc' * xc/(m-1);
```

Una covarianza positiva indica que los valores de una variable se incrementan a la par que los valores de la otra. Una covarianza negativa indica que los valores de una variable se decrementan cuando los valores de la otra se incrementan. Una covarianza cero indica que no existe una relación lineal entre ambas variables; están *descorrelacionadas*. El valor preciso de la covarianza normalmente no tiene ningún significado porque depende de las unidades de medida de X e Y, y de sus varianzas. Para medir realmente la cantidad de correlación entre dos variables utilizamos la medida estadística de *correlación*. Se define como:

$$R=\frac{n\sum x_i y_i-\sum x_i\sum y_i}{\sqrt{\left[n\sum x_i^2-\left(\sum x_i\right)^2\right]\left[n\sum y_i^2-\left(\sum y_i\right)^2\right]}}. \tag{B.1}$$

Ésta mide la relación lineal entre dos variables apareadas. Los valores de R están en el rango: de -1 (relación negativa perfecta), a 0 (ninguna relación), a $+1$ (relación positiva perfecta).

Supongamos que tenemos dos matrices x_i e y_i de números, quizá las mediciones de la altura y el peso de un grupo de personas. El coeficiente de correlación de las dos matrices es una medida de cuán exactamente puede ser predicha la variación de una de ellas (digamos, y_i) a partir de la variación de la otra (x_i). También podemos considerar x_i e y_i variables en lugar de matrices. Si nos fijamos en x_i como la variable independiente y en y_i como la variable dependiente, entonces el coeficiente de correlación de las dos variables es una la medida de cómo la variación de la variable dependiente puede ser predicha a partir de la variación de la variable independiente.

Si $y_i = ax_i$ para cada i (para algún real a), entonces la variación de y_i puede predecirse fácilmente y con precisión a partir de x_i, y el coeficiente de correlación es 1 (o -1 si a es negativo). Si y sigue a x (i.e., si $x_{i+1} < x_i$ implica que $y_{i+1} < y_i$, y $x_{i+1} > x_i$ implica que $y_{i+1} > y_i$), entonces el coeficiente de correlación es un número positivo. En el caso contrario (donde $x_{i+1} < x_i$ implica que $y_{i+1} > y_i$, y $x_{i+1} > x_i$ implica que $y_{i+1} < y_i$), el coeficiente de correlación es un número negativo.

> Varianza es lo que cualquiera de dos estadistas encuentra.
>
> —Desconocido

B.1. Conjunción y unión de eventos

En primer lugar, discutimos las probabilidades de *eventos independientes*. Ésos son eventos que no afectan a las probabilidades de cada uno. Un ejemplo es el lanzamiento de un dado o de una moneda. Cada resultado de un lanzamiento es independiente del anterior (y también de los siguientes). Incluso cuando dos dados son lanzados juntos, los dos resultados son independientes. Denotamos la probabilidad de que suceda x, como $P(x)$. La *probabilidad conjunta* $P(x \cdot y)$ de los eventos independientes x e y es la probabilidad de que un evento produzca x y otro evento, produzca y. Esta probabilidad es simplemente el producto de las probabilidades individuales. Por ejemplo, si la probabilidad nevar hoy (evento A) es del 25 % y la de nevar mañana (evento B) es del 40 %, entonces la probabilidad de que nieve hoy y mañana, $P(A \cdot B)$, es el producto $0{,}25 \times 0{,}4 = 0{,}1$, i.e., del 10 % (que es, por supuesto, más pequeña que cada una de las probabilidades individuales). La probabilidad del *complemento* $\overline{x}$ de un evento x es igual a uno menos la probabilidad del evento, $P(\overline{x}) = 1 - P(x)$. Así, la probabilidad de que *no* nieve hoy es de $1 - 0{,}25 = 0{,}75$.

Otro concepto importante es la probabilidad de la *unión* de los eventos. Por ejemplo, ¿cuál es la probabilidad de que nieve hoy *o* mañana? Para entender cómo se calcula ésto, consideremos el caso opuesto. Lo contrario de está nevando hoy o nevará mañana es el caso donde hoy no nieva (evento $\overline{A}$) y mañana no nieva (evento $\overline{B}$). La probabilidad de este evento es, por supuesto, la probabilidad conjunta $P\left(\overline{A} \cdot \overline{B}\right)$, i.e., $P\left(\overline{A}\right) P\left(\overline{B}\right)$, por lo que concluimos que este evento conjunto es el complemento de la unión que estamos buscando. Por consiguiente, acabamos con $P(A+B) = 1 - P\left(\overline{A} \cdot \overline{B}\right) = 1 - P\left(\overline{A}\right) P\left(\overline{B}\right)$. En el ejemplo de la nieve, el resultado es $P(A+B) = 1 - P\left(\overline{A}\right) P\left(\overline{B}\right) = 1 - (1 - 0{,}25)(1 - 0{,}4) = 1 - 0{,}45 = 0{,}55$. Este resultado es mayor que el 25 % y el 40 %, pero todavía es una probabilidad, i.e., está en el rango $[0, 1]$.

◇ **Ejercicio B.4 (sol. en pág. 1113):** (Para jugadores.) Calcúlese la probabilidad de ganar en los dos juegos siguientes: En el juego A, el jugador tira dos dados hasta 24 veces y gana si saca un seis doble. En juego B el jugador tira un dado y gana si obtiene un seis en cuatro tiradas. Estas probabilidades ganadoras fueron calculadas por Blaise Pascal, uno de los fundadores de la teoría de la probabilidad moderna , en 1654.

B.2. Probabilidad Condicional

No todos los eventos son independientes. Cuando se trata de eventos dependientes, tenemos que calcular *probabilidades condicionales*. Normalmente, preguntamos: ¿Cuál es la probabilidad del evento A dado que el evento B ha ocurrido? Ésta es la probabilidad condicional de A (más precisamente, la probabilidad de A condicionada a B) y se denota por $P(A \mid B)$. El campo de la probabilidad condicional se denomina a veces *estadística Bayesiana*, ya que fue desarrollada por primera vez por el reverendo Thomas Bayes [1702–1761], quien propuso la fórmula básica:

$$P(A \mid B) = \frac{P(A \cdot B)}{P(B)}. \tag{B.2}$$

Ejemplo: Se dan tres cartas. Una es roja por ambos lados, otra es negra por ambos lados, y la tercera es de color rojo en un lado y negro en el otro lado. Usted escoge una tarjeta sin mirarla, la coloca sobre la mesa, luego la mira. Si usted ve una cara roja, ¿cuál es la probabilidad de que la otra parte también sea roja? Esta probabilidad es condicional y se denota por $P(Rup \mid Rdown)$. La Ecuación se convierte en:

$$P(Rup \mid Rdown) = \frac{P(Rup \cdot Rdown)}{P(Rdown)}.$$

La probabilidad $P(Rup \mid Rdown)$ es $1/3$, ya que sólo una de las tres tarjetas tiene dos lados rojos. La probabilidad $P(Rdown)$ se puede calcular como sigue: Tenemos tres tarjetas. La probabilidad de recoger cualquier tarjeta es $1/3$. La probabilidad de que la primera tarjeta recogida tenga rojo abajo es $(1/3) \times 1$, ya que sus lados son de color rojo. La probabilidad de que la segunda tarjeta recogida tenga rojo abajo es $(1/3) \times 0$, ya que ninguno de sus lados es de color rojo. La probabilidad de que la tercera tarjeta recogida tenga rojo abajo es $(1/3) \times (1/2)$, ya que sólo uno de sus lados es de color rojo. En consecuencia, la probabilidad total de recoger una tarjeta y que tenga rojo abajo es $(1/3)(1 + 0 + 1/2) = 1/2$.

El mismo resultado puede ser obtenido considerando puntos de eventos en un *espacio de eventos*. La primera tarjeta será roja sin importar cuál es, lo que aporta un punto (punto 1, con una probabilidad de $1/3$) al espacio de eventos. La segunda carta nunca tendrá rojo abajo, sea cual sea. Contribuye con otro punto, punto 2, también con probabilidad de $1/3$, al espacio de eventos. La tercera carta tendrá rojo abajo si se recoge de cierta manera, y rojo en la parte superior si se recoge de otra manera. Por lo tanto, contribuye con dos puntos (puntos 3 y 4, con probabilidad $1/6$ cada uno) al espacio para eventos. Estamos interesados en los puntos 1 y 3, cuyas probabilidades son $1/3$ y $1/6$, que suman $1/2$. La Ecuación (B.2), por lo tanto, produce la probabilidad condicional:

$$P(Rup \mid Rdown) = \frac{P(Rup \cdot Rdown)}{P(Rdown)} = \frac{1/3}{1/2} = \frac{2}{3};$$

un resultado no intuitivo.

◇ **Ejercicio B.5 (sol. en pág. 1113):** Tres empresas A, B y C están compitiendo por un contrato de la NASA para desarrollar un nuevo vehículo espacial. Los expertos coinciden en que las probabilidades de A, B y C para obtener el contrato son de $2/5$, $2/5$, y $1/5$, respectivamente. En cierto momento la empresa B se retira de la competición. ¿Cuáles son las posibilidades de victoria de A y C ahora?

Ejemplo: Una familia tiene dos hijos, donde un hijo puede ser un niño o una niña. Si no se sabe nada a priori sobre los hijos, entonces existen las cuatro posibilidades[2] (BB, GG, BG, GB), y la probabilidad de tener dos niños es $1/4$. ¿Cuál es la probabilidad condicional de que la familia tenga dos niños, si se sabe que uno de los hijos es un niño? Intuitivamente este conocimiento elimina el caso GG y reduce las

[2] B para boy, niño; G para girl, niña.

posibilidades a (BB, BG, GB), de las que BB es lo que estamos buscando. La probabilidad condicional es, por lo tanto:

$$P\left(\text{dos niños} \mid \text{un niño}\right) = \frac{P\left(BB\right)}{P\left(BB\right) + P\left(BG\right) + P\left(GB\right)} = \frac{\frac{1}{4}}{\frac{1}{4} + \frac{1}{4} + \frac{1}{4}} = \frac{1}{3}.$$

Utilizando la estadística Bayesiana tenemos que calcular $P\left(A \mid B\right)$, donde A es el evento de tener dos niños y B es el evento de tener un niño. La cantidad $P\left(A \cdot B\right)$ es la probabilidad de tener dos niños y tener un niño. Bueno, si una familia tiene dos niños, entonces tiene un niño, por lo que mirando las tres posibilidades (BB, GG, BG), encontramos que $P\left(A \cdot B\right) = P\left(A\right) = 1/3$. La cantidad $P\left(B\right)$ es la probabilidad de tener un niño. Ésto es igual a $2/3$ porque este evento representa dos de las tres posibilidades. La Ecuación (B.2) produce:

$$P\left(A \mid B\right) = \frac{P\left(A \cdot B\right)}{P\left(B\right)} = \frac{1/3}{2/3} = \frac{1}{2}.$$

Un caso práctico importante es cuando un evento A tiene ciertas alternativas A_i. En tal caso, la Ecuación (B.2) se pueden generalizar a:

$$\begin{aligned} P\left(A_i \mid B\right) =& \frac{P\left(A_i \cdot B\right)}{P\left(A_1 \cdot B\right) + P\left(A_2 \cdot B\right) + \cdots + P\left(A_n \cdot B\right)} \\ =& \frac{P\left(A_i\right) P\left(B \mid A_i\right)}{P\left(A_1\right) P\left(B \mid A_1\right) + P\left(A_2\right) P\left(B \mid A_2\right) + \cdots + P\left(A_n\right) P\left(B \mid A_n\right)}. \end{aligned}$$

Éste es el conocido teorema de Bayes, publicado póstumamente por Thomas Bayes en 1763. Requiere el conocimiento de las probabilidades $P\left(A_i\right)$ de las alternativas y de las probabilidades condicionales $P\left(B \mid A_i\right)$ de B relativas a las alternativas. Esta teorema es utilizado por el codificador QM para construir las tablas de estimación de probabilidad usadas por JPEG, JBIG, y JBIG2 (véase, por ejemplo, la Tabla 2.68).

Ejemplo: Un estudiante tiene que seleccionar un curso de ciencias entre matemáticas, física, química y biología. Basándose en su conocimiento personal del estudiante, su tutor imagina que las probabilidades de que el estudiante seleccione una de las cuatro clases son 0,4, 0,3, 0,2, y 0,1, respectivamente (estas probabilidades suman 1). El tutor no sabe qué curso fue seleccionado por el estudiante en realidad, pero al final del semestre, el tutor se entera de que el estudiante obtuvo una A en el curso seleccionado. Basándose en las dificultades de estos cursos, el tutor estima que las probabilidades de que el estudiante obtenga una A en matemáticas, física, química y biología son 0,1, 0,2, 0,3, y 0,9, respectivamente (éstas no tienen que sumar 1). Usando el teorema de Bayes, el tutor puede ahora revisar las probabilidades originales de que el estudiante haya seleccionado uno de los cuatro cursos. La probabilidad de que el estudiante haya seleccionado realmente, por ejemplo, el curso de matemáticas es:

$$\begin{aligned} P\left(\text{seleccionar matemáticas} \mid \text{habiendo obtenido una } A\right) =& \frac{0{,}4 \cdot 0{,}1}{0{,}4 \cdot 0{,}1 + 0{,}3 \cdot 0{,}2 + 0{,}2 \cdot 0{,}3 + 0{,}1 \cdot 0{,}9} \\ =& \frac{4}{25} = 0{,}16. \end{aligned}$$

◇ **Ejercicio B.6 (sol. en pág. 1113):** Basándose en el hecho de que el estudiante ha obtenido una A, calcúlense las nuevas probabilidades de que el estudiante haya elegido realmente física, química y biología.

Ejemplo: Éste es el famoso problema de los *tres prisioneros*. Tom, Dick, y Harry están en la cárcel. La mañana siguiente, uno de ellos va a ser ejecutado y los otros dos serán liberados. Ellos no saben cuál

será su suerte, pero el guardia sí lo sabe. Cerca de la medianoche, Tom le pregunta al guardia: "No sé quién va a ser ejecutado, pero sí sé que, o bien Dick, o bien Harry, va a ser liberado. Por favor, dígame quién porque, después de todo, no cambia mi probabilidad de ser ejecutado." El guardia (un estadístico aficionado que nunca oyó hablar de Bayes) respondió: "Ahora usted estima que la probabilidad de ser ejecutado es $1/3$. Si le digo que, por ejemplo, Dick va a ser liberado, podría conocer que sólo usted y Harry siguen siendo candidatos para la ejecución, por lo que la probabilidad de ser ejecutado subiría a $1/2$ y no dormirá bien esta noche." La respuesta del guardia es sencilla pero equivocada, ya que no tiene en cuenta la probabilidad condicional. El guardia podría dar a Tom un nombre, como Dick, pero este conocimiento no habría cambiado la probabilidad condicional de que Tom sea ejecutado. Aquí está el análisis.

Sin ningún tipo de conocimiento previo, cada prisionero tiene una probabilidad de $1/3$ de ser ejecutado y una probabilidad de $2/3$ de ser liberado. Denotamos por A el evento de que Tom será ejecutado. Es evidente que $P(A) = 1/3$. Queremos calcular la probabilidad condicional de que Tom sea ejecutado, teniendo en cuenta que el guardia le dijo que Dick iba a ser liberado. El evento B es, por lo tanto, el guardia diciendo a Tom que Dick va a ser puesto en libertad. El punto sutil es que el evento B no es necesariamente la liberación de Dick; lo que el guardia está diciendo a Tom es que Dick va a ser liberado. La diferencia es fácil de ver teniendo en cuenta todos los puntos de evento posibles en un espacio de eventos. Hay cuatro de tales puntos, como sigue:

- Punto 1: Harry va a ser ejecutado. En este caso el guardia le dice a Tom que Dick va a ser liberado. Puesto que cada preso tiene una probabilidad de $1/3$ de ser ejecutado, la probabilidad de este punto es $1/3$.
- Punto 2: Dick va a ser ejecutado. En este caso, el guardia le dice a Tom que Harry va a ser liberado. La probabilidad de que este punto es, de nuevo, $1/3$.
- Puntos 3 y 4: Tom a va ser ejecutado (con probabilidad $1/3$). En este caso el guardia le dice a Tom que, o bien Harry, o bien Dick, van a ser liberados. Estos son los puntos 3 y 4, respectivamente, en el espacio de eventos, cada uno con una probabilidad de $1/6$.

La probabilidad conjunta $P(A \cdot B)$ de que Tom será ejecutado y el guardia le dice que Dick será liberado es, por lo tanto, la probabilidad del punto 4, ó $1/6$. La probabilidad del evento B (el guardia dirá que Dick va a ser liberado) es la suma de los puntos 1 y 4, ó $1/3 + 1/6$. La probabilidad condicional de que Tom será ejecutado, si el guardia dice que Dick va a ser liberado es, por lo tanto,

$$P(A \mid B) = \frac{P(A \cdot B)}{P(B)} = \frac{1/6}{1/3 + 1/6} = 1/3,$$

la misma probabilidad de que Tom será ejecutando sin que el guardia diga nada.

(El autor está en deuda con Robert J. Henderson por señalar este problema y su solución.)

La conclusión es: No se meta en problemas, pero si usted está en él (i.e., problema) ponga su confianza en el viejo reverendo Bayes.

B.3. Distribuciones de probabilidad

Imagine una cadena de entrada donde los símbolos tienen la misma probabilidad de ocurrencia. Un ejemplo puede ser una imagen formada por N píxeles de 8 bits, donde cada uno de los 256 colores posibles aparece exactamente $N/256$ veces. Decimos que los colores en esta imagen tiene una *distribución plana* de valores. El gráfico cuyo eje x muestra los 256 colores y cuyo eje y muestra el número de veces que aparece cada color será una línea horizontal. ahora imagine una imagen similar, pero con tonos en su mayoría verdes. Hay muchos píxeles con tonos de verde diferentes y unos pocos con cualquier

otro color. El mismo gráfico será alto para los valores de x que representan tonos de verde y bajo en cualquier otra parte. La distribución de colores en esta imagen no es plana.

> La teoría de la simbiogénesis asume que la explicación más probable para estructuras improbablemente complejas (vivas o cualquier otra) reside en la asociación de partes menos complicadas. Las frases son más fáciles de construir combinando palabras que combinando letras. Las frases después se combinan en párrafos, los párrafos en capítulos, y, eventualmente, los capítulos se combinan para formar un libro; altamente improbable, pero mucho más probable que la posibilidad de llegar a un libro buscando en el espacio de las combinaciones posibles a nivel de letras o palabras.
>
> —George B. Dyson, *Darwin entre las máquinas.*

B.3.1. Distribución Gaussiana

La distribución Gaussiana (también conocida como la normal) es una importante herramienta estadística que se utiliza en muchas ramas de la ciencia. Proporciona un buen modelo para distribuciones continuas que ocurren en muchas situaciones cotidianas. Ejemplos son los siguientes:

1. La distribución de las alturas de personas. La mayor parte de la gente tiene estatura mediana. Pocos son altos o bajos. Menos aún son muy altos o muy bajos. Prácticamente nadie es un gigante o un enano. Imagine una muestra de personas cuya estatura es conocida. Si la muestra es lo suficientemente grande y no es sesgada, la gráfica que describe el número de personas de altura h como una función de h tendrá un aspecto muy similar a la Figura B.1.

2. La velocidad de las moléculas de gas. Las moléculas de un gas están en constante movimiento. Se mueven aleatoriamente, chocan unas con otras y con los objetos que las rodean, y cambiar sus velocidades continuamente. Sin embargo, la mayoría de las moléculas en un volumen dado de gas se mueven a la misma velocidad, y sólo unos pocos movimientos son mucho más rápidos o más lentos que esta velocidad. Esta velocidad está relacionada con la temperatura del gas. A mayor velocidad media, más caliente notamos el gas. (Este ejemplo es asimétrico, ya que la velocidad mínima es cero, pero la velocidad máxima puede ser muy alta.)

3. El Château Chambord en el valle de Loire en Francia tiene una magnífica escalera, diseñada por Leonardo da Vinci en forma de espiral de doble rampa. Desgastada por las huellas de innumerables generaciones de turistas, el piso de mármol de esta escalera ahora se ve como una curva de distribución normal invertida. Está deteriorada principalmente en el medio, donde pisa la mayoría de la gente, y el desgaste se estrecha a cada lado desde el centro. Esta escalera, y otras como ella, son realizaciones físicas del concepto matemático abstracto de distribución de probabilidad.

4. Los números primos son familiares para la mayoría de la gente. Son atractivos e importantes para los matemáticos porque cualquier número entero positivo se puede expresar como un producto de los números primos (sus *factores primos*) de una sola forma. Los números primos son, pues, los bloques de construcción desde los que todos los otros números enteros pueden ser construidos. Resulta que el número de factores primos diferentes se distribuye normalmente. Pocos enteros tienen sólo uno o dos factores primos distintos, algunos enteros tienen muchos factores primos distintos, mientras que la mayoría de los enteros tienen un número pequeño de factores primos diferentes. Ésto se conoce el teorema de Erdös-Kac.

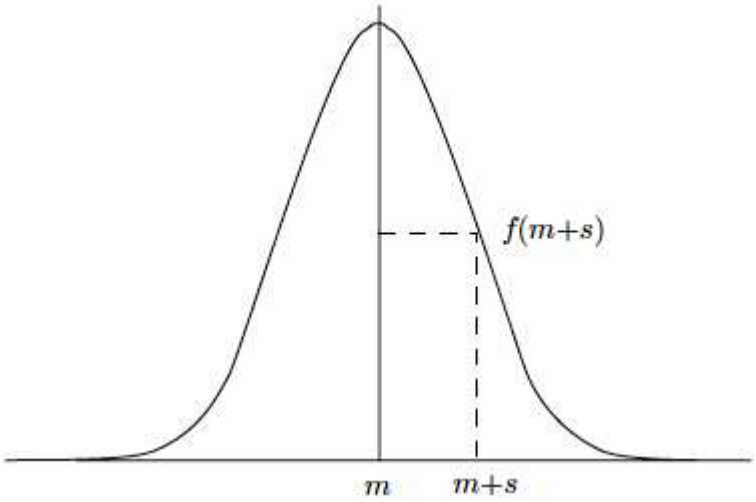


Figura B.1: Distribución Gaussiana (Normal).

> Las características generales de la curva de campana tienen un sentido intuitivo, pero por qué debe tener exactamente la forma que tiene, lo que me permite predecir con seguridad y con ayuda de unas pocas líneas de aritmética primaria, y ninguna otra forma —un poco más ancho en la cadera, digamos, o más en punta, como el sombrero de una bruja— sigue siendo un misterio.
>
> —Hans Christian von Baeyer, *El demonio de Maxwell, 1998.*

La distribución Gaussiana con media m y desviación estándar s se define como:

$$f(x) = \frac{1}{s\sqrt{2\pi}} \exp\left\{-\frac{1}{2}\left(\frac{x-m}{s}\right)^2\right\}.$$

Esta función tiene un máximo para $x = m$ (i.e., en la media), donde su valor es $f(m) = 1/(s\sqrt{2\pi})$. También es simétrica con respecto a $x = m$, ya que depende de x de acuerdo con $(x-m)^2$. Tiene la forma general de "campana" de la Figura B.1. En $x = m + s$ y $x = m - s$ su valor es:

$$f(m \pm s) = \frac{1}{s\sqrt{2\pi}} e^{-\frac{1}{2}} \approx \frac{0{,}6065}{s\sqrt{2\pi}},$$

lo que significa que en una desviación estándar de la media ha bajado a aproximadamente un 60 % de su máximo. En dos desviaciones estándar se reduce a aproximadamente 0,1353 de su valor máximo.

El área total bajo la curva normal es una unidad. El área de una desviación estándar de la media es igual a $\approx 0{,}682$. En dos desviaciones estándar es igual a $\approx 0{,}9545$ y en tres desviaciones estándar es $\approx 0{,}9973$.

(La distribución de Gauss y el concepto de desviación estándar fueron descubiertas y discutidas por Abraham de Moivre en 1733, cuando tenía 18 años.)

La forma precisa de la curva depende de s. A medida que s aumenta, la curva se hace más amplia. Para valores pequeños de s la curva se aproxima un pico estrecho de altura $1/(s\sqrt{2\pi})$ situada en $x = m$.

Cuando hablamos de números aleatorios normalmente significa que los números están uniformemente distribuidos sobre un cierto intervalo $[a, b]$. Si dividimos el intervalo $[a, b]$ en subintervalos de igual tamaño, cualquiera de ellos contiene la misma cantidad de números aleatorios. Es posible extraer números aleatorios que tienen distintas distribuciones, tales como la Gaussiana. Cuando calculamos muchos números aleatorios que están normalmente (i.e., siguiendo la Gaussiana) distribuidos con media m y desviación estándar s, entonces se cuenta la cantidad y de estos números en un pequeño subintervalo $[x, x + \epsilon]$, se representa (x, y) como un punto, y se repite para muchos subintervalos, obtenemos la distribución normal con media m y desviación estándar s.

> Debido a la naturaleza misma de las tablas, no parece necesario que revise todas las páginas del manuscrito final para capturar errores aleatorios.
>
> —*Un millón de dígitos aleatorios con* 100,000 *desviaciones normales*, RAND Corp., 1955.

Aquí mostramos dos formas de calcular números aleatorios que se distribuyen normalmente con media 0 y desviación estándar 1:

1. Dibujar n números aleatorios R_i uniformemente distribuidos en el intervalo $[-a, +a]$ para cualquier real a y calcular su media $(1/n)\sum R_i$. La media es el primero de los N_i números aleatorios normalmente distribuidos. Repetir este proceso para obtener N_2, N_3, y así sucesivamente. La n más grande, la más cercana a la normal, será la distribución de estos números. La razón por la que los N_i se distribuyen normalmente es que es raro que el promedio de R_i sea $-a$ o $+a$ o cercano a estos valores, pero es común que sea aproximadamente 0. Éste es un aspecto de la *ley de los grandes números* que dice: si R_i son números aleatorios de cualquier distribución, entonces los promedios $(1/n)\sum R_i$ se distribuyen normalmente.

2. El método 1 anterior es simple pero lento, ya que n debe ser grande. El *método Polar* (véase [Knuth 81] Vol. 2, Sec. 3.4.1) es más eficiente. Sean U_1 y U_2 dos números aleatorios uniformemente distribuidos en el rango $[0, 1]$. Calculamos dos números aleatorios normalmente distribuidos X_1, X_2 mediante los dos sencillos pasos siguientes:

   - Paso 1. Calcular $V_1 := 2U_1 - 1$, $V_2 := 2U_2 - 1$, y $S := V_1^2 + V_2^2$.
   - Paso 2. Si $S \geq 1$ ir al paso 1; si no, calcular $X_1 := V_1\sqrt{\frac{-2\ln S}{S}}$, $X_2 := V_2\sqrt{\frac{-2\ln S}{S}}$.

Una vez que se obtiene una secuencia N_i de números aleatorios normalmente distribuidos con media 0 y desviación estándar 1, es fácil para convertirlos a números aleatorios normalmente distribuidos con media m y desviación estándar s. Sólo hay que transformar cada N_i en $m + N_i \times s$.

Supongamos que tenemos una función `Rnd()`, que devuelve números aleatorios uniformemente distribuidos en el intervalo $[0, 1]$. Los números aleatorios Gaussianos con media cero y una desviación estándar de uno pueden obtenerse mediante el siguiente código:

```
x:=0.0;
for i:=1 to 12 do x:=x+Rnd();
Gauss:=x-6.0;
```

	x					
V	0	2	4	6	8	10
3 :	0,408248	0,0797489	0,015578	0,00304316	0,00059446	0,000116125
4 :	0,353553	0,0859547	0,020897	0,00508042	0,00123513	0,000300282
5 :	0,316228	0,0892598	0,025194	0,00711162	0,00200736	0,000566605
1 000 :	0,022360	0,0204475	0,018698	0,0170982	0,0156353	0,0142976

Tabla B.2: Algunos valores de la distribución de Laplace con $V = 3, 4, 5$, y 1 000.

B.3.2. Distribución de Laplace

La distribución de probabilidad de Laplace es similar a la distribución normal (Gaussiana), pero es más estrecha y se vuelve puntiaguda bruscamente. La distribución general de Laplace con varianza V y la media m viene dada por:

$$L(V, x) = \frac{1}{\sqrt{2V}} \exp\left(-\sqrt{\frac{2}{V}}\,|x - m|\right).$$

La Tabla B.2 muestra algunos valores para las distribuciones de Laplace con $m = 0$ y $V = 3, 4, 5$, y 1 000.

El factor $1/\sqrt{2V}$ está incluido en la definición de la distribución de Laplace con el fin de escalar el área bajo la curva de la distribución a 1.

◇ **Ejercicio B.7 (sol. en pág. 1114):** ¿Cuál es la integral indefinida de la distribución de Laplace?

La distribución de Laplace es utilizada por el método de compresión de imágenes MLP (Sección 4.21).

B.3.3. Distribuciones discretas

Imagine n eventos independientes que se efectúan de tal manera que en cada evento, un cierto resultado (lo que llamamos un "éxito") se produce con la misma probabilidad p. Un sencillo ejemplo es un lanzamiento de monedas, donde se define un éxito cuando la moneda cae de cara. la probabilidad es, por supuesto, 0,5. El número de éxitos en los n eventos es una variable aleatoria X que tiene una *distribución binomial*. La probabilidad de que X tome un valor x es:

$$P(X = x) = \frac{n!}{x!\,(n-x)!} p^x\,(1-p)^{n-x}, \quad \text{para } x = 0, 1, \ldots, n,$$

y la función de distribución binomial es:

$$P(X \leq x) = \sum_{i=0}^{n} \frac{n!}{i!\,(n-i)!} p^i\,(1-p)^{n-i}.$$

La media de X es np y su varianza es $np\,(1-p)$.

Hay una historia acerca de dos amigos, que eran compañeros de clase en la escuela superior, hablando de sus puestos de trabajo. Uno de ellos se convirtió en un experto en estadística y estaba trabajando en las tendencias poblacionales. Mostró una reedición de su antiguo compañero de clase. La reimpresión empezó, como es habitual, con la distribución Gaussiana, y el estadístico explicó a su antiguo compañero de clase el significado de los símbolos para la población real, para el promedio de la población, y así sucesivamente. Su compañero era un poco incrédulo y no estaba muy seguro de si el estadístico le estaba tomando el pelo.

"¿Cómo puedes saber éso?", fue su pregunta. "Y ¿cuál es este símbolo de aquí?"

"¡Oh!", dijo el experto en estadística, "este es π".

"¿Qué es eso?"

"La relación entre la circunferencia del círculo y su diámetro."

"Bueno, ahora está llevando su broma demasiado lejos", dijo el compañero de clase, "Seguramente la población no tiene nada que ver con la circunferencia del círculo."

—Eugene P. Wigner, 1960

Si el número de ocurrencias de un evento x por unidad (unidad de tiempo, longitud, área, volumen, o lo que sea) es, en promedio, un número real positivo λ, entonces el número real de tales ocurrencias es una variable aleatoria discreta X con una distribución de Poisson. La probabilidad de que X tome un valor x es:

$$P(X = x) = \frac{\lambda^x e^{-\lambda}}{x!}, \quad \text{para } x = 0, 1, \ldots, \infty,$$

y la función de distribución es:

$$P(X \leq x) = \sum_{i=0}^{x} \frac{\lambda^i e^{-\lambda}}{i!}.$$

Tanto la media como la varianza de la distribución de Poisson son iguales a λ.

La teoría de la probabilidad en el fondo no es más
que el sentido común reducido a cálculo.
Laplace, Pierre Simon de [1749-1827]

Socrates took poisson[a]
Desconocido

[a]Sócrates murió envenenado, por lo que el desconocido juega con la palabra poisson (aunque desafortunadamente, en francés, poison es veneno y poisson es pescado).

Siméon Denis Poisson [1781-1840]

Originalmente destinado por sus padres a estudiar medicina, Poisson en su lugar se inclinó por las matemáticas. Publicó su primer libro, sobre diferencias finitas, a la edad de 18 años.

Poisson enseñaba en la École Polytechnique desde 1802. En 1808 se convirtió en un astrónomo del Bureau des Longitudes. En 1809 fue nombrado presidente de matemáticas puras en la recién creada Faculté des Sciences.

Sus obras más importantes son sobre integrales definidas, series de Fourier, teoría de la probabilidad y mecánica.

En 1837 publicó una obra importante sobre probabilidad, donde introdujo la distribución de Poisson. Esta distribución describe la probabilidad de que un evento aleatorio ocurra en un tiempo o intervalo de espacio bajo las condiciones de que la probabilidad de que se produzca el evento es muy pequeña, pero el número de ensayos es muy grande, por lo que el evento acontece realmente unas pocas veces.

Su importante texto *Traité de mécanique* fue publicado en 1811 y de nuevo en 1833. Este libro ha sido la obra de referencia sobre mecánica durante muchos años.

El matemático Guglielmo Libri dijo de él: Su única pasión ha sido la ciencia: vivió y ha muerto por ella.

Una de las mejores citas conocidas de Poisson es: La vida es buena sólo para dos cosas, para el descubrimiento y para la enseñanza de las matemáticas.

Apéndice C

Curvas que llenan el espacio

Una curva de relleno de espacio llena totalmente parte del espacio pasando a través de cada punto en esa zona. Lo efectúa cambiando de dirección repetidamente. Sólo discutiremos las curvas que llenan parte del plano bidimensional, pero el concepto de un curva de relleno de espacio existe para cualquier número de dimensiones.

◇ **Ejercicio C.1 (sol. en pág. 1114):** Muéstrese un ejemplo de una curva de relleno de espacio en una dimensión.

Se conocen varias de tales curvas y todas se definen recursivamente. Una definición típica comienza con una curva simple C_0, muestra cómo utilizarla para construir otra curva más compleja C_1, y define la curva de relleno de espacio final, como el límite de la sucesión de curvas C_0, C_1, ...

C.1. La curva de Hilbert

(Véase la subsección 4.33.2.)

...El dibujado de esta curva se efectúa pues de forma recursiva. En la Figura C.1 se muestra un procedimiento para dibujar la orientación #4 de H_i. Éste realiza cuatro llamadas recursivas para extraer las cuatro curvas de orden $i-1$, y dibuja segmentos rectos entre las llamadas, para conectarlos (se utilizan los nombres A, B, C, y D en lugar de 1, 2, 3, y 4).

La Figura C.2 es un programa completo en Pascal para H_n, compilado mediante el compilador de pascal Metrowerks$^{\text{TM}}$ de Macintosh. Observe como el programa principal determina los valores iniciales del punto de partida (x, y) de la curva, y el tamaño del segmento h. La variable $h0$ define el tamaño, en píxeles, de la cuadrícula que contiene la curva, y debe ser una potencia de 2.

C.2. La curva de Sierpiński

(Véase la subsección 4.33.3.)

La Figura 4.167 puede obtenerse mediante el programa en pascal de la Figura C.10.

◇ **Ejercicio C.2 (sol. en pág. 1114):** La Figura 4.168 muestra tres iteraciones de la curva de relleno de Peano, desarrollada en 1890. Utilícense las técnicas desarrolladas anteriormente para las curvas de Hilbert y Sierpiński, para describir cómo se construye la curva de Peano. (Pista: Las curvas mostradas son P_1, P_2, y P_3. La primera curva, P_0, no se muestra en esta secuencia.)

```
PROCEDURE D(i: INTEGER);
BEGIN
IF i>0 THEN BEGIN
    A(i-1); x:=x-h; MoveTo(x,y);
    D(i-1); y:=y-h; MoveTo(x,y);
    D(i-1); x:=x+h; MoveTo(x,y);
    C(i-1);
    END;
END (*A*);
```

Figura C.1: Un procedimiento recursivo.

```
PROGRAM Hilbert; (* Curva de Hilbert *)
USES ScreenIO, Graphics;

CONST LB = 5; Width = 630; Height = 430;
(* LB=esquina inferior izda de ventana *)
n=6; (* n es el orden de la curva *)
h0=8; (* h0 debe ser una potencia de 2 *)

VAR h,x,y,x0,y0: INTEGER;

PROCEDURE B (i: INTEGER); FORWARD;
PROCEDURE C (i: INTEGER); FORWARD;
PROCEDURE D (i: INTEGER); FORWARD;

PROCEDURE A(i: INTEGER);
BEGIN
IF i>0 THEN BEGIN
  D(i-1); x:=x-h; MoveTo(x,y);
  A(i-1); y:=y-h; MoveTo(x,y);
  A(i-1); x:=x+h; MoveTo(x,y);
  B(i-1);
  END;
END (*A*);

PROCEDURE B(i: INTEGER);
BEGIN
IF i>0 THEN BEGIN
  C(i-1); y:=y+h; MoveTo(x,y);
  B(i-1); x:=x+h; MoveTo(x,y);
  B(i-1); y:=y-h; MoveTo(x,y);
  A(i-1);
  END;
END (*B*);

PROCEDURE C(i: INTEGER);
BEGIN
IF i>0 THEN BEGIN
  B(i-1); x:=x+h; MoveTo(x,y);
  C(i-1); y:=y+h; MoveTo(x,y);
  C(i-1); x:=x-h; MoveTo(x,y);
  D(i-1);
  END;
END (*C*);

PROCEDURE D(i: INTEGER);
BEGIN
IF i>0 THEN BEGIN
  A(i-1); y:=y-h; MoveTo(x,y);
  D(i-1); x:=x-h; MoveTo(x,y);
  D(i-1); y:=y+h; MoveTo(x,y);
  C(i-1);
  END;
END (*D*);

BEGIN (* Principal *)
  OpenGraphicWindow
   (LB,LB,Width,Height,'Curva de Hilbert');
  SetMode(paint);
  h:=h0; x0:=h DIV 2; y0:=x0; h:=h DIV 2;
  x0:=x0+(h DIV 2); y0:=y0+(h DIV 2);
  x:=x0+400; y:=y0+350; SetPen(x,y);
  A(n);
  ScBOL;
  ScWriteStr('Pulse una tecla & cierre la
ventana para salir');
  ScFreeze;
END.
```

Figura C.2: Programa en pascal para una curva de Hilbert de orden i.

```
program Sierpinski;

USES ScreenIO, QuickDraw;

PROCEDURE SierpinskiP(size,number:word);
(* Adaptado desde Wirth, 2nd ed, p.115 *)

VAR
H, x, y: INTEGER;

PROCEDURE B (level: word); FORWARD;
PROCEDURE C (level: word); FORWARD;
PROCEDURE D (level: word); FORWARD;

PROCEDURE A (level: word);
BEGIN
IF level > 0 THEN
BEGIN
A(level-1); Line(H,-H);
IF ScTellInput()>0 THEN BEGIN END;
B(level-1); Line(2*H,0);
IF ScTellInput()>0 THEN BEGIN END;
D(level-1); Line(H,H);
IF ScTellInput()>0 THEN BEGIN END;
A(level-1)
END
END {A};

PROCEDURE B (level: word);
BEGIN
IF level > 0 THEN
BEGIN
B(level-1); Line(-H,-H);
IF ScTellInput()>0 THEN BEGIN END;
C(level-1); Line(0,-2*H);
IF ScTellInput()>0 THEN BEGIN END;
A(level-1); Line(H,-H);
IF ScTellInput()>0 THEN BEGIN END;
B(level-1)
END
END {B};

PROCEDURE C (level: word);
BEGIN
IF level > 0 THEN
BEGIN
C(level-1); Line(-H,H);
IF ScTellInput()>0 THEN BEGIN END;
D(level-1); Line(-2*H,0);
IF ScTellInput()>0 THEN BEGIN END;
B(level-1); Line(-H,-H);
IF ScTellInput()>0 THEN BEGIN END;
C(level-1)
END
END {C};

PROCEDURE D (level: word);
BEGIN IF level > 0 THEN
BEGIN
D(level-1); Line(H,H);
IF ScTellInput()>0 THEN BEGIN END;
A(level-1); Line(0,2*H);
IF ScTellInput()>0 THEN BEGIN END;
C(level-1); Line(-H,H);
IF ScTellInput()>0 THEN BEGIN END;
D(level-1)
END
END {D};

VAR
level: WORD;
BEXIT : BOOLEAN;

BEGIN (* SierpinskiP *)
level := 0;
H := size DIV 4;
x := 2 * H;
y := 3 * H;
    BEXIT := FALSE;
REPEAT
level := level + 1;; x := x - H;;
H := H DIV 2; y := y + H;;
MoveTo (x, y);
A (level); Line (H, -H);
IF ScTellInput()>0 THEN BEXIT:=TRUE;
B (level); Line (-H, -H);
IF ScTellInput()>0 THEN BEXIT:=TRUE;
C(level); Line (-H, H);
IF ScTellInput()>0 THEN BEXIT:=TRUE;
D (level); Line (H, H);
IF ScTellInput()>0 THEN BEXIT:=TRUE;
IF level = number THEN BEXIT:=TRUE;
  UNTIL BEXIT;
END {SierpinskiP};

VAR
ch: CHAR;
prop: termProp;

BEGIN
ScOpenWindow( 10, 10, 400, 400 );
ScGetProp (prop);
prop. showCurs := FALSE;
ScSetProp(prop); (*cursor alpha oculto*)
ScClear;
SierpinskiP(400,1);ScBeep (1);ScFreeze;
SierpinskiP(400,2);ScBeep (1);ScFreeze;
SierpinskiP(400,3);ScBeep (1);ScFreeze;
SierpinskiP(400,4);ScBeep (1);ScFreeze;
SierpinskiP(400,5);ScBeep (1);ScFreeze;
ScClose;
END {Sierpinski}.
```

Figura C.3: Programa en pascal para curvas de Sierpiński de orden $1-5$.

C.3. Recorrido de la curva de Hilbert

(Véase la subsección 4.33.4.)

◇ **Ejercicio C.3 (sol. en pág. 1114):** Utilícese la Tabla 4.170 para calcular el número de nodo, del nodo H_4, cuyas coordenadas son $(13, 6)$.

◇ **Ejercicio C.4 (sol. en pág. 1114):** (Fácil.) Muéstrese cómo aplicar este método para recorrer la orientación #1 de H_i.

C.4. Recorrido de la curva de Peano

(Véase la subsección 4.33.5.)

◇ **Ejercicio C.5 (sol. en pág. 1114):** Síganse las reglas anteriores para obtener el RGC binario para $i = 3$.

C.5. Bibliografía relacionada

Backus, J. W. (1959) "The Syntax and Semantics of the Proposed International Algebraic Language," en *Proceedings of the International Conference on Information processing*, pp. 125−132, UNESCO.

Chomsky, N. (1956) "Three Models for the Description of Language," *IRE Transactions on Information Theory* **2**(3):113−124.

Cole, A. J. (1985) "A Note on Peano Polygons and Gray Codes," *International Journal of Computer Mathematics* **18**:3−13.

Cole, A. J. (1986) "Direct Transformations Between Sets of Integers and Hilbert Polygons," *International Journal of Computer Mathematics* **20**:115−122.

Gray, Frank (1953) "Pulse Code Communication," United States Patent 2,632,058, March 17.

Hilbert, D. (1891) "Ueber Stetige Abbildung Einer Linie auf ein Flachenstuck," *Math. Annalen* **38**:459−460.

Lindenmayer, A. (1968) "Mathematical Models for Cellular Interaction in Development," *Journal of Theoretical Biology* **18**:280−315.

Matlab (1999) está en la URL `http://www.mathworks.com/`.

Naur, P. et al. (1960) "Report on the Algorithmic Language ALGOL 60," *Communications of the ACM* **3**(5):299−314, revisado en *Communications of the ACM* **6**(1):1−17.

Peano, G. (1890) "Sur Une Courbe Qui Remplit Toute Une Aire Plaine," *Math. Annalen* **36**:157–160.

Sierpiński, W. (1912) "Sur Une Nouvelle Courbe Qui Remplit Toute Une Aire Plaine," *Bull. Acad. Sci. Cracovie Serie* A:462−478.

Sagan, Hans (1994) *Space-Filling Curves*, New York, Springer Verlag.

Wirth, N. (1976) *Algorithms + Data Structures = Programs*, Englewood Cliffs, NJ, Prentice-Hall, 2nd Ed.

`Todas las letras van y vienen—la L está aquí para quedarse.`
`Grzegorz Rozenberg`

Apéndice D

Estructuras de datos

Un programa de ordenador es un conjunto de instrucciones (o sentencias) que especifican operaciones sobre ítems (o elementos) de datos. A veces, los ítems de datos son independientes y cada uno se almacena por separado en la memoria. En tal caso, cada ítem se convierte en una *variable* en el programa y se le da un nombre. Un conjunto de elementos de datos pueden, sin embargo, estar relacionados, y tales elementos deben ser almacenados juntos en la memoria como una *estructura de datos*. Ejemplos son:

1. Una base de datos con información sobre personas. Cada elemento de la base de datos puede consistir en un nombre (nombre, primer apellido, y segundo apellido), un número de identificación, una dirección, y otros datos como la edad, el salario, o el título. El nombre puede ser almacenado en un arreglo (matriz o array). El número de identificación se convierte en una variable entera, la dirección se almacena en otro arreglo, y así sucesivamente. Estas matrices y variables se agrupan para convertirse en un nodo en una lista enlazada.

2. Una matriz de números, tales como las matrices de cuantificación usadas en ciertos métodos de compresión de imágenes. Tales números pueden ser almacenados en un arreglo bidimensional.

Los arreglos y las listas son ejemplos de *estructuras de datos*. Dos cosas deben ser descritas en una estructura de datos, la forma en que los ítems de datos están relacionados, y las operaciones que el programa debe ser capaz de realizar sobre la estructura y los ítems de datos. Ejemplos de operaciones son la inserción/eliminación de un elemento, el reemplazo de un elemento, la búsqueda de un elemento, y el aumento/disminución del tamaño de la estructura. Una vez que el programador sabe cómo están relacionados los ítems de datos y qué operaciones son necesarias, la estructura de datos puede ser diseñada. Las estructuras de datos son a veces sencillas, tal como un arreglo, una pila, o una lista, pero puede ser muy complejas, ya que una estructura puede combinar listas, pilas, tablas hash, y arreglos de una manera compleja.

La estructura de datos principal que se describe en este capítulo es la tabla hash, pero comenzamos con un breve estudio de algunas estructuras de datos básicas.

D.1. Arrays (arreglos o matrices)

Un array es una estructura de datos común, útil e importante. Intuitivamente, podemos pensar en un array como un conjunto de posiciones de memoria consecutivas agrupadas bajo un nombre, donde cada posición individual es accedida mediante su *índice*. Aunque ésto es una descripción práctica de los arreglos, no es una buena definición, ya que define esta estructura de datos por medio de su

representación real en la memoria. Una definición más formal es:

$$\textit{Un array es un conjunto de pares } (\textit{índice}, \textit{valor})\,.$$

Ésto define un arreglo como una asignación del conjunto de índices en el conjunto de valores.

En principio, el índice puede ser cualquier tipo de datos, pero en la práctica es normalmente un entero. En el viejo lenguaje de programación FORTRAN el primer elemento del array tiene el índice 1. En el lenguaje C el primer índice es 0, y en Pascal el programador puede especificar el primer índice del array. La mayoría de los lenguajes de programación requieren que el tamaño de todos los arrays sean estáticos, i.e., el tamaño de un array no puede ser cambiado en tiempo de ejecución.

Un programa puede utilizar muchos ítems de datos y es conveniente almacenarlos en un array, ya que así el programador tiene que memorizar un único nombre, el nombre del array, para todos estos ítems. Sin embargo, cada ítem almacenado en un array tiene un índice, y el programador, o bien tiene que memorizar los índices, o bien utilizar una aplicación donde no sea necesario memorizar cada índice, o bien buscar en el array para localizar un ítem en particular.

Una característica importante de los arrays es que pueden tener más de una dimensión. Un array bidimensional $A\,[m,n]$ es una matriz. Del mismo modo, un array tridimensional $A\,[l,m,n]$ es una matriz tridimensional. Alternativamente, podemos pensar en ello como un array unidimensional de l elementos, cada uno de los cuales es una matriz bidimensional de $m \times n$.

A pesar de que un array bidimensional se puede considerar una matriz, físicamente se almacena en la memoria como un conjunto de ubicaciones consecutivas. El array puede ser almacenado en memoria fila por fila o columna por columna. La Figura D.1 muestra que en el primer caso, un elemento $A\,[i,j]$ en un array de $m \times n$ se almacena en la ubicación $(i-1)\,n+j$ desde el inicio del array (suponiendo que los índices de la fila y de la columna comienzan a partir de 1).

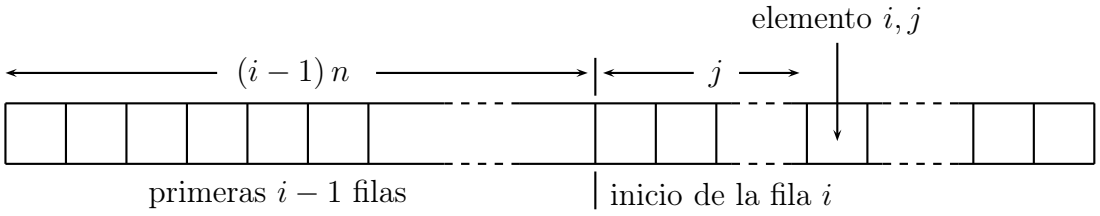


Figura D.1: Posición del elemento $A\,[i,j]$ del array.

◇ **Ejercicio D.1 (sol. en pág. 1114):** Supongamos que una matriz A de dimensiones $m \times n$ es almacenada en la memoria columna por columna y que los índices de la fila y de la columna comienzan en 0. ¿Cuál es la distancia, en la memoria, del elemento $A\,[i,j]$ del array desde el primer elemento $A\,[0,0]$ del array?

Un sencillo ejemplo de uso de un array es el almacenamiento de $n+1$ coeficientes de un polinomio de grado n en un array de tamaño $n+2$ ó más largo (véase la Ecuación (4.40) para la definición de polinomio). En este caso, el programador implícitamente conoce qué hay en cada ubicación del array.

◇ **Ejercicio D.2 (sol. en pág. 1114):** ¿Por qué el tamaño del array es $n+2$, y no $n+1$?

D.2. Pilas y colas

La pila es también una estructura de datos común. Intuitivamente pensamos en una pila como un contenedor abierto en un extremo. Las únicas operaciones en una pila son la inserción y la eliminación

(comúnmente conocidas como *push* —empujar— y *pop* —sacar—, respectivamente), y se realizan por el extremo abierto. La Figura D.2a muestra cómo se introducen tres ítems de datos en una pila vacía y como el único elemento que se puede sacar es el último que fue introducido. Por esta razón, una pila se refiere a veces como una estructura LIFO (*last-in first-out*; esto es, "el último en entrar es el primero en salir"). La pila se implementa normalmente como un array, pero debido a la manera en que se utiliza, el programa necesita seguir la pista de un único elemento, el de arriba (i.e., el último introducido), en cualquier momento dado. El programa, por lo tanto, mantiene un puntero llamado *top of stack* (tope de la pila) que apunta a ese ítem de datos.

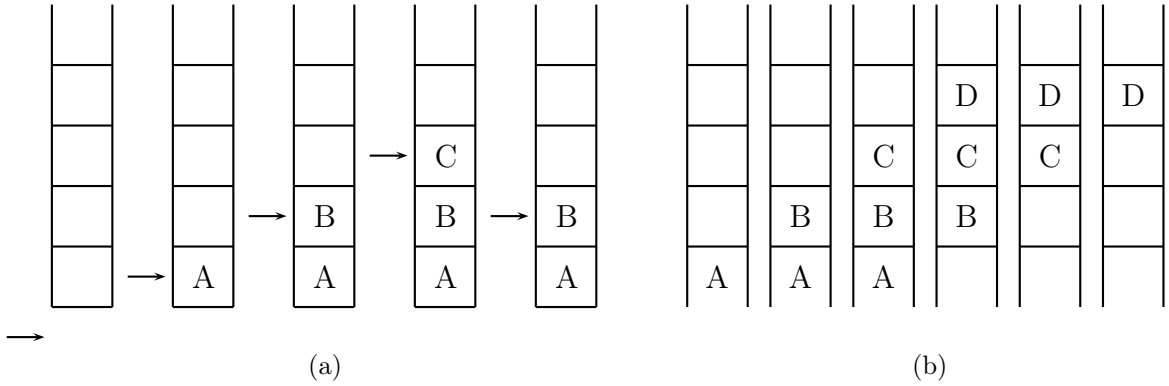


Figura D.2: Pila y cola.

Una *cola* es una estructura de datos, donde el único elemento que puede ser eliminado es el más antiguo. Una cola se basa por tanto en el principio FIFO (*first-in first-out*; esto es, "el primero en entrar es el primero en salir"). La Figura D.2b muestra una cola durante cuatro inserciones y tres eliminaciones. Es obvio que estas operaciones desplazan los datos desde el inicio hasta el final de la cola, que es por lo que la *cola circular* (discutida en la Sección 3.3.1) es una estructura de datos más útil que la cola lineal.

D.3. Listas

Una lista (o una lista enlazada) es una estructura de datos construida con nodos que apuntan el uno al otro. Un nodo puede ser una única variable, un array, una pila, otra lista, o cualquier otra estructura; pero la característica principal de las listas es el uso de *punteros*. Ésto hace que sea fácil de controlar el tamaño de la lista de forma dinámica. Un lenguaje de programación puede incluir declaraciones tales como `get_node` (para la construcción de un nuevo nodo a partir de los recursos de almacenamiento disponibles) y `put_node` (para devolver el almacenamiento utilizado por un nodo para su reutilización). La Figura D.3 muestra varias formas de organizar las listas. Una lista puede ser simplemente enlazada o doblemente enlazada, lo que puede ser cíclica, y los elementos de la lista pueden ser ellos mismos listas.

Una cola puede ser implementada como una lista enlazada. La inserción de un nuevo elemento se realiza creando un nuevo nodo y añadiéndolo a la lista. La eliminación de un elemento se efectúa suprimiendo el primero (el más antiguo) del nodo.

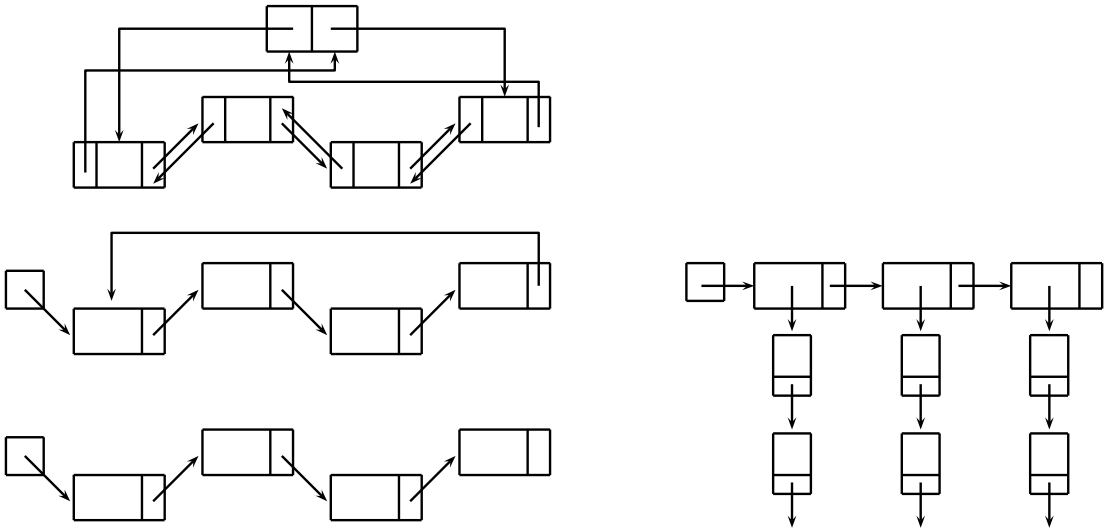


Figura D.3: Listas enlazadas.

D.4. Árboles

Un árbol puede ser definido de forma imprecisa como una estructura de datos formada por nodos conectados con aristas dirigidas, donde todos los nodos están conectados, no hay ciclos, y un nodo se considera especial. Este nodo se denomina *raíz* del árbol. La Figura D.4a muestra una estructura donde cualquiera de los nodos puede ser considerado la raíz. Observe la arista discontinua. La inclusión de esta arista en la estructura introduciría un ciclo, y por lo tanto, lo convertiría el árbol en un grafo general. La Figura D.4b muestra la misma estructura con el nodo a elegido como raíz y los nodos restantes reordenados para formar la forma familiar de un árbol. Si hay aristas que conducen directamente desde un nodo a a los nodos b, c, y d, entonces los tres nodos son llamados *hijos* de a y a es su padre. Si un nodo no tiene hijos, es una *hoja*. Un nodo que no es ni una hoja ni la raíz es un nodo *interior*. La *profundidad* (o nivel) de un nodo es la longitud de la trayectoria desde la raíz hasta el nodo. La propia raíz tiene profundidad cero. La altura de un árbol es la mayor profundidad (o, alternativamente, uno menos que el número de niveles del árbol). La Figura D.4b muestra que cada nodo en un árbol es la raíz de un subárbol (que puede ser vacío).

En una aplicación práctica típica, un nodo a es un corto array que contiene un ítem de datos y punteros. En la mayoría de los casos, la matriz contiene 1, 2, ó 3 punteros, cada uno señalando el comienzo de una lista enlazada. El primer puntero puede apuntar a una lista que contiene nodos que son hermanos de a (nótese que la raíz no tiene hermanos). El segundo puntero puede apuntar a una lista que contiene los nodos que son hijos de a (si a es un hoja, este puntero es nulo), y el tercer indicador puede apuntar a los padres de a (un puntero nulo, si a es la raíz). De esta manera es posible recorrer el árbol hacia la derecha (desde a hasta su siguiente hermano a la derecha), hacia abajo (desde a hasta su primer hijo), y hacia arriba (desde a hasta su padre). Dicho árbol puede también cambiar de forma dinámica, con nodos que son añadidos, eliminados y modificados.

A veces, es útil para añadir otro componente (o campo) al array, con un código marcando el nodo como existente o eliminado. De esta manera un nodo puede ser eliminado de forma efectiva del árbol estableciendo este campo a "eliminado", sin tener que efectuar un borrado real y cambiar punteros. Esta técnica se denomina "borrado perezoso" y es útil en aplicaciones en las que no hay necesidad de devolver nodos eliminados al repositorio de almacenamiento disponible.

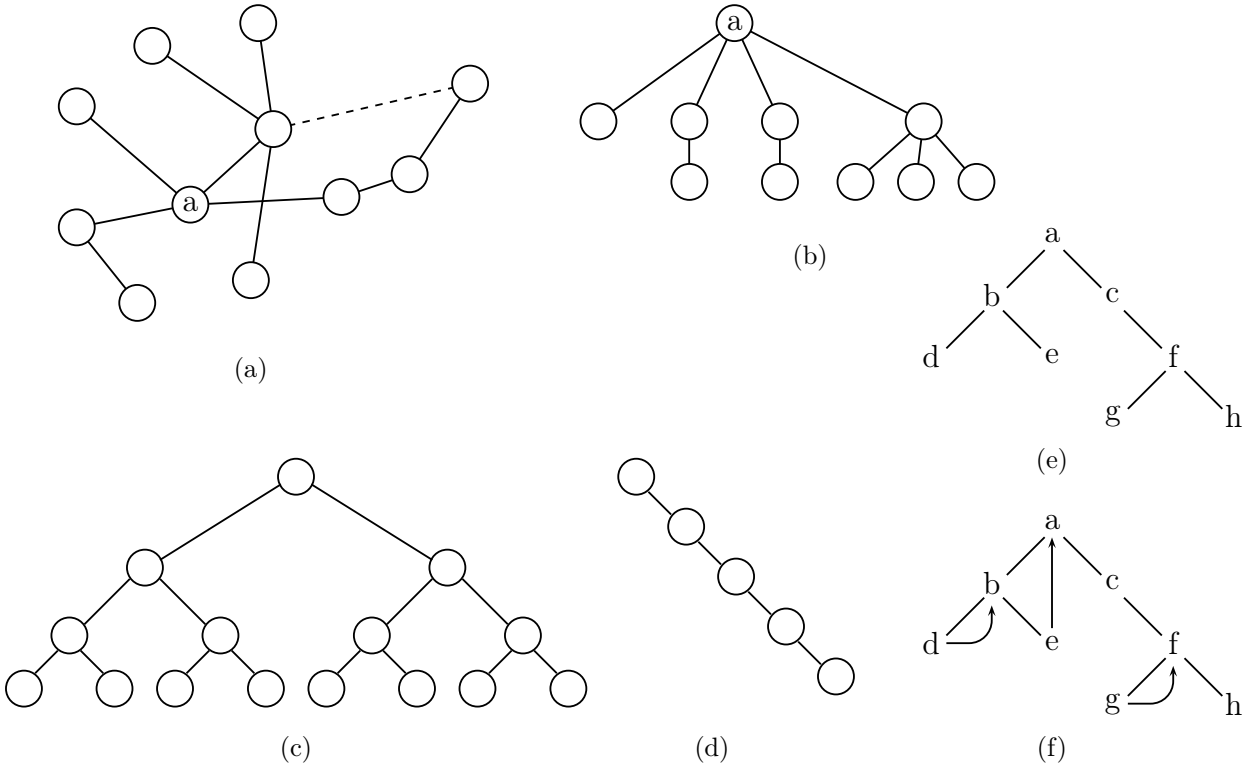


Figura D.4: Varios árboles.

Si un nodo puede tener sólo unos pocos hijos, puede ser posible implementar la totalidad del árbol en un array, sin ningún tipo de punteros. El ejemplo más simple es un árbol binario completo, donde los dos hijos del nodo a se almacenan en las posiciones $2a$ y $2a+1$ y el padre de a puede encontrarse en ubicación del array $\lceil a/2 \rceil$ (ésto se discute en la Sección 2.15).

◇ **Ejercicio D.3 (sol. en pág. 1114):** ¿Puede un árbol ternario ser implementado de esta manera?

Un árbol binario puede ser completo, sesgado, o cualquier otra cosa intermedia. Ésto se ilustra mediante la Figura D.4c,d,e.

Una operación importante en un árbol es el *recorrido*. Un recorrido sigue los punteros de manera que cada nodo del árbol es visitado una sola vez. Hay cuatro tipos de recorridos en árboles:

- *Post-orden*. Todos los hijos de un nodo son visitados, después es visitado el propio nodo. Esto se realiza recursivamente mediante las dos llamadas recursivas `postorden(L)`; `postorden(R)`; (donde L y R son los dos hijos de la raíz), seguidas por `visita(raíz)`. El recorrido del árbol en post-orden de la Figura D.4e es $(((D,E),B),(((G,H),F),C)),A$.

- *Pre-orden*: Se visita un nodo, después todos sus hijos.

- *En orden*: Éste se utiliza para recorrer un árbol binario. Visita el subárbol izquierdo de un nodo, luego el nodo, después su subárbol derecho.

- *Orden por niveles*: Visita todos los nodos del nivel L, posteriormente, procede con el nivel $L+1$.

◇ **Ejercicio D.4 (sol. en pág. 1114):** Muéstrense los recorridos del árbol de la Figura D.4e en pre-orden, en orden, y en orden por niveles.

Un recorrido de árbol se efectúa normalmente de forma recursiva, salvo que el recorrido en orden por niveles utilice una cola en lugar de una pila.

En una implementación sencilla, cada nodo de un árbol binario tiene dos punteros, para su hijos izquierdo y derecho. Es obvio que no se usan todos los punteros. Por ejemplo, en el árbol binario de la Figura D.4e hay 9 punteros sin utilizar (8 en las hojas y uno en el nodo C). Estos campos no aprovechados pueden ser ocupados por punteros extra llamados *hilos* (*threads*). Una forma de definir hilos es localizar un nodo A con un campo de un hijo derecho sin usar, y almacenar en este campo un puntero al sucesor de A considerando el recorrido en orden. Los tres hilos resultantes en el árbol de la Figura D.4e se muestran en la Figura D.4f. Obviamente, los hilos pueden ser definidos de manera diferente y puede ser muy útil en aplicaciones especiales, donde el árbol tiene que ser recorrido o buscado de manera no estándar. El precio de la adición de hilos es un bit adicional (un flag o bandera) en cada nodo A, indicando si en A está almacenado un puntero ordinario para el hijo derecho, o un hilo.

Imagine un nodo interior a en un árbol binario. Éste tiene dos hijos, l y r, que son las raíces de los subárboles izquierdo y derecho de a, respectivamente. Si l tiene el mayor valor de datos de su subárbol y r satisface el mismo hecho para su subárbol, entonces el árbol binario se llama *montón* (*heap*).

Un árbol binario de búsqueda es un tipo de árbol importante. En tal árbol, todos los nodos del subárbol izquierdo de un nodo a tienen valores de datos que son menores que los valores de datos de a. Similarmente, todos los nodos en el subárbol derecho de a tienen valores de datos mayores que a. Si los datos no son números, las relaciones "menor que" y "mayor que" tienen que ser definidas. Los árboles binarios de búsqueda se describen y utilizan en la Sección 3.4. El principal uso de tales árboles es la búsqueda rápida. La búsqueda de un nodo en un árbol binario de búsqueda toma no más de H pasos más, donde H es la altura del árbol.

Imagine un árbol de búsqueda binaria que comienza vacío. Cuando los nodos se insertan en el árbol, crece. El orden en el que los nodos se insertan determina la forma del árbol. Si los nodos que están siendo insertados tienen valores de datos aleatorios, el árbol resultante será balanceado, i.e., muy similar a un árbol completo. Su altura será aproximadamente $\log_2 n$, donde n es el número de nodos. La búsqueda en tal árbol binario de búsqueda con un millón de nodos toma no más de 20 pasos. Si, por otra parte, los nodos nuevos tienen valores de datos monótonos (ascendentes o descendentes), el árbol terminará siendo sesgado. Su altura (y por tanto el tiempo máximo de búsqueda) será n.

Los árboles binarios de búsqueda son de uso común en aplicaciones que requieren muchas búsquedas, por lo que es deseable encontrar una manera de mantener dicho árbol tan balanceado como sea posible, independientemente del orden de las inserciones de los nodos. El árbol AVL (llamado así por sus dos desarrolladores, Adelson-Velskii y Landis) es tal estructura de datos. Cada nodo en un árbol AVL tiene un *factor de balanceo*, definido como la altura de su subárbol izquierdo menos la altura de su subárbol derecho. A este factor de balanceo se le permite tomar sólo los tres valores 0, 1, y -1, i.e., las alturas de los dos subárboles de cualquier nodo pueden diferir en no más de 1.

Cuando el factor de balanceo de un nodo se hace mayor que 1, decimos que el nodo está "fuera de balance a la izquierda;" cuando se hace menor que -1, decimos que está "fuera de balance a la derecha." Estas situaciones se corrigen inmediatamente, y el nodo vuelve a estar de nuevo balaceado. Un árbol AVL es un caso especial de un *árbol binario de altura balanceada*, una estructura definida por:

1. Un árbol binario vacío es de altura balanceada.

2. Un árbol binario no vacío es de altura balanceada si sus subárboles izquierdo y derecho son de altura balanceada con factores de balanceo de 0, 1, ó -1.

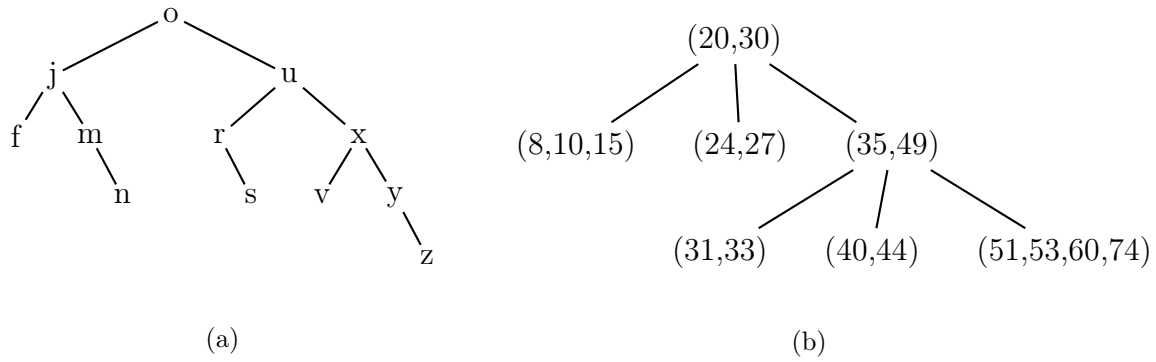


Figura D.5: (a)Un árbol AVL. (b) Un árbol B.

Un árbol AVL es un árbol binario de *búsqueda* balanceado en altura, un caso especial de un árbol binario de altura balanceada. La Figura D.5a muestra un ejemplo de árbol AVL que ilustra una cualidad importante de estos árboles, a saber, no tienen que ser completos. Las hojas de un árbol AVL no tienen que estar en el mismo nivel o incluso en niveles adyacentes. Sin embargo, se puede demostrar que un árbol AVL no es muy diferente de un árbol binario completo porque su altura puede ser a lo sumo $\sqrt{2}\log_2 n$. Un árbol AVL se crea, ya sea vacío, ya sea con 1 nodo, por lo que inicialmente es de altura balanceada. Se mantiene balanceado en altura después de cada inserción y eliminación mediante ajustes especiales (llamados *rotaciones*) que restablecen el equilibrio. Los detalles se pueden encontrar en textos sobre estructuras de datos.

El último tipo de árbol que se menciona aquí es el árbol B. Éste es un tipo de árbol importante porque se utiliza por muchos sistemas operativos para mantener el directorio de archivos de un disco. El crecimiento de un árbol B está especialmente controlado para mantenerlo bien balanceado, dando lugar a búsquedas rápidas, pero un árbol B, a pesar de su nombre, no es un árbol binario.

Un nodo en un árbol B puede tener varios hijos, y es esta propiedad, junto con que el árbol debe estar bien balanceado, lo que hace que sea un candidato natural para un directorio de un disco. El siguiente ejemplo extremo muestra por qué. Imagine un árbol en el que cada nodo puede tener 100 hijos. Si el árbol tiene una altura de 4, puede contener hasta

$$1 + 100 + 100^2 + 100^3 = 1\,010\,101$$

nodos. Un nodo entre más de un millón puede ser localizado en, a lo sumo, ¡cuatro pasos! El precio de ésto es, por supuesto, una estructura de nodos compleja, permitiendo hasta 100 hijos. En un directorio del disco cada nodo se guarda en el disco como un bloque. El tamaño de un bloque del disco varía, pero es normalmente de unos pocos cientos de bytes. Por lo tanto, un bloque tiene espacio para mucha información en forma de claves, punteros, y banderas (flags). Al mismo tiempo, un acceso a disco es mucho más lento que el acceso a la memoria, por lo que encontrar un ítem del directorio debe involucrar el menor número de accesos a disco posible. Una vez que el disco es accedido, un bloque entero es cargado en la memoria. En la memoria, puede buscarse el bloque y sus datos pueden ser procesados rápidamente.

Un nodo en un árbol B contiene un campo contador, un puntero a una lista de entradas (registros que deben ser buscados, donde cada registro tiene una clave y posiblemente algunos otros datos), y un puntero a una lista de ramas. Las listas de las entradas y las ramas deben estar ordenadas, pero pueden optar por cualquier estructura ordenada, como listas enlazadas, arrays o árboles de búsqueda binaria. El campo contador m contiene el número de entradas (m) y el número de ramas ($m + 1$).

La Figura D.5b es un árbol B sencillo. Sólo muestra la lista de entradas, y para cada entrada muestra una única clave. Ésto es suficiente para comprender cómo se busca en el árbol. La raíz contiene las

claves 20 y 30. Esto significa que la rama 0 desde la raíz conduce a una lista de entradas para todas las entradas con las claves que son menores que 20. La rama 1 desde la raíz conduce a una lista de entradas para todas las entradas con claves en el rango $[20, 30)$ y la rama 2 lleva a la lista de entradas con las claves ≥ 30. Para buscar 34, por ejemplo, empezamos desde la raíz, tomamos la rama 2 (ya que $34 \geq 30$), luego la rama 0 (ya que $34 < 35$). Llegamos al nodo con la lista de entradas $(31, 33)$ y tomamos la rama 2 (ya que $34 \geq 33$). Ésto nos lleva a un nodo nulo, que es como descubrir que la clave 34 no forma parte del árbol.

D.5. Grafos

Un grafo es una estructura de datos general formada por nodos (o vértices) y aristas (o arcos) que los conectan. El grafo es una estructura general porque no todos los nodos tienen que estar conectados y ningún nodo tiene que ser especial (tal como la cabeza o raíz del grafo). La Figura D.6 muestra ejemplos de grafos. Un grafo puede incluso estar formado por varias unidades desconectadas. Las aristas pueden ser dirigidas o no dirigidas, y pueden tener etiquetas (indicando los pesos o costes) asociados con ellas. Las principales operaciones en los grafos son las siguientes:

1. Construir un grafo g vacío.
2. Insertar un nodo a en un grafo g existente.
3. Construir una arista e entre los nodos a y b del grafo g.
4. Eliminar un nodo a de g. Todas las aristas adyacentes al nodo también deben ser eliminadas.
5. Eliminar la arista e entre los nodos a y b del grafo g.
6. Determinar si el grafo g está vacío.
7. Construir una lista de todas las aristas adyacentes a un nodo a de g.
8. Recorrer el grafo g (i.e., visitar cada nodo una sola vez).
9. Encontrar el camino de coste mínimo que conduce desde el nodo a al nodo b en g.

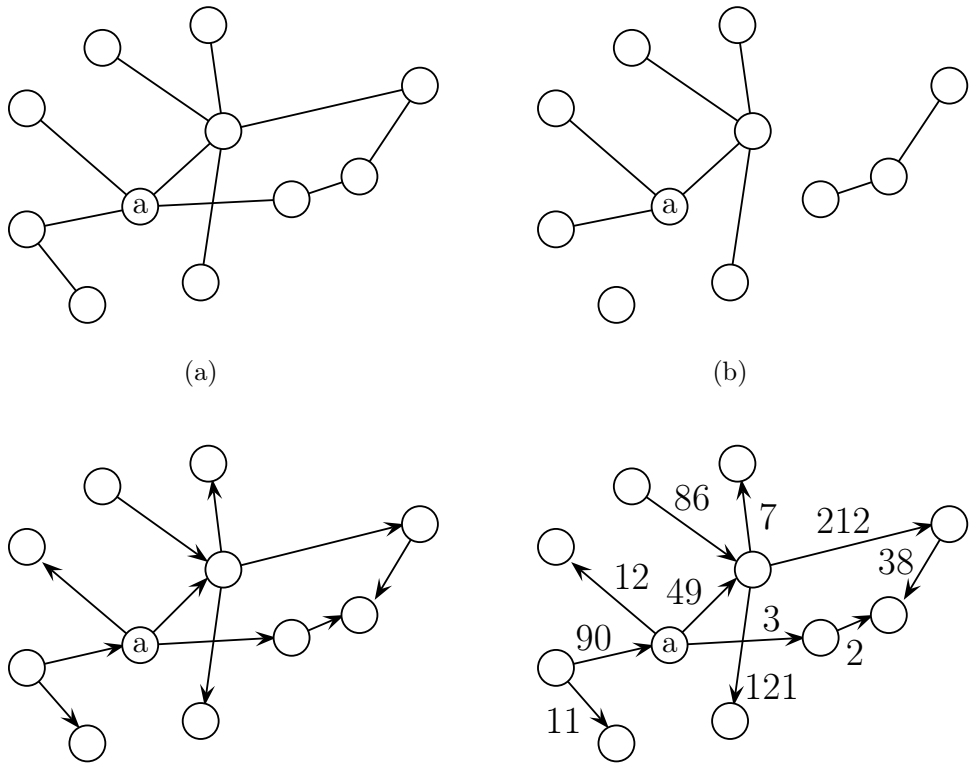
Puede haber otras operaciones, especializadas para aplicaciones específicas.

En un grafo no dirigido, una arista entre a y b es adyacente a ambos nodos. En un grafo dirigido, una arista de a a b se dice que es "adyacente desde a" y "adyacente a b".

D.6. Hashing

Una tabla hash es una estructura de datos que permite búsquedas rápidas, inserciones y eliminaciones de ítems de datos. La tabla misma es precisamente un array H, y el principio de hashing es definir una función h tal que $h(k)$ produzca un índice al array H, donde k es la clave de un ítem de datos. Los siguientes ejemplos ilustran el significado de los términos "ítem (o elemento) de datos" y "clave".

1. El método LZRW1 (Sección 3.10), emplea hashing para almacenar punteros. El método utiliza los tres primeros caracteres en el búfer de preanálisis como una clave que es pasada a una función hash para que produzca un número I de 12 bits que se usa para indexar la tabla hash, un array de $2^{12} = 4\,096$ punteros. El dato real almacenado en cada ubicación de la tabla hash de LZRW1 es un puntero.



(a) (b) (c) (d)

Figura D.6: Varios grafos.

2. Prácticamente todos los lenguajes de programación utilizan variables. Una variable proporciona un nombre para un valor que será almacenado en la memoria, en una cierta dirección A, cuando el programa es ejecutado eventualmente. Cuando se compila el programa, cada variable tiene dos atributos, su nombre N (una cadena de caracteres asignada por el programador) y su dirección de memoria A, asignada por el compilador. El compilador utiliza una tabla hash para almacenar toda la información sobre las variables. El ítem de datos en este ejemplo es la dirección A de una variable; la clave es el nombre de la variable N. El compilador lee el nombre desde el archivo fuente del programa, lo envía a la función hash, lo encuentra en la tabla hash, y recupera la dirección con el fin de compilar la instrucción en curso. Si la variable no se encuentra en la tabla hash, se le asigna una dirección, y ambos —nombre y dirección— se almacenan en la tabla (en principio, sólo necesita ser almacenada la dirección, pero es el nombre también se almacena debido a las colisiones; véase más adelante).

La función hash h toma como argumento una clave, que puede ser un número o una cadena. Reordena (scrambles[1]) o efectúa operaciones hash con los bits de la clave para producir un índice del array H. En la práctica, el tamaño del array es normalmente 2^n, por lo que el resultado producido por h debe ser un número de n bits. Hashing es una buena estructura de datos, ya que cualquier operación sobre la tabla hash —la adición, la búsqueda, o el borrado— puede realizarse en un solo paso, sin importar el tamaño de la tabla. El único problema son las *colisiones*. En la mayoría de aplicaciones,

[1]Scramble es embrollar, desorganizar para obtener una ordenación de la que se pueda sacar provecho.

es posible que la función hash proporcione el mismo índice para dos claves distintas k_1 y k_2, i.e., $h(k_1) = h(k_2)$ para $k_1 \neq k_2$. El ejemplo 2 anterior hace que sea fácil de comprender la razón de ésto. Suponiendo que los nombres de las variables consten de cinco letras, puede haber $26^5 = 11\,881\,376$ nombres de variables. Cualquier programa usaría sólo un pequeño porcentaje de este número, tal vez unos pocos cientos o unos pocos miles de nombres. El tamaño de la tabla hash, por lo tanto, no tiene que exceder de unos pocos miles de entradas y las operaciones hash sobre 11,8 millones de nombres producen unos miles de valores de índice que deben involucrar a muchas colisiones.

◇ **Ejercicio D.5 (sol. en pág. 1114):** ¿Cuántos nombres son posibles si un nombre está formado por exactamente ocho letras y dígitos?

Terminología: Dos claves diferentes que una operación hash lleva al mismo índice se llaman *sinónimas*. Si una tabla hash contiene m claves de un conjunto de M posibles, entonces m/M es la *densidad* de la tabla, y $\alpha = m/2^n$ es su *factor de carga*.

D.7. Funciones hash

Una función de hash debe ser fácil de calcular y debe minimizar las colisiones. La función debe hacer uso de todos los bits de la clave, de tal manera que el cambio incluso de un bit normalmente (aunque no siempre) genere un índice diferente. Una función hash ideal también debe producir índices que estén uniformemente distribuidos (la llamada a la función muchas veces con claves aleatorias debe producir cada índice el mismo número de veces). Una función que produce, e.g., el índice 118 la mayor parte del tiempo es evidentemente sesgada y lleva a colisiones. La función también debe asumir que muchas claves pueden ser similares. En el caso de los nombres de las variables, e.g., a veces los programadores asignan nombres como "`A1`", "`A2`", "`A3`" a las variables. Una función de hash que utiliza sólo los bits de más a la izquierda de una clave produciría el mismo índice para tales nombres, llevando a muchas colisiones. Los siguientes son algunos ejemplos de funciones hash usadas en la práctica.

Mid-Square (Mitad–Cuadrado): La clave k se considera un número entero, se eleva al cuadrado y los n bits de la mitad de k^2 son extraídos para convertirse en el índice. Elevar k al cuadrado tiene la ventaja de que el los bits de la mitad de k^2 dependerán de *todos* los bits de k. En consecuencia, dos claves que difieren en un bit tenderían a producir índices diferentes. Una variación, adecuada para claves largas, es dividir los bits de la llave original en varios grupos, agregar todos los grupos, elevar al cuadrado el resultado, y extraer sus n bits centrales.

Las claves "`A1`", "`A2`", y "`A3`", e.g., se convierten en los números de 16 bits:

$$01000001 \mid 00110001, \quad 01000001 \mid 00110010, \quad \text{y } 01000001 \mid 00110011.$$

Después de elevar al cuadrado y extraer los 8 bits centrales, los índices resultantes son 158, 166, y 175, respectivamente.

Modulo: $h(k) = k \bmod m$. El resultado es el resto de la división entera k/m, un número en el rango $[0, m-1]$. Para que el resultado sea un índice válido, el tamaño de la tabla hash debe ser m. El valor de m es crítico y debe ser seleccionado cuidadosamente. Si m es una potencia de 2, digamos, 2^i, entonces el resto de k/m es simplemente los i bits del extremo derecho de k. Ésta sería una función hash muy sesgada. Si m es par, entonces el resto de k/m tiene la misma paridad que k (es impar cuando k es impar, y par cuando k es par). De nuevo, ésta es una mala opción para m, ya que produce una función hash sesgada que mapea claves impares a ubicaciones impares de H y claves pares a ubicaciones pares. Si p es un número primo que divide uniformemente m entonces las claves que son permutaciones una de la otra (e.g., "`ABC`", "`ACB`", y "`CBA`"), a menudo pueden ser asignadas a índices que difieren en p o en un múltiplo de p, provocando de nuevo la distribución no uniforme de las claves.

Puede demostrarse que la función hash módulo produce los mejores resultados cuando m es un número primo que no divide uniformemente $8^a \pm b$, donde a y b son números pequeños. En la práctica, una buena opción para m son números primos cuyos divisores primos son > 20.

Folding (Plegado): Esta función es adecuada para claves largas. Los bits que constituyen la clave se dividen en varios grupos, que son agregados. Los n bits centrales de la suma se extraen para convertirse en el índice. Una variación es el *plegado inverso* donde cada otro grupo de bits se invierte antes de ser añadido.

> No, no. Mira, aquí al lado está el hachís (*hash*) porque no sabía qué cantidad tomó.
>
> –Amy Wright como Shelley en *Stardust Memories* (1980).

D.8. Tratamiento de las colisiones

Cuando un índice i es generado por la función hash $h(k)$, el software que utiliza el hashing primero debe comprobar si $H[i]$ provoca una colisión. Debe haber, por lo tanto, una forma para que el software pueda saber si la entrada $H[i]$ está vacía u ocupada. La inicialización de todas las entradas de H a cero normalmente no es suficiente, ya que cero puede ser un ítem de datos válido. Un enfoque simple consiste en tener un array adicional F, de tamaño $2^n/8$ bytes, donde cada bit esté asociado con una entrada de H. Cada bit de F actúa como un indicador de si la entrada correspondiente de H está vacía o no. El array completo F se establece inicialmente a ceros, lo que indica que todas las entradas de H están vacías. Cuando el software decide insertar un ítem de datos en $H[i]$ tiene que encontrar el bit en F que corresponda a la entrada i y comprobarlo. El programa, por tanto, calcula $j = [i/8]$, $k = i - 8j$, y comprueba el bit k del byte $F[j]$. Si el bit es cero, la entrada $H[i]$ está vacía y se puede utilizar para un nuevo ítem de datos. El bit, entonces tiene que ser actualizado a 1, lo que se realiza usando k para seleccionar una de las ocho máscaras:

00000001 00000010 00000100 00001000 00010000 00100000 01000000 10000000

y efectuando la operación lógica OR con $F[j]$. Si el bit es 1, la entrada $H[i]$ ya está ocupada, y ésta es una colisión. El software debe ser capaz de verificar y determinar si la entrada $H[i]$ contiene el ítem de datos que corresponde a la clave k. Es por ésto que las claves tienen que ser guardadas, junto con los elementos de datos, en la tabla hash.

¿Qué debe hacer el software en caso de colisión? Lo más sencillo es comprobar las entradas $H[i+1]$, $H[i+2]$, ..., $H[2^n - 1]$, $H[0]$, $H[1]$,... hasta encontrar una entrada vacía o hasta que la búsqueda alcance la entrada $H[i-1]$. En el último caso, el software sabe que el ítem de datos no está en la tabla (si se trataba de una búsqueda de un elemento) o que la tabla está llena (si se trataba de un intento de insertar un nuevo elemento en la tabla). Este proceso se denomina *búsqueda lineal*. La búsqueda de un ítem de datos, que en principio debería tomar un paso, puede ahora, debido a las colisiones, necesitar hasta 2^n pasos. La experiencia también demuestra que una búsqueda lineal genera entradas ocupadas en la tabla de clústeres, lo que es intuitivamente fácil de entender. Si la función hash no es ideal y lleva muchas claves hash, digamos, al índice 54, entonces las entradas de la tabla 54, 55,... se llenarán rápidamente, creando un clúster. Los clústeres también tienden a crecer y fusionarse, creando clústeres aún mayores y aumentando así el tiempo de búsqueda. Un análisis teórico muestra que el número esperado de pasos necesario para localizar un elemento cuando se usa la búsqueda lineal es $(2-\alpha)/(2-2\alpha)$, donde α es el factor de carga (tanto por ciento de la tabla completa). Para $\alpha = 0{,}5$,

podemos esperar 1,5 pasos en promedio, pero para $\alpha = 0{,}75$ el número esperado de pasos se eleva a 2,5, y para $\alpha = 0{,}9$ se transforma en 5,5. Es evidente que cuando se utiliza la búsqueda lineal, el factor de carga debe mantenerse bajo (quizás por debajo de 0,6 a 0,7). Si se necesitan añadir más elementos a la tabla, una buena solución es declarar una tabla nueva, dos veces más grande que la original, transferir todos los elementos de la tabla antigua a la nueva (utilizando una nueva función hash), y eliminar la vieja tabla.

> La cena era a la una; y el lunes, martes y miércoles consistía en ternera, carne asada, "hashed", y "minced", y el jueves, viernes, y sábado en cordero. El domingo comieron uno de sus propios pollos.
>
> —W. Somerset Maugham, *Of Human Bondage*[a]
>
> [a]Sobre el cautiverio humano

Un método más sofisticado para manejar las colisiones es la *búsqueda cuadrática*. Supongamos que H es un array de tamaño N. Cuando se localiza la entrada $H[i]$ para ser ocupada, el software comprueba las entradas $H\left[\left(i \pm j^2\right) \bmod N\right]$ donde $0 \leq j \leq {}^{(N-1)}/_{2}$. Se puede demostrar que si N es un número primo de la forma $4j+3$ (donde j es un número entero) la búsqueda cuadrática terminará examinando cada entrada de H.

Una tercera manera de tratar las colisiones es repetir la operación hash. El software debe poder elegir entre varias funciones hash h_1, h_2,.... Si $i = h_1(k)$ y $H[i]$ está ocupada, el software debe calcular $i = h_2(k)$ y luego probar el nuevo $H[i]$. Todavía otra manera es generar un array R de N números pseudo–aleatorios en el rango $[0, N-1]$, donde cada número aparece una vez. Si la entrada $H[i]$ está ocupada, el software debe establecer $i = (i + R[i]) \bmod N$ y probar el nuevo $H[i]$.

Es posible diseñar una *función hash perfecta* que, para un conjunto dado de ítems de datos, no tenga ninguna colisión. Ésto tiene sentido para grupos de datos que nunca cambian. Algunos ejemplos son la Biblia, las obras de Shakespeare, o cualquier dato escrito en un CD-ROM. El tamaño N de la tabla hash debería, en tal caso, ser normalmente mayor que el número de ítems de datos. También es posible diseñar una *función hash mínima perfecta* donde el tamaño de la tabla hash es igual al tamaño de los datos (es decir, no hay entradas permanecen vacías después de que todos los datos han sido insertados). Véase [Czech 92], [Fox 91], y [Havas 93] en la bibliografía de este apéndice, para más detalles sobre estas funciones hash especiales.

D.9. Bibliografía relacionada

Czech, Z. J., et al. (1992) "An Optimal Algorithm for Generating Minimal Perfect Hash Functions," *Information Processing Letters* **43**:257−264.

Fox, E. A. et al. (1991) "Order Preserving Minimal Perfect Hash Functions and Information Retrieval," *ACM Transactions on Information Systems* **9**(2):281–308.

Havas, G. et al. (1993) *Graphs, Hypergraphs and Hashing* in Proceedings of the International Workshop on Graph-Theoretic Concepts in Computer Science (WG'93), Berlin, Springer-Verlag.

> ‘‘Por qué’’, dijo él, ‘‘un mago puede llamar a un montón de genios, y ellos lo elevarían[a] como si nada antes de que pudiera decir Jack Robinson. Son tan altos como un árbol y tan gruesos como una iglesia.’’
>
> Mark Twain, *Las aventuras de Huckleberry Finn*

[a] El original en inglés es: ‘‘would hash you up’’.

Apéndice E

Códigos correctores de errores

El problema de añadir fiabilidad a los datos ya ha sido mencionado en la Sección 2.12. Este apéndice describe métodos generales para detectar y corregir errores. La fiabilidad es, en cierto sentido, lo contrario a la compresión de datos, ya que se consigue *aumentando la redundancia de los datos*. No obstante, muchas situaciones en la práctica demandan datos fiables, por lo que un buen programa de compresión de datos debe ser capaz de utilizar códigos para incrementar su fiabilidad, si es necesario.

Cada vez que se transmite información, en cualquier canal, puede corromperse por el ruido. De hecho, incluso cuando la información se guarda en un dispositivo de almacenamiento, puede dañarse, porque ningún hardware es absolutamente fiable. Ésto también se aplica a la información no residente en ordenadores. El habla enviada por el aire queda distorsionada por el ruido, el viento, las altas temperaturas, etc. El habla, de hecho, es un buen punto de partida para la comprensión de los principios de detección de errores y los códigos correctores de errores. Imagine una fiesta bulliciosa donde todos hablan simultáneamente, con la música a todo volumen. Sabemos que incluso en tal situación, es posible mantener una conversación, excepto que se requiere más atención de la habitual.

E.1. Primeros principios

¿Que hace nuestro lenguaje tan robusto, tan inmune a los errores? Hay dos propiedades: la *redundancia* y el *contexto*.

- Nuestro lenguaje es redundante, porque sólo una fracción muy pequeña de todas las palabras posibles son válidas. Un gran número de palabras puede ser construido con las 26 letras del inglés. Sólo el número de palabras de siete letras, e.g., es de $26^7 \approx 8{,}031$ mil millones. A pesar de todo, únicamente alrededor de 50 000 palabras son de uso común, e incluso el Diccionario Oxford lista "sólo" alrededor de 500 000 palabras. Cuando escuchamos una palabra ininteligible, nuestro cerebro busca entre muchas palabras similares, para encontrar la palabra válida "más cercana". Las computadoras son muy buenas en tales búsquedas, por lo que la redundancia es la base para la detección de errores y los códigos correctores de errores.

- Nuestro cerebro funciona mediante asociaciones. Por eso nosotros, los humanos destacamos en el uso del contexto de un mensaje para reparar los errores en el mensaje. Al recibir una oración con una palabra ininteligible o una palabra que no encaja, como en "pass the thustard please", primero utilizamos la memoria para encontrar las palabras asociadas con "thustard". Luego usamos nuestra experiencia acumulada en la vida real para seleccionar, entre los muchos posibles candidatos, la palabra que mejor encaja en el presente contexto. Si estamos en la carretera, pasa

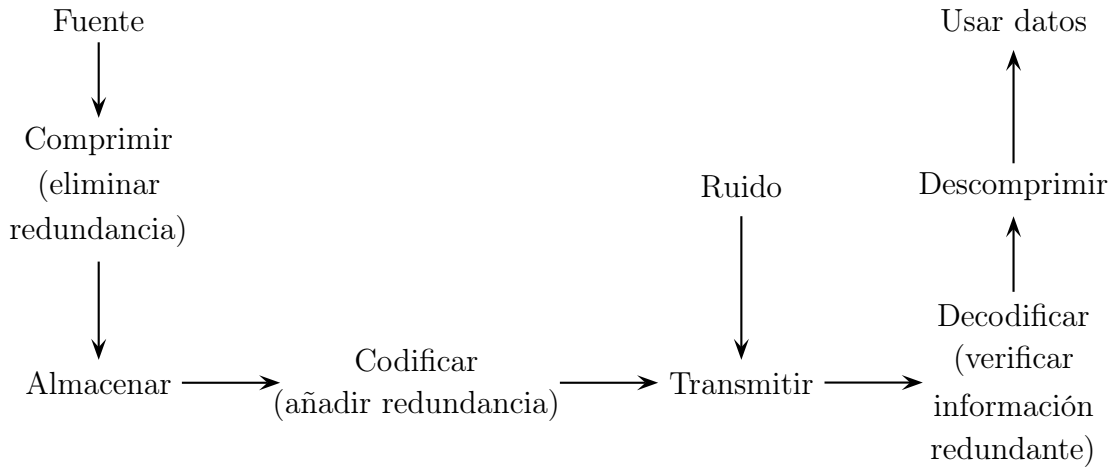


Figura E.1: Manipulación de información.

por delante de nosotros la palabra bastard;[1] si estamos en la cena, nos viene a la mente mustard (o custard).[2] Otro ejemplo es la oración (dañada) escrita en inglés:

if u cn rd ths u cn bcm a c prgmr!

que podemos entender fácilmente.[3] Las computadoras no tienen experiencia de la vida real y son notoriamente malas en tales tareas, por lo que el contexto no se utiliza en los códigos de ordenador. En casos extremos, donde gran parte de la frase es defectuosa, incluso es posible que no seamos capaces de corregirla, y podemos pedir una retransmisión "say it again, Sam".[4]

La idea de utilizar redundancia para añadir fiabilidad a la información se debe a Claude Shannon, el fundador de la teoría de la información. No es una idea trivial, ya que estamos condicionados en contra de ella. La mayoría de las veces, tratamos de *eliminar* la redundancia en los datos del ordenador, con el fin de ahorrar espacio. De hecho, todos los métodos de compresión de datos discutidos aquí hacen precisamente eso.

La Figura E.1 muestra las etapas por las que puede pasar un bloque de datos de la computadora cuando es creado, almacenado, transmitido, recibido, y utilizado en el extremo receptor.

Discutimos dos enfoques para la fiabilidad de códigos. El primer enfoque consiste en duplicar el código, lo que conduce a la idea de *códigos de escrutinio* (*voting codes*); la segunda utiliza *bits de comprobación* (*check bits*) y está basado en el concepto de *distancia de Hamming*.

E.2. Códigos de escrutinio

La primera idea que suele venir a la mente, cuando uno piensa acerca de la redundancia, es la duplicación del mensaje y el envío de dos copias. Así, si debe enviarse el código 110, puede transmitirse 110 | 110. Pensando un poco, ésto puede, quizá, ser una buena idea para la detección de errores, pero no para la corrección de errores. Si el receptor recibe dos copias diferentes, no puede decir cuál de

[1] Bastard, en español, significa bastardo, desgraciado.
[2] Mustard y custard, en español, significan "mostaza" y "crema o natillas", respectivamente.
[3] "If you can read this you can become a C programmer!", que significa en español: "si usted puede leer ésto, ¡puede llegar a ser un programador de C!".
[4] "Say it again, Sam" significa en español: "repítelo de nuevo, Sam".

las dos es buena. ¿Qué sucede al triplicar el mensaje? podemos enviar 110 | 110 | 110 y comunicar al receptor que decida cuál de las tres copias es buena mediante la comparación de las mismas. Si las tres son idénticas, el receptor asume que son correctas. Por otra parte, si sólo dos son idénticas y la tercera diferente, el receptor supone que las dos copias idénticas son las correctas. Este es el principio de los *códigos de escrutinio*. Si las tres copias son diferentes, el receptor sabe que se ha producido un error, i.e., puede detectar un error a pesar de que no sea capaz de corregirlo.

Para mantener el análisis simple, vamos a limitarnos a sólo errores de 1 bit. Cuando las tres copias son recibidas, los casos posibles son los siguientes:

1. Las tres son idénticas. Hay dos subcasos:
   a) Los tres son buenos. Éste es el caso normal.
   b) Los tres han sido dañados en la misma medida. Éste es un caso raro.
2. Dos copias son idénticas, y la tercera es diferente. Una vez más, hay dos subcasos:
   a) Las dos idénticas son buenas. Éste es el caso normal.
   b) Ellas son erróneas. Ésto es, esperamos, un caso raro.
3. Las tres copias son diferentes.

Usando el principio de escrutinio, asumimos que las tres copias idénticas en el caso 1 son buenas. En el caso 1a nuestra hipótesis es correcta, y en el caso 1b, no lo es. De manera similar, en el caso 2a el receptor toma la decisión correcta, y en el caso 2b, el equivocado. En el caso 3, el receptor no puede corregir el error, pero por lo menos es capaz de detectar uno, por tanto, no toma una decisión equivocada.

Los únicos casos en que el receptor toma la decisión equivocada (donde el principio de escrutinio no funciona), por tanto, son el 1b y el 2b. Pensando un poco vemos que la probabilidad del caso 1b es mucho menor que la del 2b. Trataremos, por lo tanto, de estimar la probabilidad del caso 2b y, si es lo suficientemente pequeña, no habría necesidad de preocuparse del caso 1b.

Es difícil calcular la probabilidad de que dos copias sean distorsionadas *de la misma manera*. Nosotros, por lo tanto, calcularemos la probabilidad de que un bit sea cambiado en dos de las tres copias (ya sea en la misma o en diferentes posiciones de bit). Si esta probabilidad es lo suficientemente pequeña, no hay necesidad de preocuparse por el caso 2b, puesto que su probabilidad es aún más baja.

Denotamos por p la probabilidad de que un bit se corrompa en nuestras transmisiones. La probabilidad de que un bit sea erróneo en dos de las tres copias es $\binom{3}{2} p^2 = 3p^2$. No es simplemente p^2, ya que es posible seleccionar dos objetos de tres de $\binom{3}{2}$ = tres formas.

[La notación $\binom{m}{n}$ se pronuncia "m sobre n" y se define como:

$$\frac{m!}{n!\,(m-n)!}.$$

Éste es el número de maneras en que n objetos pueden ser seleccionados de un conjunto de m objetos.]

Supongamos que $p = 10^{-6}$ (en promedio, un error en un millón de bits transmitidos) y queremos enviar 10^8 bits. Sin duplicación podemos esperar $10^8 \times 10^{-6} = 100$ errores, una tasa inaceptablemente alta. Enviando tres copias, tenemos que transmitir un total de 3×10^8 bits, y la probabilidad de que dos de las tres copias sea errónea es 3×10^{-12}. En consecuencia, el número esperado de errores es $\left(3 \times 10^8\right) \times \left(3 \times 10^{-12}\right) = 9 \times 10^{-4} = 0{,}0009$ errores, un número confortablemente pequeño.

Si se necesita una mayor fiabilidad, se pueden enviar más copias. Un código donde cada símbolo es duplicado y se envía nueve veces es extremadamente fiable. Los códigos de escrutinio son así de sencillos, fiables, y tienen sólo un inconveniente, que son demasiado largos. En la práctica, el envío de

nueve, o incluso tres, copias de cada mensaje puede ser prohibitivamente caro. Es por ésto que se han realizado muchas investigaciones en el campo de la codificación en los últimos 40 años y, hoy en día, se conocen muchos códigos sofisticados que son más fiables y más cortos que los códigos de escrutinio.

E.3. Bits de verificación

En la práctica, la detección y corrección de errores se efectúa generalmente por medio de *bits de verificación*, que son añadidos a los *bits de información* originales de cada palabra del mensaje. En general, se añaden k bits de verificación a los m bits de información originales, para producir una palabra de código de $n = m + k$ bits. Tal código se conoce como un código (n, m). La palabra de código es transmitida entonces al receptor. Sólo ciertas combinaciones de bits de información y bits de comprobación son válidos, en analogía a un lenguaje natural. El receptor conoce qué palabras de código son válidas. Si se recibe una palabra de código que no es válida, el receptor considera que es un error. En la Sección E.7 se muestra que, mediante la adición de más bits de verificación, el receptor también puede corregir ciertos errores, no solo detectarlos. El principio de corrección de errores consiste en que, al recibir una palabra de código errónea, el receptor selecciona la palabra de código válida "más cercana" a ella.

Ejemplo: Un conjunto de 128 símbolos necesita ser codificado. Ésto implica $m = 7$. Si seleccionamos $k = 4$, terminamos con 128 palabras de código válidas, cada una de 11 bits de longitud. Ésto es, por lo tanto, un código (11, 7). Las palabras de código válidas son seleccionadas de un total de $2^{11} = 2\,048$ palabras de código posibles, por lo que quedan $2\,048 - 128 = 1\,920$ palabras de código no válidas. La gran diferencia entre el número de palabras de código válidas y no válidas significa que si una palabra de código se corrompe, lo más probable es que se cambie a uno no válido.

Por supuesto, puede ocurrir que una palabra de código válida sea cambiada, durante la transmisión, a otra palabra de código válida. En consecuencia, nuestros códigos no son completamente fiables, pero pueden hacerse cada vez más fiables mediante la adición de más bits de verificación y seleccionando las palabras de código válidas cuidadosamente. Uno de los teoremas básicos de la teoría de la información, dice que los códigos pueden hacerse tan fiables como se desee mediante la adición de bits de verificación, siempre que n (el tamaño de una palabra de código) no exceda la capacidad del canal.

Es importante entender el significado de la palabra "error" en la transmisión de datos. Cuando una palabra de código es recibida, el receptor siempre recibe n bits, pero algunos de ellos puede ser erróneos. Un bit erróneo no desaparece, ni tampoco cambia en algo que no sea un bit. Un bit erróneo simplemente cambia su valor, de 0 a 1, ó de 1 a 0. Ésto hace que sea relativamente fácil corregir el bit. El código debe informar al receptor qué bits son erróneos, y el receptor puede corregir fácilmente los bits, invirtiéndolos.

En la práctica, los bits pueden enviarse por una línea de comunicación como voltajes. Un 0 binario puede, por ejemplo, ser representado por cualquier tensión en el rango $3-25$ voltios. Un 1 binario puede similarmente ser representado mediante el rango de tensiones: de -25v a -3v. Tales tensiones tienden a caer sobre largas líneas de comunicación, y tienen que ser amplificadas periódicamente. En la red telefónica existe un amplificador (repetidor) cada 20 millas o menos. Se mira cada bit recibido, se decide si es un 0 ó un 1 midiendo la tensión, y se envía al siguiente repetidor como un impulso limpio y fresco. Si el voltaje se ha deteriorado lo suficiente en el pasaje, el repetidor puede tomar una decisión equivocada cuando lo envíe, lo que introduce un error en la transmisión. En la actualidad, las líneas de transmisión típicas tienen tasas de error de aproximadamente uno cada mil millones, pero en condiciones extremas —tales como en una tormenta eléctrica, o cuando el energía eléctrica fluctúa de repente— la tasa de error puede aumentar repentinamente, creando una ráfaga de errores.

0	1	0	0	**1**
1	0	1	0	**0**
0	1	1	1	**1**
0	0	0	0	**0**
1	1	0	1	**1**
0	**1**	**0**	**0**	**1**

(a)

0	1	0	0	**1**
1	0	1	**0**	
0	1	**0**		
0	**0**			
1				

(b)

Tabla E.2: Bits de paridad.

E.4. Bits de paridad

Un bit de paridad puede ser añadido a un grupo de m bits de información para completar un número impar del número total de bits a 1. Así, la paridad (impar) del grupo 10110 es 0, ya que el grupo original más el bit de paridad tendrían un número impar (3) de unos. También puede usarse la paridad par, y la única diferencia entre la paridad impar y par es que, en el caso de la paridad par, es válido un grupo con todo a ceros, mientras que, con la paridad impar, cualquier grupo de bits con un bit de paridad adicionado no puede ser todo ceros.

Los bits de paridad pueden utilizarse para diseñar códigos correctores de errores sencillos, pero no muy eficientes. Para corregir errores de 1 bit, el mensaje puede ser organizado como un *rectángulo* de dimensiones $(r-1) \times (s-1)$. Se añade un bit de paridad a cada fila de $s-1$ bits, y para cada columna de $r-1$ bits. El tamaño total del mensaje (Tabla E.2a) llega a ser de $s \times r$.

Si sólo hay un bit erróneo, una comprobación de todos los $s-1+r-1$ bits de paridad lo descubre, ya que sólo una de las $s-1$ paridades y sólo una de las $r-1$ será errónea.

La sobrecarga de un código se define como el número de bits de paridad dividido por el número de bits de información. La sobrecarga del código rectangular es, por lo tanto,

$$\frac{(s-1+r-1)}{(s-1)\,(r-1)} \approx \frac{s+r}{s \times r-(s+r)}.$$

Un código similar, ligeramente más eficiente, tiene una configuración triangular, donde los bits de información están dispuestos en un triángulo, con los bits de paridad en la diagonal (Tabla E.2b). Cada bit de paridad es la paridad de todos los bits de su fila y su columna. Si la fila superior contiene r bits de información, el triángulo completo tiene $r\,(r+1)\,/2$ bits de información y r bits de paridad. La sobrecarga es, pues,

$$\frac{r}{r\,(r+1)\,/2} = \frac{2}{r+1}.$$

También es posible disponer los bits de información en un número de planos bidimensionales, para obtener un cubo tridimensional, tres de cuyas seis superficies exteriores están compuestas por los bits de paridad.

No es obvio cómo generalizar estos métodos a más de 1 bit de corrección de errores.

E.5. Distancia de Hamming y detección de errores

Richard Hamming desarrolló el concepto de distancia, en la década de 1950, como una manera general para utilizar bits de verificación para la detección y corrección de errores.

Símbolo	código_1	código_2	código_3	código_4	código_5	código_6	código_7
A	0000	0000	001	001001	01011	110100	110
B	1111	1111	010	010010	10010	010011	0
C	0110	0110	100	100100	01100	001101	10
D	0111	1001	111	111111	10101	101010	111
k :	2	2	1	4	3	4	

Tabla E.3: Ejemplos de códigos con $m = 2$.

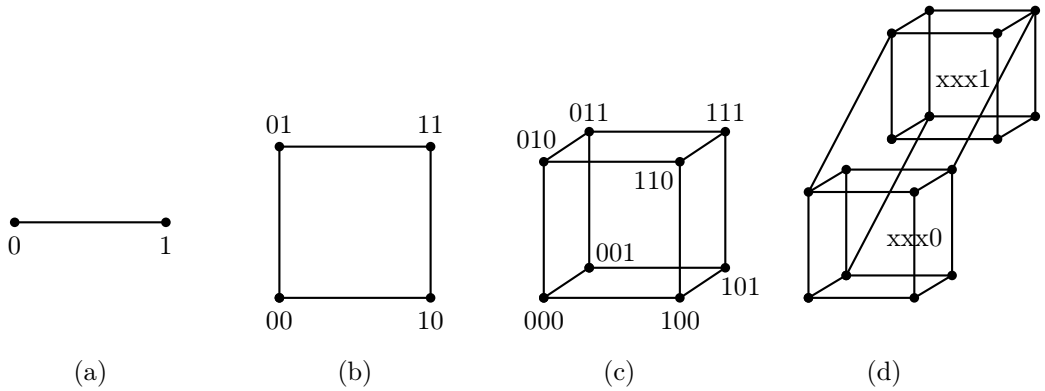


Figura E.4: Cubos de varias dimensiones y numeración de las esquinas.

Para ilustrar este concepto, comenzamos con un sencillo ejemplo que implica sólo a los cuatro símbolos A, B, C, y D. Sólo son necesarios 2 bits de información, pero los códigos de la Tabla E.3 añaden algunos bits de verificación, hasta un total de 3−6 bits por símbolo. El código_1 es simple. Sus cuatro palabras de código son seleccionadas de los 16 posibles números de 4 bits, y no son las mejores posibles. Cuando el receptor recibe una de ellas, por ejemplo, 0111, asume que no hay error y el símbolo recibido es D. Cuando se recibe una palabra de código no válido, el receptor señala un error. Puesto que el código_1 no es el mejor posible, no se detecta cada error. Incluso si nos limitamos a errores de un bit, este código es no muy bueno. Hay 16 posibles errores de un bit en nuestras palabras de código de 4 bits y, de ellos, los 4 siguientes no pueden ser detectados: Un 0110 cambiado durante la transmisión a 0111, un 0111 cambiado a 0110, un 1111 corrompido a 0111, y un 0111 distorsionado a 1111. La tasa de detección de errores es, por lo tanto, 12 de 16, ó el 75 %. En comparación, el código_2 realiza un trabajo mucho mejor, ya que puede detectar todos los errores de un único bit.

◇ **Ejercicio E.1 (sol. en pág. 1114):** Demuéstrese la declaración anterior.

Por lo tanto, decimos que las cuatro palabras de código de código_2 están suficientemente *distantes* entre sí. El concepto de distancia entre palabras de código, por fortuna, es fácil de definir.

Definiciones: (1) Dos palabras de código están separadas a una distancia d de Hamming si difieren exactamente en d de sus n bits y (2) un código tiene una distancia d de Hamming si cada par de palabras de código en el código están separadas, al menos, una distancia d de Hamming.

Estas definiciones tienen una interpretación geométrica simple. Imagine un hipercubo en espacio n-dimensional. Cada una de sus 2^n esquinas puede ser identificada mediante un número de n bits (Figura E.4), de tal manera que cada uno de los n bits corresponda a una de las n dimensiones. En tal cubo, los puntos que están conectados directamente tienen una distancia de Hamming de 1, los puntos con un vecino común tienen una distancia de Hamming de 2, etc. Si se necesita un código con una distancia de Hamming de 2, sólo pueden ser seleccionadas palabras de código válidas los puntos

que no están directamente conectados.

La razón de que el código$_2$ pueda detectar todos los errores de un bit es que tiene una distancia de Hamming de 2. La distancia entre palabras de código válidas es 2, por lo que un bit erróneo siempre cambia una palabra de código válido en una no válida. Cuando dos bits son erróneos, una palabra de código válida se convierte en otra palabra de código, a una distancia de 2. Si queremos que sea otra la palabra de código que indique esa corrupción en los bits, el código debe tener al menos una distancia de 3.

En general, un código con una distancia de Hamming de $d + 1$ puede detectar todos los errores en d bits. El código$_3$ tiene una distancia de Hamming de 2 y, por lo tanto, puede detectar todos los errores en 1 bit a pesar de ser corto ($n = 3$).

◇ **Ejercicio E.2 (sol. en pág. 1114):** Calcúlese la distancia de Hamming del código$_4$.

Ahora es obvio que podemos aumentar la fiabilidad de las transmisiones a cualquier nivel deseado, pero esta característica no es gratis. Como siempre, hay un equilibrio, o un precio a pagar, en forma de sobrecarga. Nuestros códigos son mucho más largos que m bits por símbolo, debido a los bits de verificación añadidos. Una medida del precio es $n/m = (m+k)/m = 1 + k/m$, donde la cantidad k/m se llama *sobrecarga* del código. En el caso del código$_1$ la sobrecarga es 2 y, en el caso del código$_3$, es $3/2$.

Ejemplo: Un código con un único bit de verificación, que es un bit de paridad (par o impar). Cualquier error en un bit puede ser detectado fácilmente, ya que crea una palabra de código no válida. Tal código, por lo tanto, tiene una distancia de Hamming de 2. El código$_3$ visto anteriormente utiliza un único bit de paridad impar.

Ejemplo: Un código de detección de errores de 2 bits para los mismos cuatro símbolos (véase el código$_4$). Debe tener una distancia de Hamming de 4, y una forma de generarlo es duplicar el código$_3$.

E.6. Códigos de Hamming

El principio de los códigos correctores de errores es la de alejar los códigos aún más añadiendo más redundancia (más bits de verificación). Cuando una palabra de código válida es recibida, el receptor corrige el error seleccionando la palabra de código válida que sea más cercana a la recibida. El código$_5$ tiene una distancia de Hamming de 3. Cuando una de sus cuatro palabras de código ha cambiado en un solo bit, está a 1 bit de distancia de la original, pero todavía está a 2 los bits de distancia de cualquiera de las otras palabras de código. Por consiguiente, si sólo hay un error, el receptor siempre puede corregirlo. El receptor realiza ésto comparando cada palabra de código recibida con la lista de palabras de código válidas. Si no encuentra una coincidencia, el receptor asume que se ha producido un error en 1 bit, y corrige el error seleccionando la palabra de código que esté más cercana a la recibida.

En general, cuando se corrompen d bits en una palabra de código C_1, éste se convierte en una palabra de código C_2 inválida a una distancia d de C_1. Si la distancia entre C_2 y las otras palabras de código válidas es de al menos $d + 1$, entonces C_2 está más cercana a C_1 que a cualquier otra palabra de código válida. Por ésto es que es necesario un código con una distancia de Hamming de $d + (d + 1) = 2d + 1$ para corregir todos los errores en d bits.

¿Cómo son las palabras de código seleccionadas? El problema es seleccionar un buen conjunto de 2^m palabras de código fuera de las 2^n posibles. El primer enfoque utiliza la fuerza bruta. Es fácil escribir un programa informático que examine todos los posibles conjuntos de 2^m palabras de código, y selecciona uno que tenga la distancia correcta. Los problemas con este enfoque son: (1) El tiempo y el almacenamiento requerido en el extremo receptor para verificar y corregir los códigos recibidos y (2) la cantidad de tiempo que se necesita para examinar todos los posibles conjuntos de palabras de código.

1. El receptor debe tener una lista de todas las 2^n palabras de código posibles. Para cada palabra de código debe tener un flag o bandera que indique si es válida y, si no, qué palabra de código válida

es la más cercana a ella. Cada palabra de código recibida tiene que ser buscada y localizada en esta lista con el fin de verificarla.

2. En el caso de cuatro símbolos, sólo necesitan ser seleccionadas cuatro palabras de código. Para el código$_1$ y el código$_2$, tuvieron que ser seleccionadas de entre los 16 números posibles, lo que puede hacerse de $\binom{16}{4} = 7\,280$ maneras. Es posible escribir un sencillo programa que examine sistemáticamente conjuntos de cuatro palabras de código hasta que encuentre un conjunto con la distancia requerida. En el caso del código$_4$, las cuatro palabras de código tenían que seleccionarse a partir de un conjunto de 64 números, lo que puede hacerse de $\binom{64}{4} = 635\,376$ maneras. Todavía es posible escribir un programa que sistemáticamente explore todas las posibles palabras de código para este caso. En la práctica, sin embargo, donde están involucrados conjuntos con cientos de símbolos, el número de posibilidades para la selección de los conjuntos de palabras de código es demasiado grande, incluso para que los ordenadores más rápidos los manipulen cómodamente.

Claramente, se necesita un algoritmo inteligente que seleccione las mejores palabras de código, y las verifique sobre la marcha, a medida que son recibidas. El transmisor debe utilizar el algoritmo para generar las palabras de código cuando tienen que ser enviadas, y el receptor debe utilizarlo para verificarlas cuando son recibidas. El método aquí descrito se debe a Richard Hamming. En los códigos de Hamming [Hamming 86] los n bits de un palabra de código se indexan desde 1 hasta n. Los bits de verificación son aquellos con índices que son potencias de 2. En consecuencia, los bits b_1, b_2, b_4, b_8,... son bits de verificación y b_3, b_5, b_6, b_7, b_9,... son bits de información. El índice de cada bit de información puede escribirse como la suma de los índices de ciertos bits de verificación. Así b_7 puede escribirse como b_{1+2+4} y es, por lo tanto, utilizado para la determinar los valores de los bits de verificación b_1, b_2, b_4. *Los bits de verificación son simplemente bits de paridad.* El valor de b_2, e.g., es la paridad (par o impar) de b_3, b_6, b_7, b_{10},... etc, ya que $3 = 2 + 1$, $6 = 2 + 4$, $7 = 2 + 1 + 4$, $10 = 2 + 8$,...

Ejemplo: Un código de corrección de errores de 1 bit para el conjunto de símbolos A, B, C, y D. Debe tener una distancia de Hamming de $2d + 1 = 3$. Son necesarios dos bits de información para codificar los cuatro símbolos, por lo que deben ser: b_3 y b_5. Los bits de paridad son, por lo tanto, b_1, b_2, y b_4. Puesto que $3 = 1 + 2$ y $5 = 1 + 4$, los 3 bits de paridad se definen así: b_1 es la paridad de los bits b_3 y b_5, b_2 es la paridad de b_3, y b_4 es la paridad de b_5. Así es como se construyó el código$_5$ de la Tabla E.3.

Ejemplo: Un código de corrección de errores de 1 bit para un conjunto de 256 símbolos. Debe tener una distancia de Hamming de $2d + 1 = 3$. Se requieren ocho bits de información para codificar los 256 símbolos, así que ellos deben ser b_3, b_5, b_6, b_7, b_9, b_{10}, b_{11}, y b_{12}. Los bits de paridad son, por lo tanto, b_1, b_2, b_4, y b_8. El tamaño total del código es de 12 bits. Las siguientes relaciones definen los 4 bits paridad:

$3 = 1 + 2$, $5 = 1 + 4$, $6 = 2 + 4$, $7 = 1 + 2 + 4$, $9 = 1 + 8$, $10 = 2 + 8$, $11 = 1 + 2 + 8$, y $12 = 4 + 8$.

Dan a entender que b_1 es la paridad de b_3, b_5, b_7, b_9, y b_{11}.

◇ **Ejercicio E.3 (sol. en pág. 1114):** ¿Cuáles son las definiciones de los otros bits de paridad?.

◇ **Ejercicio E.4 (sol. en pág. 1115):** Constrúyase un código de Hamming de corrección de errores de 1 bit para códigos de 16 bits ($m = 16$).

Una cuestión común en este punto es cómo se determina el número de bits de paridad. La respuesta es que se determina implícitamente. Sabemos que son necesarios m bits de datos, y también sabemos que los bits b_1, b_2, b_4, b_8,... deben ser los bits de paridad. Por lo tanto, asignamos los m bits iniciales del conjunto b_3, b_5, b_6, b_7, b_9, b_{10}, b_{11},... a los datos, y ésto determina implícitamente el número de bits de paridad necesarios.

¿Cuál es el tamaño de un código general de Hamming? El caso del código corrector de errores de 1 bit es fácil de analizar. Dado un conjunto de 2^m símbolos, son necesarias 2^m palabras de código válidas. Estamos buscando el menor k necesario para construir palabras de código de tamaño $m+k$ y distancia de Hamming 3. Las 2^m palabras de código válidas deben ser seleccionadas a partir de un total de 2^n números (donde $n = m+k$), de tal manera que cada palabra de código está formada por m bits de información y k bits de verificación.

Puesto que queremos que cualquier error en un bit de una palabra de código pueda ser corregido, tal error no debería llevarnos demasiado lejos de la palabra de código original. Un error en un único bit nos lleva a una palabra de código a la distancia 1 de la original. Como resultado, todas las palabras de código a distancia 1 de la palabra de código original debe ser no válida. Cada una de las 2^m palabras de código originales es de n bits de longitud y, por lo tanto, tiene n palabras de código a una distancia 1 de ella; deben ser declaradas inválidas. Ésto significa que el número total de palabras de código necesarias (las válidas más las inválidas) es $2^m + n2^m = (1+n)\,2^m$. Este número tiene que ser seleccionado de los 2^n números disponibles, por lo que terminamos con la relación $(1+n)\,2^m \leq 2^n$. Puesto que $2^n = 2^{m+k}$, obtenemos $1+n \leq 2^k$ ó $k \geq \log_2(1+n)$. La siguiente tabla ilustra el significado de esta relación para ciertos valores de m.

n :	4	7	12	21	38	71
k :	2	3	4	5	6	7
$m = n-k$:	2	4	8	16	32	64
k/m :	1	0,75	0,5	0,31	0,19	0,11

Existe una interpretación geométrica que proporciona otra manera de obtener el mismo resultado. Imaginemos 2^m esferas de radio uno empacadas en nuestro cubo n-dimensional. Cada esfera está centrada en una de las esquinas y abarca todas sus esquinas vecinas inmediatas. El *volumen* de una esfera se define como el número de esquinas que incluye, que es $1+n$. Las esferas están muy juntas, pero no se solapan, por lo que su volumen total es $(1+n)\,2^m$, y esto no debe exceder el volumen total del cubo, que es 2^n.

El caso de un código de corrección de errores de 2 bits se analiza de manera similar. Cada palabra de código válida debe definir un conjunto que incluye: a sí misma, las n palabras de código a distancia 1 de ella, y el conjunto de $\binom{n}{2}$ palabras de código a distancia 2 de la misma, un total de $\binom{n}{0} + \binom{n}{1} + \binom{n}{2} = 1 + n + {}^{n(n-1)}/_{2}$. Estos conjuntos no deben estar solapados, lo que implica la relación:

$$\left(1+n+\frac{n\,(n-1)}{2}\right)2^m \leq 2^n \Rightarrow 1+n+\frac{n\,(n-1)}{2} \leq 2^k \Rightarrow k \geq \log_2\left(1+n+\frac{n\,(n-1)}{2}\right).$$

En la interpretación geométrica de nuevo imaginamos 2^m esferas de radio 2 cada una. Cada esfera se centra alrededor de una esquina y contiene la esquina, sus n vecinos inmediatos, y sus $\binom{n}{2}$ vecinos secundarios (las esquinas difieren del centro en 2 bits).

E.7. El código SEC-DED

Sin embargo, a pesar de que podemos estimar la longitud de un código de Hamming de corrección de errores, ¡no sabemos cómo construirlo! Lo mejor que se puede hacer hoy en día con los códigos de Hamming es combinar la corrección de errores de un bit con la detección de errores doble. Un ejemplo de tal código SEC-DED es el código$_6$. Éste fue creado mediante simple adición de un bit de paridad al código$_5$.

◇ **Ejercicio E.5 (sol. en pág. 1115):** La Tabla E.3 contiene un código más, el código$_7$. ¿Qué es?

El receptor verifica el código SEC-DED en dos pasos. En el paso 1, el único bit de paridad es verificado. Si es erróneo, el receptor asume que se ha producido un error en 1 bit, y utiliza los otros

bits de paridad, en el paso 2, para corregir el error. Puede suceder, por supuesto, que tres o incluso cinco bits sean erróneos, pero el sencillo código SEC-DED no puede detectar tales errores.

Si la única paridad es correcta, entonces, o bien no hay errores, o bien los dos bits son erróneos. El receptor cambia al paso 2, donde utiliza los otros bits de paridad para distinguir entre estos dos casos. De nuevo, podrían ser cuatro o seis bits erróneos, pero este código no puede manejar estos casos.

El código SEC-DED tiene una distancia de Hamming de 4. En general, un código para c-bit de corrección de errores y d bits de detección de errores deben tener una distancia de $c + d + 1$.

E.8. Polinomios generadores

Hay muchos enfoques para el problema de desarrollo de códigos de corrección de errores de más de 1 bit. Ellos son, sin embargo, más complicados que el método de Hamming, y requieren una formación en teoría de grupos y campos de Galois. En esta sección esbozamos brevemente uno de estos enfoques, utilizando el concepto de *polinomio generador*.

Usamos el caso $m = 4$ como ilustración. Son necesarias dieciséis palabras de código, que pueden ser utilizadas para codificar algún conjunto de 16 símbolos. Sabemos de la discusión anterior que, para la corrección de un error en 1 bit, se necesitan 3 bits de paridad, llegando a un tamaño total del código de $n = 7$. Aquí hay un ejemplo de tal código:

0000000	0001011	0010110	0011101	0100111	0101100	0110001	0111010
1000101	1001110	1010011	1011000	1100010	1101001	1110100	1111111

Observe que tiene las siguientes propiedades:

- La suma (en módulo 2) de cualquiera de las dos palabras de código es igual a otra palabra de código. Ésto implica que la suma de cualquier número de palabras de código es una palabra de código. En consecuencia, las 16 palabras de código de arriba forman un *grupo* bajo esta operación.

  (La suma y la resta en módulo 2 se realiza mediante $0 + 0 = 1 + 1 = 0$, $0 + 1 = 1 + 0 = 1$, $1 - 0 = 0 - 1 = 1$. La definición de grupo debe ser revisada en cualquier texto sobre álgebra.)

- Cualquier desplazamiento circular de una palabra de código es otra palabra de código. Este código es por tanto *cíclico*.

- Tiene una distancia de Hamming de 3, como se requiere para la corrección del error en 1 bit.

¡Interesantes propiedades! Las 16 palabras de código son seleccionadas a partir de los 128 posibles mediante un polinomio generador. La idea es ver cada palabra de código como un polinomio, donde los bits son los coeficientes. Éstas son algunas palabras de código de 7 bits asociadas con polinomios de grado 6.

1	0	0	1	1	1	1
x^6			$+x^3$	$+x^2$	$+x$	$+1$
0	1	1	0	0	1	0
	x^5	$+x^4$			$+x$	
0	1	0	0	1	1	1
	x^5			$+x^2$	$+x$	$+1$

Las 16 palabras de código de más arriba han sido seleccionadas buscando los polinomios de grado 6 que son divisibles (módulo 2) por el polinomio generador $x^3 + x + 1$. Por ejemplo, la tercera palabra de código '0100111' de la tabla corresponde al polinomio $x^5 + x^2 + x + 1$, que es divisible por $x^3 + x + 1$ porque $x^5 + x^2 + x + 1 = \left(x^3 + x + 1\right)\left(x^2 + 1\right)$.

Para comprender cómo pueden ser calculados tales polinomios, consideremos operaciones similares con números. Supongamos que queremos conocer el mayor múltiplo de 7, que es ≤ 30. Dividimos 30 por 7, obteniendo un residuo de 2, y luego restamos el 2 de 30, obteniendo 28. Similarmente con polinomios. Comencemos con los 4 bits de información 0010, y calculemos los 3 bits de paridad restantes. Escribimos 0010*ppp*, lo que nos da el polinomio x^4. Dividimos x^4 por el polinomio generador, obteniendo un resto de $x^2 + x$. Sustrayendo este resto de x^4 nos da algo que será divisible por el polinomio generador. El resultado de la resta es $x^4 + x^2 + x$, por lo que la palabra de código completa es 0010110.

Cualquier polinomio generador puede proporcionarnos las dos primeras propiedades. Para obtener el tercera propiedad (la distancia de Hamming necesaria), debe utilizarse el polinomio generador correcto, y puede seleccionarse mediante el examen de sus raíces. Este tema está fuera del alcance de este libro, pero es tratado en todos los textos sobre códigos correctores de errores. Un ejemplo común de polinomio generador es el CRC (Sección 3.23).

E.9. Bibliografía relacionada

Hamming, Richard (1950) "Error Detecting and Error Correcting Codes," *Bell Systems Technical Journal* **29**:147−160, April.

Hamming, Richard (1986) *Coding and Information Theory*, 2nd Ed., Englewood Cliffs, NJ, Prentice-Hall.

Lin, Shu (1970) *An Introduction to Error Correcting Codes*, Englewood Cliffs, NJ, Prentice- Hall.

Los errores que usan datos inadecuados son mucho menores
que aquellos que no usan ningún dato en absoluto.
Charles Babbage (1792-1871)

Déme un error fructífero en cualquier instante, repleto de semillas, rebosante
de sus propias correcciones. Puede conservar su estéril verdad para usted.
Wilfredo Pareto (1848-1923)

Apéndice F

Autómatas de estados finitos

Un *autómata finito* (AF, también llamado un *autómata de estados finitos* o una *máquina de estados finitos*) es una herramienta matemática utilizada para describir los procesos que involucran entradas y salidas. Un AF puede estar en uno de varios estados y puede cambiar entre los estados en función de los símbolos que reciba. Una vez que se instala en un estado, lee el siguiente símbolo de entrada, realiza alguna tarea computacional asociada con la nueva entrada, emite un símbolo, y cambia a un nuevo estado dependiendo de la entrada. Observe que el nuevo estado puede ser idéntico al estado actual. La Figura F.2a muestra un ejemplo de una AF con cuatro estados etiquetados de 1 a 4. La entrada consiste en una cadena de símbolos del alfabeto `abc` y la salida es una cadena de símbolos de `mnpq`. El AF comienza en el estado 1. Si recibe una `a`, permanece en el estado 1 y emite una `m`. Si recibe una `b`, cambia al estado 2 y emite una `p`. Observe que este AF en particular puede quedar "atrapado" si cambia a un estado que no le diga qué hacer con una entrada determinada. Ésto ocurrirá, por ejemplo, con la cadena de entrada "`abbc`...", ya que la `c` será introducida mientras el AF está en el estado 4, donde espera una `a`. Decimos que "`abbc`..." es no aceptada por el AF. Las aristas de la gráfica son etiquetadas con pares (entrada, salida).

La Figura F.2b muestra un ejemplo de un AF de dos estados que puede ser utilizado para sumar números binarios. El AF comienza en el estado 1 (sin acarreo) y recibe un par de bits. Si el par es 11, el AF emite un 0 y cambia al estado 2 (con acarreo), donde el siguiente par de bits es introducido y es sumado a un bit de acarreo 1. La Tabla F.1 muestra los pasos individuales en la adición de los dos números de 6 bits, 14 y 23 (éstos son en realidad números de 5 bits, pero el sencillo diseño de este AF requiere que un bit 0 extra sea añadido al extremo izquierdo de cada número) . Como la adición de los números se efectúa de derecha a izquierda, las seis columnas de la tabla también deben leerse en esta dirección.

Paso:	6	5	4	3	2	1
14:	0	0	1	1	1	0
23:	0	1	0	1	1	1
Entrada:	00	01	10	11	11	01
Estado inicial:	2	2	2	2	1	1
Salida (37)	1	0	0	1	0	1
Estado final:	1	2	2	2	2	1

Tabla F.1: Suma de 14 y 23.

La Figura F.2c muestra un ejemplo de un AF de tres estados donde cada arista es etiquetada mediante un símbolo de entrada y una probabilidad, pero no la salida. El diagrama ilustra un AF

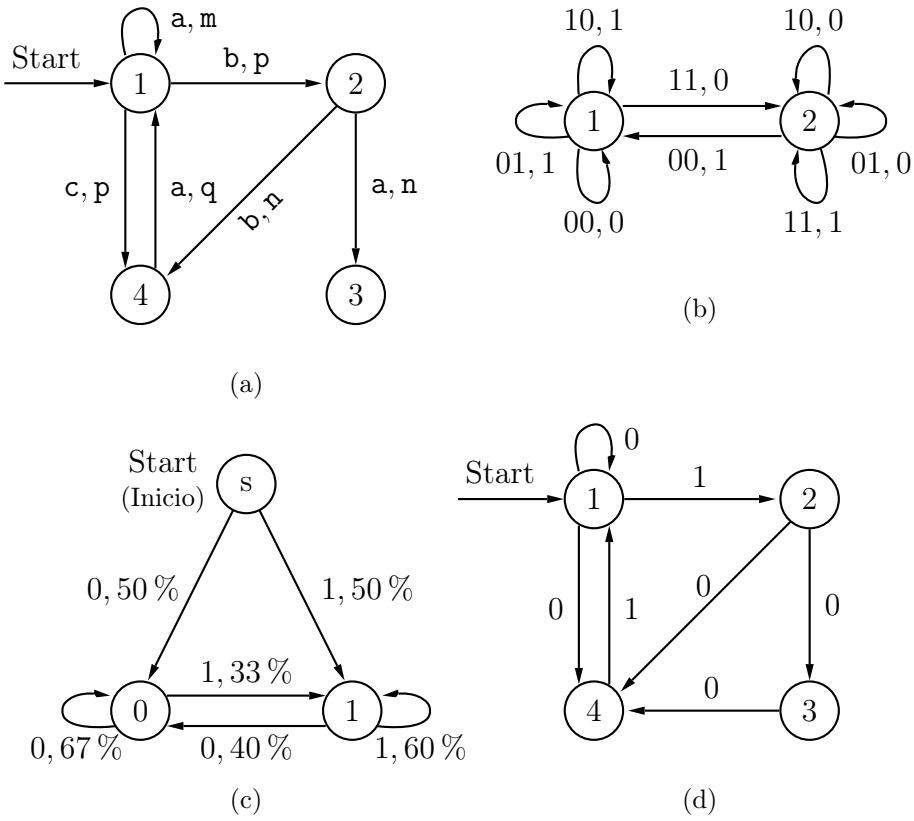


Figura F.2: Ejemplos de autómatas finitos.

que recibe una cadena de bits, donde el primer bit tiene una probabilidad del 50 % de ser un 0 ó un 1, y posiblemente cada bit consecutivo sea idéntico a su predecesor. Un cero sigue a otro cero con una probabilidad de $^2/_3$, y un uno sigue a otro uno con una probabilidad de $^3/_5$. Este AF no produce ninguna salida, pero cualquier AF práctico debe, por supuesto, hacer algo (i.e., producir una salida).

En general, un AF queda completamente definido especificando lo siguiente:

1. El conjunto de estados S.
2. El alfabeto A, de los símbolos de entrada y de salida.
3. Una función $f : S \times A \to S \times A$. Un ejemplo es $f\,(X, j) = (Y, i)$ que significa: si el estado actual es X y la entrada siguiente es j, cambia al estado Y y emite i.
4. El estado inicial S_0.

Matemáticamente, un AF (en inglés, FA) es un cuarteto (S, A, f, S_0).

Un AF es normalmente *determinista* (AFD), ya que para todos los estados y entradas hay exactamente una salida y un estado siguiente. Sin embargo, también puede haber un AF *no determinista*, denotado AFND (en inglés, NFA). La Figura F.2d muestra un ejemplo. Cuando un cero entra en el estado 2, el AFND puede pasar al estado 3 ó al estado 4. Los AFND son útiles para probar teoremas y también en la teoría de lenguajes y la teoría del cálculo computacional. Siguiendo los estados y entradas, es trivial verificar que la cadena de entrada "10101010 . . ." es aceptada por el AFND de la

Figura F.2d (alterna entre los estados 1, 2, y 4), como las cadenas 100 y 101; sin embargo, la cadena 1000 no es aceptada. Hay que señalar que cada AFD es un AFND.

◇ **Ejercicio F.1 (sol. en pág. 1115):** ¿Es aceptada la cadena "1001010010 . . ." por la AFND de la Figura F.2d?

El conjunto de todas las cadenas de entrada aceptados por un AF se llama *lenguaje* del AF. Puede demostrarse que los lenguajes aceptados por los autómatas finitos son los lenguajes designados mediante *expresiones regulares*, por lo que podemos concluir este breve capítulo definiendo las expresiones regulares. Empezamos ilustrando los conceptos de *concatenación* y *cierre*.

Sean $L_1 = (10, 1)$ y $L_2 = (011, 11)$ dos conjuntos de símbolos (cadenas binarias). La concatenación de L_1 y L_2 se denota por L_1L_2 y es el conjunto de cadenas que son concatenaciones de 10 (ó 1) y 011 (ó 11). Por consiguiente, $L_1L_2 = (10011, 1011, 111)$. El cierre de un conjunto L de símbolos se denota por L^*. Es el conjunto de todas las cadenas construidas con símbolos de L (incluyendo el conjunto vacío). Por lo tanto, si $L = (10, 11)$, entonces $L^* = (() , 10, 11, 1010, 1011, 1110, 1111, \ldots)$, donde el par de paréntesis denota el conjunto vacío. Si L es un conjunto de símbolos, entonces las expresiones regulares sobre L se definen recursivamente como sigue:

1. El símbolo vacío es una expresión regular.
2. El conjunto vacío () es una expresión regular.
3. Para cada $s \in L$, el conjunto (s) es una expresión regular.
4. Si r y s son expresiones regulares, entonces $(r + s)$ (unión de conjuntos), (rs) (concatenación de conjuntos) y (r^*) (cierre del conjunto) son expresiones regulares.

Los autómatas finitos son utilizados por varios métodos de compresión, entre ellos WFA (Sección 4.31) y la codificación dinámica de Markov (Sección 8.8).

[Hopcroft y Ullman 79] es una introducción detallada a los AFs y sus propiedades.

F.1. Bibliografía relacionada

Hopcroft, John E. and Jeffrey D. Ullman (1979) *Introduction to Automata Theory, Languages, and Computation, Reading*, MA, Addison-Wesley.

CAT, n. A soft, indestructible automaton provided by nature to be kicked when things go wrong in the domestic circle.
Ambrose Bierce. *The Devil's Dictionary*[a]

[a]GATO, n. Un autómata suave e indestructible proporcionado por la naturaleza para ser pateado cuando las cosas van mal en el círculo doméstico. El diccionario del Diablo, de Ambrose Bierce.

Apéndice G

Galería de imágenes

(Este apéndice es prácticamente idéntico a la Sección 4.7 de este mismo libro, por lo que sólo se hará referencia a las imágenes repetidas, sin embargo el texto se conserva por no ser muy extenso)

Los nuevos métodos de compresión de datos que se han desarrollado e implementado tienen que ser probados. Distintos test sobre los mismos datos hacen posible comparar su rendimiento tanto en eficiencia de compresión como en velocidad. Es por ésto que hay colecciones estándar de datos de prueba, como el "Calgary Corpus" y el "Canterbury Corpus" (que se mencionan en el prefacio), y el conjunto de ocho documentos de entrenamiento de la UIT-T para la compresión de faxes (Sección 2.13.1).

La necesidad de datos de prueba estándar se ha hecho sentir también en el campo de la compresión de imágenes, y actualmente, existen colecciones de imágenes fijas de uso general por investigadores e implementadores en este campo. Tres de las cuatro imágenes mostradas aquí, a saber, "Lena", "mandril", y "pimientos" (Figura 4.53, 4.54, y 4.56, respectivamente), son sin duda las más conocidas de ellas. Son imágenes de tonos continuos, aunque "Lena" tiene algunas características de imágenes en tonos discretos.

Cada imagen va acompañada de un detalle, que muestra los píxeles individuales. Es fácil ver por qué la imagen de los "pimientos" es de tonos continuos. Los píxeles adyacentes que difieren mucho en color son bastante raros en esta imagen. La mayor parte de los píxeles vecinos son muy similares. En contraste, la imagen del "mandril", aunque natural, es un mal ejemplo de una imagen en tonos continuos. El detalle (mostrando parte del ojo derecho y el área que lo rodea) muestra que muchos píxeles difieren considerablemente de sus vecinos inmediatos, debido al vello facial del animal en esta zona. Esta imagen se comprime mal bajo cualquier método de compresión. Sin embargo, el área de la nariz, sobre todo la parte azul y roja, es de tonos continuos. La imagen de "Lena" está formada en su mayor parte por tonos continuos puros, sobre todo en el muro y en las áreas de piel desnudas. El sombrero es bueno en tonos continuos, mientras que el pelo y la pluma en el sombrero son malos en tonos continuos. Las líneas rectas en la pared y las partes curvadas del espejo tienen características de una imagen de tonos discretos.

La imagen de "Lena" es ampliamente utilizada por la comunidad de procesamiento de imágenes, además de a ser popular en la compresión de imágenes. Debido al interés en la misma, su origen e historia han sido investigados y están bien documentados. Esta imagen forma parte de las páginas centrales de la revista *Playboy* de noviembre de 1972. Retrata a la playmate sueca Lena Soderberg (née Sjooblom) y fue descubierta, recortada, y escaneada a principios de la década de 1970 por un investigador desconocido en la Universidad del Sur de California para utilizarla como un imagen de prueba para su investigación sobre compresión de imágenes. Desde entonces se ha convertido en la imagen más importante, mejor conocida y más comúnmente usada en la historia de la formación

Figura G.1: El cubo RGB.

de imágenes y las comunicaciones electrónicas. Como resultado, Lena actualmente se considera por muchos como Primera Dama de Internet. *Playboy*, que normalmente procesa a los usuarios de sus imágenes no autorizados, se enteró del uso inusual de una de sus imágenes con derechos de autor, pero decidió darle sus bendiciones a esta particular "aplicación".

Lena vive actualmente y trabaja en Suecia. Se le comunicó su "fama" en 1988, quedó sorprendida y divertida por este hecho, y fue invitada a participar en la conferencia del 50º Aniversario de la IS&T (la sociedad para la Ciencia de la Imagen y la Tecnología) celebrada en Boston, en Mayo de 1997. En la conferencia ella autografió su imagen, posó para nuevas fotos (una está disponible en Internet) e hizo una presentación (sobre sí misma, no sobre la compresión de imágenes).

Las tres imágenes están ampliamente disponibles para su descarga en Internet.

La Figura G.1 es un diagrama en forma de cubo de la gama de colores del modelo RGB. (Véase el apéndice H, Subsección H.4.1.

La Figura 4.57 muestra una típica imagen en tonos discretos, cuyo detalle es mostrado en la Figura 4.58. Observe las líneas rectas y el texto, donde ciertos caracteres aparecen varias veces (una fuente de redundancia). Esta imagen en particular tiene pocos colores, pero en general, una imagen de tonos discretos puede tener muchos colores.

El pueblo de Lena, Illinois se encuentra a unas 9 millas al oeste de Freeport, Illinois y a 50 millas al este de Dubuque, Iowa. Estamos en el borde de las colinas del noroeste de Illinois y a sólo 25 millas al sur de Monroe, Wisconsin. La población actual de Lena es de aproximadamente 2800 almas dedicadas a la actividad agrícola.

De `http://www.lena.il.us/History.htm`

`Una onza de imagen vale más que una libra de rendimiento.`
`Anónimo`

Apéndice H

El sistema visual humano

La compresión de imágenes y vídeo debe lidiar con grandes cantidades de datos, y una forma de lograr una buena compresión de imágenes es perder algunos de estos datos. La pérdida de datos debe hacerse de manera selectiva y el principio guía es la pérdida de datos para los que el sistema visual humano no es sensible. Ésto requiere un conocimiento detallado del color, sus representaciones, así como la forma en que el ojo las percibe. El hardware utilizado para mostrar el color es también una consideración importante en cualquier aplicación de ordenador que utiliza el color. El CRT (Sección 6.1.1) es actualmente un dispositivo de salida común. Presenta un color por medio de fósforos emisores de luz, pero ellos no pueden reproducir los colores puros del espectro. Las impresoras a color utilizan varias tintas (en forma de líquido, cera, o polvo) para mezclar los colores en papel, y sufren la misma deficiencia. Impresoras, monitores CRT, pantallas LCD, y todos los otros dispositivos de salida tienen sus limitaciones y funcionan mejor cuando se utilizan ciertas representaciones del color. Este apéndice es una introducción al color, su representaciones, y la manera en que el ojo percibe el color.

Algunas de las ideas y de los resultados que aquí se presentan han sido desarrollados por el Comité International de Iluminación (Commission Internationale de l'Éclairage, o CIE), una organización internacional con una larga historia esclarecedora sobre diversos aspectos de la luz y la iluminación.

H.1. El color y los ojos

Rara vez nos encontramos en completa oscuridad. De hecho, la mayoría de las veces estamos inundados de luz. Por eso la gente se acostumbra a tener luz alrededor y, como consecuencia, sólo en raras ocasiones se preguntan *¿qué es la luz*?

Actualmente, la mejor explicación es que la luz puede entenderse (o interpretarse) de dos formas diferentes, como una onda, o como un flujo de partículas. Esta última interpretación ve la luz como un flujo de partículas sin masa, denominadas *fotones*, moviéndose a una velocidad constante. El atributo más importante de un fotón es su frecuencia, ya que la energía del fotón es proporcional a la misma. Los fotones son útiles en física para explicar multitud de fenómenos (como la interacción entre la materia y la luz). En los gráficos por computadora, la propiedad más importante de la luz es su color, que es por lo que usamos la última interpretación y consideramos la luz como una onda.

Lo que se conoce por "ondas" (u ondulaciones) en la luz son campos eléctricos y magnéticos. Cuando una región del espacio se inunda de luz, esos campos cambian periódicamente a medida que nos movemos de un punto a otro en el espacio. Si nos quedamos en un punto, los campos también cambian periódicamente con el tiempo. Por consiguiente, la luz visible es una (pequeña) parte del espectro electromagnético (Figura H.1), que incluye las ondas de radio, los ultravioleta, los infrarrojos, los rayos `X`, y otros tipos de radiación.

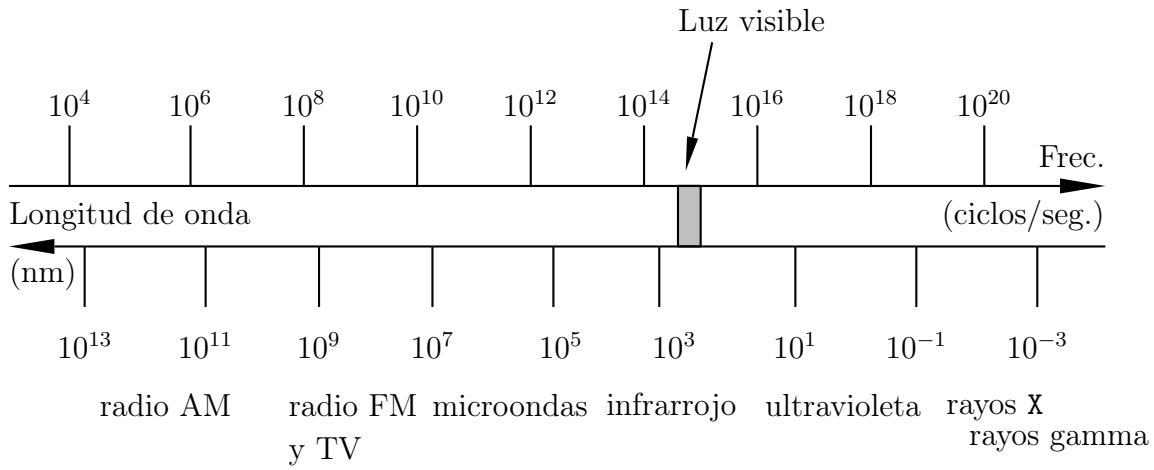


Figura H.1: El espectro electromagnético.

Las propiedades más importantes de una onda son su frecuencia f, su longitud de onda λ, y su velocidad. La luz se mueve, obviamente, a la velocidad de la luz (en el vacío, es $c \approx 3 \times 10^{10}$cm/s). Las tres cantidades están relacionadas mediante $f\lambda = c$. Es importante darse cuenta de que la velocidad de la luz depende del medio en el que se mueve. Cuando la luz se mueve desde el vacío del aire al vidrio, se ralentiza (en el vidrio, la velocidad de la luz es de aproximadamente 0,65c). Su longitud de onda también disminuye, pero su frecuencia se mantiene constante. Sin embargo, es habitual referirse a los colores por la longitud de onda y no por la frecuencia. Puesto que la luz visible tiene longitudes de onda muy cortas, una unidad conveniente es el nanómetro ($1\,\text{nm} = 10^{-9}$m).

El rango de la luz visible abarca desde aproximadamente 400 nm hasta unos 700 nm y el color es determinado por la longitud de onda. Una longitud de onda de 420 nm, por ejemplo, corresponde al violeta puro, mientras que 620 nm es percibida por el ojo como rojo puro. Usando láseres especiales, es posible crear luz pura (monocromática) formada por una única longitud de onda (Figura H.2a). La mayoría de las fuentes de luz, sin embargo, producen luz que es una mezcla de varias (o incluso muchas) longitudes de onda, normalmente con una longitud de onda dominante (Figuras H.2b y H.12).

Los colores del espectro que son más visibles para el ojo humano son (Figura H.7) el violeta (390–430), azul-violeta (460–480), cian, verde (490–530), amarillo (550–580), naranja (590–640), y rojo (650–800).

La luz blanca es una mezcla de todas las longitudes de onda, pero ¿qué es el color gris claro? Resulta que la longitud de onda de la luz no es su único atributo importante. La intensidad es otro atributo que debe ser considerado. La luz gris es una mezcla de todas las longitudes de onda, pero con una intensidad más baja. Al hacer gráficos por ordenador, el principal problema es cómo especificar el color exacto de cada píxel en el hardware. En muchas situaciones de la vida real, es suficiente decir: "Creo que me gustaría un traje azul marino"; pero el hardware del computador requiere, por supuesto, una especificación mucho más precisa. Por consiguiente, sorprende descubrir que ese color puede ser completamente especificado por sólo tres parámetros. Su significado depende del *modelo de color* en particular utilizado. El modelo RGB es popular en los gráficos por ordenador. En la industria gráfica, normalmente se utiliza el modelo CMYK. Muchos artistas utilizan el modelo HLS. Estos modelos son discutidos a continuación.

La Figura H.2b muestra un diagrama simplificado de la luz centrado sobre la gama completa de longitudes de onda visibles, con un pico en alrededor de los 620 nm (rojo), donde tiene una intensidad mucho mayor. Esta luz puede ser descrita especificando su *tonalidad*, *saturación*, y *luminancia*. La tonalidad del color es su longitud de onda dominante —620 nm en nuestro ejemplo—. La luminancia

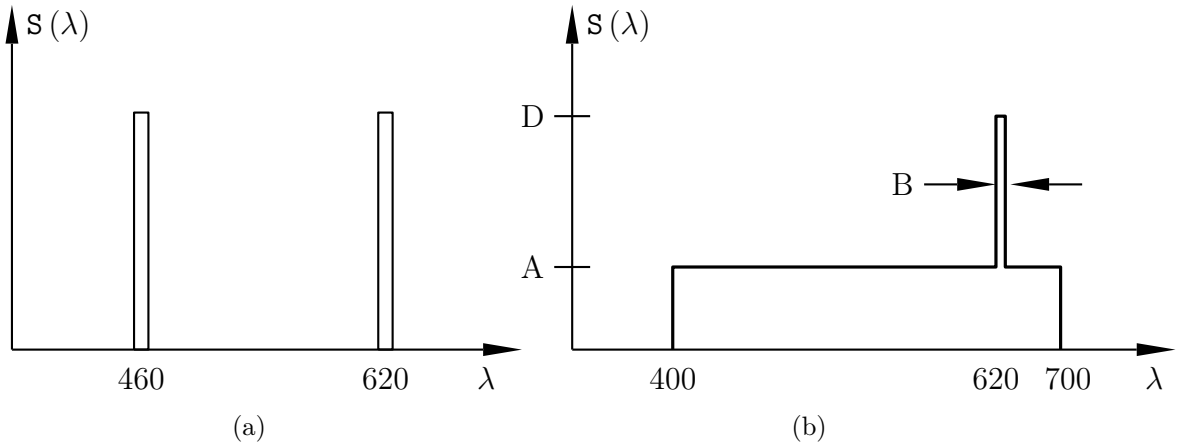


Figura H.2: (a) Colores puros. (b) Una longitud de onda dominante.

está relacionada con la intensidad de la luz. Se define como la potencia total incluida en el espectro y es proporcional al área bajo la curva, la cual es $L = (700 - 400)\, A + B\,(D - A)$. La saturación se define como el porcentaje de luminancia que reside en la longitud de onda dominante. En nuestro caso, es $B\,(D - A)\,/L$. Cuando no hay ninguna longitud de onda dominante (i.e., cuando $D = A$ ó $B = 0$), la saturación es cero y la luz es blanca. Una saturación grande significa, o bien $D - A$ grande, o bien L pequeña. En cualquier caso, hay menos blanco en el color y vemos más de la tonalidad roja. Una saturación grande, por lo tanto, corresponde al *color puro*.

H.2. El modelo de color HLS

Este modelo fue introducido en 1978 por Tektronics, con el objetivo de especificar los colores una manera intuitiva. El nombre es un acrónimo de *Hue* (tonalidad), *lightness* (luminosidad) y *saturation* (saturación). La luminosidad (o valor) se refiere a la cantidad de negro en el color. Ésta controla el brillo del color. La máxima luminosidad siempre crea el blanco, independientemente de tonalidad. La mínima luminosidad genera el negro. La saturación (o croma) se refiere a la cantidad de blanco en el color. Ésta controla la pureza o la intensidad del color. Una baja saturación significa más blanco en el color, lo que resulta en un color pastel. Una saturación muy baja produce un color desvaído. Para un color puro, intenso, la saturación debe ser máxima. Los colores acromáticos negro, blanco y gris tienen saturación cero y difieren en su valores.

El modelo HLS se resume en el doble cono de la Figura H.3. El eje vertical corresponde a L (luminosidad). Comienza en cero (negro) en la parte inferior y termina en uno (blanco) en la parte superior. La distancia desde el eje central corresponde a S (saturación). Todos los puntos del eje tienen saturación cero, por lo que se corresponden a tonos de gris. Los puntos más alejados del eje tienen más saturación; corresponden a los colores más vivos. El parámetro H (tono) corresponde a la tonalidad del color. Este parámetro se mide como un ángulo de rotación alrededor del hexágono.

H.3. El Modelo de color HSV

El modelo HSV también utiliza el tono, la saturación y el valor (luminosidad). Se resume en el cono de la Figura H.4. Éste es un único cono, donde el valor V, que corresponde a la luminosidad, va

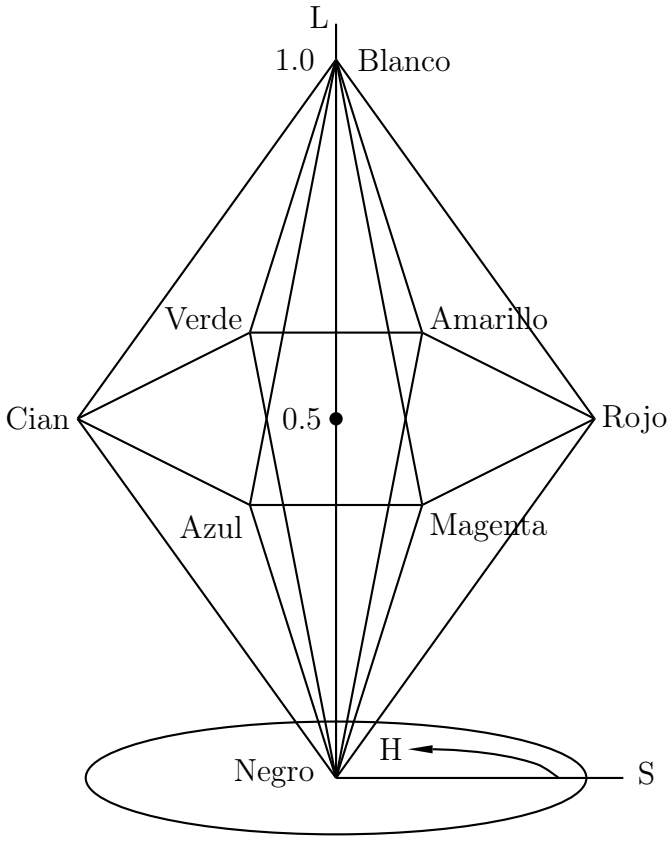


Figura H.3: El doble hexcono de HLS.

de 0 (negro) en la parte inferior, a 1 (blanco) en la parte superior plana. Los parámetros S y H tienen los mismos significados que en el doble cono HLS.

> Es la extraña combinación de colores que me alucina. Cada vez que intentas manejar uno de estos extraños controles negros, que están etiquetados en negro sobre un fondo negro, unas pequeñas luces iluminan el negro ¡para hacerte saber que lo has hecho!
>
> — Mark Wing-Davey (como Zaphod Beeblebrox) en *La Guía del Autoestopista de la Galaxia* (1981).

H.4. El modelo de color RGB

Una *tonalidad primaria* es un color en un modelo de color que no puede construirse a partir de los otros colores usados en ese modelo. Las tonalidades primarias sirven como base para la mezcla y la creación de todos los demás colores en el modelo de color. Cualquier color creado mediante la mezcla de *dos* colores primarios en un modelo de color es un *color secundario* en dicho modelo.

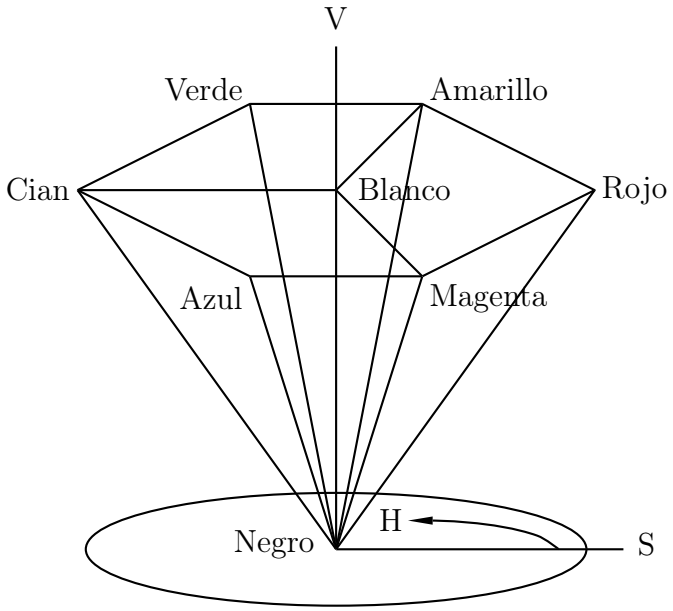


Figura H.4: El hexcono de HSV.

En el modelo de color RGB, los tres colores primarios son rojo (*Red*), verde (*Green*) y azul (*Blue*). Ellos pueden combinarse, de dos en dos para crear los tonos secundarios. El magenta (tono rosado) es (R+B), el cian (tono azulado) es (B+G), y el amarillo (R+G). Hay dos razones para usar el rojo, verde y azul como colores primarios: (1) los conos en el ojo son muy sensibles a estos colores (Figura H.7) y (2) la adición de rojo, verde, y azul puede producir muchos colores (aunque no todos los colores; véase la discusión de la gama de color RGB en la página 1017).

El modelo de color RGB es útil en gráficos por ordenador debido a la forma en que trabajan los CRTs. Éstos crean diferentes colores a partir de la luz emitida por fósforos de diferentes tipos. Los colores se mezclan luego en el ojo del observador, creando la impresión de una mezcla perfecta. Suponiendo un rango de $[0, 255]$ para cada componente de color RGB, he aquí algunos ejemplos de mezclas de colores:

$$\text{rojo} = (255, 0, 0)\,,\ \text{magenta} = (255, 0, 255)\,,\ \text{blanco} = (255, 255, 255)\,,$$
$$50\,\%\ \text{gris} = (127, 127, 127)\,,\ \text{gris claro} = (25, 25, 25)\,.$$

H.4.1. El Cubo RGB

La *gama de colores* de un modelo de color es el rango completo de colores que pueden producirse mediante el modelo. La gama de colores del modelo RGB puede resumirse en un diagrama en forma de cubo (Figuras H.5 y G.1). Cada punto en el cubo tiene tres coordenadas (r, g, b) —cada uno en el rango $[0, 1]$— que proporcionan las intensidades de rojo, verde, y azul del punto. Los valores pequeños, cercanos a $(0, 0, 0)$, significan una sombra oscura, mientras que algo cercano a $(1, 1, 1)$ es muy brillante. El punto $(1, 1, 1)$ mismo se corresponde con el blanco puro. El punto $(1, 0, 0)$ corresponde al rojo y el punto $(0, 1, 0)$ corresponde a verde. Por consiguiente, el punto $(1, 1, 0)$ describe una mezcla de rojo y verde, esto es, amarillo.

El cubo RGB es útil porque las coordenadas de los puntos en él pueden ser fácilmente traducidas a valores almacenados en la tabla de búsqueda de color de la computadora.

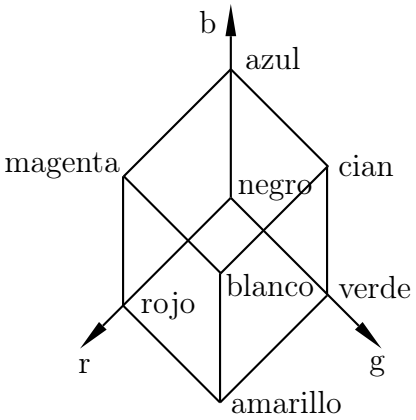


Figura H.5: El cubo RGB.

◇ **Ejercicio H.1 (sol. en pág. 1115):** (Fácil.) ¿Qué colores se corresponden con la línea diagonal que conecta el esquinas blanco y negro del cubo RGB?

> Los distintos colores que pueden obtenese mediante la mezcla de otros colores, son innumerables. Sólo propongo dar aquí los mejores y más sencillos modos de preparar aquellos cuyo uso es requerido. Los colores compuestos, formados por la unión de sólo dos colores, son llamados por los pintores tintas vírgenes. Cuanto menor sea el número de colores de cada componente de color del compuesto, más puro y rico será. Se preparan como sigue:
>
> —Daniel Young, 1861, *Young's Translation of Scientific Secrets.* (*Traducción joven de secretos científicos*).

H.5. Colores aditivos y sustractivos

Para crear una mezcla de varios colores, a veces hay que adicionarlos y a veces hay que restarlos. Imagina una pared blanca en un cuarto oscuro. No hay luz que pueda reflejar la pared, por lo que se ve negra. Ahora hacemos brillar una luz roja en ella. Puesto que la pared es blanca (refleja todos los colores), reflejará la luz roja y se verá roja. Lo mismo es cierto para la luz verde. Si ahora brillan tanto la luz roja como la verde en la pared, reflejará ambas, lo que nuestro cerebro interpreta como amarillo. Decimos que en este caso el colores se suman.

Para comprender el concepto de sustracción de colores, imagine una hoja de papel blanco en un entorno brillante. El papel refleja todos los colores, por lo que parece blanco. Si queremos para pintar un punto rojo determinado, tenemos que cubrirlo con una sustancia química (pintura roja) que absorbe todos los colores excepto el rojo. Decimos que la pintura roja *resta* el verde y el azul de la reflexión de blanco original, por lo que el punto ahora refleja sólo la luz roja. Similarmente, si queremos un punto amarillo, tenemos que utilizar pintura de color amarillo, que es una sustancia que absorbe el azul y refleja el rojo y el verde.

Concluimos que cuando hacemos brillar luz blanca sobre una superficie reflectante, tenemos que restar colores para conseguir el color exacto que queremos. Cuando hacemos brillar luz de varios colores

sobre una superficie, hay que añadir colores para obtener cualquier mezcla deseada.

> Por ejemplo, el ojo humano y su software de control encarnan implícitamente la falsa teoría de que la luz amarilla se compone de una mezcla de luz roja y verde (en el sentido de que la luz amarilla nos produce la misma sensación que una mezcla de luz roja y luz verde). En realidad, los tres tipos de luz tienen diferentes frecuencias y no pueden ser creados mezclando luz de otras frecuencias. El hecho de que una mezcla de luz roja y verde nos parezca luz amarilla no tiene nada que ver con las propiedades de la luz, sino que es una característica de nuestros ojos. Es un resultado de un compromiso de diseño que se produjo en algún momento evolutivo de nuestros antepasados.
>
> —David Deutsch, La estructura de la realidad.

H.5.1. Los modelos de color sustractivo

Existen dos modelos de este tipo: pigmentos de pintor y pigmentos de impresión.

Pigmentos de pintor: Los colores principales de este modelo de color son rojo (R), amarillo (Y), y azul (B).[1] Ellos fueron elegidos por artistas del pasado creían que eran colores puros, que no contienen trazas de otros colores. Estos tres colores primarios pueden mezclarse, de dos en dos, para producir los secundarios púrpura (R+B), verde (B+Y) y naranja (Y + R). La mezcla de cantidades iguales de los tres colores primarios sustrae todos los colores y, por lo tanto, produce el negro.

Pigmentos de impresión: Este modelo de color es también conocido como *proceso de color* y el proceso es el resultado del desarrollo de tinta de color y de los procesos de impresión. Los tres colores primarios son el magenta (M), amarillo (Y) y cian (C). Los tres secundarios son de color azul (M+C), rojo (M+Y), y verde (C+Y). La mezcla de cantidades iguales de los tres colores primarios debe producir el negro; pero, debido a las propiedades de las tintas reales, este negro no es normalmente lo suficientemente oscuro. En la práctica, el negro verdadero se incluye en este modelo como un cuarto primario artificial (también porque la tinta negra es más barata). Se utiliza cuando se requiere la impresión en escala de grises o negro. El modelo es a veces llamado CMYK (K para el negro, para evitar confusiones con el azul) y la impresión en color se conoce como proceso cuatricolor. La Figura H.6 muestra las relaciones entre los tres primarios CMY y sus secundarios.

Debido a los particulares primarios y secundarios del modelo CMY, existe una sencilla relación entre éste y el modelo RGB. La relación es:

$$(r, g, b) = (1, 1, 1) - (c, m, y)\,.$$

Esta relación muestra que, por ejemplo, aumentando la cantidad de cian en un color, se reduce la cantidad de rojo.

La impresión a color tradicional utiliza separación de colores. El primer paso es fotografiar la imagen original a través de filtros de colores diferentes. Cada filtro separa un color primario del original multicolor. Un filtro azul separa las partes amarillas del original y crea una transparencia con las partes impresas en blanco y negro. Un filtro rojo separa las partes en cian y un filtro verde separa las partes en magenta. Se prepara otra transparencia, con las partes negras. Cada una de las cuatro transparencias se convierte entonces en una imagen de semitonos (Sección H.11) y las cuatro imágenes se convierten en maestras de la impresión final. Se colocan en diferentes etapas de la máquina de impresión y, como el papel se mueve a través de la máquina, cada etapa añade puntos de semitonos de

[1] R=Red=Rojo; B=Blue=Azul; Y=Yellow=Amarillo.

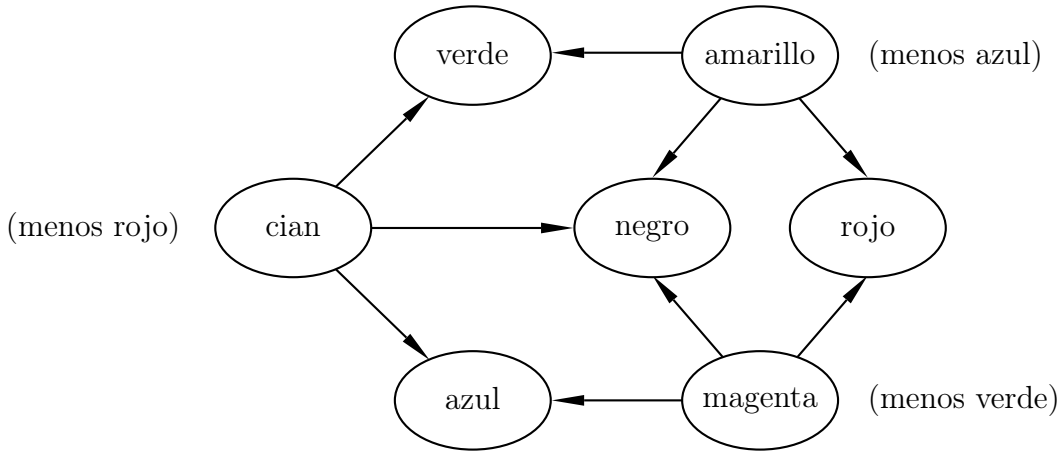


Figura H.6: Relaciones entre RGB y CMYK.

tinta de color al papel. El resultado es una imagen construida a partir de rejillas de semitonos, cada una en uno de los colores CMYK. Las rejillas no se superponen en el papel, sino que se imprimen desplazadas. El ojo ve puntos en los cuatro colores primarios, y el cerebro crea un color mixto que depende del número de puntos de semitono de cada primario.

Cuando una impresión en color se mantiene cerca del ojo, pueden verse los puntos individuales en los cuatro colores. Hoy en día, hay impresoras de sublimación de tinta que mezclan ceras de tintes diferentes dentro de la impresora para crear una gota de cera del color adecuado que es entonces depositada en el papel. No se utiliza ningún semitono. El resultado es una imagen en colores vivos, pero estas impresoras son actualmente demasiado caras para el uso casero. Para obtener más información sobre la impresión en color y el modelo CMYK, véase [Stone et al. 88].

Para simplificar la tarea de los editores y diseñadores gráficos, se han desarrollado varios estándares de color. En lugar de calcular las proporciones de CMYK, los diseñadores gráficos miran una tabla que tiene muchas muestras de color, seleccionan una, y utilizan su nombre para especificarlo a la impresora. Una de tales normas de uso común hoy en día es el sistema de correspondencia Pantone. Éste se describe en [Pantone 91].

◇ **Ejercicio H.2 (sol. en pág. 1115):** Una superficie tiene un color determinado debido a su capacidad de absorber y reflejar la luz. Una superficie que absorbe la mayor parte de las frecuencias de luz aparece oscura; una superficie que refleja la mayoría de las frecuencias aparece brillante. ¿Qué colores son absorbidos y reflejados en una superficie de color amarillo?

Blueness doth express trueness[a].
—Ben Jonson

[a] Lo azulado hace explícita la veracidad.

H.6. Colores complementarios

El concepto de los colores complementarios se basa en la idea de que dos colores aparecen psicológicamente armoniosos si su mezcla produce el blanco. Imagina el color del espectro entero. La suma

de todos los colores produce el blanco. Si restamos un color, digamos, el azul, la suma de los colores restantes produce el color complementario, el amarillo. El azul y el amarillo son, por lo tanto, colores complementarios (una díada) en un modelo de color aditivo. Otras díadas son: verde y magenta, rojo y cian, amarillo-naranja y azul cian, cian-rojo y verde-magenta, y amarillo-verde y azul-violeta.

Colores complementarios sustractivos: Éstos se basan en la idea de que dos colores tienen aspecto armonioso si su mezcla produce un tono de gris. Las díadas de sustracción son: amarillo y violeta, rojo y verde, azul y naranja, amarillo-naranja y azul-violeta, azul-verde y rojo-naranja, amarillo-verde y rojo-violeta.

Los colores complementarios producen contraste visual fuerte, lo que crea una sensación de vibraciones o actividad de color.

◇ **Ejercicio H.3 (sol. en pág. 1115):** ¿Existe tal cosa como tríadas de colores aditivos?

H.7. Visión humana

Vemos la luz que entra en el ojo y cae sobre la retina, donde hay dos tipos de células fotosensibles. Éstas contienen pigmentos que absorben la luz visible y por eso nos dan el sentido de la visión. Un tipo son los *bastones* (o *bastoncillos*), que son numerosos, se extienden por toda la retina, y sólo responden a la luz y a la oscuridad. Son muy sensibles y pueden responder a un solo fotón de luz. Hay alrededor de 110 000 000 a 125 000 000 bastoncillos en el ojo [Osterberg 35]. El otro tipo son los *conos*, que están localizados en la pequeña área de la retina (la fóvea). Su número es de aproximadamente 6 400 000; son sensibles al color, pero requieren una luz más intensa, del orden de cientos de fotones. Incidentemente, los conos son muy sensibles al rojo, verde y azul (Figura H.7), que es una de las razones por las que los CRTs utilizan estos colores como primarios. Hay tres tipos de conos. Los conos A son sensibles a la luz roja. Los conos B son sensibles a la luz verde (un poco más que los conos A), y los conos C son sensibles a la luz azul, pero su sensibilidad es de aproximadamente 1/30 la de los conos A o B.

La teoría tricromática de la visión del color dice que la luz de una determinada longitud de onda estimula cada uno de los tres tipos de conos en una forma diferente, y es la *ratio* (o *razón*) de esos estímulos, no sus valores absolutos, lo que crea la sensación de color en el cerebro. Como resultado, cualquier intensidad de luz e iluminación de fondo que casualmente produzcan la misma ratio de estímulos parecerán tener el mismo color. Esta teoría explica por qué cualquier representación de color utiliza tres parámetros.

Cada uno de los sensores de luz en el ojo, bastones y conos, envía una sensación de luz al cerebro que es esencialmente un píxel, y el cerebro combina estos píxeles en una imagen continua. El ojo humano es, por lo tanto, similar a una cámara digital. Una vez que uno se apercibe de ello, naturalmente deseamos comparar la resolución del ojo con la de una cámara digital moderna. Las cámaras digitales actuales tienen de 300 000 sensores (para un cámara barata) a cerca de seis millones de sensores (para una de alta calidad).

Por consiguiente, el ojo tiene una resolución mucho más alta, pero su resolución efectiva es incluso mayor si tenemos en cuenta que el ojo puede moverse y reorientarse a sí mismo alrededor de tres a cuatro veces por segundo. Esto significa que en un segundo, el ojo puede detectar y enviar al cerebro alrededor de quinientos millones de píxeles. Asumiendo que nuestra cámara toma una instantánea una vez por segundo, la ratio de las resoluciones es de aproximadamente 100.

Ciertos colores —como el rojo, el naranja y el amarillo— se asocian psicológicamente con el calor. Ellos son considerados *calientes* y provocan que una imagen parezca más grande y cercana de lo que realmente es. Otros colores —tales como el azul, el violeta y el verde— se asocian con cosas frías (aire, cielo, agua, hielo) y por eso se llaman colores *fríos*. Hacen que una imagen parezca más pequeña y lejana.

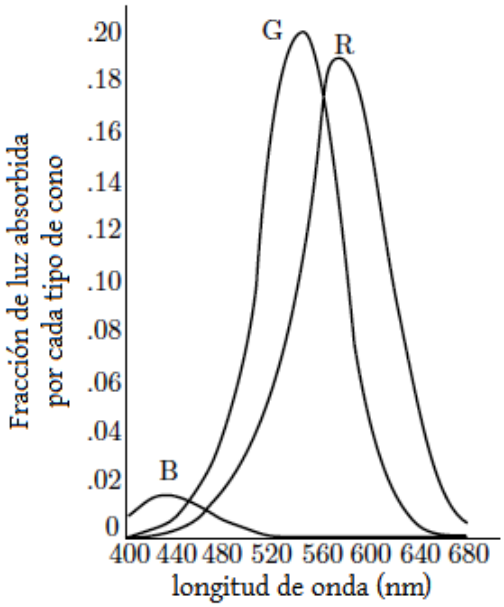


Figura H.7: Sensibilidad de los conos.

Color

Los colores, como los rasgos, siguen los cambios de las emociones.
— Pablo Picasso
No hay azul sin amarillo y sin naranja.
—Vincent Van Gogh

H.8. Luminancia y crominancia

RGB y otras representaciones de color discutidas anteriormente tienen un inconveniente importante: no se basan en el sistema visual humano. Sabemos que los tres tipos de conos en el ojo son sensibles al verde y (un poco menos) al rojo, y son menos sensibles al azul. También se sabe que el ojo puede resolver mejor detalles espaciales pequeños en el centro del espectro visible, que es verde, y es más pobre en este respecto con el azul. Ésto es importante para un algoritmo de compresión de imagen, puesto que significa que no es necesario preservar los pequeños detalles en las regiones en azul de una imagen, pero los pequeños detalles en verde son importantes y deben ser preservados.

Resulta que una representación de color basada en las dos cantidades *luminancia* y *crominancia* puede tomar ventaja de estas características más que la representación RGB. La luminancia es una cantidad que está estrechamente relacionada con cómo es percibido por el ojo el brillo de una fuente de luz. Se define como la proporción de energía luminosa emitida por la fuente de luz por unidad de área. La crominancia se relaciona con la manera en que el ojo percibe el tono y la saturación.

Un aspecto importante del sistema visual humano es que la respuesta perceptual L^* del ojo a una fuente de luz con luminancia Y no es lineal, sino que es proporcional a $Y^{1/3}$. Específicamente, la CIE

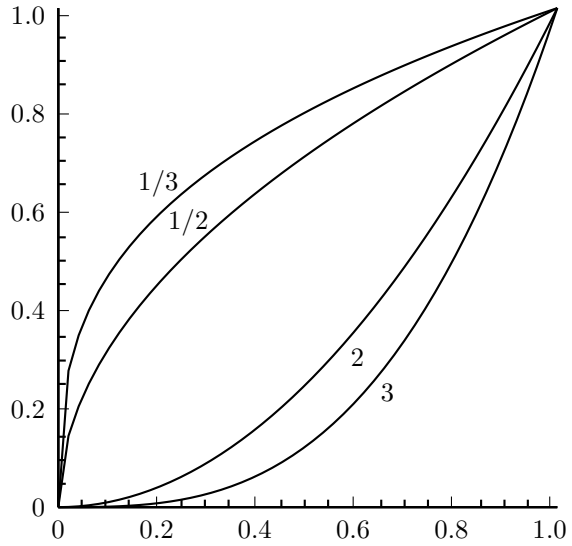


Figura H.8: Representaciones de x^γ y $x^{1/\gamma}$ para $\gamma = 2, 3$.

ha encontrado que:

$$L^* = \begin{cases} 116\,(Y/Y_n)^{1/3} - 16, & \text{si } Y/Y_n > 0{,}008856, \\ 903{,}3\,(Y/Y_n)\,, & \text{en otro caso,} \end{cases}$$

donde la cantidad L^* es llamada *luminosidad* e Y_n es la luminancia de referencia blanca (véase la discusión sobre el blanco luminoso en la Sección H.10). La región $Y/Y_n \leq 0{,}008856$ corresponde a valores bajos de Y (muy oscuro) y no es importante.

Además de esta función (o problema) del sistema visual humano, el CRT es en sí mismo la fuente de otra problema físico. Resulta que la intensidad I de la luz emitida por un CRT depende linealmente de la tensión V que alimenta el cañón de electrones. La relación es $I = V^\gamma$, donde la tensión (o voltaje) V se asume que está en el rango $[0, 1]$, y γ (gamma) es un número real, típicamente en el rango $[2, 3]$. En consecuencia, la intensidad de la luz es una función en algún lugar entre el cuadrado y el cubo de la tensión.

Una de las muchas coincidencias afortunadas de la vida es que estas dos características se cancelan entre sí. Las dos relaciones $L^* = 116\,(Y/Y_n)^{1/3} - 16$ e $I = V^\gamma$ implican que la luminosidad percibida por el ojo es proporcional al voltaje que alimenta al CRT si γ es 3 ó cercano a 3. La Figura H.8 muestra los gráficos de x^2 y $x^{(1/2)}$ (y también x^3 y $x^{1/3}$). Es fácil ver cómo la adición de dos gráficos de este tipo produce una línea recta (i.e., dependencia lineal).

Cabe destacar que este comportamiento no lineal del CRT no es causado por el fósforo, pero se deriva de las propiedades del cañón de electrones (el cátodo). Como la mayoría de los CRTs son similares eléctricamente, el valor gamma para un determinado CRT depende principalmente de su brillo y contraste. Los usuarios de los CRTs deberían por tanto ajustar el valor gamma de su monitor con los controles de intensidad y contraste, usando patrones de prueba especiales. La Figura H.9 es un patrón donde el objetivo es encontrar la barra donde el brillo de las partes superior e inferior concuerden. Ésto debe hacerse a partir de una distancia de 6–10 pies.[2] Para algunas personas es más cómodo buscar la barra donde la línea del medio desaparece, o casi. Una vez que la barra es encontrada, el valor gamma de la pantalla es conocido y los expertos recomiendan ajustarlo entre 2,2 y 2,5.

[2]Un pie son unos 30,48 cm.

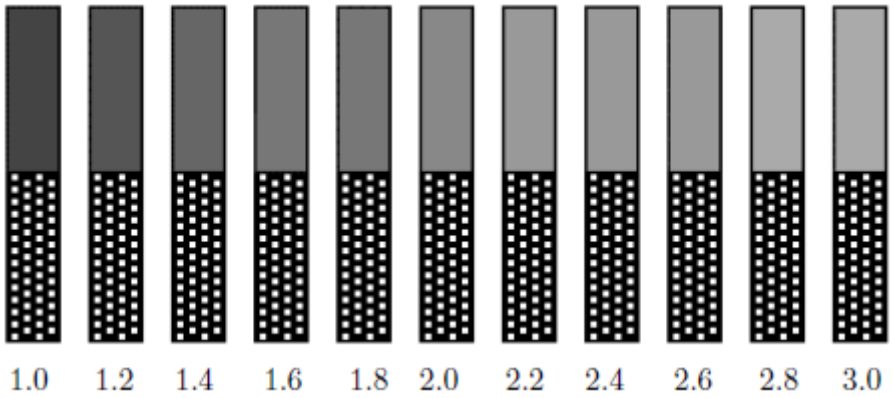


Figura H.9: Un patrón de prueba gamma.

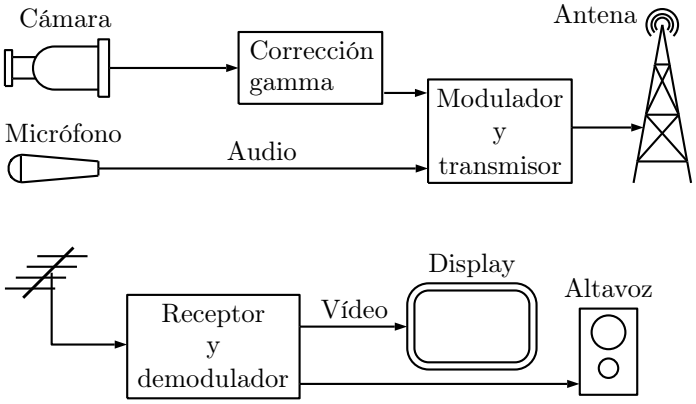


Figura H.10: Transmisión de televisión con corrección gamma.

Pasamos ahora al importante (y confuso) concepto de *corrección gamma*. Acabamos de indicar que la luminosidad percibida por el ojo es proporcional al voltaje que alimenta al CRT. Obviamente, tiene que asegurarse que la tensión que alimenta el CRT es proporcional a la intensidad de la luz vista por la cámara de vídeo. El sensor de luz en la cámara de vídeo (típicamente CCD) también tiene un valor gamma y emite una tensión V de la forma $V = I^{\gamma}$ donde I es la intensidad "vista" por el sensor. Es por ésto que la camára tiene un circuito adicional incorporado, que corrige el comportamiento no lineal del sensor efectuando el cambio $V \leftarrow V^{1/\gamma}$ y produce una tensión V que es proporcional a la intensidad de la luz que incide sobre el sensor. La Figura H.10 muestra un diagrama de bloques de una cámara de vídeo y un receptor de televisión.

En los gráficos de ordenador, la corrección gamma es implementada en la tabla de búsqueda de colores. Para tener una idea de cómo se hace ésto, el lector debe considerar otra característica del sistema visual humano. El sistema ojo-cerebro puede detectar diferencias en la intensidad de la luz, pero esta capacidad no es uniforme de negro a blanco. Es aproximadamente el 1 % de la intensidad misma. Al mirar imágenes oscuras, podemos detectar pequeñas variaciones en la intensidad. Al ver una imagen brillante, sólo podemos detectar los cambios grandes. Con ésto en mente, imagine una tabla de búsqueda de colores con valores de luminancia de 8 bits, donde 0 representa el negro y 255 representa el blanco. Si dos entradas en la tabla tienen valores de 25 y 26, entonces la diferencia

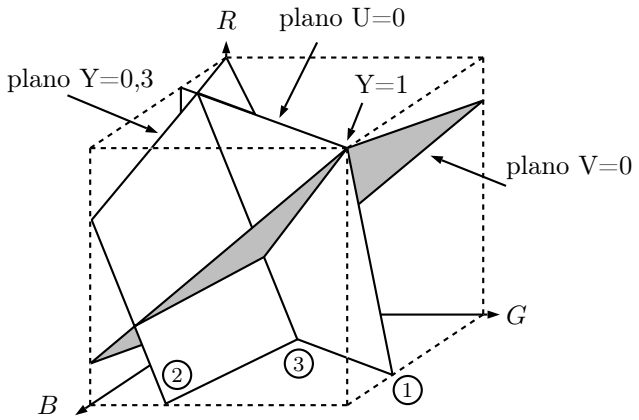


Figura H.11: Relación entre las representaciones de color RGB e YUV.

entre ellos es 1, pero esta unidad de diferencia es el 4 % de 25. El cambio del valor de un píxel de 25 a 26 incrementaría el brillo del píxel en un 4 %, una diferencia en intensidad grande y perceptible. Similarmente, si el valor de un píxel es cambiado de 100 a 101, entonces esta única unidad de cambio representa el 1 % de 100, y es, por lo tanto, apenas perceptible para el ojo. El cambio en un píxel de 200 a 201 aumenta el brillo del píxel un 0,5 % y sería imperceptible. El valor 201 es, en efecto, inútil, y por otro lado, nos gustaría tener uno o dos valores entre 25 y 26. Ésto sugiere una corrección gamma que asigna pequeñas diferencias de intensidad a valores de píxeles pequeños y grandes diferencias de intensidad a valores grandes de píxeles. Idealmente, los valores adyacentes deben corresponder valores de luminancia que difieren en alrededor del 1 %, y una forma de lograr ésto es interpretar los valores de píxel como el logaritmo de la luminancia.

¡Los lectores suelen encontrar que la corrección gamma es un tema confuso! Nuestro objetivo, sin embargo, es entender la luminancia y la crominancia, y comenzamos declarando que los componentes RGB de un color reciben una corrección gamma (en la cámara de vídeo o en la tabla de búsqueda), y los nuevos valores corregidos, (que también están normalizados en el rango $[0, 1]$) se denotan R', G', y B'. Cada uno de estos tres componentes contribuye de manera diferente a la luminosidad percibida por el ojo (debido a los tres tipos de conos en la retina), por lo que se define una nueva cantidad Y', como la suma ponderada:

$$Y' = 0{,}299R' + 0{,}587G' + 0{,}114B'. \tag{H.1}$$

Y' se llama *luma*, pero la mayoría de autores lo llaman luminancia. Dado que los tres componentes RGB corregidos están normalizados, Y' también está normalizado en el rango $[0, 1]$. Los dos componentes de crominancia U' y V' se definen como las diferencias $U' = B' - Y'$ y $V' = R' - Y'$. Ellas pueden interpretarse como la presencia o ausencia de azul y rojo en el color, respectivamente. Puesto que todos —R', G', B' e Y'— varían en el rango $[0, 1]$, es fácil verificar que U' varía en el rango $[-0{,}886, +0{,}886]$ y V' varía en el rango $[-0{,}701, +0{,}701]$, lo que sugiere que están normalizados. Antes de hacer eso vamos a tratar de obtener una mejor comprensión de las relaciones entre las representaciones de color RGB y YUV (los primos se omiten por conveniencia). La Figura H.11 muestra el cubo RGB en discontinuo y los tres planos $Y = 0{,}3$, $U = 0$, y $V = 0$. Estos dos últimos planos se cortan en puntos que satisfacen $R = G = B = Y$, por lo que la intersección es la línea que corresponde a todos los colores grises.

Un punto en el plano $U = 0$ es (0, 0,755, 0,5). Un punto en el plano $V = 0$ es (0,5, 0,5, 0,5). El lector debe intentar localizar estos puntos en el diagrama.

◇ **Ejercicio H.4 (sol. en pág. 1115):** ¿Cuáles son las coordenadas RGB de los tres puntos etiquetados

(1), (2), y (3) en la Figura H.11?

A continuación, llegamos a las componentes de crominancia. Debido a los rangos de variación de U' y V', son fáciles de normalizar. Las dos cantidades normalizadas se denotan por Cb y Cr (para crominancia azul y crominancia roja) y se definen por

$$Cb = (U'/2) + 0{,}5, \quad Cr = (V'/1{,}6) + 0{,}5.$$

La representación de color $Y\,CbCr$ se utiliza comúnmente en vídeo y gráficos por ordenador, por lo que los métodos de compresión de imagen y vídeo a menudo asumen que las imágenes a ser comprimidas están representadas de esta manera.

Las tres representaciones de color RGB, YUV e $Y\,CbCr$ son matemáticamente equivalentes, ya que es posible transformar cada uno de ellos en los otros. La ventaja de $Y\,CbCr$ es que el ojo es más sensible a las pequeñas variaciones de luminancia espacial que a las pequeñas variaciones de color. Un método de compresión que lidia con una imagen en formato $Y\,CbCr$ debe mantener el componente Y sin pérdidas (o casi sin pérdidas), pero puede permitirse de perder la información de los componentes de crominancia.

La transformación de $R'G'B'$ en $Y'\,CbCr$ se efectúa mediante:

$$\begin{pmatrix} Y' \\ Cb \\ Cr \end{pmatrix} = \begin{pmatrix} 16 \\ 128 \\ 128 \end{pmatrix} + \begin{pmatrix} 65{,}481 & 128{,}553 & 24{,}966 \\ -37{,}797 & -74{,}203 & 112 \\ 112 & -93{,}786 & -18{,}214 \end{pmatrix} \begin{pmatrix} R' \\ G' \\ B' \end{pmatrix}.$$

La transformación inversa es:

$$\begin{pmatrix} R' \\ G' \\ B' \end{pmatrix} = \begin{pmatrix} 0{,}00456621 & 0 & 0{,}00625893 \\ 0{,}00456621 & -0{,}00153632 & -0{,}00318811 \\ 0{,}00456621 & 0{,}00791061 & 0 \end{pmatrix} \left[\begin{pmatrix} Y' \\ Cb \\ Cr \end{pmatrix} - \begin{pmatrix} 16 \\ 128 \\ 128 \end{pmatrix} \right].$$

Nota: La luminancia (o luma) definida por la Ecuación (H.1) es parte de la recomendación BT.601-4 de la ITU-R. Estrictamente hablando debería ser denotado Y_{601} y llamarse luma de video no lineal. Existe también una cantidad llamada luma lineal, la cual es definida mediante la RGB lineal (sin corregir). Se define por la recomendación 709 de la ITU-R como:

$$Y_{709} = 0{,}2125R + 0{,}7154G + 0{,}0721B.$$

H.9. Densidad espectral

Un láser es capaz de emitir luz "pura", i.e., exactamente una longitud de onda. La mayor parte de las fuentes de luz, sin embargo, emite luz "sucia" que es una mezcla de muchas longitudes de onda, normalmente con una dominante. Para cada fuente de luz, el gráfico de la intensidad de la luz como una función de la longitud de onda λ se llama *densidad espectral* de la fuente de luz. La Figura H.12 muestra las densidades espectrales de varias fuentes de luz típicas.

Estos sencillos diagramas ilustran un problema al tratar de especificar el color sistemáticamente y sin ambigüedades. Distintas densidades espectrales pueden ser percibidas por nosotros como idénticas. Cuando los colores creados por estas densidades espectrales están situados uno junto a otro, nos resulta imposible distinguir entre ellos. El primer paso en la solución del problema es la correspondencia de color. Supongamos que usamos un modelo de color definido por los tres primarios $A(\lambda)$, $B(\lambda)$, y $C(\lambda)$ y tenemos un color descrito por la densidad espectral $S(\lambda)$. ¿Cómo podemos expresar $S(\lambda)$ en términos de los tres primarios? Una forma de hacer esto es hacer brillar un punto de $S(\lambda)$ en una pantalla blanca y, justo al lado del mismo, un punto de luz $P(\lambda) = \alpha A(\lambda) + \beta B(\lambda) + \gamma C(\lambda)$, creado mediante la mezcla de los tres primarios (donde $0 \leq \alpha, \beta, \gamma \leq 1$). Ahora se cambian las cantidades α,

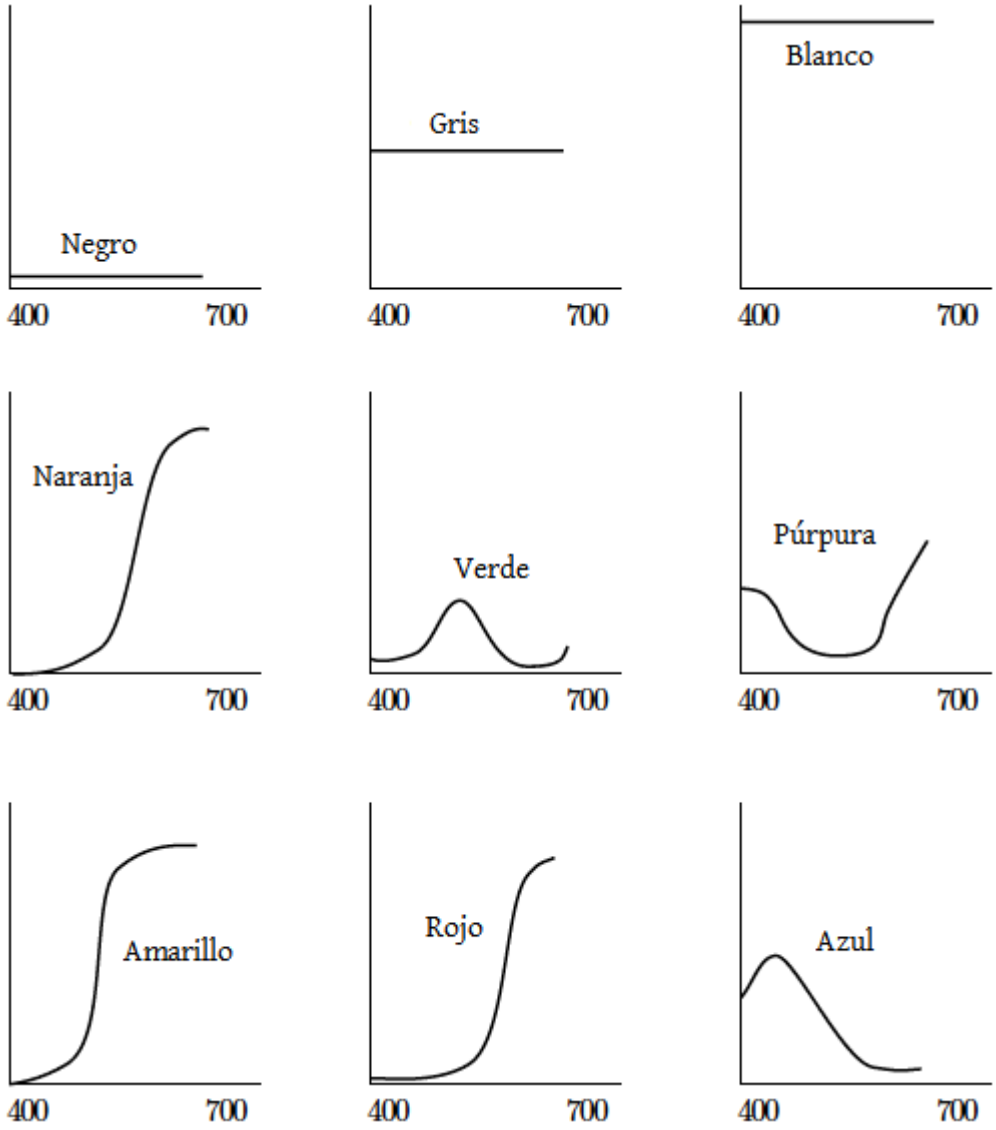


Figura H.12: Algunas densidades espectrales.

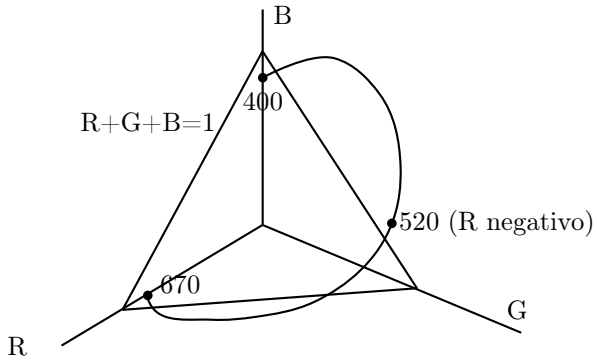


Figura H.13: Curva de color espectral RGB puro.

β, y γ hasta que un observador entrenado convenga que los puntos son indistinguibles. Ahora podemos decir que, en cierto sentido, $S(\lambda)$ y $P(\lambda)$ son idénticos, y escribir $S = P$.

¿En qué sentido es verdadero lo precedente? Resulta que la declaración anterior es significativa debido a una propiedad notable de los colores. Supongamos que dos densidades espectrales $S(\lambda)$ y $P(\lambda)$ tienen el mismo color percibido, por lo que escribimos $S = P$. Ahora seleccionamos otro color Q y lo hacemos brillar en ambos puntos, S y P. Sabemos por experiencia que los dos nuevos puntos también son indistinguibles. Ésto significa que podemos utilizar el símbolo + para agregar luces y podemos describir los dos puntos mediante $S(\lambda) + Q(\lambda)$ y $P(\lambda) + Q(\lambda)$. En resumen, podemos decir que "si $S = P$, entonces $S + Q = P + Q$." Lo mismo es cierto para el cambio de intensidades. Si $S = P$, entonces $\alpha S = \alpha P$ para cualquier intensidad α. Nosotros, por lo tanto, terminamos con un álgebra vectorial para los colores, donde un color puede ser entendido como un vector tridimensional, con las operaciones habituales de los vectores.

Dado lo anterior, podemos seleccionar un modelo de color basado en tres colores primarios, A, B, y C, y podemos representar cualquier color de S como una combinación lineal de los primarios $S = \alpha A + \beta B + \gamma C$. Podemos decir que el vector (α, β, γ) es la representación de S en la base (A, B, C). De forma equivalente, podemos decir que S está representado como el punto (α, β, γ) en el espacio tridimensional definido por los vectores $A = (1, 0, 0)$, $B = (0, 1, 0)$, y $C = (0, 0, 1)$.

Puesto que los gráficos tridimensionales son difíciles de dibujar sobre el papel, nos gustaría reducir artificialmente la representación de tres a dos dimensiones. Ésto se efectúa teniendo en cuenta que el vector $(2\alpha, 2\beta, 2\gamma)$ representa el mismo color como (α, β, γ), sólo dos veces tan brillante. Por lo tanto, nos limitamos a los vectores (α, β, γ), donde $\alpha + \beta + \gamma = 1$. Éstos son vectores normalizados a la unidad de brillo. Todos los vectores de la unidad brillo se encuentran en el plano $\alpha + \beta + \gamma = 1$ y éste es el plano (que, por supuesto, es bidimensional) que trazamos sobre el papel. Podemos especificar un color mediante el uso de dos números, digamos, α y β y calculamos γ como $1 - \alpha - \beta$.

Para el modelo de color RGB, ahora seleccionamos los colores espectrales puros utilizando expertos humanos entrenados. La idea es hacer brillar un punto de un color puro, digamos, de 500 nm, en una pantalla y, junto a él, un punto que es una mezcla $(r, g, b = 1 - r - g)$ de los tres primarios del modelo RGB. Los valores de r y g se van variando hasta que el observador juzga que los dos puntos son indistinguibles. El punto (r, g, b) es entonces representado en el espacio de color tridimensional RGB. Cuando todos los colores puros han sido representados de esta manera, los puntos se conectan para formar una curva suave, $\mathbf{P}(\lambda) = (r(\lambda), g(\lambda), b(\lambda))$. Ésta es la *curva de color espectral puro* del modelo RGB (Figura).

Una propiedad importante de la curva es que algunos de r, g, y b a veces tienen que ser negativos. Un ejemplo es $\lambda \approx 520$ nm, donde r resulta ser negativo. ¿Cuál es el significado de la adición de una

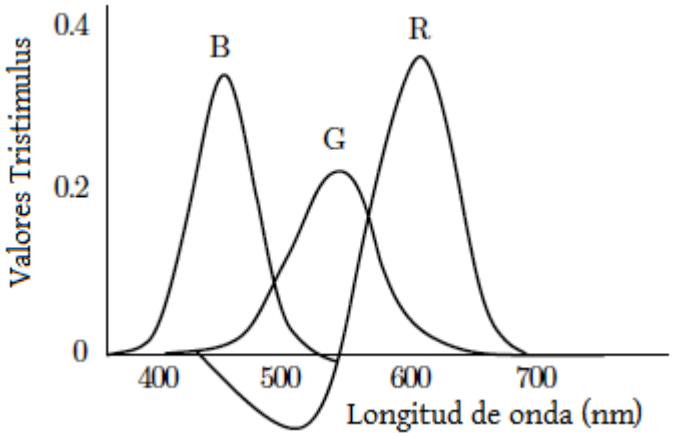


Figura H.14: Combinaciones de color RGB.

cantidad negativa de verde en un color definido mediante, por ejemplo, $S = 0{,}8R - 0{,}1G + 0{,}3B$?. La manera de comprender esto es escribir la ecuación $S + 0{,}1G = 0{,}8R + 0{,}3B$. Ahora significa que el color de S no puede ser construido a partir de los primarios RGB, pero puede colorear $S + 0{,}1G$. La conclusión importante es que ¡no todos los colores pueden ser creados en el modelo RGB! Ésto se ilustra en la Figura H.14. Para algunos colores sólo podemos crear una aproximación. Ésto es cierto para todos los modelos de color que pueden crearse en la práctica.

> ¿Alguna vez te he dicho que mi color favorito es el azul?
>
> —Jürgen Prochnow (como Sutter Cane) en *En la boca del miedo.*

H.10. El estándar CIE

Este estándar fue creado en 1931 por el CIE. Se basa en tres colores primarios artificiales cuidadosamente elegidos X, Y, y Z. No corresponden a colores reales, pero tienen la importante propiedad de que cualquier color real puede ser representado como una combinación lineal $xX + yY + zZ$, donde $x + y + z = 1$ y ninguno de x, y, y z son negativos (Figura H.15).

El plano $x + y + z = 1$ en el espacio $XY\,Z$ se llama diagrama de cromaticidad CIE (Figura H.16). La curva de color espectral puro en el diagrama CIE cubre todos los colores puros, de 420 nm a 660 nm. Tiene la forma de una herradura.

El punto $\omega = (0{,}310,\ 0{,}316)$ en el diagrama CIE es especial y se llama "blanco luminoso". Se asume que es un blanco totalmente insaturado y se utiliza en la práctica para concordar con colores que deben ser blanco puro.

◇ **Ejercicio H.5 (sol. en pág. 1115):** ¿Si el blanco luminoso es blanco puro, ¿por qué no está en la curva de color espectral puro en el diagrama CIE?

El diagrama CIE proporciona un estándar para describir los colores. Hay instrumentos que, dada una muestra de color, calculan las coordenadas (x, y) del color en la diagrama. Además, dadas las coordenadas CIE de un color, esos instrumentos pueden generar una muestra del color. El diagrama también puede utilizarse para los cálculos de color útiles. He aquí algunos ejemplos:

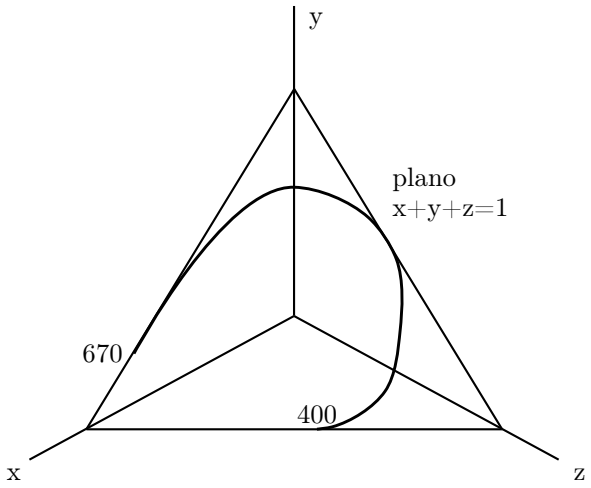


Figura H.15: Curva de color espectral puro.

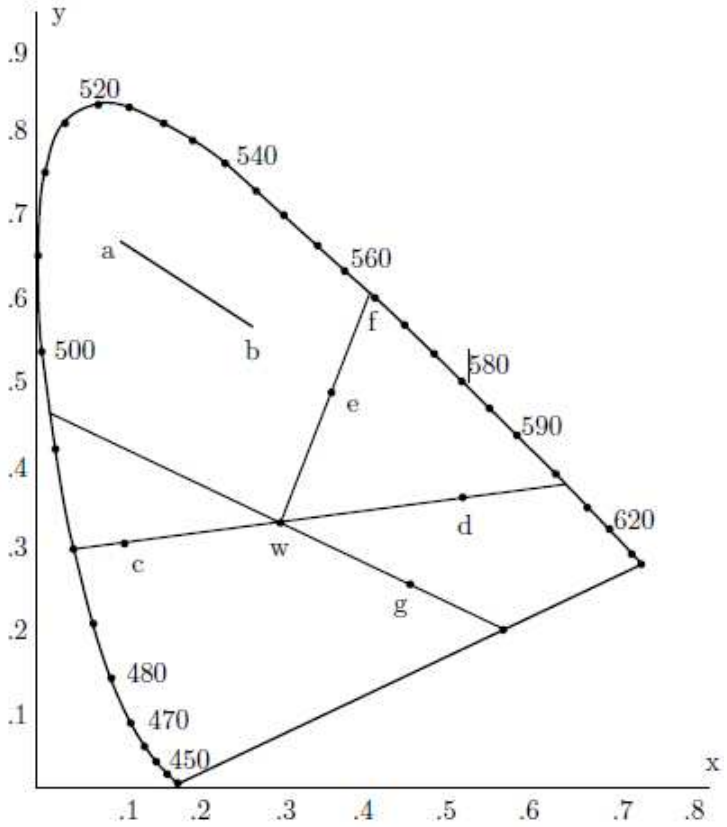


Figura H.16: El diagrama de cromaticidad CIE.

1. Dados dos puntos a y b en el diagrama (Figura H.16), la línea de conexión ellos tiene la forma $(1-\alpha)\,a + \alpha b$ para $0 \leq \alpha \leq 1$. Esta línea muestra todos los colores que pueden ser creados mediante la adición de cantidades variables de los colores a y b.

2. Imaginemos dos puntos, tales como c y d en la Figura H.16. Se encuentran en los lados opuestos del blanco luminoso y, por lo tanto, corresponden a colores complementarios.

◇ **Ejercicio H.6 (sol. en pág. 1115):** ¿Por qué es ésto cierto?

3. La longitud de onda dominante de cualquier color, como e en la Figura H.16 puede ser medido en el diagrama. Sólo hay que dibujar una línea recta desde el blanco luminoso ω hasta e y continuar hasta que intercepte a la curva de color espectral puro. A continuación se lee la longitud de onda en el punto de intercepción f (564 nm en nuestro ejemplo).

◇ **Ejercicio H.7 (sol. en pág. 1115):** ¿Cómo puede calcularse la saturación de color e a partir del diagrama?

◇ **Ejercicio H.8 (sol. en pág. 1115):** ¿Cuál es la longitud de onda dominante del punto g en el diagrama CIE?

> ¿Has perdido el juicio? ¿De qué color es este billete?
>
> —Lori Petty (como Georgia "George" Sanders) en *Vida lujuriosa (Lush Life)* (1996).

4. La *gama de color* de un dispositivo es el rango de colores que pueden ser mostrados por el dispositivo. Ésto también puede ser calculado con el diagrama CIE. Un monitor RGB, por ejemplo, puede mostrar combinaciones de rojo, verde y azul, pero ¿qué colores están incluidos en esas combinaciones? Para encontrar la gama de colores de un monitor RGB, que primero tiene que encontrar la ubicación del rojo, verde y azul puros en el diagrama [éstos son los puntos (0,628, 0,330), (0,285, 0,590), y (0,1507, 0,060)], y luego conectarlos con líneas rectas. La gama de colores se compone de todos los colores que se encuentren dentro del triángulo resultante. Cada uno puede expresarse como una combinación lineal de rojo, verde y azul, con coeficientes no negativos (una combinación convexa).

Resulta interesante, que debido a la forma de herradura, tres colores ubicados sobre o en su interior no pueden servir como primarios ideales. No importa qué tres puntos seleccionemos, algunos colores estarán fuera del triángulo definido por ellos. Ésto significa que ningún conjunto de tres primarios puede ser utilizado para crear todos los colores. El conjunto RGB tiene la ventaja de que el triángulo creado por él es grande, y por tanto contiene muchos colores. El triángulo CMY, por ejemplo, es mucho más pequeño en comparación. Ésta es otra razón para el uso del rojo, verde y azul como primarios RGB.

H.11. Semitonos (halftoning)

Las impresoras a color de inyección de tinta son comunes hoy en día, pero las tintas son todavía caras. La mayor parte de las impresoras láser son en blanco y negro. Semitonos es un método que hace

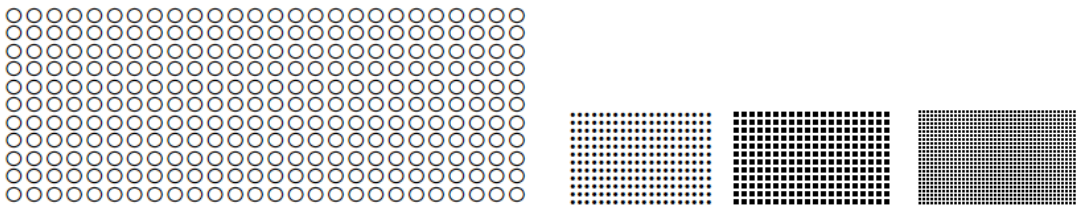


Figura H.17: Fondos grises obtenidos por integración espacial.

posible la impresión o visualización de imágenes con tonos de grises en un dispositivo de salida de blanco y negro (i.e., binivel). La desventaja es la pérdida de resolución. En lugar de píxeles pequeños, individuales, semitonos utiliza grupos de píxeles, donde sólo algunos de los píxeles en un grupo son de color negro. Semitonos es importante, ya que permite imprimir imágenes formadas por más colores que el blanco y negro en una impresora en blanco y negro. Se utiliza comúnmente en periódicos y libros. Una referencia clásica es [Ulichney 87].

El ojo humano puede resolver detalles tan pequeños como de 1 minuto de arco en condiciones de luz normales. Ésto se llama *agudeza visual*. Si vemos un área muy pequeña desde una distancia normal, nuestros ojos no pueden ver los detalles de esa área y termina integrándolos, de manera que sólo vemos una intensidad media procedente de ella. Esta propiedad se llama *integración espacial* y queda perfectamente demostrada mediante la Figura H.17. La figura consta de círculos negros y puntos en un fondo blanco, pero la integración espacial, crea el efecto de un fondo gris.

El principio de semitonos es utilizar grupos de $n \times n$ píxeles (con n normalmente en el rango de 2–4) y establecer algunos de los píxeles de un grupo en negro. Dependiendo del porcentaje de negro sobre blanco de los píxeles de un grupo, el grupo se asemeja a un cierto tono de gris. Un grupo de $n \times n$ píxeles contiene n^2 píxeles y por lo tanto puede proporcionar $n^2 + 1$ niveles de gris. El único problema práctico es encontrar el mejor patrón para cada uno de esos niveles. Los $n^2 + 1$ patrones de píxeles seleccionados deberán cumplir las siguientes condiciones:

1. Las áreas cubiertas por copias del mismo patrón no deben presentar ninguna textura.
2. Cualquier píxel establecido en negro para el patrón k también debe ser negro en todos los patrones de niveles de intensidad $> k$. Ésto se considera una buena *secuencia de crecimiento* y minimiza las diferencias entre patrones de intensidades sucesivas.
3. Los patrones deben crecer desde el centro del área $n \times n$, para crear el efecto de un punto de crecimiento.
4. Todos los píxeles negros de un patrón deben ser adyacentes entre sí. Esta propiedad se denomina *clustered dot halftoning* (semitono por agrupación de puntos) y es importante si la salida está destinada a una impresora (las impresoras láser no siempre pueden reproducir completamente pequeños puntos aislados). Si la salida está destinada solamente al CRT, entonces se utiliza el *dispersed dot halftoning* (semitono por dispersión de puntos), donde los píxeles negros de un patrón no son adyacentes.

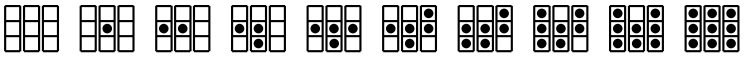
Como ejemplo simple de la condición 1, un patrón tal como:

debe evitarse, ya que las áreas grandes con grupos de 3 niveles generarían a largo plazo líneas horizontales. Otros patrones pueden producir molestas texturas similares. Con un grupo de 2×2, tales efectos pueden ser imposibles de evitar. Lo mejor que puede hacerse es usar los patrones: .

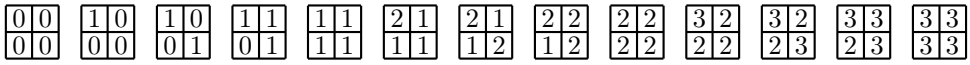
Un grupo de 3×3 ofrece más posibilidades. Los 10 patrones de más abajo ($= 3^2 + 1$) son los mejores posibles (las reflexiones y rotaciones de estos patrones se consideran idénticas) y normalmente evitan el problema anterior. Fueron producidos por la matriz:

$$\begin{bmatrix} 7 & 9 & 5 \\ 2 & 1 & 4 \\ 6 & 3 & 8 \end{bmatrix}$$

con la siguiente regla: Para crear un grupo con intensidad n, sólo las celdas con valores $\leq n$ en la matriz anterior deben ser negras.



El método de semitonos no se limita a una visualización monocromática. Imagine una pantalla con cuatro niveles de gris por píxel (píxeles de 2 bits). Cada píxel, o bien es negro, o bien puede estar en uno de los otros tres niveles. Un grupo de 2×2 consta de cuatro píxeles, cada uno de los cuales puede estar en uno de tres niveles de gris o en negro. El número total de niveles, por lo tanto, es de $4 \times 3 + 1 = 13$. A continuación se muestra un posible conjunto de los 13 patrones:



H.11.1. Bibliografía relacionada

Ulichney, Robert (1987) *Digital Halftoning*, Cambridge, MA, MIT Press.

H.12. Difuminado (dithering)

La desventaja de los semitonos es la pérdida de resolución. También es posible mostrar imágenes de tonos continuos (i.e., las imágenes con diferentes tonos de gris) en un dispositivo binivel *sin* pérdida de resolución. Tales métodos son a veces denominados *difuminado* (*dithering*) y su desventaja es la pérdida de detalles de la imagen. Si la resolución del dispositivo es lo suficientemente alta y si la imagen se observa desde la distancia adecuada, entonces nuestros ojos perciben una imagen en escala de grises, pero con menos detalles que en el original.

El problema del difuminado puede expresarse de la siguiente manera: dado un array A de $m \times n$ píxeles en escala de grises, calcular una matriz B del mismo tamaño con ceros y unos (correspondientes a los píxeles en blanco y negro, respectivamente) tal que para cada píxel $B[i, j]$ el valor medio del píxel y un grupo de sus vecinos cercanos igualará aproximadamente al valor normalizado de $A[i, j]$. (Suponga que un píxel $A[i, j]$ tiene un valor entero I en el intervalo $[0, a]$; entonces su valor normalizado es la fracción I/a, que está en el intervalo $[0, 1]$.)

El método de difuminado más simple utiliza un umbral y el único test: Establecer $B[i, j]$ a blanco (0) si $A[i, j]$ es lo suficientemente brillante (i.e., menor que el valor del umbral); de lo contrario, establecer $B[i, j]$ a negro (1). Este método es rápido y simple, pero genera resultados muy pobres, como muestra el siguiente ejemplo, por lo que no se usa nunca en la práctica. Como ejemplo, imagine una cabeza humana. El pelo es generalmente más oscuro que la cara debajo de él, por lo que el método del umbral simple puede cuantificar el área entera del pelo a negro y el área entera de la cara a blanco; un resultado muy pobre, irreconocible e inaceptable. (Hay que señalar, sin embargo, que algunas imágenes son inmediatamente reconocibles incluso en sólo blanco y negro, como muestra acertadamente la Figura H.18.) Este método puede ser mejorado mediante el uso de un umbral aleatorio, diferente para cada píxel, pero incluso ésto produce resultados de baja calidad.

Figura H.18: Una familiar imagen en negro y blanco.

El panda gigante se parece a un oso, aunque anatómicamente es más como un mapache. Vive en los bosques altos de bambú de la China central. Su cuerpo es en su mayoría blanco, con extremidades, orejas y parches oculares negros. Los adultos pesan de 200 a 300 libras (de 90 a 140 kg). Las bajas tasas de natalidad y la invasión humana de su hábitat ponen seriamente en peligro esta especie.

En esta sección se presentan cuatro enfoques para el difuminado, a saber, difuminado ordenado, difuminado promedio restringido, difuminado por difusión, y difusión de puntos. Otro enfoque, llamado ARIES, se discute en [Roetling 76] y [Roetling 77].

H.12.1. Difuminado (dither) ordenado

El principio de este método es dibujar un píxel $B[i,j]$ negro o dejarlo blanco, dependiendo de la intensidad del píxel $A[i,j]$ **y** de su posición en la imagen [i.e., de sus coordenadas (i,j)]. Si $A[i,j]$ es una sombra oscura de color gris, entonces $B[i,j]$ debería ser idealmente oscuro, en consecuencia, es pintado de negro la mayor parte de las veces, pero a veces se deja blanco. La decisión de pintar de negro o blanco depende de sus coordenadas i y j. Lo contrario es cierto para un píxel brillante. Este método se describe en [Jarvis et al. 76].

El método comienza con una matriz de difuminado D_{mn} de $m \times n$ que se utiliza para determinar el color (negro = 1 ó blanco = 0) de todos los píxeles B. En el ejemplo siguiente suponemos que los A píxeles tienen 16 niveles de gris, con 0 como blanco y 15 como negro. Las matrices de difuminado para $n = 2$ y $n = 4$ se muestra a continuación. La idea en estas matrices es minimizar la cantidad de textura en las zonas con un nivel de gris uniforme.

$$D_{22} = \begin{bmatrix} 0 & 2 \\ 3 & 1 \end{bmatrix}, \quad D_{44} = \begin{bmatrix} 0 & 8 & 2 & 10 \\ 12 & 4 & 14 & 6 \\ 3 & 11 & 1 & 9 \\ 15 & 7 & 13 & 5 \end{bmatrix}.$$

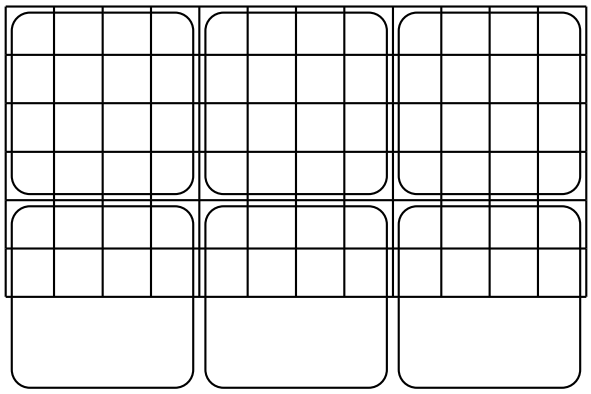


Figura H.19: Difuminado (*dither*) ordenado.

La regla es la siguiente: Dado un píxel $A[x, y]$ calcular $i = x \bmod m$, $j = y \bmod n$, luego seleccionar el negro (i.e., establecer $B[x, y]$ a 1) si $A[x, y] \geq D_{mn}[i, j]$ y seleccionar el color blanco en caso contrario.

Para ver cómo se utiliza la matriz de difuminado, imagine un área grande y uniforme en la imagen A donde todos los píxeles tienen un nivel de gris de 4. Puesto que el 4 es el quinto de los 16 niveles, nos gustaría terminar con $5/16$ de los píxeles en el área en negro (idealmente deben ser distribuidos al azar en esta área). Cuando una fila de píxeles es escaneada en esta zona, y es incrementado, pero x no cambia. Dado que i depende de x, y j depende de y, los píxeles escaneados son comparados con una de las filas de la matriz de D_{44}. Si resulta estar en la primera fila, entonces nos encontramos con la secuencia $10101010\ldots$ en un mapa de bits B.

Cuando la siguiente línea de píxeles es escaneada, x y, como resultado, i han sido incrementados, por lo que nos fijamos en la fila siguiente de D_{44}, que produce el patrón $01000100\ldots$ en B. El resultado final es un área en B que se asemeja a:

$$\begin{array}{l} 10101010\ldots \\ 01000100\ldots \\ 10101010\ldots \\ 00000000\ldots \end{array}$$

Diez de los 32 píxeles son de negros, pero $10/32 = 5/16$. Los píxeles negros no están distribuidos aleatoriamente en el área, pero su distribución no crea ningún patrón molesto.

◇ **Ejercicio H.9 (sol. en pág. 1115):** Asúmase que la imagen A tiene tres grandes áreas uniformes con niveles de gris 0, 1, y 15 y calcúlense los píxeles que forman el mapa de bits B para estas áreas.

El difuminado ordenado es fácil de entender si visualizamos las copias de la matriz de difuminado una junto a la otra en la parte superior del mapa de bits. La Figura H.19 muestra un mapa de bits 6×12 con seis copias de una matriz de difuminado de 4×4 yaciendo en la parte superior de la misma. El umbral para la interpolación de un píxel $A[i, j]$ es el elemento de la matriz de difuminado que descansa sobre la parte superior de $A[i, j]$.

La Matrix D_{44} de arriba fue creada a partir de D_{22} mediante la regla recursiva:

$$D_{nn} = \begin{pmatrix} 4D_{n/2,n/2} & 4D_{n/2,n/2} + 2U_{n/2,n/2} \\ 4D_{n/2,n/2} + 3U_{n/2,n/2} & 4D_{n/2,n/2} + U_{n/2,n/2} \end{pmatrix} \tag{H.2}$$

donde U_{nn} es una matriz $n \times n$ con todo unos. Otras matrices son fáciles de generar con esta regla.

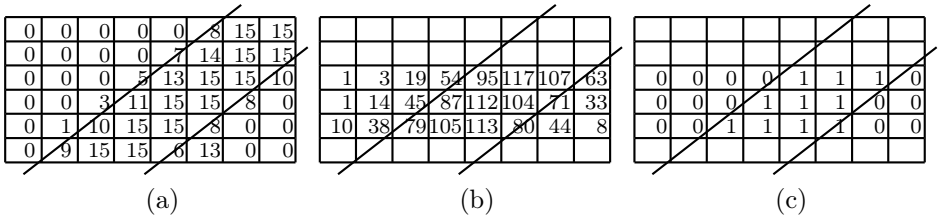


Figura H.20: Difuminación (*dithering*) promedio restringida.

◇ **Ejercicio H.10 (sol. en pág. 1115):** Utilícese la regla de la Ecuación (H.2) para construir D_{88}.

La regla básica del difuminado ordenado puede generalizarse como sigue: Dado un píxel $A[x, y]$, calcular $i = x \bmod m$ y $j = y \bmod n$, después seleccionar negro (i.e., asignar $B[x, y] \leftarrow 1$) si $Ave[x, y] \geq D_{mn}[i, j]$, donde $Ave[x, y]$ es el promedio del grupo de píxeles de 3×3 centrados en $A[x, y]$. Ésto es computacionalmente más intensivo, pero tiende a producir mejores resultados en la mayor parte de los casos, puesto que considera el brillo medio de un grupo de píxeles.

El difuminado ordenado es un método sencillo y rápido, pero tiende a crear imágenes que se han descrito por varias personas como "informatizadas", "frías" o "artificiales". La razón de ello es, probablemente, la naturaleza recursiva de la matriz de difuminado.

H.12.2. Difuminado promedio restringido

En aquellos casos en que la alta velocidad no es importante, este método [Jarvis y Roberts 76] da buenos resultados, aunque involucra más cálculos que el difuminado ordenado. La idea consiste en calcular, para cada píxel $A[i, j]$, el promedio $\overline{A}[i, j]$ del píxel y su ocho vecinos más cercanos. El píxel se compara entonces con un umbral de la forma

$$\gamma + \left(1 - \frac{2\gamma}{M}\right) \bar{A}[i, j],$$

donde γ es un parámetro seleccionado por el usuario y M es el valor máximo de $A[i, j]$. Observe que el umbral puede tener valores en el rango $[\gamma, M - \gamma]$. El paso final es comparar $A[i, j]$ con el umbral y establecer $B[i, j]$ a 1 si $A[i, j] \geq$ umbral y a 0 en caso contrario.

La principal ventaja de este método es la mejora de los bordes. Un ejemplo es la Figura H.20 que muestra un mapa de bits A de 6×8, donde los píxeles tienen valores de 4 bits. La mayoría de los píxeles tiene un valor de 0, pero el mapa de bits también contiene una línea gruesa inclinada mostrada en la figura. Ésta separa los píxeles 0 (blanco) en las esquinas superior izquierda e inferior derecha de los píxeles más oscuros ubicados en el medio.

La Figura H.20a muestra cómo los píxeles en torno a la línea tienen valores que se acercan al valor máximo (que es 15). La Figura H.20b muestra (para algunos píxeles) el promedio del píxel y sus ocho vecinos cercanos (los promedios se muestran como números enteros, por lo que un promedio de 54 realmente indica $54/9$). El resultado de la comparación de estos promedios respecto al umbral (que en este ejemplo es 75) se muestra en la Figura H.20c. Es fácil ver cómo la línea gruesa está claramente definida en la imagen binivel.

So I'm a ditherer? Well, I'm jolly well going to dither, then![a]

—Roland Young (as Cosmo Topper) in *Topper* (1937).

[a] ¡Así que estoy de los nervios? Bueno, ¡entonces me encanta estar de los nervios!

H.12.3. Difuminado (dither) por difusión

Imagine una fotografía, rica en color, que está siendo digitalizada con un escáner que se pueden distinguir millones de colores. El resultado puede ser un archivo de imagen en el que cada píxel $A[i,j]$ está representado por, digamos, 24 bits. El píxel puede tener uno de 2^{24} colores. Ahora, imagine que queremos mostrar este archivo en un equipo que sólo puede mostrar 256 colores simultáneamente en la pantalla. Un buen ejemplo es un equipo que utiliza una tabla de búsqueda de colores cuyo tamaño es de 3×256 bytes.

Comenzamos cargando una paleta de 256 colores en la tabla de búsqueda. Cada píxel $A[i,j]$ de la imagen original tendrá ahora que ser mostrado en la pantalla como un píxel $B[i,j]$ en uno de los 256 colores de la paleta. El siguiente diagrama muestra un píxel $A[i,j]$ con el color original $(255, 52, 80)$. Si decidimos asignar al píxel $B[i,j]$ el color de paleta $(207, 62, 86)$, entonces nos queda una diferencia de $A[i,i] - B[i,j] = (48, -10, -6)$. Esta diferencia se denomina *error de color* del píxel.

$$\begin{bmatrix} R = 255 \\ G = 52 \\ B = 80 \end{bmatrix} - \begin{bmatrix} R = 207 \\ G = 62 \\ B = 86 \end{bmatrix} = \begin{bmatrix} R = 48 \\ G = -10 \\ B = -6 \end{bmatrix}$$

Grandes errores de color degradan la calidad de la imagen mostrada, por lo que se necesita un algoritmo para minimizar el error de color total de la imagen. El difuminado por difusión efectúa ésto distribuyendo los errores de color entre todos los píxeles de forma que el error total de color para la imagen completa es cero (o muy cercano a cero).

El algoritmo es muy sencillo. Los píxeles $A[i,j]$ son escaneados línea a línea desde la parte superior hasta abajo. Cada línea, se escanea de izquierda a derecha. Para cada píxel, el algoritmo realiza lo siguiente:

1. Recoge la paleta de colores que está más cercana al color del píxel original. Esta paleta de colores está almacenada en el mapa de bits de destino $B[i,j]$.
2. Calcula el error de color $A[i,j] - B[i,j]$ para el píxel.
3. Distribuye este error a los cuatro vecinos más cercanos de los $A[i,j]$'s que no han sido escaneados todavía (el de la derecha y los otros tres por debajo centrados) en concordancia con el *filtro de Floyd-Steinberg* [Floyd y Steinberg 75] (donde la `X` representa el píxel actual):

	`X`	$^7/_{16}$
$^3/_{16}$	$^5/_{16}$	$^1/_{16}$

Consideremos el ejemplo de la Figura H.21a. El píxel actual es $(255, 52, 80)$ y nosotros (arbitrariamente) asumimos que el color de paleta más cercano es $(207, 62, 86)$. El error de color es $(48, -10, -6)$ y se distribuye como se muestra en la Figura H.21c. A los cinco vecinos más cercanos se les asignan nuevos colores como se muestra en la Figura H.21b. El algoritmo se muestra en la Figura H.22a donde los pesos $p1$, $p2$, $p3$, y $p4$ pueden ser asignados, o bien los valores de Floyd-Steinberg $^7/_{16}$, $^3/_{16}$, $^5/_{16}$, y $^1/_{16}$ o bien cualesquier otros valores.

Antes	R=255 G=52 B=80	R=178 G=20 B=60
R=192 G=45 B=75	R=250 G=49 B=83	R=191 G=31 B=72

(a)

Después	R=207 G=62 B=86	R=199 G=16 B=57
R=201 G=43 B=74	R=265 G=46 B=81	R=194 G=30 B=72

(b)

$$\begin{array}{llll} \frac{7}{16}\times 48 = 21, & \frac{1}{16}\times 48 = 3, & \frac{5}{16}\times 48 = 15, & \frac{3}{16}\times 48 = 9, \\ \frac{7}{16}\times (-10) = -4, & \frac{1}{16}\times (-10) = -1, & \frac{5}{16}\times (-10) = -3, & \frac{3}{16}\times (-10) = -2, \\ \frac{7}{16}\times (-6) = -3, & \frac{1}{16}\times (-6) = 0, & \frac{5}{16}\times (-6) = -2, & \frac{3}{16}\times (-6) = -1. \end{array}$$

(c)

Figura H.21: Difuminado (*dither*) por difusión.

```
for i := 1 to m do
  for j := 1 to n do
    begin
    B[i,j] := BuscarPaleta(A[i,j]);

    err:=A[i,j] - B[i,j];
    A[i,j+1]:=A[i,j+1] + err * p1;
    A[i+1,j-1]:=A[i+1,j-1] + err * p2;
    A[i+1,j]:=A[i+1,j] + err * p3;
    A[i+1,j+1]:=A[i+1,j+1] + err * p4;
    end.
```

(a)

```
for i := 1 to m do
  for j := 1 to n do
    begin
    if A[i,j] < 0.5 then B[i,j] := 0
      else B[i,j] := 1;
    err:=A[i,j] - B[i,j];
    A[i,j+1]:=A[i,j+1] + err * p1;
    A[i+1,j-1]:=A[i+1,j-1] + err * p2;
    A[i+1,j]:=A[i+1,j] + err * p3;
    A[i+1,j+1]:=A[i+1,j+1] + err * p4;
    end.
```

(b)

Figura H.22: Algoritmo de difuminado por difusión. Para (a) color. (b) binivel.

El error de color total no puede ser exactamente cero debido a que el método no funcionan bien para la columna del extremo izquierdo y para la fila inferior de píxeles. Sin embargo, los resultados pueden ser muy buenos si los colores de la paleta son cuidadosamente seleccionados.

Este método puede aplicarse fácilmente al caso de una pantalla monocromática (o cualquier dispositivo de salida binivel), tal como muestra el pseudocódigo de la Figura H.22b, donde $p1$, $p2$, $p3$, y $p4$ son los cuatro parámetros de difusión del error. Ellos pueden ser los ya dados (i.e., $^7/_{16}$, $^3/_{16}$, $^5/_{16}$, y $^1/_{16}$) o diferentes, pero su suma debe ser 1.

◇ **Ejercicio H.11 (sol. en pág. 1116):** Considérese una imagen totalmente gris, donde $A[i,j]$ es un número real en el rango $[0,1]$ y es igual a 0,5 para todos los píxeles. ¿Qué imagen B sería generada por el difuminado de difusión en este caso?

◇ **Ejercicio H.12 (sol. en pág. 1116):** Imagínese una imagen de escala de grises que consiste en una sola fila de píxeles donde los píxeles tienen valores reales en el intervalo $[0,1]$. El valor de cada píxel p es comparado con el valor de umbral de 0,5 y el error se propaga al vecino de la derecha. Muéstrese el resultado del difuminado de una fila de píxeles, todos con valores 0,5.

La difusión de errores también puede utilizarse para la impresión en color. Una impresora a color de chorro de tinta de gama baja típica tiene cuatro cartuchos de tinta para cian, magenta, amarillo, y negro. La impresora ubica puntos de tinta en la página de forma que cada punto tenga uno de los cuatro colores. Si una zona concreta de la página debe tener el color `L`, donde `L` no es ninguno de CMYK, entonces `L` puede ser simulado mediante el difuminado. Ésto se efectúa imprimiendo puntos adyacentes en el área con colores CMYK de manera que el ojo (que integra los colores de un área pequeña) perciba el color `L`. Ésto puede hacerse con la difusión del error, donde la paleta se compone de los cuatro colores —cian $(255,0,0)$, magenta $(0,255,0)$, amarillo $(0,0,255)$, y negro $(255,255,255)$— y el error para un píxel es la diferencia entre el color del píxel y el color de la paleta más cercano.

Una versión un poco más simple sobre difusión del error es el método *error promedio mínimo*. Los errores no se propagan, sino más bien son calculados y almacenados en una tabla separada. Cuando un píxel $A[x,y]$ es examinado, la tabla de errores es utilizada para localizar los errores $E[x+i,y+j]$ ya calculados para algunos vecinos $A[i,j]$ del píxel vistos anteriormente. Al píxel $B[x,y]$ se le asigna un valor de 0 ó 1 en función de la intensidad corregida:

$$A[x,y] + \frac{1}{\sum_{ij}\alpha_{ij}} \sum \alpha_{ij} E[x+i,y+j]\,.$$

El nuevo error, $E[x,y] = A[x,y] - B[x,y]$, es añadido entonces a la tabla de errores que se utilizará para los píxeles futuros. Las cantidades α_{ij} son los pesos asignados a los vecinos cercanos de $A[x,y]$. Pueden ser asignados de muchas maneras diferentes, pero deben asignar más peso a los vecinos cercanos, por lo que el siguiente es un ejemplo típico:

$$\alpha = \begin{pmatrix} 1 & 3 & 5 & 3 & 1 \\ 3 & 5 & 7 & 5 & 3 \\ 5 & 7 & x & - & - \end{pmatrix},$$

donde x es el píxel actual $A[x,y]$ y los pesos han sido definidos para algunos vecinos vistos anteriormente más arriba y a la izquierda de $A[x,y]$. Si los pesos suman 1, entonces la intensidad corregida anteriormente se simplifica y se convierte en:

$$A[x,y] + \sum_{ij} \alpha_{ij} E[x+i,y+j]\,.$$

La difusión de errores de Floyd-Steinberg generalmente produce mejores resultados que el difuminado ordenado, pero tiene dos inconvenientes, a saber, que es de naturaleza serie y que produce a veces

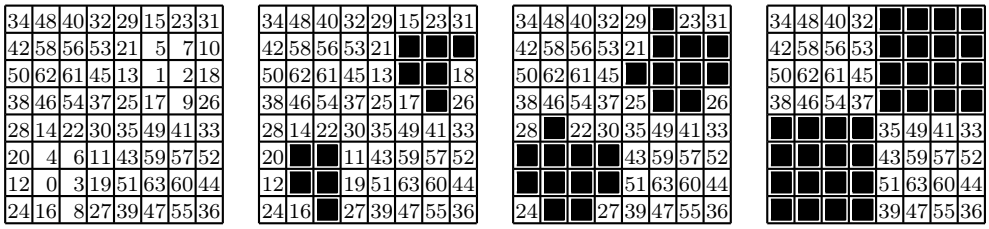


Figura H.23: Matrices de 8×8 para difusión de punto.

los molestos "fantasmas". El difuminado por difusión es en serie, ya que los vecinos cercanos de $B[i,j]$ no se pueden ser calculados hasta que el cómputo de $B[i,j]$ esté completo y el error $A[i,j] - B[i,j]$ haya sido distribuido a los cuatro vecinos más próximos de $A[i,j]$. Para comprender por qué se crean fantasmas, imagine una zona oscura situada sobre un área brillante (por ejemplo, un cielo oscuro sobre un mar brillante). Cuando el algoritmo trabaja sobre los píxeles oscuros pasados A, se distribuyen multitud de errores de abajo, para los primeros píxeles brillantes A (y también hacia la derecha). Cuando el algoritmo llega a los primeros píxeles brillantes, éstos han recogido tantos errores previos que ya no pueden ser brillantes, creando quizás varias filas de píxeles B oscuros. Experimentalmente se ha encontrado que los fantasmas pueden ser "exorcizados" escalando los píxeles A antes de que el algoritmo comience. Por ejemplo, cada píxel $A[i,j]$ puede ser sustituido por $0{,}1 + 0{,}8A[i,j]$, que "suaviza" las diferencias en el brillo (el contraste) entre los píxeles claros y oscuros, con lo que se reducen los fantasmas. Esta solución también cambia todas las intensidades de los píxeles, pero el ojo es menos sensible a las intensidades absolutas que a cambios de contraste, por lo que cambiar intensidades puede ser aceptable en muchas situaciones prácticas.

H.12.4. Difusión de punto

Esta sección se basa en [Knuth 87], un artículo muy detallado que incluye análisis minuciosos e imágenes reales difuminadas utilizando diversos métodos diferentes. El algoritmo de difusión de punto es algo similar al difuminado de difusión; también produce imágenes binivel nítidas de buena calidad, pero no es de naturaleza serie y puede ser más fácil de implementar en un computador paralelo.

Comenzamos con la *matriz de clase* de 8×8 de la Figura H.23a. La forma en que esta matriz fue construida se discutirá más adelante. Por ahora, simplemente consideramos una permutación de los enteros $(0, 1, \ldots, 63)$, que llamaremos *clases*. El número de clase k de un píxel $A[i,j]$ se encuentra en la posición (i,j) de la matriz de clase. El algoritmo principal se muestra en la Figura H.24.

El algoritmo calcula primero todos los píxeles de la clase 0, luego los de la clase 1, y así sucesivamente. El procedimiento `Distribuir` es llamado para cada clase k y difunde el error *err* a los vecinos cercanos de $A[i,j]$ cuyos números de clase excedan k. El algoritmo distingue entre los cuatro vecinos ortogonales y los cuatro vecinos diagonales de $A[i,j]$. Si un vecino es $A[u,v]$, entonces el primer tipo satisface $(u-i)^2 + (v-j)^2 = 1$, mientras que el último tipo se identifica por $(u-i)^2 + (v-j)^2 = 2$. Es razonable distribuir más error a los vecinos ortogonales que a los diagonales; en consecuencia, una posible función de peso es $\text{peso}(x,y) = 3 - x^2 - y^2$. Para un vecino ortogonal, o bien $(u-i)$, o bien $(v-j)$ es igual a 1, por tanto $\text{peso}(u-i, v-j) = 2$, mientras que para un vecino diagonal, tanto $(u-i)$ como $(v-j)$ son iguales a 1, por lo que $\text{peso}(u-i, v-j) = 1$. El Procedimiento `Distribuir` aparece en pseudocódigo de la Figura H.24.

Una vez que se conocen las coordenadas (i,j) de un píxel $A[i,j]$, la matriz de clase proporciona el número k de clase del píxel que es independiente del color (o brillo) del píxel. La matriz de clase también proporciona las clases de los ocho vecinos cercanos a $A[i,j]$, por lo que los vecinos cuyas

```
for k := 0 to 63 do
  for all (i,j) of clase k do
    begin
    if A[i,j] <.5 then B[i,j] := 0 else B[i,j] := 1;
    err:=A[i,j] - B[i,j];
    Distribuir(err,i,j,k);
    end.

procedure Distribuir(err,i,j,k);
  ω:= 0;
  for all vecinos A[u,v] of A[i,j] do
    if clase(u,v)> k then ω:=ω+peso(u-i,v-j);
  if ω > 0 then for all vecinos A[u,v] of A[i,j] do
    if clase(u,v)> k then A[u,v]:=A[u,v]+err×peso(u - i, v - j) /ω;
end;
```

Figura H.24: El algoritmo de difusión de punto.

clases exceden k pueden seleccionarse y enlazarse en una lista. Es una buena idea construir esas listas una vez para todos, ya que ésto acelera el algoritmo considerablemente.

Queda por mostrar cómo fue construida la matriz de clase, la Figura H.23a. La consideración principal es el posicionamiento relativo de las clases pequeñas y grandes. Imagine una clase grande rodeada, en la matriz de clases, por clases más pequeñas. Un ejemplo es la clase 63, que está rodeada por las "clases inferiores" 43, 59, 57, 51, 60, 39, 47, y 55. Pensando un poco se ve que a medida que el algoritmo itera hacia 63, se absorben cada vez más errores en los píxeles que pertenecen a esta clase, independientemente de su brillo. Una clase grande rodeada por clases inferiores es, por lo tanto, indeseable y puede ser llamada "baron". La matriz de clase de la Figura H.23a tiene sólo dos barones. Del mismo modo, las posiciones "nearbaron", que tienen sólo un vecino de clase superior, no son deseables y deben ser evitadas. Nuestra matriz clase tiene sólo dos de ellas.

◇ **Ejercicio H.13 (sol. en pág. 1116):** ¿Cuáles son los barones y near-barones de nuestra matriz de clase?

◇ **Ejercicio H.14 (sol. en pág. 1116):** Considérese una imagen completamente gris donde $A[i,j] = 0{,}5$ para todos los píxeles. ¿Qué imagen B se generaría mediante difusión de punto en este caso?

Otra consideración importante son las posiciones de clases consecutivas en la clase matriz. La Figura H.23b,c,d muestra la matriz de clase después de que 11, 21, y 32 de sus clases más bajas se hayan ennegrecido. Es fácil ver cómo las áreas negras forman rejillas de 45° que crecen y eventualmente forman un tablero de ajedrez de 2×2. Ésto ayuda a crear una diagonal, en lugar de los patrones de punto rectilíneos en el array binivel B, y sabemos por experiencia de que dichos patrones son menos perceptibles para el ojo. La Figura H.25a muestra una matriz de clase con sólo un barón y un near-baron, pero es fácil ver cómo las clases más bajas se concentran principalmente en la esquina inferior izquierda de la matriz.

Un examen detallado de la matriz de clase muestra que el número de clases en las posiciones (i,j) e $(i,j+4)$ siempre suman 63. Esto significa que el patrón de cuadrícula de $63-k$ píxeles blancos después de k pasos es idéntico al patrón de rejilla de $63-k$ píxeles negros después de $63-k$ pasos, desplazado hacia la derecha cuatro posiciones. Esta relación entre el patrón de punto y el patrón de difusión es la razón para el nombre *difusión de punto*.

◇ **Ejercicio H.15 (sol. en pág. 1116):** La Figura H.25b muestra una matriz de clase de 4×4. Identifíquense los barones, near-barones y los patrones de rejilla.

25	21	13	39	47	57	53	45
48	32	29	43	55	63	61	56
40	30	35	51	59	62	60	52
36	14	22	26	46	54	58	44
16	6	10	18	38	42	50	24
8	0	2	7	15	31	34	20
4	1	3	11	23	33	28	12
17	9	5	19	27	49	41	37

(a)

14	13	1	2
4	6	11	9
0	3	15	12
10	8	5	7

(b)

Figura H.25: Dos matrices de clase para la difusión de punto.

Experimentos con los cuatro métodos descritos en esta sección parecen indicar que el método de difusión de punto produce los mejores resultados para la impresión, ya que tiende para generar áreas contiguas de píxeles negros, en lugar de áreas en "tablero de ajedrez" que alternan en blanco y negro. Las impresoras láser y de chorro de tinta modernas tienen resoluciones de 600 dpi o más, pero generalmente no pueden producir un tablero de ajedrez de alta calidad de 300 píxeles negros y 300 píxeles blancos alternos por pulgada.

H.12.5. Bibliografía relacionada

Floyd, R., and L. Steinberg (1975) "An Adaptive Algorithm for Spatial Gray Scale," en *Society for Information Display 1975 Symposium Digest of Technical Papers*, p. 36.

Jarvis, J. F., C. N. Judice, and W. H. Ninke (1976) "A Survey of Techniques for the Image Display of Continuous Tone Pictures on Bilevel Displays" *Computer Graphics and Image Processing* **5**(1):13–40.

Jarvis, J. F. and C. S. Roberts (1976) "A New Technique for Displaying Continuous Tone Images on a Bilevel Display" *IEEE Transactions on Communications* **24**(8):891–898, August.

Knuth, Donald E., (1987) "Digital Halftones by Dot Diffusion," *ACM Transactions on Graphics* **6**(4):245–273.

Roetling, P. G. (1976) "Halftone Method with Edge Enhancement and Moir´e Suppression," *Journal of the Optical Society of America*, **66**:985–989.

Roetling, P. G. (1977) "Binary Approximation of Continuous Tone Images," *Photography Science and Engineering*, **21**:60–65.

Mientras se preguntaba que debía hacer en esta emergencia se encontró con una chica sentada al borde del camino. Llevaba un traje que maravilló al chico al ser tan sorprendentemente brillante: su cintura de seda era de color verde esmeralda y su falda de cuatro colores diferentes —azul al frente, amarillo en el lado izquierdo, rojo en la espalda y púrpura en el lado derecho—. Sujetando la cintura en el frente había cuatro botones —el de arriba azul, el siguiente amarillo, un tercero rojo y el último morado—.

L. Frank Baum, *La maravillosa tierra de Oz*

Apéndice I

Matemáticas introductorias

> No se preocupe demasiado acerca de sus dificultades en matemáticas, le puedo asegurar que las mías son todavía mayores.
>
> —Albert Einstein.

I.1. Sumas útiles

1. La suma de una serie geométrica es:

$$\sum_{i=0}^{n} a^i = \begin{cases} 0, & \text{si } a = 0, \\ n+1, & \text{si } a = 1, \\ \frac{1-a^{n+1}}{1-a}, & \text{en otro caso.} \end{cases} \tag{I.1}$$

   Un sencillo corolario es:

$$\sum_{i=0}^{\infty} a^i = \frac{a}{1-a} \text{ para } |a| < 1. \tag{I.2}$$

   Diferenciando la Ecuación (I.2) se obtiene:

$$\sum_{i=0}^{\infty} i a^i = \frac{a}{(1-a)^2} \; para \; |a| < 1.$$

2. Teorema del binomio:

$$(a+b)^n = \sum_{i=0}^{n} \left(\begin{array}{c} n \\ i \end{array} \right) a^i b^{n-i}.$$

3. Serie de desarrollo de una exponencial (serie de Taylor):

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}.$$

4. Suma de los n primeros enteros:

$$\sum_{i=1}^{n} i = \frac{n\,(n+1)}{2}.$$

5. Suma de los n primeros cuadrados:

$$\sum_{i=1}^{n} i^2 = \frac{n\,(n+1)\,(2n+1)}{6}.$$

6. Suma de potencias de 2:

$$\sum_{i=0}^{n} 2^i = 2^0 + 2^1 + 2^2 + \cdots + 2^n = 2^{n+1} - 1.$$

I.2. Matrices

Una **T** matriz es un array (o arreglo) rectangular de números, donde cada elemento a_{ij} se identifica mediante su fila y su columna. La matriz $\mathbf{T}_1$ de abajo es "genérica", con m filas y n columnas. Observe cómo los elementos a_{ii} constituyen la diagonal principal de la matriz. La matriz $\mathbf{T}_2$ es diagonal ($a_{ij} = 0$ para $i \neq j$), la matriz $\mathbf{T}_3$, es simétrica ($a_{ij} = a_{ji}$), y $\mathbf{T}_4$ es una matriz identidad.

$$\mathbf{T}_1 = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{T}_2 = \begin{pmatrix} a_{11} & 0 & 0 & 0 \\ 0 & a_{22} & 0 & 0 \\ 0 & 0 & a_{33} & 0 \\ 0 & 0 & 0 & a_{44} \end{pmatrix},$$

$$\mathbf{T}_3 = \begin{pmatrix} 33 & -17 & 201 & -5 \\ -17 & 66 & 26 & -68 \\ 201 & 26 & 21 & -9 \\ -5 & -68 & -9 & 0 \end{pmatrix}, \quad \mathbf{T}_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

La transpuesta de la matriz **A** (denotado por $\mathbf{A}^T$) se obtiene a partir de **A** reflejando todos los elementos con respecto a la diagonal principal. Una matriz simétrica es igual a su transpuesta.

> Todos los problemas con gráficos por ordenador pueden ser resueltos con la inversión de una matriz.
>
> James F. Blinn, 1993.

I.2.1. Operaciones con matrices

La regla para la suma/resta de matrices es $c_{ij} = a_{ij} \pm b_{ij}$, donde $\mathbf{C} = \mathbf{A} \pm \mathbf{B}$. La regla para la multiplicación de matrices es un poco más compleja: $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$. Cada elemento de **C** es el *producto punto* (o *escalar*) de una fila de **A** y una columna de **B**. En el producto punto, se multiplican los elementos correspondientes de **A** y **B**, y se suman los productos. Para que la multiplicación esté bien definida, cada fila de **A** debe tener el mismo tamaño que una columna de **B**. Las matrices **A** y **B** pueden, por lo tanto, ser multiplicadas sólo si el número de columnas de **A** es igual al número de filas de **B**. Nótese que la matriz multiplicación no es conmutativa, i.e., en general $\mathbf{AB} \neq \mathbf{BA}$.

Un ejemplo de multiplicación de matrices es el producto de las matrices de 1×3 y 3×1

$$(1, -1, 5) \begin{pmatrix} 4 \\ -2 \\ 3 \end{pmatrix},$$

que produce[1] la matriz de 1×1, (21).

◇ **Ejercicio H.13 (sol. en pág. 1116):** ¿Cuáles son los barones y near-barones de nuestra matriz de clase?

.

Productos tensoriales. Este es un caso especial de la multiplicación de matrices. Si $\mathbf{A}$ es un vector columna y $\mathbf{B}$ es un vector fila (cada uno con n elementos), entonces su producto tensorial $\mathbf{C}$ se define como $\mathbf{C}_{ij} = \mathbf{A}_i \mathbf{B}_j$. Ejemplo:

$$\begin{pmatrix} 4 \\ -2 \\ 3 \end{pmatrix} (1, -1, 5) = \begin{pmatrix} 4 & -4 & 20 \\ -2 & 2 & -10 \\ 3 & -3 & 15 \end{pmatrix}.$$

Una matriz cuadrada tiene un determinante, denotado "det $\mathbf{A}$" o $|\mathbf{A}|$, que es un número. El determinante de 2×2 de la matriz $\left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right)$ se define como $ad - bc$. El determinante de una matriz más grande puede ser calculado mediante la regla (nótese la alternancia de signos):

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = a_{11} \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} - a_{12} \begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} + a_{13} \begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix}.$$

La división de matrices no está definida, pero ciertas matrices tienen *inversa*. La inversa de $\mathbf{A}$ se denota $\mathbf{A}^{-1}$, y tiene la propiedad $\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$, donde $\mathbf{I}$ es la *matriz identidad* (con unos en la diagonal y ceros en el resto). La inversa de una matriz se utiliza, e.g., para resolver sistemas de ecuaciones algebraicas lineales. Tales sistemas, pueden ser denotados $\mathbf{A}\mathbf{x} = \mathbf{b}$, donde $\mathbf{A}$ es la matriz de coeficientes, $\mathbf{x}$ es la columna de incógnitas, y $\mathbf{b}$ es la columna de los coeficientes del lado derecho. La solución es $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$.

Ejemplo: El siguiente sistema de tres ecuaciones con tres incógnitas x, y, y z

$$\begin{aligned} x - y &= 1, \\ -x + y &= 2, \\ 25x + 2y + z &= 3, \end{aligned} \tag{I.3}$$

puede ser escrito:

$$\begin{pmatrix} 1 & -1 & 0 \\ -1 & 1 & 0 \\ 25 & 2 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}.$$

La inversa de la matriz de transformación de 3×3 (utilizada en la Sección 4.35.1)

$$T = \begin{pmatrix} a & b & 0 \\ c & d & 0 \\ m & n & 1 \end{pmatrix} \text{ es } T^{-1} = \frac{1}{ad - bc} \begin{pmatrix} d & -b & 0 \\ -c & a & 0 \\ cn - dm & bm - an & ad - bc \end{pmatrix}. \tag{I.4}$$

[1] Expícitamente, $(1 \cdot 4 + (-1)(-2) + 5 \cdot 3) = (4 + 2 + 15) = (21)$.

En general, sin embargo, el cálculo de la inversa no es trivial[2] y puede encontrarse en todos los textos de álgebra lineal, y también en [Press et al. 88]. La página 227 tiene un interesante ejemplo de la inversa de una matriz.

He aquí un resumen de las propiedades de las operaciones con matrices:

$$\begin{gathered}
\mathbf{A}+\mathbf{B}=\mathbf{B}+\mathbf{A}, \quad \mathbf{A}+(\mathbf{B}+\mathbf{C})=(\mathbf{A}+\mathbf{B})+\mathbf{C}, \\
k(\mathbf{A}+\mathbf{B})=k\mathbf{A}+k\mathbf{B}, \quad (k+m)\mathbf{A}=k\mathbf{A}+m\mathbf{A}, \quad k(m\mathbf{A})=(km)\mathbf{A}=m(k\mathbf{A}), \\
\mathbf{A}(\mathbf{BC})=(\mathbf{AB})\mathbf{C}, \quad \mathbf{A}(\mathbf{B}+\mathbf{C})=\mathbf{AB}+\mathbf{AC}, \\
(\mathbf{A}+\mathbf{B})\mathbf{C}=\mathbf{AB}+\mathbf{AC}, \quad \mathbf{A}(k\mathbf{B})=k(\mathbf{AB})=(k\mathbf{A})\mathbf{B}, \\
(\mathbf{A}+\mathbf{B})^T=\mathbf{A}^T+\mathbf{B}^T, \quad (k\mathbf{A})^T=k^T\mathbf{A}^T, \quad (\mathbf{AB})^T=\mathbf{B}^T\mathbf{A}^T.
\end{gathered}$$

Puede encontrarse información histórica de matrices y determinantes en la URL `http://www-groups.dcs.st-and.ac.uk/~history/HistTopics/`, archivo `Matrices_and_determinants.html`.

◇ **Ejercicio I.1 (sol. en pág. 1116):** Súmense, réstense y multiplíquense las dos matrices:

$$\mathbf{A}=\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}, \quad \mathbf{B}=\begin{pmatrix} 7 & 8 & 9 \\ 4 & 5 & 6 \\ 1 & 2 & 3 \end{pmatrix}.$$

◇ **Ejercicio I.2 (sol. en pág. 1116):** Calcúlese la inversa de:

$$\mathbf{T}=\begin{pmatrix} 1 & -1 & 0 \\ -1 & 1 & 0 \\ 25 & 2 & 1 \end{pmatrix}.$$

Una matriz es *ortogonal* si el producto punto (escalar) de dos filas diferentes es cero (y lo mismo para las columnas). Una matriz es *ortonormal* si es ortogonal y el producto punto de una fila con sí misma es uno (y lo mismo para las columnas). Imagine una matriz cuadrada $\mathbf{A}$. Cuando se transpone, sus filas y columnas cambian los roles. Un elemento general (i,j) del producto $\mathbf{AA}^T$ es, pues, el producto punto de la fila i de $\mathbf{A}$ y fila j de la misma $\mathbf{A}$. Por lo tanto, si $\mathbf{A}$ es ortonormal, entonces $\mathbf{AA}^T$ es la matriz identidad $\mathbf{I}$. Sin embargo, el producto $\mathbf{BB}^{-1}$ para cualquier matriz $\mathbf{B}$ es $\mathbf{I}$ (si $\mathbf{B}$ tiene una inversa), por lo que podemos concluir que la transpuesta $\mathbf{A}^T$ de una matriz ortonormal $\mathbf{A}$ es igual a su inversa $\mathbf{A}^{-1}$. Lo contrario también es cierto. Si $\mathbf{A}^T=\mathbf{A}^{-1}$ para alguna matriz $\mathbf{A}$, entonces $\mathbf{A}$ es ortonormal. Puede demostrarse que una matriz ortonormal es siempre una matriz de rotación, y que cualquier matriz de rotación [Ecuación (4.48)] es ortonormal.

Los valores propios y vectores propios (de la palabra alemana para "propio") son cantidades matemáticas útiles asociadas con matrices. Se definen como sigue: Si $\mathbf{A}$ es una matriz de $n\times n$ y si existen vectores $\mathbf{x}$ y escalares λ tales que $\mathbf{Ax}=\lambda\mathbf{x}$ [ó $(\mathbf{A}-\lambda\mathbf{I})\,\mathbf{x}=0$, entonces λ se llama un *valor propio* de $\mathbf{A}$, y $\mathbf{x}$ es el *vector propio* asociado con λ. Los vectores propios de una matriz simétrica son ortogonales.

[2] **Recordatorio:** Dada una matriz $\mathbf{A}$, su inversa es $\mathbf{A}^{-1}=\frac{1}{|\mathbf{A}|}(\mathbf{A}^*)^T$, siendo $\mathbf{A}^*$ la matriz adjunta, resultante de sustituir cada elemento de la matriz por su adjunto (El adjunto del elemento a_{ij} es el menor de su complementario, por ejemplo, si la matriz $\mathbf{A}=\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$, el adjunto de a_{21} es $A_{21}=\begin{vmatrix} a_{12} & a_{13} \\ a_{32} & a_{33} \end{vmatrix}$); $|\mathbf{A}|$ es el determinante de $\mathbf{A}$; y $(\mathbf{A}^*)^T$ es la transpuesta de la matriz adjunta, esto es, $(\mathbf{A}^*)^T=\begin{pmatrix} A_{11} & A_{21} & A_{31} \\ A_{12} & A_{22} & A_{32} \\ A_{13} & A_{23} & A_{33} \end{pmatrix}$.

En principio, el cálculo de los valores propios de una matriz de $n \times n$ consiste en la resolución de una ecuación polinómica de n-ésimo grado. Por lo tanto, para $n \geq 5$, los resultados no pueden en general expresarse únicamente en términos de radicales explícitos. Incluso para la sencilla matriz

$$\begin{pmatrix} a & b \\ -b & 2a \end{pmatrix},$$

los valores propios tienen las dos complicadas expresiones:

$$\frac{1}{2}\left(3a - \sqrt{a^2 - 4b^2}\right), \text{ y } \frac{1}{2}\left(3a + \sqrt{a^2 - 4b^2}\right).$$

Ésta es la razón por la que el software matemático es utilizado en la práctica para obtener valores aproximados (reales y complejos) de los valores propios y vectores propios de una matriz dada.

Los valores propios y vectores propios se mencionan en las Secciones 4.5.4, 4.6.4 y 4.6.9.

Bibliografía

Press, W. H., B. P. Flannery, et al. (1988) *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press. (También disponible online desde `http://www.nr.com/`.)

> Nosotros [Kaplansky y Halmos] compartimos una filosofía sobre el álgebra lineal: pensamos bases libres, escribimos bases libres, pero cuando la suerte está echada cerramos la puerta de la oficina y calculamos mediante matrices con frenesí.
>
> —Irving Kaplansky.

I.3. Identidades trigonométricas

Muchas de las identidades mostradas aquí pueden deducirse con la ayuda del teorema de DeMoivre [Ecuación (Sol.2), pág. 1117].

Identidades básicas

$$\tan\alpha = \frac{\sin\alpha}{\cos\alpha}, \quad \cot\alpha = \frac{\cos\alpha}{\sin\alpha} = \frac{1}{\tan\alpha}, \quad \csc\alpha = \frac{1}{\sin\alpha}, \quad \sec\alpha = \frac{1}{\cos\alpha}.$$

$$\sin(-\alpha) = -\sin\alpha, \quad \cos(-\alpha) = \cos\alpha, \quad \tan(-\alpha) = -\tan\alpha.$$

$$\sin^2\alpha + \cos^2\alpha = 1, \quad \tan^2\alpha + 1 = \sec^2\alpha, \quad \cot^2\alpha + 1 = \csc^2\alpha.$$

Identidades suma y diferencia

$$\cos(\alpha \pm B) = \cos\alpha\cos\beta \mp \sin\alpha\sin\beta, \quad \sin(\alpha \pm \beta) = \sin\alpha\cos\beta \pm \cos\alpha\sin\beta,$$

$$\tan(\alpha \pm \beta) = \frac{\tan\alpha \pm \tan\beta}{1 \mp \tan\alpha\tan\beta}.$$

Identidades de cofunción (ángulos complementarios)

$$\sin\left(\pi/2-\alpha\right)=\cos\alpha,\quad \cos\left(\pi/2-\alpha\right)=\sin\alpha,\quad \tan\left(\pi/2-\alpha\right)=\cot\alpha.$$

Identidades de ángulos dobles y ángulos mitad

$$\cos 2\alpha=\cos^2\alpha-\sin^2\alpha=1-2\sin^2\alpha=2\cos^2\alpha-1,\quad \sin 2\alpha=2\sin\alpha\cos\alpha,$$

$$\tan 2\alpha=\frac{2\tan\alpha}{1-\tan^2\alpha}.$$

$$\cos\left(\alpha/2\right)=\pm\sqrt{\left(1+\cos\alpha\right)/2},\quad \sin\left(\alpha/2\right)=\pm\sqrt{\left(1-\cos\alpha\right)/2}.$$

$$\tan\left(\alpha/2\right)=\pm\sqrt{\frac{1-\cos\alpha}{1+\cos\alpha}}=\frac{\sin\alpha}{1+\cos\alpha}=\frac{1-\cos\alpha}{\sin\alpha}.$$

Identidades Suma y producto

$$\sin\alpha+\sin\beta=2\sin\left(\frac{\alpha+\beta}{2}\right)\cos\left(\frac{\alpha-\beta}{2}\right),$$

$$\sin\alpha-\sin\beta=2\cos\left(\frac{\alpha+\beta}{2}\right)\sin\left(\frac{\alpha-\beta}{2}\right).$$

$$\cos\alpha+\cos\beta=2\cos\left(\frac{\alpha+\beta}{2}\right)\cos\left(\frac{\alpha-\beta}{2}\right),$$

$$\cos\alpha-\cos\beta=-2\sin\left(\frac{\alpha+\beta}{2}\right)\sin\left(\frac{\alpha-\beta}{2}\right).$$

$$\sin\alpha\cos\beta=\frac{1}{2}\left[\sin\left(\alpha+\beta\right)+\sin\left(\alpha-\beta\right)\right],$$

$$\cos\alpha\sin\beta=\frac{1}{2}\left[\sin\left(\alpha+\beta\right)-\sin\left(\alpha-\beta\right)\right].$$

$$\cos\alpha\cos\beta=\frac{1}{2}\left[\cos\left(\alpha+\beta\right)+\cos\left(\alpha-\beta\right)\right],$$

$$\sin\alpha\sin\beta=-\frac{1}{2}\left[\cos\left(\alpha+\beta\right)-\cos\left(\alpha-\beta\right)\right].$$

Observe que las dos últimas líneas también implican:

$$\cos^2\alpha=\frac{1}{2}\left(\cos\left(2\alpha\right)+1\right),\quad \sin^2\alpha=\frac{1}{2}\left(1-\cos\left(2\alpha\right)\right).$$

Leyes de senos y cosenos

Cualquier triángulo con lados a,b,c y ángulos α,β,γ satisface la ley de los senos:

$$\frac{a}{\sin\alpha}=\frac{b}{\sin\beta}=\frac{c}{\sin\gamma}$$

y la ley de los cosenos:

$$a^2 = b^2 + c^2 - 2bc\cos\alpha, \quad b^2 = a^2 + c^2 - 2ac\cos\beta, \quad c^2 = a^2 + b^2 - 2ab\cos\gamma.$$

¡Las matemáticas son el único lenguaje universal que existe, senador!

—Jodie Foster (como Ellie Arroway) en *Contacto* (1977).

I.4. Álgebra vectorial

Un vector es una entidad matemática con dos atributos, dirección y magnitud (nótese que un vector no tiene una posición espacial). La magnitud del vector $\mathbf{P} = (x, y, z)$ (también llamada su *valor absoluto*) es $|\mathbf{P}| = \sqrt{x^2 + y^2 + z^2}$. La dirección de un vector puede ser expresada por los cosenos de los ángulos que forman con los ejes de coordenadas: $x/|P|$, $y/|P|$, y $z/|P|$. Obsérvese que el vector $(x/|P|, y/|P|, z/|P|)$ tiene una magnitud de 1 (es un *vector unitario*).

Los tres vectores unitarios en las direcciones de los ejes de coordenadas son tradicionalmente denotados $\mathbf{i} = (1, 0, 0)$, $\mathbf{j} = (0, 1, 0)$, y $\mathbf{k} = (0, 0, 1)$.

I.4.1. Operaciones con vectores

La suma vectorial se define como la suma de los elementos individuales de los vectores que están siendo sumados. En consecuencia, $\mathbf{P}+\mathbf{Q} = (P_x, P_y, P_z)+(Q_x, Q_y, Q_z) = (P_x + Q_x, P_y + Q_y, P_z + Q_z)$. Esta operación es tanto conmutativa ($\mathbf{P} + \mathbf{Q} = \mathbf{Q} + \mathbf{P}$) como asociativa $\mathbf{P} + (\mathbf{Q} + \mathbf{T}) = (\mathbf{P} + \mathbf{Q}) + \mathbf{T}$. La resta vectorial $\mathbf{P} - \mathbf{Q}$ se efectúa de manera similar y produce un vector desde $\mathbf{Q}$ hasta $\mathbf{P}$.

Los vectores pueden ser multiplicados en tres formas diferentes:

1. La *multiplicación de un escalar por un vector* se define mediante $\alpha\mathbf{P} = (\alpha x, \alpha y, \alpha z)$. Ésto cambia la magnitud del vector (por un factor α), pero no su dirección. Esta operación es distributiva con respecto a la suma o la resta de vectores, $\alpha(\mathbf{P} \pm \mathbf{Q}) = \alpha\mathbf{P} \pm \alpha\mathbf{Q}$.

2. El *producto escalar* (*interior* o *punto*) *de dos vectores* se denota por $\mathbf{P} \bullet \mathbf{Q}$ y se define como el escalar:
$$(P_x, P_y, P_z)(Q_x, Q_y, Q_z)^T = \mathbf{P}\mathbf{Q}^T = P_xQ_x + P_yQ_y + P_zQ_z.$$
Esto también es igual a $|\mathbf{P}||\mathbf{Q}|\cos\theta$, donde θ es el ángulo entre los vectores. El producto escalar de vectores perpendiculares (también llamados *vectores ortogonales*) es cero. El producto escalar es conmutativo, $\mathbf{P} \bullet \mathbf{Q} = \mathbf{Q} \bullet \mathbf{P}$ y también es distributiva con respecto a la suma o la resta de vectores, $\mathbf{P} \bullet (\mathbf{Q} \pm \mathbf{T}) = \mathbf{P} \bullet \mathbf{Q} \pm \mathbf{P} \bullet \mathbf{T}$.

   En ocasiones, el producto triple $(\mathbf{P} \bullet \mathbf{Q})\mathbf{R}$ es útil. Puede representarse como:

$$\begin{aligned}
&(\mathbf{P} \bullet \mathbf{Q})\mathbf{R} \\
&= (P_xQ_x + P_yQ_y + P_zQ_z)(R_x, R_y, R_z) \\
&= ((P_xQ_x + P_yQ_y + P_zQ_z)R_x, (P_xQ_x + P_yQ_y + P_zQ_z)R_y, (P_xQ_x + P_yQ_y + P_zQ_z)R_z) \\
&= (Q_x, Q_y, Q_z)\begin{pmatrix} P_xR_x & P_xR_y & P_xR_z \\ P_yR_x & P_yR_y & P_yR_z \\ P_zR_x & P_zR_y & P_zR_z \end{pmatrix} \\
&= \mathbf{Q}(\mathbf{P}\mathbf{R}),
\end{aligned} \tag{I.5}$$

   donde la notación $(\mathbf{P}\mathbf{R})$ se refiere a la matriz de 3×3 de arriba.

3. El *producto cruz* de dos vectores (también llamado *producto vectorial*) se denota $\mathbf{P} \times \mathbf{Q}$ y se define como el vector:

$$(P_2Q_3 - P_3Q_2, -P_1Q_3 + P_3Q_1, P_1Q_2 - P_2Q_1)\,. \tag{I.6}$$

Es fácil demostrar que $\mathbf{P} \times \mathbf{Q}$ es perpendicular tanto a $\mathbf{P}$ como a $\mathbf{Q}$.

◇ **Ejercicio I.3 (sol. en pág. 1117):** ¡Demuéstrese!

Las siguientes expresiones muestran cómo puede expresarse $\mathbf{P} \times \mathbf{Q}$ por medio de un determinante:

$$\begin{aligned}\mathbf{P} \times \mathbf{Q} &= \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ P_1 & P_2 & P_3 \\ Q_1 & Q_2 & Q_3 \end{vmatrix} = \mathbf{i} \begin{vmatrix} P_2 & P_3 \\ Q_2 & Q_3 \end{vmatrix} - \mathbf{j} \begin{vmatrix} P_1 & P_3 \\ Q_1 & Q_3 \end{vmatrix} + \mathbf{k} \begin{vmatrix} P_1 & P_2 \\ Q_1 & Q_2 \end{vmatrix} \\ &= (P_2Q_3 - P_3Q_2, -P_1Q_3 + P_3Q_1, P_1Q_2 - P_2Q_1)\,,\end{aligned}$$

o, de forma alternativa, por medio de una matriz:

$$\mathbf{P} \times \mathbf{Q} = (Q_1, Q_2, Q_3) \begin{pmatrix} 0 & P_3 & -P_2 \\ -P_3 & 0 & P_1 \\ P_2 & -P_1 & 0 \end{pmatrix}. \tag{I.7}$$

◇ **Ejercicio I.4 (sol. en pág. 1117):** El producto vectorial $\mathbf{P} \times \mathbf{Q}$ es perpendicular tanto a $\mathbf{P}$ como a $\mathbf{Q}$. ¿En qué dirección apunta?

El producto vectorial no es conmutativo ni asociativo. Sin embargo, es distributivo con respecto a la suma o resta de vectores. Por lo tanto $\mathbf{P} \times (\mathbf{Q} \pm \mathbf{T}) = \mathbf{P} \times \mathbf{Q} \pm \mathbf{P} \times \mathbf{T}$.

La magnitud $\mathbf{P} \times \mathbf{Q}$ es igual a $|\mathbf{P}|\,|\mathbf{Q}| \sin\theta$, donde θ es el ángulo que forman los dos vectores. El producto vectorial, por lo tanto, tiene una interpretación geométrica simple. Su magnitud es igual al área del paralelogramo definido por los dos vectores.

◇ **Ejercicio I.5 (sol. en pág. 1117):** Dado que $\mathbf{P} \times \mathbf{Q} = 0$, ¿qué nos dice acerca de los vectores implicados?

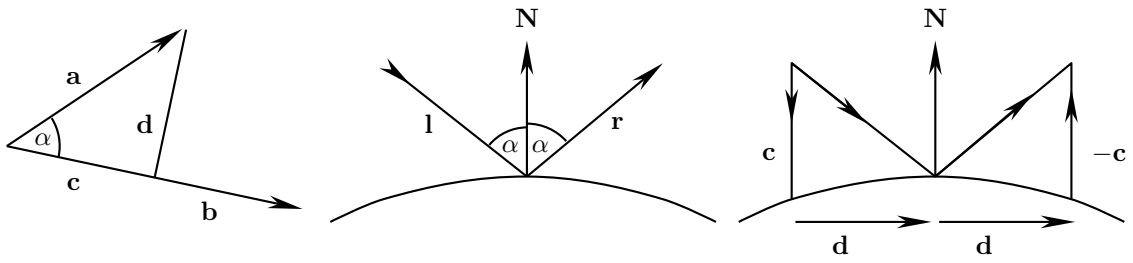
Como ejemplo, a continuación se muestra la ecuación vectorial de una línea recta para el caso en el que se conocen la dirección de la recta y un punto de la misma. Se asume que $\mathbf{d}$ es un vector unitario en la dirección de la recta y $\mathbf{P}_1$ es un punto dado de la recta. La ecuación de la recta entera es:

$$\mathbf{P}(t) = \mathbf{P}_1 + t\mathbf{d}, \quad \text{para cualquier real } t. \tag{I.8}$$

◇ **Ejercicio I.6 (sol. en pág. 1117):** Dedúzcase la ecuación vectorial de la recta para el segmento de recta entre dos puntos dados $\mathbf{P}_1$ y $\mathbf{P}_2$.

What if angry vectors veer
Round your sleeping head, and form.
There's never need to fear
Violence of the poor world's abstract storm.[a]
—Robert Penn Warren, *Lullaby in Encounter*[b], 1957.

[a] Y qué si viran los vectores enojados; Alrededor de su somnolienta cabeza y figura; Nunca debemos temer; La violencia de la tormenta abstracta de los mundos humildes.
[b] *Canción de cuna en un encuentro.*



(a) (b) (c)

Figura I.1: Proyección de un vector.

I.4.2. El triple producto escalar o producto mixto

El producto mixto de tres vectores **P**, **Q**, y **R** se define como:

$$\begin{aligned} S =& \mathbf{P} \bullet (\mathbf{Q} \times \mathbf{R}) = P_1 (Q_2 R_3 - Q_3 R_2) + P_2 (Q_3 R_1 - Q_1 R_3) + P_3 (Q_1 R_2 - Q_2 R_1) \\ =& \begin{vmatrix} P_1 & P_2 & P_3 \\ Q_1 & Q_2 & Q_3 \\ R_1 & R_2 & R_3 \end{vmatrix} . \end{aligned} \tag{I.9}$$

El intercambio de dos filas en un determinante cambia su signo, por lo que intercambiando las filas un número par de veces deja el determinante invariable. Por esta razón el producto mixto no es afectado por una permutación cíclica de sus tres componentes. Por tanto, podemos escribir:

$$S = \mathbf{P} \bullet (\mathbf{Q} \times \mathbf{R}) = \mathbf{Q} \bullet (\mathbf{R} \times \mathbf{P}) = \mathbf{R} \bullet (\mathbf{P} \times \mathbf{Q}) .$$

El triple producto escalar tiene una interpretación geométrica simple. Es igual al volumen del paralelepípedo definido por los tres vectores. Un corolario importante es: Si los tres vectores son coplanarios, entonces el paralelepípedo definido por ellos tiene volumen cero, lo que implica que su producto mixto es cero.

I.4.3. Proyección de un vector

Una operación común y útil en los vectores es la proyección de un vector **a** sobre otro **b**. La idea es fraccionar el vector **a** en dos componentes perpendiculares **c** y **d**, de tal manera que **c** esté en la dirección de **b**.

La Figura I.1a muestra que $\mathbf{a} = \mathbf{c} + \mathbf{d}$ y $|\mathbf{c}| = |\mathbf{a}| \cos\alpha$. Por otra parte $\mathbf{a} \bullet \mathbf{b} = |\mathbf{a}| \, |\mathbf{b}| \cos\alpha$, proporcionando la magnitud de **c**:

$$|\mathbf{c}| = |\mathbf{a}| \, \frac{(\mathbf{a} \bullet \mathbf{b})}{|\mathbf{a}| \, |\mathbf{b}|} = \frac{(\mathbf{a} \bullet \mathbf{b})}{|\mathbf{b}|}. \tag{I.10}$$

La dirección de **c** es idéntica a la dirección de **b**, por lo que podemos escribir el vector **c** como:

$$\mathbf{c} = |\mathbf{c}| \, \frac{\mathbf{b}}{|\mathbf{b}|} = \frac{(\mathbf{a} \bullet \mathbf{b})}{|\mathbf{b}|^2} \mathbf{b}. \tag{I.11}$$

Ejemplo: Dados los vectores $\mathbf{a} = (2, 1)$ y $\mathbf{b} = (1, 0)$, es fácil calcular

$$\mathbf{c} = \frac{(\mathbf{a} \bullet \mathbf{b})}{|\mathbf{b}|^2} \mathbf{b} = \frac{2 \times 1 + 1 \times 0}{1^2 + 0^2} \, (2, 0) = (4, 0) , \qquad \mathbf{d} = \mathbf{a} - \mathbf{c} = (-2, 1) .$$

◇ **Ejercicio I.7 (sol. en pág. 1117):** El método de proyección anterior funciona también para vectores tridimensionales. Dados los vectores $\mathbf{a} = (2, 1, 3)$ y $\mathbf{b} = (1, 0, -1)$, calcúlese la proyección de $\mathbf{a}$ sobre $\mathbf{b}$.

◇ **Ejercicio I.8 (sol. en pág. 1117):** Los vectores y sus operaciones se conocen desde hace mucho tiempo. Explíquese por qué han cobrado importancia en las últimas décadas, desde la introducción de los computadores digitales.

I.5. Números complejos

Los números complejos se expresan en términos de un número especial i que se define como $\sqrt{-1}$ y, por consiguiente, satisface $i \times i = i^2 = -1$. Cualquier número complejo z puede representarse, ya sea como la suma $a + bi$, ya sea como el par (a, b), donde a y b son reales. El *conjugado* de z se denota por z^* y se define como $a - bi$. Los complejos conjugados se corresponden grosso modo con los números reales negativos. La suma de los números reales a y $-a$ es cero y la suma $z + z^*$ es $2a$, que es real. La *magnitud* o *valor absoluto* de un número complejo se denota por $|z|$ y se define como $\sqrt{z \cdot z^*} = \sqrt{a^2 + b^2}$. La suma y la diferencia de los números complejos $a + bi$ y $c + di$ es lo obvio, $(a + b) \pm (c + d)\, i$. El producto utiliza la relación $i^2 = -1$ y es $(a + bi)\,(c + di) = (ac - bd) + (ad + bc)\, i$. El inverso, z^{-1}, de z se define como $z^*/|z|$. Se corresponde con el recíproco $1/a$ de un número real a. La división z_1/z_2 es fácil de realizar para $|z_2| \neq 0$:

$$\frac{z_1}{z_2} = \frac{z_1 z_2^*}{z_2 z_2^*} = \frac{(a + bi)\,(c - di)}{c^2 + d^2} = \left(\frac{ac + bd}{c^2 + d^2}, \frac{bc - ad}{c^2 + d^2} \right).$$

La regla de la multiplicación de números complejos puede interpretarse como una rotación en dos dimensiones. Ésto es fácil de ver si consideramos el producto de los dos números complejos (x, y) y $(\cos\theta, \sin\theta)$.

$$\begin{aligned}(x, y) \cdot (\cos\theta, \sin\theta) &= (x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta) \\ &= (x, y) \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}. \end{aligned} \tag{I.12}$$

Este producto rota el punto bidimensional (x, y) hasta un ángulo θ sobre el origen.

◇ **Ejercicio I.9 (sol. en pág. 1117):** Utilícese la regla de la multiplicación de números complejos para multiplicar el número complejo $(0, 1)$ por sí mismo.

◇ **Ejercicio I.10 (sol. en pág. 1117):** Utilícese la regla de la multiplicación de números complejos para multiplicar los números complejos (a, b) y $(-a, -b)$.

> El camino más corto entre dos verdades en el dominio real pasa a través del dominio complejo.
>
> —Jacques Hadamard.

Los números complejos pueden representarse gráficamente como puntos bidimensionales donde la parte real es la coordenada x y la parte imaginaria es la coordenada y. Tal representación se llama *diagrama de Argand*.

Generalmente utilizamos las *coordenadas cartesianas* (a, b) de un punto $\mathbf{P}$. Las *coordenadas polares* de $\mathbf{P}$ son (r, θ), donde $r = \sqrt{a^2 + b^2}$ es la distancia del punto desde el origen y $\theta = \arctan(b/a)$ es

el ángulo entre el eje x y el vector r. Dado un número complejo $z = (x, y)$, r es su valor absoluto y θ se conoce como su *argumento* (`arg` para abreviar). Las coordenadas polares pueden obtenerse a partir de las cartesianas mediante $(a, b) = (r\cos\theta, r\sin\theta)$. Puesto que el número complejo $z = (a, b)$ puede interpretarse como un punto bidimensional, tiene la representación polar $z = (r\cos\theta, r\sin\theta) = r\,(\cos\theta + i\sin\theta)$. Esta representación es útil en muchas aplicaciones.

La canción del número complejo

(Afinación: cuerpo de John Brown)

Mis ojos han visto la gloria del diagrama de Argand
Han visto las i's y thetas del poderoso plan de De Moivre
Ahora puedo encontrar las raíces complejas con completo brío
Con la raíz de menos uno

Los números complejos son tan fáciles
Los números complejos son tan fáciles
Los números complejos son tan fáciles
Con la raíz de menos uno

En coordenadas cartesianas el plano complejo está muy bien,
Pero la grandeza de la forma polar esta belleza llega a eclipsar
Puedes elevar $i + 40$ a la potencia de 99
Con la raíz de menos uno

Te darás cuenta de que su comprensión llevó sólo segundo
Al ver el poder y la magia de la conjugada compleja
dibujando vectores correspondientes a las raíces de menos ocho
Con la raíz de menos uno

(Atribuido a la Sra. P. E. Perella.)

La famosa fórmula de Euler

$$e^{i\theta} = \cos\theta + i\sin\theta$$

nos permite escribir $z = re^{i\theta}$, una representación que hace que resulte fácil multiplicar y dividir números complejos:

$$z_1 z_2 = r_1 r_2 e^{i(\theta_1+\theta_2)}, \quad \frac{z_1}{z_2} = r_1 r_2 e^{i(\theta_1-\theta_2)},$$

e incluso extraer raíces:

$$\sqrt[n]{z} = \sqrt[n]{r}\left[\cos\left(\frac{\theta+2k\pi}{n}\right) + i\sin\left(\frac{\theta+2k\pi}{n}\right)\right], \quad k = 0, 1, \ldots, n-1.$$

Las n raíces de z pueden ser visualizadas como puntos igualmente espaciados situados en la circunferencia de un círculo de radio $\sqrt[n]{r}$ cuyo centro está en el origen. Su conexión produce un polígono regular de n lados.

◇ **Ejercicio I.11 (sol. en pág. 1117):** (Matemáticas.) Sabemos que $i = \sqrt{-1}$. ¿Cómo es $\sqrt{i}$?

◇ **Ejercicio I.12 (sol. en pág. 1118):** Ya que estamos en ello, ¿cómo son i^i y $\ln i$?

Bibliografía

Nahin, Paul J., (1998) *An Imaginary Tale: The Story of* $\sqrt{-1}$, Princeton, NJ, Princeton University Press.

I.6. Convolución

Ésta es una cifra importante que tiene varias aplicaciones prácticas. Se utiliza en las Secciones 5.6.1 y 5.8. Comenzamos con el simple e intuitivo concepto de *sistema*. Ésto es algo que recibe la entrada y genera una salida en respuesta. La entrada y la salida pueden ser unidimensionales (una función del tiempo), bidimensionales (una función de dos variables espaciales), o pueden tener cualquier número de dimensiones. Nos interesaremos en la relación entre la salida y la entrada, no en el funcionamiento interno del sistema. También nos concentraremos en los *sistemas lineales*, ya que son a la vez sencillos e importantes. Un sistema lineal se define como sigue: Si la entrada $x_1(t)$ produce una salida $y_1(t)$ [denotamos ésto $x_1(t) \to y_1(t)$] y si $x_2(t) \to y_2(t)$, entonces $x_1(t) + x_2(t) \to y_1(t) + y_2(t)$. Cualquier sistema que no satisfaga esta condición es considerado no lineal.

Esta definición implica que $2x_1(t) = x_1(t) + x_1(t) \to y_1(t) + y_1(t) = 2y_1(t)$ ó, en general, $ax_1(t) \to ay_1(t)$ para cualquier real a.

Algunos sistemas lineales son de *desplazamiento invariante* (*shift invariant*). Si un sistema lineal tal satisface $x(t) \to y(t)$, entonces $x(t-T) \to y(t-T)$, i.e., el desplazamiento de la entrada en una cantidad T produce un desplazamiento en la salida en la misma cantidad, pero no afectará de otro modo a la salida. En la discusión de convolución, se asume que los sistemas en cuestión son lineales y de desplazamiento invariante. Ésto es cierto (o fiel con una muy buena aproximación) para los sistemas de redes eléctricas y los sistemas ópticos, para las piezas principales de hardware usadas en el procesamiento de imágenes y para la compresión.

> Sistema. Se utiliza frecuentemente sin necesidad.
>
> | | |
> |---|---|
> | Dayton Dayton ha adoptado el sistema de comisiones del gobierno. | Dayton Dayton ha adoptado el gobierno por la comisión. |
> | El sistema de dormitorios | Dormitorios |
>
> —Strunk y White, Los elementos del estilo.

Es útil disponer de una relación general entre la entrada y la salida de un sistema lineal, de desplazamiento invariante. Resulta que la expresión

$$y(t) = \int_{-\infty}^{+\infty} f(t,\tau)\, x(\tau)\, d\tau, \tag{I.13}$$

es lo suficientemente general para este propósito. En otras palabras, siempre existe una función de dos parámetros $f(t,\tau)$ que se puede utilizar para predecir la salida $y(t)$ si la entrada $x(\tau)$ es conocida. Sin embargo, queremos expresar esta relación con una función de un único parámetro, y utilizamos el desplazamiento invariante del sistema para este propósito. Para un sistema de desplazamiento invariante podemos escribir:

$$y(t-T) = \int_{-\infty}^{+\infty} f(t,\tau)\, x(\tau - T)\, d\tau.$$

Si cambiamos las variables sumando T tanto a t como a τ, obtenemos:

$$y(t) = \int_{-\infty}^{+\infty} f(t+T, \tau+T)\, x(\tau)\, d\tau. \tag{I.14}$$

La comparación de las Ecuaciones (I.13) e (I.14) muestra que $f(t,\tau) = f(t+T, \tau+T)$. En consecuencia, la función f tiene la propiedad de que si sumamos T a todos sus parámetros, no la hace cambiar. La función es constante siempre que la diferencia entre sus parámetros sea constante. La función f depende solamente de la diferencia de sus parámetros, por lo que es esencialmente una función de un único parámetro. Por tanto, podemos escribir $g(t-\tau) = f(t,\tau)$, que cambia la Ecuación (I.13) a:

$$y(t) = \int_{-\infty}^{+\infty} g(t-\tau)\, x(\tau)\, d\tau. \tag{I.15}$$

Ésta es la *integral de convolución*, una importante relación entre $x(t)$ e $y(t)$ o entre $x(t)$ y $g(t)$. Esta relación se denota $y = g \star x$ y se dice que la salida de un sistema lineal de desplazamiento invariante viene dado por la convolución de su entrada con una cierta función $g(t)$ (o mediante la *convolución* de x con g). La función g, que es característica del sistema, se denomina *respuesta de impulso* del sistema. La Figura I.2 muestra una descripción gráfica de una convolución, donde el resultado final (la integral) es el área gris bajo la curva.

La convolución tiene un número de propiedades importantes. Es conmutativa, asociativa y distributiva sobre la suma. Estas propiedades se muestran en la Ecuación (I.16)

$$\begin{aligned} f \star g &= g \star f, \\ f \star (g \star h) &= (f \star g) \star h, \\ f \star (g + h) &= f \star g + f \star h. \end{aligned} \tag{I.16}$$

Los problemas prácticos normalmente implican secuencias discretas de números, en lugar de funciones continuas, por lo que la *convolución discreta* es útil. La convolución discreta de las dos secuencias $f(i)$ y $g(i)$ se define como:

$$h(i) = f(i) \star g(i) = \sum_j f(j)\, g(i-j). \tag{I.17}$$

Si las longitudes de $f(i)$ y $g(i)$ son m y n, respectivamente, entonces $h(i)$ tiene una longitud de $m+n-1$.

Ejemplo: Dadas las dos secuencias $f = (f(0), f(1), \ldots, f(5))$ (seis elementos) y $g = (g(0), g(1), \ldots g(4))$

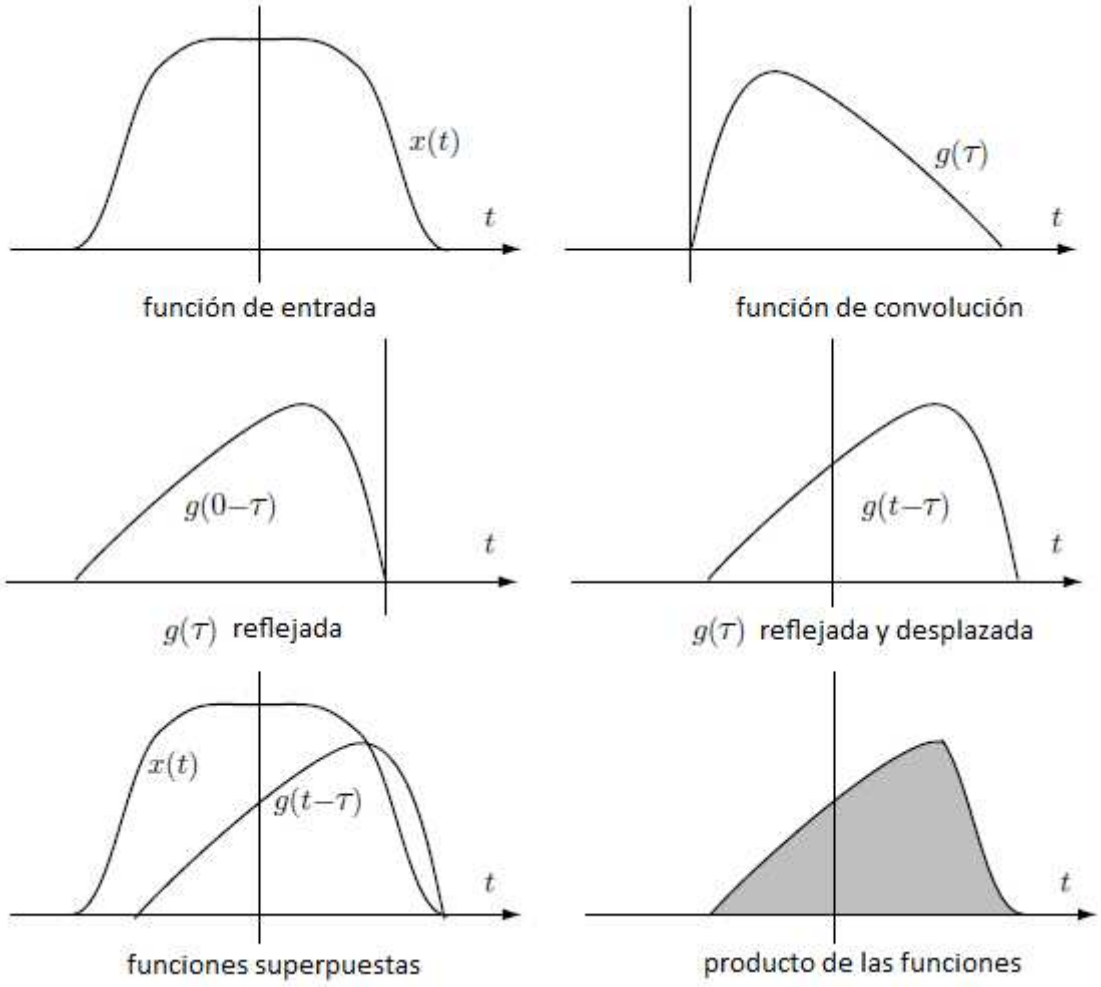


Figura I.2: La convolución de $x\,(t)$ y $g\,(t)$.

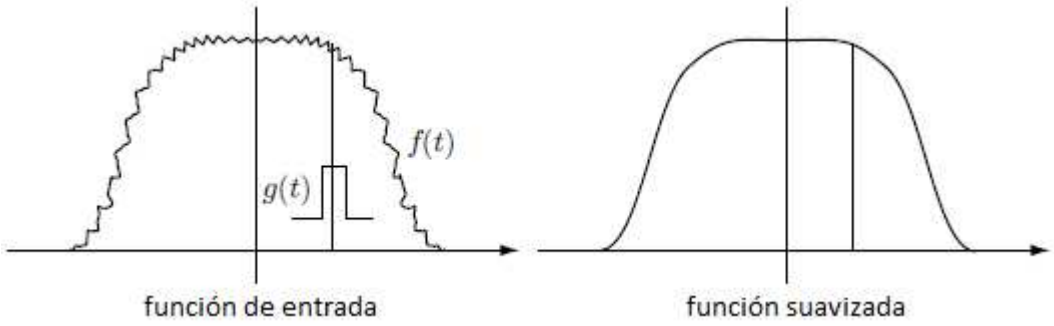


Figura I.3: Aplicación de la convolución a una función de eliminación de ruido.

(cinco elementos), la Ecuación (I.17) produce los diez elementos de convolución $h = f \star g$:

$$h(0) = \sum_{j=0}^{0} f(j)\, g(0-j) = f(0)\, g(0)$$

$$h(1) = \sum_{j=0}^{1} f(j)\, g(1-j) = f(0)\, g(1) + f(1)\, g(0)$$

$$h(2) = \sum_{j=0}^{2} f(j)\, g(2-j) = f(0)\, g(2) + f(1)\, g(1) + f(2)\, g(0)$$

$$h(3) = \sum_{j=0}^{3} f(j)\, g(3-j) = f(0)\, g(3) + f(1)\, g(2) + f(2)\, g(1) + f(3)\, g(0)$$

$$h(4) = \sum_{j=0}^{4} f(j)\, g(4-j) = f(0)\, g(4) + f(1)\, g(3) + f(2)\, g(2) + f(3)\, g(1) + f(4)\, g(0)$$

$$h(5) = \sum_{j=1}^{5} f(j)\, g(5-j) = f(1)\, g(4) + f(2)\, g(3) + f(3)\, g(2) + f(4)\, g(1) + f(5)\, g(0)$$

$$h(6) = \sum_{j=2}^{5} f(j)\, g(6-j) = f(2)\, g(4) + f(3)\, g(3) + f(4)\, g(2) + f(5)\, g(1)$$

$$h(7) = \sum_{j=3}^{5} f(j)\, g(7-j) = f(3)\, g(4) + f(4)\, g(3) + f(5)\, g(2)$$

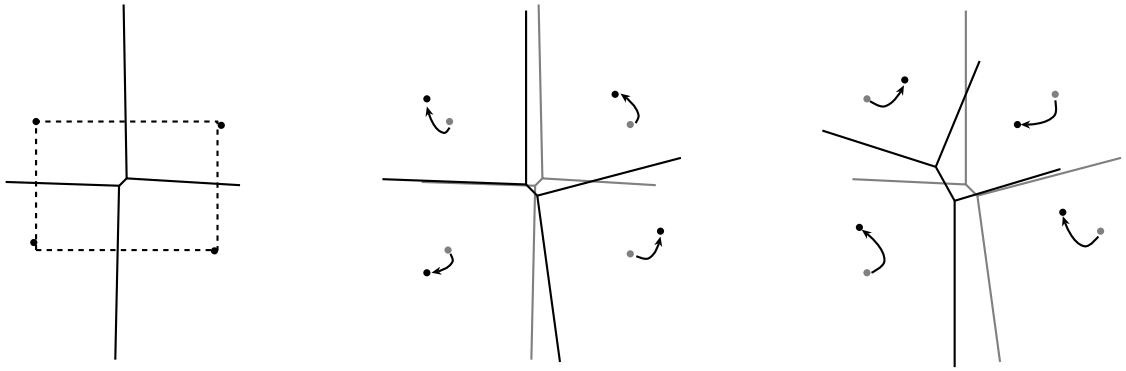
$$h(8) = \sum_{j=4}^{5} f(j)\, g(8-j) = f(4)\, g(4) + f(5)\, g(3)$$

$$h(9) = \sum_{j=5}^{5} f(j)\, g(9-j) = f(5)\, g(4)$$

> "Oh, no", dijo George. "Fue más que el dinero."
>
> Apoyó su frente en su mano y trató de recordar qué era más que el dinero. La oscuridad dentro de su cabeza estaba llena de convoluciones. Sus tímpanos estaban demasiado rígidos. Sólo estaban llegando los registros más altos del sonido.
>
> —Paul Scott, *El doblador*

Un sencillo ejemplo de uso de una convolución es el suavizado (o eliminación de ruido). Éste muestra como puede utilizarse una convolución como un filtro. Dada una función con ruido $f(t)$ (Figura I.3), seleccionamos un pulso rectangular como la función de convolución $g(t)$. Ésta se define como:

$$g(t) = \begin{cases} 1, & -a/2 < t < a/2, \\ \frac{1}{2}, & t = \pm a/2, \\ 0, & \text{en otro caso,} \end{cases}$$



(a) (b) (c)

Figura I.4: Tres diagramas de Voronoi de cuatro puntos.

donde a es un valor adecuadamente pequeño (típicamente 1, pero podría ser cualquier otro). Según se va introduciendo la convolución, el pulso se va moviendo de izquierda a derecha y es multiplicado por $f(t)$. El resultado del producto es una media local de $f(t)$ sobre un intervalo de anchura a. Ésto tiene el efecto de suprimir las fluctuaciones de alta frecuencia de $f(t)$.

Del diccionario

convolution: coiling together[a]
convolve: roll together[b]

[a] bobinado o enrollamiento conjunto
[b] enrollar conjuntamente

I.7. Diagramas de Voronoi

Imagínese una placa de Petri preparada para bacterias de crecimiento. Cuatro bacterias de diferentes tipos son simultáneamente depositadas en ella en puntos diferentes e inmediatamente comienzan a multiplicarse. Suponemos que sus colonias crecen a la misma velocidad. Inicialmente, cada colonia está formada por un círculo creciente alrededor de cada punto de partida. Después de un tiempo algunas de ellas se encuentran y dejan de crecer en el área de reunión debido a la falta de alimentos. El resultado final es que la placa entera se divide en cuatro áreas, una alrededor de cada uno de los cuatro puntos de inicio, de tal manera que todos los puntos dentro de la zona i están más cerca del punto de partida i que de cualquier otro punto de partida. Dichas áreas se denominan *regiones de Voronoi* o *teselas de Dirichlet*. La Figura I.4a muestra las regiones de Voronoi para cuatro puntos situados aproximadamente en las cuatro esquinas de un rectángulo a trazos. Las regiones están cerca de los cuatro cuadrantes del rectángulo. La Figura I.4b,c muestra cómo cambian las regiones cuando los puntos son desplazados.

En el momento de la escritura de esta sección existen en la web varios applets de Java que demuestran los conceptos discutidos aquí. Un ejemplo típico es [Zhao 98].

Bibliografía

Zhao, Zhiyuan (1998) es un applet en `http://ra.cfm.ohio-state.edu/~zhao/algorithms/algorithms.html`. (A fecha de marzo de 2013 esta URL ya no está disponible. Existe otro applet interesante en `http://www.pi6.fernuni-hagen.de/GeomLab/VoroGlide/`.)

I.8. L-sistemas

Los sistemas de Lindenmayer (o L-sistemas para abreviar) fueron desarrollados por el biólogo Aristid Lindenmayer en 1968 como una herramienta [Lindenmayer 68] para describir la morfología de las plantas. Fueron utilizados inicialmente en la informática, en la década de 1970, como una herramienta para definir lenguajes formales, pero se ha hecho realmente popular sólo después de 1984, cuando Alvy Ray Smith señaló [Smith 84] que los L-sistemas pueden ser utilizados para dibujar muchos tipos de los fractales, además de su uso en botánica. Hoy en día los L-sistemas también se utilizan para generar mosaicos, arte geométrico e incluso partituras musicales.

La idea principal de los L-sistemas consiste en definir un objeto complejo (1) determinando un objeto inicial simple, llamado *axioma*, y (2) dando reglas que muestren cómo reemplazar partes del axioma.

> La siguiente historia verdadera es un ejemplo de modestia de Aristid. En una de las conferencias interamericanas alguien le preguntó qué significaba la L de "L-sistemas". La respuesta de Aristid fue "Lenguajes".
>
> —Grzegorz Rozenberg.

Las reglas se aplican sucesivamente, creando piezas que se vuelven más y más complejas, transformando así el sencillo axioma en el complejo objetivo final. Las reglas se denominan reglas de *reescritura* o de *producción*, y son una extensión del trabajo de Chomsky sobre gramáticas formales, y también de la notación BNF. N. Chomsky mostró, en la década de 1950, cómo describir la sintaxis de un lenguaje natural mediante la producción de reglas. Aproximadamente al mismo tiempo, Backus y Naur desarrollaron la notación BNF, que se basa en las reglas de reescritura, específicamente para proporcionar una definición formal [Naur 60] de la sintaxis del ALGOL 60.

La Figura I.5 muestra cómo se construye un fractal, la curva copo de nieve de Koch, en varios pasos, sobre un axioma que es un simple triángulo (I.5a) y una regla de reescritura que dice: Reemplazar cada segmento recto con la curva de I.5B. La Figura I.5c es el resultado de aplicar la regla a **todos** los lados (tres) del triángulo. La Figura I.5d es el resultado de aplicar la misma regla a los 12 lados de I.5c, y así sucesivamente.

Observe que para construir $i+1$ iteraciones de un objeto, la regla tiene que aplicarse a **todas** las partes de la iteración i del objeto. Ésta es la principal diferencia entre los L-sistemas y las gramáticas de Chomsky, y ésta es también una de las razones por la cual los L-sistemas son tan poderosos. Otra razón es la notación utilizada en los L-sistemas modernos; una notación introducida en 1979 por A. Szilard y R. E. Quinton y mejorada por P. Prusinkiewicz en 1986. Se basa en el lenguaje LOGO [Abelson 82] y el concepto de los movimientos de la tortuga. Estas dos diferencias se ilustran a continuación.

Ejemplo: Un L-sistema que trata con las dos letras x e y. El axioma es y y las dos reglas de reescritura son: $x \to xy$ (cada ocurrencia de x debe ser sustituida por xy) e $y \to x$ (cada ocurrencia de y debe ser sustituida por x). La primera iteración comienza con el axioma y, y se le aplican **ambas** reglas. La primera regla no es de aplicación, y la segunda produce x. El resultado de la primera iteración es, pues, x. La iteración 2 aplica ambas reglas a x. La primera regla reemplaza x por xy, y la segunda regla no se aplica, ya que la cadena original x no tenían ninguna y en ella. La iteración 3

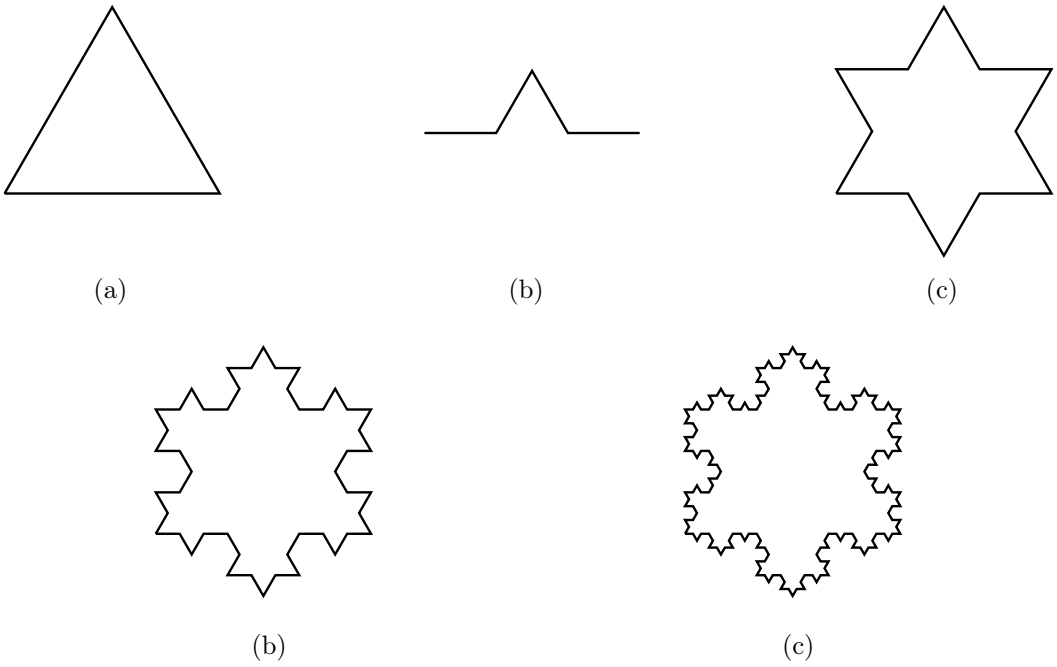


Figura I.5: Generaciones sucesivas del copo de nieve de Koch.

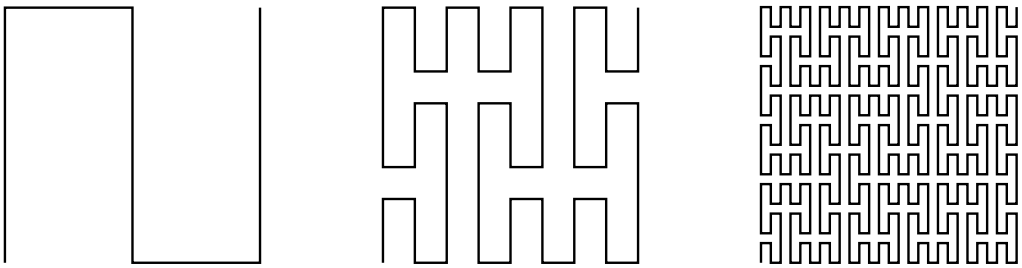
sustituye la x de xy por xy y la y de xy por x. El resultado es xyx. Iteraciones sucesivas producen las cadenas:

$$y \to x \to xy \to xyx \to xyxxy \to xyxxyxyx.$$

(Ésto no parece útil, pero espere hasta aplicar este método a las formas geométricas.) Las dos partes, izquierda y derecha, de una regla de producción se denominan su *predecesor* y su *sucesor*, respectivamente. Un L-sistema como el que arriba se llama D0L-sistema. (D0L procede de L-sistema de libre contexto determinista. Tenga en cuenta que la mayoría de los textos sobre el L-sistemas corrompen este nombre y lo escriben "DOL" en lugar de "D0L".)

Movimientos de la tortuga: Es posible definir formas geométricas, imaginando una tortuga que se mueve en el plano bidimensional, dejando en ocasiones marcas tras de sí. El lenguaje de programación LOGO suporta los comandos de dibujo para "mover" la tortuga de punto a punto y hacer que gire en un ángulo cuando se llega a un punto. Las reglas de producción de los L-sistemas también utilizan esta notación. Matemáticamente, el estado de la tortuga está representado por un triplete (x, y, α), donde (x, y) son las coordenadas actuales de la tortuga y α es su rumbo. La notación básica utilizada en dicha regla emplea los siguientes caracteres:

- **F :** La tortuga se mueve hacia adelante una distancia d, dibujando una línea recta de un espesor dado W. El estado de la tortuga cambia de (x, y, α) a $(x + d\cos\alpha, y + d\sin\alpha, \alpha)$.
- **f :** La tortuga se mueve hacia adelante como antes, pero sin dibujar nada.
- **+ :** La tortuga gira hacia la derecha (en sentido horario) en un ángulo determinado δ. Su nuevo estado es, por lo tanto, $(x, y, \alpha + \delta)$.
- **− :** La tortuga gira hacia la izquierda (en sentido antihorario) el mismo ángulo δ. Su nuevo estado es $(x, y, \alpha - \delta)$.



(a) (b) (c)

Figura I.6: Tres iteraciones de la curva de Peano.

La Tabla I.7 muestra varios comandos de carácter que tradicionalmente han sido utilizados en los L-sistemas. A medida que se realicen más investigaciones en este campo, el número de comandos de tortuga crecerá, pero el lector tiene que mantener una convención importante en mente: Cuando una regla de reescritura contiene un comando que la tortuga (i.e., la implementación de los L-sistemas en el ordenador) no entiende, *este comando es ignorado*, sin emitir ningún mensaje de error. Esta convención es útil y se utiliza comúnmente en la elaboración de formas complejas.

La Tabla I.7 implica que se necesitan varios parámetros más, tales como C, sl, y $\triangle$, para especificar completamente la forma que se está dibujando. Estos parámetros deben ser soportados por cualquier implementación informática de L-sistemas; deben tener valores predeterminados, y deben ser fáciles de modificar por el usuario. Estos parámetros se enumeran en la Tabla I.8.

La cadena `F+F+F+F` es un comando para avanzar un tramo de línea, girar a la derecha, y repetirlo tres veces más. Si el ángulo de giro es de 90º, el resultado es un cuadrado de tamaño d. Si el rumbo inicial de la tortuga es $\alpha = 90º$, entonces el punto inicial/final es la parte esquina inferior izquierda del cuadrado (Figura I.9a). La cadena `FFF+FF+F+F-F-F-FF+F+FFF` dibuja la forma de la Figura I.9B.

El copo de nieve de Koch de la Figura I.5 fue generada mediante un L-sistema con axioma `F++F++F` y la regla de producción única `F`→`F-F++F-F`. El rumbo inicial era de 0º y el ángulo de giro de 60º. La Figura I.6 muestra tres iteraciones de la curva de relleno de espacio de Peano dibujadas con un rumbo inicial y un ángulo de giro de 90º.

El L-sistema para esta curva consiste en el axioma `X` y las dos reglas de producción:

`X`→`XFYFX+F+YFXFY-F-XFYFX` e `Y`→`YFXFY-F-XFYFX+F+YFXFY.`

La clave para comprender este L-sistema es la regla de que todo comando de tortuga desconocido (en este caso los caracteres `X` e `Y`) deben ser ignorados. La primera iteración toma el axioma `X`, que es desconocido, provocando que nada sea dibujado. La siguiente iteración ejecuta las dos reglas de reescritura. La primera regla sustituye el axioma `X` por `XFYFX+F+YFXFY-F-XFYFX` que es representado (ya que `X` e `Y` son desconocidos) como `FF+F+FF-F-FF`. La segunda regla busca una `Y` en el axioma, pero no la encuentra. En consecuencia la iteración dibuja `FF+F+FF-F-FF`, lo que resulta en la Figura I.6A. La siguiente iteración comienza con `XFYFX+F+YFXFY-F-XFYFX`, reemplazando cada `X` con el sucesor de la regla 1, y cada `Y` con el sucesor de la regla 2. El resultado es una cadena muy larga, que, cuando es representada, produce la curva de la Figura I.6b.

El L-sistema para la curva de Hilbert se define de manera similar mediante el axioma `X` y las 2 reglas de producción:

`F`	Avanzar d unidades y dibujar una línea.
`f`	Avanzar d unidades sin dibujar.
`+`	Girar en sentido horario un ángulo δ.
`-`	Girar en sentido antihorario un ángulo δ.
`|`	Invertir dirección (rotar 180º)
`[`	Introducir el estado actual de la tortuga en la pila.
`]`	Sacar el estado actual de la tortuga de la pila.
`#`	Incrementar la anchura W de la línea en una cantidad ω.
`!`	Decrementar la anchura W de la línea en una cantidad ω.
`@`	Dibujar un punto con radio W.
`{`	Abrir un polígono.
`}`	Cerrar un polígono y rellenarlo con el color C.
`<`	Dividir la longitud d de la línea por un factor de escala sl.
`>`	Multiplicar la longitud d de la línea por un factor de escala sl.
`&`	Intercambiar el significado de $+$ y $-$.
`(`	Decrementar el ángulo de giro δ en Δ.
`)`	Incrementar el ángulo de giro δ en Δ.
`*`	Emparejar con cualquier carácter (usado sólo en L-sistemas sensibles al contexto).
...	Ignorar regla (usado sólo en L-sistemas sensibles al contexto).

Tabla I.7: Convenciones de un L-sistema para comandos de tortuga.

d	Longitud de la línea.
sl	Factor de escala para la longitud de la línea d
W	Anchura de la línea.
ω	Incremento del ancho de línea.
α	Rumbo inicial de la tortuga.
δ	Ángulo de giro.
Δ	Incrementar/Decrementar el ángulo de giro δ.
C	Color por defecto del relleno del polígono.

Tabla I.8: Parámetros de tortuga adicionales.

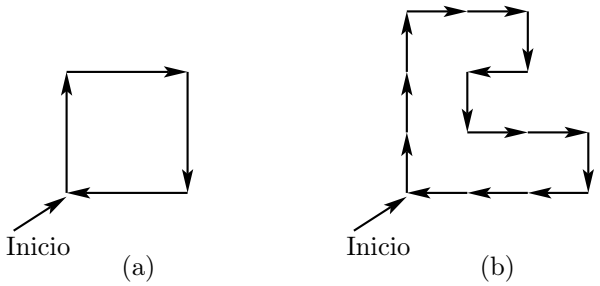


Figura I.9: Ejemplos de movimientos de tortuga.

`X→-YF+XFX+FY-` e `Y→+XF-YFY-FX+.`

◇ **Ejercicio I.13 (sol. en pág. 1118):** Muéstrese cómo obtener las 4 orientaciones de la curva de Hilbert sobre el L-sistema anterior.

Bibliografía

Abelson, H. and A. A. diSessa (1982) *Turtle Geometry*, Cambridge, MA, MIT Press.

Prusinkiewicz, Przemyselaw (1986) *Graphical Applications of L-systems*, en Proc. of Graphics Interface '86—Vision Interface '86, pp .247–253.

Prusinkiewicz, P., and A. Lindenmayer (1990) *The Algorithmic Beauty of Plants*, New York, Springer Verlag.

Prusinkiewicz, P., A. Lindenmayer, and F. D. Fracchia (1991) "Synthesis of Space-Filling Curves on the Square Grid," en *Fractals in the Fundamental and Applied Sciences*, edited by Peitgen, H.-O. et al., Amsterdam, Elsevier Science Publishers, pp. 341–366.

Smith, Alvy Ray (1984) "Plants, Fractals and Formal Languages," *Computer Graphics* **18**(3):1–10.

Szilard, A. L. and R. E. Quinton (1979) "An Interpretation for D0L Systems by Computer Graphics," *The Science Terrapin* **4**:8–13.

I.9. El alfabeto griego

A	α	alpha	I	ι	iota	P	ρ	ϱ	rho
B	β	beta	K	κ	kappa	Σ	σ	ς	sigma
Γ	γ	gamma	Λ	λ	lambda	T	τ		tau
Δ	δ	delta	M	μ	mu	Y	υ		upsilon
E	ϵ	épsilon	*N*	ν	nu	Φ	ϕ	φ	phi
Z	ζ	zeta	Ξ	ξ	xi	X	ξ		xi
H	η	eta	O	o	omicrom	Ψ	ψ		psi
Θ	θ	theta	Π	π	pi	Ω	ω		omega

I.10. Polinomios de interpolación

Véase la Sección 4.21.4.

> Sentado en su desordenado escritorio, acariciado por una nube de tabaco, podría estudiar minuciosamente el manuscrito, tachando, interpolando, argumentando de nuevo, y posteriormente, hacer referencia a los volúmenes en sus estantes.
>
> —Christopher Morley, *La librería embrujada.*

Soluciones a los ejercicios

> Un pájaro no canta porque tenga una respuesta,
> canta porque tiene una canción.
>
> —Proverbio Chino

Intro.1: abstemious, abstentious, adventitious, annelidous, arsenious, arterious, facetious, sacrilegious[1].

Intro.2: Cuando una casa de software tiene un producto popular, cuida la propuesta de nuevas versiones. Un usuario puede actualizar una versión antigua a una nueva y la actualización —por lo general— viene como un archivo comprimido en un disquete. Con el tiempo, las actualizaciones se hacen más grandes y —en algún momento— una actualización puede no caber en un solo disquete. Por ello, es importante una buena compresión en el caso de las actualizaciones de software. El tiempo que requiere comprimir y descomprimir la actualización no es importante ya que estas operaciones se suelen hacer una sola vez. Recientemente, los fabricantes de software han decidido proporcionar las actualizaciones a través de Internet; pero incluso en estos casos, es importante contar con archivos pequeños debido a los tiempos de descarga involucrados.

1.1: (1) hacer una pregunta, (2) es absolutamente necesario, (3) con previo aviso, (4) punto de ebullición caliente, (5) subir, (6) un examen riguroso, (7) exactamente lo mismo, (8) obsequio, (9) calentador de agua, (10) mi opinión personal, (11) recién nacido, (12) aplazado hasta más tarde, (13) sorpresa inesperada, (14) misterios sin resolver.[2]

1.2: Una forma razonable de utilizarlos es codificar las cinco cadenas más frecuentes en el texto. Debido a que la compresión irreversible de texto es un método de propósito particular, el usuario puede saber qué cadenas son las más comunes en el flujo de datos a comprimir; tiene que proporcionárselas al codificador y además debe escribirlas al principio de la secuencia de salida (para el uso del decodificador).

1.3: $6, 8, 0, 1, 3, 1, 4, 1, 3, 1, 4, 1, 3, 1, 4, 1, 3, 1, 2, 2, 2, 2, 6, 1, 1$. Los dos primeros, son la resolución del bitmap (6×8). El siguiente, indica que el primer píxel es negro. Si se almacenan usando un byte por número, el tamaño es de 25 bytes —en comparación con el del mapa de bits de tan sólo $6 \times 8\,\text{bits} = 6\,\text{bytes}$—. El método no funciona con imágenes pequeñas.

[1] Abstemio, abstención, accidental, anélido, arsénico, arterioso, gracioso, sacrilegio.

[2] (1) ask a question, (2) absolutely necessary, (3) advance warning, (4) boiling hot, (5) climb up, (6) close scrutiny, (7) exactly the same, (8) free gift, (9) hot water heater, (10) my personal opinion, (11) newborn baby, (12) postponed until later, (13) unexpected surprise, (14) unsolved mysteries.

1.4: El método RLE para imágenes se basa en la idea de que los píxeles adyacentes tienden a ser idénticos. No es común que el último píxel de una fila sea idéntico al primer píxel de la fila siguiente.

1.5: Cada una de las cuatro primeras filas produce los ocho *runs*: 1, 1, 1, 2, 1, 1, 1, `eol`. Cada una de las filas 6 y 8 genera los cuatro *runs*: 0, 7, 1, `eol`. Cada una de las filas 5 y 7 produce los dos *runs*: 8, `eol`. El número total de *runs* (incluido el carácter `eol`) es, por lo tanto: $4 \cdot 8 + 2 \cdot 4 + 2 \cdot 2 = 44$.
Cuando se explora por columnas, las columnas 1, 3 y 6 generan cinco *runs* cada una: 5, 1, 1, 1, `eol`. Cada una de las columnas 2, 4, 5 y 7 genera los seis *runs*: 0, 5, 1, 1, 1, `eol`. La columna 8 pasa a ser 4, 4, `eol`, por lo que el número total de *runs* es de $3 \cdot 5 + 4 \cdot 6 + 1 \cdot 3 = 42$.
Por lo tanto, esta imagen es "equilibrada" con respecto al muestreo por filas y columnas.

1.6: El resultado son los cinco grupos siguientes:

1. De W_1 a W_2: 00000, 11111,
2. De W_3 a W_{10}: 00001, 00011, 00111, 01111, 11110, 11100, 11000, 10000.
3. De W_{11} a W_{22}: 00010, 00100, 01000, 00110, 01100, 01110, 11101, 11011, 10111, 11001, 10011, 10001.
4. De W_{23} a W_{30}: 01011, 10110, 01101, 11010, 10100, 01001, 10010, 00101.
5. De W_{31} a W_{32}: 01010, 10101.

1.7: Los siete códigos son:

$$0000,\ 1111,\ 0001,\ 1110,\ 0000,\ 0011,\ 1111,$$

formando una cadena con *seis runs*. Cogiendo en orden cada elemento de la secuencia anterior y aplicando la regla de complementación obtenemos

$$0000,\ 1111,\ 1110,\ 1110,\ 0000,\ 0011,\ 0000,$$

con *siete runs*. La regla de complementación no reduce siempre el número de *runs*.

La transformación detallada es la siguiente:

1. 0000 termina en 0; 1111 no se complementa $(0000, 1111)$,
2. 1111 termina en 1; 0001 se complementa $(\ldots, 1111, 1110)$,
3. 1110 termina en 0; 1110 no se complementa $(\ldots, 1110, 1110)$,
4. 1110 termina en 0; 0000 no se complementa $(\ldots, 1110, 0000)$,
5. 0000 termina en 0; 0011 no se complementa $(\ldots, 0000, 0011)$,
6. 0011 termina en 1; 1111 se complementa $(\ldots, 0011, 0000)$.

1.8: Como "11 22 90 00 00 33 44". El 00 siguiente al 90 indica que no hay *run* y el siguiente 00 se interpreta como un carácter normal.

1.9: Los seis caracteres "123ABC" tienen códigos ASCII 31, 32, 33, 41, 42 y 43. La traducción de estos números hexadecimales a binario produce "00110001 00110010 00110011 01000001 01000010 01000011".
El siguiente paso es dividir esta cadena de 48 bits en bloques de 6 bits. Son 001100=12, 010011=19, 001000=8, 110011=51, 010000=16, 010100=20, 001001=9 y 000011=3. El carácter en la posición 12 en la tabla BinHex es "-" (la numeración de la posición comienza desde cero). El que está en la posición 19 es "6". El resultado final es la cadena "-6)c38*$".

a	abcdmnop	0	a	abcdmnop	0	a	abcdmnop	0	a	abcdmnop	0
b	abcdmnop	1	b	abcdmnop	1	b	abcdmnop	1	b	abcdmnop	1
c	bacdmnop	2	c	bacdmnop	2	c	bacdmnop	2	c	bacdmnop	2
d	bcadmnop	3	d	cbadmnop	3	d	bcadmnop	3	d	cbadmnop	3
d	bcdamnop	2	d	cdbamnop	1	m	bcdamnop	4	m	cdbamnop	4
c	bdcamnop	2	c	dcbamnop	1	n	bcdmanop	5	n	cdmbanop	5
b	bcdamnop	0	b	cdbamnop	2	o	bcdmnaop	6	o	cdmnbaop	6
a	bcdamnop	3	a	bcdamnop	3	p	bcdmnoap	7	p	cdmnobap	7
m	bcadmnop	4	m	bacdmnop	4	a	bcdmnopa	7	a	cdmnopba	7
n	bcamdnop	5	n	bamcdnop	5	b	bcdmnoap	0	b	cdmnoapb	7
o	bcamndop	6	o	bamncdop	6	c	bcdmnoap	1	c	cdmnobap	0
p	bcamnodp	7	p	bamnocdp	7	d	cbdmnoap	2	d	cdmnobap	1
p	bcamnopd	6	p	bamnopcd	5	m	cdbmnoap	3	m	dcmnobap	2
o	bcamnpod	6	o	bampnocd	5	n	cdmbnoap	4	n	mdcnobap	3
n	bcamnopd	4	n	bamopncd	5	o	cdmnboap	5	o	mndcobap	4
m	bcanmopd	4	m	bamnopcd	2	p	cdmnobap	7	p	mnodcbap	7
	bcamnopd			mbanopcd			cdmnobpa			mnodcpba	
	(a)			(b)			(c)			(d)	

Tabla Sol.1: Codificación con el método *avanzar-k*.

1.10: El Ejercicio 2.1 muestra que el código binario de un entero i es $1 + [\log_2 i]$ bits de longitud. Nosotros añadimos $[\log_2 i]$ ceros, resultando un tamaño total de $1 + 2\,[\log_2 i]$ bits.

1.11: La Tabla Sol.1 resume los resultados. En (a), la primera cadena es codificada con $k = 1$. En (b) se codifica con $k = 2$. Las columnas (c) y (d) son las codificaciones de la segunda cadena con $k = 1$ y $k = 2$, respectivamente. Los promedios[3] de las cuatro columnas son 3,4375, 3,25, 3,5625 y 3,6875; ¡muy similares! El método *avanzar-k* que utiliza valores pequeños de k no favorece a las cadenas que cumplan la propiedad de concentración.

1.12: La Tabla Sol.2 resume los pasos de la decodificación. Observe cómo es similar a la Tabla 1.16, indicando que mover–al–frente es un método de compresión de datos simétrico.

2.1: Es $1 + [\log_2 i]$ como puede verse mediante simple experimentación.

2.2: El número 2 es el entero más pequeño que puede servir como base para un sistema de numeración.

2.3: Reemplazando 10 por 3 obtenemos $x = k \cdot \log_2 3 \approx 1{,}58 \cdot k$. Un trit, por lo tanto, tiene un valor aproximado de 1,58 bits.

2.4: Asumimos un alfabeto con dos símbolos a_1 y a_2, con probabilidades P_1 y P_2, respectivamente. Puesto que $P_1 + P_2 = 1$, la entropía del alfabeto es $-P_1 \cdot \log_2 P_1 - (1 - P_1) \cdot \log_2 (1 - P_1)$. La Tabla Sol.3 muestra las entropías para ciertos valores de probabilidades[4]. Cuando $P_1 = P_2$, al menos se requiere 1 bit para codificar cada símbolo, lo que refleja el hecho de que la entropía es

[3] Suma de elementos / nº de elementos.

[4] Para hallar $\log_2 x$ con una calculadora sólo hay que tener en cuenta que $\log_a x = {}^{\ln x}/_{\ln a}$. Por ejemplo, para calcular la entropía de la segunda fila, la cuenta a realizar es:

$$-0{,}90 \cdot \left[\frac{\ln 0{,}90}{\ln 2}\right] - 0{,}10 \cdot \left[\frac{\ln 0{,}10}{\ln 2}\right].$$

Código emitido	A (antes de la adición)	A (después de la adición)	Palabra
0the	()	(the)	the
1boy	(the)	(the, boy)	boy
2on	(boy, the)	(boy, the, on)	on
3my	(on, boy, the)	(on, boy, the, my)	my
4right	(my, on, boy, the)	(my, on, boy, the, right)	right
5is	(right, my, on, boy, the)	(right, my, on, boy, the, is)	is
5	(is, right, my, on, boy, the)	(is, right, my, on, boy, the)	the
2	(the, is, right, my, on, boy)	(the, is, right, my, on, boy)	right
5	(right, the, is, my, on, boy)	(right, the, is, my, on, boy)	boy
	(boy, right, the, is, my, on)		

Tabla Sol.2: Decodificación de palabras de múltiples letras.

P1	P2	Entropía
99	1	0,08
90	10	0,47
80	20	0,72
70	30	0,88
60	40	0,97
50	50	1,00

Tabla Sol.3: Probabilidades y entropías de dos símbolos.

máxima, la redundancia es cero y los datos no pueden ser comprimidos. Sin embargo, cuando las probabilidades son muy diferentes, el número mínimo de bits necesarios por símbolo desciende de manera significativa. Tal vez no seamos capaces de desarrollar un método de compresión que use 0,08 bits por símbolo, pero sabemos que cuando $P_1 = 99\,\%$, este es el mínimo teórico.

> Una herramienta esencial de esta teoría [información] es un cuantificador para medir la cantidad de información transmitida por un mensaje. Supongamos que un mensaje se codifica como un long (entero largo). Para cuantificar la información contenida en este mensaje, Shannon propuso contar el número de sus dígitos. De acuerdo con este criterio, 3,14159, por ejemplo, transmite dos veces más información que 3,14 y seis veces más que 3. Golpeado por la similitud entre esta receta y la famosa ecuación inscrita en la tumba de Boltzman[a] (la entropía es el número de dígitos de la probabilidad), Shannon llamó a su fórmula "entropía de la información."
>
> Hans Christian von Baeyer, *El demonio de Maxwell* (1998).
>
> [a] $S = k \cdot logW$.

2.5: Es fácil ver que el código unario satisface la propiedad prefijo, por lo que puede ser utilizado con seguridad como código de tamaño variable. Ya que su longitud L cumple que $L = n$ obtenemos $2^{-L} = 2^{-n}$, por lo que tiene sentido usarlo en aquellos casos donde los datos de entrada se

n	$a = 10 + n \cdot 2$	n-ésima palabra clave	Número de palabras clave	Rango de enteros
0	10	$0\underbrace{x\ldots x}_{10}$	$2^{10} = 1\text{K}$	0-1023
1	12	$10\underbrace{xx\ldots x}_{12}$	$2^{12} = 4\text{K}$	1024-5119
2	14	$11\underbrace{xx\ldots xx}_{14}$	$2^{14} = 16\text{K}$	5120-21503
		Total	21504	

Tabla Sol.4: El código unario general $(10, 2, 14)$.

componen de n números enteros con probabilidades $P(n) \approx 2^{-n}$. Si los datos se prestan al uso del código unario, el algoritmo Huffman puede ser omitido por completo y los códigos de todos los símbolos pueden construirse fácil y rápidamente antes de que la compresión o la descompresión se inicie.

2.6: El triplete $(n, 1, n)$, define los códigos binarios estándar de n bits, como puede ser verificado por construcción directa. Es fácil ver que el número de estos códigos es

$$\frac{2^{n+1} - 2^n}{2^1 - 1} = 2^n.$$

El triplete $(0, 0, \infty)$ define los códigos 0, 10, 110, 1110,... que son los códigos unarios, pero asignados a los números enteros 0, 1, 2,... en vez de a 1, 2, 3,....

2.7: El triplete $(1, 1, 30)$ produce:

$$\frac{\left(2^{30} - 2^1\right)}{(2^1 - 1)} \approx \text{Mil millones}$$

de códigos.

2.8: Esto es sencillo. La Tabla Sol.4 muestra el código. Sólo hay tres palabras clave diferentes, ya que "inicio" y "parada" están muy cercanos; pero hay muchos códigos, ya que "inicio" es grande.

2.9: Cada parte de C_4 es el código binario estándar de un número entero, por lo que comienza con un 1. Una parte que se inicia con un 0, por lo tanto, indica al decodificador que se trata del último bit del código.

2.10: Usamos la siguiente propiedad: la representación de Fibonacci de un número entero no tiene unos adyacentes. Si R es un entero positivo, construimos la representación de Fibonacci y añadimos un bit 1 al final del resultado. La representación de Fibonacci del entero 5 es 0001 —pues $5 = 0 \times 1 + 0 \times 2 + 0 \times 3 + 1 \times 5$—, así que el código prefijo de Fibonacci de 5 es 00011. Del mismo modo, la representación de Fibonacci de 33 es 1010101, por lo que su código prefijo de Fibonacci es 10101011. Es obvio que cada uno de estos códigos termina con dos unos adyacentes, por lo que puede ser decodificado de forma única. Sin embargo, la propiedad de no tener unos adyacentes restringe el número de patrones binarios disponibles para dichos códigos, por lo que son más largos que los otros códigos mostrados aquí.

2.11: Las divisiones se pueden hacer de diferentes maneras. La Tabla Sol.5 muestra una forma de asignar los códigos de Shannon–Fano a los 7 símbolos. El tamaño medio en este caso es:

$$0{,}25 \times 2 + 0{,}20 \times 3 + 0{,}15 \times 3 + 0{,}15 \times 2 + 0{,}10 \times 3 + 0{,}10 \times 4 + 0{,}05 \times 4 = 2{,}75\,{}^{bits}/_{símbolo}$$

	Probabilidad	Pasos					Final
1.	0,25	1	1				:11
2.	0,20	1	0	1			:101
3.	0,15	1	0	0			:100
4.	0,15	0	1				:01
5.	0,10	0	0		1		:001
6.	0,10	0	0		0	1	:0001
7.	0,05	0	0		0	0	:0000

Tabla Sol.5: Ejemplo de Shannon–Fano.

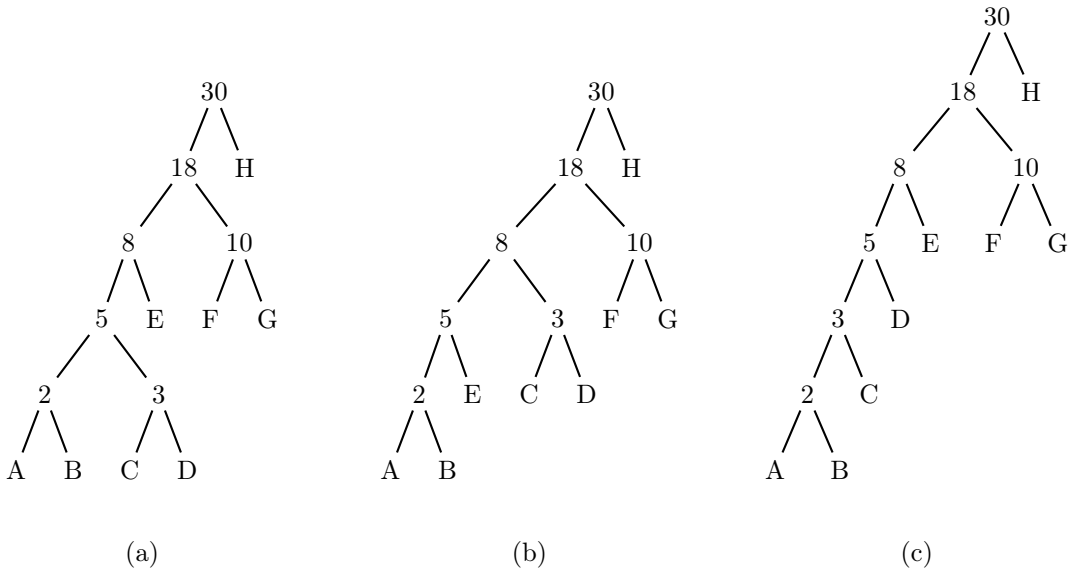


Figura Sol.6: Tres árboles de Huffman para ocho símbolos.

2.12: La entropía es: $-2 \cdot (0{,}25 \times \log_2 0{,}25) - 4 \cdot (0{,}125 \times \log_2 0{,}125) = 2{,}5$.

2.13: En la Figura Sol.6 se muestran los tres árboles —a, b y c—. Los tamaños respectivos de los códigos de los árboles son:

$$(5{\cdot}1{+}5{\cdot}1{+}5{\cdot}1{+}5{\cdot}2{+}3{\cdot}3{+}3{\cdot}5{+}3{\cdot}5{+}1{\cdot}12)/30 = 76/30,$$
$$(5{\cdot}1{+}5{\cdot}1{+}4{\cdot}1{+}4{\cdot}2{+}4{\cdot}3{+}3{\cdot}5{+}3{\cdot}5{+}1{\cdot}12)/30 = 76/30,$$
$$(6{\cdot}1{+}6{\cdot}1{+}5{\cdot}1{+}4{\cdot}2{+}3{\cdot}3{+}3{\cdot}5{+}3{\cdot}5{+}1{\cdot}12)/30 = 76/30.$$

2.14: Después de añadir los símbolos A, B, C, D, E, F y G al árbol, quedan tres símbolos: a_{ABEF} (con una probabilidad de $10/30$), a_{CDG} (con una probabilidad de $8/30$) y H (con una probabilidad de $12/30$). Los dos símbolos con probabilidades más bajas son: a_{ABEF} y a_{CDG}, por lo que tendrían que haberse fusionado. En cambio, se han unido los símbolos a_{CDG} y H, creando un árbol que no es de Huffman (Figura Sol.7).

2.15: La segunda fila de la Tabla Sol.8 (proporcionada por Guy Blelloch), muestra un símbolo cuyo código de Huffman es de tres bits de longitud, pero para el cual $[-\log_2 0{,}3] = [1{,}737] = 2$.

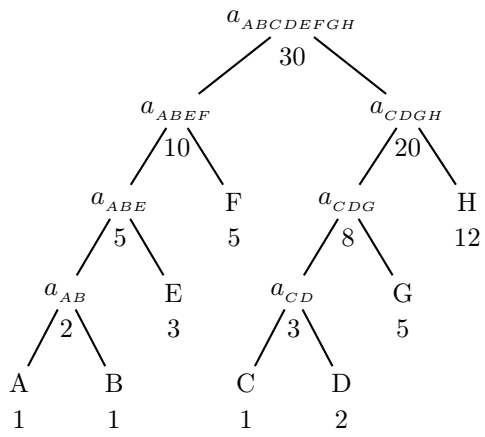


Figura Sol.7: Construcción errónea de un árbol de Huffman (8 símbolos).

P_i	Código	$-\log_2 P_i$	$[-\log_2 P_i]$
,01	000	6,644	7
∗,30	001	1,737	2
,34	01	1,556	2
,35	1	1,515	2

Tabla Sol.8: Ejemplo de código de Huffman.

2.16: La explicación es simple. Imagine un gran alfabeto donde todos los símbolos tienen —aproximadamente— la misma probabilidad de aparición. Puesto que el alfabeto es grande, una probabilidad muy pequeña produce códigos largos. Imagine el otro caso extremo, donde los símbolos tienen probabilidades altas (y por tanto, códigos cortos). Ya que las probabilidades tienen que sumar 1, el resto de los símbolos tendrá probabilidades bajas (y por ello, códigos largos). Por consiguiente, vemos que el tamaño de un código depende de la probabilidad, pero se ve afectada indirectamente por el tamaño del alfabeto.

2.17: La Figura Sol.9 muestra los códigos Huffman para 5, 6, 7 y 8 símbolos con probabilidades iguales. Cuando n es una potencia de 2, todos los códigos tienen el mismo tamaño ($\log_2 n$ bits). En los demás casos, los códigos se representan con un número de bits muy cercano a ese tamaño. Esto demuestra que los símbolos con la misma probabilidad no se benefician de los códigos de longitud variable. (Esta es otra forma de decir que un texto aleatorio no se puede comprimir.) La Tabla (Sol.10) muestra los códigos, sus tamaños medios y las varianzas.

2.18: Aumenta exponencialmente de 2^s a $2^{s+n} = 2^s \times 2^n$.

n	p	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	Tamaño medio	Var.
5	0,200	111	110	101	100	0				2,6	0,64
6	0,167	111	110	101	100	01	00			2,672	0,2227
7	0,143	111	110	101	100	011	010	00		2,86	0,1226
8	0,125	111	110	101	100	011	010	001	000	3	0

Tabla Sol.10: Códigos de Huffman para 5-8 símbolos.

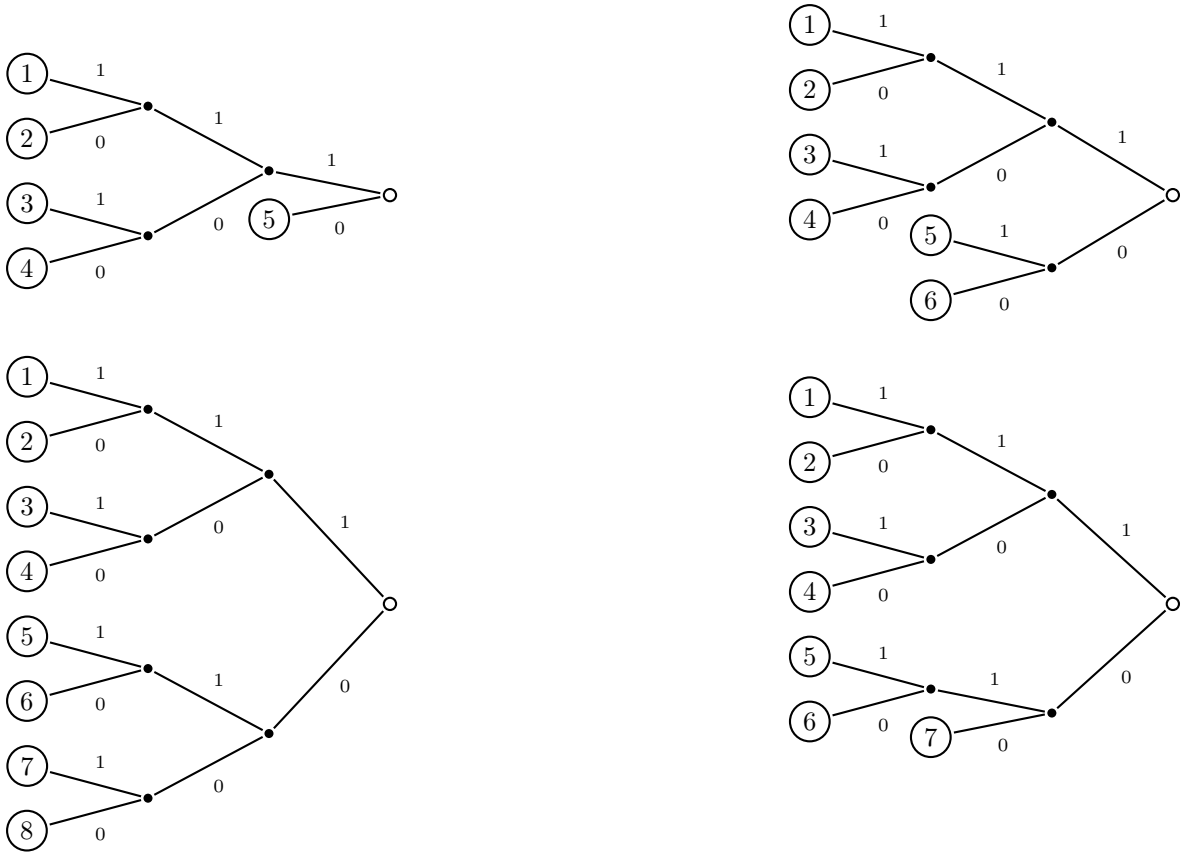


Figura Sol.9: Códigos de Huffman para probabilidades iguales.

2.19: El valor binario de 127 es 01111111 y el de 128 es 10000000. La mitad de los píxeles en cada plano de bits será, por lo tanto, 0 y la otra mitad, 1. En el peor de los casos, cada plano de bits será un tablero de ajedrez, i.e., tendrá muchos *runs* de tamaño uno; en cuyo caso, cada *run* requiere un código de 1 bit, dando lugar a un bit de código por cada píxel y plano de bits, u ocho bits de código por píxel de la imagen completa, lo que lleva a que no se obtenga ninguna compresión. En comparación, un código Huffman para este tipo de imagen requiere sólo dos códigos (ya que hay sólo dos valores de píxel) y pueden ser de un bit cada uno. Esto conduce a un bit de código por píxel, o un factor de compresión de ocho.

2.20: Los dos árboles se muestran en la Figura 2.26c,d (pág. 88). El tamaño medio del árbol de Huffman de código binario es:

$$1 \times 0{,}49 + 2 \times 0{,}26 + 5 \times 0{,}02 + 5 \times 0{,}03 \\ +5 \times 0{,}04 + 5 \times 0{,}04 + 3 \times 0{,}12 = 2{,}02 \text{ bits por símbolo},$$

y la del árbol ternario es:

$$1 \times 0{,}26 + 3 \times 0{,}02 + 3 \times 0{,}03 + 3 \times 0{,}04 \\ +2 \times 0{,}04 + 2 \times 0{,}12 + 1 \times 0{,}49 = 1{,}34 \text{ trits por símbolo}.$$

2.21: La Figura Sol.11 muestra cómo continúa el bucle hasta que la pila se reduce a un solo nodo, que es el puntero único 2. Ésto indica que la frecuencia total (lo que ocurre cuando es 100 en

nuestro ejemplo) se almacena en `A`[2]. Todas las otras frecuencias se han sustituido por punteros. La Figura Sol.12a muestra las pilas generadas durante el bucle.

2.22: El resultado final del bucle es:

1	2	3	4	5	6	7	8	9	10	11	12	13	14
[2]	100	2	2	3	4	5	3	4	6	5	7	6	7

de donde es fácil de averiguar las longitudes de los códigos de los siete símbolos. Para encontrar la longitud del código del símbolo 14, e.g., seguimos los punteros 7, 5, 3 y 2 desde `A`[14] hasta la raíz. Son necesarios cuatro pasos, por lo que la longitud del código es 4.

2.23: Las longitudes de los códigos para los siete símbolos asociados a las frecuencias 25, 20, 13, 17, 9, 11 y 5 son 2, 2, 3, 3, 4, 3 y 4 bits, respectivamente. Ésto puede verificarse también a partir del árbol de código de Huffman de la Figura Sol.12b. Un conjunto de códigos derivados de dicho árbol se muestra en la siguiente tabla:

Cuenta:	25	20	13	17	9	11	5
Código:	01	11	101	000	0011	100	0010
Longitud:	2	2	3	3	4	3	4

2.24: A un símbolo con alta frecuencia de ocurrencia debe asignarsele un código más corto. Por lo tanto, tiene que aparecer en la parte alta del árbol. El requisito de ordenar las frecuencias de izquierda a derecha en cada nivel es artificial. En principio, no es necesario, pero simplifica el proceso de actualización del árbol.

2.25: Las Figuras Sol.13-I, Sol.13-II, Sol.13-III y Sol.13-IV, muestran el árbol inicial y cómo se actualiza en los 11 pasos, desde (a) hasta (k). Observe cómo al símbolo `esc` se le asignan códigos diferentes todo el tiempo y cómo se mueven los distintos símbolos por el árbol y cambian sus códigos. El código `10`, e.g., es el código del símbolo "`i`" en los pasos (f) e (i), pero es el código de "`s`" en los pasos (e) y (j). El código de un espacio en blanco es `011` en el paso (h), pero es `00` en el paso (k). La salida final es: "`s0i00r100␣1010000d011101000`". Un total de $5 \times 8 + 22 = 62$ bits. Por lo tanto, la razón de compresión es $^{62}/_{88} \approx 0{,}7$.

2.26: Un sencillo cálculo muestra que el tamaño medio de un token de la Tabla 2.35 (pág. 101) está cercano a los nueve bits. En la etapa 2, cada byte de 8 bits será reemplazado, en promedio, por un token de 9 bits, lo que produce un factor de expansión de $^{9}/_{8} = 1{,}125$ o un 12,5 %.

2.27: El descompresor interpreta los datos de entrada como 111110 0110 11000 0..., que es la cadena `XRP`....

2.28: Debido a que una máquina de fax típica explora líneas de unas 8,2 pulgadas de ancho ($\approx$ 208 mm), un escaneo de una línea en blanco produce 1664 pels en blanco consecutivos.

2.29: Estos códigos son necesarios para casos como el ejemplo 4, donde la longitud del *run* es de 64, 128 o cualquier tipo de longitud que haya sido asignada a un código de establecimiento.

2.30: Es posible que las máquinas de fax (en la actualidad o en el futuro), admitan un papel más ancho, por lo que los códigos del Grupo 3 se diseñaron para acomodarse a esta circunstancia.

2.31: Cada línea escaneada comienza con un pel blanco, por lo que cuando el decodificador introduce el siguiente código sabe si es por un *run* blanco o por uno negro. Por esta razón los códigos de las Tablas 2.41a (pág. 112), 2.41b-I (pág. 113) y 2.41b-II (pág. 113) tienen que satisfacer la propiedad del prefijo en cada columna, pero no entre las columnas.

$$\frac{1}{[7} \quad \frac{2}{11} \quad \frac{3}{6} \quad \frac{4}{8} \quad \frac{5}{9]} \quad \frac{6}{24} \quad \frac{7}{14} \quad \frac{8}{25} \quad \frac{9}{20} \quad \frac{10}{6} \quad \frac{11}{17} \quad \frac{12}{7} \quad \frac{13}{6} \quad \frac{14}{7}$$

$$\frac{1}{[11} \quad \frac{2}{9} \quad \frac{3}{8} \quad \frac{4}{6]} \quad \frac{5}{} \quad \frac{6}{24} \quad \frac{7}{14} \quad \frac{8}{25} \quad \frac{9}{20} \quad \frac{10}{6} \quad \frac{11}{17} \quad \frac{12}{7} \quad \frac{13}{6} \quad \frac{14}{7}$$

$$\frac{1}{[11} \quad \frac{2}{9} \quad \frac{3}{8} \quad \frac{4}{6]} \quad \frac{5}{17+14} \quad \frac{6}{24} \quad \frac{7}{14} \quad \frac{8}{25} \quad \frac{9}{20} \quad \frac{10}{6} \quad \frac{11}{17} \quad \frac{12}{7} \quad \frac{13}{6} \quad \frac{14}{7}$$

$$\frac{1}{[5} \quad \frac{2}{9} \quad \frac{3}{8} \quad \frac{4}{6]} \quad \frac{5}{31} \quad \frac{6}{24} \quad \frac{7}{5} \quad \frac{8}{25} \quad \frac{9}{20} \quad \frac{10}{6} \quad \frac{11}{5} \quad \frac{12}{7} \quad \frac{13}{6} \quad \frac{14}{7}$$

$$\frac{1}{[9} \quad \frac{2}{6} \quad \frac{3}{8} \quad \frac{4}{5]} \quad \frac{5}{31} \quad \frac{6}{24} \quad \frac{7}{5} \quad \frac{8}{25} \quad \frac{9}{20} \quad \frac{10}{6} \quad \frac{11}{5} \quad \frac{12}{7} \quad \frac{13}{6} \quad \frac{14}{7}$$

$$\frac{1}{[6} \quad \frac{2}{8} \quad \frac{3}{5]} \quad \frac{4}{} \quad \frac{5}{31} \quad \frac{6}{24} \quad \frac{7}{5} \quad \frac{8}{25} \quad \frac{9}{20} \quad \frac{10}{6} \quad \frac{11}{5} \quad \frac{12}{7} \quad \frac{13}{6} \quad \frac{14}{7}$$

$$\frac{1}{[6} \quad \frac{2}{8} \quad \frac{3}{5]} \quad \frac{4}{20+24} \quad \frac{5}{31} \quad \frac{6}{24} \quad \frac{7}{5} \quad \frac{8}{25} \quad \frac{9}{20} \quad \frac{10}{6} \quad \frac{11}{5} \quad \frac{12}{7} \quad \frac{13}{6} \quad \frac{14}{7}$$

$$\frac{1}{[4} \quad \frac{2}{8} \quad \frac{3}{5]} \quad \frac{4}{44} \quad \frac{5}{31} \quad \frac{6}{4} \quad \frac{7}{5} \quad \frac{8}{25} \quad \frac{9}{4} \quad \frac{10}{6} \quad \frac{11}{5} \quad \frac{12}{7} \quad \frac{13}{6} \quad \frac{14}{7}$$

$$\frac{1}{[8} \quad \frac{2}{5} \quad \frac{3}{4]} \quad \frac{4}{44} \quad \frac{5}{31} \quad \frac{6}{4} \quad \frac{7}{5} \quad \frac{8}{25} \quad \frac{9}{4} \quad \frac{10}{6} \quad \frac{11}{5} \quad \frac{12}{7} \quad \frac{13}{6} \quad \frac{14}{7}$$

$$\frac{1}{[5} \quad \frac{2}{4]} \quad \frac{3}{} \quad \frac{4}{44} \quad \frac{5}{31} \quad \frac{6}{4} \quad \frac{7}{5} \quad \frac{8}{25} \quad \frac{9}{4} \quad \frac{10}{6} \quad \frac{11}{5} \quad \frac{12}{7} \quad \frac{13}{6} \quad \frac{14}{7}$$

$$\frac{1}{[5} \quad \frac{2}{4]} \quad \frac{3}{25+31} \quad \frac{4}{44} \quad \frac{5}{31} \quad \frac{6}{4} \quad \frac{7}{5} \quad \frac{8}{25} \quad \frac{9}{4} \quad \frac{10}{6} \quad \frac{11}{5} \quad \frac{12}{7} \quad \frac{13}{6} \quad \frac{14}{7}$$

$$\frac{1}{[3} \quad \frac{2}{4]} \quad \frac{3}{56} \quad \frac{4}{44} \quad \frac{5}{3} \quad \frac{6}{4} \quad \frac{7}{5} \quad \frac{8}{3} \quad \frac{9}{4} \quad \frac{10}{6} \quad \frac{11}{5} \quad \frac{12}{7} \quad \frac{13}{6} \quad \frac{14}{7}$$

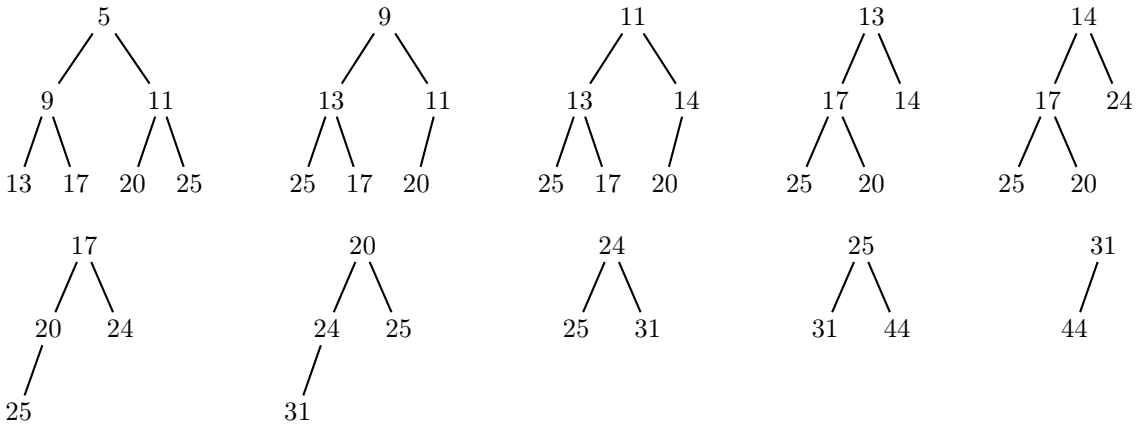
$$\frac{1}{[4} \quad \frac{2}{3]} \quad \frac{3}{56} \quad \frac{4}{44} \quad \frac{5}{3} \quad \frac{6}{4} \quad \frac{7}{5} \quad \frac{8}{3} \quad \frac{9}{4} \quad \frac{10}{6} \quad \frac{11}{5} \quad \frac{12}{7} \quad \frac{13}{6} \quad \frac{14}{7}$$

$$\frac{1}{[3]} \quad \frac{2}{} \quad \frac{3}{56} \quad \frac{4}{44} \quad \frac{5}{3} \quad \frac{6}{4} \quad \frac{7}{5} \quad \frac{8}{3} \quad \frac{9}{4} \quad \frac{10}{6} \quad \frac{11}{5} \quad \frac{12}{7} \quad \frac{13}{6} \quad \frac{14}{7}$$

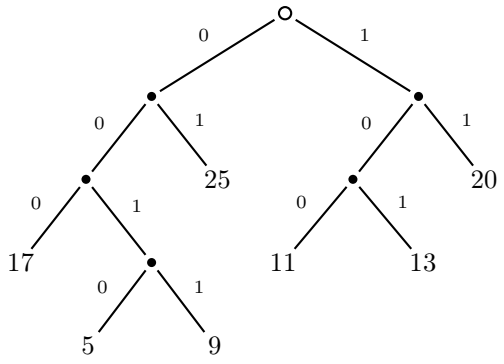
$$\frac{1}{[3]} \quad \frac{2}{56+44} \quad \frac{3}{56} \quad \frac{4}{44} \quad \frac{5}{3} \quad \frac{6}{4} \quad \frac{7}{5} \quad \frac{8}{3} \quad \frac{9}{4} \quad \frac{10}{6} \quad \frac{11}{5} \quad \frac{12}{7} \quad \frac{13}{6} \quad \frac{14}{7}$$

$$\frac{1}{[2]} \quad \frac{2}{100} \quad \frac{3}{2} \quad \frac{4}{2} \quad \frac{5}{3} \quad \frac{6}{4} \quad \frac{7}{5} \quad \frac{8}{3} \quad \frac{9}{4} \quad \frac{10}{6} \quad \frac{11}{5} \quad \frac{12}{7} \quad \frac{13}{6} \quad \frac{14}{7}$$

Figura Sol.11: El cribado de la pila.



(a)



(b)

Figura Sol.12: (a) Pilas. (b) Árbol de código de Huffman.

2.32: Imagine una línea de exploración en la que todos los *runs* tienen longitud uno (estrictamente pels alternos). Es fácil ver que este caso produce expansión. El código de un *run length* de un pel blanco es *000111* y el de un pel negro es *010*. Dos pels consecutivos de diferente color, por lo tanto, se codifica en 9 bits. Ya que los datos no codificados requieren sólo dos bits (*01* ó *10*), la razón de compresión es $9/2 = 4{,}5$ (la secuencia de datos comprimidos es 4,5 veces más larga que la de los datos sin comprimir; una gran expansión).

2.33: La Figura Sol.14 muestra los modos y el código real generado a partir de las dos líneas.

2.34: La Tabla Sol.15 muestra los pasos de la codificación de la cadena $a_2\,a_2\,a_2\,a_2$. Debido a la alta probabilidad de a_2 las variables `low` y `high` comienzan en valores muy diferentes y se aproximan entre sí lentamente.

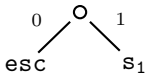
2.35: Puede ser escrito, ya sea como[5] 0,1000..., ya sea como su equivalente: 0,0111....

2.36: En la práctica, el símbolo eof debe ser incluido en la tabla original de frecuencias y probabilidades. Este símbolo es el último en ser codificado y el decodificador se detiene cuando detecta

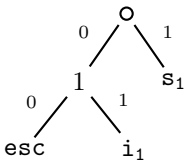
[5]Este número se obtiene cogiendo las partes enteras, resultado de ir multiplicando la parte fraccionaria del número por 2; esto es: $0{,}5 \times 2 = \mathbf{1}{,}0$; $0{,}0 \times 2 = \mathbf{0}{,}0; \ldots \Longrightarrow 0{,}5_{10} = 0{,}1000\ldots_2$.

Árbol inicial:

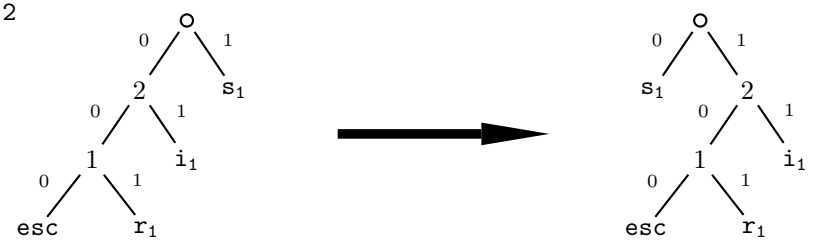
(*a*). Entrada: `s`. Salida: '`s`'.
`esc` $\mathtt{s}_1$



(*b*). Entrada: `i`. Salida: `0'i'`.
`esc` $\mathtt{i}_1$ `1` $\mathtt{s}_1$



(*c*). Entrada: `r`. Salida: `00'r'`.
`esc` $\mathtt{r}_1$ `1` $\mathtt{i}_1$ `2` $\mathtt{s}_1$ $\rightarrow$
`esc` $\mathtt{r}_1$ `1` $\mathtt{i}_1$ $\mathtt{s}_1$ `2`



(*d*). Entrada: ␣. Salida: `100'␣'`.
`esc` $\mathtt{␣}_1$ `1` $\mathtt{r}_1$ `2` $\mathtt{i}_1$ $\mathtt{s}_1$ `3` $\rightarrow$
`esc` $\mathtt{␣}_1$ `1` $\mathtt{r}_1$ $\mathtt{s}_1$ $\mathtt{i}_1$ `2 2`

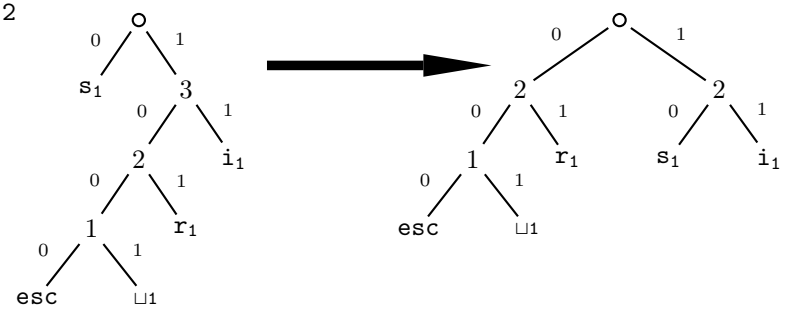
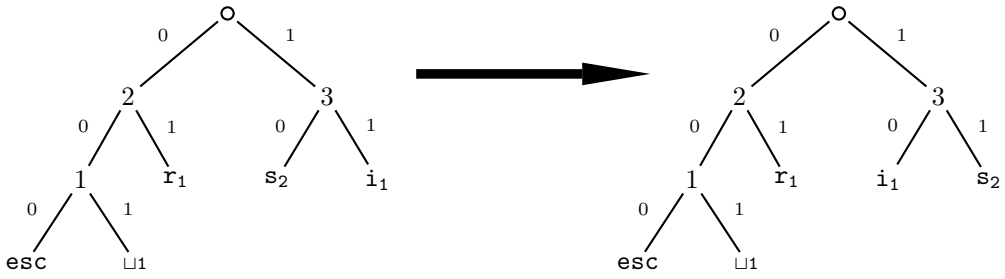
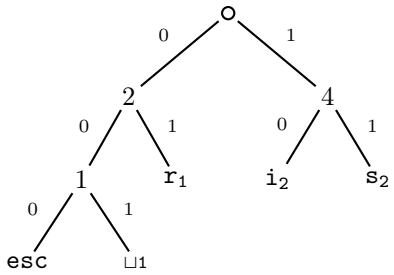


Figura Sol.13: Ejercicio 2.25. Parte I.

(e). Entrada: `s`. Salida: `10`.
`esc` $\sqcup_1$ `1 r`$_1$ `s`$_2$ `i`$_1$ `2 3` $\rightarrow$
`esc` $\sqcup_1$ `1 r`$_1$ `i`$_1$ `s`$_2$ `2 3`



(f). Entrada: `i`. Salida: `10`.
`esc` $\sqcup_1$ `1 r`$_1$ `i`$_2$ `s`$_2$ `2 4`



(g). Entrada: `d`. Salida: `000'd'`.
`esc d`$_1$ `1` $\sqcup_1$ `2 r`$_1$ `i`$_2$ `s`$_2$ `3 4` $\rightarrow$
`esc d`$_1$ `1` $\sqcup_1$ `r`$_1$ `2 i`$_2$ `s`$_2$ `3 4`

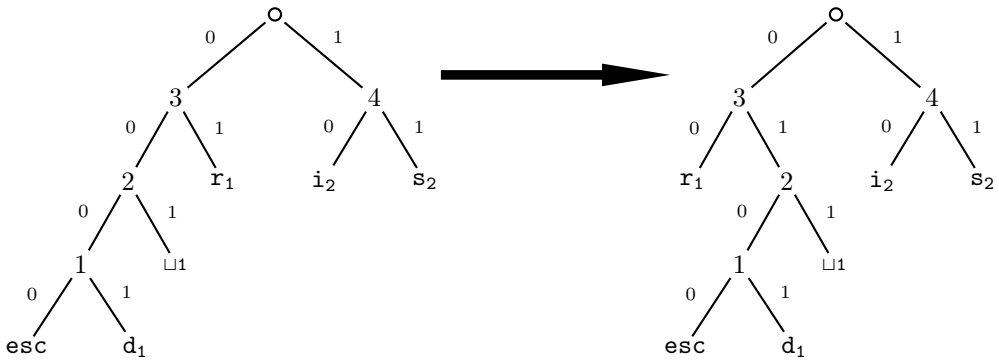
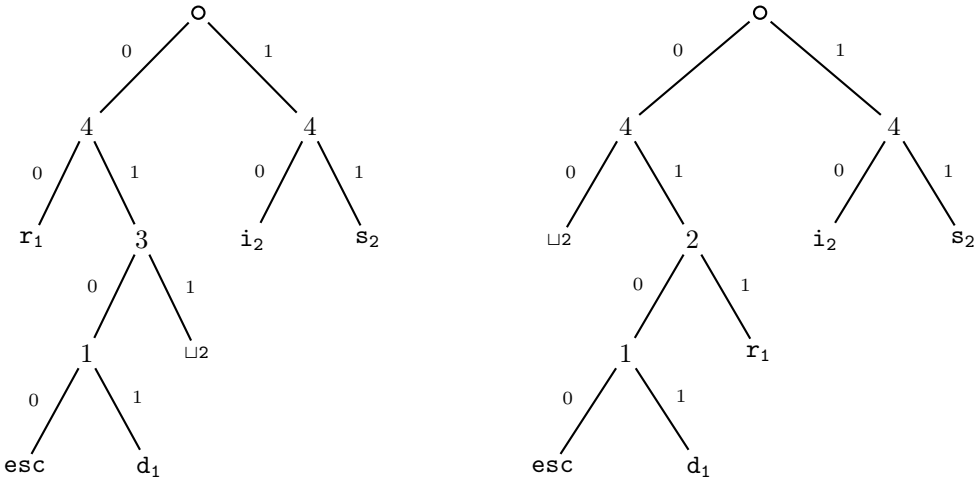


Figura Sol.13: Ejercicio 2.25. Parte II.

(h). Entrada: $\sqcup$. Salida: `011`.
`esc` d_1 1 $\sqcup_2$ r_1 3 i_2 s_2 4 4 $\rightarrow$
`esc` d_1 1 r_1 $\sqcup_2$ 2 i_2 s_2 4 4



(i). Entrada: `i`. Salida: `10`.
`esc` d_1 1 r_1 $\sqcup_2$ 2 i_3 s_2 4 5 $\rightarrow$
`esc` d_1 1 r_1 $\sqcup_2$ 2 s_2 i_3 4 5

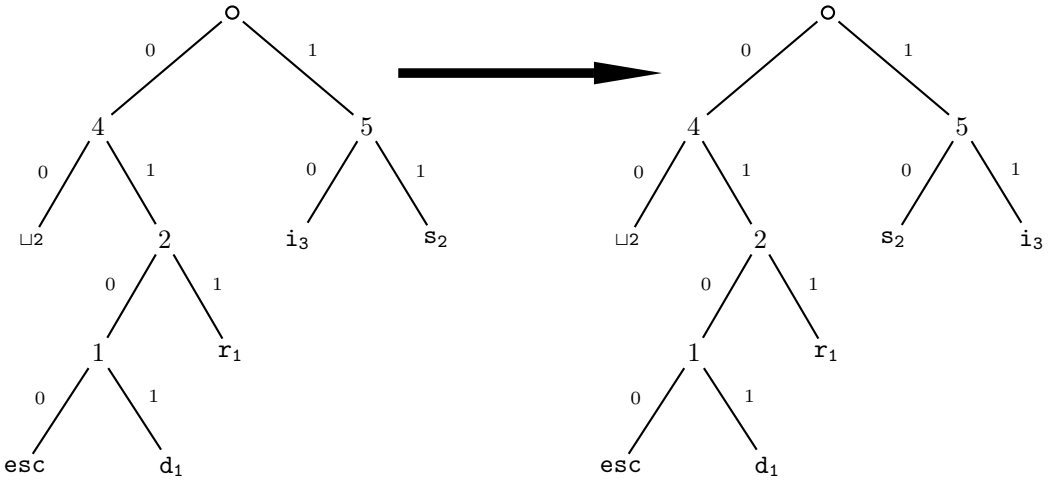
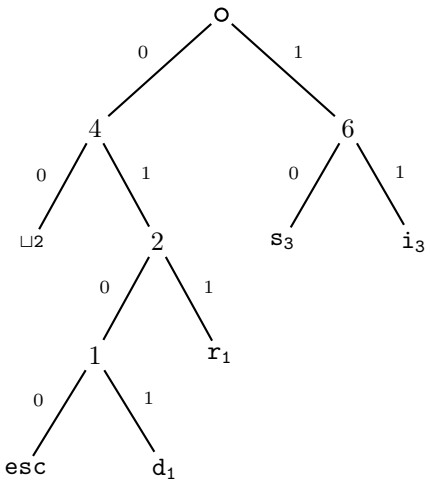


Figura Sol.13: Ejercicio 2.25. Parte III.

(j). Entrada: `s`. Salida: `10`.
`esc` $\mathtt{d}_1$ `1` $\mathtt{r}_1$ $\sqcup_2$ `2` $\mathtt{s}_3$ $\mathtt{i}_3$ `4 6`



(k). Entrada: $\sqcup$. Salida: `00`.
`esc` $\mathtt{d}_1$ `1` $\mathtt{r}_1$ $\sqcup_3$ `2` $\mathtt{s}_3$ $\mathtt{i}_3$ `5 6` $\rightarrow$
`esc` $\mathtt{d}_1$ `1` $\mathtt{r}_1$ `2` $\sqcup_3$ $\mathtt{s}_3$ $\mathtt{i}_3$ `5 6`

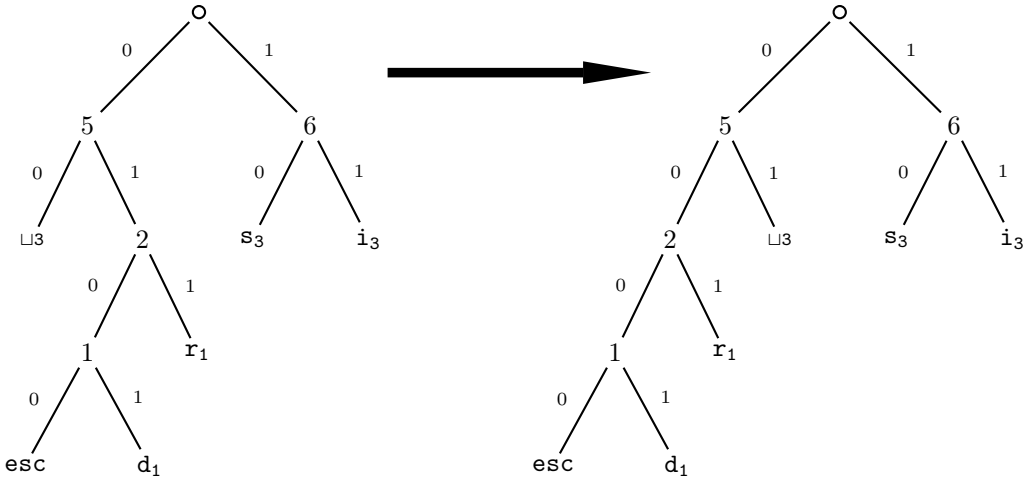


Figura Sol.13: Ejercicio 2.25. Parte IV.

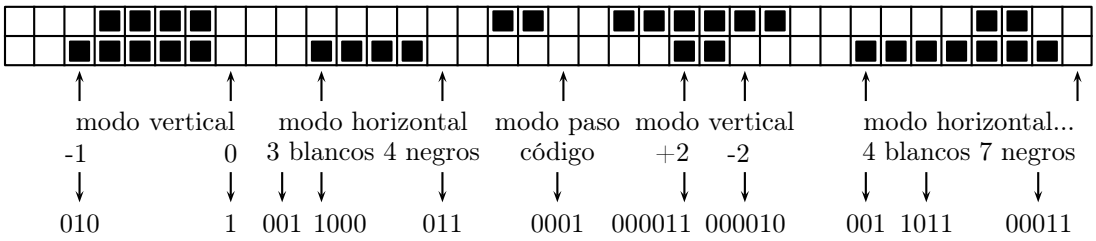


Figura Sol.14: Ejemplo de codificación bidimensional.

a_2	$0{,}0 + (1{,}0 - 0{,}0) \times 0{,}023162 =$	0,023162
	$0{,}0 + (1{,}0 - 0{,}0) \times 0{,}998162 =$	0,998162
a_2	$0{,}023162 + {,}975 \times 0{,}023162 =$	0,04574495
	$0{,}023162 + {,}975 \times 0{,}998162 =$	0,99636995
a_2	$0{,}04574495 + 0{,}950625 \times 0{,}023162 =$	0,06776332625
	$0{,}04574495 + 0{,}950625 \times 0{,}998162 =$	0,99462270125
a_2	$0{,}06776332625 + 0{,}926859375 \times 0{,}023162 =$	0,08923124309375
	$0{,}06776332625 + 0{,}926859375 \times 0{,}998162 =$	0,99291913371875

Tabla Sol.15: Codificación de la cadena $a_2a_2a_2a_2$.

un eof.

2.37: Los pasos de codificación son sencillos (véase el primer ejemplo en la página 121). Comenzamos con el intervalo [0,1). El primer símbolo a_2 reduce el intervalo a [0,4,0,9); el segundo, a [0,6,0,85), el tercero a [0,7,0,825) y el símbolo *eof*, a [0,8125,0,8250). Los valores binarios aproximados del último intervalo son: 0.**110100**00000 y 0.**110100**11001, respectivamente, por lo que seleccionamos el número de 7 bits **110100**0 como nuestro código.
La probabilidad de $a_2\,a_2\,a_2\,eof$ es $(0{,}5)^3 \times 0{,}1 = 0{,}0125$, pero puesto que $-\log_2 0{,}0125 \approx 6{,}322$ se deduce que el tamaño práctico mínimo del código es de 7 bits.

2.38: Tal vez la forma más sencilla de hacerlo es calcular un conjunto de códigos Huffman para los símbolos, utilizando sus probabilidades. Esto convierte cada símbolo en una cadena binaria, por lo que la secuencia de datos de la entrada puede ser codificada por el codificador QM. Después de decodificar la cadena comprimida con el decodificador QM, es necesario un paso adicional para convertir las cadenas binarias resultantes en los símbolos originales.

2.39: Los resultados se muestran en las Tablas Sol.16 y Sol.17. Cuando todos los símbolos son LPS, la salida C siempre apunta al extremo inferior $A\,(1 - Q_e)$ del subintervalo superior (LPS). Cuando los símbolos son MPS, la salida siempre apunta al extremo inferior del subintervalo inferior (MPS), i.e., 0.

2.40: Si el bit actual de la entrada es un LPS, A se reduce a Q_e, que siempre es 0,5 ó menos, en cuyo caso A siempre tiene que ser renormalizado.

2.41: Los resultados se muestran en las Tablas Sol.18 y Sol.19. (Compárese con la respuesta del ejercicio 2.39).

2.42: Los cuatro pasos de la decodificación son los siguientes:

Símbolo	C	A
Inicialmente	0	1
s_1 (LPS)	$0 + 1(1-0{,}5) = 0{,}5$	$1 \times 0{,}5 = 0{,}5$
s_2 (LPS)	$0{,}5 + 0{,}5(1-0{,}5) = 0{,}75$	$0{,}5 \times 0{,}5 = 0{,}25$
s_3 (LPS)	$0{,}75 + 0{,}25(1-0{,}5) = 0{,}875$	$0{,}25 \times 0{,}5 = 0{,}125$
s_4 (LPS)	$0{,}875 + 0{,}125(1-0{,}5) = 0{,}9375$	$0{,}125 \times 0{,}5 = 0{,}0625$

Tabla Sol.16: Codificación de cuatro símbolos con $Q_e = 0{,}5$.

Símbolo	C	A
Inicialmente	0	1
s_1 (MPS)	0	$1 \times (1 - 0{,}1) = 0{,}9$
s_2 (MPS)	0	$0{,}9 \times (1 - 0{,}1) = 0{,}81$
s_3 (MPS)	0	$0{,}81 \times (1 - 0{,}1) = 0{,}729$
s_4 (MPS)	0	$0{,}729 \times (1 - 0{,}1) = 0{,}6561$

Tabla Sol.17: Codificación de 4 símbolos con $Q_e = 0{,}1$.

Símbolo	C	A	Renor. A	Renor. C
Inicialmente	0	1		
s_1 (LPS)	$0 + 1-0{,}5 = 0{,}5$	0,5	1	1
s_2 (LPS)	$1 + 1-0{,}5 = 1{,}5$	0,5	1	3
s_3 (LPS)	$3 + 1-0{,}5 = 3{,}5$	0,5	1	7
s_4 (LPS)	$7 + 1-0{,}5 = 7{,}5$	0,5	1	15

Tabla Sol.18: Renormalización añadida a la Tabla Sol.16.

Símbolo	C	A	Renor. A	Renor. C
Inicialmente	0	1		
s_1 (MPS)	0	$1 - 0{,}1 = 0{,}9$		
s_2 (MPS)	0	$0{,}9 - 0{,}1 = 0{,}8$		
s_3 (MPS)	0	$0{,}8 - 0{,}1 = 0{,}7$	1,4	0
s_4 (MPS)	0	$1{,}4 - 0{,}1 = 1{,}3$		

Tabla Sol.19: Renormalización añadida a la Tabla Sol.17.

- *Paso 1:* $C = 0{,}981$, $A = 1$; la línea divisoria es $A\,(1 - Q_e) = 1\,(1 - 0{,}1) = 0{,}9$, por lo que los subintervalos LPS y MPS son $[0, 0{,}9)$ y $[0{,}9, 1)$, respectivamente. Puesto que C apunta al subintervalo superior, se decodifica como un LPS. El nuevo C es $0{,}981 - 1\,(1 - 0{,}1) = 0{,}081$ y la nueva A es $1 \times 0{,}1 = 0{,}1$.
- *Paso 2:* $C = 0{,}081$, $A = 0{,}1$; la línea divisoria es $A\,(1 - Q_e) = 0{,}1\,(1 - 0{,}1) = 0{,}09$, por lo que los subintervalos LPS y MPS son $[0, 0{,}09)$ y $[0{,}09, 0{,}1)$, respectivamente y se decodifica como un MPS. C no cambia y la nueva A es $0{,}1\,(1 - 0{,}1) = 0{,}09$.
- *Paso 3:* $C = 0{,}081$, $A = 0{,}09$; la línea divisoria es $A\,(1 - Q_e) = 0{,}09\,(1 - 0{,}1) = 0{,}081$, por lo que los subintervalos LPS y MPS son $[0, 0{,}081)$ y $[0{,}081, 0{,}09)$, respectivamente y se decodifica como un LPS. El nuevo C es $0{,}081 - 0{,}09\,(1 - 0{,}1) = 0$ y la nueva A es $0{,}09 \times 0{,}1 = 0{,}009$.
- *Paso 4:* $C = 0$, $A = 0{,}009$; la línea divisoria es $A\,(1 - Q_e) = 0{,}009\,(1 - 0{,}1) = 0{,}0081$, por lo que los subintervalos LPS y MPS son $[0, 0{,}0081)$ y $[0{,}0081, 0{,}009)$, respectivamente y se decodifica como un MPS. C no cambia y la nueva A es $0{,}009\,(1 - 0{,}1) = 0{,}0081$.

2.43: En la práctica, un codificador puede codificar otros textos distintos al inglés, como un lenguaje extranjero o el código fuente de un programa de ordenador. Los acrónimos, como QED y las abreviaturas, como qwerty, son también buenos ejemplos. Incluso en inglés hay algunos ejemplos en los que a una q no le sigue una u, como en esta frase. (El autor se ha dado cuenta que los escritores de ciencia ficción tienden a usar palabras que no parecen inglesas, como Qaal, para nombrar a los personajes en sus obras.)

2.44: El número de contextos de orden 2 y de orden 3 para un alfabeto de tamaño $2^8 = 256$ es $256^2 = 65\,536$ y $256^3 = 16\,777\,216$ respectivamente. El primero es manejable, mientras que el último es quizás demasiado grande para una aplicación práctica, a menos que se utilice una estructura de base de datos sofisticada o que el codificador se deshaga de los datos más antiguos de vez en cuando.
Para un alfabeto pequeño, pueden usarse grandes valores de N. Para un alfabeto de 16 símbolos hay $16^4 = 65\,536$ contextos de orden 4 y $16^6 = 166\,777\,216$ contextos de orden 6.

2.45: Un ejemplo práctico de un alfabeto de 16 símbolos es una imagen en color o en escala de grises con 4 bits por píxel. Cada símbolo es un píxel, y hay $2^4 = 16$ símbolos diferentes

2.46: Un archivo objeto generado por un compilador o un ensamblador normalmente tiene varias partes distintas: las instrucciones en lenguaje máquina, la tabla de símbolos, los bits de reubicación y las constantes. Estas secciones pueden tener diferentes distribuciones de bits.

2.47: El alfabeto tiene que ser extendido, en tal caso, para incluir un símbolo más. Si el alfabeto original constaba de todos los posibles 256 bytes de 8 bits, ha de ampliarse a símbolos de 9 bits, y debe incluir 257 valores.

2.48: La Tabla Sol.20 muestra los grupos generados en ambos casos y deja claro por qué se asignan estas probabilidades en particular.

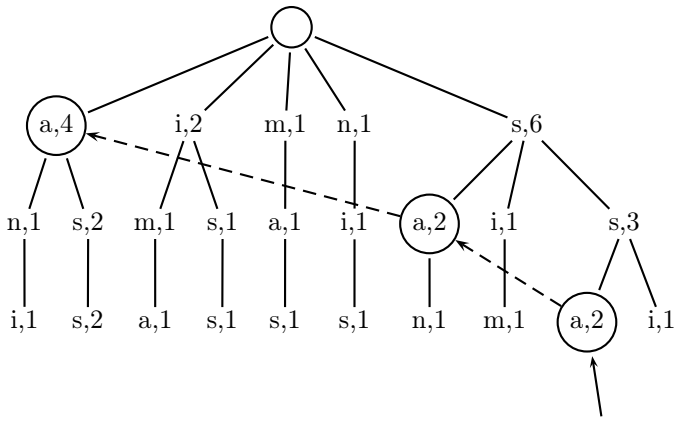
2.49: La `d` se añade al contexto de orden 0 con frecuencia 1. La frecuencia de `escape` se incrementa desde 5 hasta 6, aumentando las frecuencias totales desde 19 hasta 21. La probabilidad asignada a la nueva `d` es, por consiguiente de $1/21$, y la del símbolo `escape` es de $6/21$. Todas las otras probabilidades se reducen desde $x/19$ hasta $x/21$.

2.50: El nuevo `d` requiere el cambio desde el orden 2 hasta el orden 0, enviando dos escapes que gastan 1 y 1,32 bits. La `d` se encuentra ahora en el orden 0 con una probabilidad de $1/21$, por lo que se codifica en 4,39 bits. El número total de bits necesarios para codificar el segundo `d` es, por lo tanto: $1 + 1{,}32 + 4{,}39 = 6{,}71$, aún mayor que 5.

Contexto	f	p
`abc` $\rightarrow$ `x`	10	$^{10}/_{11}$
`Esc`	1	$^{1}/_{11}$
$Total =$	11	

Contexto	f	p
`abc` $\rightarrow$ $\mathtt{a}_1$	1	$^{1}/_{20}$
`abc` $\rightarrow$ $\mathtt{a}_2$	1	$^{1}/_{20}$
`abc` $\rightarrow$ $\mathtt{a}_3$	1	$^{1}/_{20}$
`abc` $\rightarrow$ $\mathtt{a}_4$	1	$^{1}/_{20}$
`abc` $\rightarrow$ $\mathtt{a}_5$	1	$^{1}/_{20}$
`abc` $\rightarrow$ $\mathtt{a}_6$	1	$^{1}/_{20}$
`abc` $\rightarrow$ $\mathtt{a}_7$	1	$^{1}/_{20}$
`abc` $\rightarrow$ $\mathtt{a}_8$	1	$^{1}/_{20}$
`abc` $\rightarrow$ $\mathtt{a}_9$	1	$^{1}/_{20}$
`abc` $\rightarrow$ $\mathtt{a}_{10}$	1	$^{1}/_{20}$
`Esc`	10	$^{1}/_{20}$
$Total =$	20	

Tabla Sol.20: Datos estables vs. datos variables.



14.‘a’

Figura Sol.21: El trie final de “`assanissimassa`”.

2.51: Los tres primeros casos no cambian. Todavía codifican el símbolo `a` con 1, 1,32 y 6,57 bits, que es inferior a los 8 bits necesarios para almacenarlo sin compresión utilizando un alfabeto de 256 símbolos. El caso 4 es diferente, ya que el `d` se codifica ahora con una probabilidad de $^{1}/_{256}$, produciendo un gasto de 8 bits en lugar de 4,8. El número total de bits necesarios para codificar `d` en el caso 4 es ahora: $1 + 1{,}32 + 1{,}93 + 8 = 12{,}25$.

2.52: El trie final se muestra en la Figura Sol.21.

2.53: La probabilidad es, por supuesto:

$$1 - P_e\left(b_{t+1} = 1 \mid b_1^t\right) = 1 - \frac{b + {}^{1}/_{2}}{a + b + 1} = \frac{a + {}^{1}/_{2}}{a + b + 1}.$$

2.54: Para la primera cadena de un solo bit, tiene un sufijo 00, por lo que la probabilidad de la hoja 00 es $P_e(1, 0) = {}^{1}/_{2}$. Esto es igual que la probabilidad de aparición de la cadena 0 sin ningún sufijo. Para la segunda cadena, cada uno de los dos bits cero tiene el sufijo 00, por lo que la probabilidad

de la hoja 00 es $P_e\,(2,0) = {}^3\!/_8 = 0{,}375$. Ésta es mayor que la probabilidad de aparición de la cadena 00 sin ningún sufijo: 0,25. De forma similar, las probabilidades de las otras tres cadenas son: $P_e\,(3,0) = {}^5\!/_8 \approx 0{,}625$, $P_e\,(4,0) = {}^{35}\!/_{128} \approx 0{,}273$ y $P_e\,(5,0) = {}^{63}\!/_{256} = 0{,}246$. Como las cadenas se hacen más largas, sus probabilidades se hacen más pequeñas; pero son mayores que las probabilidades sin sufijo. Tener un sufijo 00 por tanto, aumenta la probabilidad de tener cadenas de ceros que le sigan.

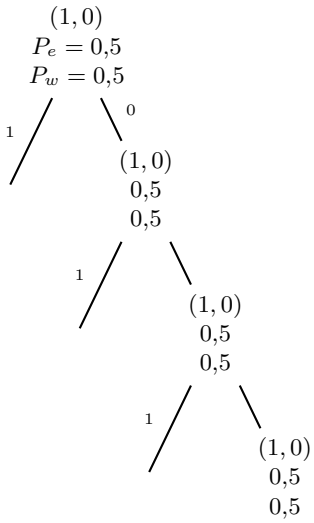
2.55: Los cuatro árboles se muestran en la Figura Sol.22a-d. La probabilidad ponderada de que el siguiente bit sea un cero, dados los tres ceros que acaban de ser generados es de 0,5. La probabilidad ponderada de aparición de dos ceros consecutivos, dado el sufijo 000 es 0,375, más alta que sin el sufijo: 0,25.

3.1: El tamaño de la secuencia de salida es $N\,[48 - 28P] = N\,[48 - 25{,}2] = 22{,}8N$. El tamaño de la cadena de entrada es, como antes, $40N$. El factor de compresión es, por lo tanto, ${}^{40}\!/_{22{,}8} \approx 1{,}75$.

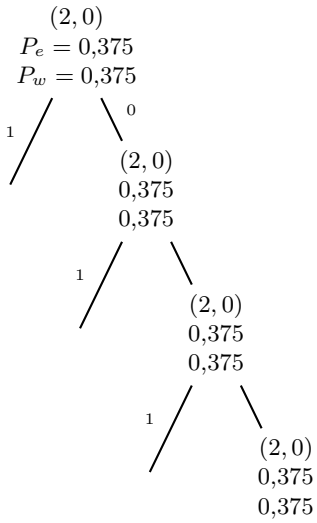
3.2: La lista no tiene más de 256 entradas, cada una compuesta por un byte y un contador. Un byte ocupa ocho bits, pero el contador depende del tamaño y el contenido del archivo que se está comprimiendo. Si el archivo tiene una redundancia alta, unos pocos bytes pueden necesitar contadores que permitan almacenar valores tan grandes como la longitud del archivo, mientras que otros bytes sólo necesitan almacenar valores muy bajos. Por otra parte, si el archivo es más bien aleatorio, entonces cada byte distinto precisa de contadores de aproximadamente el mismo tamaño, pues el número de elementos distintos será similar.
Así, el primer paso en la organización de la lista es reservar espacio suficiente para cada campo "contador" para que pueda contener el valor de cuenta máximo posible. Denotemos la longitud del archivo con L y encontremos el entero positivo k que satisface $2^{k-1} < L < 2^k$. Por lo tanto, L es un número de k bits. Si k no es ya un múltiplo de 8, lo incrementamos hasta el siguiente múltiplo de 8. Ahora denotamos $k = 8m$, y asignamos m bytes para cada campo "contador".
Una vez que el archivo ha sido introducido y procesado, y la lista ha sido ordenada, examinamos el mayor de los contadores. El valor puede ser grande y ocupar los m bytes, o puede ser más pequeño. Si suponemos que el mayor recuento ocupa n bytes (donde $n \leq m$), podemos almacenar cada uno de los otros recuentos en n bytes.
Cuando la lista se escribe en el archivo comprimido, para generar el diccionario, se escribe primero su longitud, s, en un byte; s es el número de bytes distintos en el archivo original. A continuación se escribe n, seguido por s grupos —cada uno con uno de los bytes de datos distintos, seguido por un recuento de su frecuencia (de n bytes)—. Observe que el valor de n debe ser bastante pequeño y debe caber en un solo byte. Si no es así, entonces es mayor que 255, lo que implica que el mayor recuento no cabe en 255 bytes, lo que a su vez implica que el archivo tiene una longitud L mayor que $2^{255} \approx 10^{76}$ bytes.
Una alternativa es comenzar con s, seguido por n, y a continuación, por los s bytes de datos distintos, seguidos por los $n \times s$ bytes de sus contadores. La última parte también podría estar en forma comprimida, porque sólo unos pocos contadores necesitan albergar un valor más grande y ocupan los n bytes totalmente. La mayoría de los recuentos pueden ser pequeños y ocupar sólo uno o dos bytes, lo que implica que muchos de los $n \times s$ bytes de los contadores serán cero, lo que produce una alta redundancia y, por lo tanto, una buena compresión.

3.3: La respuesta directa es: El decodificador no lo sabe, pero no tiene por qué saberlo. El decodificador simplemente lee los tokens y utiliza cada desplazamiento para localizar una cadena de texto sin necesidad de saber si ésta era el primer patrón de coincidencia o el segundo.

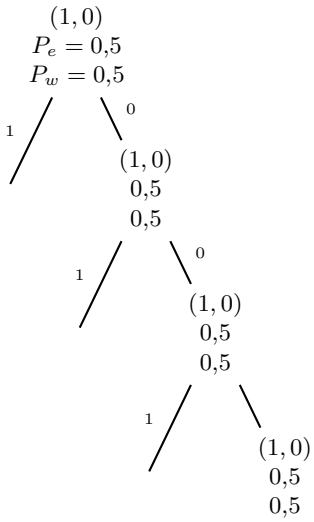
3.4: El siguiente paso localiza el espacio y codifica la cadena ␣e;



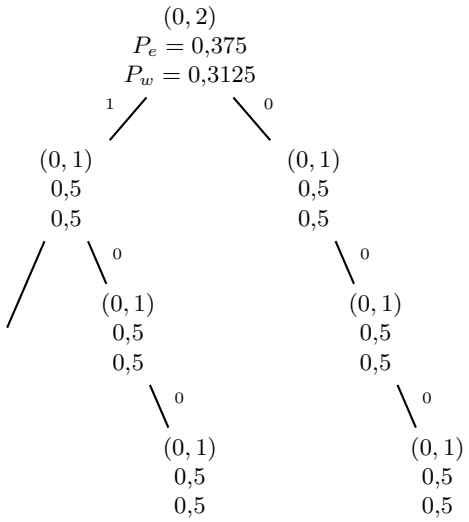
(a) 000|0



(b) 000|00



(c) 000|1



(d) 000|11

Figura Sol.22: Árboles de contexto para 000|0, 000|00 , 000|1 y 000|11.

Diccionario		Token	Diccionario		Token
15	"␣t"	(4, "t")	21	"␣si"	(19, "i")
16	"e"	(0, "e")	22	"c"	(0, "c")
17	"as"	(8, "s")	23	"k"	(0, "k")
18	"es"	(16, "s")	24	"␣se"	(19, "e")
19	"␣s"	(4, "s")	25	"al"	(8, "l")
20	"ea"	(16, "a")	26	"s(eof)"	(1, "(eof)")

Tabla Sol.23: Los 12 pasos siguientes en el ejemplo de LZ78.

sir␣sid	␣eastman␣easily␣	⇒	(0,0,"e")
sir␣sid␣e	astman␣easily␣te	⇒	(0,0,"a")

y el siguiente paso no encuentra nada y codifica la a.

3.5: Los dos primeros caracteres —CA, en las posiciones 17 a 18— son una repetición de la cadena situada en las posiciones 9 a 10, por lo que se codifica como una cadena de longitud 2 y desplazamiento $18 - 10 = 8$. Los dos caracteres siguientes —AC, en las posiciones 19 a 20— son una repetición de la cadena localizada en las posiciones 8 a 9, por lo que se se codifica como una cadena de longitud 2 y desplazamiento $20 - 9 = 11$.

3.6: El decodificador interpreta el primer 1 del marcador de finalización como el comienzo de un token. El segundo 1, se interpreta como el prefijo de un desplazamiento de 7 bits. Los 7 bits siguiente son 0, e identifican el marcador de finalización como tal, ya que un desplazamiento "normal" no puede ser cero.

3.7: Es sencillo. Los pasos restantes se muestran en la Tabla Sol.23.

3.8: La Tabla Sol.24 muestra los últimos tres pasos.

p_src	3 car.	Índice hash	P	Salida	Salida binaria
11	h␣t	7	*nada* $\rightarrow 11$	h	01101000
12	␣th	5	$5 \rightarrow 12$	$4, 7$	0000 | 0011 | 00000111
16	ws			ws	01110111 | 01110011

Tabla Sol.24: Últimos pasos para codificar "that␣thatch␣thaws".

La cadena comprimida final consiste en 1 palabra de control seguida de 11 ítems (9 literales y 2 ítems de copia —en negrita—):

0000010010000000 | 01110100 | 01101000 | 01100001 | 01110100 | 00100000
| **0000** | **0011** | **00000101** | 01100011 | 01101000
| **0000** | **0011** | **00000111** | 01110111 | 01110011.

3.9: Un ejemplo, es una utilidad de compresión para un ordenador personal que mantiene todos los archivos —o grupos de archivos— en el disco duro en formato comprimido, para ahorrar espacio. Esta utilidad debe ser transparente para el usuario —no tiene que ser consciente de su ejecución—; debe descomprimir automáticamente un archivo cada vez que se abra y comprimirlo cuando se cierre. Con el fin de conseguir la transparencia, tal utilidad debe ser rápida, dejando en segundo plano la razón de compresión obtenida.

I	¿En diccionario?	Nueva entrada	Salida	I	¿En diccionario?	Nueva entrada	Salida
a	S			s␣	N	263-s␣	115 (s)
al	N	256-al	97 (a)	␣	S		
l	S			␣a	N	264-␣a	32 (␣)
lf	N	257-lf	108 (l)	a	S		
f	S			al	S		
f␣	N	258-f␣	102 (f)	alf	N	265-alf	256 (al)
␣	S			f	S		
␣e	N	259-␣e	32 (␣)	fa	N	266-fa	102 (f)
e	S			a	S		
ea	N	260-ea	101 (e)	al	S		
a	S			alf	S		
at	N	261-at	97 (a)	alfa	N	267-alfa	265 (alf)
t	S			a	S		
ts	N	262-ts	116 (t)	a, eof	N		97 (a)
s	S						

Tabla Sol.25: Codificación LZW de "alf␣eats␣alfalfa".

3.10: La Tabla Sol.25 resume los pasos. La salida emitida por el codificador es:

97 (a), 108 (l), 102 (f), 32 (␣), 101 (e), 97 (a), 116 (t), 115 (s), 32 (␣), 256 (al), 102 (f), 265 (alf), 97 (a),

y las nuevas entradas añadidas al diccionario:

(256 : al), (257 : lf), (258 : f␣), (259 : ␣e), (260 : ea), (261 : at), (262 : ts), (263 : s␣), (264 : ␣a), (265 : alf), (266 : fa), (267 : alfa).

3.11: El codificador introduce la primera a en I, busca y encuentra a en el diccionario. Introduce la siguiente a, pero observa que Ix —que ahora es aa— no está en el diccionario. Por consiguiente, el codificador añade la cadena aa en la entrada 256 del diccionario y produce como salida el token 97 (a). La variable I se inicializa con el segundo a. Introduce la tercera a, por lo que Ix es la cadena aa —que ahora está en el diccionario—. I se convierte en esta cadena, y el codificador introduce la cuarta a. Ahora Ix es aaa, que no está en el diccionario. En consecuencia, el codificador añade la cadena aaa al diccionario en la entrada 257 y genera la salida 256 (aa). I se inicializa con la cuarta a. La continuación de este proceso es sencilla.

El resultado es que las cadenas a, aa, aaa,..., se agregan al diccionario en las entradas 256, 257, 258,..., y la salida es:

$$97\,(\mathtt{a})\,, 256\,(\mathtt{aa})\,, 257\,(\mathtt{aaa})\,, 258\,(\mathtt{aaaa})\,, \ldots .$$

La salida está formada por punteros que apuntan a cadenas cada vez más largas de aes. Por consiguiente, los primeros k punteros apuntan a cadenas cuya longitud total es: $1+2+\cdots+k = {}^{(k+k^2)}/_{2}$.

Suponiendo un flujo de entrada que consta de un millón de aes, podemos hallar el tamaño de la secuencia de datos comprimidos resolviendo la ecuación cuadrática ${}^{(k+k^2)}/_{2} = 1000000$ —para k desconocido—. La solución es $k \approx 1414$. La entrada original —de 8 millones de bits— se comprime, por lo tanto, en 1414 punteros —cada uno de al menos 9 bits (y en la práctica, probablemente de 16 bits) de longitud—. El factor de compresión es, pues, o bien ${}^{8M}/_{(1414\times 9)} \approx 628,6$ o bien ${}^{8M}/_{(1414\times 16)} \approx 353,6$.

Este es un resultado impresionante, pero este tipo de flujo de datos de entrada son poco

frecuentes (nótese que esta entrada en particular puede ser comprimida más eficientemente, generando como secuencia de salida la cadena "1000000 `a`" —y sin utilizar LZW—).

3.12: Simplemente síganse los pasos descritos sobre la decodificación en el texto. Los resultados son los siguientes:

1. Se introduce 97. Está en el diccionario, por lo que se inicializa I = `a` y se saca `a`. La cadena `ax` necesita ser guardada en el diccionario, pero x es aún desconocida.
2. Se introduce 108. Está en el diccionario, por lo que se inicializa J = `l` y se saca `l`. Se guarda `al` en la entrada 256. Se inicializa I = `l`.
3. Se introduce 102. Está en el diccionario, por lo que se inicializa J = `f` y se saca `f`. Se guarda `lf` en la entrada 257. Se inicializa I = `f`.
4. Se introduce 32. Está en el diccionario, por lo que se inicializa J = ␣ y se saca ␣. Se guarda `f`␣ en la entrada 258. Se inicializa I = ␣.
5. Se introduce 101. Está en el diccionario, por lo que se inicializa J = `e` y se saca `e`. Se guarda ␣`e` en la entrada 259. Se inicializa I = `e`.
6. Se introduce 97. Está en el diccionario, por lo que se inicializa J = `a` y se saca `a`. Se guarda `ea` en la entrada 260. Se inicializa I = `a`.
7. Se introduce 116. Está en el diccionario, por lo que se inicializa J = `t` y se saca `t`. Se guarda `at` en la entrada 261. Se inicializa I = `t`.
8. Se introduce 115. Está en el diccionario, por lo que se inicializa J = `s` y se saca `s`. Se guarda `ts` en la entrada 262. Se inicializa I = `s`.
9. Se introduce 32. Está en el diccionario, por lo que se inicializa J = ␣ y se saca ␣. Se guarda `s`␣ en la entrada 263. Se inicializa I = ␣.
10. Se introduce 256. Está en el diccionario, por lo que se inicializa J = `al` y se saca `al`. Se guarda ␣`a` en la entrada 264. Se inicializa I = `al`.
11. Se introduce 102. Está en el diccionario, por lo que se inicializa J = `f` y se saca `f`. Se guarda `alf` en la entrada 265. Se inicializa I = `f`.
12. Se introduce 265. Acaba de almacenarse el diccionario, por lo que se inicializa J = `alf` y se saca `alf`. Se guarda `fa` en la entrada 266 del diccionario. Se inicializa I = `alf`.
13. Se introduce 97. Está en el diccionario, por lo que se inicializa J = `a` y se saca `a`. Se guarda `alfa` en la entrada 267 (a pesar de que nunca será utilizada). Se inicializa I = `a`.
14. Finalmente, se lee `eof`.

3.13: Suponemos que el diccionario se inicializa con exactamente las dos entradas: (1 : `a`) y (1 : `b`). El codificador produce como salida:

$$1\,(\texttt{a})\,, 2\,(\texttt{b})\,, 3\,(\texttt{ab})\,, 5\,(\texttt{aba})\,, 4\,(\texttt{ba})\,, 7\,(\texttt{bab})\,, 6\,(\texttt{abab})\,, 9\,(\texttt{ababa})\,, 8\,(\texttt{baba})\,, \ldots$$

y agrega las nuevas entradas (3 : `ab`), (4 : `ba`), (5 : `aba`), (6 : `abab`), (7 : `bab`), (8 : `baba`), (9 : `ababa`), (10 : `ababab`), (11 : `babab`),... al diccionario. Este comportamiento tan regular puede ser analizado y se pueden predecir tanto el k-ésimo puntero de salida y como la entrada correspondiente del diccionario, pero el esfuerzo probablemente no vale la pena.

3.14: La respuesta del Ejercicio 3.11 muestra la relación entre el tamaño del archivo comprimido y el tamaño de la cadena del diccionario más larga en el "peor caso" (la entrada que crea las cadenas más largas). Para un flujo de entrada de 1 Mbyte, habrá 1 414 cadenas en el diccionario; la más larga de todas consta de 1 414 símbolos.

Paso	Entrada	Salida	S	Añadir al diccionario	S′
	swiss␣miss				
1	s	115	s	—	s
2	w	119	w	256-sw	w
3	i	105	i	257-wi	i
4	s	115	s	258-is	s
5	s	115	s	259-ss	s
6	␣	32	␣	260-s␣	␣
7	m	109	m	261-␣m	m
8	is	258	is	262-mis	is
9	s	115	s	263-iss	s

Tabla Sol.26: Compresión LZMW de "swiss␣miss".

Paso	Entrada	Salida	S	Añadir al diccionario	S′
	yabbadabbadabbadoo				
1	y	121	y	—	y
2	a	97	a	256-ya	a
3	b	98	b	257-ab	b
4	b	98	b	258-bb	b
5	a	97	a	259-ba	a
6	d	100	d	260-ad	d
7	ab	257	ab	261-dab	ab
8	ba	259	ba	262-abba	ba
9	dab	261	dab	263-badab	dab
10	ba	259	ba	264-dabba	ba
11	d	100	d	265-bad	d
12	o	111	o	266-do	o
13	o	111	o	267-oo	o

Tabla Sol.27: Compresión LZMW de "yabbadabbadabbadoo".

3.15: El proceso es sencillo (Tabla Sol.26) pero no muy eficiente, ya que sólo se utiliza una frase de dos símbolos del diccionario.

3.16: La Tabla Sol.27 muestra todos los pasos. A pesar de la corta cadena de entrada, el resultado es bastante bueno (13 códigos para comprimir 18 símbolos) debido a que la entrada contiene concentraciones de aes y bes.

3.17: Los pasos son los siguientes:

1. El codificador comienza desplazando los dos primeros símbolos —xy— en el búfer de búsqueda, sacandolos como literales e inicializando todas las posiciones de la tabla de índices al puntero nulo.

2. El símbolo actual es a (la primera a) y el contexto es xy. Éste se pasa a la función hash, que proporciona un valor —digamos, 5—; pero la ubicación 5 de la tabla de índices contiene un puntero nulo, por lo que P es nulo. La posición 5 se actualiza de manera que apunte a la primera a, la cual sale entonces como un literal. Los datos del buffer del codificador se desplazan hacia la izquierda.

3. El símbolo actual es la segunda `a` y el contexto es `ya`. Éste se pasa a la función hash, que proporciona un valor —digamos, 1—; pero la ubicación 1 de la tabla de índices contiene un puntero nulo, por lo que `P` es nulo. La posición 1 se actualiza de manera que apunte a la segunda `a`, la cual sale entonces como un literal. Los datos del buffer del codificador se desplazan hacia la izquierda.

4. El símbolo actual es la tercera `a` y el contexto es `aa`. Éste se pasa a la función hash, que proporciona un valor —digamos, 2—; pero la ubicación 2 de la tabla de índices contiene un puntero nulo, por lo que `P` es nulo. La posición 2 se actualiza de manera que apunte a la tercera `a`, la cual sale entonces como un literal. Los datos del buffer del codificador se desplazan hacia la izquierda.

5. El símbolo actual es la cuarta `a` y el contexto es `aa`. Sabemos que desde por el paso 4 que al pasarlo a la fución hash, ésta proporciona el valor 2, y la posición 2 de la tabla de índices apunta a la tercera `a`. La ubicación 2 se actualiza de manera que apunte a la cuarta `a`, y el codificador trata de emparejar la cadena que comienza en la tercera `a` con la que se encuentra a partir de la cuarta `a`. Suponiendo que el buffer de preanálisis está lleno de `a`es, el número de elementos emparejados —L— será el tamaño de este buffer. El valor codificado de L se escribe en la secuencia de datos comprimidos, y los elementos del buffer se desplazan L posiciones hacia la izquierda.

6. Si la secuencia de datos de entrada original es larga, se desplazarán más `a`es en el buffer de preanálisis, y este paso también producirá un emparejamiento de longitud L. Si sólo quedan n `a`es en la secuencia de entrada, éstos serán emparejados, y se ofrecerá como salida el valor codificado de n.

   La cadena comprimida constará de los tres literales `x`, `y`, y `a`, seguidos por —tal vez varios valores de— L, y posiblemente terminará con un valor pequeño.

3.18: El T por ciento de la secuencia de datos comprimidos está formado por literales, algunos aparecen consecutivamente (y por lo tanto, consiguen el flag "1" para dos literales —medio bit por literal—) y otros, seguidos por una longitud de emparejamiento (y por lo tanto, obtienen el flag "01" —un bit por literal—). Suponemos que dos tercios de los literales aparecen de forma consecutiva y un tercio son seguidos por longitudes de emparejamiento. En consecuencia, el número total de bits de flag creados para los literales es:

$$\frac{2}{3}T \times 0{,}5 + \frac{1}{3}T \times 1.$$

Con un argumento similar para longitud de emparejamiento se obtiene:

$$\frac{2}{3}(1-T) \times 2 + \frac{1}{3}(1-T) \times 1$$

para hallar el número total de bits de flag. Podemos escribir la ecuación:

$$\frac{2}{3}T \times 0{,}5 + \frac{1}{3}T \times 1 + \frac{2}{3}(1-T) \times 2 + \frac{1}{3}(1-T) \times 1 = 1,$$

que tiene como solución $T = 2/3$. Esto significa que si dos terceras partes de los ítems de la secuencia de datos comprimidos son literales, debería haber —en promedio— un bit de flag por ítem. Un número mayor de literales se traduce en menos bits de flag.

3.19: Los tres primeros unos indican seis literales. El 01 siguiente indica un literal —`b`— seguido por una longitud de emparejamiento (de 3). El 10 es el código de la longitud de emparejamiento: 3, y el último 1 indica dos literales más (`x` e `y`).

43210	Gray	43210	Gray	43210	Gray	43210	Gray
00000	00000	0**1000**	01100	**10000**	11000	1**1000**	10100
0000**1**	00001	0100**1**	01101	1000**1**	11001	1100**1**	10101
000**10**	00011	010**10**	01111	100**10**	11011	110**10**	10111
0001**1**	00010	0101**1**	01110	1001**1**	11010	1101**1**	10110
00**100**	00110	01**100**	01010	10**100**	11110	11**100**	10010
0010**1**	00111	0110**1**	01011	1010**1**	11111	1110**1**	10011
001**10**	00101	011**10**	01001	101**10**	11101	111**10**	10001
0011**1**	00100	0111**1**	01000	1011**1**	11100	1111**1**	10000

Tabla Sol.29: Primeros 32 binarios y códigos de Gray.

```
a=linspace(0,31,32); b=bitshift(a,-1);
b=bitxor(a,b); dec2bin(b)
```

Código en Matlab para la Tabla Sol.29.

4.1: Una imagen sin redundancia no es siempre aleatoria. La definición de redundancia (Sección 2.1) nos dice que una imagen en la que cada color aparece con la misma frecuencia no tiene redundancia (estadísticamente); sin embargo, no es necesariamente aleatoria y puede ser incluso interesante y/o útil.

4.2: La Figura Sol.28 muestra dos matrices de 32×32. La primera, a, con valores aleatorios (y, por lo tanto, descorrelacionados) y la segunda, b, es su inversa (y, por tanto, con valores correlacionados). Se muestran también sus matrices de covarianza y es obvio que la matriz $cov\,(a)$ está cercana a la diagonal, mientras que la matriz $cov\,(b)$ está lejos de la diagonal. También se muestra el código en Matlab para esta figura.

4.3: Los resultados se muestran en la Tabla Sol.29 junto con el código en Matlab utilizado para calcularla: `linspace` crea en `a` un vector lineal espaciado de 32 elementos entre 0 y 31 (ambos inclusive); `bitshift` convierte el vector `a` en otro `b` de valores desplazados un bit a la derecha (división por 2); `bitxor` hace la operación `XOR` (OR eXclusivo) entre los elementos de igual posición de los vectores `a` y `b`, y la guarda en `b`; y `dec2bin` convierte el vector `b` —que está en base 10— a binario.

4.4: Una característica es la manera regular en la que cada uno de los cinco bits de código alterna periódicamente entre 0 y 1. Es fácil escribir un programa que establezca los cinco bits a 0, invierta el bit de más a la derecha —después de haber calculado los dos códigos—, e invierta cualquiera de los otros cuatro bits de código de la mitad del período de su vecino inmediato a la derecha. Otra característica es el hecho de que la segunda mitad de la tabla es una imagen especular de la primera mitad, pero con el bit más significativo a uno. Después de calcular la primera mitad de la tabla, utilizando cualquier método, esta simetría puede ser utilizada para calcular rápidamente la segunda mitad.

4.5: La Figura Sol.30 es una representación de *código angular de rueda*[6] de la representación de 4 bits y 6 bits de los códigos RGC (parte a) y los códigos binarios de 4 bits y 6 bits (parte b). Los planos de bit individuales se muestran como anillos, con los bits más significativos en la parte más interior del anillo. Es fácil ver que la frecuencia angular máxima de RGC es la mitad que la

[6] Angular code wheel

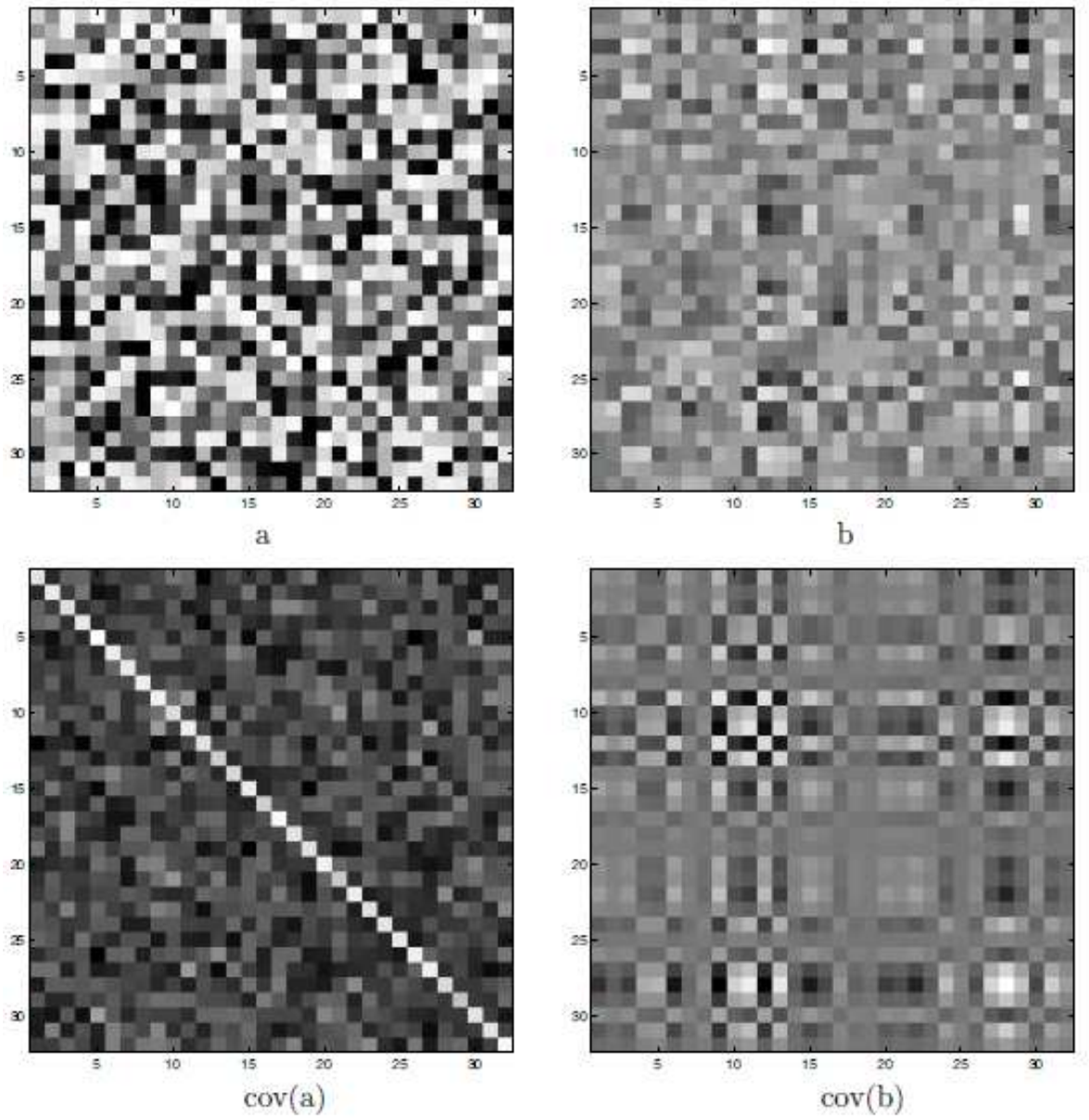


Figura Sol.28: Matrices de covarianza de valores correlacionados y descorrelacionados.

```
a=rand(32); b=inv(a);
figure(1), imagesc(a),colormap(gray); axis square
figure(2), imagesc(b), colormap(gray); axis square
figure(3), imagesc(cov(a)), colormap(gray); axis square
figure(4), imagesc(cov(b)), colormap(gray); axis square
```

Código Matlab para la Figura Sol.28.

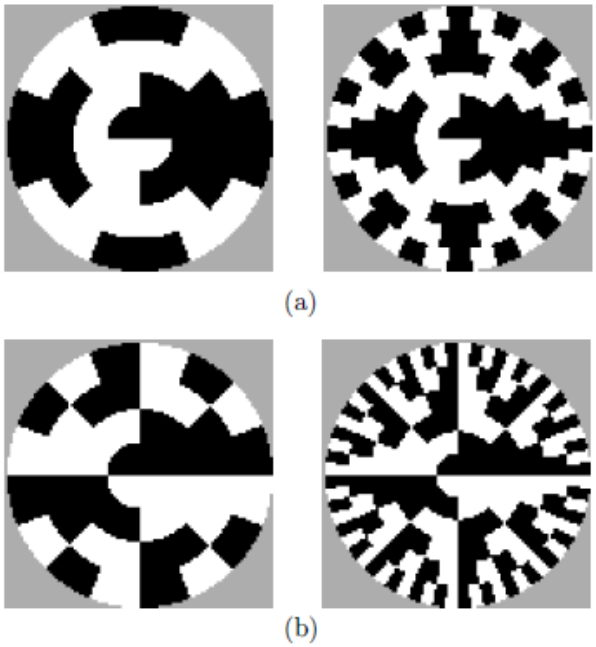


Figura Sol.30: Códigos angulares de rueda RGC y binario.

de código binario, y que los códigos de la primera y de la última también se diferencian en un solo bit.

4.6: Si los valores del píxel están en el intervalo $[0, 255]$, una diferencia $(P_i - Q_i)$ puede ser, a lo sumo, 255. El peor caso ocurre cuando todas las diferencias son 255. Es fácil ver que tal caso produce una RMSE de 255.

4.7: El código de la Figura 4.15 produce las coordenadas de los puntos de rotación:

$$(7{,}071, 0)\,,\ (9{,}19, 0{,}7071)\,,\ (17{,}9, 0{,}78)\,,\ (33{,}9, 1{,}41)\,,\ (43{,}13, -2{,}12)$$

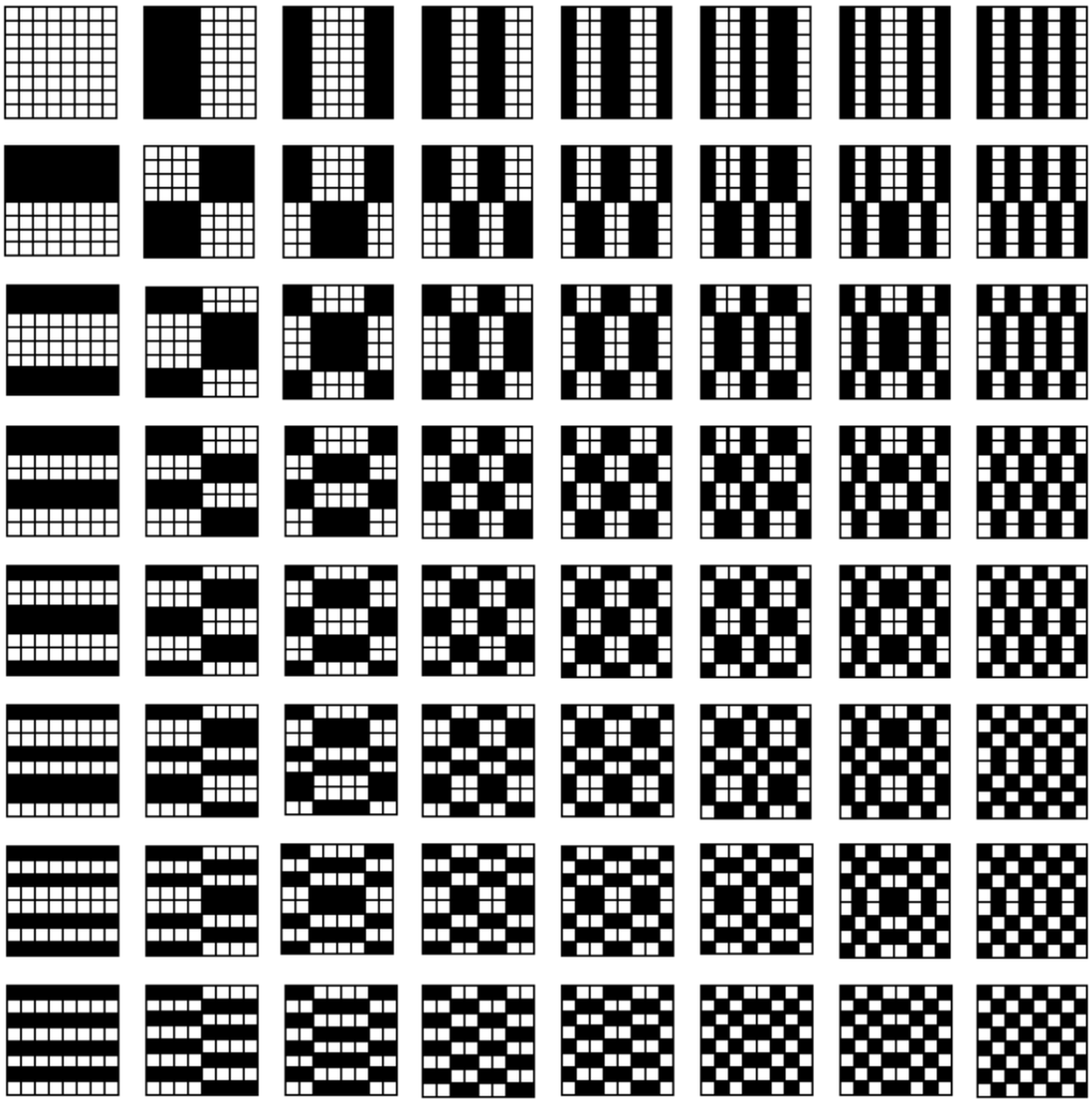
(Nótese cómo todas las coordenadas y son números pequeños) y muestra que la correlación cruzada cae de 1729,72 antes de la rotación a $-23{,}0846$ después de ella. ¡Una reducción significativa!

4.8: La Figura Sol.31 muestra las 64 imágenes básicas y el código en Matlab para calcularlas y mostrarlas[7]. Cada imagen básica es una matriz de 8×8.

4.9: A_4 es la matriz de 4×4:

$$A_4 = \begin{pmatrix} h_0\,(0/4) & h_0\,(1/4) & h_0\,(2/4) & h_0\,(3/4) \\ h_1\,(0/4) & h_1\,(1/4) & h_1\,(2/4) & h_1\,(3/4) \\ h_2\,(0/4) & h_2\,(1/4) & h_2\,(2/4) & h_2\,(3/4) \\ h_3\,(0/4) & h_3\,(1/4) & h_3\,(2/4) & h_3\,(3/4) \end{pmatrix} = \frac{1}{\sqrt{4}} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ \sqrt{2} & -\sqrt{2} & 0 & 0 \\ 0 & 0 & \sqrt{2} & -\sqrt{2} \end{pmatrix}.$$

[7]La función `sqrt(x)` proporciona la raíz cuadrada de x. Las demás funciones se explican por sí mismas.



```
M=3; N=2^M; H=[1 1; 1 -1]/sqrt(2);
for m=1:(M-1) % recursión
  H=[H H; H -H]/sqrt(2);
end
A=H';
map=[1 5 7 3 4 8 6 2]; % 1:N
for n=1:N, B(:,n)=A(:,map(n)); end;
A=B; sc=1/(max(abs(A(:))).^2); % factor de escala
for row=1:N
  for col=1:N
    BI=A(:,row)*A(:,col).'; % tensor producto
    subplot(N,N,(row-1)*N+col)
    oe=round(BI*sc); % resultados en -1, +1
    imagesc(oe), colormap([1 1 1; .5 .5 .5; 0 0 0])
    drawnow
  end
end
```

Figura Sol.31: Imágenes básicas WHT de 8×8 y el código en Matlab.

De forma similar, A_8 es la matriz:

$$A_8 = \frac{1}{\sqrt{8}} \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ \sqrt{2} & \sqrt{2} & -\sqrt{2} & -\sqrt{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \sqrt{2} & \sqrt{2} & -\sqrt{2} & -\sqrt{2} \\ 2 & -2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & -2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & -2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & -2 \end{pmatrix}.$$

4.10: El vector promedio $\mathbf{w}^{(i)}$ es cero, por lo que la Ecuación (4.12) (pág. 301) produce:

$$\left(\mathbf{W}\cdot\mathbf{W}^T\right)_{jj} = \sum_{i=1}^{k} \omega_j^{(i)}\omega_j^{(i)} = \\ = \sum_{i=1}^{k}\left(\omega_j^{(i)}-0\right)^2 = \sum_{i=1}^{k}\left(c_i^{(j)}-0\right)^2 = k\,\text{Varianza}\left(\mathbf{c}^{(j)}\right).$$

4.11: El código en *Mathematica* de la Figura 4.21 (pág. 305) produce los coeficientes: 140, −71, 0, −7, 0, −2, 0 y 0. Ahora cuantificamos este conjunto toscamente eliminando los dos últimos pesos distintos de cero −7 y −2. Cuando se aplica la IDCT a la secuencia 140, −71, 0, 0, 0, 0, 0, 0, produce: 15, 20, 30, 43, 56, 69, 79 y 84. Estos valores son idénticos a los originales, pero la diferencia máxima es de sólo 4; un resultado excelente, teniendo en cuenta que sólo dos de los ocho coeficientes DCT son distintos de cero.

4.12: Los ocho valores en la fila superior son muy similares (las diferencias entre ellos son, o bien 2 ó 3). Cada una de las otras filas se obtiene mediante un desplazamiento circular a la derecha de la fila precedente.

4.13: Es obvio que tal bloque se puede representar como una combinación lineal de patrones en la columna del extremo izquierdo de la Figura 4.39 (pág. 318). El cálculo real produce los ocho pesos: 4, 0,72, 0, 0,85, 0, 1,27, 0 y 3,62 para los patrones de esta columna. Los otros 56 pesos son iguales a cero o muy cercanos a cero.

4.14: Los argumentos de las funciones coseno utilizadas por la DCT son de la forma $^{(2x+1)i\pi}/_{16}$, donde i y x son números enteros del intervalo $[0, 7]$. Tal argumento se puede escribir en la forma $^{n\pi}/_{16}$, donde n es un número entero en el intervalo $[0,\ 15\times 7]$. Puesto que la función coseno es periódica, satisface: $\cos\left(^{32\pi}/_{16}\right) = \cos\left(^{0\pi}/_{16}\right)$, $\cos\left(^{33\pi}/_{16}\right) = \cos\left(^{\pi}/_{16}\right)$, y así sucesivamente. Como resultado, sólo son necesarios los 32 valores $\cos\left(^{n\pi}/_{16}\right)$ para $n = 0, 1, 2, \ldots, 31$. El autor está en deuda con V. Saravanan por señalar esta característica de la DCT.

4.15: La Figura 4.52 (pág. 337) muestra los resultados (que se asemejan a la Figura 4.39 —pag. 318—) y el código en Matlab. Observe que el mismo código también se puede utilizar para calcular y mostrar las imágenes base de la DCT.

4.16: En primer lugar se determina el camino en zigzag manualmente, luego se graba en un array `zz` de estructuras, donde cada estructura contiene un par de coordenadas para la ruta, como se muestra, e.g., en la Figura Sol.32.
Si los dos componentes de una estructura son `zz.r` y `zz.c`, entonces el recorrido en zigzag puede realizarse mediante un bucle de la forma:

$(0,0)$	$(0,1)$	$(1,0)$	$(2,0)$	$(1,1)$	$(0,2)$	$(0,3)$	$(1,2)$
$(2,1)$	$(3,0)$	$(4,0)$	$(3,1)$	$(2,2)$	$(1,3)$	$(0,4)$	$(0,5)$
$(1,4)$	$(2,3)$	$(3,2)$	$(4,1)$	$(5,0)$	$(6,0)$	$(5,1)$	$(4,2)$
$(3,3)$	$(2,4)$	$(1,5)$	$(0,6)$	$(0,7)$	$(1,6)$	$(2,5)$	$(3,4)$
$(4,3)$	$(5,2)$	$(6,1)$	$(7,0)$	$(7,1)$	$(6,2)$	$(5,3)$	$(4,4)$
$(3,5)$	$(2,6)$	$(1,7)$	$(2,7)$	$(3,6)$	$(4,5)$	$(5,4)$	$(6,3)$
$(7,2)$	$(7,3)$	$(6,4)$	$(5,5)$	$(4,6)$	$(3,7)$	$(4,7)$	$(5,6)$
$(6,5)$	$(7,4)$	$(7,5)$	$(6,6)$	$(5,7)$	$(6,7)$	$(7,6)$	$(7,7)$

Figura Sol.32: Coordenadas para la ruta en zigzag.

```
for (i=0; i<64; i++){
row:=zz[i].r; col:=zz[i].c
...data_unit[row][col]...}
```

4.17: La tercera diferencia DC —5—; está situada en la fila 3, columna 5, por lo que se codifica como 1110|101.

4.18: Trece ceros consecutivos preceden a este coeficiente, por lo que Z = 13. El coeficiente en sí mismo se encuentra en la Tabla 4.63 (pág. 352), en la fila 1, columna 0, por lo que R = 1 y C = 0. Suponiendo que el código de Huffman en la posición $(\mathrm{R},\mathrm{Z}) = (1,13)$ de la Tabla 4.664.66 (pág. 356) es 1110101, el código final emitido para 1 es 1110101|0.

4.19: Ésto se demuestra multiplicando el mayor de los cuatro números de n bits, $\underbrace{11\ldots1}_{n}$ por 4; lo que se hace fácilmente realizando un desplazamiento —a nivel de bit— de dos posiciones a la izquierda. El resultado es el número de $n+2$ bits: $\underbrace{11\ldots1}_{n}00$.

4.20: Consiguen un crecimiento progresivo más natural de la imagen. Facilitan a una persona que observa el desarrollo de la imagen en la pantalla decidir si y cuándo detener el proceso de decodificación, y aceptar o descartar la imagen.

4.21: La única especificación que depende de los bits particulares asignados a los dos colores es la Ecuación (4.25). Todas las otras partes de JBIG son independientes de la asignación de bits.

4.22: Para la plantilla de 16 bits de la Figura 4.97a (pág. 389), las coordenadas relativas son:

$$A_1 = (3,-1)\,, \quad A_2 = (-3,-1)\,, \quad A_3 = (2,-2)\,, \quad A_4 = (-2,-2)\,.$$

Para la plantilla de 13 bits de la Figura 4.97b, las coordenadas relativas de A_1 son $(3,-1)$. Para las plantillas de 10 bits de la Figura 4.97c,d, las coordenadas relativas de A_1 son $(2,-1)$.

4.23: La transposición de S y T produce una mejor compresión en aquellos casos en que el texto se dispone verticalmente.

4.24: Volviendo al paso 1 tenemos los mismos puntos participando en la partición para cada entrada en el libro de códigos (ésto ocurre porque nuestros puntos se concentran en cuatro regiones diferentes; pero, en general, una partición $P_i^{(k)}$ puede estar formada por bloques de imágenes

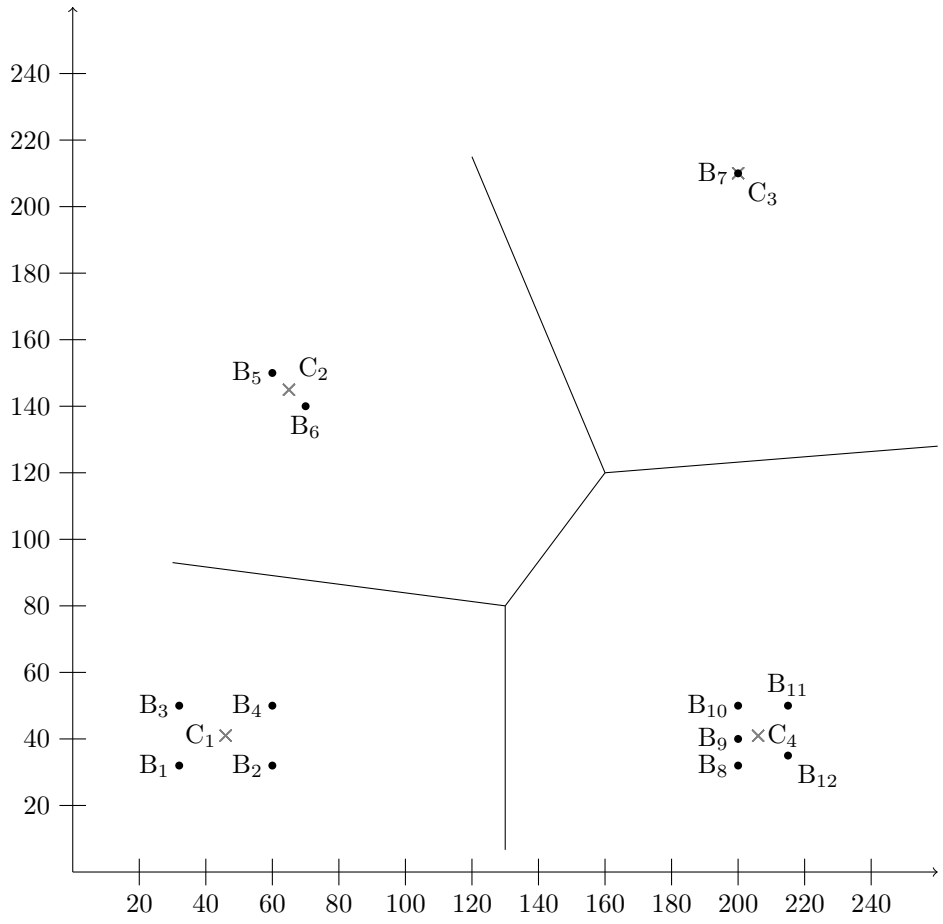


Figura Sol.33: Doce puntos y cuatro entradas $C_i^{(1)}$ del libro de códigos.

diferentes en cada iteración k). Las distorsiones calculadas en el paso 2, se resumen en la Tabla Sol.34. La distorsión $D_i^{(1)}$, en promedio, es 163,17:

$$D^{(1)} = \frac{(277 + 277 + 277 + 277 + 50 + 50 + 200 + 117 + 37 + 117 + 162 + 117)}{12},$$

mucho menor que el valor original: 645. Si el paso 3 indica que no hay convergencia, deben hacerse más iteraciones (Ejercicio 4.25), reduciendo la distorsión media y mejorando los valores de las cuatro entradas del libro de códigos.

4.25: Cada nueva entrada del libro de códigos $C_i^{(k)}$ se calcula, en el paso 4 de la iteración k, como el promedio de los bloques de imágenes que comprende la partición $P_i^{(k-1)}$. En nuestro ejemplo, los bloques de imágenes (puntos) se concentran en cuatro regiones distintas, por lo que las particiones calculadas para la iteración $k = 1$ son las mismas que las calculadas para $k = 0$. Otra iteración —para $k = 2$— calculará, por lo tanto, las mismas particiones en su paso 1; dando —en el paso 3— un promedio de distorsión $D^{(2)}$ que es igual a $D^{(1)}$. Por consiguiente, el paso 3 indicará convergencia.

4.26: Es $4^{-8} \approx 0{,}000015$ del área del espacio completo.

$$
\begin{array}{rll}
\text{I:} & (46-32)^2+(41-32)^2=277, & (46-60)^2+(41-32)^2=277,\\
 & (46-32)^2+(41-50)^2=277, & (46-60)^2+(41-50)^2=277,\\
\text{II:} & (65-60)^2+(145-150)^2=\ \ 50, & (65-70)^2+(145-140)^2=\ \ 50,\\
\text{III:} & (210-200)^2+(200-210)^2=200, & \\
\text{IV:} & (206-200)^2+(41-32)^2=117, & (206-200)^2+(41-40)^2=\ \ 37,\\
 & (206-200)^2+(41-50)^2=117, & (206-215)^2+(41-50)^2=162,\\
 & (206-215)^2+(41-35)^2=117. &
\end{array}
$$

Tabla Sol.34: Doce distorsiones para $k=1$.

4.27: Supervisar la razón de compresión, y vaciar el diccionario y empezar de nuevo cada vez que el rendimiento de la compresión esté por debajo de un cierto umbral.

4.28: *Paso 4*: Se saca el punto $(2,0)$ del GPP. El valor del píxel en esta posición es 7. La mejor correspondencia en este instante es con la entrada del diccionario que contiene 7. El codificador emite el puntero 7. El emparejamiento no contiene ninguna esquina cóncava, por lo que podemos desplazar el punto hacia la derecha del bloque emparejado, $(2,1)$, y el punto por debajo de él, $(3,0)$, en el GPP. El GPP ahora contiene los puntos $(2,1)$, $(3,0)$, $(0,2)$ y $(1,1)$. El diccionario se actualiza añadiendo al mismo (en la localización 18) el bloque $\begin{array}{|c|}\hline 4\\\hline 7\\\hline\end{array}$.

Paso 5: Se saca el punto $(1,1)$ del GPP. El valor del píxel en esta posición es 5. La mejor correspondencia en este instante es con la entrada de diccionario que contiene 5. El codificador emite el puntero 5. El emparejamiento no contiene ninguna esquina cóncava, por lo que podemos desplazar el punto hacia la derecha del bloque emparejado, $(1,2)$, y el punto por debajo de él, $(2,1)$, en el GPP. El GPP contiene los puntos $(1,2)$, $(2,1)$, $(3,0)$ y $(0,2)$. El diccionario se actualiza añadiendo al mismo (en las localizaciones 19 y 20, respectivamente) los dos bloques: $\begin{array}{|c|}\hline 2\\\hline 5\\\hline\end{array}$ y $\begin{array}{|c|c|}\hline 4 & 5\\\hline\end{array}$.

4.29: Simplemente, puede ser demasiado largo. Cuando se comprime texto, cada símbolo es normalmente de 1 byte de longitud (dos bytes en Unicode). Sin embargo, las imágenes de 24 bits por píxel son muy comunes, y un bloque de 16 píxeles en dicha imagen ocupa $^{24}/_{8}\times 16=48$ bytes de longitud.

4.30: Si el codificador utiliza un código unario general $(2,1,k)$, entonces el valor de k también debería incluirse en la cabecera.

4.31: La media y la desviación estándar son $\bar{p}=115$ y $\sigma=77{,}93$, respectivamente. Los recuentos se convierten en: $n^+=n^-=8$, y se resuelven las Ecuaciones (4.33) para obtener $p^+=193$ y $p^-=37$. El bloque original se comprime en los 16 bits:

$$\begin{pmatrix} 1 & 0 & 1 & 1\\ 1 & 0 & 0 & 1\\ 1 & 1 & 0 & 1\\ 0 & 0 & 0 & 0 \end{pmatrix},$$

y en los dos valores de 8 bits: 37 y 193.

4.32: La Tabla Sol.35 resume los resultados. Observe cómo a un píxel 1, con un contexto 00, se le asigna una probabilidad alta después de haberse visto 3 veces.

#	Píxel	Contexto	Recuentos	Probabilidades	Nuevos recuentos
5	0	10 = 2	1, 1	1/2	2, 1
6	1	00 = 0	1, 3	3/4	1, 4
7	0	11 = 3	1, 1	1/2	2, 1
8	1	10 = 2	2, 1	1/3	2, 2

Tabla Sol.35: Recuentos y probabilidades de los cuatro píxeles siguientes.

4.33: Tal cosa es posible para el codificador, pero no para el decodificador. Un método de compresión que utiliza píxeles "futuros" en el contexto es inútil, debido a que su salida sería imposible de descomprimir.

4.34: El modelo utilizado por FELICS para predecir el píxel actual es un modelo de Markov de segundo orden. En este modelo, el valor del ítem de datos actual depende de sólo dos de sus vecinos pasados, no necesariamente los dos más inmediatos.

4.35: Los dos vecinos de P = 8 vistos previamente son: A = 1 y B = 11. P está, por lo tanto, en la región central, donde todos los códigos comienzan con un cero, y L = 1, H = 11. Los cálculos son sencillos:

$$k = [\log_2 (11 - 1 + 1)] = 3, \qquad a = 2^{3+1} - 11 = 5, \qquad b = 2\left(11 - 2^3\right) = 6.$$

La Tabla Sol.36 muestra los cinco códigos de 3 bits y los seis códigos de 4 bits para la región central. El código para 8 es, por lo tanto, 0|111.

Los dos vecinos de P = 7 vistos previamente son: A = 2 y B = 5. P está, por lo tanto, en la región exterior derecha, donde todos los códigos empiezan con 11, y L = 2, H = 7. Estamos buscando el código para $7 - 5 = 2$. Eligiendo $m = 1$ se obtiene, a partir de la Tabla 4.120 (pág. 422), el código 11|01.

Los dos vecinos de P = 0 vistos previamente son: A = 3 y B = 5. P está, por lo tanto, en la región exterior izquierda, donde todos los códigos empiezan con 10, y L = 3, H = 5. Estamos buscando el código para $3 - 0 = 3$. Eligiendo $m = 1$ se obtiene, a partir de la Tabla 4.120, el código 10|100.

Píxel P	Código de la región	Código del píxel
1	0	0000
2	0	0010
3	0	0100
4	0	011
5	0	100
6	0	101
7	0	110
8	0	111
9	0	0001
10	0	0011
11	0	0101

Tabla Sol.36: Los códigos para una región central.

4.36: Debido a que el decodificador tiene que resolver los vínculos de la misma manera que el codificador.

4.37: Los pesos tienen que sumar un máximo de 1, porque esto se traduce en una suma ponderada, cuya valor está en el mismo rango que los valores de los píxeles. Si los valores del píxel están, por ejemplo, en el rango $[0, 15]$ y los pesos suman hasta 2, una predicción puede dar lugar a valores de hasta 30.

4.38: Cada uno de los tres pesos 0,0039, −0,0351 y 0,3164 se usa dos veces. La suma de los pesos es, por lo tanto, 0,5704, y el resultado de dividir cada peso por esta suma es 0,0068, −0,0615 y 0,5547. Es fácil comprobar que la suma los pesos renormalizados $2\,(0{,}0068 - 0{,}0615 + 0{,}5547)$ es igual a 1.

4.39: Un archivo de imágenes estáticas es un ejemplo donde este enfoque es práctico. La NASA tiene un gran archivo de imágenes tomadas por varios satélites. Deben mantenerse altamente comprimidas, pero nunca cambian, por lo que cada imagen tiene que ser comprimida sólo una vez. Un codificador lento, por tanto, es aceptable, pero un decodificador rápido es sin duda muy útil. Otro ejemplo es una colección de arte. Muchos museos han escaneado y digitalizado sus colecciones de pintura, y ésas también son estáticas.

4.40: Tal polinomio depende de tres coeficientes **b**, **c** y **d** que pueden considerarse puntos tridimensionales, y cualesquier tres puntos están en el mismo plano.

4.41: Ésto es sencillo:

$$\begin{aligned}\mathbf{P}\,(2/3) &= (0,-9)\,(2/3)^3 + (-4{,}5,\,13{,}5)\,(2/3)^2 + (4{,}5,-3{,}5)\,(2/3)\\ &= (0,-8/3) + (-2,\,6) + (3,-7/3)\\ &= (1,\,1) = \mathbf{P_3}.\end{aligned}$$

4.42: Usamos las relaciones $\sin 30^o = \cos 60^o = 1/2 = 0{,}5$ y la aproximación $\cos 30^o = \sin 60^o = \sqrt{3}/2 \approx 0{,}866$. Los cuatro puntos son: $\mathbf{P}_1 = (1, 0)$, $\mathbf{P}_2 = (\cos 30^o, \sin 30^o) = (0{,}866,\,0{,}5)$, $\mathbf{P}_3 = (0{,}5,\,0{,}866)$, y $\mathbf{P}_4 = (0, 1)$. La relación $\mathbf{A} = \mathbf{N} \cdot \mathbf{P}$ se convierte en:

$$\begin{pmatrix}\mathbf{a}\\ \mathbf{b}\\ \mathbf{c}\\ \mathbf{d}\end{pmatrix} = \mathbf{A} = \mathbf{N}\cdot\mathbf{P} = \begin{pmatrix} -4{,}5 & 13{,}5 & -13{,}5 & 4{,}5\\ 9{,}0 & -22{,}5 & 18 & -4{,}5\\ -5{,}5 & 9{,}0 & -4{,}5 & 1{,}0\\ 1{,}0 & 0 & 0 & 0\end{pmatrix}\begin{pmatrix}(1,0)\\ (0{,}866,\,0{,}5)\\ (0{,}5,\,0{,}866)\\ (0,1)\end{pmatrix}$$

y las soluciones son:

$$\begin{aligned}\mathbf{a} &= -\,4{,}5\,(1,0) + 13{,}5\,(,866,\,,5) - 13{,}5\,(,5,\,,866) + 4{,}5\,(0,1) = (,441,\,-,441)\,,\\ \mathbf{b} &= 9\,(1,0) - 22{,}5\,(,866,\,,5) + 18\,(,5,\,,866) - 4{,}5\,(0,1) = (-1{,}485,-0{,}162)\,,\\ \mathbf{c} &= -\,5{,}5\,(1,0) + 9\,(,866,\,,5) - 4{,}5\,(,5,\,,866) + 1\,(0,1) = (0{,}044,\,1{,}603)\,,\\ \mathbf{d} &= 1\,(1,0) - 0\,(,866,\,,5) + 0\,(,5,\,,866) - 0\,(0,1) = (1,0)\,.\end{aligned}$$

Por consiguiente, el PC es $\mathbf{P}\,(t) = (,441,-,441)\,t^3 + (-1{,}485,-0{,}162)\,t^2 + (0{,}044,\,1{,}603)\,t + (1,0)$. El punto medio es $\mathbf{P}\,(0{,}5) = (0{,}7058,\,0{,}7058)$, sólo alejado un 0,2 % del punto medio del arco, que se encuentra en: $(\cos 45^o, \operatorname{sen} 45^o) = \left(\sqrt{2}/2, \sqrt{2}/2\right) \approx (0{,}7071,\,0{,}7071)$.

4.43: Las nuevas ecuaciones son bastante fáciles de construir. Usando *Mathematica*, también son fáciles de resolver. El siguiente código

```
Solve[{d==p1,
a al^3+b al^2+c al+d==p2,
a be^3+b be^2+c be+d==p3,
a+b+c+d==p4},{a,b,c,d}];
ExpandAll[Simplify[%]]
```

(donde `al` y `be` representan a α y β, respectivamente) produce las (confusas) soluciones:

$$\begin{aligned}
\mathbf{a} &= -\frac{\mathbf{P}_1}{\alpha\beta} + \frac{\mathbf{P}_2}{-\alpha^2+\alpha^3+\alpha\beta-\alpha^2\beta} + \frac{\mathbf{P}_3}{\alpha\beta-\beta^2-\alpha\beta^2+\beta^3} + \frac{\mathbf{P}_4}{1-\alpha-\beta+\alpha\beta},\\
\mathbf{b} &= \mathbf{P}_1{}^{(-\alpha+\alpha^3+\beta-\alpha^3\beta-\beta^3+\alpha\beta^3)}/_{\gamma} + \mathbf{P}_2{}^{(-\beta+\beta^3)}/_{\gamma}\\
&\quad + \mathbf{P}_3{}^{(\alpha-\alpha^3)}/_{\gamma} + \mathbf{P}_4{}^{(\alpha^3\beta-\alpha\beta^3)}/_{\gamma},\\
\mathbf{c} &= -\mathbf{P}_1\left(1+\frac{1}{\alpha}+\frac{1}{\beta}\right) + \frac{\beta\mathbf{P}_2}{-\alpha^2+\alpha^3+\alpha\beta-\alpha^2\beta}\\
&\quad + \frac{\alpha\mathbf{P}_3}{\alpha\beta-\beta^2-\alpha\beta^2+\beta^3} + \frac{\alpha\beta\mathbf{P}_4}{1-\alpha-\beta+\alpha\beta},\\
\mathbf{d} &= \mathbf{P}_1,
\end{aligned}$$

donde $\gamma = (-1+\alpha)\,\alpha\,(-1+\beta)\,\beta\,(-\alpha+\beta)$.

De aquí, sale inmediatamente la matriz base:

$$\begin{pmatrix}
-\frac{1}{\alpha\beta} & \frac{1}{-\alpha^2+\alpha^3+\alpha\beta-\alpha^2\beta} & \frac{1}{\alpha\beta-\beta^2-\alpha\beta^2+\beta^3} & \frac{1}{1-\alpha-\beta+\alpha\beta}\\
\frac{-\alpha+\alpha^3+\beta-\alpha^3\beta-\beta^3+\alpha\beta^3}{\gamma} & \frac{-\beta+\beta^3}{\gamma} & \frac{\alpha-\alpha^3}{\gamma} & \frac{\alpha^3\beta-\alpha\beta^3}{\gamma}\\
-\left(1+\frac{1}{\alpha}+\frac{1}{\beta}\right) & \frac{\beta}{-\alpha^2+\alpha^3+\alpha\beta-\alpha^2\beta} & \frac{\alpha}{\alpha\beta-\beta^2-\alpha\beta^2+\beta^3} & \frac{\alpha\beta}{1-\alpha-\beta+\alpha\beta}\\
1 & 0 & 0 & 0
\end{pmatrix}$$

Una comprobación directa, de nuevo usando *Mathematica*, para $\alpha = 1/3$ y $\alpha = 2/3$, reduce ésta a la matriz $\mathbf{N}$ de la Ecuación (4.44).

4.44: Los puntos que faltan tienen que ser estimados por interpolación o extrapolación de los puntos conocidos antes de poder aplicar nuestro método. Obviamente, cuantos menos puntos se conozcan, peor será la interpolación final. Nótese que son necesarios 16 puntos, porque un polinomio bicúbico tiene 16 coeficientes.

4.45: La Figura Sol.37a muestra una cuadrícula en forma de diamante de 16 puntos igualmente espaciados. Los ocho puntos con pesos negativos se muestran en negro. La Figura Sol.37b muestra un corte (etiquetado `xx`) a través de cuatro puntos de esta superficie. El corte es una curva que pasa a través de puntos de datos que sobresalen. Es fácil ver que cuando los dos puntos exteriores (en negro) se elevan, el centro de la curva (y, como resultado, el centro de la superficie) desciende. Ahora está claro que los puntos con pesos negativos empujan el centro de la superficie en una dirección opuesta a los puntos.

La Figura Sol.37c es un ejemplo más detallado, que también muestra por qué las cuatro puntos de esquina deben tener pesos positivos. Se muestra una simple porción de superficie simétrica que interpola los 16 puntos:

$$\begin{array}{llll}
\mathbf{P}_{00} = (0,0,0)\,, & \mathbf{P}_{10} = (1,0,1)\,, & \mathbf{P}_{20} = (2,0,1)\,, & \mathbf{P}_{30} = (3,0,0)\,,\\
\mathbf{P}_{01} = (0,1,1)\,, & \mathbf{P}_{11} = (1,1,2)\,, & \mathbf{P}_{21} = (2,1,2)\,, & \mathbf{P}_{31} = (3,1,1)\,,\\
\mathbf{P}_{02} = (0,2,1)\,, & \mathbf{P}_{12} = (1,2,2)\,, & \mathbf{P}_{22} = (2,2,2)\,, & \mathbf{P}_{32} = (3,2,1)\,,\\
\mathbf{P}_{03} = (0,3,0)\,, & \mathbf{P}_{13} = (1,3,1)\,, & \mathbf{P}_{23} = (2,3,1)\,, & \mathbf{P}_{33} = (3,3,0)\,.
\end{array}$$

En primer lugar, se elevan los ocho puntos límite desde $z = 1$ hasta $z = 1{,}5$. La Figura Sol.37d muestra cómo el punto central $\mathbf{P}\,(0{,}5,\ 0{,}5)$ descenció desde (1,5, 1,5, 2,25) hasta (1,5, 1,5, 2,10938). Inmediatamente, devolvemos ésos puntos a sus posiciones originales, y elevamos los cuatro puntos de las esquinas desde $z = 0$ a $z = 1$. La Figura Sol.37e muestra cómo se eleva el punto central desde (1,5, 1,5, 2,25) hasta (1,5, 1,5, 2,26563).

4.46: El decodificador conoce este píxel, ya que ha calculado el valor promedio

$$\mu\,[i-1,j] = 0{,}5\,(I\,[2i-2,2j] + I\,[2i-1,2j+1])\,,$$

y además, ya ha decodificado píxel $I\,[2i-2,2j]$.

4.47: El decodificador es capaz de hacer ésto, porque cuando el decodificador introduce el 5, sabe que la diferencia entre p (el píxel que está siendo decodificado) y el píxel de referencia, comienza en la posición 6 (contando desde la izquierda). Puesto que el bit 6 del píxel de referencia es 0, el de p debe ser un 1.

4.48: Sí, pero la compresión podría degradarse. Una manera de aplicar este método consiste en separar cada byte en dos píxeles de 4 bits y codificar cada píxel por separado. Este enfoque es malo puesto que el prefijo y sufijo de un píxel de 4 bits, a menudo puede estar formado por más de cuatro bits. Otro enfoque es ignorar el hecho de que un byte contiene dos píxeles, y utilizar el método como se había descrito originalmente. Ésto todavía puede comprimir la imagen, pero no es muy eficiente, como ilustra el ejemplo siguiente.

Ejemplo: Los dos bytes de 1100|1101 y 1110|1111 representan cuatro píxeles, cada uno distinto de su vecino inmediato por su bit menos significativo. Los cuatro píxeles, por lo tanto, tienen colores similares (o en escala de grises). La comparación de píxeles consecutivos produce los prefijos 3 ó 2, pero la comparación de los dos bytes produce el prefijo 2.

4.49: La suma de los pesos es 1, porque ésto produce un valor X en el mismo rango que A, B y C. Si los pesos fueran, por ejemplo, 1, 100, y 1, X tendría valores mucho más grandes que cualquiera de los tres píxeles.

4.50: Los cuatro vectores son:

$$\begin{aligned}\mathbf{a} &= (90, 95, 100, 80, 90, 85)\,,\\ \mathbf{b}^{(1)} &= (100, 90, 95, 102, 80, 90)\,,\\ \mathbf{b}^{(2)} &= (101, 128, 108, 100, 90, 95)\,,\\ \mathbf{b}^{(3)} &= (128, 108, 110, 90, 95, 100)\,,\end{aligned}$$

y el código de la Figura Sol.38 produce las soluciones $\omega_1 = 0{,}1051$, $\omega_2 = 0{,}3974$, y $\omega_3 = 0{,}3690$. Su total es de 0,8715, en comparación con las soluciones originales, que sumaban 0,9061. La cuestión es, que los números involucrados en las ecuaciones (los elementos de los cuatro vectores) no son independientes (por ejemplo, el píxel 80 aparece en $\mathbf{a}$ y en $\mathbf{b}^{(1)}$), excepto para el último elemento (85 ó 91) de $\mathbf{a}$ y el primer elemento (101) de $\mathbf{b}^{(2)}$, que son independientes. La modificación de estos dos elementos afecta a las soluciones, que es por lo que las soluciones no siempre suman la unidad. Sin embargo, la compresión de nueve píxeles produce soluciones cuyo total es más cercano a uno que en el caso de seis píxeles. La compresión de una imagen completa, con muchos miles de píxeles, produce soluciones cuya suma está muy cercana a 1.

4.51: La Figura Sol.39a,b,c muestra los resultados, con todos los valores de H_i mostrados en pequeño. La mayor parte de los valores de H_i son cero porque los píxeles de la imagen original están

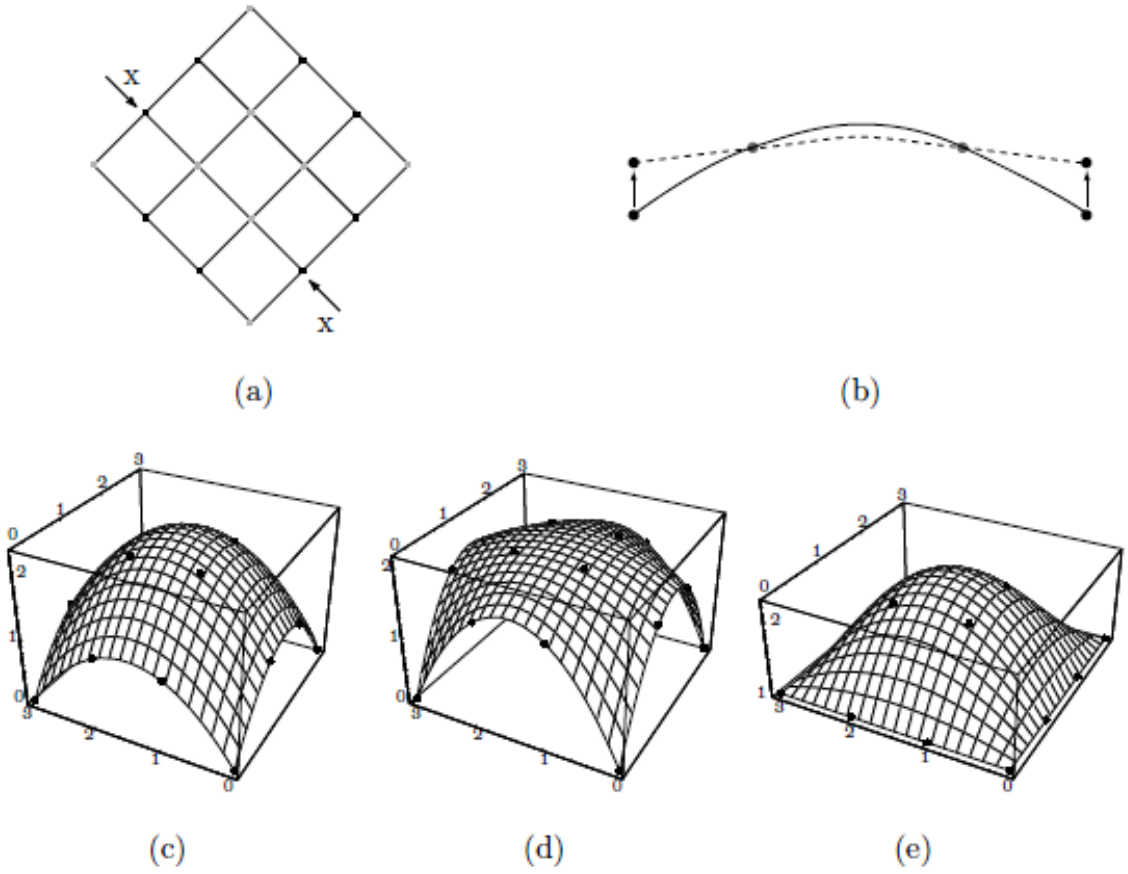


Figura Sol.37: Una interpolación bicúbica de una porción de superficie.

```
Clear[Nh,p,pnts,U,W];
p00={0,0,0}; p10={1,0,1}; p20={2,0,1}; p30={3,0,0};
p01={0,1,1}; p11={1,1,2};
p21={2,1,2}; p31={3,1,1};
p02={0,2,1};
p12={1,2,2}; p22={2,2,2}; p32={3,2,1};
p03={0,3,0}; p13={1,3,1}; p23={2,3,1};
p33={3,3,0};
Nh={{-4.5,13.5,-13.5,4.5},{9,-22.5,18,-4.5},
  {-5.5,9,-4.5,1},{1,0,0,0}}; pnts={{p33,p32,p31,p30},{p23,p22,p21,p20},
  {p13,p12,p11,p10},{p03,p02,p01,p00}}; U[u_]:={u^3,u^2,u,1};
W[w_]:={w^3,w^2,w,1};
(* prt [i] extrae el componente i de la tercera dimensión de P *)
prt[i_]:=pnts[[Range[1,4],Range[1,4],i]];
p[u_,w_]:={U[u].Nh.prt[1].Transpose[Nh].W[w],
  U[u].Nh.prt[2].Transpose[Nh].W[w], \
  U[u].Nh.prt[3].Transpose[Nh].W[w]};
g1=ParametricPlot3D[p[u,w], {u,0,1},{w,0,1},
Compiled->False, DisplayFunction->Identity];
g2=Graphics3D[{AbsolutePointSize[2],
  Table[Point[pnts[[i,j]]],{i,1,4},{j,1,4}]}];
Show[g1,g2, ViewPoint->{-2.576, -1.365, 1.718}]
```

Código para la Figura Sol.37

```
a={90.,95,100,80,90,85}; b1={100,90,95,100,80,90};
b2={100,128,108,100,90,95};
b3={128,108,110,90,95,100};
Solve[{b1.(a-w1 b1-w2 b2-w3 b3)==0,
b2.(a-w1 b1-w2 b2-w3 b3)==0,
b3.(a-w1 b1-w2 b2-w3 b3)==0},{w1,w2,w3}]
```

Figura Sol.38: Resolución para tres pesos.

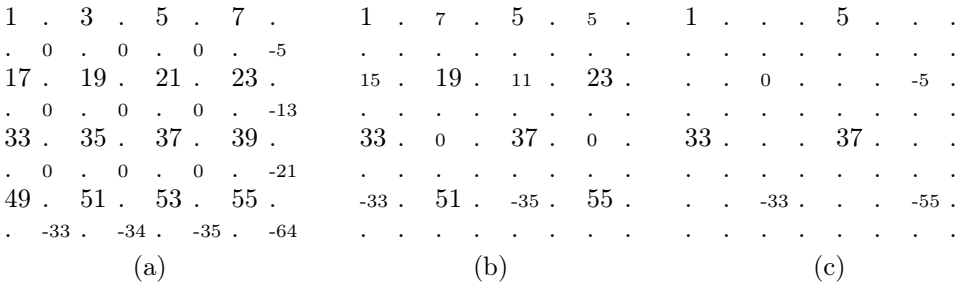


Figura Sol.39: (a) Bandas L_2 y H_2. (b) Bandas L_3 y H_3. (c) Bandas L_4 y H_4.

altamente correlacionados. Los valores de H_i a lo largo de los bordes son muy diferentes debido a la sencilla regla de borde utilizada. El resultado es que los valores de H_i están altamente descorrelacionados y tienen una baja entropía. Por lo tanto, son candidatos para la codificación de entropía.

4.52: Hay 16 valores. El valor 0 aparece nueve veces, y cada uno de los otros siete valores aparece una vez. La entropía es, por lo tanto:

$$-\sum p_i \log_2 p_i = -\frac{9}{16}\log_2\left(\frac{9}{16}\right) - 7\frac{1}{16}\log_2\left(\frac{1}{16}\right) \approx 2{,}2169.$$

No es muy pequeña, ya que siete de los 16 valores tienen la misma probabilidad. En la práctica, los valores con una banda de diferencia H_i tienden a ser pequeños; son tanto positivos como negativos, y se concentran en torno a cero, por lo que su entropía es pequeña.

4.53: Porque que el decodificador tiene que saber cómo estima el codificador X para cada valor de diferencia H_i. Si el codificador utiliza uno de los tres métodos para la predicción, tiene que preceder cada valor de diferencia en la cadena comprimida con un código, que indique al decodificador qué método ha utilizado. Dicho código puede ser de tamaño variable (por ejemplo: 0, 10, 11), pero incluso la adición de sólo uno o dos bits para cada predicción reduce el rendimiento de la compresión significativamente, ya que cada valor H_i necesita ser predicho, y el número de estos valores es cercano al tamaño de la imagen.

4.54: El árbol binario se muestra en la Figura Sol.40. Para este árbol, es fácil ver que el archivo de imagen progresiva es 3 6|5 7|7 7 10 5.

4.55: Éstas se muestran en la Figura Sol.41.

4.56: No. Una imagen con poca o ninguna correlación entre píxeles no se comprime con quadrisection, aunque el tamaño de la última matriz es siempre pequeño. Incluso sin conocer los detalles de

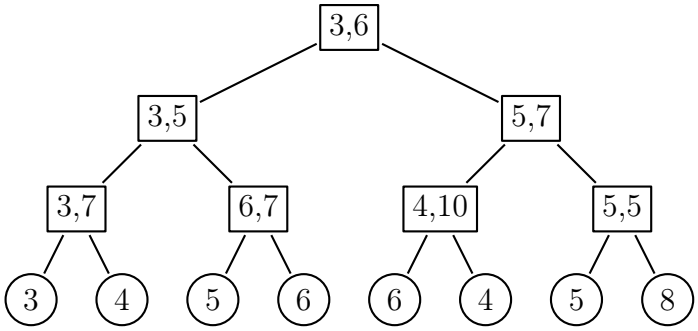


Figura Sol.40: Un árbol binario para una imagen de 8 bits por píxel.

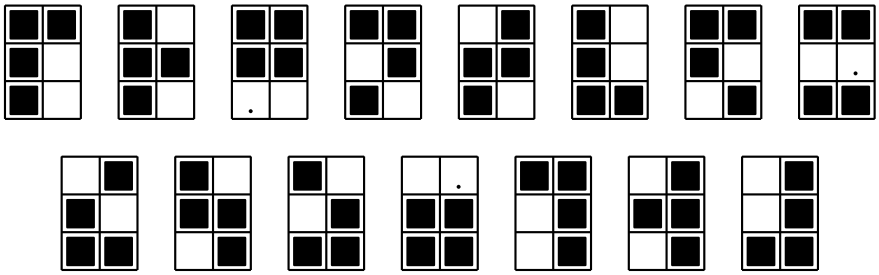


Figura Sol.41: Las 15 tuplas de 6 con dos píxeles blancos.

quadrisection, podemos estar seguros de que esa imagen producirá una secuencia de matrices M_j con pocas o ninguna fila idéntica. En el caso extremo, donde las filas de cualquier M_j son todas distintas, cada M_j tendrá cuatro veces el número de filas de su predecesor. Ésto creará vectores indicadores I_j que se hacen más y más largos, lo que aumenta el tamaño de la secuencia de datos comprimidos y reduce el rendimiento de la compresión completa.

4.57: La matriz M_5 es la concatenación de las 12 filas distintas de M_4

$$M_5^T = (0000|0001|1111|0011|1010|1101|1000|0111|1110|0101|1011|0010)\,.$$

4.58: M_4 tiene cuatro columnas, por lo que puede tener un máximo de 16 filas distintas, lo que implica que M_5 puede tener como máximo $4 \times 16 = 64$ elementos.

4.59: El decodificador tiene que leer la cadena comprimida completa, guardarla en la memoria, e iniciar la decodificación con L_5. Agrupando los ocho elementos de L_5 obtiene los cuatro elementos distintos: 01, 11, 00, y 10 de L_4, por lo que I_4 ahora puede utilizarse para reconstruir L_4. Los cuatro ceros de I_4 corresponden a los cuatro elementos distintos de L_4, y los 10 elementos restantes de L_4 pueden obtenerse a partir de ellos. Una vez que se ha construido L_4, se agrupan sus 14 elementos para formar los siete elementos distintos de L_3. Estos elementos son: 0111, 0010, 1100, 0110, 1111, 0101 y 1010; y corresponden a los siete ceros de I_3. Una vez que se ha construido L_3, sus ocho elementos se agrupan para formar los cuatro elementos distintos de L_2. Esos cuatro elementos son L_2 completo, ya que I_2 es todo ceros. La reconstrucción de L_1 y L_0 ahora es trivial.

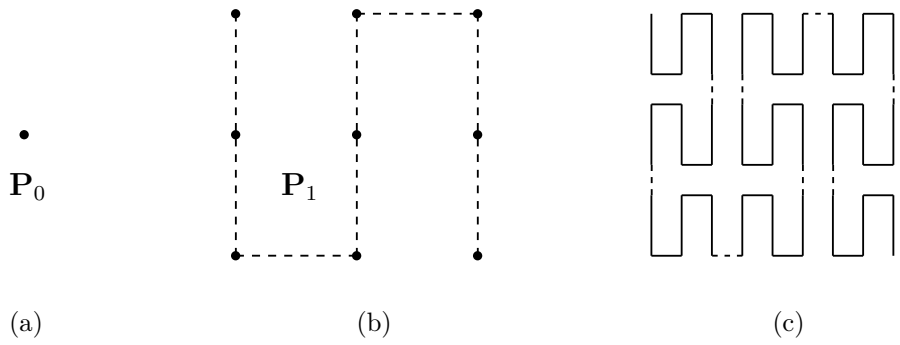


Figura Sol.42: Las primeras tres iteraciones de la curva de Peano.

4.60: Las dos mitades de L_0 son distintas, por lo que L_1 está formada por los dos elementos:

$$L_1 = (0101010101010101, 1010101010101010)\,,$$

y el primer vector indicador es $I_1 = (0, 0)$. Los dos elementos de L_1 son distintos, por lo que L_2 tiene los cuatro elementos

$$L_2 = (01010101, 01010101, 10101010, 10101010)\,,$$

y el segundo vector indicador es $I_2 = (0, 1, 0, 2)$. Dos elementos de L_2 son distintos, por lo que L_3 tiene los cuatro elementos $L_3 = (0101, 0101, 1010, 1010)$, y el tercer vector indicador es $I_3 = (0, 1, 0, 2)$. De nuevo, dos elementos de L_3 son distintos, por lo que L_4 tiene los cuatro elementos $L_4 = (01, 01, 10, 10)$, y el cuarto vector indicador es $I_4 = (0, 1, 0, 2)$. Sólo dos elementos de L_4 son distintos, por lo que L_5 consta de los cuatro elementos $L_5 = (0, 1, 1, 0)$.

En consecuencia, la salida está formada por: $k = 5$, el valor 2 (indicando que I_2 es el primer vector distinto de cero) I_2, I_3, e I_4 (codificados), seguidos por $L_5 = (0, 1, 1, 0)$.

4.61: El uso de una curva de Hilbert produce los 21 *runs*: 5, 1, 2, 1, 2, 7, 3, 1, 2, 1, 5, 1, 2, 2, 11, 7, 2, 1, 1, 1, 6. RLE produce los 27 *runs*: 0, 1, 7, eol, 2, 1, 5, eol, 5, 1, 2, eol, 0, 3, 2, 3, eol, 0, 3, 2, 3, eol, 0, 3, 2, 3, eol, 4, 1, 3, eol, 3, 1, 4, eol.

4.62: Un segmento de línea recta desde a hasta b es un ejemplo de una curva unidimensional que pasa a través de cada punto en el intervalo a, b.

4.63: La clave es darse cuenta de que P_0 es un solo punto, y P_1 se construye conectando nueve copias de P_0 con segmentos rectos. De manera similar, P_2 está formado por nueve copias de P_1, en orientaciones diferentes, conectadas por segmentos (los segmentos discontinuos en la Figura Sol.42).

4.64: Escritas en el sistema binario, las coordenadas son: $(1101, 0110)$. Repetimos cuatro veces, cada vez tomando un bit de la coordenada x y 1 bit de la coordenada y para formar un par (x, y). Los pares son: 10, 11, 01, 10. Del primero de ellos se obtiene [de la Tabla 4.170(1)] 01. Del segundo par [también de la Tabla 4.170(1)] 10. Del tercer par [de la Tabla 4.170(1)] 11, y del último par [de la Tabla 4.170(4)] 01. Por lo tanto, el resultado es: $01|10|11|01 = 109$.

4.65: Tabla Sol.43 muestra que este recorrido se basa en la secuencia 2114.

1 :	2	↑	1	→	1	↓	4
2 :	1	→	2	↑	2	←	3
3 :	4	↓	3	←	3	↑	2
4 :	3	←	4	↓	4	→	1

Tabla Sol.43: Las cuatro orientaciones de H_2.

4.66: Ésto es sencillo:

$$\begin{aligned}(00,01,11,10) &\to (000,001,011,010)\,(100,101,111,110)\\ &\to (000,001,011,010)\,(110,111,101,100)\\ &\to (000,001,011,010,110,111,101,100)\,.\end{aligned}$$

4.67: El área gris de la Figura 4.171c (pág. 501) se identifica mediante la cadena 2011.

4.68: Esta particular numeración hace que la conversión entre el número de un subcuadrado y sus coordenadas en la imagen sea fácil. (Suponemos que el origen se encuentra en la esquina inferior izquierda de la imagen y que las coordenadas de la imagen varían entre 0 y 1.) Como ejemplo, la traducción de los dígitos del número 1032 a binario produce como resultado: (01) (00) (11) (10). Los primeros bits de estos grupos, constituyen la coordenada x del subcuadrado; y los segundos bits, constituyen la coordenada y. Por consiguiente, las coordenadas en la imagen del subcuadrado 1032 son $x = 0{,}0011_2 = {}^3\!/_{16}$, $y = 0{,}1010_2 = {}^5\!/_8$, como puede verificarse directamente en la Figura 4.171c.

4.69: Ésto se muestra en la Figura Sol.44.

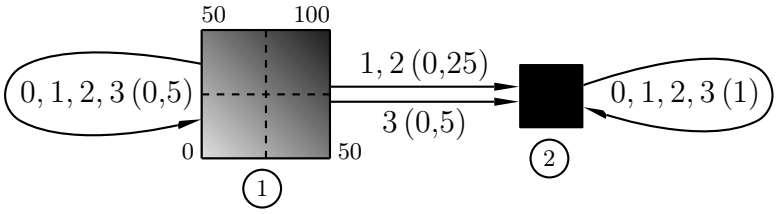


Figura Sol.44: Un grafo de dos estados.

4.70: Esta imagen se describe mediante la función:

$$f\,(x,y) = \begin{cases} x+y, & \text{si } x+y \le 1,\\ 0, & \text{si } x+y > 1.\end{cases}$$

4.71: El grafo tiene cinco estados, por lo que cada matriz de transición es de tamaño 5×5. Con el

```
dim=256;
for i=1:dim
  for j=1:dim
    m(i,j)=(i+j-2)/(2*dim-2);
  end
end
m
```

Figura Sol.45: Código en Matlab para la matriz $m_{i,j} = (i+j)/2$.

cálculo directo desde el grafo se obtiene:

$$W_0 = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0{,}5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & -0{,}5 & 0 & 0 & 1{,}5 \\ 0 & -0{,}25 & 0 & 0 & 1 \end{pmatrix}, \quad W_3 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1{,}5 \end{pmatrix},$$

$$W_1 = W_2 = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0{,}25 & 0 & 0 & 0{,}5 \\ 0 & 0 & 0 & 1{,}5 & 0 \\ 0 & 0 & -0{,}5 & 1{,}5 & 0 \\ 0 & -0{,}375 & 0 & 0 & 1{,}25 \end{pmatrix}.$$

La distribución final es el vector de cinco componentes:

$$F = (0{,}25,\ 0{,}5,\ 0{,}375,\ 0{,}4125,\ 0{,}75)^T.$$

4.72: Una forma de especificar el centro es la construcción de la cadena $033\ldots3$. Ésto produce:

$$\begin{aligned} \psi_i(03\ldots3) &= (W_0 \cdot W_3 \cdots W_3 \cdot F)_i \\ &= \begin{pmatrix} 0{,}5 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0{,}5 & 0{,}5 \\ 0 & 1 \end{pmatrix} \cdots \begin{pmatrix} 0{,}5 & 0{,}5 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0{,}5 \\ 1 \end{pmatrix}_i \\ &= \begin{pmatrix} 0{,}5 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0{,}5 \\ 1 \end{pmatrix}_i = \begin{pmatrix} 0{,}5 \\ 1 \end{pmatrix}_i. \end{aligned}$$

4.73: La Figura Sol.45 muestra el código en Matlab para calcular una matriz como la de la Figura 4.175 (pág. 506).

4.74: Con un examen directo de la gráfica se obtienen los valores de ψ_i:

$$\begin{aligned} \psi_i(0) &= (W_0 \cdot F)_i = (0{,}5, 0{,}25, 0{,}75, 0{,}875, 0{,}625)_i^T, \\ \psi_i(01) &= (W_0 \cdot W_1 \cdot F)_i = (0{,}5, 0{,}25, 0{,}75, 0{,}875, 0{,}625)_i^T, \\ \psi_i(0) &= (W_1 \cdot F)_i = (0{,}375, 0{,}5, 0{,}61875, 0{,}43125, 0{,}75)_i^T, \\ \psi_i(00) &= (W_0 \cdot W_0 \cdot F)_i = (0{,}25, 0{,}125, 0{,}625, 0{,}8125, 0{,}5625)_i^T, \\ \psi_i(03) &= (W_0 \cdot W_3 \cdot F)_i = (0{,}75, 0{,}375, 0{,}625, 0{,}5625, 0{,}4375)_i^T, \\ \psi_i(3) &= (W_3 \cdot F)_i = (0, 0{,}75, 0, 0, 0{,}625)_i^T, \end{aligned}$$

y los valores de f:

$$\begin{aligned} f(0) = I \cdot \psi(0) = 0{,}5, \quad f(01) = I \cdot \psi(01) = 0{,}5, \quad f(1) = I \cdot \psi(1) = 0{,}375, \\ f(00) = I \cdot \psi(00) = 0{,}25, \quad f(03) = I \cdot \psi(03) = 0{,}75, \quad f(3) = I \cdot \psi(3) = 0. \end{aligned}$$

4.75: La Figura Sol.46a,b muestra los seis estados y las 21 aristas. Utilizamos la notación $i\,(q,t)\,j$ para la arista con el número de cuadrante q y la transformación t, desde el estado i al estado j. Este GFA es más complejo que los previos ya que la imagen original tiene menos autosimilitud.

4.76: La transformación puede ser escrita $(x,y) \rightarrow (x,-x+y)$, por lo que: $(1,0) \rightarrow (1,-1)$, $(3,0) \rightarrow (3,-3)$, $(1,1) \rightarrow (1,0)$ y $(3,1) \rightarrow (3,-2)$. Por consiguiente, el rectángulo original es transformado en un paralelogramo.

4.77: La explicación es, que los dos conjuntos de transformaciones producen el mismo triángulo de Sierpiński pero en diferentes tamaños y orientaciones.

4.78: Las tres transformaciones reducen una imagen a la mitad de su tamaño original. Además, ω_2 y ω_3 colocan dos copias de la imagen en los desplazamientos relativos $(0, 1/2)$ y $(1/2, 0)$, como se muestra en la Figura Sol.47. El resultado es el conocido *gasket*[8] de Sierpiński pero en una orientación diferente.

4.79: Hay $32 \times 32 = 1024$ rangos y $(256-15) \times (256-15) = 58\,081$ dominios. Por consiguiente, el número total de pasos es $1024 \times 58\,081 \times 8 = 475\,799\,552$, aún un gran número. PIFS es, por lo tanto, de cómputo intensivo.

4.80: Supongamos que la imagen tiene G niveles de gris. Una buena medida de la pérdida de datos es la diferencia entre el valor promedio de un píxel descomprimido y su valor correcto, expresado en el número de niveles de gris. Para valores grandes de G (cientos de niveles de gris) una diferencia promedio de $\log_2 G$ niveles de gris (o menor) se considera satisfactoria.

5.1: Una página escrita es un ejemplo. Una persona puede colocar marcas —como texto, expresiones matemáticas, y dibujos— en una página y leerla más tarde . Ésta es una representación bidimensional de la información de la página. La página puede ser escaneada más adelante por, e.g., una máquina de fax, y su contenido transmitido como un flujo unidimensional de bits que constituye una representación diferente de la misma información.

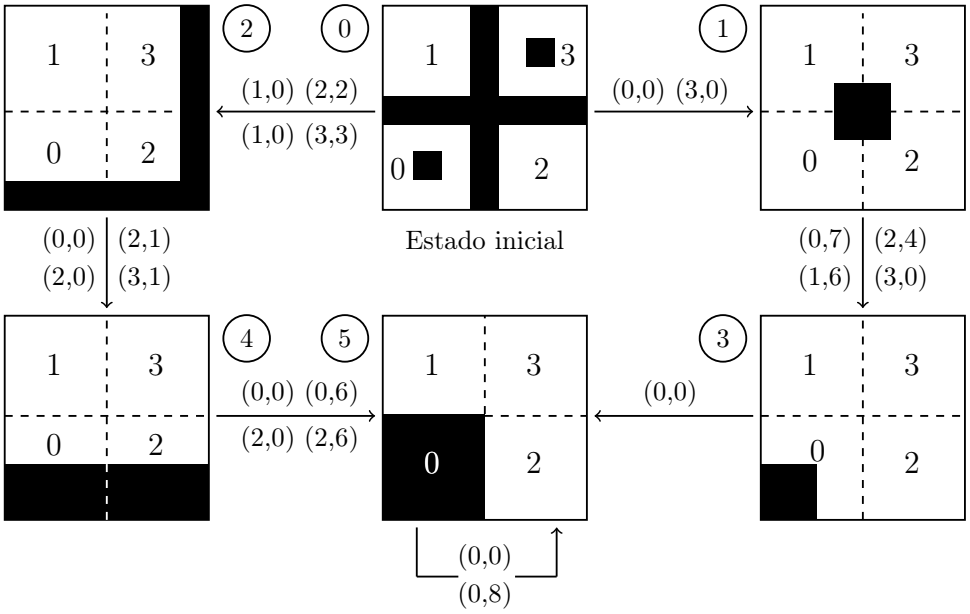
5.2: La Figura Sol.48 muestra $f\,(t)$ y tres copias desplazadas de la onda, para $a = 1$ y $b = 2$, 4, y 6. El producto interno $W\,(a,b)$ se representa debajo de cada copia de la onda. Es fácil ver cómo los productos internos están afectados por el aumento de la frecuencia.

La tabla de la Figura Sol.49 muestra 15 valores de $W\,(a,b)$, para $a = 1$, 2, y 3 y para $b = 2$ *a* 6. La gráfica de densidad de la figura, donde las partes brillantes corresponden a valores grandes, muestra gráficamente ésos valores. Para cada valor de a, los rendimientos de los valores de la CWT disminuyen con b, lo que refleja el hecho de que la frecuencia de $f\,(t)$ se incrementa con t. Los cinco valores de $W\,(1,b)$ son pequeños y muy similares, mientras que los cinco valores de $W\,(3,b)$ son más grandes y más diferentes. Ésto demuestra que el escalado de la onda hacia arriba hace a la CWT más sensible a los cambios de frecuencia en $f\,(t)$.

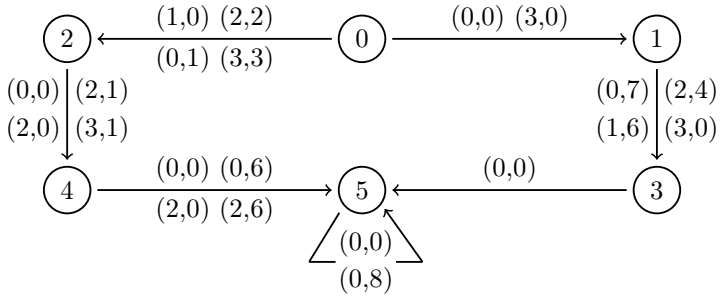
5.3: La Figura 5.11c muestra estas ondas.

5.4: La Figura Sol.50a muestra una sencilla imagen, de 8×8 con una línea diagonal por encima de la diagonal principal. Las Figuras Sol.50b,c muestran los dos primeros pasos de su descomposición en pirámide. Es evidente que los coeficientes de transformada en la subbanda inferior derecha (HH) muestran un artefacto en diagonal situado por encima de la diagonal principal. También es fácil ver que subbanda superior izquierda (LL) es una versión de baja resolución de la imagen original.

[8] Empaque, conglomerado, etc.



(a)



(b)

0(0,0)1	0(3,0)1	0(0,1)2	0(1,0)2	0(2,2)2	0(3,3)2	1(0,7)3
1(1,6)3	1(2,4)3	1(3,0)3	2(0,0)4	2(2,0)4	2(2,1)4	2(3,1)4
3(0,0)5	4(0,0)5	4(0,6)5	4(2,0)5	4(2,6)5	5(0,0)5	5(0,8)5

Figura Sol.46: Un GFA para el Ejercicio 4.75.

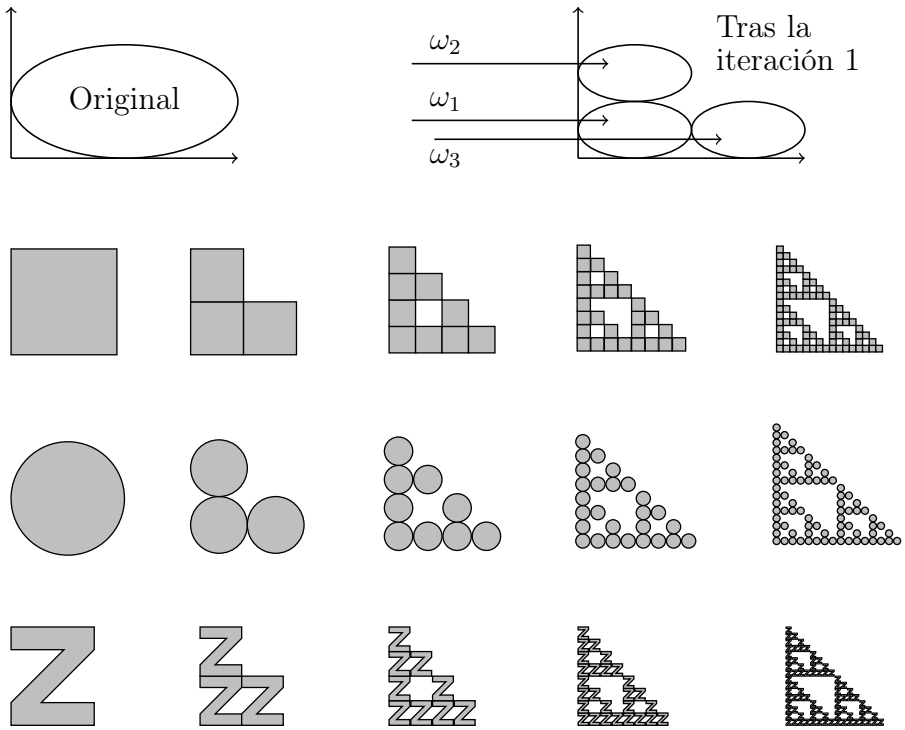


Figura Sol.47: Otro *gasket* de Sierpiński .

5.5: La media puede calcularse fácilmente. Resulta ser 131,375, que es exactamente 1/8 de 1051. La razón del coeficiente de la parte superior izquierda de la transformada es ocho veces superior al promedio que el código en Matlab proporcionó en los cálculos utilizando $\sqrt{2}$ en lugar de 2 (véase la función `individ(n)` en la Figura 5.22).

5.6: La Figura Sol.51a−c muestra los resultados de la reconstrucción para los coeficientes 3277, 1639, y 820, respectivamente. A pesar de la fuerte pérdida de coeficientes wavelet, sólo se aprecia una muy pequeña pérdida de calidad de imagen. El número de coeficientes wavelet es, por supuesto, el mismo que la resolución de la imagen $128 \times 128 = 16\,384$. La utilización de 820 de los 16 384 coeficientes corresponde a descartar el 95 % de los coeficientes más pequeños de la transformada (nótese, sin embargo, que algunos de los coeficientes eran originalmente cero, por lo que la pérdida real puede ascender a menos del 95 %).

5.7: El código en Matlab de la Figura Sol.52 calcula W como el producto de las tres matrices A_1, A_2, y A_3 y calcula los coeficientes de la matriz 8×8 de la transformada. Observe que el valor de la parte superior izquierda 131,375 es el promedio de todos los 64 píxeles de la imagen.

5.8: El vector $x = (\ldots, 1, -1, 1, -1, 1, \ldots)$ de valores alternos es transformado por el filtro paso bajo H_0 en un vector de todo ceros.

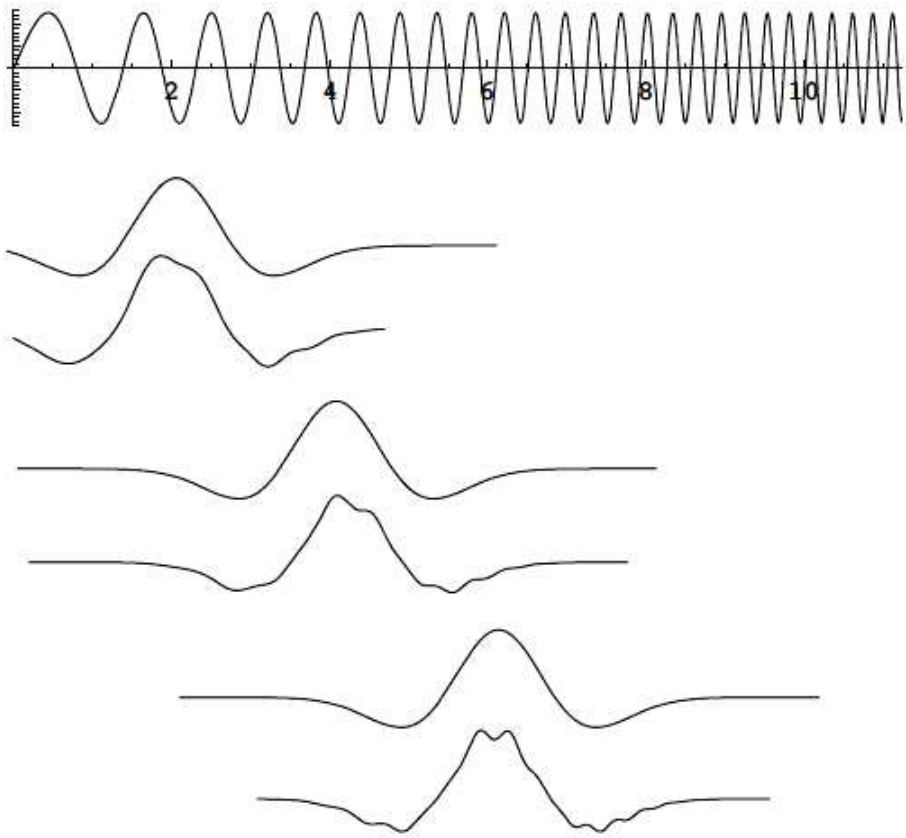


Figura Sol.48: Un producto interno para $a = 1$ y $b = 2, 4, 6$.

a	$b = 2$	3	4	5	6
1	0,032512	0,000299	$1{,}10923 \times 10^{-6}$	$2{,}73032 \times 10^{-9}$	$8{,}33866 \times 10^{-11}$
2	0,510418	0,212575	0,0481292	0,00626348	0,00048097
3	0,743313	0,629473	0,380634	0,173591	0,064264

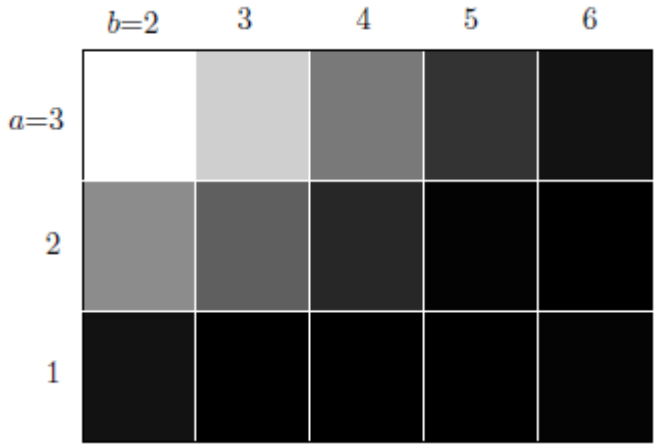


Figura Sol.49: Quince valores y gráfica de densidad de $W\,(a, b)$.

$$\begin{array}{cccccccc}
12 & 16 & 12 & 12 & 12 & 12 & 12 & 12 \\
12 & 12 & 16 & 12 & 12 & 12 & 12 & 12 \\
12 & 12 & 12 & 16 & 12 & 12 & 12 & 12 \\
12 & 12 & 12 & 12 & 16 & 12 & 12 & 12 \\
12 & 12 & 12 & 12 & 12 & 16 & 12 & 12 \\
12 & 12 & 12 & 12 & 12 & 12 & 16 & 12 \\
12 & 12 & 12 & 12 & 12 & 12 & 12 & 12 \\
12 & 12 & 12 & 12 & 12 & 12 & 12 & 16
\end{array}$$

(a)

$$\begin{array}{cccc|cccc}
14 & 12 & 12 & 12 & \underline{4} & 0 & 0 & 0 \\
12 & 14 & 12 & 12 & 0 & 4 & 0 & 0 \\
12 & 14 & 12 & 12 & 0 & \underline{4} & 0 & 0 \\
12 & 12 & 14 & 12 & 0 & 0 & 4 & 0 \\
12 & 12 & 14 & 12 & 0 & 0 & \underline{4} & 0 \\
12 & 12 & 12 & 14 & 0 & 0 & 0 & 4 \\
12 & 12 & 12 & 14 & 0 & 0 & 0 & \underline{4} \\
12 & 12 & 12 & 12 & 0 & 0 & 0 & 0
\end{array}$$

(b)

$$\begin{array}{cccc|cccc}
13 & 13 & 12 & 12 & \underline{2} & 2 & 0 & 0 \\
12 & 13 & 13 & 12 & 0 & \underline{2} & 2 & 0 \\
12 & 12 & 13 & 13 & 0 & 0 & \underline{2} & 2 \\
\underline{12} & \underline{12} & \underline{12} & \underline{13} & \underline{0} & \underline{0} & \underline{0} & \underline{2} \\
\hline
2 & \underline{2} & 0 & 0 & \underline{4} & \underline{4} & 0 & 0 \\
0 & 2 & \underline{2} & 0 & 0 & \underline{4} & \underline{4} & 0 \\
0 & 0 & 2 & \underline{2} & 0 & 0 & \underline{4} & \underline{4} \\
0 & 0 & 0 & 2 & 0 & 0 & 0 & \underline{4}
\end{array}$$

(c)

Figura Sol.50: La descomposición en subbandas de una línea diagonal.

5.9: Para estos filtros, las reglas 1 y 2 implican:

$$\begin{gathered}
h_0^2(0)+h_0^2(1)+h_0^2(2)+h_0^2(3)+h_0^2(4)+h_0^2(5)+h_0^2(6)+h_0^2(7)=1, \\
h_0(0)h_0(2)+h_0(1)h_0(3)+h_0(2)h_0(4)+h_0(3)h_0(5)+h_0(4)h_0(6)+h_0(5)h_0(7)=0, \\
h_0(0)h_0(4)+h_0(1)h_0(5)+h_0(2)h_0(6)+h_0(3)h_0(7)=0, \\
h_0(0)h_0(6)+h_0(1)h_0(7)=0,
\end{gathered}$$

y las reglas 3–5 producen:

$$\begin{aligned}
f_0 &= (h_0(7), h_0(6), h_0(5), h_0(4), h_0(3), h_0(2), h_0(1), h_0(0)), \\
h_1 &= (-h_0(7), h_0(6), -h_0(5), h_0(4), -h_0(3), h_0(2), -h_0(1), h_0(0)), \\
f_1 &= (h_0(0), -h_0(1), h_0(2), -h_0(3), h_0(4), -h_0(5), h_0(6), -h_0(7)).
\end{aligned}$$

Los ocho coeficientes se muestran en la Tabla 5.35 (éste es el filtro de Daubechies D8).

5.10: La Figura Sol.53 muestra el código en Matlab de la función de la transformada wavelet inversa `iwt1(wc, grueso, filtro)` y un test.

5.11: La Figura Sol.54 muestra el resultado del difuminado de la imagen de "Lena". Las partes (a) y (b) muestran el árbol de multirresolución logarítmica y la estructura de subbanda, respectivamente. La parte (c) muestra los resultados de la cuantificación. Los coeficientes de la transformada de las subbandas 5–7 se han dividido en dos, y todos los coeficientes de las subbandas 8–13 han sido eliminados. Podemos decir que la imagen borrosa de la parte (d) ha sido reconstruida a partir de los coeficientes de las subbandas 1–4 ($1/64$-avo del número total de coeficientes de la transformada) y la mitad de los coeficientes de las subbandas 5–7 (la mitad de $3/64$, ó $3/128$). En promedio, la imagen ha sido reconstruida a partir de $5/128 \approx 0{,}039$ ó $3{,}9\,\%$ coeficientes de la transformada. Observe que se utilizó el filtro D8 de Daubechies en los cálculos. Animo a los lectores a utilizar este código y experimentar con el rendimiento de otros filtros.

5.12: Éstas se escriben en la forma: `a-=b/2; b+=a;`.

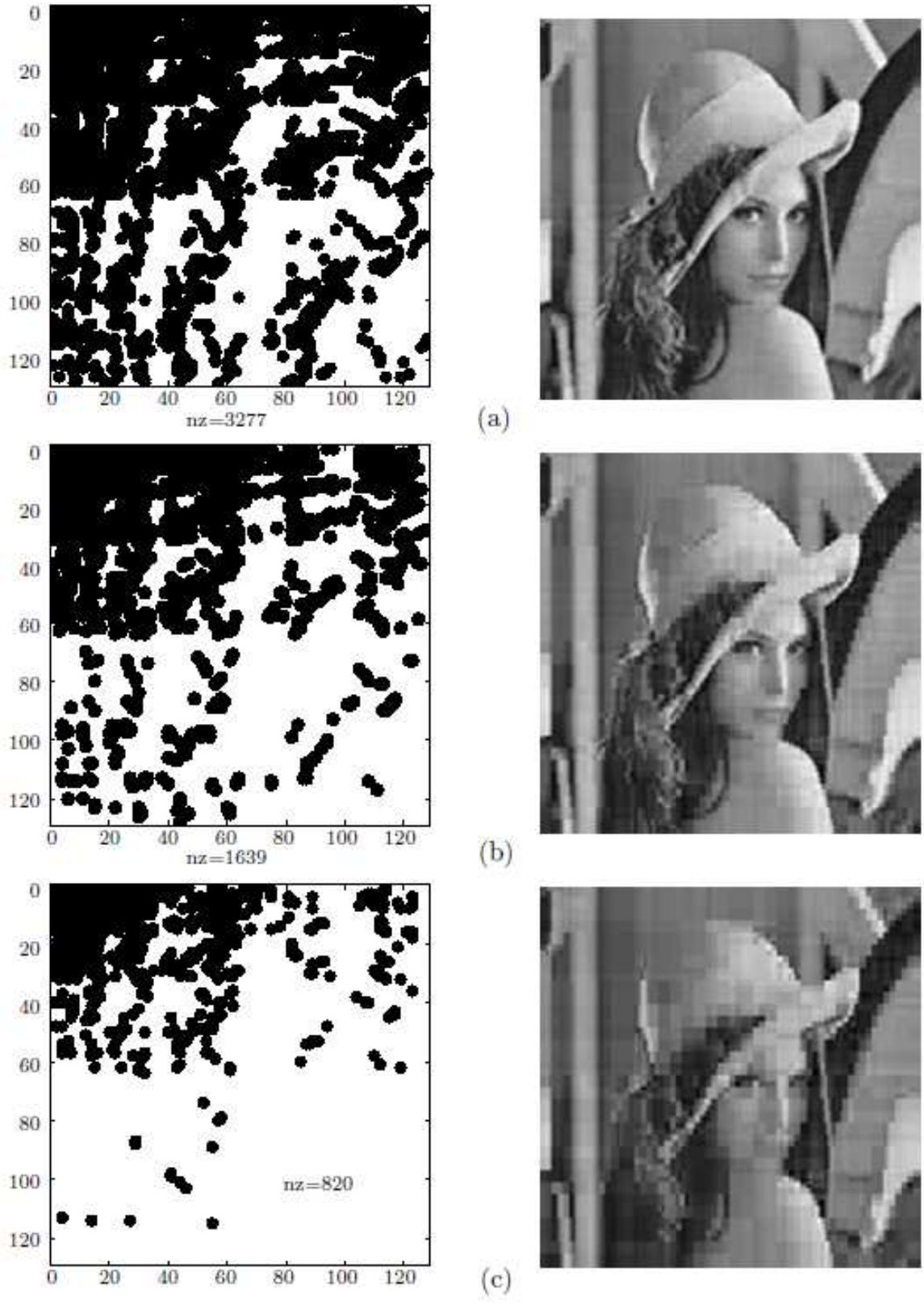


Figura Sol.51: Imagen 128×128 de Lena: tres reconstrucciones con pérdidas.

```
clear
a1=[1/2 1/2 0 0 0 0 0 0; 0 0 1/2 1/2 0 0 0 0;
  0 0 0 0 1/2 1/2 0 0; 0 0 0 0 0 0 1/2 1/2;
  1/2 -1/2 0 0 0 0 0 0; 0 0 1/2 -1/2 0 0 0 0;
  0 0 0 0 1/2 -1/2 0 0; 0 0 0 0 0 0 1/2 -1/2];
% a1*[255; 224; 192; 159; 127; 95; 63; 32];
a2=[1/2 1/2 0 0 0 0 0 0; 0 0 1/2 1/2 0 0 0 0;
  1/2 -1/2 0 0 0 0 0 0; 0 0 1/2 -1/2 0 0 0 0;
  0 0 0 0 1 0 0 0; 0 0 0 0 0 1 0 0;
  0 0 0 0 0 0 1 0; 0 0 0 0 0 0 0 1];
a3=[1/2 1/2 0 0 0 0 0 0; 1/2 -1/2 0 0 0 0 0 0;
  0 0 1 0 0 0 0 0; 0 0 0 1 0 0 0 0;
  0 0 0 0 1 0 0 0; 0 0 0 0 0 1 0 0;
  0 0 0 0 0 0 1 0; 0 0 0 0 0 0 0 1];
w=a3*a2*a1;
dim=8; fid=fopen('8x8','r');
img=fread(fid,[dim,dim])'; fclose(fid);
w*img*w'% Resultado de la transformada
```

131,375	4,250	−7,875	−0,125	−0,25	−15,5	0	−0,25
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
12,000	59,875	39,875	31,875	15,75	32,0	16	15,75
12,000	59,875	39,875	31,875	15,75	32,0	16	15,75
12,000	59,875	39,875	31,875	15,75	32,0	16	15,75
12,000	59,875	39,875	31,875	15,75	32,0	16	15,75

Figura Sol.52: Código y resultados para el cálculo de la matriz W y de la transformada $W \cdot I \cdot W^T$.

5.13: Sumamos la Ecuación (5.13) para todos los valores de l para obtener:

$$\sum_{l=0}^{2^{j-1}-1} s_{j-1,l} = \sum_{l=0}^{2^{j-1}-1} (s_{j,2l} + d_{j-1,l}/2) = \frac{1}{2} \sum_{l=0}^{2^{j-1}-1} (s_{j,2l} + s_{j,2l+1}) = \frac{1}{2} \sum_{l=0}^{2^j-1} s_{j,l}.$$

Por lo tanto, la media del conjunto s_{j-1} es igual a:

$$\frac{1}{2^{j-1}} \sum_{l=0}^{2^{j-1}-1} s_{j-1,l} = \frac{1}{2^{j-1}} \frac{1}{2} \sum_{l=0}^{2^j-1} s_{j,l} = \frac{1}{2^j} \sum_{l=0}^{2^j-1} s_{j,l}$$

la media del conjunto s_j.

5.14: El código de la Figura Sol.55 produce la expresión:

$$0{,}0117\mathbf{P}_1 - 0{,}0977\mathbf{P}_2 + 0{,}05859\mathbf{P}_3 + 0{,}05859\mathbf{P}_4 - 0{,}0977\mathbf{P}_5 + 0{,}0117\mathbf{P}_6.$$

5.15: El código en Matlab[9] de la Figura Sol.56 realiza el trabajo y produce el vector de enteros de la transformada $y = (111, -1, 84, 0, 120, 25, 84, 3)$. La transformada inversa genera un vector z

[9]La función `floor` proporciona un redondeo a $-\infty$.

```
function dat=iwt1(wc,grueso,filtro)
% Transformada Wavelet Discreta Inversa
dat=wc(1:2^grueso);
n=length(wc); j=log2(n);
for i=grueso:j-1
  dat=ILoPass(dat,filtro)+ ...
    IHiPass(wc((2^(i)+1):(2^(i+1))),filtro);
end

function f=ILoPass(dt,filtro)
f=iconv(filtro,AltrntZro(dt));

function f=IHiPass(dt,filtro)
f=aconv(mirror(filtro),rshift(AltrntZro(dt)));

function sgn=mirror(filt)
% devuelve los coeficientes filtrados con signos alternos
sgn=-((-1).^(1:length(filt))).*filt;

function f=AltrntZro(dt)
% devuelve un vector de longitud 2*n con ceros
% situados entre valores consecutivos
n =length(dt)*2; f =zeros(1,n);
f(1:2:(n-1))=dt;
```

Un sencillo test de `iwt1` es:

```
n=16; t=(1:n)./n;
dat=sin(2*pi*t)
filt=[0.4830 0.8365 0.2241 -0.1294];
wc=fwt1(dat,1,filt)
rec=iwt1(wc,1,filt)
```

Figura Sol.53: Código para la transformada wavelet discreta inversa 1D.

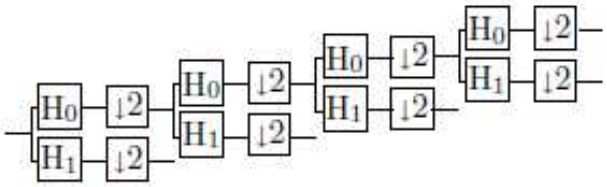
idéntico a los datos originales x. Obsérvese cómo los coeficientes de detalle son mucho menores que los promedios ponderados. Nótese también que los arrays (matrices o arreglos) en Matlab se encuentran indexados desde 1, mientras que la discusión en el texto asume que los arrays están indexados desde 0. Ésto produce que los valores del índice sean distintos en la Figura Sol.56.

5.16: Para el caso $M_C = 3$, $M_R = 4$, y $M = 5$ las primeras seis imágenes —de g_0 a g_5— tendrán las dimensiones:

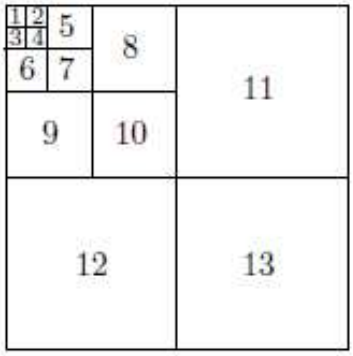
$$\left(4 \cdot 2^5 + 1 \times 3 \cdot 2^5 + 1\right) = 129 \times 97, \quad 65 \times 49, \quad 33 \times 25, \quad 17 \times 13, \quad y\ 9 \times 7.$$

5.17: En el paso de ordenación de la tercera iteración, el codificador transmite el número $l = 3$ (el número de coeficientes $c_{i,j}$ en nuestro ejemplo que satisfacen $2^{12} \leq |c_{i,j}| < 2^{13}$), seguido por los tres pares de coordenadas $(3, 3)$, $(4, 2)$, y $(4, 1)$, y por los signos de los tres coeficientes. En el paso de refinamiento transmite los seis bits $cdefgh$. Éstos son el 13-ésimo bit más significativo de los coeficientes transmitidos en todas las iteraciones previas.

La información recibida hasta el momento permite al decodificador mejorar aún más los apro-



(a)



(b)

(c)

(d)

```
clear, colormap(gray);
filename='lena128'; dim=128;
fid=fopen(filename,'r');
img=fread(fid,[dim,dim])';
filt=[0.23037,0.71484,0.63088,-0.02798, ...
  -0.18703,0.03084,0.03288,-0.01059];
fwim=fwt2(img,3,filt);
figure(1), imagesc(fwim), axis square
  fwim(1:16,17:32)=fwim(1:16,17:32)/2;
  fwim(1:16,33:128)=0;
  fwim(17:32,1:32)=fwim(17:32,1:32)/2;
  fwim(17:32,33:128)=0;
  fwim(33:128,:)=0;
figure(2), colormap(gray), imagesc(fwim)
rec=iwt2(fwim,3,filt);
figure(3), colormap(gray), imagesc(rec)
```

Figura Sol.54: Difuminado como resultado de una cuantificación tosca.

```
Clear[p,a,b,c,d,e,f];
p[t_]:=a t^5+b t^4+c t^3+d t^2+e t+f;
Solve[{p[0]==p1, p[1/5.]==p2, p[2/5.]==p3,
p[3/5.]==p4, p[4/5.]==p5, p[1]==p6}, {a,b,c,d,e,f}];
sol=ExpandAll[Simplify[%]];
Simplify[p[0.5] /.sol]
```

Figura Sol.55: Código para un polinomio de interpolación de grado 5.

```
clear;
N=8; k=N/2;
x=[112,97,85,99,114,120,77,80];
% IWT directa en y
for i=0:k-2,
  y(2*i+2)=x(2*i+2)-floor((x(2*i+1)+x(2*i+3))/2);
end;
y(N)=x(N)-x(N-1);
y(1)=x(1)+floor(y(2)/2);
for i=1:k-1,
  y(2*i+1)=x(2*i+1)+floor((y(2*i)+y(2*i+2))/4);
end;
% IWT inversa en z
z(1)=y(1)-floor(y(2)/2);
for i=1:k-1,
  z(2*i+1)=y(2*i+1)-floor((y(2*i)+y(2*i+2))/4);
end;
for i=0:k-2,
  z(2*i+2)=y(2*i+2)+floor((z(2*i+1)+x(2*i+3))/2);
end;
z(N)=y(N)+z(N-1);
```

Figura Sol.56: Código en Matlab para la IWT directa e inversa.

ximadamente 16 coeficientes. Los primeros nueve se convienten en:

$$c_{2,3} = s1ac0\ldots0,\ c_{3,4} = s1bd0\ldots0,\ c_{3,2} = s01e00\ldots0,$$
$$c_{4,4} = s01f00\ldots0,\ c_{1,2} = s01g00\ldots0,\ c_{3,1} = s01h00\ldots0,$$
$$c_{3,3} = s0010\ldots0,\ c_{4,2} = s0010\ldots0,\ c_{4,1} = s0010\ldots0,$$

y los siete restantes no cambian.

5.18: Se resuelve la sencilla ecuación $10\times 2^{20}\times 8 = (500x)\times(500x)\times 8$, para obtener $x^2 = 40$ pulgadas cuadradas. Si la tarjeta es cuadrada, es de aproximadamente 6,32 pulgadas de lado. Tal tarjeta tiene 10 impresiones arrolladas (de alrededor de $1{,}5\times 1{,}5$ cada una), dos impresiones simples de los pulgares (cercanas a $0{,}875\times 0{,}875$ cada una), y las impresiones simultáneas de ambas manos (de aproximadamente $3{,}125\times 1{,}875$ cada una). Todas las dimensiones están en pulgadas[10].

5.19: El bit de 10 es codificado, como es habitual, en la pasada 2. El bit 1 se codifica en la pasada 1, ya que este coeficiente es todavía insignificante pero tiene vecinos importantes. Este bit es 1, por lo que el coeficiente 1 se vuelve significativo (un hecho que no se utiliza más adelante). Además,

[10] Una pulgada son: 25, 4 mm (2,54 cm).

dicho bit es el primer 1 de este coeficiente, por lo que el bit de signo del coeficiente se codifica siguiendo el citado bit. Los bits de los coeficientes 3 y -7 son codificados en la pasada 2, ya que estos coeficientes son significativos.

6.1: Es fácil calcular que $525 \times 4/3 = 700$ píxeles.

6.2: La altura vertical de la imagen en el televisor de 27 in. (pulgadas) del autor es de 16 in., que se traduce en una distancia de visualización de $7{,}12 \times 16 = 114$ in. o cerca de 9,5 pies[11]. Es fácil ver que las líneas de exploración individuales son visibles a cualquier distancia más corta que aproximadamente 6 pies.

6.3: Tres ejemplos comunes son: (1) una cámara de vigilancia, (2) una película antigua muda, que se restaura y convierte desde un rollo de película a vídeo, y (3) una presentación de video tomada bajo el agua.

6.4: La razón áurea $\phi \approx 1{,}618$ ha sido considerada tradicionalmente la relación de aspecto que es más agradable a la vista. Esto sugiere que 1,77 es la mejor relación de aspecto.

6.5: Imagine una cámara panorámica trabajando de izquierda a derecha. Los nuevos objetos entrarán en el campo de visión desde la derecha todo el tiempo. Un bloque en el lado derecho del frame, por lo tanto, puede contener objetos que no existían en el frame anterior.

6.6: Puesto que $(4, 4)$ está en el centro del "+", el valor de s se reduce a la mitad: a 2. El siguiente paso busca los cuatro bloques etiquetados 4, centrados en $(4, 4)$. Suponiendo que el mejor emparejamiento es $(6, 4)$, se buscan los dos bloques con etiqueta 5. Suponiendo que $(6, 4)$ es el más adecuado, s se reduce a la mitad: a 1, y se buscan los ocho bloques etiquetados 6. El diagrama muestra que el mejor emparejamiento encontrado está finalmente en la ubicación $(7, 4)$.

6.7: Esta figura consta de 18×17 macrobloques, y cada macrobloque está formado por seis bloques de muestras de 8×8. El número total de muestras es, por lo tanto, de $18 \times 17 \times 6 \times 64 = 117\,504$.

6.8: El tamaño de la categoría de cero es 0, por lo que se emite el código 100, seguido de cero bits. El tamaño de la categoría de 4 es 3, por lo que se emite primero el código 110, seguido por los tres bits menos significativos de 4, que son 100.

6.9: La secuencia en zigzag es:

$$118, 2, 0, -2, \underbrace{0, \ldots 0}_{13}, -1, 0, \ldots.$$

Los pares run-level son $(0, 2)$, $(1, -2)$ y $(13, -1)$, por lo que los códigos finales son (observe los bits de signo a continuación de los códigos run-level):

$$0100\,0 \mid 000110\,1 \mid 00100000\,1 \mid 10,$$

(sin las barras verticales).

6.10: No hay coeficientes distintos de cero, no hay códigos run-level, únicamente el código EOB de 2 bits. Sin embargo, en la codificación nointra, tal bloque es codificado de una manera especial.

7.1: Un libro puede contener una media de 60 caracteres por línea, 45 líneas por página y 400 páginas. Ésto se traduce en $60 \times 45 \times 400 = 1\,080\,000$ caracteres, que requieren un byte de almacenamiento para cada uno.

[11] Un pie son: 30,48 cm

7.2: El periodo de una onda es su velocidad dividida por su frecuencia. Para un sonido obtenemos:

$$\frac{34380\,\text{cm/s}}{22000\,\text{Hz}} = 1{,}562\,\text{cm}, \quad \frac{34380}{20} = 1719\,\text{cm}.$$

7.3: El logaritmo (en base 10) de x es un número y tal que $10^y = x$. El número 2 es el logaritmo de 100, ya que $10^2 = 100$. De forma similar, 0,3 es el logaritmo de 2, ya que $10^{0,3} \approx 2$. Además, el logaritmo en base b de x es un número y tal que $b^y = x$ (para cualquier número real $b > 1$).

7.4: Cada duplicación de la intensidad del sonido incrementa el nivel de dB en 3. Por lo tanto, la diferencia de 9 dB $(3 + 3 + 3)$ entre A y B corresponde a tres duplicaciones de la intensidad del sonido. Por consiguiente, la fuente B es $2 \cdot 2 \cdot 2 = 8$ veces más fuerte (ruidosa) que la fuente original A.

7.5: Cada 0 se puede traducir como un silencio y cada muestra 1, como el mismo tono. El resultado sería un zumbido no uniforme. Estos sonidos eran comunes en los primeros ordenadores personales.

7.6: Este experimento debería repetirse con varias personas, preferiblemente de diferentes edades. La persona debe colocarse en una cámara de sonido aislado, y allí se genera un tono puro de frecuencia f. La amplitud del tono debe aumentarse gradualmente desde cero hasta que la persona apenas puede oirlo. Si ésto sucede en un valor de decibelios d, debe dibujarse el punto (d, f). Ésto hay que repetirlo para muchas frecuencias hasta obtener un gráfica similar a la Figura 7.5a.

7.7: En primer lugar, seleccionamos los ítems idénticos. Si todo $s\,(t - i)$ es igual a s, la Ecuación (7.7) obtiene la misma s. A continuación, seleccionamos los valores de una línea recta. Dados los cuatro valores a, $a + 2$, $a + 4$, y $a + 6$, la Ecuación (7.7) produce $a + 8$: el valor lineal siguiente. Finalmente, seleccionamos valores aproximadamente equidistantes en un círculo. La coordenada y de los puntos del primer cuadrante de un círculo puede calcularse mediante $y = \sqrt{r^2 - x^2}$. Seleccionamos los cuatro puntos con coordenadas x: 0, $0{,}08r$, $0{,}16r$ y $0{,}24r$; calculamos sus coordenadas y para $r = 10$, y los sustituimos en la Ecuación (7.7). El resultado es 9,96926, que comparado con la coordenada y real para $x = 0{,}32r$ —que es $\sqrt{r^2 - (0{,}32r)^2} = 9{,}47418$—, da una diferencia de aproximadamente un 5 %. El código que hizo los cálculos se muestra en la Figura Sol.57.

```
(* Puntos en un círculo. Usado en el ejercicio para
comprobar la predicción de 4º orden en FLAC *)
r = 10;
ci[x_] := Sqrt[100 - x^2];
ci[0.32r]
4ci[0] - 6ci[0.08r] + 4ci[0.16r] - ci[0.24r]
```

Figura Sol.57: Código para comprobar la predicción de 4º orden.

7.8: Imagine que el sonido que se está comprimiendo contiene un segundo de un tono puro (de sólo una frecuencia). Este segundo será digitalizado en 44 100 muestras consecutivas por canal. Las muestras indican amplitudes, por lo que no tienen que ser iguales. Sin embargo, después de la filtración, sólo una subbanda (aunque en la práctica quizás dos subbandas) tendrá señales distintas de cero. Todas las otras subbandas corresponden a frecuencias diferentes, por lo que contendrán señales que son cero o muy cercanas a cero.

7.9: Suponiendo que un nivel de ruido P_1 se traduce a x decibelios:

$$20 \log \left(\frac{P_1}{P_2} \right) = x \text{ dB SPL},$$

lo que da lugar a la relación:

$$20 \log \left(\frac{\sqrt[3]{2} P_1}{P_2} \right) = 20 \left[\log_{10} \sqrt[3]{2} + \log \left(\frac{P_1}{P_2} \right) \right] = 20 \left(0{,}1 + x/20 \right) = x + 2.$$

Por lo tanto, aumentando el nivel de sonido en un factor de $\sqrt[3]{2}$ se incrementa el nivel de decibelios en 2 dB SPL.

7.10: Para una frecuencia de muestreo de 44 100 $^{\text{muestras}}/_{\text{seg.}}$, los cálculos son similares. El decodificador debe decodificar $^{44\,100}/_{384} \approx 114{,}84$ frames por segundo. Por consiguiente, cada frame tiene que ser decodificado en aproximadamente 8,7 ms. Con el fin de emitir 114,84 frames en 64 000 bits, cada uno debe tener $B_f = 557$ bits disponibles para codificarlo. En consecuencia, el número de slots por bloque es $^{557}/_{32} \approx 17{,}41$. Por lo tanto, el última (18-ésimo) slot no está lleno y tiene que ser completado.

7.11: La Tabla 7.58 muestra que el factor de escala es 111 y la información seleccionada es 2. La tercera regla en la Tabla 7.59 muestra que una scfsi de 2 significa que sólo se codifica un factor de escala, ocupando tan solo seis bits en la salida comprimida. El decodificador asigna estos seis bits como los valores de los tres factores de escala.

7.12: Los parámetros típicos para la capa II son: (1) una tasa de muestreo de 48 000 muestras/seg, (2) una tasa de bits de 64 000 bits/seg, y (3) 1152 señales cuantificadas por bloque. El decodificador tiene que decodificar $^{48\,000}/_{1152} = 41{,}66$ bloques por segundo. Por consiguiente, cada bloque tiene que ser decodificado en 24 ms. Con el fin de sacar 41,66 bloques en 64 000 bits, cada uno debe tener $B_f = 1536$ bits disponibles para codificarlo.

7.13: Un programa para reproducir archivos `.mp3` es un *decodificador* MPEG de III capas, no un codificador. La decodificación es mucho más simple, ya que no utiliza un modelo psicoacústico, ni tiene que prever ecos previos ni mantener el repositorio de bits.

8.1: Debido a que la cadena original `S` se puede reconstruir a partir de `L`, pero no de `F`.

8.2: Una aplicación directa de la Ecuación (8.1), ocho veces más produce:

$$\begin{aligned}
\texttt{S}[10-1-2] &= \texttt{L}\left[\texttt{T}^2[\texttt{I}]\right] = \texttt{L}\left[\texttt{T}\left[\texttt{T}^1[\texttt{I}]\right]\right] = \texttt{L}[\texttt{T}[7]] = \texttt{L}[6] = \texttt{i};\\
\texttt{S}[10-1-3] &= \texttt{L}\left[\texttt{T}^3[\texttt{I}]\right] = \texttt{L}\left[\texttt{T}\left[\texttt{T}^2[\texttt{I}]\right]\right] = \texttt{L}[\texttt{T}[6]] = \texttt{L}[2] = \texttt{m};\\
\texttt{S}[10-1-4] &= \texttt{L}\left[\texttt{T}^4[\texttt{I}]\right] = \texttt{L}\left[\texttt{T}\left[\texttt{T}^3[\texttt{I}]\right]\right] = \texttt{L}[\texttt{T}[2]] = \texttt{L}[3] = \texttt{␣};\\
\texttt{S}[10-1-5] &= \texttt{L}\left[\texttt{T}^5[\texttt{I}]\right] = \texttt{L}\left[\texttt{T}\left[\texttt{T}^4[\texttt{I}]\right]\right] = \texttt{L}[\texttt{T}[3]] = \texttt{L}[0] = \texttt{s};\\
\texttt{S}[10-1-6] &= \texttt{L}\left[\texttt{T}^6[\texttt{I}]\right] = \texttt{L}\left[\texttt{T}\left[\texttt{T}^5[\texttt{I}]\right]\right] = \texttt{L}[\texttt{T}[0]] = \texttt{L}[4] = \texttt{s};\\
\texttt{S}[10-1-7] &= \texttt{L}\left[\texttt{T}^7[\texttt{I}]\right] = \texttt{L}\left[\texttt{T}\left[\texttt{T}^6[\texttt{I}]\right]\right] = \texttt{L}[\texttt{T}[4]] = \texttt{L}[5] = \texttt{i};\\
\texttt{S}[10-1-8] &= \texttt{L}\left[\texttt{T}^8[\texttt{I}]\right] = \texttt{L}\left[\texttt{T}\left[\texttt{T}^7[\texttt{I}]\right]\right] = \texttt{L}[\texttt{T}[5]] = \texttt{L}[1] = \texttt{w};\\
\texttt{S}[10-1-9] &= \texttt{L}\left[\texttt{T}^9[\texttt{I}]\right] = \texttt{L}\left[\texttt{T}\left[\texttt{T}^8[\texttt{I}]\right]\right] = \texttt{L}[\texttt{T}[1]] = \texttt{L}[9] = \texttt{s};
\end{aligned}$$

La cadena original `swiss␣miss` está, en efecto, reconstruida en `S`, de derecha a izquierda.

8.3: La Figura Sol.58 muestra las rotaciones de S y la matriz ordenada. La última columna, L de Sol.58b pasa a ser idéntica a S, por lo que S = L = `sssssssssh`. Puesto que A = (`s`, `h`), una compresión mover–al–frente de L produce $C = (1, 0, 0, 0, 0, 0, 0, 0, 0, 1)$. Como C contiene sólo los dos valores 0 y 1, pueden servir como sus propios códigos de Huffman, por lo que el resultado final es 1000000001: ¡1 bit por carácter!

`ssssssssh`	`hsssssssss`
`sssssssshs`	`shssssssss`
`ssssssshss`	`sshsssssss`
`sssssshsss`	`ssshssssss`
`ssssshssss`	`sssshsssss`
`sssshsssss`	`ssssshssss`
`ssshssssss`	`sssssshsss`
`sshsssssss`	`ssssssshss`
`shssssssss`	`sssssssshs`
`hsssssssss`	`sssssssssh`
(a)	(b)

Figura Sol.58: Permutaciones de "`sssssssssh`".

8.4: El codificador comienza con T [0], que contiene 5. El primer elemento de L es, por lo tanto, el último símbolo de la permutación 5. Esta permutación comienza en la posición 5 de S, por lo que su último elemento está en la posición 4. Por consiguiente, el codificador tiene que ir recorriendo los símbolos S [T [$i-1$]] para $i = 0, \ldots, n-1$, donde la notación $i-1$ debe ser interpretada cíclicamente (i.e., $0-1$ debe ser $n-1$). Cada símbolo S [T [$i-1$]] que se encuentre, se comprime utilizando mover–al–frente. El valor de I, es la posición donde T contiene 0. En nuestro ejemplo, T [8] = 0, por lo que I = 8.

8.5: El primer elemento de un triplete es la distancia entre dos entradas del diccionario, la que coincida mejor con el contenido y la que cuadre mejor con el contexto. En este caso no hay coincidencia de contenido, no hay distancia, por lo que cualquier número podría servir como primer elemento, siendo 0 (el más pequeño) la mejor elección.

8.6: Debido a que las tres líneas están clasificadas en orden ascendente. Las dos últimas líneas de la Tabla 8.13c no están ordenadas. Por ello, la parte `zz...z` de la cadena `S` debe ir precedida y seguida por bits complementarios.

8.7: El codificador coloca `S` entre las dos entradas de la lista asociatica ordenada y escribe el índice (codificado) de la entrada —por encima o por debajo de `S`—, en la cadena comprimida. Cuanto menor sea el número de entradas, más pequeño es el índice, y mejor es la compresión.

8.8: El contexto 5 se compara con los tres contextos restantes —6, 7, y 8—, y es más similar al contexto 6 (comparten el sufijo "`b`"). El contexto 6 se compara con el 7 y 8, y puesto que no comparten ningún sufijo, se selecciona el contexto 7 —el más corto de los dos—. El contexto 8 —el que queda— es, por supuesto, el último de todos. La clasificación final del contexto es:

$$1 \to 3 \to 4 \to 0 \to 5 \to 6 \to 7 \to 8.$$

8.9: La Ecuación (8.3) muestra que a la tercera "`a`" se le asigna el rango 1, y a la "`b`" y la "`a`" que le siguen se les asignan los rangos 2 y 3, respectivamente.

																		0		λ	u
												0		λ	u			1		ubla	d
								0		λ	u	1		ubla	d			2		ub	l
				0		λ	u	1		ubla	x	2		ub	l			3		ublad	i
		0		λ	u	1		ub	l	2		ub	l	3		ublad	x	4		ubladi	x
0		λ	u	1		ub	x	2		ubl	x	3		ubl	a	4		ubl	a	5		ubl	a
1		u	x	2		u	b	3		u	b	4		u	b	5		u	b	6		u	b
	(a)				(b)				(c)				(d)				(e)				(f)		

Tabla Sol.59: La construcción de las listas ordenadas para `ubladiu`.

8.10: La Tabla Sol.59 muestra los contextos en orden. La Ecuación (Sol.1) muestra la clasificación del contexto en cada paso.

$$\begin{array}{llll} 0, & 0 \to 2, & 1 \to 3 \to 0, & \\ u & u \quad b & l \quad b \quad u & \\ 0 \to 2 \to 3 \to 4, & & 2 \to 4 \to 1 \to 5 \to 0, & \\ u \quad l \quad a \quad b & & l \quad a \quad d \quad b \quad u & \\ 3 \to 5 \to 2 \to 6 \to 5 \to 0. & & & \\ i \quad a \quad l \quad b \quad d \quad u & & & \end{array} \tag{Sol.1}$$

La salida final es "`u2b3l4a5d6i6`". Observe que cada uno de los símbolos de entrada distintos aparece una vez en esta salida en bruto (formato raw).

8.11: Todos los n_1 bits de la cadena L_1 necesitan ser escritos en el flujo de salida. Ésto ya muestra que no va a haber compresión. La cadena L_2 consta de n_1/k unos, por lo que todos tienen que escribirse en la secuencia de salida. De forma similar, la cadena L_3 está formada por n_1/k^2 unos, y así sucesivamente. Por consiguiente, el tamaño de la secuencia de salida es:

$$n_1 + \frac{n_1}{k} + \frac{n_1}{k^2} + \frac{n_1}{k^3} + \cdots + \frac{n_1}{k^m} = n_1 \frac{k^{m+1}-1}{k^m\,(k-1)},$$

para un cierto valor de m. El límite de esta expresión, cuando $m \to \infty$, es $n_1 k/\,(k-1)$. Para $k = 2$, ésto equivale a $2n_1$. Para grandes valores de k este límite está siempre entre n_1 y $2n_1$.

Para el lector curioso, indicamos aquí cómo se calcula la suma anterior. Dada la serie:

$$S = \sum_{i=0}^{m} \frac{1}{k^i} = 1 + \frac{1}{k} + \frac{1}{k^2} + \frac{1}{k^3} + \cdots + \frac{1}{k^{m-1}} + \frac{1}{k^m},$$

multiplicamos ambos miembros por $1/k$:

$$\frac{S}{k} = \frac{1}{k} + \frac{1}{k^2} + \frac{1}{k^3} + \cdots + \frac{1}{k^m} + \frac{1}{k^{m+1}} = S + \frac{1}{k^{m+1}} - 1,$$

y restamos:

$$\frac{S}{k}\,(k-1) = \frac{k^{m+1}-1}{k^{m+1}} \to S = \frac{k^{m+1}-1}{k^m\,(k-1)}.$$

8.12: La cadena de entrada se compone de:

1. Un *run* de tres grupos de ceros, codificado como 10|1, ya que 3 se encuentra en segunda posición en la clase 2.

2. El grupo distinto de cero 0100, codificado como 111100.

3. Otro *run* de tres grupos de ceros, de nuevo codificado como 10|1.

4. El grupo distinto de cero 1000, codificado como 01100.

5. Un *run* de cuatro grupos de ceros, codificado como 010|00, ya que 4 está en la primera posición en la clase 3.

6. 0010, codificado como 111110.

7. Un *run* de dos grupos ceros, codificado como 10|0.

La salida es, por lo tanto: 10111110010101100010001111110100, una cadena de 31 bits.

8.13: La cadena de entrada se compone de:

1. Un *run* de tres grupos de ceros, codificado como R_2R_1, ó 101|11.

2. El grupo distinto de cero 0100, codificado como 00100.

3. Otro *run* de tres grupos de ceros, de nuevo codificado como 101|11.

4. El grupo distinto de cero 1000, codificado como 01000.

5. Un *run* de cuatro grupos de ceros, codificado como $R_4 = 1001$.

6. 0010, codificado como 00010.

7. Un *run* de dos grupos de ceros, codificado como $R_2 = 101$.

La salida es, por lo tanto: 10111001001011101000100100010101, una cadena de 32 bits.

8.14: La cadena de entrada se compone de:

1. Un *run* de tres grupos de ceros, codificado como F_3, ó 1001.

2. El grupo distinto de cero 0100, codificado como 00100.

3. Otro *run* de tres grupos de ceros, de nuevo codificado como 1001.

4. El grupo distinto de cero 1000, codificado como 01000.

5. Un *run* de cuatro grupos de ceros, codificado como $F_3F_1 = 1001|11$.

6. 0010, codificado como 00010.

7. Un *run* de dos grupos de ceros, codificado como $F_2 = 101$.

La salida es, por lo tanto: 10010010010010100010011100010101, una cadena de 32 bits.

8.15: Sí, cuando se encuentran en diferentes cuadrantes o subcuadrantes. Los píxeles 123 y 301, por ejemplo, son adyacentes en la Figura 4.157; pero tienen diferentes prefijos.

8.16: No, porque todos los prefijos tienen la misma probabilidad de ocurrencia. En nuestro ejemplo, los prefijos son de cuatro bits de longitud y los 16 prefijos posibles tienen la misma probabilidad porque un píxel puede estar situado en cualquier parte de la imagen. Un código de Huffman construido para 16 símbolos equiprobables, tiene un tamaño promedio de cuatro bits por símbolo, de modo que no habría ninguna ganancia. Lo mismo es cierto para los sufijos.

8.17: Ésto es posible, pero impone severas limitaciones en el tamaño de la cadena. Para volver a organizar una cadena unidimensional en un cubo de cuatro dimensiones, el tamaño de la cadena debe ser de 2^{4n}. Si el tamaño de la cadena pasa a ser $2^{4n} + 1$, tiene que extenderse a $2^{4(n+1)}$, lo que aumenta su tamaño por un factor de 16. Es posible reorganizar la cadena como una caja rectangular —no sólo un cubo—, pero entonces su tamaño tendrá que ser de la forma $2^{n1}2^{n2}2^{n3}2^{n4}$, donde los cuatro ni's son números enteros.

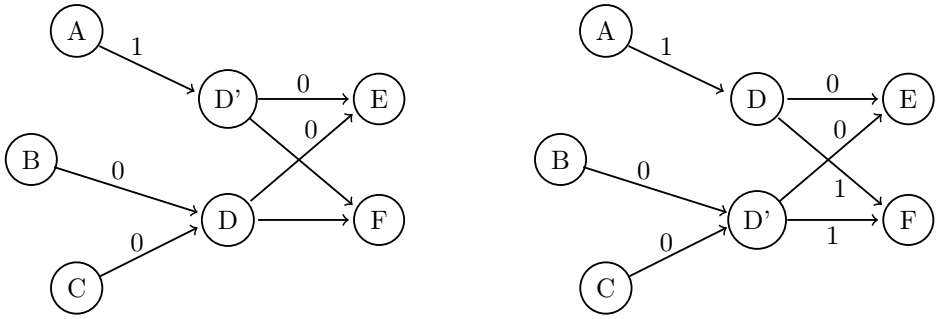


Figura Sol.60: Nuevo estado clonado D'.

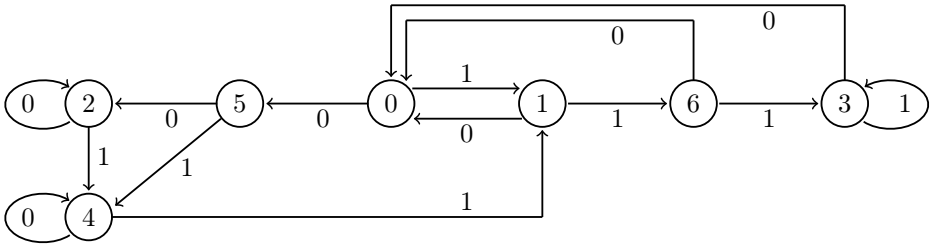


Figura Sol.61: Estado 6 añadido.

8.18: El algoritmo LZW, que comienza con el alfabeto entero almacenado al principio de su diccionario, es un ejemplo de tal método. Sin embargo, puede diseñarse una versión adaptada de LZW para comprimir palabras en lugar de caracteres individuales.

8.19: Una mejor opción para las coordenadas pueden ser valores relativos (o *desplazamientos*). Cada par (x, y) puede especificar la posición de un carácter relativo a su predecesor. Esto da como resultado un número menor para las coordenadas, y los números más pequeños son más fáciles de comprimir.

8.20: Es posible encontrar tales letras en otros alfabetos "exóticos", pero un ejemplo más común es una caja rectangular que contiene texto. Las cuatro rectas que constituyen una caja semejante deben ser consideradas una huella; sin embargo, los caracteres de texto dentro de la caja deben ser identificados como huellas separadas.

8.21: Ésto garantiza que las dos probabilidades sumen 1.

8.22: La Figura Sol.60 muestra cómo el estado A lleva al nuevo estado D' que, a su vez, permite ir a los estados E y F. Observe cómo los estados B y C no han cambiado. Puesto que el nuevo estado D' es idéntico a D, es posible enlazar A con D o con D' (la clonación puede realizarse de dos maneras diferentes pero idénticas). Los recuentos originales del estado D, deben dividirse ahora entre D y D' proporcionalmente a los recuentos de las transiciones $A \to D$ y B, $C \to D$.

8.23: La Figura Sol.61 muestra el nuevo estado 6 tras realizar la operación 1, $1 \to 6$. Su salida 1 es idéntica a la del estado 1, y su salida 0 es una copia de la salida 0 del estado 3.

8.24: Una respuesta precisa requiere de muchos experimentos con varios archivos de datos. Pensando un poco, sin embargo, nos damos cuenta de que cuanto mayor es k, mejor es el modelo inicial que se crea cuando se desecha el viejo. Por consiguiente, los valores más grandes de k minimizan la

Entrada	Gramática
S → abcdbcabcdbc	S → CC
	A → bc
	C → aAdA

Figura Sol.62: La mejora de la gramática de la Figura 8.41.

pérdida de compresión. Sin embargo, los valores muy grandes pueden producir un modelo inicial que ya es grande y no puede crecer mucho más. El mejor valor para k, por tanto, es aquel que produce un modelo inicial lo suficientemente grande como para proporcionar información acerca de las correlaciones en los datos recientes, pero lo suficientemente pequeño para que pueda crecer antes de que también tenga que ser desechado.

8.25: El número de puntos marcados se pueden escribir $8\,(1+2+3+5+8+13) = 256$, y los números entre paréntesis son los números de Fibonacci.

8.26: La probabilidad condicional $P\,(D_i \mid D_i)$ es muy pequeña. Un segmento que apunta en la dirección D_i puede estar precedido por otro segmento que apunta en la misma dirección sólo si la curva original es recta o muy cercana a una recta por más de 26 unidades de coordenadas (la mitad del ancho de la rejilla S_{13}).

8.27: Observe que un punto tiene dos coordenadas. Si cada coordenada ocupa ocho bits, entonces la utilización de números de Fibonacci reduce las coordenadas de 16 bits a números de 8 bits, una relación de compresión de 0,5. El uso de códigos de Huffman puede típicamente reducir este número de 8 bits a (en promedio) un código de 4 bits, y el uso del modelo de Markov puede quizás acortar éste en otro bit. El resultado es una razón de compresión estimada de $3/16 = 0{,}1875$. Si cada coordenada es un número 16 bits, entonces esta relación mejora a $3/32 = 0{,}09375$.

8.28: La gramática resultante, más corta, se muestra en la Figura Sol.62. Es una regla y un símbolo más corta.

8.29: Porque la generación de la regla C ha estado infrautilizando la regla B (i.e., se usa una sola vez).

8.30: La Regla S está formada por dos copias de la regla A. La primera vez que se encuentra la regla A, se envía su contenido aBdB. Ésto implica mandar la regla B dos veces. La primera vez que se necesita la regla B, se envía su contenido bc (y el decodificador no sabe que la cadena bc que está recibiendo pertenece a una regla). La segunda vez que se utiliza la regla B, se envía el par $(1, 2)$ (desplazamiento 1, contador 2). El decodificador identifica el par y lo usa para establecer la regla 1 → bc. El envío de la primera copia de la regla A, por lo tanto, equivale a mandar abcd$\,(1, 2)$. Por consiguiente, envía la segunda copia de la regla A como el par $(0, 4)$, ya que A empieza en el desplazamiento 0 en S y su longitud es 4. El decodificador identifica este par y lo utiliza para establecer la regla 2 → a 1 d 1. El resultado final es, por lo tanto, abcd$\,(1, 2)\,(0, 4)$.

8.31: En cada uno de estos casos, el codificador elimina una arista de la frontera e inserta dos nuevas aristas. Hay una ganancia neta de una arista.

8.32: Ellos crean los triángulos $(18, 2, 3)$ y $(18, 3, 4)$, y reducen la frontera a la secuencia de vértices:

$$(4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18)\,.$$

A.1: Éste es una combinación de ''4x y ''9, o ''49.

B.1: A continuación se muestran los valores de X. Cada uno tiene una probabilidad de $1/8$:

$$\begin{array}{cccc} X(HHH)=3, & X(HHT)=2, & X(HTH)=2, & X(HTT)=1, \\ X(THH)=2, & X(THT)=1, & X(TTH)=1, & X(TTT)=0. \end{array}$$

B.2: La definición de esperanza implica que $E(X)=v$. El valor esperado de una variable aleatoria constante es un valor constante.

B.3: Denotamos $m=E(X)$. De $\mathrm{Var}(X)=E\left[(X-m)^2\right]$ obtenemos:

$$\begin{aligned} \mathrm{Var}(X) &= E\left(X^2-2Xm+m^2\right) \\ &= E\left(X^2\right)+E\left(-2mX\right)+E\left(m^2\right) \\ &= E\left(X^2\right)-2mE(X)+m^2 \\ &= E\left(X^2\right)-2m^2+m^2 \\ &= E\left(X^2\right)-m^2. \end{aligned}$$

B.4: La probabilidad de sacar un seis doble con dos dados es $1/36$. La probabilidad del complemento es $35/36$. La probabilidad de sacar un seis doble en la primera tirada, o en la segunda, ..., o en la 24 es:

$$1-\underbrace{(35/36)(35/36)\cdots(35/36)}_{24}\approx 1-0{,}5086=0{,}4914.$$

Para el juego B, la probabilidad de sacar un seis es $1/6$, su complemento es $5/6$, por lo que la probabilidad de sacar un seis en cuatro intentos es:

$$1-(5/6)(5/6)(5/6)(5/6)\approx 1-0{,}482=0{,}518;$$

ligeramente mayor que la probabilidad de ganar el juego A.

B.5: Este problema es fácil de resolver intuitivamente. Una vez que B se ha retirado, una de las dos empresas restantes se adjudica el contrato. Todo lo que tenemos que hacer para encontrar sus nuevas posibilidades es ampliar las viejas de tal manera que sumen 1. Dado que las viejas posibilidades suman hasta $3/5$, tienen que ser escaladas en $5/3$ para llevar su nueva suma a 1. Las nuevas posibilidades son, por lo tanto, $(2/5)(5/3)=2/3$ y $(1/5)(5/3)=1/3$.

A continuación, utilizamos las probabilidades condicionales para resolver el mismo problema. Estamos buscando las probabilidades condicionales $P\left(A|\overline{B}\right)$ y $P\left(C|\overline{B}\right)$. Sabemos que $P(B)$ era $2/5$, por lo que $P\left(\overline{B}\right)=1-P(B)=3/5$. La cantidad $P\left(A\cdot\overline{B}\right)$ es la probabilidad de que A gane *y* B no gane. Es igual a $P(A)$, ya que si A gana, B no puede ganar. La Ecuación (B.2), por lo tanto, produce las probabilidades condicionales:

$$\begin{aligned} P\left(A|\overline{B}\right) &= \frac{P\left(A\cdot\overline{B}\right)}{P\left(\overline{B}\right)} = \frac{2/5}{3/5}=\frac{2}{3}, \\ P\left(C|\overline{B}\right) &= \frac{P\left(C\cdot\overline{B}\right)}{P\left(\overline{B}\right)} = \frac{1/5}{3/5}=\frac{1}{3}. \end{aligned}$$

B.6: Aplicando el teorema de Bayes, los resultados son 0,24, 0,24, y 0,36, respectivamente. El conocimiento de que el estudiante consiga una A, hace menos probable que él seleccione matemáticas, más probable que seleccione biología, y no afecta mucho a las probabilidades de que seleccione física o química.

B.7: Utilizando el software matemático adecuado, es fácil obtener esta integral por separado para los valores negativos y los no negativos de x.

$$\int L\left(V,x\right)dx=\begin{cases}\frac{-1}{V\exp\left(\sqrt{\frac{2}{V}}x\right)}, & x\geq 0,\\ \frac{1}{\sqrt{2V}}\exp\left(\sqrt{\frac{2}{V}}x\right), & x<0.\end{cases}$$

C.1: Un segmento de línea recta desde a hasta b es un ejemplo de curva unidimensional que pasa a través de cada punto en el intervalo a, b.

C.2: La clave es darse cuenta de que P_0 es un único punto, y P_1 se construye conectando nueve copias de P_0 con segmentos rectos. De manera similar, P_2 está formado por nueve copias de P_1, en diferentes orientaciones, conectados por segmentos (los segmentos discontinuos en la Figura Sol.42).

C.3: Escrito en binario, las coordenadas son (1101, 0110). Iteramos cuatro veces, cada vez tomando un bit de la coordenada x y 1 bit de la coordenada y para formar un par (x, y). Los pares son: 10, 11, 01, 10. El primero de ellos produce [a partir de la Tabla 4.170(1)] 01. El segundo par [también de la Tabla 4.170(1)] 10. El tercer par [desde la Tabla 4.170(1)] 11, y el último par [desde la Tabla 4.170 (4)] 01. El resultado es, pues, $01|10|11|01 = 109$.

C.4: La Tabla Sol.43 muestra que este recorrido se basa en la secuencia 2114.

C.5: Esto es sencillo:

$$\begin{aligned}(00,01,11,10)&\rightarrow(000,001,011,010)\,(100,101,111,110)\\&\rightarrow(000,001,011,010)\,(110,111,101,100)\\&\rightarrow(000,001,011,010,110,111,101,100)\,.\end{aligned}$$

D.1: Este es $j\cdot m+(i+1)$.

D.2: Tiene sentido almacenar el grado n del polinomio en la primera posición del array.

D.3: Si. En un árbol ternario, los tres hijos de un nodo a se almacenan en las ubicaciones $2a$, $2a+1$, y $2a+2$, y el padre de a puede encontrarse en la posición $[a/3]$ del array.

D.4: Los recorridos en pre-orden, en orden, y en orden por niveles son, respectivamente:

$$\begin{gathered}A,(B,(D,E)),(C,(F,(G,H)))\,,\\((D,B,E),A,(nulo,C,(G,F,H)))\,,\\(A,(B,C),(D,E,F),(G,H))\,.\end{gathered}$$

D.5: Cada uno de los 8 caracteres de un nombre puede ser una de las 26 letras o los diez dígitos, por lo que el número total de nombres es: $36^8 = 2\,821\,109\,907\,456$; cerca de 3 billones.

E.1: Una comprobación directa muestra que cuando se cambia sólo un único bit de algunas palabras de código de código$_2$, el resultado no es cualquiera de las otras palabras de código.

E.2: Una comprobación directa muestra que el código$_4$ tiene una distancia de Hamming de 4, que es más que suficiente para detectar todos los errores de 2 bits.

E.3: b_2 es la paridad de b_3, b_6, b_7, b_{10} y b_{11}. b_4 es la paridad de b_5, b_6 y b_7. b_8 es la paridad de b_9, b_{10}, y b_{11}.

Bits de	Bits de datos															
paridad	3	5	6	7	9	10	11	12	13	14	15	17	18	19	20	21
1	x	x		x	x		x		x		x	x		x		x
2	x		x	x		x	x			x	x		x	x		
4		x	x	x				x	x	x	x				x	x
8					x	x	x	x	x	x	x					
16												x	x	x	x	x

Tabla Sol.63: Código de Hamming para $m = 16$.

E.4: La Tabla Sol.63 resume las definiciones de los cinco bits de paridad requeridos para este caso.

E5: Es un código de tamaño variable (Capítulo 2). Es fácil ver que éste salisface la propiedad del prefijo.

F.1: Sí. Alterna entre los estados 1, 2, 3, 4, y 1.

H.1: Todos los tonos de gris.

H.2: Una superficie de color amarillo absorbe el azul y refleja el verde y el rojo.

H.3: ¡Sí! Tales son los tres colores que producen el blanco cuando se mezclan. Ejemplos son: rojo, verde, y azul; cian, magenta, y amarillo.

H.4: Para el punto 1, es fácil ver que $R = 0$ y $G = 1$. Puesto que está en el plano $U = 0$, donde $Y = B$, es fácil calcular que $B = 0{,}663$. Para el punto 2, tenemos $R = 0$ y $B = 1$. Puesto que está en el plano $Y = 0{,}3$, obtenemos $G = 0{,}317$. Similarmente, el punto 3 tiene $R = 0$ y se resuelven las dos ecuaciones $Y = 0{,}3$ y $U = 0$ para obtener $G = 0{,}453$ y $B = 0{,}3$.

H.5: Debido a que el blanco no es un color puro, sino que es una mezcla de todos los colores.

H.6: Recordemos que la suma de una díada es blanca. Puesto que el blanco luminoso está en la mitad de la línea que conecta c y d, se obtiene añadiendo cantidades iguales de ellos $(0{,}5c + 0{,}5d)$. Es por eso que son complementarios.

H.7: La saturación se refiere a la cantidad de blanco en un color. El punto f corresponde a la saturación total, mientras que el blanco luminoso no corresponde a ninguna saturación. La saturación del color del punto e es, por lo tanto, la razón de las distancias $f\omega/e\omega$.

H.8: Si extendemos la línea que pasa por ω y g, ésta intercepta a la curva espectral pura en la parte inferior, una zona que no se corresponde con ninguna longitud de onda. Por lo tanto, continuamos la línea en la dirección opuesta hasta que intercepta la curva espectral pura en h y digamos que la longitud de onda dominante del punto g es $497c$ (donde c es procede de "complemento").

H.9: Cálculos directos utilizando la matriz D_{44} producen las áreas de la Figura Sol.64. Las tres áreas tienen porcentajes de píxeles negros de $1/16$, $2/16$, y $16/16$, respectivamente.

H.10: Una aplicación directa de la Ecuación (H.2) produce:

$$A\,[x,y] = \begin{array}{ccc} 0 & 1 & 15 \\ 10001000\ldots & 10001000\ldots & 11111111\ldots \\ 00000000\ldots & 00000000\ldots & 11111111\ldots \\ 00000000\ldots & 00100010\ldots & 11111111\ldots \\ 00000000\ldots & 00000000\ldots & 11111111\ldots \end{array}$$

Figura Sol.64: Difuminado en orden: Tres áreas uniformes.

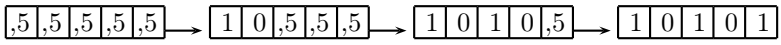
$$D_{88} = \begin{array}{|cccccccc|} \hline 0 & 32 & 8 & 40 & 2 & 34 & 10 & 42 \\ 48 & 16 & 56 & 24 & 50 & 18 & 58 & 26 \\ 12 & 44 & 4 & 36 & 14 & 46 & 6 & 38 \\ 60 & 28 & 52 & 20 & 62 & 30 & 54 & 22 \\ 3 & 35 & 11 & 43 & 1 & 33 & 9 & 41 \\ 51 & 19 & 59 & 27 & 49 & 17 & 57 & 25 \\ 15 & 47 & 7 & 39 & 13 & 45 & 5 & 37 \\ 63 & 31 & 55 & 23 & 61 & 29 & 53 & 21 \\ \hline \end{array}.$$

H.11: Un tablero de ajedrez. Esto puede verse manualmente simulando el algoritmo de la Figura H.22b para unos pocos píxeles.

H.12: Suponemos que la prueba es:

$$\text{si } p \geq 0{,}5, \text{ entonces } p := 1 \text{ si no } p := 0;\ \text{añadir el error } 0{,}5 - p \text{ al siguiente píxel } q.$$

El primer píxel queda, por lo tanto, establecido en 1; y el error $0{,}5 - 1 = -0{,}5$ es agregado al segundo píxel, cambiándolo de 0,5 a 0. El segundo píxel se establece en 0; y el error, que es $0 - 0 = 0$, es añadido al tercer píxel, dejándolo a 0,5. El tercer píxel queda, por lo tanto, establecido en 1; y el error $0{,}5 - 1 = -0{,}5$, es adicionado al cuarto píxel, cambiándolo de 0,5 a 0. Los resultados son:



H.13: El examen directo muestra que los *barons* (barones) son 62 y 63 y los *near-barones* (pseudo–barones) son 60 y 61.

H.14: Un patrón de tablero de ajedrez, similar a la producida por un difuminado de difusión. Esto puede verse manualmente ejecutando el algoritmo de la Figura H.24 para unos pocos píxeles.

H.15: Las clases 14, 15, y 10 son *barons*. Las clases 12 y 13 son *near-barons*. Los números de clase en las posiciones (i, j) y $(i, j + 2)$ suman 15.

I.1: Ésto es sencillo:

$$\mathbf{A} + \mathbf{B} = \begin{pmatrix} 8 & 10 & 12 \\ 8 & 10 & 12 \\ 8 & 10 & 12 \end{pmatrix}, \quad \mathbf{A} - \mathbf{B} = \begin{pmatrix} -6 & -6 & -6 \\ 0 & 0 & 0 \\ 6 & 6 & 6 \end{pmatrix}, \quad \mathbf{A} \times \mathbf{B} = \begin{pmatrix} 18 & 24 & 30 \\ 54 & 69 & 84 \\ 90 & 114 & 138 \end{pmatrix}$$

I.2: La Ecuación (I.4) produce:

$$T^{-1} = \frac{1}{1 \cdot 1 - 1 \cdot 1}\,(\cdots)\,,$$

que está indefinido. Por consiguiente, la matriz $\mathbf{T}$ es *singular*; ¡no tiene inversa! Ésto se puede entender fácilmente cuando pensamos en $\mathbf{T}$ como la matriz de coeficientes del sistema de Ecuaciones (I.3) visto anteriormente. Éste es un sistema de tres ecuaciones con tres incógnitas x, y,

y z, pero sus dos primeras ecuaciones son contradictorias. La primera de ellas dice que $x - y$ es igual a 1, mientras que la segunda dice que el mismo $x - y$ es igual a -2. Matemáticamente, tal sistema tiene una matriz de coeficientes singular.

I.3: Ésto se prueba fácilmente, mostrando que ambos productos punto, $(\mathbf{P} \times \mathbf{Q}) \bullet \mathbf{P}$ y $(\mathbf{P} \times \mathbf{Q}) \bullet \mathbf{Q}$, son iguales a cero.

$$(\mathbf{P} \times \mathbf{Q}) \bullet \mathbf{P} = P_1 (P_2 Q_3 - P_3 Q_2) + P_2 (-P_1 Q_3 + P_3 Q_1) + P_3 (P_1 Q_2 - P_2 Q_1) = 0.$$

Y similarmente para $(\mathbf{P} \times \mathbf{Q}) \bullet \mathbf{Q}$.

I.4: En el caso especial donde $\mathbf{i} = (1, 0, 0)$ y $\mathbf{j} = (0, 1, 0)$, es fácil verificar que el producto $i \times j$ es igual a $(0, 0, 1) = \mathbf{k}$. En consecuencia, el triplete $(\mathbf{i}, \mathbf{j}, \mathbf{i} \times \mathbf{j} = \mathbf{k})$, tiene el control del sistema de coordenadas (está, o bien orientado a la derecha (*right-handed*), o bien a izquierda (*left-handed*), dependiendo del sistema de coordenadas). En un sistema de coordenadas orientado a la derecha, la regla de la mano derecha hace que sea fácil predecir la dirección de $\mathbf{P} \times \mathbf{Q}$. La regla es: si su pulgar apunta en la dirección de $\mathbf{P}$ y su índice (el segundo dedo), en la dirección de $\mathbf{Q}$, entonces su dedo corazón (el del centro) apunta en la dirección de $\mathbf{P} \times \mathbf{Q}$. En un sistema de coordenadas orientado a la izquierda, se aplica una regla similar.

I.5: O bien tienen la misma dirección, o bien apuntan en direcciones opuestas.

I.6: Estamos buscando un vector $\mathbf{P}(t)$ que es lineal en t que satisfaga $\mathbf{P}(0) = \mathbf{P}_1$ y $\mathbf{P}(1) = \mathbf{P}_2$. Es fácil adivinar que:

$$\mathbf{P}(t) = (1 - t)\mathbf{P}_1 + t\mathbf{P}_2 = t(\mathbf{P}_2 - \mathbf{P}_1) + \mathbf{P}_1$$

satisface ambas condiciones.

I.7: Ésto no es especialmente difícil:

$$\mathbf{c} = \frac{2 \cdot 1 + 1 \cdot 0 + 3 \cdot (-1)}{1^2 + 0^2 + (-1)^2} (1, 0, -1) = (-1/2, 0, 1/2),$$
$$\mathbf{d} = \mathbf{a} - \mathbf{c} = (2{,}5,\ 1,\ 2{,}5).$$

I.8: Debido a la amplia difusión del uso de computadoras, el mundo de la ciencia y la ingeniería ha pasado de analógico a digital. En lugar de una función continua $f(t)$, ahora tenemos una n-tupla $(f_1, f_2, \ldots, f_n)$, y esto es un vector (de n dimensiones).

I.9: La regla de multiplicación produce: $(0, 1) \times (0, 1) = (-1, 0)$ ó $i \times i = -1$. La extraña regla de multiplicación de números complejos genera: $i^2 = -1$ ó $\sqrt{-1} = i$.

I.10: La regla de multiplicación produce: $(a, b) \times (-a, -b) = (0, 0)$, lo que justifica llamar $(-a, -b)$ al opuesto de (a, b).

I.11: Comenzamos con $i = \cos(\pi/2) + i \sin(\pi/2)$, de la cual obtenemos:

$$\sqrt{i} = \left(\cos\frac{\pi}{2} + i \sin\frac{\pi}{2}\right)^{1/2}.$$

Por el teorema de DeMoivre, ésto es igual a:

$$\cos\frac{\pi}{4} + i \sin\frac{\pi}{4} = \frac{1}{\sqrt{2}} + i\frac{1}{\sqrt{2}} = \frac{1 + i}{\sqrt{2}}. \qquad \text{(Sol.2)}$$

Una simple comprobación produce:

$$\left(\frac{1+i}{\sqrt{2}}\right)^2 = \frac{1+2i+i^2}{2} = i.$$

(El teorema de DeMoivre afirma que $\sin(nx)+i\cos(nx)$ es uno de los valores de $(\sin x + i\cos x)^n$.)

I.12: Empezamos con la elegante fórmula:

$$e^{it} = \cos t + i \sin t.$$

Sustituyendo $t = \pi/2$ se obtiene:

$$e^{i\pi/2} = i \sin(\pi/2) = i.$$

Ahora se elevan ambos miembros a la i-ésima potencia, obteniendo:

$$i^i = \left(e^{i\pi/2}\right)^i = e^{i^2\pi/2} = e^{-\pi/2}.$$

Sorprendentemente, éste es un número real. En el pasado, el cálculo de muchos dígitos de este número (u otros como él) implicó años de duro trabajo. Hoy, sin embargo, se necesita únicamente la sentencia en Matlab `vpa('i ^ i',140)` para generar, en menos de un segundo, el número de 140 dígitos:

0,20787 95763 50761 90854 69556 19834 97877 00338 77841
63176 96080 75135 88305 54198 77285 48213 97886 00277
86542 60353 40521 77330 72350 21808 19061 97030 74663
98700

Puesto que $a = \exp(\ln a)$ para cualquier a, también tenemos

$$e^{\ln i} = i = e^{i\pi/2},$$

de donde es evidente que $\ln i = i\pi/2$.

Por cierto, la función exponencial e^z se define para cualquier número complejo $z = x + iy$ como:

$$e^z = 1 + \frac{z}{1!} + \frac{z^2}{2!} + \frac{z^3}{3!} + \cdots .$$

Leonhard Euler, el gran matemático del siglo XVIII, introdujo la notación i para $\sqrt{-1}$ en 1777. Es interesante observar que los ingenieros eléctricos utilizan la notación j en su lugar, ya que tratan tensiones y corrientes eléctricas, y les resulta conveniente reservar la i para denotarlas.

I.13: El axioma y las reglas de producción son los mismos. Sólo cambian la cabecera inicial y el ángulo de rotación. La cabecera inicial puede ser 0 ó 90º y el ángulo de rotación, o bien 90º, o bien 270º.

Se puede encontrar un problema en casi todas las soluciones.
Desconocido.

Bibliografía

[DCC] Todas las URLs han sido revisadas y actualizadas a finales de julio de 2006 por los autores del libro original en inglés, y comprobadas de nuevo [12] y ampliadas por el traductor —en diciembre de 2011—. Todos los enlaces rotos informados al autor se añadirán a la lista de erratas en el sitio Web del libro.

El evento principal en la vida de la comunidad de compresión de datos es la conferencia anual sobre compresión de datos (DCC, consúltese Unirse a la Comunidad sobre Compresión de Datos[13]) cuyos procedimientos son publicados por el IEEE. Los editores han sido tradicionalmente James Andrew Storer y Martin Cohn. En vez de listar las muchas referencias que sólo difieren en un año, comenzamos esta bibliografía con una referencia genérica a la DCC, donde "XX" son los dos últimos dígitos del año de la conferencia.

Storer, James A., and Martin Cohn (eds.) (anual) DCC 'XX: *Data Compression Conference*, Los Alamitos, CA, IEEE Computer Society Press.

[3R 06] 3R (2006) es `http://f-cpu.seul.org/whygee/ddj-3r/ddj-3R.html` y 3R (2003-2007) `http://ygdes.com/ddj-3r/ddj-3r_compact.html`.

[7z 06] 7z (2006) es `http://www.7-zip.org/sdk.html`.

[Abramson 63] Abramson, N. (1963) *Information Theory and Coding*, New York, McGraw-Hill.

[Abousleman 06] Abousleman, Glen P. (2006) "Coding of Hyperspectral Imagery with Trellis-Coded Quantization," en G. Motta, F. Rizzo, and J. A. Storer, editors, *Hyperspectral Data Compression*, New York, Springer Verlag.

[acronyms 06] acronyms (2006) es: `http://acronyms.thefreedictionary.com/Refund-Anticipated+Return`.

[adaptiv9 06] adaptiv9 (2006) es `http://www.weizmann.ac.il/matlab/toolbox/filterdesign/`, archivo *adaptiv9.html*.

[Adelson et al. 87] Adelson, E. H., E. Simoncelli, y R. Hingorani (1987) "Orthogonal Pyramid Transforms for Image Coding," *Proceedings SPIE*, vol. 845, Cambridge, MA, pp. 50-58, October.

[adobepdf 06] adobepdf (2006) es `http://www.adobe.com/products/acrobat/adobepdf.html`.

[afb 06] afb (2006) es `http://www.afb.org/prodProfile.asp?ProdID=42`.

[12] Los enlaces rotos se han mantenido, pues proporcionan información de dónde o qué buscar, aunque se han añadido algunas URLs nuevas.

[13] Data Compression Community o, abreviado, DC community

[Ahmed et al. 74] Ahmed, N., T. Natarajan, y R. K. Rao (1974) "Discrete Cosine Transform," *IEEE Transactions on Computers*, **C-23**:90-93.

[Akansu y Haddad 92] Akansu, Ali, y R. Haddad (1992) *Multiresolution Signal Decomposition*, San Diego, CA, Academic Press.

[Anderson et al. 87] Anderson, K. L., et al., (1987) "Binary-Image-Manipulation Algorithm in the Image View Facility," *IBM Journal of Research and Development*, **31**(1): 16-31, January.

[Anedda y Felician 88] Anedda, C. y L. Felician (1988) "P-Compressed Quadtrees for Image Storing," *The Computer Journal*, **31**(4): 353-357.

[ATSC 06 y 10] ATSC (2006) es `http://www.atsc.org/standards/a_52b.pdf` y ATSC (2010) es http://www.atsc.org/cms/standards/a_52-2010.pdf.

[ATT 96] ATT (1996) es `http://www.djvu.att.com/`. URLs adicionales: (2011) *DjVu.org* es `http://djvu.org/` y *DjVuLibre* es http://djvu.sourceforge.net/.

[Baker et al. 99] Baker, Brenda, Udi Manber, y Robert Muth (1999) "Compressing Differences of Executable Code," en *ACM SIGPLAN Workshop on Compiler Support for System Software (WCSSS '99)*.

[Banister y Fischer 99] Banister, Brian, y Thomas R. Fischer (1999) "Quadtree Classification and TCQ Image Coding," en Storer, James A., and Martin Cohn (eds.) (1999) *DCC '99: Data Compression Conference*, Los Alamitos, CA, IEEE Computer Society Press, pp. 149–157.

[Barnsley 88] Barnsley, M. F., y Sloan, A. D. (1988) "A Better Way to Compress Images," *Byte Magazine*, pp. 215-222, January.

Barnsley, M. F. (1988) *Fractals Everywhere*, New York, Academic Press.

[Bass 92] Bass, Thomas A. (1992) Eudaemonic Pie, New York, Penguin Books.

[Bentley 86] Bentley, J. L. et al. (1986) "A Locally Adaptive Data Compression Algorithm," *Communications of the ACM*, **29**(4): 320-330, April.

[Blackstock 87 y 89] Blackstock, Steve (1987) "LZW and GIF Explained," disponible[14] en `http://www.fileformat.info/format/gif/spec/5cbf9be5608145fe83063ce517c9b501/index.htm`. (1989) estándar GIF 89a en http://www.w3.org/Graphics/GIF/spec-gif89a.txt

[Bloom 96] Bloom, Charles R. (1996) "LZP: A New Data Compression Algorithm," en *Proceedings of Data Compression Conference*, J. Storer, editor, Los Alamitos, CA, IEEE Computer Society Press, p. 425.

[Bloom 98] Bloom, Charles R. (1998) "Solving the Problems of Context Modeling," disponible por ftp en `http://www.cbloom.com/papers/ppmz.zip`.

[BOCU 01] BOCU (2001) la vieja URL es `http://oss.software.ibm.com/icu/docs/papers/binary_ordered_compression_for_unicode.html`. El enlace aún disponible en diciembre de 2011 es[15] `http://icu-project.org/docs/papers/binary_ordered_compression_for_unicode.html`.

[BOCU 02] BOCU-1 (2002) es `http://www.unicode.org/notes/tn6/`.

[14] Vieja ULR: `http://www.ece.uiuc.edu/~ece291/class-resources/gpe/gif.txt.html`.

[15] hyperenlace: http://icu-project.org/docs/papers/binary_ordered_compression_for_unicode.html

[Born 95] Born, Günter (1995) *The File Formats Handbook*, London, New York, International Thomson Computer Press.

[Bosi y Goldberg 03] Bosi, Marina, y Richard E. Goldberg (2003) *Introduction To Digital Audio Coding and Standards*, Boston, MA, Kluwer Academic.

[Boussakta y Alshibami 04] Boussakta, Said, y Hamoud O. Alshibami (2004) "Fast Algorithm for the 3-D DCTII," *IEEE Transactions on Signal Processing*, **52**(4).

[Bradley, Brislawn y Hopper 93] Bradley, Jonathan N., Christopher M. Brislawn, y Tom Hopper (1993) "The FBI Wavelet/Scalar Quantization Standard for Grayscale Fingerprint Image Compression," *Proceedings of Visual Information Processing II*, Orlando, FL, SPIE vol. 1961, pp. 293-304, April.

[Brandenburg y Stoll 94] Brandenburg, Karlheinz, y Gerhard Stoll (1994) "ISO-MPEG-1 Audio: A Generic Standard for Coding of High-Quality Digital Audio," *Journal of the Audio Engineering Society*, **42**(10): 780-792, October.

[Brandenburg 99] Brandenburg, Karlheinz (1999) "MP3 and AAC Explained," *The AES 17th International Conference*, Florence, Italy, Sept. 2-5. Disponible en[16] `http://www.cselt.it/mpeg/tutorials.htm`. y —en diciembre de 2011— en[17] `http://www.telos-systems.com/techtalk/hosted/Brandenburg_mp3_aac.pdf`. Véase también la URL `http://www.aes.org/`.

[Brislawn, Bradley y Hopper 96] Brislawn, Christopher, Jonathan Bradley, R. Onyshczak, y Tom Hopper (1996) "The FBI Compression Standard for Digitized Fingerprint Images," en *Proceedings SPIE*, Vol. 2847, Denver, CO, pp. 344-355, August.

[BSDiff 05] BSDiff (2005) es `http://www.daemonology.net/bsdiff/bsdiff-4.3.tar.gz`.

[Burrows et al. 92] Burrows, Michael, et al. (1992) *On-line Data Compression in a Log-Structured File System*, Digital, Systems Research Center, Palo Alto, CA.

[Burrows y Wheeler 94] Burrows, Michael, y D. J. Wheeler (1994) *A Block-Sorting Lossless Data Compression Algorithm*, Digital Systems Research Center Report 124, Palo Alto, CA, May 10.

[Burt y Adelson 83] Burt, Peter J., y Edward H. Adelson (1983) "The Laplacian Pyramid as a Compact Image Code," *IEEE Transactions on Communications*, **COM-31**(4): 532-540, April.

[Buyanovsky 94] Buyanovsky, George (1994) "Associative Coding" (en Ruso), *Monitor*, Moscow, #8, 10-19, August. (El autor de este libro dispone de las copias impresas de la fuente rusa y la traducción al inglés. Envíense las solicitudes a la dirección electrónica del autor del Prefacio.)

[Buyanovsky 02] Buyanovsky, George (2002) Private communications[18] (buyanovsky@home.com).

[Cachin 98] Cachin, Christian (1998) "An Information-Theoretic Model for Steganography," en *Proceedings of the Second International Workshop on Information Hiding*, D. Aucsmith, ed. vol. 1525 de Lecture Notes in Computer Science, Berlin, Springer-Verlag, pp. 306-318.

[Calgary 06] Calgary (2006) es `ftp://ftp.cpsc.ucalgary.ca/pub/projects/` archivos de `text.compression.corpus`.

[16] Enlace roto.
[17] Hiperenlace: http://www.telos-systems.com/techtalk/hosted/Brandenburg_mp3_aac.pdf
[18] Comunicaciones privadas.

[Campos 06] Campos, Arturo San Emeterio (2006) Range coder, en `http://www.arturocampos.com/ac_range.html`.

[Canterbury 06] Canterbury (2006) es `http://corpus.canterbury.ac.nz`.

[Capon 59] Capon, J. (1959) "A Probabilistic Model for Run-length Coding of Pictures," *IEEE Transactions on Information Theory*, **5**(4): 157-163, December.

[Carpentieri et al. 00] Carpentieri, B., M.J. Weinberger, y G. Seroussi (2000) "Lossless Compression of Continuous-Tone Images," *Proceedings of the IEEE*, **88**(11):1797-1809, November.

[Chaitin 77] Chaitin, Gregory J. (1977) "Algorithmic Information Theory," *IBM Journal of Research and Development*, **21**:350–359, July.

[Chaitin 97] Chaitin, Gregory J. (1997) *The Limits of Mathematics*, Singapore, Springer-Verlag.

[Chomsky 56] Chomsky, N. (1956) "Three Models for the Description of Language," *IRE Transactions on Information Theory*, **2**(3): 113-124.

[Cleary y Witten 84] Cleary, John G., y I. H. Witten (1984) "Data Compression Using Adaptive Coding and Partial String Matching," *IEEE Transactions on Communications*, **COM-32**(4):396–402, April.

[Cleary et al. 95] Cleary, John G., W. J. Teahan, e Ian H. Witten (1995) "Unbounded Length Contexts for PPM," *Data Compression Conference, 1995*, 52-61.

[Cleary y Teahan 97] Cleary, John G. y W. J. Teahan (1997) "Unbounded Length Contexts for PPM," *The Computer Journal*, **40**(2/3): 67-75.

[Cole 85] Cole, A. J. (1985) "A Note on Peano Polygons and Gray Codes," *International Journal of Computer Mathematics*, **18**: 3-13.

[Cole 86] Cole, A. J. (1986) "Direct Transformations Between Sets of Integers and Hilbert Polygons," *International Journal of Computer Mathematics*, **20**: 115-122.

[Constantinescu y Storer 94a,b] Constantinescu, C., y J. A. Storer (1994a) "Online Adaptive Vector Quantization with Variable Size Codebook Entries," *Information Processing and Management*, **30**(6): 745-758.

Constantinescu, C., y J. A. Storer (1994b) "Improved Techniques for Single-Pass Adaptive Vector Quantization," *Proceedings of the IEEE*, **82**(6): 933-939, June.

[Constantinescu y Arps 97] Constantinescu, C., y R. Arps (1997) "Fast Residue Coding for Lossless Textual Image Compression," en *Proceedings of the 1997 Data Compression Conference*, J. Storer, ed., Los Alamitos, CA, IEEE Computer Society Press, pp. 397-406.

[Cormack y Horspool 87] Cormack G. V., y R. N. S. Horspool (1987) "Data Compression Using Dynamic Markov Modelling," *The Computer Journal*, **30**(6): 541-550.

[Cormen et al. 01] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest y Clifford Stein (2001) *Introduction to Algorithms*, 2nd Edition, MIT Press and McGraw-Hill.

[corr.pdf 02] corr.pdf (2002) es `http://www.davidsalomon.name/DC2advertis/Corr.pdf`.

[CRC 98] CRC (1998) Soulodre, G. A., T. Grusec, M. Lavoie, y L. Thibault, "Subjective Evaluation of State-of-the-Art 2-Channel Audio Codecs," *Journal of the Audio Engineering Society*, **46**(3): 164-176, March.

[CREW 00] CREW 2000 es `http://www.crc.ricoh.com/CREW/`. Búsquese el año 1995 en la siguiente URL `http://rii.ricoh.com/technical-publications`.

[Crocker 95] Crocker, Lee Daniel (1995) "PNG: The Portable Network Graphic Format," *Dr. Dobb's Journal of Software Tools*, **20**(7): 36-44.

[Culik y Kari 93] Culik, Karel II, y J. Kari (1993) "Image Compression Using Weighted Finite Automata," *Computer and Graphics*, **17**(3): 305-313.

[Culik y Kari 94a,b] Culik, Karel II, y J. Kari (1994a) "Image-Data Compression Using Edge-Optimizing Algorithm for WFA Inference," *Journal of Information Processing and Management*, **30**(6): 829-838.

Culik, Karel II, y Jarkko Kari (1994b) "Inference Algorithm for Weighted Finite Automata and Image Compression," en *Fractal Image Encoding and Compression*, Y. Fisher, editor, New York, NY, Springer-Verlag.

[Culik y Kari 95] Culik, Karel II, y Jarkko Kari (1995) "Finite State Methods for Compression and Manipulation of Images," en *DCC '96, Data Compression Conference*, J. Storer, editor, Los Alamitos, CA, IEEE Computer Society Press, pp. 142-151.

[Culik y Valenta 96] Culik, Karel II, y V. Valenta (1996) "Finite Automata Based Compression of Bi-Level Images," en Storer, James A. (ed.), *DCC '96, Data Compression Conference*, Los Alamitos, CA, IEEE Computer Society Press, pp. 280-289.

[Culik y Valenta 97a,b] Culik, Karel II, y V. Valenta (1997a) "Finite Automata Based Compression of Bi-Level and Simple Color Images," *Computer and Graphics*, **21**: 61-68.

Culik, Karel II, y V. Valenta (1997b) "Compression of Silhouette-like Images Based on WFA," *Journal of Universal Computer Science*, **3**: 1100-1113.

[Dasarathy 95] Dasarathy, Belur V. (ed.) (1995) *Image Data Compression: Block Truncation Coding (BTC) Techniques*, Los Alamitos, CA, IEEE Computer Society Press.

[Daubechies 88] Daubechies, Ingrid (1988) "Orthonormal Bases of Compactly Supported Wavelets," *Communications on Pure and Applied Mathematics*, **41**: 909-996.

[Deflate 03] Deflate (2003) es `http://www.gzip.org/zlib/`.

[della Porta 58] della Porta, Giambattista (1558) *Magia Naturalis*, Naples, first edition, cuatro volúmenes 1558, second edition, 20 volúmenes 1589. Traducido por Thomas Young y Samuel Speed, *Natural Magick by John Baptista Porta, a Neopolitane*, London 1658.

[Demko et al. 85] Demko, S., L. Hodges, y B. Naylor (1985) "Construction of Fractal Objects with Iterated Function Systems," *Computer Graphics*, **19**(3): 271-278, July.

[DeVore et al. 92] DeVore, R., et al. (1992) "Image Compression Through Wavelet Transform Coding," *IEEE Transactions on Information Theory*, **38**(2): 719-746, March.

[Dewitte y Ronson 83] Dewitte, J., y J. Ronson (1983) "Original Block Coding Scheme for Low Bit Rate Image Transmission," en *Signal Processing II: Theories and Applications—Proceedings of EUSIPCO 83*, H. W. Schussler, ed., Amsterdam, Elsevier Science Publishers B. V. (North-Holland), pp. 143-146.

[Dolby 06] Dolby (2006) es `http://www.dolby.com/`.

[donationcoder 06] donationcoder (2006) es `http://www.donationcoder.com/Reviews/Archive/ArchiveTools/index.html`.

[Durbin 60] Durbin J. (1960) "The Fitting of Time-Series Models," *JSTOR: Revue de l'Institut International de Statistique*, **28**: 233-344.

[DVB 06] DVB (2006) es `http://www.dvb.org/`.

[Ekstrand 96] Ekstrand, Nicklas (1996) "Lossless Compression of Gray Images via Context TreeWeighting," en Storer, James A. (ed.), *DCC '96: Data Compression Conference*, Los Alamitos, CA, IEEE Computer Society Press, pp. 132-139, April.

[Elias 75] Elias, P. (1975) "Universal Codeword Sets and Representations of the Integers," *IEEE Transactions on Information Theory*, **IT-21**(2):194-203, March.

[Faller 73] Faller N. (1973) "An Adaptive System for Data Compression," en *Record of the 7th Asilomar Conference on Circuits, Systems, and Computers*, pp. 593-597.

[Fang 66] Fang I. (1966) "It Isn't ETAOIN SHRDLU; It's ETAONI RSHDLC," *Journalism Quarterly*, **43**: 761-762.

[Feder 88] Feder, Jens (1988) Fractals, New York, Plenum Press.

[Federal Bureau of Investigations 93] Federal Bureau of Investigation (1993) *WSQ Grayscale Fingerprint Image Compression Specification, ver. 2.0*, Documento #IAFIS-IC-0110v2, Criminal Justice Information Services, February.

[Feig y Linzer 90] Feig, E., y E. Linzer (1990) "Discrete Cosine Transform Algorithms for Image Data Compression," en *Proceedings Electronic Imaging '90 East*, pp. 84-87, Boston, MA.

[Feldspar 03] Feldspar (2003) es `http://www.zlib.org/feldspar.html`.

[Fenwick 96] Fenwick, P. (1996) *Symbol Ranking Text Compression*, Tech. Rep. 132, Dept. of Computer Science, University of Auckland, New Zealand, June.

[Fenwick 96a] Fenwick, Peter (1996a) "Punctured Elias Codes for variable-length coding of the integers," Technical Report 137, Department of Computer Science, The University of Auckland, December. Éste está también disponible (diciembre de 2011) *online* en la URL `http://www.firstpr.com.au/audiocomp/lossless/TechRep137.pdf`.

[Fiala y Greene 89] Fiala, E. R., y D. H. Greene (1989), "Data Compression with Finite Windows," *Communications of the ACM*, **32**(4): 490-505.

[Fibonacci 99] Fibonacci (1999) es el archivo `Fibonacci.html` en `http://www-groups.dcs.st-and.ac.uk/~history/References/`.

[FIPS197 01] FIPS197 (2001) *Advanced Encryption Standard*, FIPS Publication 197, November 26, 2001. Disponible en `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

[firstpr 06] firstpr (2006) es `http://www.firstpr.com.au/audiocomp/lossless/#rice`.

[Fisher 95] Fisher, Yuval (ed.) (1995) *Fractal Image Compression: Theory and Application*, New York, Springer-Verlag.

[flac.devices 06] flac.devices (2006) es `http://flac.sourceforge.net/links.html#hardware`.

[flacID 06] flacID (2006) es `http://flac.sourceforge.net/id.html`.

[Fox et al.] Fox, E. A., et al. (1991) "Order Preserving Minimal Perfect Hash Functions and Information Retrieval," *ACM Transactions on Information Systems*, **9**(2): 281-308.

[Fraenkel y Kein 85] Fraenkel, A. S., y S. T. Klein (1985) "Novel Compression of Sparse Bit-Strings—Preliminary Report," en A. Apostolico and Z. Galil, eds. *Combinatorial Algorithms on Words*, Vol. 12, NATO ASI Series F: 169-183, New York, Springer-Verlag.

[Frank et al. 80] Frank, Amalie J., J. D. Daniels, y Diane R. Unangst (1980) "Progressive Image Transmission Using a Growth-Geometry Coding," *Proceedings of the IEEE*, **68**(7): 897-909, July.

[Freeman 61] Freeman, H. (1961) "On The Encoding of Arbitrary Geometric Configurations," *IRE Transactions on Electronic Computers*, **EC-10**(2): 260-268, June.

[G131 06] G131 (2006) ITU-T Recommendation G.131, *Talker echo and its control*.

[G.711 72] G.711 (1972) es `http://en.wikipedia.org/wiki/G.711`.

[Gallager y van Voorhis 75] Gallager, Robert G., y David C. van Voorhis (1975) "Optimal Source Codes for Geometrically Distributed Integer Alphabets," *IEEE Transactions on Information Theory*, **IT-21**(3): 228-230, March.

[Gallager 78] Gallager, Robert G. (1978) "Variations On a Theme By Huffman," *IEEE Transactions on Information Theory*, **IT-24**(6): 668-674, November.

[Gardner 72] Gardner, Martin (1972) "Mathematical Games," *Scientific American*, **227**(2): 106, August.

[Gersho y Robert 92] Gersho, Allen, y Robert M. Gray (1992) *Vector Quantization and Signal Compression*, Boston, MA, Kluwer Academic Publishers.

[Gharavi 87] Gharavi, H. (1987) "Conditional Run-Length and Variable-Length Coding of Digital Pictures," *IEEE Transactions on Communications*, **COM-35**(6): 671-677, June.

[Gilbert y Moore 59] Gilbert, E. N., y E. F. Moore (1959) "Variable Length Binary Encodings," *Bell System Technical Journal, Monograph 3515*, **38**:933-967, July.

[Gilbert y Brodersen 98] Gilbert, Jeffrey M., y Robert W. Brodersen (1998) "A Lossless 2-D Image Compression Technique for Synthetic Discrete-Tone Images," en *Proceedings of the 1998 Data Compression Conference*, J. Storer, ed., Los Alamitos, CA, IEEE Computer Society Press, pp. 359-368, March. Éste está tambien disponible[19] en la URL `http://bwrc.eecs.berkeley.edu/Publications/`, año 1999.

[Givens 58] Givens, Wallace (1958) "Computation of Plane Unitary Rotations Transforming a General Matrix to Triangular Form," *Journal of the Society for Industrial and Applied Mathematics*, **6**(1): 26-50, March.

[19] Archivo: `A_lossless_2-D_image_compression_technique/JMG_DCC98.pdf`.

[Golomb 66] Golomb, Solomon W. (1966) "Run-Length Encodings," *IEEE Transactions on Information Theory*, **IT-12**(3): 399-401.

[Gonzalez y Woods 92] Gonzalez, Rafael C., y Richard E. Woods (1992) *Digital Image Processing*, Reading, MA, Addison-Wesley.

[Gottlieb et al. 75] Gottlieb, D., et al. (1975) *A Classification of Compression Methods and their Usefulness for a Large Data Processing Center*, Proceedings of National Computer Conference, **44**: 453-458.

[Gray 53] Gray, Frank (1953) "Pulse Code Communication," United States Patent 2,632,058, March 17.

[H.264Draft 06] H.264Draft (2006) es `ftp://standards.polycom.com/JVT_Site/draft_standard/`, archivo `JVT-G050r1.zip`. Éste es el borrador del estándar H.264.

[H264PaperIR 06 y 10] H.264PaperIR (2006) es[20] `http://www.vcodex.com/h264_transform.pdf` y (2010) es `http://www.vcodex.com/h264.html`, archivo `H264_4x4_transform_whitepaper_Nov10.pdf` (un documento de Iain Richardson).

[H.264PaperRM 06] H.264PaperRM (2006) es[21] `http://research.microsoft.com/apps/pubs/default.aspx?id=77882`, archivo `MalvarCSVTJuly03.pdf`. (un documento de Rico Malvar).

[H.264Standards 06] H.264Standards (2006) es `ftp://standards.polycom.com/JVT_Site/` (el repositorio de los estándares de H.264).

[H.264Transform 06] H.264Transform (2006) `ftp://standards.polycom.com/JVT_Site/2002_01_Geneva/` o en `http://wftp3.itu.int/av-arch/jvt-site/2002_01_Geneva/`, archivos `JVT-B038r2.doc` y `JVT-B039r2.doc` (la transformada entera de H.264).

[h2g2 06] h2g2 (2006) es `http://www.bbc.co.uk/dna/h2g2/A406973`.

[Haffner et al. 98] Haffner, Patrick, et al. (1998) "High-Quality Document Image Compression with DjVu," *Journal of Electronic Imaging*, **7**(3): 410-425, SPIE. Éste está también disponible[22] desde (dic. 2011) `http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.38.4518`.

[Hafner 95] Hafner, Ullrich (1995) "Asymmetric Coding in (m)-WFA Image Compression," Report 132, Department of Computer Science, University of Würzburg, December.

[Hans y Schafer 01] Hans, Mat y R. W. Schafer (2001) "Lossless Compression of Digital Audio," *IEEE Signal Processing Magazine*, **18**(4): 21-32, July.

[Havas et al. 93] Havas, G., et al. (1993) *Graphs, Hypergraphs and Hashing, in Proceedings of the International Workshop on Graph-Theoretic Concepts in Computer Science (WG'93)*, Berlin, Springer-Verlag.

[Heath 72] Heath, F. G. (1972) "Origins of the Binary Code," *Scientific American*, **227**(2): 76, August.

[Hilbert 91] Hilbert, D. (1891) "Ueber stetige Abbildung einer Linie auf ein Flächenstück," *Math. Annalen*, **38**:459-460.

[Hirschberg y Lelewer 90] Hirschberg, D., y D. Lelewer (1990) "Efficient Decoding of Prefix Codes," *Communications of the ACM*, **33**(4): 449-459.

[20] Enlace roto.
[21] El enlace roto estaba en la URL `http://research.microsoft.com/~malvar/papers/`.
[22] La vieja URL `http://citeseer.nj.nec.com/bottou98high.html` no es válida.

[Horspool 91] Horspool, N. R. (1991) "Improving LZW," en *Proceedings of the 1991 Data Compression Conference*, J. Storer, ed., Los Alamitos, CA, IEEE Computer Society Press, pp .332-341.

[Horspool y Cormack 92] Horspool, N. R., y G. V. Cormack (1992) "Constructing Word-Based Text Compression Algorithms," en *Proceedings of the 1992 Data Compression Conference*, J. Storer, ed., Los Alamitos, CA, IEEE Computer Society Press, PP. 62-71, April.

[Horstmann 06] Horstmann (2006) `http://www.horstmann.com/bigj/help/windows/tutorial.html`.

[Howard y Vitter 92a] Howard, Paul G., y J. S. Vitter (1992a) "New Methods for Lossless Image Compression Using Arithmetic Coding," *Information Processing and Management*, **28**(6):765-779. Disponible desde la URL `http://www.ittc.ku.edu/~jsv/Papers/catalog/3DATA_COMPRESSION.html`, archivo `Hov91.lossless_image_ac.pdf`.

[Howard y Vitter 92b] Howard, Paul G., y J. S. Vitter (1992b) "Error Modeling for Hierarchical Lossless Image Compression," en *Proceedings of the 1992 Data Compression Conference*, J. Storer, ed., Los Alamitos, CA, IEEE Computer Society Press, pp. 269-278. Disponible desde la URL `http://www.ittc.ku.edu/~jsv/Papers/catalog/3DATA_COMPRESSION.html`, archivo `HoV95.vifull.pdf`.

[Howard y Vitter 92c] Howard, Paul G., y J. S. Vitter (1992c) "Practical Implementations of Arithmetic Coding," en *Image and Text Compression*, J. A. Storer, ed., Norwell, MA, Kluwer Academic Publishers, PP. 85-112. Disponible[23] desde la URL `http://www.ittc.ku.edu/~jsv/Papers/catalog/3DATA_COMPRESSION.html`, archivo `HoV92.actech.pdf`.

[Howard y Vitter 93] Howard, Paul G., y J. S. Vitter, (1993) "Fast and Efficient Lossless Image Compression," en *Proceedings of the 1993 Data Compression Conference*, J. Storer, ed., Los Alamitos, CA, IEEE Computer Society Press, pp. 351-360. Disponible desde la URL `http://www.ittc.ku.edu/~jsv/Papers/catalog/3DATA_COMPRESSION.html`, archivo `HoV93.fitech.pdf`.

[Howard y Vitter 94a] Howard, Paul G., y J. S. Vitter (1994a) "Fast Progressive Lossless Image Compression," Proceedings of the Image and Video Compression Conference, *IS&T/SPIE 1994 Symposium on Electronic Imaging: Science & Technology*, 2186, San Jose, CA, pp. 98-109, February. Disponible desde la URL `http://www.ittc.ku.edu/~jsv/Papers/catalog/3DATA_COMPRESSION.html`, archivo `HoV94.progressive_FELICS.pdf`.

[Howard y Vitter 94b] Howard, Paul G., y J. S. Vitter (1994b) "Design and Analysis of Fast text Compression Based on Quasi-Arithmetic Coding," *Journal of Information Processing and Management*, **30**(6): 777-790. Disponible[24] desde la URL `http://www.ittc.ku.edu/~jsv/Papers/catalog/3DATA_COMPRESSION.html`, archivo `HoV93.qtfull.pdf`.

[Huffman 52] Huffman, David (1952) "A Method for the Construction of Minimum Redundancy Codes," *Proceedings of the IRE*, **40**(9):1098-1101.

[Hunt 76] Hunt, James W. y M. Douglas McIlroy (1976) "An Algorithm for Differential File Comparison," Computing Science Technical Report No. 41, Murray Hill, NJ, Bell Labs, June.

[Hunter y Robinson 80] Hunter, R., y A. H. Robinson (1980) "International Digital Facsimile Coding Standards," *Proceedings of the IEEE*, **68**(7): 854-867, July.

[hydrogenaudio 06] hydrogenaudio (2006) es `http://www.hydrogenaudio.org/forums/`.

[23] Vieja URL (`http://www.cs.duke.edu/~jsv/Papers/catalog/node66.html`) inválida.
[24] Vieja URL (`http://www.cs.duke.edu/~jsv/Papers/catalog/node70.html`) inválida.

[IA-32 06] IA-32 (2006) es `http://en.wikipedia.org/wiki/IA-32`.

[IBM 88] IBM (1988) *IBM Journal of Research and Development, #6* (la edición íntegra).

[IEEE754 85] IEEE754 (1985) ANSI/IEEE Standard 754-1985, "IEEE Standard for Binary Floating-Point Arithmetic."

[IMA 06] IMA (2006) es `http://www.ima.org/`.

[ISO 84] ISO (1984) "Information Processing Systems-Data Communication High-Level Data Link Control Procedure-Frame Structure," IS 3309, 3rd ed., October.

[ISO 03] ISO (2003) es `http://www.iso.ch/`.

[ISO/IEC 93] ISO/IEC (1993) International Standard IS 11172-3 "Information Technology, Coding of Moving Pictures and Associated Audio for Digital Storage Media at up to about 1.5 Mbits/s—Part 3: Audio."

[ISO/IEC 00] ISO/IEC (2000), International Standard IS 15444-1 "Information Technology—JPEG 2000 Image Coding System." Éste es el FDC[25] (Final Committee Draft) version 1.0, 16 March 2000.

[ISO/IEC 03] ISO/IEC (2003) International Standard ISO/IEC 13818-7, "Information technology, Generic coding of moving pictures and associated audio information, Part 7: Advanced Audio Coding (AAC)," 2nd ed., 2003-08-01.

[ITU-R/BS1116 97] ITU-R/BS1116 (1997) ITU-R, documento BS 1116 "Methods for the Subjective Assessment of Small Impairments in Audio Systems Including Multichannel Sound Systems," Rev. 1, Geneva. Disponible también en español en la URL `http://www.itu.int/rec/R-REC-BS.1116-1-199710-I/es`.

[ITU-T 89] ITU-T (1989) CCITT Recommendation G.711: "Pulse Code Modulation (PCM) of Voice Frequencies." Disponible también en español en la URL `http://www.itu.int/rec/T-REC-G.711/es`.

[ITU-T 90] ITU-T (1990), Recommendation G.726 (12/90), *40, 32, 24, 16 kbit/s Adaptive Differential Pulse Code Modulation (ADPCM)*. Disponible también en español en la URL `http://www.itu.int/rec/T-REC-G.726/es`.

[ITU-T 94] (1994) ITU-T Recommendation V.42, Revision 1 "Error-correcting Procedures for DCEs Using Asynchronous-to-Synchronous Conversion." Disponible también en español en la URL `http://www.itu.int/rec/T-REC-V.42/es`.

[ITU-T264 02] ITU-T264 (2002) ITU-T Recommendation H.264, ISO/IEC 11496-10, "Advanced Video Coding," El último borrador del comité (FCD), Documento JVT-E022, September. Disponible también en español en la URL `http://www.itu.int/rec/T-REC-H.264/es`.

[ITU/TG10 91] ITU/TG10 (1991) ITU-R documento TG-10-2/3-E "Basic Audio Quality Requirements for Digital Audio Bit-Rate Reduction Systems for Broadcast Emission and Primary Distribution," 28 October[26].

[Jayant 97] Jayant N. (ed.) (1997) *Signal Compression: Coding of Speech, Audio, Text, Image and Video*, Singapore, World Scientific Publications.

[JBIG] JBIG (2003) es[27] `http://www.jpeg.org/jbig/index.html`.

[25] El último borrador del comité.
[26] No he podido localizar este documento en internet.
[27] Vieja URL (`http://www.jpeg.org/jbighomepage.html`) inválida.

[JBIG2 03] JBIG2 (2003) es[28] es `http://www.jpeg.org/jbig/jbigpt2.html`.

[JBIG2 06] JBIG2 (2006) es `http://jbig2.com/`.

[Jordan y Barrett 74] Jordan, B. W., y R. C. Barrett (1974) "A Cell Organized Raster Display for Line Drawings," *Communications of the ACM*, **17**(2): 70-77.

[Joshi, Crump, y Fisher 93] Joshi, R. L., V. J. Crump, y T. R. Fischer (1993) "Image Subband Coding Using Arithmetic and Trellis Coded Quantization," *IEEE Transactions on Circuits and Systems Video Technology*, **5**(6): 515-523, December.

[JPEG 00] JPEG 2000 Organization (2000) es `http://www.jpeg.org/JPEG2000.htm`.

[Kendall 61] Kendall, Maurice G. (1961) *A Course in the Geometry of n-Dimensions*, New York, Hafner.

[Kieffer et al. 96a,b 11] Kieffer, J., G. Nelson, y E-H. Yang (1996a) "Tutorial on the quadrisection method and related methods for lossless data compression." Disponible[29] en la URL `http://www.ece.umn.edu/users/kieffer/index.html`.

Kieffer, J., E-H. Yang, G. Nelson, y P. Cosman (1996b) "Lossless compression via bisection trees,". Disponible en la URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.49.1493`.

Kieffer, J., Flajolet P., y E-h Yang (2011) "Universal Lossless Data Compression Via Binary Decision Diagrams". Disponible en la URL `http://arxiv.org/abs/1111.1432`.

[Kleijn y Paliwal 95] Kleijn, W. B., y K. K. Paliwal (1995) *Speech Coding and Synthesis*, Elsevier, Amsterdam.

[Knowlton 80] Knowlton, Kenneth (1980) "Progressive Transmission of Grey-Scale and Binary Pictures by Simple, Efficient, and Lossless Encoding Schemes," *Proceedings of the IEEE*, **68**(7): 885-896, July.

[Knuth 73] Knuth, Donald E. (1973) *The Art of Computer Programming*, Vol. 1, 2nd Ed., Reading, MA, Addison-Wesley.

[Knuth 81] Knuth, Donald E. (1981) *The Art of Computer Programming*, Vol. 2, 2nd Ed., Reading, MA, Addison-Wesley.

[Knuth 85] Knuth, Donald E. (1985) "Dynamic Huffman Coding," *Journal of Algorithms*, **6**: 163-180.

[Korn et al. 02] Korn D., et al. (2002) "The VCDIFF Generic Differencing and Compression Data Format," RFC 3284, June 2002, disponible en internet como el archivo de texto "rfc3284.txt"; por ejemplo, en la URL `http://www.ietf.org/rfc/rfc3284.txt`.

[Krichevsky 81] Krichevsky, R. E., y V. K. Trofimov (1981) "The Performance of Universal Coding," *IEEE Transactions on Information Theory*, **IT-27**: 199-207, March.

[Lambert 99] Lambert, Sean M. (1999) "Implementing Associative Coder of Buyanovsky (ACB) Data Compression," M.S. thesis, Bozeman, MT, Montana State University (disponible bajo petición a Sean Lambert en sum1els@mindless.com).

[28] Vieja URL (`http://www.jpeg.org/jbigpt2.html`) inválida.
[29] La URL existe, pero no se puede bajar ningún documento.

[Langdon y Rissanen 81] Langdon, Glen G., y J. Rissanen (1981) "Compression of Black White Images with Arithmetic Coding," *IEEE Transactions on Communications*, **COM-29**(6): 858-867, June.

[Langdon 83] Langdon, Glen G. (1983) "A Note on the Ziv-Lempel Model for Compressing Individual Sequences," *IEEE Transactions on Information Theory*, **IT-29**(2): 284-287, March.

[Langdon 83a] Langdon, Glenn G. (1983a) "An Adaptive Run Length Coding Algorithm," *IBM Technical Disclosure Bulletin*, **26**(7B): 3783-3785, December.

[Langdon 84] Langdon, Glen G. (1984) *On Parsing vs. Mixed-Order Model Structures for Data Compression*, informe de investigación de IBM RJ-4163 (46091), January 18, 1984, San Jose, CA.

[Levinson 47] Levinson, N. (1947) "The Weiner RMS Error Criterion in Filter Design and Prediction," *Journal of Mathematical Physics*, **25**: 261-278.

[Lewalle 95] Lewalle, Jacques (1995) "Tutorial on Continuous Wavelet Analysis of Experimental Data", disponible en `http://www.ecs.syr.edu/faculty/lewalle/tutor/tutor.html`.

[Li y Furht 03] Li, Xiuqi y Borko Furht (2003) "An Approach to Image Compression Using Three-Dimensional DCT," *Proceeding of The Sixth International Conference on Visual Information System 2003 (VIS2003)*, September 24-26.

[Liebchen et al. 05] Liebchen, Tilman et al. (2005) "The MPEG-4 Audio Lossless Coding (ALS) Standard - Technology and Applications," AES 119th Convention, New York, October 7–10, 2005. Disponible[30] en la URL `http://www.nue.tu-berlin.de/menue/publikationen/`, archivo `0737Liebchen2005.pdf`.

[Liefke y Suciu 99] Liefke, Hartmut y Dan Suciu (1999) "XMill: an Efficient Compressor for XML Data," *Proceedings of the ACM SIGMOD Symposium on the Management of Data, 2000*, pp. 153-164. Disponible [31] en `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.33.2632`.

[Linde, Buzo, y Gray 80] Linde, Y., A. Buzo, y R. M. Gray (1980) "An Algorithm for Vector Quantization Design," *IEEE Transactions on Communications*, **COM-28**: 84-95, January.

[Liou 91] Liou, Ming (1991) "Overview of the p×64 kbits/s Video Coding Standard," *Communications of the ACM*, **34**(4): 59-63, April.

[Litow y Olivier 95] Litow, Bruce, y Olivier de Val (1995) "The Weighted Finite Automaton Inference Problem," Technical Report 95-1, James Cook University, Queensland.

[Loeffler et al. 89] Loeffler, C., A. Ligtenberg, y G. Moschytz (1989) "Practical Fast 1-D DCT Algorithms with 11 Multiplications," en *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP '89)*, pp. 988-991.

[Mallat 89] Mallat, Stephane (1989) "A Theory for Multiresolution Signal Decomposition: The Wavelet Representation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *11*(7): 674-693, July.

[Manber y Myers 93] Manber, U., y E. W. Myers (1993) "Suffix Arrays: A New Method for On-Line String Searches," *SIAM Journal on Computing*, **22**(5): 935-948, October.

[30] Vieja URL(`http://www.nue.tu-berlin.de/forschung/projekte/lossless/mp4als.html`) inválida.
[31] Vieja URL(`http://citeseer.nj.nec.com/liefke99xmill.html`) inválida.

[Mandelbrot 82] Mandelbrot, B. (1982) *The Fractal Geometry of Nature*, San Francisco, CA, W. H. Freeman.

[Manning 98] Manning (1998), es el archivo `compression/adv08.html` en la URL `http://www.newmediarepublic.com/dvideo/`.

[Marking 90] Marking, Michael P. (1990) "Decoding Group 3 Images," *The C Users Journal* pp. 45-54, June.

[Matlab 99] Matlab (1999) es `http://www.mathworks.com/`.

[McConnell 92] McConnell, Kenneth R. (1992) FAX: *Digital Facsimile Technology and Applications*, Norwood, MA, Artech House.

[McCreight 76] McCreight, E. M (1976) "A Space Economical Suffix Tree Construction Algorithm," *Journal of the ACM*, **32**(2): 262-272, April.

[Meridian 03] Meridian (2003) es `http://www.meridian-audio.com/`.

[Meyer et al 98] Meyer, F. G., A. Averbuch, y J.O. Strömberg (1998) "Fast Adaptive Wavelet Packet Image Compression," *IEEE Transactions on Image Processing*, **9**(5) May 2000.

[Miano 99] Miano, John (1999) *Compressed Image File Formats*, New York, ACM Press and Addison-Wesley.

[Miller y Wegman 85] Miller, V. S., y M. N.Wegman (1985) "Variations On a Theme by Ziv and Lempel," en A. Apostolico y Z. Galil, eds., NATO ASI series Vol. F12, *Combinatorial Algorithms on Words*, Berlin, Springer, pp. 131-140.

[Mitchell et al. 97] Mitchell, Joan L., W. B. Pennebaker, C. E. Fogg, y D. J. LeGall, eds. (1997) *MPEG Video Compression Standard*, New York, Chapman and Hall and International Thomson Publishing.

[MNG 03] MNG (2003) es `http://www.libpng.org/pub/mng/spec/`.

[Moffat 90] Moffat, Alistair (1990) "Implementing the PPM Data Compression Scheme," *IEEE Transactions on Communications*, **COM-38**(11): 1917-1921, November.

[Moffat 91] Moffat, Alistair (1991) "Two-Level Context Based Compression of Binary Images," en *Proceedings of the 1991 Data Compression Conference*, J. Storer, ed., Los Alamitos, CA, IEEE Computer Society Press, pp. 382-391.

[Moffat et al. 98] Moffat, Alistair, Radford Neal, e Ian H. Witten (1998) "Arithmetic Coding Revisited," *ACM Transactions on Information Systems*, **16**(3): 256-294, July.

[monkeyaudio 06] monkeyaudio (2006) es `http://www.monkeysaudio.com/index.html`.

[Motta et al. 06] Motta, G., F. Rizzo, y J. A. Storer, eds. (2006) *Hyperspectral Data Compression*, New York, Springer Verlag.

[Motte 98] Motte, Warren F. (1998) *Oulipo, A Primer of Potential Literature*, Normal, Ill, Daleky Archive Press.

[MPEG 98] (1998), es `http://www.mpeg.org/`.

[MPEG 00] (1998), es `http://www.mpeg.org/`.

[mpeg-4.als 06] mpeg-4.als (2006) es[32] (versión en inglés) `http://www.nue.tu-berlin.de/menue/forschung/projekte/beendete_projekte/mpeg-4_audio_lossless_coding_als/parameter/en/`

[MPThree 06] (2006) es `http://inventors.about.com/od/mstartinventions/a/`, archivo `MPThree.htm`.

[Mulcahy 96] Mulcahy, Colm (1996) "Plotting and Scheming with Wavelets," *Mathematics Magazine*, **69**(5): 323-343, December. véase también `http://www.spelman.edu/~colm/csam.ps`.

[Mulcahy 97] Mulcahy, Colm (1997) "Image Compression Using the Haar Wavelet Transform," *Spelman College Science and Mathematics Journal*, **1**(1): 22-31, April. También disponible en la URL `http://www.spelman.edu/~colm/wav.ps`. (Se ha afirmado que cualquier persona inteligente de 15 años de edad podría seguir esta introducción a las wavelets.)

[Murray y vanRyper 94] Murray, James D., y William (1994) *Encyclopedia of Graphics File Formats*, Sebastopol, CA, O'Reilly y Assoc.

[Myers 86] Myers, Eugene W. (1986) "An O(ND) Difference Algorithm and its Variations," *Algorithmica*, **1**(2): 251-266.

[Netravali y Limb 80] Netravali, A. y J. O. Limb (1980) "Picture Coding: A Preview," *Proceedings of the IEEE*, **68**: 366-406.

[Nevill-Manning 96] Nevill-Manning, C. G. (1996) "Inferring Sequential Structure," Ph.D. thesis, Department of Computer Science, University of Waikato, New Zealand.

[Nevill-Manning y Witten 97] Nevill-Manning, C. G., e Ian H. Witten (1997) "Compression and Explanation Using Hierarchical Grammars," *The Computer Journal*, **40**(2/3): 104-116.

[NHK 06] NHK (2006) es `http://www.nhk.or.jp/english/`.

[Nix 81] Nix, R. (1981) "Experience With a Space Efficient Way to Store a Dictionary," *Communications of the ACM*, **24**(5): 297-298.

[ntfs 06] ntfs (2006) es `http://www.ntfs.com/`.

[Nyquist 28] Nyquist, Harry (1928) "Certain Topics in Telegraph Transmission Theory," *AIEE Transactions*, **47**: 617-644.

[Ogg squish 06] Ogg squish (2006) es el punto `FLAC` de la URL[33] `http://www.xiph.org/downloads/`.

[Okumura 98] Okumura, Haruhiko (1998) es `http://oku.edu.mie-u.ac.jp/~okumura/compression/` archivos `compression.html e history.html`.

[Osterberg 35] Osterberg, G. (1935) "Topography of the Layer of Rods and Cones in the Human Retina," *Acta Ophthalmologica*, (suppl. 6): 1-103.

[Paeth 91] Paeth, Alan W. (1991) "Image File Compression Made Easy," en *Graphics Gems II*, James Arvo, editor, San Diego, CA, Academic Press.

[Parsons 87] Parsons, Thomas W. (1987) *Voice and Speech Processing*, New York, McGraw-Hill.

[32] Vieja URL (`http://www.nue.tu-berlin.de/forschung/projekte/lossless/mp4als.html`) inválida.
[33] Vieja URL (`http://www.xiph.org/ogg/flac.html`) no válida

[Pan 95] Pan, Davis Yen (1995) "A Tutorial on MPEG/Audio Compression," *IEEE Multimedia*, **2**: 60-74, Summer.

[Pasco 76] Pasco, R. (1976) "Source Coding Algorithms for Fast Data Compression," Ph.D. dissertation, Dept. of Electrical Engineering, Stanford University, Stanford, CA.

[patents 06] patents (2006) es `www.ross.net/compression/patents.html`.

[PDF 01] PDF (2001) *Adobe Portable Document Format Version 1.4*, 3rd ed., Reading, MA, Addison-Wesley, December.

[Peano 90] Peano, G. (1890) "Sur Une Courbe Qui Remplit Toute Une Aire Plaine," *Math. Annalen*, **36**: 157-160.

[Peitgen et al. 82] Peitgen, H. -O., et al. (eds.) (1982) The Beauty of Fractals, Berlin, Springer-Verlag.

[Peitgen y Saupe 85] Peitgen, H. -O., y Dietmar Saupe (1985) *The Science of Fractal Images*, Berlin, Springer-Verlag.

[Pennebaker y Mitchell 88a] Pennebaker, William B., y Joan L. Mitchell (1988a) "Probability Estimation for the Q-coder," *IBM Journal of Research and Development*, **32**(6): 717-726.

[Pennebaker y Mitchell 88b] Pennebaker, William B., Joan L. Mitchell, et al. (1988b) "An Overview of the Basic Principles of the Q-coder Adaptive Binary Arithmetic Coder," *IBM Journal of Research and Development*, **32**(6): 737-752.

[Pennebaker y Mitchell 92] Pennebaker, William B., y Joan L. Mitchell (1992) *JPEG Still Image Data Compression Standard*, New York, Van Nostrand Reinhold.

[Percival 03a] Percival, Colin (2003a) "An Automated Binary Security Update System For FreeBSD," *Proceedings of BSDCon '03*, PP. 29–34.

[Percival 03b] Percival, Colin (2003b) "Naive Differences of Executable Code," Computing Lab, Oxford University. Disponible en[34] la URL `http://www.daemonology.net/papers/bsdiff.pdf`.

[Percival 06] Percival, Colin (2006) "Matching with Mismatches and Assorted Applications," Ph.D. Thesis (trámites pendientes). Disponible en la URL `http://www.daemonology.net/papers/thesis.pdf`.

[Pereira y Ebrahimi 02] Pereira, Fernando y Touradj Ebrahimi (2002) The MPEG-4 Book, Upper Saddle River, NJ, Prentice-Hall.

[Phillips 92] Phillips, Dwayne (1992) "LZW Data Compression," *The Computer Application Journal* Circuit Cellar Inc., **27**:36-48, June/July.

[PKWare 03] PKWare (2003) es `http://www.pkware.com`.

[PNG 03] PNG (2003) es `http://www.libpng.org/pub/png/`.

[Pohlmann 85] Pohlmann, Ken (1985) *Principles of Digital Audio*, Indianapolis, IN, Howard Sams & Co.

[polyvalens 06] polyvalens (2006) es `http://perso.orange.fr/polyvalens/clemens/wavelets/`, enlace `wavelets`.

[34] Vieja URL (`http://www.daemonology.net/bsdiff/bsdiff.pdf`) inválida.

[Press et al. 88] Press, W. H., B. P. Flannery, et al. (1988) *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge, UK, Cambridge University Press. (También disponible en la URL `http://www.nr.com/`.)

[Prusinkiewicz y Lindenmayer 90] Prusinkiewicz, P., y A. (1990) *The Algorithmic Beauty of Plants*, New York, Springer-Verlag.

[Prusinkiewicz et al. 91] Prusinkiewicz, P., A. Lindenmayer, y F. D. Fracchia (1991) "Synthesis of Space-Filling Curves on the Square Grid," en *Fractals in the Fundamental and Applied Sciences*, Peitgen, H.-O., et al. (eds.), Amsterdam, Elsevier Science Publishers, pp. 341-366.

[quicktimeAAC 06] quicktimeAAC (2006) es[35] `http://www.apple.com/quicktime/technologies/aac/`.

[Rabbani y Jones 91] Rabbani, Majid, y Paul W. Jones (1991) *Digital Image Compression Techniques*, Bellingham, WA, Spie Optical Engineering Press.

[Rabiner y Schafer 78] Rabiner, Lawrence R. y Ronald W. Schafer (1978) *Digital Processing of Speech Signals*, Englewood Cliffs, NJ, Prentice-Hall Series in Signal Processing.

[Ramabadran y Gaitonde 88] Ramabadran, Tenkasi V., y Sunil S. Gaitonde (1988) "A Tutorial on CRC Computations," *IEEE Micro*, pp. 62-75, August.

[Ramstad et al. 95] Ramstad, T. A., et al. (1995) *Subband Compression of Images: Principles and Examples*, Amsterdam, Elsevier Science Publishers.

[Rao y Hwang 96] Rao, K. R., y J. J. Hwang (1996) *Techniques and Standards for Image, Video, and Audio Coding*, Upper Saddle River, NJ, Prentice Hall.

[Rao y Yip 90] Rao, K. R., y P. Yip (1990) *Discrete Cosine Transform—Algorithms, Advantages, Applications*, London, Academic Press.

[Rao y Bopardikar 98] Rao, Raghuveer M., y Ajit S. (1998) *Wavelet Transforms: Introduction to Theory and Applications*, Reading, MA, Addison-Wesley.

[rarlab 06] rarlab (2006) es `http://www.rarlab.com/`. La página principal de WinRAR (en castellano) es `http://www.winrar.es/`.

[Reghbati 81] Reghbati, H. K. (1981) "An Overview of Data Compression Techniques," *IEEE Computer*, **14**(4): 71-76.

[Reznik 04] Reznik, Yuriy (2004) "Coding Of Prediction Residual In MPEG-4 Standard For Lossless Audio Coding (MPEG-4 ALS)," Disponible[36] en la URL `http://www.reznik.org/papers/ICASSP04_prediction_residual.pdf`.

[RFC1945 96] RFC1945 (1996) *Hypertext Transfer Protocol—HTTP/1.0*, disponible en la URL `http://www.faqs.org/rfcs/rfc1945.html`.

[RFC1950 96] RFC1950 (1996) ZLIB *Compressed Data Format Specification version 3.3*, es `http://www.ietf.org/rfc/rfc1950`.

[RFC1951 96] RFC1951 (1996) DEFLATE *Compressed Data Format Specification version 1.3*, es `http://www.ietf.org/rfc/rfc1951`.

[35] El enlace no es válido. La URL de QuickTime es `http://www.apple.com/quicktime/`.
[36] Vieja URL(`http://viola.usc.edu/paper/ICASSP2004/HTML/SESSIDX.HTM`) inváida.

[RFC1952 96] RFC1952 (1996) *GZIP File Format Specification Version 4.3*. Disponible en formato PDF en la URL `http://www.ietf.org/rfc/rfc1952.txt.pdf` (véase también `http://www.gzip.org/zlib/rfc-gzip.html`).

[RFC1962 96] RFC1962 (1996) *The PPP Compression Control Protocol (CCP)*, disponible desde la URL `http://msdn.microsoft.com/en-us/library/aa505937.aspx` y en otras muchas fuentes.

[RFC1979 96] RFC1979 (1996) PPP *Deflate Protocol*, es `http://www.faqs.org/rfcs/rfc1979.html`.

[RFC2616 99] RFC2616 (1999) *Hypertext Transfer Protocol—HTTP/1.1*. Disponible en la URL `http://www.faqs.org/rfcs/rfc2616.html` (y en formato PDF en `http://ftp.ics.uci.edu/pub/ietf/http/rfc2616.pdf`).

[Rice 79] Rice, Robert F. (1979) "Some Practical Universal Noiseless Coding Techniques," Jet Propulsion Laboratory, JPL Publication 79-22, Pasadena, CA, March. Disponible como PDF en `http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19790014634_1979014634.pdf`.

[Rice 91] Rice, Robert F. (1991) "Some Practical Universal Noiseless Coding Techniques—Part III. Module PSI14.K," Jet Propulsion Laboratory, JPL Publication 91-3, Pasadena, CA, November. Disponible comp PDF en `http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19920005393_1992005393.pdf`.

[Richardson 03] Richardson, Iain G. (2003) *H.264 and MPEG-4 Video Compression Video Coding for Next-generation Multimedia*, Chichester, West Sussex, UK, John Wiley & Sons. Disponible en formato PDF en la URL `http://mirror05.x264.nl/Dark/videobook.pdf`.

[Rissanen 76] Rissanen, J. J. (1976) "Generalized Kraft Inequality and Arithmetic Coding," *IBM Journal of Research and Development*, **20**: 198-203, May.

[Robinson 97] Robinson, John A. (1997) "Efficient General-Purpose Image Compression with Binary Tree Predictive Coding," *IEEE Transactions on Image Processing*, **6**(4): 601-607 April.

[Robinson y Singer 81] Robinson, P. y D. Singer (1981) "Another Spelling Correction Program," *Communications of the ACM*, **24**(5): 296-297.

[Robinson 94] Robinson, Tony (1994) "Simple Lossless and Near-Lossless Waveform Compression," Technical Report CUED/F-INFENG/TR.156, Cambridge University, December. Disponible en[37] la URL`http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.114.856`.

[Rodriguez 95] Rodriguez, Karen (1995) "Graphics File Format Patent Unisys Seeks Royalties from GIF Developers," *InfoWorld*, January 9, **17**(2): 3.

[Roetling 76] Roetling, P. G. (1976) "Halftone Method with Edge Enhancement and Moiré Suppression," *Journal of the Optical Society of America*, **66**: 985-989.

[Roetling 77] Roetling, P. G. (1977) "Binary Approximation of Continuous Tone Images," *Photography Science and Engineering*, **21**: 60-65.

[Roger y Cavenor 96] Roger, R. E., y M. C. Cavenor (1996) "Lossless Compression of AVIRIS Images," *IEEE Transactions on Image Processing*, **5**(5): 713-719, May.

[37] Vieja URL(`http://citeseer.nj.nec.com/robinson94shorten.html`) inválida.

[Ronson y Dewitte 82] Ronson, J. y J. (1982) "Adaptive Block Truncation Coding Scheme Using an Edge Following Algorithm," en *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, Piscataway, NJ, IEEE Press, pp. 1235-1238.

[Rossignac 98] Rossignac, J. (1998) "Edgebreaker: Connectivity Compression for Triangle Meshes," GVU Technical Report GIT-GVU-98-35, Atlanta, GA, Georgia Institute of Technology.

[Rubin 79] Rubin, F. (1979) "Arithmetic Stream Coding Using Fixed Precision Registers," *IEEE Transactions on Information Theory*, **25**(6): 672-675, November.

[Sacco et al. 88] Sacco, William, et al. (1988) *Information Theory, Saving Bits*, Providence, RI, Janson Publications.

[Sagan 94] Sagan, Hans (1994) *Space-Filling Curves*, New York, Springer-Verlag.

[Said y Pearlman 96] Said, A. y W. A. (1996), "A New Fast and Efficient Image Codec Based on Set Partitioning in Hierarchical Trees," *IEEE Transactions on Circuits and Systems for Video Technology*, **6**(6): 243-250, June.

[Salomon 99] Salomon, David (1999) *Computer Graphics and Geometric Modeling*, New York, Springer.

[Salomon 00] Salomon, David (2000) "Prefix Compression of Sparse Binary Strings," *ACM Crossroads Magazine*, **6**(3), February.

[Salomon 06] Salomon, David (2006) *Curves and Surfaces for Computer Graphics*, New York, Springer.

[Samet 90a,b] Samet, Hanan (1990a) *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*, Reading, MA, Addison-Wesley.

Samet, Hanan (1990b) *The Design and Analysis of Spatial Data Structures*, Reading, MA, Addison-Wesley.

[Sampath y Ansari 93] Sampath, Ashwin, y Ahmad C. Ansari (1993) "Combined Peano Scan and VQ Approach to Image Compression," *Image and Video Processing*, Bellingham, WA, SPIE vol. 1903, pp. 175-186.

[Saponara et al. 03] Saponara, Sergio, Luca Fanucci, Pierangelo Terren (2003) "Low-Power VLSI Architectures for 3D Discrete Cosine Transform (DCT)," en *Midwest Symposium on Circuits and Systems (MWSCAS)*.

[Sayood y Anderson 92] Sayood, Khalid y K. (1992) "A Differential Lossless Image Compression Scheme," *IEEE Transactions on Signal Processing*, **40**(1): 236-241, January.

[Sayood 05] Sayood, Khalid (2005) *Introduction to Data Compression*, 3rd Ed., San Francisco, CA, Morgan Kaufmann.

[Schindler 98] Schindler, Michael (1998) "A Fast Renormalisation for Arithmetic Coding," un documento procedente de *Data Compression Conference, 1998*, disponible[38] en la URL `http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=672314`.

[SHA256 02] SHA256 (2002) *Secure Hash Standard*, FIPS Publication 180-2, August 2002. Disponible en `http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf`.

[38] Vieja URL(`http://www.compressconsult.com/rangecoder/`) inválida.

[Shannon 51] Shannon, Claude (1951) "Prediction and Entropy of Printed English," *Bell System Technical Journal*, **30**(1): 50-64, January.

[Shapiro 93] Shapiro, J. (1993) "Embedded Image Coding Using Zerotrees of Wavelet Coefficients," *IEEE Transactions on Signal Processing*, **41**(12): 3445-3462, October.

[Shenoi 95] Shenoi, Kishan (1995) *Digital Signal Processing in Telecommunications*, Upper Saddle River, NJ, Prentice Hall.

[Shlien 94] Shlien, Seymour (1994) "Guide to MPEG-1 Audio Standard," *IEEE Transactions on Broadcasting*, **40**(4): 206-218, December.

[Sieminski 88] Sieminski, A. (1988) "Fast Decoding of the Huffman Codes," *Information Processing Letters*, **26**(5): 237-241.

[Sierpiński 12] Sierpiński, W. (1912) "Sur Une Nouvelle Courbe Qui Remplit Toute Une Aire Plaine," *Bull. Acad. Sci. Cracovie*, **Serie A**: 462-478.

[sighted 06] sighted (2006) es[39] `http://www.indexbraille.com/`.

[Simoncelli y Adelson 90] Simoncelli, Eero P., y Edward. H. Adelson (1990) "Subband Transforms," en *Subband Coding*, John Woods, ed., Boston, MA, Kluwer Academic Press, pp. 143-192.

[Smith 84] Smith, Alvy Ray (1984) "Plants, Fractals and Formal Languages," *Computer Graphics*, **18**(3): 1-10.

[softexperience 06] softexperience (2006) es `http://peccatte.karefil.com/software/Rarissimo/RarissimoEN.htm`.

[Softsound 03] Softsound (2003) era[40] `http://www.softsound.com/`pero pruébese también la URL `http://mi.eng.cam.ac.uk/reports/ajr/TR156/tr156.html`.

[sourceforge.flac 06] sourceforge.flac (2006) es `http://sourceforge.net/projects/flac`.

[Starck et al. 98] Starck, J. L., F. Murtagh, y A. Bijaoui (1998) *Image Processing and Data Analysis: The Multiscale Approach*, Cambridge, UK, Cambridge University Press.

[Stollnitz et al. 95] Stollnitz, E. J., T. D. DeRose, y D. H. Salesin "Wavelets for Computer Graphics: A Primer Part 1", *IEEE Computer Graphics and Applications*, **15**(3): 76-84, May 1995. Disponible como PDF en la URL `http://grail.cs.washington.edu/pub/stoll/wavelet1.pdf`.

Stollnitz, E. J., T. D. DeRose, y D. H. Salesin "Wavelets for Computer Graphics: A Primer Part 2", *IEEE Computer Graphics and Applications*, **15**(4): 75-85, July 1995. Disponible como PDF en la URL `http://grail.cs.washington.edu/pub/stoll/wavelet2.pdf`.

[Stollnitz et al. 96] Stollnitz, E. J., T. D. DeRose, y D. H. Salesin (1996) *Wavelets for Computer Graphics*, San Francisco, CA, Morgan Kaufmann.

[Storer y Szymanski 82] Storer, James A. y T. G. (1982) "Data Compression via Textual Substitution," *Journal of the ACM*, **29**: 928-951.

[Storer 88] Storer, James A. (1988) *Data Compression: Methods and Theory*, Rockville, MD, Computer Science Press.

[39] Vieja URL(`http://www.sighted.com/`) inválida.
[40] Vieja URL(`http://www.softsound.com/Shorten.html`) inválida.

[Storer y Cohn] Storer, James A., y Martin Cohn (eds.) (annual) *DCC 'XX: Data Compression Conference*, Los Alamitos, CA, IEEE Computer Society Press.

[Storer y Helfgott 97] Storer, James A., y Harald (1997) "Lossless Image Compression by Block Matching," *The Computer Journal*, **40**(2/3): 137-145.

[Strang y Nguyen 96] Strang, Gilbert, y Truong (1996) *Wavelets and Filter Banks*, Wellesley, MA, Wellesley-Cambridge Press.

[Strang 99] Strang, Gilbert (1999) "The Discrete Cosine Transform," *SIAM Review*, **41**(1): 135-147.

[Strømme y McGregor 97] Strømme, Øyvind, y Douglas R. (1997) "Comparison of Fidelity of Reproduction of Images After Lossy Compression Using Standard and Nonstandard Wavelet Decompositions," en *Proceedings of The First European Conference on Signal Analysis and Prediction (ECSAP 97)*, Prague, June.

[Strømme 99] Strømme, Øyvind (1999) *On The Applicability of Wavelet Transforms to Image and Video Compression*, Ph.D. thesis, University of Strathclyde, February.

[Stuart et al. 99] Stuart, J. R. et al. (1999) "MLP Lossless Compression," *AES 9th Regional Convention, Tokyo*. Disponible en `http://www.meridian-audio.com/w_paper/mlp_jap_new.PDF`.

[suzannevega 06] suzannevega (2006) es[41] .`http://www.suzannevega.com/suzanne/fun-facts/`.

[Swan 93] Swan, Tom (1993) *Inside Windows File Formats*, Indianapolis, IN, Sams Publications.

[Sweldens y Schröder 96] Sweldens, Wim y Peter (1996), *Building Your Own Wavelets At Home*, SIGGRAPH 96 Course Notes. Disponible en la URL `http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.141.422`.

[Symes 03] Symes, Peter D. (2003) *MPEG-4 Demystified*, New York, NY, McGraw-Hill Professional.

[Taubman 99] Taubman, David (1999) "High Performance Scalable Image Compression with EBCOT," *IEEE Transactions on Image Processing*, **9**(7): 1158-1170.

[Taubman y Marcellin 02] Taubman, David S., y Michael W. (2002) *JPEG 2000, Image Compression Fundamentals, Standards and Practice*, Norwell, MA, Kluwer Academic.

[Thomborson 92] Thomborson, Clark, (1992) "The V.42bis Standard for Data-Compressing Modems," *IEEE Micro*, pp. 41-53, October.

[Thomas Dolby 06] Thomas Dolby (2006) es[42] `http://thomasdolby.com/`.

[Trendafilov et al. 02] Trendafilov, Dimitre, Nasir Memon, y Torsten Suel (2002) "Zdelta: An Efficient Delta Compression Tool," Technical Report TR-CIS-2002-02, New York, NY, Polytechnic University.

[Tunstall 67] Tunstall, B. P., (1967) "Synthesis of Noiseless Compression Codes," Ph.D. dissertation, Georgia Institute of Technology, Atlanta, GA.

[Udupa et al. 99] Udupa, Raghavendra U., Vinayaka D. Pandit, y Ashok Rao (1999), Private Communication.

[Unicode 03] Unicode (2003) es `http://unicode.org/`.

[41] Vieja URL (`http://www.suzannevega.com/about/funfactsMusic.htm`) inválida.
[42] Vieja URL (`http://version.thomasdolby.com/index_frameset.html`) inválida.

[Unisys 03] Unisys (2003) es `http://www.unisys.com`.

[unrarsrc 06] unrarsrc (2006) es `http://www.rarlab.com/rar/unrarsrc-3.5.4.tar.gz`.

[UPX 03] UPX (2003) es `http://upx.sourceforge.net/`.

[UTF16 06] UTF16 (2006) es `http://en.wikipedia.org/wiki/UTF-16`.

[Vetterli y Kovacevic 95] Vetterli, M., y J. (1995) *Wavelets and Subband Coding*, Englewood Cliffs, NJ, Prentice-Hall.

[Vitter 87] Vitter, Jeffrey S. (1987) "Design and Analysis of Dynamic Huffman Codes," *Journal of the ACM*, **34**(4): 825-845, October.

[Volf 97] Volf, Paul A. J. (1997) "A Context-Tree Weighting Algorithm for Text Generating Sources," en Storer, James A. (ed.), *DCC '97: Data Compression Conference*, Los Alamitos, CA, IEEE Computer Society Press, pp. 132-139, (Póster).

[Vorobev 83] Vorobev, Nikolai N. (1983) en Ian N. Sneddon (ed.), y Halina Moss (translator), *Fibonacci Numbers*, New Classics Library.

[Wallace 91] Wallace, Gregory K. (1991) "The JPEG Still Image Compression Standard," *Communications of the ACM*, **34**(4): 30-44, April.

[Watson 94] Watson, Andrew (1994) "Image Compression Using the Discrete Cosine Transform," *Mathematica Journal*, **4**(1): 81-88.

[WavPack 06] WavPack (2006) es `http://www.wavpack.com/`.

[Weinberger et al. 96 y 00] Weinberger, M. J., G. Seroussi, y G. Sapiro (1996) "LOCO-I: A Low Complexity, Context-Based, Lossless Image Compression Algorithm," en *Proceedings of Data Compression Conference*, J. Storer, editor, Los Alamitos, CA, IEEE Computer Society Press, pp. 140-149.

Weinberger, M. J., G. Seroussi, y G. Sapiro (2000) "The LOCO-I Lossless Image Compression Algorithm: Principles and Standardization Into JPEG-LS," *IEEE Transactions on Image Processing*, **9**(8): 1309-1324, August.

[Welch 84] Welch, T. A. (1984) "A Technique for High-Performance Data Compression," *IEEE Computer*, **17**(6): 8-19, June.

[Wikipedia 03] Wikipedia (2003) es el archivo `Nyquist-Shannon sampling theorem`[43] en `http://www.wikipedia.org/wiki/`.

[wiki.audio 06] wiki.audio (2006) es `http://en.wikipedia.org/wiki/Audio_data_compression`.

[Willems 89] Willems, F. M. J. (1989) "Universal Data Compression and Repetition Times," *IEEE Transactions on Information Theory*, **IT-35**(1): 54-58, January.

[Willems et al. 95] Willems, F. M. J., Y. M. Shtarkov, y Tj. J. Tjalkens (1995) "The Context-Tree Weighting Method: Basic Properties," *IEEE Transactions on Information Theory*, **IT-41**: 653-664, May.

[Williams 91a] Williams, Ross N. (1991a) *Adaptive Data Compression*, Boston, MA, Kluwer Academic Publishers.

[43] `Hiperenlace(dic.2011):http://en.wikipedia.org/wiki/Nyquist%E2%80%93Shannon_sampling_theorem.`

[Williams 91b] Williams, Ross N. (1991b) "An Extremely Fast Ziv-Lempel Data Compression Algorithm," en *Proceedings of the 1991 Data Compression Conference*, J. Storer, ed., Los Alamitos, CA, IEEE Computer Society Press, pp. 362-371.

[Williams 93] Williams, Ross N. (1993), "A Painless Guide to CRC Error Detection Algorithms," disponible en `http://ross.net/crc/download/crc_v3.txt`.

[WinAce 03] WinAce (2003) es `http://www.winace.com/`.

[windots 06] windots (2006) es `http://www.uiciechi.it/vecchio/cnt/schede/windots-eng.html`.

[Wirth 76] Wirth, N. (1976) *Algorithms + Data Structures = Programs*, 2nd ed., Englewood Cliffs, NJ, Prentice-Hall.

[Witten et al. 87] Witten, Ian H., Radford M. Neal, y John G. Cleary (1987) "Arithmetic Coding for Data Compression," *Communications of the ACM*, **30**(6): 520-540.

[Witten y Bell 91] Witten, Ian H. y Timothy C. Bell (1991) "The Zero-Frequency Problem: Estimating the Probabilities of Novel Events in Adaptive Text Compression," *IEEE Transactions on Information Theory*, **IT-37**(4): 1085-1094.

[Witten et al. 92] Witten, Ian H., T. C. Bell, M. E. Harrison, M. L. James, y A. Moffat (1992) "Textual Image Compression," en *Proceedings of the 1992 Data Compression Conference*, J. Storer, ed., Los Alamitos, CA, IEEE Computer Society Press, pp. 42-51.

[Witten et al. 94] Witten, Ian H., T. C. Bell, H. Emberson, S. Inglis, y A. Moffat, (1994) "Textual image compression: two-stage lossy/lossless encoding of textual images," *Proceedings of the IEEE*, **82**(6): 878-888, June.

[Wolf et al. 00] Wolf, Misha et al. (2000) "A Standard Compression Scheme for Unicode," Unicode Technical Report #6, disponible en `http://unicode.org/unicode/reports/tr6/index.html`.

[Wolff 99] Wolff, Gerry (1999) Búsquese "`SP theory`" en[44] la URL `http://www.cognitionresearch.org.uk/` .

[Wong y Kuo 93] Wong, Kwo-Jyr, y C. C. Jay Kuo (1993) "A Full Wavelet Transform (FWT) Approach to Image Compression," *Image and Video Processing*, Bellingham, WA, SPIE vol. 1903: 153-164.

[Wong y Koplowitz 92] Wong, P. W., y J. (1992) "Chain Codes and Their Linear Reconstruction Filters," *IEEE Transactions on Information Theory*, **IT-38**(2): 268-280, May.

[Wright 39] Wright, E. V. (1939) *Gadsby*, Los Angeles, Wetzel. Reprinted by University Microfilms, Ann Arbor, MI, 1991.

[Wu 95] Wu, Xiaolin (1995), "Context Selection and Quantization for Lossless Image Coding," en James A. Storer and Martin Cohn (eds.), *DCC '95, Data Compression Conference*, Los Alamitos, CA, IEEE Computer Society Press, p. 453.

[Wu 96] Wu, Xiaolin (1996), "An Algorithmic Study on Lossless Image Compression," en James A. Storer, ed., *DCC '96, Data Compression Conference*, Los Alamitos, CA, IEEE Computer Society Press.

[44] La URL(`http://www.cognitionresearch.org.uk/sp.htm`) no lleva directamente a la teoría SP.

[XMill 03] XMill (2003) es[45] `http://sourceforge.net/projects/xmill/`.

[XML 03] XML (2003) es `http://www.xml.com/`.

[Yokoo 91] Yokoo, Hidetoshi (1991) "An Improvement of Dynamic Huffman Coding with a Simple Repetition Finder," *IEEE Transactions on Communications*, **39**(1): 8-10, January.

[Yokoo 96] Yokoo, Hidetoshi (1996) "An Adaptive Data Compression Method Based on Context Sorting," en *Proceedings of the 1996 Data Compression Conference*, J. Storer, ed., Los Alamitos, CA, IEEE Computer Society Press, pp. 160-169.

[Yokoo 97] Yokoo, Hidetoshi (1997) "Data Compression Using Sort-Based Context Similarity Measure," *The Computer Journal*, **40**(2/3): 94-102.

[Yokoo 99a] Yokoo, Hidetoshi (1999a) "A Dynamic Data Structure for Reverse Lexicographically Sorted Prefixes," en *Combinatorial Pattern Matching, Lecture Notes in Computer Science 1645*, M. Crochemore and M. Paterson, eds., Berlin, Springer Verlag, pp. 150-162.

[Yokoo 99b] Yokoo, Hidetoshi (1999b) Private Communication.

[Yoo et al. 98] Yoo, Youngjun, Younggap Kwon, y Antonio Ortega (1998) "Embedded Image-Domain Adaptive Compression of Simple Images," en *Proceedings of the 32nd Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, CA, Nov. 1998.

[Young 85] Young, D. M. (1985) "MacWrite File Format," *Wheels for the Mind*, 1:34, Fall.

[Yu 96] Yu, Tong Lai (1996) "Dynamic Markov Compression," *Dr Dobb's Journal*, pp. 30-31, January.

[Zalta 88] Zalta, Edward N. (1988) "Are Algorithms Patentable?" *Notices of the American Mathematical Society*, **35**(6): 796-799.

[Zandi et al. 95] Zandi A., J. Allen, E. Schwartz, y M. Boliek, (1995), "CREW: Compression with Reversible Embedded Wavelets," en James A. Storer and Martin Cohn (eds.) *DCC '95: Data Compression Conference*, Los Alamitos, CA, IEEE Computer Society Press, pp. 212-221, March.

[Zhang 90] Zhang, Manyun (1990) *The JPEG and Image Data Compression Algorithms* (disertación).

[Ziv y Lempel 77] Ziv, Jacob, y A. Lempel (1977) "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory*, **IT-23**(3): 337-343.

[Ziv y Lempel 78] Ziv, Jacob y A. (1978) "Compression of Individual Sequences via Variable- Rate Coding," *IEEE Transactions on Information Theory*, **IT-24**(5): 530-536.

[zlib 03] zlib (2003) es `http://www.zlib.org/zlib_tech.html`.

[Zurek 89] Zurek, Wojciech (1989) "Thermodynamic Cost of Computation, Algorithmic Complexity, and the Information Metric," *Nature*, **341**(6238): 119-124, September 14.

[45] La URL (`http://www.research.att.com/sw/tools/xmill/`) no lleva a XMill.

Puesto que la guía es fragmentaria, incompleta, y
en ningún sentido, una bibliografía. Su énfasis varía acorde
a mis propias indiferencias e ignorancia, además de
estar en armonía con mis propias aficiones y conocimiento.

J. Frank Dobie, Guide to Life and Literature of the Southwest[46] (1943)

[46] Guía para la Vida y la literatura del Sudoeste.

Glosario

7-Zip. Un archivador de ficheros con un índice de compresión alto. Idea original de Igor Pavlov, este software libre para Windows se basa en el algoritmo LZMA. Tanto 7z como LZMA fueron diseñados para proporcionar una compresión alta, una descompresión rápida, y requerimientos bajos de memoria para la descompresión. (Véase también: LZMA.)

AAC. Un método de compresión de audio complejo y eficiente. AAC es una extensión de y el sucesor de mp3. Como mp3, AAC es un codificador tiempo/frecuencia (T/F), que emplea un modelo psicoacústico para determinar cómo varía el umbral normal de audición en el presencia de sonidos de enmascaramiento. Una vez que se conoce el umbral de perturbación, las muestras del audio original se convierten en coeficientes de frecuencia cuantificados (proporcionando de este modo una compresión con pérdida) y luego son codificados mediante Huffman (proporcionando una compresión adicional, sin pérdidas).

AC-3. Un codificador de audio perceptivo diseñado por Laboratorios Dolby para soportar varios canales de audio.

ACB. Un método de compresión de texto muy eficaz, de G. Buyanovsky (Sección 8.3). Utiliza un diccionario con contextos y contenidos ilimitados para seleccionar el contexto que coincida mejor con el buffer de búsqueda y el contenido que se adapte mejor al buffer de lectura anticipada.

Adaptive Compression (*Compresión adaptativa*). Un método de compresión que modifica sus operaciones y/o sus parámetros de acuerdo con los nuevos datos leídos desde el flujo de datos de entrada. Ejemplos son: el método de Huffman adaptativo de la Sección 2.9 y los métodos basados en diccionarios del Capítulo 3. (Véase también: Semiadaptive Compression, Locally Adaptive Compression.)

Affine Transformations (*Transformaciones afines*). Transformaciones geométricas bidimensionales o tridimensionales, tales como el escalado, la reflexión, la rotación y la traslación, que conservan las líneas paralelas (Sección 4.35.1)

Alphabet (*Alfabeto*). El conjunto de todos los símbolos posibles de la secuencia de datos de entrada. En la compresión de texto, el alfabeto es normalmente el conjunto de los 128 códigos del ASCII estándar. En la compresión de imágenes, es el conjunto de los valores que un píxel puede tomar (2, 16, 256, o cualquier otra cosa). (Véase también: Symbol.)

ALS. La codificación sin pérdidas de audio (Audio Lossless Coding o ALS) de MPEG-4 es la última adición a la familia de codificadores de audio de MPEG-4. ALS puede manejar muestras de audio en enteros y en coma flotante, y está basada en una combinación de predicción lineal (tanto con términos cortos como con términos largos), codificación multicanal, y una eficiente codificación de los residuos de audio por medio de códigos de Rice y códigos de bloque. ARC.

ARC. Un programa de compresión/archivo/catalogación escrito por Robert A. Freed a mediados de 1980 (Sección 3.22). Ofrece una buena compresión y la capacidad de combinar varios archivos en un fichero. (Véase también: Archive, ARJ.)

Archive (*Archivo*). Un conjunto de uno o más archivos combinados en un fichero (Sección 3.22). Los miembros individuales de un archivo pueden ser comprimidos. Un archivo proporciona una manera conveniente para la transferencia o el almacenamiento de grupos de archivos relacionados. (Véase también: ARC, ARJ.)

Arithmetic Coding (*Codificación aritmética*). Un método de compresión estadística (Sección 2.14), que asigna un código (normalmente largo) a una cadena de entrada completa, en lugar de asignar códigos individuales a cada símbolo. El método lee la secuencia de datos de entrada símbolo a símbolo y anexa más bits al código cada vez que un símbolo es introducido y procesado. La codificación aritmética es lenta; pero el rendimiento de la compresión está muy cerca del valor de entropía, incluso cuando las probabilidades de los símbolos están sesgadas. (Véase también: Model of Compression, Statistical Methods, QM Coder.)

ARJ. Una utilidad libre de compresión/gestión de archivos para MS-DOS (Sección 3.22), escrita por Robert K. Jung para competir con ARC y las distintas utilidades PK. (Véase también: Archive, ARC).

Array (*Matriz o arreglo*). Se utiliza para indicar una estructura de datos de elementos adyacentes del mismo tipo. La estructura entera se almacena en un solo bloque y, como consecuencia, el acceso a sus elementos será más rápido. En este libro aparece casi siempre sin traducir por ser un término muy común en informática.

ASCII Code (*Código ASCII*). El código de caracteres estándar en todos los equipos modernos (aunque Unicode se está convirtiendo en un competidor). ASCII proviene de American Standard Code for Information Interchange (Código eStándar Americano para el Intercambio de Información). Es un código de $(1 + 7)$ bits, con un bit de paridad y siete bits de datos por símbolo. Como consecuencia, pueden ser codificados 128 símbolos. Incluye las letras mayúsculas y las minúsculas, los diez dígitos, algunos signos de puntuación y caracteres de control. (Véase también: Unicode.)

Bark. Unidad de tasa de banda crítica. Denominado así en honor a Heinrich Georg Barkhausen y utilizado en aplicaciones de audio. La escala de Bark es un mapeo no lineal de la escala de frecuencias sobre el rango del audio, una asignación que empareja las frecuencias selectivamente para que sean perceptibles por el oído humano.

Bayesian Statistics (*Estadísticas bayesianas*). (Véase Conditional Probability.)

Bi-level Image (*Imagen binivel*). Una imagen cuyos píxeles tienen dos colores diferentes. Los colores se identifican normalmente como blanco y negro, "foreground (primer plano)" y "background (fondo)", ó 1 y 0. (Véase también: Bitplane.)

BinHex.Un formato de archivo para la transferencia de archivos de forma fiable, diseñado por Yves Lempereur para su uso en el ordenador Macintosh (Sección 1.4.3).

Bintrees. Un método, algo similar a quadtrees, para dividir una imagen en partes no superpuestas. La imagen se divide (horizontalmente) en dos mitades, cada mitad se divide (verticalmente) en pequeñas mitades, y el proceso continúa de forma recursiva, alternando entre divisiones horizontales y verticales. El resultado es un árbol binario donde cualquier parte uniforme de la imagen se convierte en una hoja. (Véase también: Prefix Compression, Quadtrees.)

Bitplane (*Plano de bits*).Cada píxel de una imagen digital está representado por varios bits. El conjunto de todos los k-ésimos bits de todos los píxeles de la imagen es el k-ésimo plano de bits de la imagen. Una imagen binivel, por ejemplo, consta de un bitplane. (Véase también: Bi-level Image.)

Bitrate (*Tasa de bits*). En general, el término "bitrate" se refiere tanto a bpb (bit por bit) como a bpc (bit por carácter). Sin embargo, en la compresión de audio, este término se utiliza para indicar la velocidad a la que se lee el *stream* comprimido por el decodificador. Esta tasa depende de dónde

proviene dicha secuencia de datos (de un disco, de un canal de comunicaciones, de la memoria). Si la tasa de bits de un archivo de audio MPEG es, e.g., 128 Kbps. entonces, el codificador convertirá cada segundo de audio en 128 K bits de datos comprimidos, y el decodificador convertirá cada grupo de 128 K bits de datos comprimidos en un segundo de sonido. Un menor bitrate genera archivos más pequeños. Sin embargo, a medida que disminuye la tasa de bits, el codificador debe comprimir más datos de audio en menos bits; eventualmente produce una pérdida notable de la calidad del audio. Para obtener un audio de calidad CD, la experiencia indica que las mejores tasas de bits están en el intervalo de 112 Kbps a 160 Kbps. (Véase también Bits/Char, Stream.)

Bits/Char (*Bits/Car.*). Bits por carácter (bpc). Una medida del rendimiento en la compresión de texto. También una medida de la entropía. (Véase también: Bitrate, Entropy.)

Bits/Symbol (*Bits/Símb.*). Bits por símbolo. Una medida general del rendimiento de la compresión.

Block Coding (*Codificación en bloques*). Un término general para los métodos de compresión de imágenes que funcionan dividiendo la imagen en pequeños bloques de píxeles, y que codifican cada bloque por separado. JPEG (Sección 4.8) es un buen ejemplo, ya que procesa bloques de 8×8 píxeles.

Block Decomposition (*Descomposición en bloques*). Un método para la compresión sin pérdidas para imágenes en tonos discretos. El método funciona mediante la búsqueda y localización de bloques idénticos de píxeles. Una copia B de un bloque A se comprime preparando la altura, anchura, y la ubicación (coordenadas de la imagen) de A, y comprimiendo ésos cuatro números por medio de códigos de Huffman. (Véase también: Discrete-Tone Image.)

Block Matching (*Emparejamiento de bloques*). Un método de compresión sin pérdidas para imágenes, basado en el método de ventana deslizante de LZ77 desarrollado originalmente para la compresión de texto. (Véase también: LZ Methods.)

Block Truncation Coding (*Codificación por truncamiento de bloques*). BTC es un método de compresión con pérdida para imágenes que cuantifica los píxeles de una imagen al mismo tiempo que preserva los dos o tres primeros *momentos estadísticos*. (Véase también: Vector Quantization.)

BMP. BMP (Sección 1.4.4) es un formato de archivo de gráficos basado en paleta para imágenes con 1, 2, 4, 8, 16, 24, ó 32 planos de bits. Utiliza una forma simple de RLE para comprimir las imágenes con 4 u 8 planos de bits.

BOCU-1. Un sencillo algoritmo para la compresión Unicode (Sección 8.12.1).

BSDiff. Un algoritmo de diferenciación de archivos creado por Colin Percival. El algoritmo está dirigido al problema de la compresión diferencial de archivos de código ejecutable, manteniendo al mismo tiempo la independencia de la plataforma. BSDiff combina las coincidencias con las diferencias y la codificación de entropía de las diferencias, utilizando `bzip2`. El decodificador BSDiff se llama bspatch. (Véase también: Exediff, File differencing, UNIX diff, VCDIFF, y Zdelta.)

Burrows-Wheeler Method (*Método de Burrows-Wheeler*). Este método (Sección 8.1) prepara una serie de datos para su posterior compresión. La compresión en sí misma se realiza con el método move-to-front (mover al frente) (Sección 1.5), quizás en combinación con RLE. El método BW convierte una cadena S a otra cadena L que satisce dos condiciones:

1. Cualquier región de L tenderá a tener una concentración de sólo unos pocos símbolos.
2. Es posible reconstruir la cadena original S a partir de L (pude ser necesario un pequeño conjunto de datos más para la reconstrucción —además de L— pero no muchos).

CALIC. Un método de compresión sin pérdidas de imágenes basada en el contexto (Sección 4.24), cuyas dos características principales son: (1) el uso de tres pasadas con el fin de lograr contextos

simétricos y (2) la cuantificación del contexto, para reducir significativamente el número de contextos posibles, sin degradar la compresión.

CCITT. El Comité Consultivo Internacional Telegráfico y Telefónico (Comité Consultatif International Télégraphique et Téléphonique), el antiguo nombre de la ITU (the International Telecommunications Union; en español, la Unión Internacional de Telecomunicaciones o UIT). La UIT es una organización de las Naciones Unidas responsable de desarrollar y recomendar los estándares para las comunicaciones de datos (no sólo de compresión). (Véase también: ITU.)

Cell Encoding (*Codificación en celdas*). Un método de compresión de imágenes en donde se divide el mapa de bits completo en celdas de, digamos, 8×8 píxeles cada una y se escanea celda a celda. La primera celda se almacena en la entrada 0 de una tabla y se codifica (i.e., se escribe en el archivo comprimido) como el puntero 0. Cada celda posterior se busca en la tabla. Si se encuentra, su índice en la tabla se convierte en su código y se escribe en el archivo comprimido. En caso contrario, se añade a la tabla. Si una imagen está compuesta por sólo segmentos rectos, se puede demostrar que el tamaño de la tabla es de sólo 108 entradas.

CIE. CIE es una abreviatura para Commission Internationale de l'Eclairage (Comisión Internacional para la Iluminación). Esta es la principal organización internacional dedicada a la luz y el color. Es responsable de la elaboración de normas y definiciones en este área. (Véase: Luminance.)

Circular Queue (*Cola circular*). Una estructura de datos básica (Sección 3.3.1) que mueve los datos por el interior de un array (arreglo o matriz) de manera circular, actualizando dos punteros que apuntan al inicio y final de los datos en el arreglo.

Codec (*Codificador*). Un término utilizado para referirse tanto a un codificador como a un decodificador.

Codes (*Códigos*). Un código es un símbolo que representa otro símbolo. En aplicaciones informáticas y de telecomunicaciones, los códigos son virtualmente siempre números binarios. El código ASCII es el estándar de facto, aunque el nuevo Unicode se utiliza en algunos equipos nuevos y el viejo EBCDIC se utiliza aún en ciertas computadoras antiguas de IBM. (Véase también: ASCII, Unicode.)

Composite and Difference Values (*Valores compuestos y diferencias*). Un método de imagen progresiva que separa la imagen en capas utilizando el método *bintrees*. Primero las capas están formadas por unos pocos bloques grandes y de baja resolución; seguidamente, las capas posteriores constan de pequeños bloques y de alta resolución. El principio fundamental consiste en transformar un par de píxeles en dos valores, uno compuesto y un diferenciador. (Véase también: Bintrees, Progressive Image Compression.)

Compress. En el gran mundo de UNIX, `compress` se utiliza comúnmente para comprimir los datos. Esta utilidad usa LZW con un diccionario creciente. Comienza con un pequeño diccionario de sólo 512 entradas y duplica su tamaño cada vez que se llena, hasta que llegue a 64K bytes (Sección 3.18).

Compression Factor (*Factor de compresión*). La inversa de la razón (o relación) de compresión se define como

$$\text{factor de compresión} = \frac{\text{tamaño del stream de entrada}}{\text{tamaño del stream de salida}}.$$

Valores superiores a 1 indican compresión, y valores inferiores a 1 implican expansión. (Véase también: Compression Ratio, Stream).

Compression Gain (*Ganancia de compresión*). Esta medida se define como:

$$100 \log_e \frac{\text{tamaño de referencia}}{\text{tamaño comprimido}},$$

donde el tamaño de referencia es el tamaño del stream de entrada o el tamaño del stream comprimido producido por algún método de compresión sin pérdidas estándar. (Véase también: Stream)

Compression Ratio (*Ratio o razón de compresión*). Una de varias medidas que se utilizan comúnmente para expresar el eficiencia de un método de compresión. Es la relación

$$\text{razón de compresión} = \frac{\text{tamaño del stream de salida}}{\text{tamaño del stream de entrada}}.$$

Un valor de 0,6 indica que los datos ocupan el 60 % de su tamaño original después de la compresión. Valores mayores que 1 significan un stream de salida más grande que el stream de entrada (compresión negativa).

A veces, se utiliza la cantidad $100\times(1 - \text{ratio de compresión})$ para expresar la calidad de la compresión. Un valor de 60 significa que el stream de salida ocupa el 40 % de su tamaño original (o que la compresión ha producido un ahorro del 60 %). (Véase también: Compression Factor, Stream).

Conditional Image RLE (*Imagen condicional RLE*). Un método de compresión para imágenes en escala de grises con n tonos de gris. El método comienza asignando un código de n bits para cada píxel, en función de su los vecinos más cercanos. A continuación, concatena los códigos de n bits en una cadena larga, y calcula los *run lengths*. Los *run lengths* son codificados mediante códigos de prefijos. (Véase también RLE, Relative Encoding, Run length.)

Conditional Probability (*Probabilidad condicional*). Tendemos a pensar en la probabilidad como algo que se construye en un experimento. Un verdadero dado, por ejemplo, tiene una probabilidad de 1/6 de caer sobre cualquier lado, y tendemos a considerar ésta como una característica intrínseca del dado. La probabilidad condicional es una manera diferente de ver la probabilidad. Se dice que el conocimiento afecta a la probabilidad. La principal tarea de este campo es calcular la probabilidad de un evento A, sabiendo que otro evento —B— ya ha ocurrido. Ésta es la probabilidad condicional de A (de forma más precisa, la probabilidad de A condicionada por B), y se denota por $P(A \mid B)$. El campo de la probabilidad condicional se llama a veces *estadística Bayesiana*, ya que fue desarrollada por primera vez por el Reverendo Thomas Bayes, quien la construyó a partir de la fórmula básica de la probabilidad condicional.

Context (*Contexto*). Los N símbolos que preceden a uno dado. Un modelo basado en el contexto utiliza el contexto para asignar probabilidades a los símbolos.

Context-Free Grammars (*Gramáticas libres de contexto*). Un lenguaje formal utiliza un pequeño número de símbolos (llamados *símbolos terminales*) con los que se pueden construir secuencias válidas. Cualquier secuencia válida es finita, el número de secuencias válidas es normalmente ilimitada, y las secuencias se construyen acorde con ciertas reglas (a veces llamadas *reglas de producción*). Las reglas se pueden usar para construir secuencias válidas y también para determinar si una secuencia dada es válida. Una regla de producción consta de un símbolo no terminal a la izquierda y una cadena de símbolos terminales y no terminales a la derecha. El símbolo no terminal de la izquierda se convierte en el nombre de la cadena de la derecha. El conjunto de reglas de producción constituye la gramática del lenguaje formal.Si las normas de producción no dependen del contexto de un símbolo, la gramática es libre de contexto. También hay gramáticas sensibles al contexto. El método sequitur (de incongruencia) de la Sección 8.10 se basa en gramáticas libres de contexto.

Context-Tree Weighting (*Ponderación del árbol de contexto*). Un método para la compresión de cadenas de bits. Se puede aplicar a texto e imágenes, pero tienen que ser cuidadosamente convertidas en cadenas de bits. El método construye un árbol de contexto en el que los bits introducidos en el pasado inmediato (el contexto) se utilizan para estimar la probabilidad de aparición del bit actual. El bit actual y su probabilidad estimada se envían a un codificador aritmético, y el árbol se actualiza para incluir el bit actual en el contexto. (Véase también: KT Probability Estimator.)

Continuous-Tone Image (*Imagen de tonos continuos*). Una imagen digital con un gran número de colores, de tal manera que las áreas adyacentes de la imagen con colores que se diferencian en una sola unidad se perciben por la vista como una variación continua de los colores. Un ejemplo es una imagen con 256 valores en escala de grises. Cuando los píxeles adyacentes en dicha imagen tienen niveles consecutivos de grises, se muestran a la vista como una variación continua del nivel de gris. (Véase también: Bi-level image, Discrete-Tone Image, Grayscale Image.)

Continuous Wavelet Transform (*Transformada wavelet continua*). Un método moderno importante para el análisis del tiempo y de la frecuencia contenidos en una función $f(t)$ por medio de una wavelet. La wavelet es en sí una función (que tiene que satisfacer ciertas condiciones), y la transformación se realiza multiplicando la wavelet (onda pequeña) y $f(t)$, y calculando la integral del producto. Después, la wavelet es interpretada, y se repite el proceso. Cuando se ha terminado, la onda se escala, y todo el proceso se lleva a cabo de nuevo con el fin de analizar $f(t)$ en una escala diferente. (Véase también: Discrete Wavelet Transform, Lifting Scheme, Multiresolution Decomposition, Taps.)

Convolution (*Convolución*). Una manera de describir la salida de un sistema de desplazamiento invariante lineal por medio de su entrada.

Correlation (*Correlación*). Una medida estadística de la relación lineal entre dos variables pareadas. R toma los valores: de -1 (relación negativa perfecta), a 0 (ninguna relación), a $+1$ (relación positiva perfecta).

CRC. CRC procede de *Cyclical Redundancy Check* (or *Cyclical Redundancy Code*) —*control de redundancia cíclica* (o *código de redundancia cíclica*)—. Es una regla que muestra cómo obtener bits de verificación vertical de todos los bits de un *stream* de datos (Sección 3.28). La idea es generar un código que depende de todos los bits del *stream* de datos, y utilizarlo para detectar los errores (bits erróneos) cuando se transmiten los datos (o cuando se almacenan y se recuperan).

CRT. Un tubo de rayos catódicos (CRT o cathode ray tube) es un tubo de vidrio con una forma familiar. En la parte posterior tiene un cañón de electrones (el cátodo) que emite una corriente de electrones. Su superficie frontal está cargada positivamente, por lo que atrae a los electrones (que tienen una carga eléctrica negativa). La parte frontal está recubierta con un compuesto de fósforo que convierte la energía cinética de los electrones que golpean en luz. El flash de luz dura sólo una fracción de segundo, por lo que para conseguir una visualización constante, la imagen tiene que ser refrescada varias veces por segundo.

Data Compression Conference (*Conferencia sobre compresión de datos*). Una reunión de investigadores y desarrolladores en el área de compresión de datos. La DCC se lleva a cabo cada año en Snowbird, Utah, USA. Tiene una duración de tres días y las próximas reuniones están previstas para finales de marzo.

Data Structure (*Estructura de datos*). Un conjunto de ítems de datos utilizado por un programa y almacenado en la memoria de tal manera que ciertas operaciones (por ejemplo, búsquedas , adiciones, modificaciones y eliminaciones de elementos) pueden llevarse a cabo sobre los ítems de datos, rápida y fácilmente. Las estructuras de datos más comunes son: array (arreglo o matriz), pila, cola, lista enlazada, árbol, grafo y tabla hash. (Véase también: Circular Queue.)

Decibel (*Decibelio*). Una medida logarítmica que se puede utilizar para medir cualquier cantidad que toma valores por encima de una gama muy amplia. Un ejemplo común es la intensidad del sonido. La intensidad (amplitud) del sonido puede variar en un rango de $11-12$ órdenes de magnitud. En lugar de utilizar una medida lineal, donde se necesitarían números tan pequeños como 1 y tan grandes como 10^{11}, se usa una escala logarítmica, donde el rango de valores es $[0, 11]$.

Decoder (*Decodificador*). Un programa (o algoritmo) de descompresión.

Deflate (*Deflación*). Un popular algoritmo de compresión sin pérdidas (Sección 3.23) utilizado por Zip y gzip. Deflate emplea una variante de LZ77 combinada con una codificación de Huffman estática. Utiliza un diccionario deslizante de 32 Kb de longitud, y un buffer de preanálisis de 258 bytes. Cuando una cadena no se encuentra en el diccionario, su primer símbolo se emite como un byte literal. (Véase también: Gzip, Zip.)

Dictionary-Based Compression (*Compresión basada en diccionarios*). Métodos de compresión (Capítulo 3) que guardan trozos de datos en una estructura de datos de "diccionario". Si una cadena con nuevos datos es idéntica a un pedazo ya guardado en el diccionario, se proporciona como salida al stream comprimido un puntero a dicho pedazo. (Véase también: LZ Methods.)

Differential Image Compression (*Compresión de imágenes diferencial*). Un método de compresión de imágenes sin pérdidas en el que cada píxel p es comparado con un *píxel de referencia*, el cual es uno de sus vecinos inmediatos y, a continuación, es codificado en dos partes: un prefijo, que es el número formado por los bits más significativos de p que son idénticos a los del píxel de referencia, y un sufijo, que es (prácticamente todos) los bits restantes menos significativos de p. (Véase también: DPCM).

Digital Video (*Vídeo digital*). Una forma de video en la cual se genera la imagen original, en la cámara, en forma de píxeles. (Véase también: High-Definition Television.)

Digram (*Digrama*). Un par de símbolos consecutivos.

Discrete Cosine Transform (*Transformada discreta del coseno*). Una variante de la transformada discreta de Fourier (DFT) que produce sólo números reales. La DCT (Secciones 4.6, 4.8.2 y 8.15.2) transforma un conjunto de números combinando n de ellos para convertirlos en un punto n-dimensional y lo rota en n-dimensiones de forma que la primera coordenada se hace dominante. La DCT y su inversa, la IDCT, se utilizan en JPEG (Sección 4.8) para comprimir una imagen con pérdidas aceptables, aislando los componentes de alta frecuencia de una imagen, de modo que más tarde puedan ser cuantificados. (Véase también: Fourier Transform, Transform.)

Discrete-Tone Image (*Imagen de tonos discretos*). Una imagen de tonos discretos puede ser: binivel, en escala de grises o en color. Estas imágenes son (con algunas excepciones) artificiales, y han sido obtenidas mediante el escaneo de un documento, o la captura de la pantalla de un ordenador. Los colores de los píxeles de dichas imágenes no varían de forma continua o suavemente, sino que tienen un pequeño conjunto de valores, de manera que los píxeles adyacentes pueden diferir mucho en intensidad o color. La Figura 4.57 es un ejemplo de tales imágenes. (Véase también: Block Decomposition, Continuous-Tone Image.)

Discrete Wavelet Transform (*Transformada wavelet discreta*). Es la versión discreta de la transformada wavelet continua. Una wavelet se representa por medio de varios coeficientes filtro, y la transformada se lleva a cabo mediante multiplicación de matrices (o una versión más simple de la misma) en lugar de mediante el cálculo integral. (Véase también: Continuous Wavelet Transform, Multiresolution Decomposition.)

DjVu. Ciertas imágenes combinan las propiedades de los tres tipos de imágenes (binivel, de tonos discretos, y de tonos continuos). Un importante ejemplo de estas imágenes es un documento escaneado que contiene texto, dibujos, líneas y regiones con imágenes de tonos continuos, como pinturas o fotografías. DjVu (pronunciado "déjà vu") está diseñado para una alta compresión y una rápida descompresión de dichos documentos.

Comienza descomponiendo el documento en tres componentes: máscara, primer plano, y fondo. El último contiene los píxeles que constituyen las imágenes y el fondo del documento. La máscara contiene el texto y las líneas en formato binivel (i.e., un bit por píxel). El primer plano contiene el color de los píxeles de la máscara. El fondo es una imagen de tonos continuos y se puede comprimir en la baja

resolución de 100 dpi. El primer plano normalmente contiene grandes áreas uniformes y se comprime también como una imagen de tonos continuos con la misma baja resolución. La máscara queda a 300 dpi, pero puede ser comprimida eficientemente, ya que es binivel. El fondo y el primer plano se comprimen con un método basado en wavelet llamado IW44, mientras que la máscara se comprime con JB2, una versión de JBIG2 (Sección 4.12), desarrollado en AT&T.

DPCM. La compresión DPCM es un miembro de la familia de los métodos de compresión y codificación diferencial, que en sí es una generalización del sencillo concepto de codificación relativa (Sección 1.3.1). Se basa en el hecho de que los píxeles vecinos en una imagen (y también las muestras adyacentes en el sonido digitalizado) están correlacionados. (Véase también: Differential Image Compression, Relative Encoding.)

Embedded Coding. (*Codificación embebida*). Esta característica se define como sigue: Imagine que un codificador de imágenes se aplica dos veces a la misma imagen, con diferentes cantidades de pérdidas. Se producen dos archivos: uno de gran tamaño M y otro de pequeño tamaño m. Si el codificador utiliza la codificación embebida, el archivo más pequeño es idéntico a los m bits iniciales del archivo más grande.

El siguiente ejemplo ilustra muy bien el significado de esta definición. Supongamos que tres usuarios esperan el envío de cierta imagen comprimida, pero necesitan diferentes calidades de imagen. El primero de ellos necesita la calidad que ofrece un archivo de 10 Mb. Las calidades de imagen requeridas por los usuarios segundo y tercero responden a tamaños de archivo de 20 Kb y 50 Kb, respectivamente. La mayoría de los métodos compresión de imágenes con pérdidas tendrían que comprimir la misma imagen tres veces, en distintas calidades, para generar tres archivos con los tamaños adecuados. Un codificador embebido, por otro lado, produce un archivo, y luego puede enviar tres trozos —de longitudes 10 Kb, 20 Kb y 50 Kb, todos comenzando al principio del fichero— a los tres usuarios, satisfaciendo sus necesidades. (Véase también: SPIHT, EZW.)

Encoder (*Codificador*). Un programa (o algoritmo) de compresión.

Entropy (*Entropía*). La entropía de un solo símbolo a_i se define (en la Sección 2.1) como $-P_i \log_2 P_i$, donde P_i es la probabilidad de ocurrencia de a_i en los datos. La entropía de a_i es el número más pequeño de bits necesarios, en promedio, para representar el símbolo a_i. Claude Shannon, el creador de la teoría de la información, acuñó el término *entropía* en 1948, porque este término es utilizado en termodinámica para indicar la cantidad de desorden en un sistema físico. (Véase también: Entropy Encoding, Information Theory.)

Entropy Encoding (*Codificación de entropía*). Un método de compresión sin pérdidas donde se pueden comprimir los datos de manera que el número medio de bits/símbolo se aproxima a la entropía de los símbolos entrantes. (Véase también: Entropy).

Error-Correcting Codes (*Códigos correctores de errores*). La cara opuesta de la compresión de datos; estos códigos detectan y corrigen errores en los datos digitales aumentando la redundancia de los mismos. Utilizan bits de vefificación o bits de paridad, y algunas veces se diseñan con la ayuda de polinomios generadores.

EXE Compressor (*Compresor EXE*). Un programa de compresión para comprimir archivos EXE en el PC. Este tipo de archivo comprimido puede ser descomprimido y ejecutado con un solo comando. El compresor EXE original es LZEXE, de Fabrice Bellard (Sección 3.27).

Exediff. Un algoritmo de compresión de archivos diferencial creado por Brenda Baker, Udi Manber, y Robert Muth para la compresión diferencial del código ejecutable. Exediff es un algoritmo iterativo que utiliza una transformación con pérdidas para reducir el efecto de los cambios secundarios en el código ejecutable. Dos operaciones denominadas *pre-matching* (emparejamiento previo) y *value recovery* (valor de recuperación), son iteradas hasta que el tamaño del *patch* (parche o corrección) converge

a un mínimo. El decodificador Exediff se llama exepatch. (Véase también: BSdiff, File differencing, UNIX diff, VCDIFF, y Zdelta.)

EZW. Un método de codificación de imágenes progresivo y embebido basado en la estructura de datos árbol-cero (*zerotree*). Ha sido ampliamente superado por el más eficiente método SPIHT. (Véase también SPIHT, Progressive Image Compression, Embedded Coding.)

Facsimile Compression (*Compresión de facsímiles*). La transferencia de una página típica entre dos máquinas de fax puede tomar hasta 10−11 minutos sin compresión. Esta es la razón por la que la ITU (UIT en español) ha desarrollado varios estándares para la compresión de datos de fax. Los estándares actuales (Sección 2.13) son T4 y T6, también llamados Grupo 3 y Grupo 4, respectivamente. (Véase también: UIT.)

FELICS. Un método de compresión de imágenes rápido, eficiente, y sin pérdidas, diseñado para imágenes en escala de grises, que compite con el modo sin pérdidas de JPEG. El principio consiste en codificar cada píxel con un código de tamaño variable basado en los valores de dos de sus píxeles vecinos vistos anteriormente. Se utiliza tanto el código unario como el de Golomb. También existe una versión progresiva de FELICS (Sección 4.20). (Véase también: Progressive FELICS.)

FHM Curve Compression (*Compresión de curvas FHM*). Un método para la compresión de curvas. El acrónimo FHM proviene de Fibonacci, Huffman, y Markov. (Véase también: Fibonacci Numbers.)

Fibonacci Numbers (*Números de Fibonacci*). Una secuencia de números definida por

$$F_1 = 1, \quad F_2 = 1, \quad F_i = F_{i-1} + F_{i-2}, \quad i = 3, 4, \ldots$$

Los primeros números en la secuencia son 1, 1, 2, 3, 5, 8, 13 y 21. Estos números tienen muchas aplicaciones en matemáticas y en ciencias. También se encuentran en la naturaleza, y están relacionados con la proporción áurea. (Véase también: FHM Curve Compression.)

File Differencing (*Diferenciación de archivos*). Un método de compresión que localiza y comprime las diferencias entre dos conjuntos de datos ligeramente diferentes. El decodificador, que tiene acceso a uno de los dos conjuntos de datos, puede utilizar las diferencias y reconstruir el otro. Las aplicaciones de esta técnica de compresión incluyen la distribución de software y actualizaciones (o parches), la revisión de sistemas de control, la compresión de archivos de copia de seguridad, almacenamiento de varias versiones de los datos. (Véase También: VCDIFF.) (Véase también: BSdiff, Exediff, UNIX diff, VCDIFF, y Zdelta.)

FLAC. Un acrónimo para *free lossless audio compression* (compresión de audio sin pérdidas libre); FLAC es un método de compresión de audio, algo parecido a *Shorten*, que se basa en la predicción de las muestras de audio (*samples*) y la codificación de los residuos de predicción con códigos de Rice. (Véase también: Rice codes.)

Fourier Transform (*Transformada de Fourier*). Una transformación matemática que produce los componentes de frecuencia de una función (Sección 5.1). La transformada de Fourier muestra cómo una función periódica puede escribirse como la suma de senos y cosenos, lo que muestra de forma explícita las frecuencias "ocultas" en la representación original de la función. (Véase también: Discrete Cosine Transform, Transform.)

Gaussian Distribution (*Distribución de Gauss*). (Véase: Normal Distribution.)

GFA. Un método de compresión desarrollado originalmente para imágenes binivel que también puede ser utilizado para imágenes en color. GFA utiliza el hecho de que la mayoría de las imágenes de interés tienen una cierta cantidad de auto-similitud (i.e., partes de la imagen son similares, hasta el tamaño,

orientación o brillo, a la imagen completa o a otras partes). GFA divide la imagen en subcuadrados utilizando un *quadtree* (árbol cuaternario), y expresa las relaciones entre las partes de la imagen en un grafo. El grafo es similar a los utilizados para describir autómatas de estado finito. El método es con pérdidas, porque las partes de una imagen real pueden ser muy (aunque no completamente) similares a otras partes. (Véase también: Quadtrees, Resolution Independent Compression, WFA.)

GIF. Un acrónimo de *Graphics Interchange Format* (formato de intercambio de gráficos). Este formato (Sección 3.19) fue desarrollado por los Servicios de Información de CompuServe en 1987 como un eficiente formato de archivo de gráficos comprimido que permite enviar imágenes entre ordenadores. La versión original de GIF se conoce como GIF 87a. El estándar actual es GIF 89a. (Véase también: Patents.)

Golomb Code (*Código de Golomb*). Los códigos de Golomb consisten en un conjunto infinito de *códigos prefijo parametrizados*. Son los mejores para la compresión de ítems de datos que estén distribuidos geométricamente. (Véase también: Unary Code.)

Gray Codes (*Códigos de Gray*). Son códigos binarios de números enteros, donde los códigos de enteros consecutivos se diferencian en un bit. Tales códigos se utilizan cuando una imagen en escala de grises se separa en planos de bits, cada uno una imagen binivel. (Véase también: Grayscale Image.)

Grayscale Image (*Imagen en escala de grises*). Una imagen de tonos continuos, con sombras de un solo color. (Véase también: Continuous-Tone Image.)

Growth Geometry Coding (*Codificación de la geometría de crecimiento*). Un método de compresión progresiva sin pérdidas para imágenes binivel. El método selecciona algunos píxeles *semilla* y aplica reglas geométricas para hacer crecer cada píxel semilla hasta conseguir un patrón de píxeles determinado. (Véase también: Progressive Image Compression.)

Gzip. Software popular que implementa el algoritmo *Deflate* (Sección 3.23), que utiliza una variación de LZ77 combinado con la codificación de Huffman estática. Utiliza un diccionario de deslizamiento de 32 Kb de longitud, y un buffer de preanálisis de 258 bytes. Cuando una cadena no se encuentra en el diccionario, se emite como una secuencia de bytes literales. (Véase también: Zip.)

H.261. A finales de 1984, el CCITT (actualmente la UIT-T), organizó un grupo de expertos para desarrollar un estándar para la telefonía visual para los servicios RDSI (ISDN en inglés). La idea era enviar imágenes sonido entre los terminales especiales, por lo que los usuarios pudieran verse y hablar entre sí. Este tipo de aplicación requiere el envío de grandes cantidades de datos, por lo que la compresión se convirtió en una consideración importante. El grupo finalmente llegó con un cierto número de estándares, conocidos como recomendaciones de serie H (para vídeo) y de serie G (para audio), todas operando a velocidades de $p \times 64$ Kbit/seg para $p = 1, 2, \ldots, 30$. Estas normas se conocen hoy bajo el nombre $p \times 64$. (Véase también: ITU, ISDN.)

H.264. Un sofisticado método para la compresión de vídeo. Este método es un sucesor de H.261, H.262, y H.263. Ha sido aprobado en 2003 y emplea los bloques principales de construcción de sus predecesores, pero con muchas adiciones y mejoras.

Halftoning (*Semitonos*). Un método para la visualización de escalas de grises en una imagen binivel. Al colocar grupos de píxels negros y blancos en los patrones cuidadosamente diseñados, es posible crear el efecto de una zona gris. La contraparte de los medios tonos es la pérdida de resolución. (Véase también: Bi-level Image, Dithering.)

Hamming Codes (*Códigos de Hamming*). Un tipo de código de corrección de errores para errores en 1 bit, donde es fácil generar los bits de paridad requeridos.

Hierarchical Progressive Image Compression (*Compr. de imág. progresiva jerárquica*). Un método de compresión de imágenes (o una parte opcional de tal método) donde el codificador escribe la

imagen comprimida en capas de resolución incremental. El decodificador primero descomprime la capa de más baja resolución, muestra esta ruda imagen, y continúa con las capas de mayor resolución. Cada capa en el *stream* comprimido utiliza los datos de la capa precedente. (Véase también: Progressive Image Compression.)

High-Definition Television (*Televisión de alta definición*). Un nombre general para varios estándares que se encuentran actualmente reemplazando la televisión tradicional. HDTV utiliza video digital, imágenes de alta resolución, y relaciones de aspecto diferentes a la tradicional 3:4. (Véase también: Digital Video.)

Huffman Coding (*Codificación de Huffman*). Un popular método para la compresión de datos (Sección 2.8). Asigna un conjunto de los "mejores" códigos de tamaño variable a un conjunto de símbolos basados en sus probabilidades. Sirve como base para varios programas populares comunes en ordenadores personales. Algunos de ellos usan sólo el método de Huffman, mientras que otros lo emplean como un paso en un proceso de compresión multipaso. El método de Huffman es algo similar al método de Shannon-Fano. Generalmente produce mejores códigos, y al igual que el método de Shannon-Fano, genera un mejor código cuando las probabilidades de los símbolos son potencias negativas de 2. La principal diferencia entre los dos métodos es que el de Shannon-Fano construye sus códigos de arriba abajo (de los bits el extremo izquierdo a los del derecho), mientras que Huffman construye un árbol de códigos de abajo arriba (construye los códigos de derecha a izquierda). (Véase también: Shannon-Fano Coding, Statistical Methods.)

Hyperspectral data (*Datos hiperespectrales*). Un conjunto de ítems de datos (llamados píxeles) que se disponen en filas y columnas, donde cada píxel es un vector. Un ejemplo es una imagen donde cada píxel está formado por la radiación reflejada desde el suelo en muchas frecuencias. Podemos pensar en tales datos, como varios planos de la imagen (llamados bandas) que se apilan verticalmente. Los datos hiperespectrales son normalmente grandes y, por ello, candidatos ideales para la compresión. Cualquier método de compresión para este tipo de datos debe tomar la ventaja de la correlación entre las bandas, así como de las correlaciones entre los píxeles de la misma banda.

Information Theory (*Teoría de la Información*). Una teoría matemática que cuantifica la información. Muestra cómo medir la información, de modo que se pueda responder (con un número preciso) a la pregunta: ¿Cuánta información se incluye en un bloque de datos dado? La teoría de la información es la creación, en 1948, de Claude Shannon de los laboratorios Bell. (Véase también: Entropy).

Interpolating Polynomials (*Polinomios de interpolación*). Dados dos números a y b sabemos que $m = 0{,}5a + 0{,}5b$ es su promedio, ya que está ubicado en la mitad entre a y b . Decimos que el promedio es una interpolación de los dos números. Similarmente, la suma ponderada $0{,}1a + 0{,}9b$ representa un valor de interpolación separado un 10 % de b y un 90 % de a. La extensión de este concepto a puntos (en dos o tres dimensiones) se efectúa mediante *polinomios de interpolación*. Dado un conjunto de puntos, comenzamos alojando un polinomio paramétrico $\mathbf{P}(t)$ o $\mathbf{P}(u, w)$ entre ellos. Una vez que se conoce el polinomio, puede usarse para obtener puntos interpolados mediante el cálculo $\mathbf{P}(0{,}5)$, $\mathbf{P}(0{,}1)$, u otros valores.

ISDN (*RDSI*). Acrónimo de *Integrated Services Digital Network* (Red digital de servicios integrados). La UIT-T la definió en 1988 en su libro rojo como "una red que procede por evolución de la Red Digital Integrada (RDI) y que facilita conexiones digitales extremo a extremo para proporcionar una amplia gama de servicios, tanto de voz como de otros tipos, y a la que los usuarios acceden a través de un conjunto de interfaces normalizados."

ISO. Acrónimo de International Standards Organization (Organización de eStándares Internacional). Ésta es una de las organizaciones responsables del desarrollo de estándares. Entre otras cosas, es

responsable (junto con la UIT) de los estándares de compresión JPEG y MPEG. (Véase también: ITU, CCITT, MPEG.)

Iterated Function Systems (IFS) (*Sistemas de funciones iteradas*). Una imagen comprimida mediante IFS está definida únicamente por unas pocas transformaciones afínes (Sección 4.35.1). La única regla es que los factores de escala de estas transformaciones deben ser menores que 1 (*shrinking* o encogimiento). La imagen se guarda en el flujo de datos de salida escribiendo los conjuntos de seis números que definen cada transformación. (Véase también: Affine Transformations, Resolution Independent Compression.)

ITU (*UIT*). La *International Telecommunications Union* (Unión Internacional de Telecomunicaciones), el nuevo nombre de la CCITT, es una organización de las Naciones Unidas responsable de desarrollar y recomendar estándares para las comunicaciones de datos (no sólo sobre compresión). (Véase también: CCITT.)

JBIG. Un método de compresión de propósito especial (Sección 4.11) desarrollado específicamente para la compresión progresiva de imágenes binivel. El nombre JBIG procede de Joint Bi-Level Image Processing Group (grupo conjunto de procesamiento de imágenes binivel). Éste es un grupo de expertos de varias organizaciones internacionales, formado en 1988 para recomendar dicho estándar. JBIG utiliza la codificación aritmética múltiple y una técnica de resolución–reducción para lograr sus objetivos. (Véase también: Bi-level Image, JBIG2.)

JBIG2. Un estándar internacional reciente para la compresión de imágenes binivel. Está destinada a sustituir al JBIG original. Sus principales características son:

1. Grandes aumentos en el rendimiento de la compresión (típicamente 3−5 veces mejor que el Grupo 4/MMR, y 2−4 veces mejor que JBIG).
2. Métodos de compresión especiales para el texto, semitonos, y otras partes de una imagen binivel.
3. Modos de compresión con y sin pérdidas.
4. Dos modos de compresión progresiva. El Modo 1 es la compresión de calidad progresiva, donde la decodificación de la imagen progresa desde la más baja calidad, hasta la más alta. El Modo 2 es la codificación progresiva de contenido, donde las partes importantes de la imagen (como el texto) se decodifican en primer lugar, seguidas por las partes menos importantes (tales como patrones de semitonos).
5. Compresión de documentos multipágina.
6. Formato flexible, diseñado para ser embebido fácilmente en otros formatos de archivos de imagen, tales como TIFF.
7. Descompresión rápida. En algunos modos de codificación, las imágenes pueden ser descomprimidas en más de 250 millones de píxeles/segundo en software.

(Véase también: Bi-level Image, JBIG).

JFIF. El nombre completo de este método (Sección 4.8.7) es JPEG File Interchange Format (formato de intercambio de archivos JPEG). Se trata de un formato de archivo de gráficos que hace posible el intercambio de imágenes JPEG comprimidas entre distintos ordenadores. Las principales características de JFIF son: el uso del espacio de color de componente triple YCbCr para imágenes en color (sólo un componente en escala de grises) y el uso de un *marcador* para especificar las características que faltan en JPEG, tales como resolución de la imagen, la relación de aspecto y características que son específicas de la aplicación.

JPEG. Un sofisticado método de compresión con pérdida (Sección 4.8) para imágenes en color o en escala de grises (películas no). Funciona mejor en imágenes de tonos continuos, donde píxeles adyacentes tienen colores similares. Una ventaja de JPEG es el uso de muchos parámetros, permitiendo al usuario ajustar la cantidad de datos perdidos (y, por tanto, también la razón de compresión) en un rango muy amplio. Hay dos modos principales: con pérdidas (también llamado *baseline* o base) y sin pérdidas (que típicamente produce una relación de compresión 2:1). La mayor parte de las implementaciones soportan sólo el modo con pérdidas. Este modo incluye codificación progresiva y jerárquica.

La idea principal tras JPEG es que una imagen existe para que la gente pueda verla, por lo que al comprimir la imagen, es aceptable perder aquellas características de la imagen a las que el ojo humano no es sensible.

El nombre JPEG es un acrónimo de Joint Photographic Experts Group (grupo conjunto de expertos en fotagrafía). Éste fue un esfuerzo conjunto por parte del CCITT y de la ISO que comenzó en junio de 1987. El estándar JPEG ha sido exitosamente probado y se ha utilizado ampliamente para la presentación de imágenes, especialmente en las páginas Web. (Véase también: JPEG-LS, MPEG.)

JPEG-LS. El modo sin pérdidas de JPEG es ineficiente y muchas veces ni siquiera está implementado. Como resultado, la ISO decidió desarrollar un nuevo estándar para la compresión sin pérdidas (o casi sin pérdidas) de imágenes de tonos continuos. El resultado se conoce popularmente como JPEG-LS. Este método no es simplemente una ampliación o una modificación de los archivos de JPEG. Es un método nuevo, diseñado para ser sencillo y rápido. No emplea la DCT, no utiliza codificación aritmética, y aplica la cuantificación de forma limitada, y sólo en la opción casi sin pérdidas. JPEG-LS examina algunos de los vecinos previamente observados del píxel actual, los usa como *contexto* del píxel; emplea el contexto para predecir el píxel y para seleccionar una distribución de probabilidad entre varias de tales distribuciones, y utiliza ésa distribución para codificar la predicción de errores con un código especial de Golomb. Hay también un modo *run*, donde se codifica la longitud de una racha (*runlength*) de píxeles idénticos. (Véase también: Golomb Code, JPEG.)

> En cuanto a mi madre, quizás el embajador no tenía el tipo de la mente hacia la que ella se sentía más atraída. Debo añadir que su conversación está adornada de un exhaustivo glosario de formas de hablar obsoletas propias de una profesión determinada, clase y período.
>
> —Marcel Proust, *Within a Budding Grove (en un bosque en ciernes)* (1913−1927)

Kraft–MacMillan Inequality (*Desigualdad de Kraft–MacMillan*). Una relación (Sección 2.6) que dice algo acerca de códigos de tamaño variable no ambiguos. Sus primera parte expone: Dado un código de tamaño variable no ambiguo, con n códigos de tamaños L_i, entonces:

$$\sum_{i=1}^{n} 2^{-L_i} \leq 1.$$

[Ésta es la Ecuación 2.5.] La segunda parte indica lo contrario, es decir, dado un conjunto de n números enteros positivos $(L_1, L_2, \ldots, L_n)$ que satisfacen la Ecuación 2.5, existe un código de tamaño variable no ambiguo tal que los L_i son los tamaños de sus códigos individuales. Juntas, ambas partes manifiestan que un código es no ambiguo si y sólo si satisface la relación 2.5.

KT Probability Estimator (*Estimador de probabilidad KT*). Un método para estimar la probabilidad de una cadena de bits que contiene a ceros y b unos. Es debido a Krichevski y Trofimov. (Véase también: Context-Tree Weighting.)

Laplace Distribution (*Distribución de Laplace*). Una distribución de probabilidad similar a la distribución normal (Gaussiana), pero más estrecha y llega rápidamente a un máximo. La distribución general de Laplace con varianza V y media m viene dada por:

$$L(V,x) = \frac{1}{\sqrt{2V}} \exp\left(-\sqrt{\frac{2}{V}} \mid x-m \mid\right).$$

La experiencia parece sugerir que los valores de los residuos calculados por muchos de los algoritmos de compresión de imágenes siguen la distribución de Laplace, por lo que esta distribución es empleada por ésos métodos de compresión, más especialmente MLP. (Véase también: Normal Distribution.)

Laplacian Pyramid (*Pirámide Laplaciana*). Una técnica de compresión de imágenes progresiva donde la imagen original se transforma en un conjunto de imágenes diferencia que posteriormente pueden ser descomprimidas y se muestran como una pequeña imagen borrosa, que se va haciendo cada vez más nítida. (Véase también: Progressive Image Compression.)

LHArc. Este método (Sección 3.22) es de Haruyasu Yoshizaki. Su predecesor es LHA, diseñado conjuntamente por Haruyasu Yoshizaki y Haruhiko Okumura. Estos métodos están basados en la codificación de Huffman adaptativa con características extraídas de LZSS.

Lifting Scheme (*Esquema de elevación*). Un método para el cálculo de la transformada wavelet discreta in situ, por lo que no se requiere memoria adicional alguna. (Véase también: Discrete Wavelet Transform.)

Locally Adaptive Compression (*Compresión localmente adaptativa*). Un método de compresión que se adapta a las condiciones locales en el flujo de datos de entrada, y varía esta adaptación a medida que se mueve de una zona a otra en la entrada. Un ejemplo es el método mover–al–frente de la Sección 1.5. (Véase también: Adaptive Compression, Semiadaptive Compression.)

Lossless Compression (*Compresión sin pérdidas*). Un método de compresión donde la salida del decodificador es idéntica a los datos originales comprimidos por el codificador. (Véase también: also Lossy Compression.)

Lossy Compression (*compresión con pérdida*). Un método de compresión donde la salida del decodificador difiere de los datos originales comprimidos por el codificador, pero es, sin embargo, aceptable para un usuario. Tales métodos son comunes en la compresión de imágenes y de audio, pero no en la compresión de texto, donde incluso la pérdida de un carácter puede dar lugar a un texto erróneo, ambiguo o incomprensible. (Véase también: Lossless Compression, Subsampling.)

LPVQ. Es el acrónimo de Locally Optimal Partitioned Vector Quantization (cuantificación vectorial con particiones locales óptimas). LPVQ es un algoritmo de cuantificación propuesto por Giovanni Motta, Francesco Rizzo, y James Storer [Motta et al. 06] para la compresión sin pérdidas y casi sin pérdidas de datos hiperespectrales. Las firmas espectrales se dividen primero en subvectores de longitud desigual e independientemente cuantificada. Luego, los índices son codificados entrópicamente aprovechando tanto la correlación espectral como espacial. El error residual también se codifica entrópicamente, con las probabilidades condicionadas por los índices de cuantificación. La partición localmente óptima de las firmas espectrales se decide en tiempo de diseño, durante el entrenamiento del cuantificador.

Luminance (*Luminancia*). Esta cantidad se define por la CIE (Sección 4.8.1) como la energía radiante ponderada mediante una función de sensibilidad espectral que es característica de la visión. (Véase también: CIE.)

LZ Methods (*Métodos LZ*). Todos los métodos de compresión basados en diccionarios se apoyan en la obra de J. Ziv y A. Lempel, publicada en 1977 y 1978. Hoy, estos métodos reciben el nombre de LZ77

y LZ78, respectivamente. Sus ideas han sido una fuente de inspiración para muchos investigadores, que los han generalizado, mejorado, y combinado con los métodos RLE y los estadísticos para formar muchos de los métodos de compresión adaptativos más comunes, para texto, imágenes y audio. (Véase también: Block Matching, Dictionary-Based Compression, Sliding-Window Compression.)

LZAP. El método LZAP (Sección 3.14) es una variante de LZW basada en la siguiente idea: En lugar de la concatenación de las dos últimas frases y la ubicación del resultado en el diccionario, se colocan todos los prefijos de la concatenación en el diccionario. El sufijo AP procede de All Prefixes (todos los prefijos).

LZARI. Una mejora en LZSS, desarrollada en 1988 por Haruhiko Okumura (Véase también: LZSS.)

LZFG. Este es el nombre de varios métodos relacionados (Sección 3.9) que son híbridos de LZ77 y LZ78. Fueron desarrollados por Edward Fiala y Daniel Greene. Todos estos métodos están basados en el siguiente esquema: El codificador genera un archivo comprimido con tokens y literales (códigos ASCII puros) entremezclados. Hay dos tipos de tokens: *literal* y *copia*. Un token *literal* indica que inmediatamente a continuación se encuentra una cadena de literales, un token copia apunta a una cadena vista previamente en los datos. (Véase también: Methods, Patents, Token.)

LZMA. LZMA (Lempel-Ziv-Markov chain-Algorithm (algoritmo Lempel-Ziv-cadena de Markov) es una de las muchas variantes de LZ77. Desarrollado por Igor Pavlov, este algoritmo, que se utiliza en su popular software `7z`, se basa en un buffer de búsqueda grande, una función hash (de dispersión) que genera índices, algo similar a LZRW4, y dos métodos de búsqueda. El método más rápido usa un hash-array (arreglo de dispersión) de listas de índices y el método normal utiliza un hash-array de árboles de decisión binarios. (Véase también: 7-Zip.)

LZMW. Una variante de LZW. El método LZMW (Sección 3.13) funciona como sigue: En lugar de añadir I más un carácter de la siguiente frase al diccionario, añade I más la siguiente frase entera más próxima, al diccionario. (Véase también: LZW.)

LZP. Una variante de LZ77 desarrollada por C. Bloom (Sección 3.16). Se basa en el principio de predicción contexto que dice: "si una cadena determinada `abcde` ha aparecido en el flujo datos de entrada en el pasado e iba seguida por `fg`..., entonces cuando `abcde` aparezca de nuevo en el flujo de entrada, hay una buena posibilidad de que vaya seguido por el mismo `fg`...". (Véase también: Context.)

LZSS. Esta versión de LZ77 (Sección 3.4) fue desarrollada por Storer y Szymanski en 1982 (véase [Storer y Szymanski 82]). Mejora el LZ77 base de tres maneras: (1) mantiene el buffer de búsqueda anticipada en una cola circular, (2) implementa el buffer de búsqueda (el diccionario) en un árbol de búsqueda binaria, y (3) que crea tokens con dos campos en vez de tres. (Véase también: LZ Methods, LZARI.)

LZW. Esta es una variante popular (Sección 3.12) de LZ78, desarrollada por Terry Welch en 1984. Su característica principal es la eliminación del segundo campo de un token. Un token de LZW se compone un único puntero al diccionario. Como resultado, tal token siempre codifica una cadena de más de un símbolo. (Véase también: Patents.)

LZX. LZX es una variante de LZ77 para la compresión de archivos cabinet (Sección 3.7).

LZY. LZY (Sección 3.15) es una variante de LZW que añade una cadena de diccionario por cada carácter de entrada e incrementa las cadenas un carácter cada vez.

MLP. Un método de compresión progresiva para imágenes en escala de grises. Una imagen se comprime en niveles. Un píxel se predice mediante un patrón simétrico y sus vecinos de los niveles precedentes, y el error de predicción se codifica aritméticamente. Se utiliza la distribución de Laplace para estimar la probabilidad del error. (Véase también: Laplace Distribution, Progressive FELICS.)

MLP Audio (*Audio MLP*). El nuevo estándar de compresión sin pérdidas aprobado para el DVD-A (audio) se denomina MLP. Es el tema de la Sección 7.7.

MNP5, MNP7. Éstos han sido desarrollados por Microcom, Inc., un fabricante de módems, para utilizarlos en sus módems. MNP5 (Sección 2.10) es un proceso de dos etapas que comienza con una codificación run-length, seguida por una codificación de frecuencia adaptativa. MNP7 (Sección 2.11) combina la codificación run-length con una variante bidimensional del método de Huffman adaptativo.

Model of Compression (*Modelo de compresión*). Un modelo es un método para "predecir" (asignar probabilidades a) los datos que se van a comprimir. Este concepto es importante en la compresión de datos estadísticos. Cuando se utiliza un método estadístico, debe construirse un modelo para los datos antes de que la compresión pueda comenzar. Se puede construir un modelo simple mediante la lectura de la cadena de entrada completa, contando el número de veces que aparece cada símbolo (su frecuencia de ocurrencia), y calculando la probabilidad de aparición de cada símbolo. Entonces, la cadena de datos es introducida de nuevo, símbolo a símbolo, y se comprime utilizando la información del modelo de probabilidad. (Véase también: Statistical Methods, Statistical Model.)

Una característica de la codificación aritmética es que es fácil de separar el modelo estadístico (la tabla con las frecuencias y las probabilidades) de las operaciones de codificación y decodificación. Es fácil codificar, por ejemplo, la primera mitad de una cadena de datos utilizando un modelo, y la segunda mitad usando otro modelo.

Monkey's audio (*Audio del mono*). Audio Monkey es un rápido, eficiente y libre algoritmo de compresión de audio sin pérdidas e implementación que ofrece detección de errores, etiquetado y soporte externo.

Move-to-Front Coding (*Codificación mover–al–frente*). La idea básica tras este método (Sección 1.5) es mantener el alfabeto de símbolos A como una lista, donde los símbolos que aparecen frecuentemente se encuentren cerca del frente. Un símbolo `s` se codifica como el número de símbolos que le preceden en dicha lista. Después de haber codificado el símbolo `s`, éste se mueve al frente la lista A.

MPEG. Es el acrónimo de Moving Pictures Experts Group (grupo de expertos sobre imágenes en movimiento). El estándar MPEG consta de varios métodos para la compresión de vídeo, incluyendo la compresión de imágenes digitales y sonido digital, así como la sincronización de ambas. Actualmente hay varios estándares MPEG. MPEG-1 está diseñado para velocidades de datos intermedios, del orden de 1,5 Mbits/seg. MPEG-2 está diseñado para altas velocidades de datos de al menos 10 Mbits/seg. MPEG-3 fue diseñado para la compresión de la HDTV (televisión de alta definición), pero se vio que era redundante y se fusionó con MPEG-2. MPEG-4 está diseñado para velocidades de datos muy bajas de menos de 64 Kbits/seg. La ITU-T, ha estado involucrada en el diseño de ambos formatos: MPEG-2 y MPEG-4. Un grupo de trabajo de la ISO se encuentra todavía trabajando en MPEG. (Véase también: ISO, JPEG.)

Multiresolution Decomposition (*Descomposición multirresolución*). Este método agrupa todos los coeficientes de la transformada wavelet discreta para una escala determinada, muestra su superposición, y repite el proceso para todas las escalas. (Véase también: Continuous Wavelet Transform, Discrete Wavelet Transform.)

Multiresolution Image (*Imagen multirresolución*). Una imagen comprimida que puede ser descomprimida en cualquier resolución. (Véase también: Resolution Independent Compression, Iterated Function Systems, WFA.)

Normal Distribution (*Distribución normal*). Una distribución de probabilidad con la bien conocida forma de campana (*bell curve*). Se encuentra en muchos lugares, tanto en modelos teóricos como

en situaciones de la vida real. La distribución normal con media m y desviación estándar s se define mediante:

$$f(x) = \frac{1}{s\sqrt{2\pi}} \exp\left\{-\frac{1}{2}\left(\frac{x-m}{s}\right)^2\right\}.$$

Patents (*Patentes*). Un algoritmo matemático puede ser patentado si está íntimamente asociado con su implementación software o firmware. Varios métodos de compresión, más especialmente LZW, han sido patentados (Sección 3.30), creando dificultades para los desarrolladores de software que trabajan con GIF, `compress` de UNIX, o cualquier otro sistema que utilice LZW. (Véase también: GIF, LZW, Compress.)

Pel. La unidad más pequeña de una imagen de fax; un punto. (Véase también: Pixel.)

Phrase (*Frase*). Un fragmento de datos situado en un diccionario para ser utilizado en la compresión de datos en el futuro. El concepto de frase es fundamental en los métodos basados en diccionarios en la compresión de datos, ya que el éxito de tales métodos depende mucho de la forma en que seleccionan las frases que guardan en sus diccionarios. (Véase también: Dictionary-Based Compression, LZ Methods.)

Pixel (*Píxel*). La unidad más pequeña de una imagen digital; un punto. (Véase también: Pel.)

PKZip. Un programa de compresión para MS-DOS (Sección 3.22), escrito por Phil Katz, quien fundó la empresa PKWare que también comercializa el software PKunzip, PKlite, y PKARC (`http://www.pkware.com`).

Portable Network Graphics (PNG) (*gráficos de red portátiles*). Un formato de archivo de imagen (Sección 3.25), que incluye compresión sin pérdidas con *Deflate* y predicción de píxeles. PNG es libre y soporta varios tipos de imágenes y números de planos de bits, así como transparencias sofisticadas.

Portable Document Format (PDF) (*Formato de Documento Portátil*). Un estándar desarrollado por Adobe en 1991−1992 que permite crear, editar, transferir entre distintas plataformas informáticas, e imprimir, documentos arbitrarios. PDF comprime los datos en el documento (texto e imágenes) mediante LZW, Flate (una variante de Deflate), codificación run-length, JPEG, JBIG2, y JPEG 2000.

PPM. Un método de compresión que asigna probabilidades a los símbolos en función del contexto (largo o corto) en el que aparecen. (Véase también: Prediction, PPPM.)

PPPM. Un método de compresión sin pérdidas para imágenes en escala de grises (y color) que asigna probabilidades a los símbolos basadas en la distribución de Laplace, como MLP. Los distintos contextos de un píxel son examinados y sus estadísticas utilizadas para seleccionar la media y la varianza para una distribución de Laplace en particular. (Véase también: Laplace Distribution, Prediction, PPM, MLP.)

Prediction (*Predicción*). Asignación de probabilidades a los símbolos. (Véase también: PPM.)

Prefix Compression (*Compresión prefija o mediante prefijos*). Una variante de los quadtrees, diseñada para imágenes binivel con texto o diagramas, donde el número de píxeles negros es relativamente pequeño. A cada píxel de una imagen de $2^n \times 2^n$ se le asigna un número de n-dígitos, o $2n$ bits, basado en el concepto de quadtrees. Los números de píxeles adyacentes tienden a tener el mismo prefijo (los bits más significativos), por lo que el prefijo común y los distintos sufijos de un grupo de píxeles se comprimen por separado. (Véase también: Quadtrees.)

Prefix Property (*Propiedad prefija*). Uno de los principios de los códigos de tamaño variable. Afirma los siguiente: una vez que un determinado un patrón de bits ha sido asignado como código

de un símbolo, ningún otro código debe comenzar con ese patrón (el patrón no puede ser *prefijo* de cualquier otro código). Una vez que la cadena 1, por ejemplo, es asignada como código de a_1, ningún otro código debe comenzar por 1 (i.e., todos ellos tienen que empezar por 0). Una vez que 01, por ejemplo, es asignado como código de a_2, ningún otro código puede empezar por 01 (todos ellos deben comenzar por 00). (Véase también: Variable-Size Codes, Statistical Methods.)

Progressive FELICS (*FELICS progresivo*). Una versión progresiva de FELICS donde los píxeles son codificados en niveles. Cada nivel duplica el número de píxeles codificados. Para decidir qué píxeles están incluidos en un cierto nivel, el nivel precedente puede rotarse conceptualmente 45º y escalarse en un factor $\sqrt{2}$ en ambas dimensiones. (Véase también: FELICS, MLP, Progressive Image Compression.)

Progressive Image Compression (*Compresión de imágenes progresiva*). Un método de compresión de imágenes en el que el *stream* comprimido consta de "capas", donde cada capa contiene más detalles de la imagen. El decodificador puede mostrar la imagen completa rápidamente en un formato de baja calidad, y luego mejorar la calidad de la misma a medida que se leen y se descomprimen más y más capas. Un usuario que observa el proceso de descompresión de una imagen en la pantalla normalmente puede reconocer la mayoría de las características de la imagen después de haberse descomprimido sólo un 5−10 % de la imagen. La mejora de la calidad de la imagen con el tiempo se puede efectuar mediante (1) agudizamiento (*sharpening*), (2) adición de colores, o (3) aumento de resolución. (Véase también: Progressive FELICS, Hierarchical Progressive Image Compression, MLP, JBIG.)

Psychoacoustic Model (*Modelo psicoacústico*). Un modelo matemático de las propiedades de enmascaramiento del sonido del sistema de audición humana (percepción del sonido por el cerebro).

QIC-122 Compression (*Compresión QIC-122*). Una variante de LZ77 que ha sido desarrollada por la organización QIC para la compresión de texto en unidades de cartuchos de cinta de datos de 1/4 de pulgada.

QM Coder (*Codificador QM*). Este es el codificador aritmético de JPEG y JBIG. Se ha diseñado para ser sencillo y veloz, por lo que se limita a símbolos de entrada que son simples bits individuales y emplea una aproximación en lugar de la multiplicación exacta. También utiliza la aritmética de enteros de precisión fija, por lo que tiene que recurrir a la *renormalización* del intervalo de probabilidad cada cierto tiempo, a fin de que la aproximación permanezca cercana a la multiplicación verdadera. (Véase también: Arithmetic Coding.)

Quadrisection (*Cuatrisección*). Este pertenece a la familia del método quadtree. Supone que la imagen original es una matriz M_0 cuadrada de $2^k \times 2^k$, y construye las matrices $M_1, M_2, \ldots, M_{k+1}$ con cada vez menos columnas. Estas matrices tienen, naturalmente, cada vez más filas, y quadrisection logra la compresión buscando y eliminando filas duplicadas. Dos variantes estrechamente relacionadas con quadrisection son: bisection y octasection. (Véase también: Quadtrees.)

Quadtrees (*Árboles de 4 ramas*). Este es un método de compresión de datos para imágenes de mapas de bits (bitmaps). Un quadtree (Sección 4.30) es un árbol donde cada hoja corresponde a una parte uniforme de la imagen (un cuadrante, un subcuadrante, o un solo píxel) y cada nodo interior tiene exactamente cuatro hijos. (Véase también: Bintrees, Prefix Compression, Quadrisection.)

Quaternary (*Cuaternario*). Un dígito en base 4. Puede ser 0, 1, 2, ó 3.

RAR. RAR es una variante de LZ77 diseñada y desarrollada por Eugene Roshal. RAR es muy popular entre los usuarios de Windows y está disponible para varias plataformas. Además de una compresión excelente y una buena velocidad de codificación, RAR ofrece opciones tales como códigos correctores de errores y de cifrado. (Véase también: Rarissimo.)

Rarissimo. Una utilidad de archivo que se utiliza siempre conjuntamente con RAR. Está diseñado para comprobar periódicamente ciertas carpetas de origen, comprimir y descomprimir automáticamente

los archivos que allí se encuentren, y luego mover esos archivos a las carpetas de destino designadas. (Véase también: RAR.)

Raster. Una palabra se utiliza sin traducir muchas veces en este libro, para definir un orden determinado para realizar una exploración de datos (busqueda, escaneo, etc.). Dicho orden es un barrido en zigzag, normalmente de arriba abajo y de izquierda a derecha.

Recursive range reduction (3R) (*Reducción recursiva del rango*). La reducción recursiva del rango (3R) es un sencillo algoritmo de codificación que ofrece una compresión decente, es fácil de programar, y su rendimiento es independiente de la cantidad de datos a comprimir.

Relative Encoding (*Codificación relativa*). Una variante de RLE, llamada a veces *diferenciación* (Sección 1.3.1). Se utiliza en aquellos casos en que los datos a ser comprimidos están formados por una cadena de números que no difieren en mucho, o bien por cadenas que son similares entre sí. El principio de codificación relativa consiste en enviar el primer ítem de datos a_1, seguido por las diferencias $a_{i+1} - a_i$. (Véase también: DPCM, RLE.)

Reliability (*Fiabilidad*). Los códigos de tamaño variable y otros códigos son vulnerables errores. En aquellos casos en que la fiabilidad del almacenamiento y la transmisión de códigos son importantes, los códigos se pueden hacer más fiables mediante la adición de bits de verificación, bits de paridad o CRC (Sección 2.12). Observe que la fiabilidad es, en cierto sentido, lo opuesto a la compresión de datos, ya que se consigue aumentando la redundancia. (Véase también: CRC.)

Resolution Independent Compression (*Compresión independiente de la resolución*). Un método de compresión de imágenes que no depende de la resolución específica de la imagen que está siendo comprimida. La imagen puede ser descomprimida en cualquier resolución. (Véase también: Multiresolution Images, Iterated Function Systems, WFA.)

Rice Codes (*Códigos de Rice*). Un caso especial del código de Golomb. (Véase también: Golomb Codes.)

RLE. Un nombre genérico para los métodos que comprimen los datos mediante la sustitución de una racha de símbolos idénticos (*run*) con un código, o token, que contiene el símbolo y la longitud de la racha (*run length*). RLE sirve a veces como un paso en un método estadístico de varios pasos o uno basado en diccionarios. (Véase también: Relative Encoding, Conditional Image RLE, Run Length.)

Run Length (*Racha*). Un *run* indica una secuencia o racha de elementos adyacentes idénticos. Un *run length* es la longitud de la misma. En este libro se encontrará muchas veces sin traducir por ser un término más corto y menos abstracto que sus equivalentes en español (que son una circunlocución del término en inglés).

Scalar Quantization (*Cuantificación escalar*). La definición del diccionario del término "cuantificación" es "restringir una cantidad variable a valores discretos en lugar de a un conjunto continuo de valores". Si los datos a ser comprimidos están en forma de grandes números, se utiliza la cuantificación para convertirlos a pequeñas cantidades. Ésto produce compresión (con pérdidas). Si los datos a ser comprimidos son analógicos (e.g., un voltaje que cambia con el tiempo), se utiliza la cuantificación para digitalizarlos en pequeñas cantidades. Este aspecto de la cuantificación es utilizado por varios métodos de compresión de audio. (Véase también: Vector Quantization.)

SCSU. Un algoritmo de compresión diseñado específicamente para la compresión de archivos de texto en Unicode (Sección 8.12).

SemiAdaptive Compression (*Compresión semiadaptativa*). Un método de compresión que utiliza un algoritmo de dos pasadas, donde en la primera pasada lee la secuencia de datos de entrada para recoger estadísticas sobre los datos que van a ser comprimidos, y en la segunda pasada realiza

la compresión real. Las estadísticas (modelo) se incluyen en el flujo de datos comprimidos. (Véase también: Adaptive Compression, Locally Adaptive Compression.)

Semistructured Text (*Texto semiestructurado*). Este tipo de texto se define como datos que son legibles por humanos y también son adecuados para el procesamiento por una máquina. Un ejemplo común es el HTML. El método sequitur de la Sección 8.10 funciona especialmente bien para dicho texto.

Shannon-Fano Coding (*Codificación de Shannon–Fano*). Un algoritmo para la búsqueda de un código de tamaño variable y longitud mínima, dadas las probabilidades de todos los símbolos en los datos (Sección 2.7). Este método fue reemplazado posteriormente por el método de Huffman. (Véase también: Statistical Methods, Huffman Coding.)

Shorten. Un sencillo algoritmo de compresión para datos en forma de onda en general y para el habla en particular (Sección 7.9). Shorten emplea la predicción lineal para calcular los residuos (de las muestras de audio) que codifica por medio de códigos de Rice. (Véase también: Rice codes.)

Simple Image (*Imagen simple*). Una imagen simple es aquella que utiliza una pequeña fracción de los posibles valores de escala de grises o colores disponibles. Un ejemplo común es una imagen binivel, donde cada píxel se representa con ocho bits. Esta imagen utiliza sólo dos colores de una paleta de 256 colores posibles. Otro ejemplo es una imagen en escala de grises escaneada a partir de una imagen binivel. La mayor parte de los píxeles serán negros o blancos, pero algunos píxeles pueden tener otras tonalidades de gris. Un *cartoon* (caricatura, dibujo animado o no, y similares) es también un ejemplo de imagen simple (especialmente un *cartoon* barato, donde sólo se usan unos pocos colores). Un *cartoon* típico consta de áreas uniformes, por lo que puede utilizar un pequeño número de colores de una paleta potencialmente grande. El método EIDAC de la Sección 4.13 está especialmente diseñado para las imágenes simples.

Sliding Window Compression (*Compresión mediante ventana deslizante*). El método LZ77 (Sección 3.3) utiliza parte de la secuencia de entrada vista anteriormente como diccionario. El codificador mantiene una ventana al *stream* de entrada, y desplaza los ítems introducidos en dicha ventana de derecha a izquierda a medida que las cadenas de símbolos van siendo codificadas. El método se basa, por tanto, en una *ventana deslizante*. (Véase también: LZ Methods.)

Space-Filling Curves (*Curvas de relleno de espacio*). Una curva de relleno de espacio (Sección 4.32) es una función $\mathbf{P}(t)$ que pasa a través de cada punto de una región determinada bidimensional, normalmente el cuadrado unidad, donde t varía de 0 a 1. Tales curvas se definen de forma recursiva y se utilizan en la compresión de imágenes.

Sparse Strings (*Cadenas dispersas*). Independientemente de lo que representen los datos de entrada —texto, binarios, imágenes, o cualquier otra cosa—, podemos pensar en el flujo de entrada como una cadena de bits. Si la mayoría de los bits son ceros, la cadena es *dispersa*. Las cadenas dispersas pueden ser comprimidas muy eficientemente por métodos especialmente diseñados (Sección 8.5).

SPIHT. Un método progresivo de codificación de imágenes que codifica eficientemente la imagen después de que ha sido transformada por algún filtro wavelet. SPIHT es embebido, progresivo, y tiene una opción intuitiva de compresión con pérdida. También es fácil de implementar, rápida, y produce excelentes resultados para todo tipo de imágenes. (Véase también: EZW, Progressive Image Compression, Embedded Coding, Discrete Wavelet Transform.)

Statistical Methods (*Métodos estadísticos*). Estos métodos (Capítulo 2) trabajan asignando códigos de tamaño variable a los símbolos en los datos, de manera que los códigos más cortos pertenecen a los símbolos o grupos de símbolos que aparecen con más frecuencia en los datos (tienen una mayor

probabilidad de ocurrencia). (Véase también: Variable-Size Codes, Prefix Property, Shannon-Fano Coding, Huffman Coding, y Arithmetic Coding.)

Statistical Model (*Modelo estadístico*). (Véase: Model of Compression.)

Stream (*Flujo, cadena o secuencia*). Indica un flujo continuo. En este libro dicho flujo suele ser de datos, en cuyo caso *stream* equivale a "flujo, cadena, serie o secuencia de datos"; aunque también puede ser un flujo de electrones, en cuyo caso equivale a "corriente".

String Compression (*Compresión de cadenas*). En general, los métodos de compresión basados en cadenas de símbolos puede ser más eficientes que los métodos que comprimen símbolos individuales (Sección 3.1).

Subsampling (*Submuestreo*). El submuestreo es, posiblemente, la forma más sencilla de comprimir una imagen. Un método de submuestreo es simplemente ignorar algunos píxeles. El codificador puede, por ejemplo, ignorar una fila de cada dos y una columna de cada dos, de la imagen, y escribir los píxeles restantes (que constituyen el 25 % de la imagen) en el *stream* comprimido. El decodificador introduce los datos comprimidos y utiliza cada píxel para generar cuatro píxeles idénticos de la imagen reconstruida. Ésto, por supuesto, implica la pérdida de muchos detalles de la imagen y rara vez es aceptable. (Véase también: Lossy Compression.)

Symbol (*Símbolo*). La unidad más pequeña de datos a ser comprimida. Un símbolo es a menudo un byte, pero también puede ser un bit, un trit $\{0, 1, 2\}$, o cualquier otra cosa. (Véase también: Alphabet.)

Symbol Ranking (*Clasificación de símbolos*). Un método basado en el contexto (Sección 8.2), donde el contexto C del símbolo actual S (los N símbolos que preceden a S) se utiliza para preparar una lista de símbolos que muy posiblemente sigan a C. La lista está ordenada de mayor a menor probabilidad. La posición de S en esta lista (la numeración de la posición comienza en 0) es entonces escrita por el codificador, después de haber sido adecuadamente codificada, en la secuencia de salida.

Taps (*Tomas*). Filtros de coeficientes wavelet. (Véase también: Continuous Wavelet Transform, Discrete Wavelet Transform.)

TAR. Un archivador estándar de UNIX. El nombre TAR es un acrónimo de Tape ARchive (archivo de cinta). Agrupa varios archivos en un fichero sin compresión. Después de haber sido comprimidos por el programa `compress` de UNIX, un archivo TAR recibe como nombre de extensión `tar.z`.

Textual Image Compression (*Compresión de imágenes de texto*). Un método de compresión para documentos impresos que contienen texto impreso o mecanografiado (pero no a mano). El texto puede estar en muchas fuentes y puede estar formado por cualesquier notas musicales, jeroglíficos, y símbolos. Utiliza técnicas de reconocimiento de patrones para reconocer los caracteres del texto que son idénticos o al menos similares. Mantiene una librería formada por una copia de cada grupo de caracteres idénticos. Cualquier material sobrante se considera un residuo. El método utiliza diferentes técnicas de compresión para los símbolos y para el residuo. Además incluye una opción de pérdida donde el residuo es ignorado.

Thumbnail (*Miniatura*). Este término se refiere a la miniatura de una imagen, que se utiliza normalmente para ofrecer una vista preliminar de la misma.

Time/frequency (T/F) codec (*Codificador tiempo/frecuencia*). Un codificador de audio que emplea un modelo psicoacústico para determinar cómo varía el umbral normal de audición (tanto en tiempo, como en frecuencia) en presencia de sonidos de enmascaramiento.

Token (*ficha, muestra*). Una unidad de datos que es escrita en el *stream* comprimido por algunos algoritmos de compresión. Un token se compone de varios campos que pueden tener tanto tamaños fijos como variables.

Transform (*Transformada*). Una imagen puede ser comprimida mediante la transformación de sus píxeles (que están correlacionados) en una representación donde estén *descorrelacionados*. La compresión se logra si los nuevos valores son más pequeños, en promedio, que los originales. La compresión con pérdida puede lograrse mediante la cuantificación de los valores transformados. El decodificador introduce los valores transformados del *stream* comprimido y reconstruye (de forma precisa o aproximada) los datos originales mediante la aplicación de la transformación opuesta. (Véase también: Discrete Cosine Transform, Fourier Transform, Continuous Wavelet Transform, Discrete Wavelet Transform.)

Triangle Mesh (*Malla de triángulo*). Superficies poligonales que son muy populares en los gráficos por ordenador. Tales superficies se componen de polígonos planos, en su mayoría triángulos, por lo que existe una necesidad de métodos especiales para comprimir una malla de triángulo. Uno de tales métodos es edgebreaker (rompedor de bordes) (Sección 8.11).

Trit. Un dígito ternario (en base 3). Puede ser 0, 1, ó 2.

Tunstall codes (*Códigos de Tunstall*). Los códigos de Tunstall son una variación de los códigos de tamaño variable. Son códigos de tamaño fijo, cada uno codificando una cadena de tamaño variable formada por símbolos.

Unary Code (*Código unario*). Una forma de generar códigos de tamaño variable de números enteros en un solo paso. El código unario de un entero no negativo n se define (Sección 2.3.1) como $n-1$ unos, seguidos por un solo 0 (Tabla 2.3). También hay un código unario general. (Véase también: Golomb Code.)

Unicode. Se ha propuesto un nuevo código internacional estándar —Unicode— y está siendo desarrollado por la organización internacional Unicode (`www.unicode.org`). Unicode usa códigos de 16 bits para sus caracteres, por lo que provee $2^{16} = 64\text{K} = 65\,536$ códigos. (Observe que al duplicar el tamaño del código, el número de códigos posibles es mucho mayor que el doble de los que había antes. De hecho, *eleva al cuadrado* el número de códigos.) Unicode incluye todos los códigos ASCII, además de los códigos de caracteres de lenguas extranjeras (incluyendo juegos completos de caracteres coreanos, japoneses, y chinos) y muchos símbolos matemáticos y de de otro tipo. Actualmente, se han asignado aproximadamente 39 000 de los 65 536 códigos posibles, por lo que hay espacio para añadir más símbolos en el futuro.

El sistema operativo Microsoft Windows NT ha adoptado Unicode, como lo han hecho también de AT&T Plan 9 y Lucent Inferno. (Véase también: ASCII, Codes.)

UNIX diff. Un algoritmo de diferenciación de archivos que utiliza `APPEND`, `DELETE` y `CHANGE` para codificar las diferencias entre dos archivos de texto. `diff` genera una salida legible para el usuario u, opcionalmente, puede generar comandos por lotes (un archivo *batch*) para un editor de texto como `ed`. (Véase también: BSdiff, Exediff, File differencing, VCDIFF, y Zdelta.)

V.42bis Protocol (*Protocolo V.42bis*). Este es un estándar publicado por la UIT-T (Sección 2.13) para su uso en módems rápidos. Se basa en el viejo protocolo V.32bis y se supone que se utiliza para altas velocidades de transmisión, de hasta 57,6K baudios. El estándar contiene especificaciones para la compresión de datos y la corrección de errores, pero sólo se discute la primera, en la Sección 3.21.

V.42bis especifica dos modos: un modo *transparente*, sin compresión, y un modo *comprimido* que emplea una variante de LZW. El primero se utiliza para flujos de datos que no se comprimen bien, y puede incluso causar expansión. Un buen ejemplo es un archivo ya comprimido. Este tipo de archivo se asemeja a datos aleatorios, no tiene ningún tipo de patrón repetitivo, y al tratar de comprimirlos con LZW llena el diccionario con frases cortas, de dos símbolos.

Variable-Size Codes (*Códigos de tamaño variable*). Éstos son utilizados por los métodos estadísticos. Tales códigos deben satisfacer la propiedad prefija (Sección 2.2) y deben ser asignados a

los símbolos en función de las probabilidades de aparición de éstos. (Véase también: Prefix Property, Statistical Methods.)

VCDIFF. Un método para la compresión de las diferencias entre dos ficheros. (Véase también: BSdiff, Exediff, File differencing, UNIX diff, y Zdelta.)

Vector Quantization (*Cuantificación vectorial*). Es una generalización del método de cuantificación escalar. Se utiliza tanto para la imagen como para la compresión de audio. En la práctica, la cuantificación vectorial es comúnmente usada para comprimir los datos que han sido digitalizados a partir de una fuente analógica, como las muestras de sonido y las imágenes escaneadas (dibujos o fotografías). Tales datos se denominan *datos analógicos muestreados digitalmente* (*digitally sampled analog data* o DSAD). (Véase también: Scalar Quantization.)

Video Compression (*Compresión de vídeo*). La compresión de vídeo se basa en dos principios. El primero es la redundancia espacial que existe en cada *frame* (cuadro) de video. El segundo es el hecho de que muy a menudo, un *frame* de video es muy similar a sus vecinos inmediatos. Ésto se denomina *redundancia temporal*. Una técnica típica para la compresión de vídeo debe comenzar, por tanto, con la codificación del primer *frame* utilizando un método de compresión de imágenes. A continuación, debe codificar cada *frame* sucesivo mediante la identificación de las diferencias entre el *frame* y su predecesor, y codificar estas diferencias.

Voronoi Diagrams (*Diagramas de Voronoi*). Imagínese un plato de petri listo para el crecimiento de bacterias. Se colocan allí simultáneamente cuatro tipos diferentes de bacterias en distintos puntos e inmediatamente comienzan a multiplicarse. Suponemos que sus colonias crecen a la misma velocidad. Inicialmente, cada colonia se compone de un círculo creciente alrededor de los puntos de partida. Tras cierto tiempo, algunos de ellos se encuentran y dejan de crecer en el área de reunión debido a la falta de alimento. El resultado final es que el plato entero se divide en cuatro zonas, una en torno a cada uno de los cuatro puntos de partida, de tal manera que todos los puntos dentro del área i están más cerca al punto de partida i que a cualquier otro punto de partida. Tales áreas se denominan *regiones de Voronoi* o *mosaico (teselaciones) de Dirichlet*.

WavPack. WavPack es un algoritmo de compresión de audio multiplataforma abierto y un software que soporta tres modos de compresión: con pérdidas, de alta calidad sin pérdidas, y un único modo híbrido. WavPack maneja muestras de audio de enteros de hasta 32 bits y también muestras de audio de 32 bits en punto flotante de IEEE. Emplea un codificador de entropía original que asigna códigos de Golomb de tamaño variable a los residuos y también tiene un modo de codificación de Golomb recursiva para aquellos casos en que la distribución de los residuos no es geométrica.

WFA. Este método utiliza el hecho de que la mayoría de las imágenes de interés tienen una cierta cantidad de auto-similitud (i.e., partes de la imagen son similares, hasta en tamaño o brillo, a la imagen completa o a otras partes). Divide la imagen en subcuadrados con un *quadtree*, y usa un algoritmo recursivo de inferencia para expresar las relaciones entre las partes de la imagen en un grafo. El grafo es similar a los utilizados para describir autómatas de estado finito. El método es con pérdidas, ya que las partes de una imagen real pueden ser muy similares a otras partes. WFA es un método muy eficiente para la compresión de imágenes en escala de grises y en color. (Véase también: GFA, Quadtrees, Resolution-Independent Compression.)

WSQ. Un eficaz método de compresión con pérdida desarrollado específicamente para la compresión de imágenes de huellas dactilares. El método implica una transformada wavelet de la imagen, seguida por la cuantificación escalar de los coeficientes wavelet, y por una codificación RLE y de Huffman de los resultados. (Véase también: Discrete Wavelet Transform.)

XMill. La Sección 3.26 es una breve descripción de XMill, un compresor de propósito especial para archivos XML.

Zdelta. Un algoritmo de diferenciación de archivos desarrollado por Dimitre Trendafilov, Nasir Memon y Torsten Suel. Zdelta adapta la librería compresión de `zlib` al problema de la compresión diferencial de archivos. Zdelta representa el archivo objetivo mediante la combinación de copias tanto de la referencia como del archivo objetivo ya comprimido. Utiliza un codificador de Huffman para comprimir aún más esta representación. (Véase también: BSdiff, Exediff, File differencing, UNIX diff, y VCDIFF.)

Zero-Probability Problem (*Problema de la probabilidad cero*). Cuando las muestras de los datos son leídas y analizadas con el fin de generar un modelo estadístico de los datos, ciertos contextos pueden no aparecer, dejando entradas con los contadores a cero y, por lo tanto, una probabilidad cero en la tabla de frecuencias. Cualquier método de compresión requiere que a tales entradas se les asignen probabilidades distintas de cero de alguna manera.

Zip. Software popular que implementa el algoritmo *Deflate* (Sección 3.23) que utiliza una variante de LZ77 combinada con una codificación de Huffman estática. Usa un diccionario deslizante de 32 Kb de longitud y un buffer de preanálisis de 258 bytes. Cuando una cadena no se encuentra en el diccionario, su primer símbolo se emite como un byte literal. (Véase también: Deflate, Gzip.)

La expresión del rostro de un hombre es normalmente
una ayuda a sus pensamientos, o un glosario de su discurso.

—Charles Dickens, *Vida y aventuras de Nicholas Nickleby (1839)*

Unirse a la comunidad sobre la compresión de datos

Las personas interesadas en un toque personal pueden unirse a la "DC community" y comunicarse personalmente con los investigadores y desarrolladores en este área en crecimiento asistiendo a las conferencias sobre compresión de datos (DCC[1]). Se llevan a cabo, en su mayoría, a finales de marzo, todos los años desde 1991, en Snowbird, Utah, USA, y duran tres días. La información detallada sobre la conferencia, incluyendo los organizadores y la ubicación geográfica, se puede encontrar en `http://www.cs.brandeis.edu/~dcc/`.

Además de las invitaciones a las presentaciones y sesiones técnicas, hay una sesión de carteles y "Midday Talks" ("Charlas de mediodía") sobre temas de interés actual.

La sesión de carteles es el evento central de la DCC. Cada ponente pone una descripción de los trabajos recientes (incluyendo textos, diagramas, fotografías y gráficos) en un póster de 4 pies de ancho por 3 metros de altura. A continuación, discuten el trabajo con cualquier persona interesada, en un ambiente de atmósfera relajada, con un refrigerio servido. El premio Capocelli se concede anualmente para el mejor documento estudiante-autor de la DCC. Ésto es en memoria de Renato M. Capocelli.

El comité del programa parece un quién es quién de la compresión de datos, pero las dos figuras centrales son James Andrew Storer y Martin Cohn, ambos de la Universidad de Brandeis, quienes presidirán la conferencia y el programa de la conferencia, respectivamente. (En el 2006, el presidente del comité de programas fue Michael W. Marcellin.)

Los debates de la conferencia han sido tradicionalmente editados por Storer y Cohn. Se publican en la IEEE Computer Society (`http://www.computer.org/`) y se distribuiyen antes de la conferencia, una característica atractiva. Una bibliografía completa (en formato bibTEX) de los artículos publicados en pasados DCCs se pueden encontrar en `http://liinwww.ira.uka.de/bibliography/Misc/dcc.html`.

`Hemos nacido para unirnos a nuestros semejantes,`
`y participar en comunidad con la raza humana.`
`—Cicero.`

[1] Data Compression Conference.

Notas sobre el índice alfabético

El índice alfabético (o analítico) está dirigido a aquéllos que ya han leído el libro y quieren localizar un ítem familiar, así como para aquéllos nuevos en el libro que están buscando un tema en particular. He incluido algunos términos que pueden con los que se puede encontrar un lector interesado en alguno de los temas tratados en el libro (incluso los temas que se acaban de mencionar, de pasada). Como resultado, incluso una rápida mirada por encima del índice le da al lector una idea de los términos y los temas incluidos en el libro. Tenga en cuenta que los elementos del índice "compresión de datos" y "compresión de imágenes" sólo tienen subelementos generales como "lógico", "sin pérdida", y "binivel". Ningún método de compresión específico está específicamente listado como subelemento.

He tratado de hacer que los elementos de índice lo más completos posible, incluyendo medios nombres y fechas. Los errores y omisiones comunicados a mi atención son bienvenidos. Ellos serán agregados a la lista de erratas y se incluirán en las próximas ediciones.

Este párrafo es del traductor de la obra. He procurado que la construcción de los índices en castellano se asemeje lo más posible a su original en inglés. Se pueden encontrar palabras en cursiva en algún lugar de la frase tras la referencia "*véase*..."; éstas indican el orden de búsqueda. Por ejemplo, en "RGC, *véase* código de Gray *reflejado*", la palabra *reflejado* es la primera que hay que localizar, seguida de "código de Gray". Por lo demás, no hay mucho más que decir; los términos de más a la derecha completan aquéllos de su izquierda.

Índice alfabético

.mp3, archivos de audio, XLVII, 791, 813, 1107
    y shorten, 754
2 pasadas (compresión de), 10, 94, 121, 226, 356, 431, 869, 878, 1161
3R, *véase* Reducción de rango
7 (como número de la suerte), 816
7z, XLI, XLII, 248–251, 253, 1157
7-Zip, XLI, 248–251, 253, 1143, 1157
8 1/2 (película), 152
90º (rotación), 518

A-law companding, 734–739, 750
AAC, *véase* codificación de audio *avanzada*
AAC-LD, *véase* codificación de audio *avanzada*
Abish, Walter (1931-), 148
Abousleman, Glen P., 939
AbS (compresión de voz), 753
AC, coeficientes (de una transformada), 294, 304, 307
ACB, XLIX, 146, 855–862, 1143
acumuladas, frecuencias, 127, 128, 132
ad hoc, compresión de texto, 21–24
Adair, Gilbert (1944-), 148
Adams, Douglas (1952-2001), 235, 331
adaptativa modulación, por codificación de pulsos diferencial (ADPCM), 740
adaptativa, codificación aritmética, 134
adaptativa, codificación de frecuencia, 101, 1158
adaptativa, codificación Golomb (Goladap), 74, 75
adaptativa, codificación Huffman, XLV, 10, 41, 94–100, 105, 228, 235, 463, 1143, 1156, 1158
    basada en palabras, 879
    y compresión de video, 668
adaptativa, compresión, 10, 17, 1143
adaptativa, compresión de imágenes Golomb, 440, 441
adaptativa, descomposición de paquetes wavelet (descomposición de imágenes wavelet), 597
adaptativa, modulación por codificación de pulsos diferencial (ADPCM), 452, 739, 741–743
ADC (conversor analógico-a-digital), 721
Addison, Joseph (asesor, 1672-1719), 3
adiestramiento (en compresión de datos), 35, 111, 147, 338, 397, 398, 403, 410, 445, 446, 491, 680, 862, 897
Adler, Mark (1959-), 236
Adobe acrobat, *véase* Formato de Documento Portátil
Adobe Acrobat Capture (software), 882
Adobe Inc. (y patente LZW), 264
ADPCM, *véase* modulación por codificación de pulsos diferencial *adaptativa*
ADPCM, compresión de audio, 739–741, 750
    IMA, 452, 741–743
AES, *véase* sociedad de ingeniería de *audio*
afines, transformaciones, 516–518
    atractor, 519, 522
Aldus Corp. (y patente LZW), 264
aleatorios, datos, 7, 81, 235, 1057, 1164
alfabeto (definición del), 1143
alfabeto, redundancia, 4
algorítmica, contenido de información, 57
algorítmico, codificador, 12
Alicia (en el país de las maravillas), LII, 351
Alphabetical Africa (libro), 148
ALS, *véase* codificación de *audio* sin pérdidas
Amer, Paul, LII
America Online (y patente LZW), 264
Amis, Martin (1949-), 109
analógico, vídeo, 653–659
analógico-a-digital, conversor, *véase* ADC
analógicos, datos, 43, 1161
Anderson, Karen,, 447
anomalías (en una imagen), 546
ANSI, 107
apple audio codec (nombre equivocado), 821
árbol
    binario de búsqueda, 183, 184, 186, 418, 885, 1157
    asimétrico, 183, 185

balanceado, 183, 185
de caminos múltiples, 207
estructura de datos, 1148
Huffman, 79, 80, 82, 84, 94–97, 1153
único, 240
altura del, 88, 89
Huffman (decodificación), 98
Huffman (desbordamiento), 98
Huffman (reconstrucción), 98
Huffman adaptativo, 94–97
logarítmico, 571
LZ78, 195
desbordamiento, 196
LZW, 207, 208, 210
orientación espacial, 618–620, 624, 625
recorrido, 79
ARC, 235, 1143
archivo (software de archivado), 235, 1144
archivos, diferenciación, XLII, XLV, 12, 846, 920–931, 1145, 1151, 1164, 1166
BSDiff, 930, 931
bspatch, 930, 931
exediff, 926–929
exepatch, 926–929
VCDIFF, 922–924, 1165
zdelta, 925, 926
aritmética, codificación, 51, 120–131, 375, 1144, 1154, 1158, 1163
adaptativa, 132, 133, 135, 277, 627, 900
basada en el contexto (CABAC), 706
codificador MQ, 640, 641, 646
codificador QM, 136–145, 167, 345, 356, 1160
en JPEG, 136–145, 344, 356, 1160
en JPEG 2000, 640, 641
principios de la, 121
y compresión de vídeo, 668
y mover-al-frente, 41
y sonido, 729
ARJ, 236, 1144
armónicos (en audio), 786, 787
array (estructura de datos), *véase* arreglo
arreglo (estructura de datos), 1144
ASCII, 38, 183, 261, 945–948, 1144, 1164
características, 945, 946
caracteres de control, 946–948
ASCII85 (codificación de binario a texto), 920
Ashland, Matt, XLII, 778
asimétrica, compresión, 11, 176, 182, 396, 456, 701, 868, 907
en audio, 717
FLAC, 761, 1151
Asimov, Isaac (1920-1992), LII, 448
asociativa, codificación de Buyanovsky, *véase* ACB
aspecto (relación de), 655–658, 661–663
de la HDTV, 642, 661–663
de la televisión, 642, 655
definición de la, 642, 656
ATC (compresión de voz), 750
atípica, imagen, 410
audición
propiedades de la, 726–729
rango de, 544
audio, codificación sin pérdidas (ALS), XLI, 779, 780, 782–789, 836, 1143
audio, compresión, XLVII, 12, 717–831, 833–843
A-law, 452, 734–739, 750
μ-law, 734–739, 750
ADPCM, 452, 739–741, 750
asimétrica, 717
audio (de) monkey, XLI, 778, 1158
companding, 730, 731
Dolby AC-3, 840–843, 1143
DPCM, 450
enmascaramiento de frecuencia, 726, 728, 792
enmascaramiento temporal, 726, 728, 792
FLAC, XLI, XLII, 759–769, 782, 1151
lossy vs. lossless debate, 780
LZ, 177, 1156
MLP, XLV, 744–747, 1158
MPEG-1, 12, 790–814, 816
MPEG-2, XLII, 815–840
silencios, 730
y diccionarios, 717
audio, digital, 721–726
audio, sociedad de ingeniería (AES), 791
áurea, razón, 65, 1105, 1151
Austen Jane (1775–1817), 105
autómata, *véase* máquina de estados *finitos*
autómata finito (AF), 993–995
determinista (AFD), 994
lenguaje del, 995
no determinista (AFND o NFA), 994
autómatas finitos ponderados, *véase* WFA
auto-similitud en imágenes, 279, 500, 507, 1151, 1165
avanzada, codificación de audio (AAC), XLII, 815–840, 1143
low delay (baja latencia), 836–838
y Apple Computer, XLII, 821
avanzada, codificación de video (AVC), *véase* H.264

avanzada, estándar de encriptación
claves de 128 bits, 232
claves de 256 bits, 249
avanzados, comité de sistemas de televisión, 661

background, píxel (blanco), 374, 380, 1144
Baeyer, Hans Christian von (1938), 55, 957, 1054
Baker, Brenda, 928
balanceado, árbol binario, 92, 132
bandpass (paso-banda), función, 537
baricéntricas, funciones, 437, 606
baricéntricos, pesos, 451
Bark (unidad de tasa de banda crítica), 728, 1144
Barkhausen, Heinrich Georg (1881–1956), 728, 1144
y bandas críticas, 728
Barnsley, Michael Fielding (1946–), 516
base, matriz, 437
bastones (en la retina), 348, 1007
Baudot, código, 22, 23, 287
Baudot, Jean Maurice Emile (1845–1903), 22, 287
Bayes, Thomas (1702–1761), 953, 954, 1147
Bayesiana, estadística, 143, 953, 1147
Beebe, Nelson F. H., LII
bel (antigua unidad logarítmica), 719
bell curve, *véase* distribución *Gaussiana*
Bell, Quentin (1910–1996), XLIII
Bellard, Fabrice, 260, 1150
Berbinzana, Manuel Lorenzo de Lizarazu y, 148
Bernstein, Dan, 217
BGMC, *véase* códigos de *bloque* de Gilbert-Moore
Biblia, índice de (concordancia), 867
bicúbica, interpolación, 438
bicúbica, superficie
representación algebraica de, 439
representación geométrica de, 439
bicúbico, polinomio, 439
big endian (orden de bits), 765
binaria, búsqueda, 53, 133, 135
árbol, 183, 184, 186, 418, 885, 1157
binario, árbol, 91
balanceado, 92, 132
codificación predictiva (BTPC), XLVIII, LII, 458–464
completo, 92, 132
BinHex, 1144
BinHex4, 37–39, 1052
binivel, imagen, 82, 269, 276, 287, 371, 374, 550, 1144, 1151, 1152, 1154
binivel, imagen, compresión (extendida a escala de grises), 279, 374, 453
binomial, distribución, 959, 960
bintrees, 460, 468–474, 511, 1144, 1146
progresivo, 468–474
biortogonal, filtros, 571
bisección, XLVIII, 486–488, 1160
bisection, *véase* bisección
bit budget, *véase bit* de coste
bit de coste (definición de), 13
bitmap, 30
bitplane, 1144
definición de, 270
separación de, 279, 375, 453
y código de Gray, 280
y correlación entre píxeles, 281
y redundancia, 279
bitrate (definición de), 13, 1144
bits/carácter (bpc), 13, 1145
bits/símbolo., 1145
bitstream (definición de), 9
Blelloch, Guy, LII, 81, 1056
blending (funciones de), 437
Bloom, Charles R., 1157
LZP, 220–227
PPMZ, 163
bloque, códigos de Gilbert-Moore (BGMC), XLII, 779, 789
bloque, descomposición, XLVIII, 278, 454–458, 629, 1145
bloque, diferenciación (en compresión de vídeo), 665
bloque, emparejamiento (compresión de imágenes), XLVIII, 60, 408–411, 1145
bloque, modo, 12, 846
bloques, artefactos en JPEG, 344, 638
bloques, clasificación de, 146, 846
bloques, codificación, 1145
bloques, codificacion por truncamiento, XLVIII, 412–417, 1145
BMP, compresión de archivos, XLIV, 38, 39, 1145
BNF (Formato de Backus Naur), 190, 898
BOCU-1 (Compresión unicode), XLV, 846, 918, 1145
Bohr, Niels David (1885–1962), 429
Boltzman, Ludwig Eduard (1844–1906), 1054
Boutell, Thomas (desarrollador de PNG), 254
bpb (bit por bit), 12
bpc (bits por carácter), 13, 1144
bpp (bits por píxel), 13
Braille, 19–21
Braille Louis (1809–1852), 19

Brandenburg, Karlheinz (desarrollador de mp3, 1954–), 839
break even point en LZ, 186
Brislawn, Christopher M., LII, 136, 633
Broukhis, Leonid A., 855
browsers (Web), *véase* nvegadores *Web*456
Bryant, David (WavPack), XLI, XLII, 769
BSDiff, 930, 931, 1145
bspatch, 930, 931, 1145
BTC, *véase* codificación por truncamiento de *bloques*
BTPC (codificación predictiva con árboles binarios), 458–463
Burmann, Gottlob (1737–1805), 148
Burroughs, Edgar Rice (1875–1950), 65
Burrows-Wheeler, método, XLIX, 12, 145, 248, 845–850, 862, 1145
  y ACB, 858
Buyanovsky, George (Georgii) Mechislavovich, 855, 862, 1143
BWT, *véase* método de Burrows-Wheeler

CABAC (codificación aritmética basada en el contexto), 706
cabinet (formato de soporte de Microsoft), 191
cadenas, 1163
  dispersas, 1162
cadenas, compresión de, 177, 178, 1163
Calgary Corpus, 14, 163, 165, 174, 338, 900
CALIC, 443–446, 1145
  y EIDAC, 395
calientes, colores, 348
canónicos, códigos de Huffman, 90–93, 241
Canterbury Corpus, 14, 338
Capocelli, premio, 1167
Capocelli, Renato Maria (1940–1992), 1167
Capon, modelo para imágenes binivel, 441
Carroll, Lewis (1832–1898), LII, 203, 351
Cartesiano, producto, 438
cartoon , *véase* imagen simple
cartoon-like, imagen, 270, 515
cascada (compresión en), 10
casos (reducción de), 21
Castaneda, Carlos (Cesar Arana, 1925–1998), 190
causales, filtros wavelet, 573
CAVLC (código de tamaño variable de contexto adaptativo), 356, 706, 707, 712
CCITT, 110, 262, 343, 1146, 1154, 1155
CDC, código de visualización, 22
celdas, codificación (compresión de imágenes), XLIX, 529–531, 1146
CELP (compresión de voz), 700, 754
cero (probabilidad), problema de la, 146, 150, 370, 419, 441, 680, 889, 1166
cero (redundancia), estimación (en CTW), 174
Chaitin, Gregory J. (1947–), 57
Chekhov, Anton (1860–1904), 329
chirp (señal de test), 550
Chomsky, Noam (1928–), 898
Christie Mallowan, Dama Agatha Mary Clarissa (Miller 1890–1976), 100
Cicero, Marcus Tullius (106–43) B.C., 1167
cíclica, código (o control) de redundancia, *véase* CRC
ciclos, 13
CIE, 347, 1146
  diagrama de color, 347
circular, cola, 182, 183, 225, 260, 1146
Clancy, Thomas Leo (1947–)„ 638
Clarke, Arthur Charles (1917–), 837
Clausius, Rudolph (1822–1888), 55
Cleary, John G., 146
clonación (en DMC), 891
Coalson, Josh, XLI, XLII, 759, 769
codec, 9, 1146
código, desbordamiento (en Huffman adaptativo), 98
códigos
  ASCII, 1146
  Baudot, 22, 287
  binario por etapas (*phased-in*), 96, 230
  CDC, 22
  correctores de errores, 981, 1150
    bits de paridad, 985
    bits de verificación, 984
    contexto, 981
    distancia de Hamming, 985–987
    escrutinio, 983, 984
    Hamming, 987–989
    polinomios generadores, 990, 991
    redundancia, 981
    SEC-DEC, 989, 990
  definición de, 1146
  EBCDIC, 1146
  Elias Gamma, 758
  Golomb, XLIV, 64, 67–75, 421, 423, 707, 757, 868, 1152
  Gray, 1152
  Hamming, 1152

inicio-paso-parada (*Start-step-stop*), 60
pod, XL, 758
prefijo, 37, 59–65, 188, 229, 276, 277, 414, 884
y compresión de vídeo, 668
y números de Fibonacci, 65, 1055
Rice, 46, 47, 64, 71, 167, 426, 757, 759, 764, 773, 937, 1143, 1151, 1161, 1162
sesgado, Elias Gamma, 758
subexponencial, 64, 426, 427, 757
tamaño variable, 58–65, 82, 94, 99, 101, 106, 107, 111, 120, 177, 228, 247, 276, 277, 413, 420, 1057, 1161, 1164
no ambiguos, 76, 1155
Tunstall, 66, 67, 1164
unario, 60–65, 197, 226, 1164
Unicode, 1146, 1164
coeficientes de filtro, 576, 577
Cohn, Martin, XLVI, 1119, 1167
y patente de LZ78, 265
cola (estructura de datos), 182, 183, 1146, 1148
collating sequence (secuencia de cotejo), 183
color
caliente, 348
edad del, 458
frío, 348
color, espacio, 347
color, imágenes (y compresión de escala de grises), 32, 286, 427, 444, 448, 453, 527, 528, 552, 562
Commission Internationale de l'Éclairage, *véase* CIE
compactación, 21
compacto, disco (y compresión de audio), 815
compacto, soporte (de una función), 553
compacto, soporte (definición de), 585
companding, 9, 730
ALS, 783, 784
compresión de audio, 730, 731
compansión, *véase* companding
complejidad (Kolmogorov-Chaitin), 57
complejos, métodos (disminución del rendimiento), 7, 198, 230, 371
completo, árbol binario, 92, 132
composición, valores para imágenes progresivas, 468–473, 1146
compresión, factor de, 13, 564, 1146
compresión, ganancia, 13, 1146
compresión, medidas del rendimiento, 12–14
compresión, ratio o razón de, 12, 1147
conocida con antelación, 21, 22, 44, 272, 289, 290, 397, 413, 628, 644, 665, 731, 734, 741, 791
en UNIX, 230
Compresión localmente adaptativa, 1156
compresor (definición de), 9
compresor COMPRESS, 1146
Compuserve, Servicios de Información de, 231, 264, 1152
computador, aritmética del, 344
concordancia, 867
condicional, imagen RLE, XLVII, 35–37, 1147
condicional, intercambio (en codificador QM), 141–145
condicional, probabilidad, 1147
Conferencia sobre compresión de datos, 1148
conos (en la retina), 348
contexto
a partir de otros planos de bits, 395
bipartito, 395
definición de, 1147
inter, XLVIII, 395
intra, XLVIII, 395
simétrico, 419, 429, 443, 446, 646
contexto (árbol de ponderación), XLVIII, 167, 169–174, 187, 1147
para imágenes, XLVIII, 279, 453, 935
contexto (basado en el), codificación aritmética, *véase* CABAC
contexto (basado en el), compresión de imágenes, 417–419, 443–447
contexto (libres de), gramáticas, XLIX, 846, 897, 1147
contexto adaptativo, códigos de tamaño variable, *véase* CAVLC
contexto, modelado, 146
adaptativo, 147
de orden N, 148
estático, 146
contextual, redundancia, 4
continua, transformada wavelet (CWT), XLVII, 349, 546–552, 1148
continuos (tonos), imagen, 270, 338, 458, 550, 597, 1148, 1152, 1155
convolución, 569, 1148
y bancos de filtros, 569
Cormack, Gordon V., 887
correlación, 1148
correlaciones, 276
entre cantidades, 275

entre estados, 890–892
entre frases, 266
entre píxeles, 82, 290, 419
entre palabras, 881
entre planos en datos hiperespectrales, 934
Costello, Adam M., 255
covarianza, 275, 301
y descorrelación, 301
CRC, 261–263, 1148
en audio MPEG, 796, 804, 812
en CELP, 754
en PNG, 254
crew (compresión de imágenes), 625
en JPEG 2000, 641
Crichton, Michael (1942–), 266
Crick, Francis Harry Compton (1916–2004), 155
cromaticidad (diagrama de), 347, 1015
crominancia, 562
CRT, XLVII, 653–658, 1148
color, 656
cruzada, correlación de puntos, 291
CS-ACELP (compresión de voz), 754
CTW, *véase contexto* (árbol de ponderación)
cuadrado integrables, funciones, 546
cuadrante, numeración (en un quadtree), 465, 501
cuantificación
codificación por truncamiento de bloques, XLVIII, 412–417, 1145
definición de, 43, 44
en H.261, 701
en JPEG, 350, 351
en MPEG, 679–682, 684
escalar, XLVII, 43–45, 272, 290, 634, 1161
en SPIHT, 617
midriser, 737
midtread, 737, 801
transformación de imágenes, 278, 290, 1164
vectorial, XLVIII, 290, 367, 396–402, 515, 1165
adaptativa, XLVIII, 403, 405–407
cuantificación, ruido, 450
cuaternario (numeración en base 4), 466, 1160
cuatrisección, XLVIII, 482–488, 1160
Culik, Karel II, XLIX, LII, 500, 501, 506
curvas
de relleno de espacio, XLVI, XLIX, LI, 488–500, 1162
Hilbert, 489–494
Peano, 495, 499, 500
Sierpiński, 489, 494
CWT, *véase* transformada wavelet continua
DAC (conversor digital-a-analógico), 722
datos, compresión
óptima, 12, 188
adaptativa, 10
anagrama de (en inglés), 220
asimétrica, 11, 176, 182, 396, 456, 701, 717, 868, 907
audio, XLVII, 717–843
clasificación de bloques, 846
como opuesta de fiabilidad, 19
compactación, 21
compactación (*packing*), 22
con pérdidas (*lossy*), 10, 1156
conferencia, LI, 1119, 1167
cuantificación vectorial, XLVIII, 290, 367
adaptativa, XLVIII
definición, 3
disminución del rendimiento, 7, 198, 230, 371, 622
dos pasadas, 10, 94, 121, 226, 356, 431, 869, 878, 1161
e irrelevancia, 271
físicos, 12
geométrica, 846
hiperespectral, 931–942
historia en Japón, 17, 236
imágenes, 269–531
irreversible, 21
lógica, 12, 175
ley general, 5, 266
métodos basados en diccionarios, 9, 145, 175–268, 273
métodos estadísticos, 9, 51–174, 176, 272, 273
métodos intuitivos, XLVIII, 19–24, 289, 290
medidas de rendimiento, 12–14
modelo, 17, 370, 1158
modo bloque, 12, 846
modo continuo (*streaming*), 12, 846
números pequeños, 41, 352, 356, 370, 448, 454, 458, 459, 849, 1111
no adaptativo, 10
patentes, XLVIII, 248, 264–266, 1159
progresiva, 374
quién es quién, 1167
razones para la, 3
referencias, 17
semiadaptativa, 10, 94, 1161
simétrica, 11, 176, 190, 345, 634
sin pérdidas (*lossless*), 10, 21, 1156

un caso especial de diferenciación de archivos, 12
unirse a la comunidad, 1119, 1167
universal, 12, 188
vídeo, XLVII, 664–714
y redundancia, 4, 271, 889
datos, estructuras, 17, 98, 99, 182, 195, 207, 1146, 1148
árbol, 1148
array, 1148
cola, 182, 183, 1148
grafo, 1148
lista, 1148
pila, 1148
tabla hash, 1148
DC, coeficiente (de una transformada), 294, 304, 307, 336, 345, 351–353, 356
DCC, *véase* datos, compresión, conferencia
DCT, *véase* transformada *discreta* del coseno
decibelio (dB), 288, 719, 829, 1148
decimación (en bancos de filtros), 570, 795
decodificador, 1148
definición de, 9
determinista, 12
definición de compresión de datos, 3
Deflación, *véase* Deflate
Deflate, XLV, 182, 236–250, 254, 255, 920, 1149, 1152, 1159, 1166
y RAR, XL, 233
descompresor (definición de), 9
descorrelacionados, píxeles, 275, 278, 290, 298, 301, 336
descorrelacionados, valores (y covarianza), 276
desplazamiento invariante, 1148
determinista, decodificador, 12
Deutsch, Peter, 242
DFT, *véase* transformada *discreta* de Fourier
dibujo
y cadenas dispersas, 867
y JBIG, 374
diccionario (en VQ adaptativo), 403
diccionarios, métodos basados en, 9, 145, 175–268, 273, 1149
comparados con la compresión de audio, 717
sin diccionario, 227–229
unificación con métodos estadísticos, 266–268
y sequitur, 902
y sonido, 729
Dickens, Charles (1812–1870), 418, 1166
DIET, 261
diferencia, valores para imágenes progresivas, 468–473, 1146
diferenciables, funciones, 585
diferenciación, 29, 213, 356, 453, 1161
de archivos, XLII, XLV, 846, 920–931, 1151
en BOCU, 918
en compresión de vídeo, 665
diferencial, codificación, 30, 448, 1150
diferencial, compresión de imágenes, 447, 448, 1149
diferencial, modulación de impulsos codificados, 448–452
adaptativa, 452
y sonido, 448, 1150
diff, 921, 1164
digital, audio, 721–726
digital, cámara, 348
digital, imagen, *véase* imagen
digital, silencio, 763
digital, televisión (DTV), 663
digital, vídeo, 660, 661, 1149
digital-a-analógico, conversor, *véase* DAC
digitalmente, datos analógicos muestreados, 396, 1165
digrama, 5, 29, 146, 147, 898, 1149
codificación, 29, 898
frecuencias, 106
y redundancia, 4
dinámica, codificación de Markov, XLVI, XLIX, 845, 887–895
dinámica, compresión, *véase* compresión *adaptativa*
dinámico, diccionario, 175
GIF, 231
dirección e-mail del autor, XLIII, XLVI, XLIX, 18
dirección e-mail del traductor, XXXIII
discreta, transfomada de Fourier, 349
en audio MPEG, 792
discreta, transformada del coseno, XLVIII, 298, 303–335, 344, 349, 350, 560, 707, 711, 1149
en H.261, 701
en MPEG, 678–684
modificada, 795
mp3, 809
tridimensional, 303, 937–939
y compresión de voz, 750
discreta, transformada del seno, XLVIII, 335–337
discreta, transformada wavelet (DWT), XLVII, 349, 578–590, 1149

discretos, imagen de tonos, 270, 338, 339, 454, 458, 515, 550, 1145, 1149
dispersas, cadenas, 22, 73, 845, 867–877
  compresión mediante prefijos, XLIX, 873–877
dispersión o raleza, proporción de, 13, 564
Disraeli (Beaconsfield), Benjamin (1804–1881), 28, 1194
distorsión, medidas de
  en compresión de vídeo, 667, 668
  en cuantificación vectorial, 397
distribuciones
  energía de, 294
  Gaussiana, 1151
  geométrica, 68, 363, 773
  Laplace, 277, 427, 429–432, 434, 443, 453, 724–726, 746, 757, 758, 764, 1156, 1157, 1159
  normal, 472, 1158
  plana, 729
  Poisson, 155
  sesgada, 74
dithering (interpolación o tramado), 381, 474
DjVu, compresión de documentos, XLIX, 629–632, 1149
DMC, *véase* codificación *dinámica* de Markov
Dobie, J. Frank (1888–1964), 1142
Dolby AC-3, XLII, 840–843, 1143
Dolby Digital, *véase* Dolby AC-3
Dolby, Ray (1933–), 840
Dolby, Thomas (Thomas Morgan Robertson, 1958–), 843
downsampling (disminución de resolución), 344, 379
Doyle, Arthur Conan (1859–1930), 172
Doyle, Mark, LII
DPCM, *véase* modulación de impulsos codificados *diferencial*
DSAD, *véase* datos analógicos muestreados *digitalmente*
DST, *véase* transformada *discreta* del seno
dualidad onda-partícula, 543
Dumpty, Humpty, 881
Durbin J., 769
DWT, *véase* transformada wavelet *discreta*
Dyson, George Bernard (1953–), 51, 887

Eastman, Willard L. (y patente LZ78), 265
EBCDIC, 183, 1146
EBCOT (en JPEG 2000), 640, 641
edad de piedra, binario (código unario), 60
Eddings, David (1931–), 43
edgebreaker, XLIX, 846, 902, 903, 905–913, 1164
Edison, Thomas Alva (1847–1931), 653, 655
EIDAC, compresión de imágenes simples, XLVIII, 395, 1162
Einstein, Albert (1879–1955) y $E = mc^2$, 25
Elias, código (WavPack), 774
Elias, código Gamma, 758
Elias, Peter (1923–2001), 60, 121
Eliot, Thomas Stearns (1888–1965), 942
Elton, John, 520
email, dirección del autor, XLIII, XLVI, XLIX, 18
email, dirección del traductor, XXXIII
embebida, codificación en compresión de imágenes, 614, 1150
embebida, codificación usando árboles-cero (EZW), XLVII, 625–629, 1151
encoder (codificador), 1150
  algorítmico, 12
  definición de, 9
energía
  concentración de, 294, 298, 301, 336
  de una distribución, 294
  de una función, 546, 547
enmascarado (Lempel-Ziv), utilidad (una variante de LZW), 790
entera, transformada wavelet, *véase* IWT
entrelazado, escaneo de líneas, 32, 662
entropía, 58, 76, 78, 81, 121, 131, 375, 1056
  de una imagen, 278, 459
  definición de, 54, 1150
  física, 58
entropía, codificador de, 575
  basado en diccionarios, 175
  definición de, 57, 1150
error, métricas en compresión de imágenes, 287, 288
errores, códigos correctores de, 1150
  en RAR, 232
errores, códigos detectores de, 261
ESARTUNILOC (probabilidad de letras), 5
escala de grises, imagen, 270, 277, 287, 1152, 1154, 1157, 1159
escala de grises, imagen, compresión (extendida a imágenes en color), 32, 286, 427, 444, 448, 453, 527, 528, 552, 562
escalado (scaling), 517
escalar, cuantificación, XLVII, 43–45, 272, 290, 1161
  en SPIHT, 617
  en WSQ, 634
escaneo de líneas, entrelazado, 662

escape (código)
en compresión basada en palabras, 878
en Huffman adaptativo, 95
en imágenes de texto, 885
en Macwrite, 24
en patrones de substitución, 29
en PPM, 151
en RLE, 25
espacial, orientación, árboles, 618–620, 624, 625
espacial, redundancia
en datos hiperespectrales, 274, 664, 933, 1165
Espacio de color, see JFIF1154
espectral, dimensión (en datos hiperespectrales), 933
espectral, selección (en JPEG), 345
estándar (descomposición de imágenes wavelet), 556, 592, 594
estándar, imágenes de test, 338, 339, 341
estándares (organizaciones para), 107, 108
estándares de televisión, 562, 654–659, 841
estático, diccionario, 175, 176, 196, 230
estadísticas, distribuciones, *véase* distribuciones
estadístico, modelo, 122, 145, 175, 375, 376, 1158
estadísticos, métodos, 9, 51–174, 176, 272, 273, 1162
modelado de contexto, 146
unificación con métodos de diccionario, 266–268
Estadísticas bayesianas, 1144
esteganografía (ocultación de datos), 188
estructura de árbol, cuantificación vectorial con, 402, 403
estructura de datos, 967
árbol, 970
AVL, 973
binario de búsqueda, 972
hijos, 970
hoja, 970
recorrido (orden), 971
recorrido (por niveles), 971
recorrido (post-orden), 971
recorrido (pre-orden), 971
array o arreglo, 967, 968
cola, 969
grafo, 974
hash (tabla), 974
lista enlazada, 969, 970
cola, 969
pila, 968, 969
ETAOINSHRDLU (probabilidad de letras), 5
Euler, ecuación de, 907
exclusión
en ACB, 861
en clasificación de símbolos, 852
en PPM, 154
EXE, compresión, 260
EXE, compresores, 260, 261, 926–929, 1150
exediff, 926–929, 1150
exepatch, 926–929, 1151
extrusión (shearing), 516
EZW, *véase* codificación embebida usando árboles-cero

física, compresión, 12
FABD, *véase* descomposición de *bloque*
Fabre, Jean Henri (1823–1915), 338
facsímile, compresión, 110, 112–120, 272, 276, 388, 882, 1151
1D, 110
2D, 115
grupo 3, 110
factor de compresión, 13, 564, 1146
Fano, Robert Mario (1917–), 77
Fantastic Voyage (novela), LII
FBI, compresión estándar de huellas dactilares, XLVII, 632–638, 1165
FELICS, 420–422, 1151
progresivo, 423–428, 1160
Feynman, Richard Philips (1918–1988), 723
FHM, compresión, XLIX, 845, 895–897, 1151
Fiabilidad, 1161
fiabilidad, 107
como opuesta a la compresión, 19
en RAR, 232
y códigos de Huffman, 107
Fiala, Edward R., 7, 196, 265, 1157
Fibonacci, números, 1151
y altura de los árboles de Huffman, 89
y bases numéricas, 65
y códigos prefijo, 65, 1055
y cadenas dispersas, 872, 873
y compresión FHM, XLIX, 895–897
Fichero, *véase* archivo
fijo, códigos de tamaño, 5
filtros
causales, 573
derivación de coeficientes, 576, 577
impulsos de respuesta finita, 570, 578
taps o derivaciones, 573
filtros espejo de cuadratura, *véase* QMF

filtros, bancos de, 569–577
biortogonal, 571
decimación, 570, 795
derivación de coeficientes de filtro, 576, 577
ortogonales, 571
finita (respuesta), filtros de impulsos (FIR), 570, 578
finito, autómata, XLVI, XLIX, LI, LII
finito, autómata, métodos, 500–515
finitos (máquina de estados), XLVI, XLIX, LI, 82, 500, 846
y compresión, 440, 887, 888
Fisher, Yuval, 516
física, entropía, 58
FLAC, *véase* compresión de audio sin pérdidas *libre*
Flate (una variante de Deflate), 920
flujo, see cadena1163
Forbes, Jonathan (LZX), 191
Ford, Paul, 880
foreground, píxel (negro), 374, 380, 1144
forma de onda, codificadores de, 750
Formato de Documento Portátil, *véase* PDF
Fourier, Jean Baptiste Joseph (1768–1830), 533, 536, 1151
Fourier, serie, 537
Fourier, transformada, LI, 290, 306, 349, 534–545, 560, 572, 574, 1149, 1151, 1164
dominio de la frecuencia, 534–542
y compresión de imágenes, 544, 545
y principio de incertidumbre, 542, 543
fríos, colores, 348
fractal, compresión de imágenes, 515–529, 1154
fractales (como funciones no diferenciables), 590
Francés (frecuencias de letras), 5
Frank, Amalie J., 371
frase, 1159
en LZW, 203
frecuencia de eof, 125
frecuencia fundamental (de una función), 535
frecuencia, dominio, 534–542, 728
frecuencia, enmascaramiento, 726, 728, 792
frecuencias
acumuladas, 127, 128, 132
de digramas, 106
de píxeles, 295, 375
de símbolos, 17, 59, 82, 94–96, 99, 101, 102, 121, 122, 375, 1158
en LZ77, 182
Freed, Robert A., 235
fuente sin memoria, 44, 168, 169, 171, 172
definición de, 4
fuente, codificadores de voz de, 748, 750–753
funciones
baricéntricas, 437, 606
cuadrado integrables, 546
de blending (mezcla o fusión), 437
diferenciables, 585
dominio de la frecuencia, 534–542
en ninguna parte diferenciables, 585
energía de las, 546, 547
hash, 975–977
tratamiento de colisiones, 977, 978
impares, 335
no diferenciables, 590
pares, 335
paridad, 335
paso-banda, 537
representación de (una aplicación de las wavelets), 580–584
soporte compacto, 553, 585
soporte de, 553

Gadsby (libro), 147
Gailly, Jean-Loup, 236
ganancia en compresión, 13, 1146
gasket (Sierpiński), 524
Gaussiana, distribución, 956–958, 1151, 1156, 1158
generadores, polinomios, 1150
CRC, 263, 798
CRC-32, 263
generalizado, autómata finito, 512–515, 1151
geométrica, compresión, XLIX, 846
geométrica, distribución, 363, 773
en probabilidad, 68
geometría de crecimiento, codificación, 371–373, 1152
GFA, *véase* autómata finito *generalizado*
GIF, 231, 1152
comparado con FABD, 454
y compresión de imágenes, 274
y DjVu, 630
y navegadores web, 231
y patente LZW, 264–266, 1159
Gilbert, Jeffrey M., LII, 454, 458
Givens, J. Wallace (1910–1993), 330
Givens, rotaciones, 319, 322–325, 327–330
Glenn, Ford (Gwyllyn Samuel Newton 1916–), 76
Goladap (codificación Golomb adaptativa), 74, 75
golden ratio, *véase* razón *áurea*

Golomb, código, XLIV, 64, 67–75, 421, 423, 757, 868, 1151, 1152, 1161
adaptativo, 74, 75
Goladap, 74, 75
H.264, 707, 712
WavPack, 773
y JPEG-LS, 361, 363, 364, 1155
Golomb, Solomon Wolf (1932–), 76
Gouraud, sombreado (para superficies poligonales), 902
grabados, 442
distribución de píxeles inusual, 442
gráfica, imagen, 270
gráfica, interfaz de usuario, *véase* GUI
grafos, (estructuras de datos), 1148
Graham, Heather, 730
gramáticas libres de contexto, XLIX, 846, 897, 1147
Gray, código, *véase* código de Gray *reflejado*
Gray, Frank, 287
Greene, Daniel H., 7, 196, 265, 1157
grupo 3, compresión de faxes, 110, 1151
PDF, 920
grupo 4, compresión de faxes, 110, 1151
PDF, 920
GUI, 271
Guidon, Yann, XL, XLII, 46, 49
Gulliver, los viajes de (libro), LII
Gzip, 230, 236, 264, 1149, 1152

H.261, compresión de vídeo, XLVII, 700, 701, 1152
DCT en, 701
H.264, compresión de vídeo, XLI, 356, 702–714, 1152
Haar, transformada, XLVII, XLVIII, 298, 300, 332, 552–556
Hafner, Ullrich, 511
Hagen, Hans, LII
halftoning, *véase* semitonos
Hamming, códigos, 1152
Hardy, Thomas (1840–1928), 135
Haro, Fernando Jacinto de Zurita y, 148
hashing, XLVI, XLIX, 207, 209, 220, 411, 418, 457, 885, 1148
HDTV, 642, 841
relación de aspecto de la, 642, 661–663
resolución de la, 661–663
stándares usados en, 1153
y MPEG-3, 675, 816, 1158
heap, *véase* pila
Heisenberg Werner (1901–1976), 542
Heisenberg, principio de incertidumbre, 542
hercio (Hz), 534, 718
Herd, Bernd (LZH), 180
hermanos, propiedad de los, 96
Herrera, Alonso Alcalá y (1599–1682), 148
híbridos, codificadores de voz, 748, 753, 754
Hilbert, curva, 467, 489–493
recorrido de la, 495, 498
y cuantificación vectorial, 490–492
Hilbert, David (1862–1943), 493
hiperespectrales, compresión de datos, 846, 931–942
hiperespectrales, datos, 931, 1153
hiperhilo, 249
historia de la compresión de datos en Japón, 17, 236
homeomorfismo, 903
homogéneas, coordenadas, 518
Horspool, R. Nigel, 887
Hotelling, transformada, *véase* transformada de *Karhunen-Loève*
Householder, Alston Scott (1904–1993), 330
HTML (como texto semiestructurado), 902, 1162
huellas dactilares, compresión, XLVII, 590, 632, 634–638, 1165
Huffman, codificación, XLIX, 51, 59, 79–84, 111–115, 120, 147, 177, 260, 272, 848, 1066, 1149, 1152, 1153, 1163, 1166
adaptativa, 10, 41, 94–100, 105, 228, 235, 463, 1143, 1156
alfabeto de dos símbolos, 82
árbol único, 241
canónicos, 90–93, 241
en compresión de imágenes, 447
en JPEG, 345, 351, 638
en LZP, 226
en WSQ, 634
número de códigos, 86, 87
no único, 79
para imágenes, 82
semiadaptativo, 94
tamaño del código, 84, 85
ternario, 87
varianza, 80
y cadenas dispersas, 870, 871
y Deflate, 237
y FABD, 458
y fiabilidad, 107
y mover-al-frente, 41

y MPEG, 680
y sonido, 729
y wavelets, 554, 562
Huffman, David A. (1925–1999), 79
humana, audición, 544, 726–729
humana, voz (rango de la), 726
humano, sistema visual, 287, 413, 415
Humpty Dumpty, *véase* Dumpty, Humpty
hyperthreading, *véase* hiperhilo

ICE, 236
IDCT, *véase* transformada discreta del coseno *inversa*
IEC, 108, 674
IFS, compresión, 515–529, 1154
PIFS, 525
IGS, *véase* improved grayscale quantization
IID, *véase fuente* sin memoria
IMA, *véase* asociación multimedia *interactiva*
IMA, compresión ADPCM, 741–743
imagen, 269
atípica, 397
binivel, 82, 269, 287, 371, 374, 550, 1144, 1151, 1152, 1154
bitplane (plano de bits), 1144
cartoon-like, 270, 515
definición de, 269
entrelazada, 32
escala de grises, 270, 277, 287, 1152, 1154, 1157, 1159
gráfica, 270
reconstrucción, 544
resolución de la, 269
simple, 394, 395, 1162
sintética, 270
tipos de, 269, 270
tonos continuos, 270, 338, 458, 550, 597, 1148, 1152, 1155
tonos discretos, 270, 338, 339, 454, 458, 515, 550, 1145, 1149
imagen, compresión, 8, 10, 28, 35, 269–531
auto-similitud, 279, 1151, 1165
binivel, 374
binivel (extendido a escala de grises), 279, 374, 453
con pérdidas (*lossy*), 271
cuantificación vectorial, XLVIII
adaptativa, XLVIII
diferencial, 1149
IFS, XLVI, XLIX
métodos basados en dicionarios, 273
métodos estadísticos, 272, 273
métodos intuitivos, XLVIII, 289, 290
métricas de error, 287, 288
principio de la, 30, 274, 276–278, 408, 418, 464, 489
y RGC, 279
progresiva, 279, 366–373, 1160
codificación de la geometría de crecimiento, 372, 373
mediana, 371
razones para la, 271
RLE, 272
submuestreo, XLVIII, 289
imagen, frecuencias, 294
imagen, transformadas, XLVIII, 290–337, 490, 556–562, 1164
imagen, wavelet, descomposiciones, 590–598
imágenes (estándar), 338, 339, 341
impares, funciones, 335
impropias, rotaciones, 319
improved grayscale quantization (IGS), 272
incertidumbre (principio de), 542, 543, 552
y MDCT, 810
independiente (de la resolución), compresión, 1161
inecuación, (desigualdad de Kraft-MacMillan), 76, 77, 1155
y códigos de Huffman, 81
información, teoría, 51–58, 177, 1153
inglés (texto en), 17, 146, 176
frecuencias de las letras, 4
integer wavelet transform (IWT), 608–610
Interactiva, Asociación Multimedia (IMA), 741
Internacional, Comisión sobre Iluminación, *véase* CIE
interpolación, polinomios de, XLVI, XLIX, 429, 435–440, 605–608, 613, 1153
grado 5, 607
Lagrange, 755
intuitiva, compresión, 19
intuitivos (métodos) para la compresión de imágenes, XLVIII, 289, 290
inversa, DCT modificada, 795
inversa, transformada de Walsh-Hadamard, 298, 299
inversa, transformada discreta del coseno, 304–311, 313, 315–329, 331, 333, 335, 349, 350, 1081
en MPEG, 678–694
mismatch (desfase), 680, 695

modificada, 795
inversa, transformada discreta del seno, 335–337
inversión del intervalo (en codificador QM), 140, 141
invertido, archivo, 868
irrelevancia (y compresión con pérdida), 271
irreversible, compresión de texto, 21
ISDN, *véase* RDSI
Ismail, G. Mohamed, XLIV, 178
ISO, 107, 343, 674, 1153, 1155, 1158
estándar 15444, JPEG2000, 641
JBIG2, 384
recomendación CD 14495, 360, 1155
iteradas, sistemas de funciones, 515–529, 1154
ITU, 108, 1146, 1151, 1154
ITU-R, 348
recomendación BT.601, 348, 659
ITU-T, 110, 375
JBIG2, 384
recomendación H.261, 700, 701, 1152
recomendación H.264, 703, 1152
recomendación T.4, 110, 115
recomendación T.6, 110, 115, 1151
recomendación T.82, 375
recomendación V.42bis, 234, 1164
y documentos de adiestramiento de faxes, 111, 338, 410
y MPEG, 675, 1158
IWT, *véase* integer wavelet transform

JBIG, XLVIII, 374–382, 384, 629, 1154, 1160
modelo de probabilidad, 376
y EIDAC, 395
y FABD, 454
JBIG2, XLVIII, 136, 384–394, 920, 1154
y DjVu, 631, 1150
jerárquica, codificación (en compresión progresiva), 367, 610–613
jerárquica, compresión de imágenes, 347, 356, 1152
JFIF, 357, 359, 360, 1154
Joyce, James Aloysius Augustine (1882–1941), 18, 21
JPEG, XLVIII, 35, 136, 143, 272, 294, 298, 343–357, 360, 394, 453, 629, 680, 681, 937, 1145, 1154, 1155, 1160
comparado con H.264, 714
modo sin pérdidas (lossless), 356, 357
similar a MPEG, 677
y artefactos bloque, 344, 638
y cadenas dispersas, 868
y compresión progresiva de imágenes, 367
y DjVu, 630
y WSQ, 638
JPEG 2000, XLV, XLVII–XLIX, 136, 348, 533, 638–651, 920
JPEG-LS, XLVIII, 357, 360–366, 641, 935, 1155
Jung, Robert K., 236, 1144

Karhunen-Loève, transformada, XLVIII, 298, *véase* transformada de *Hotelling*, 301, 303, 334, 560
Kari, Jarkko, 500
Katz, Philip W. (1962–2000), 235, 236, 245, 248, 1159
King, Stephen Edwin (escritor, 1947–), 919
Klaasen, Donna, 858
KLT, *véase* transformada de *Karhunen-Loève*
Knowlton, Kenneth (1938–), 468
Knuth, Donald Ervin (1938–), XL, 1195
Kolmogorov-Chaitin, complejidad, 57
Kraft-MacMillan, inecuación, 76, 77, 1155
y códigos de Huffman, 81
Kronecker, función delta, 331
Kronecker, Leopold (1823–1891), 332
KT, estimación de probabilidad, 168, 1155
KT, límite (y extinción de los dinosaurios), 169

L Sistemas, 898
curva de Hilbert, 489
La Disparition (novela), 148
Lagrange, interpolación, 755, 767
Lanchester, John (1962–), XLVI
Lang, Andrew (1844–1912), 91
Langdon, Glen G., 267, 417
Laplace, distribución, 277, 427, 429–432, 434, 443, 453, 757, 758, 959, 1156, 1157, 1159
de diferencias, 724–726
en audio MLP, 746
en FLAC, 764
en imagen MLP, 427
Laplaciana, pirámide, XLVII, 533, 591, 610–613, 1156
Lau, Daniel Leo, 499
lazy wavelet transform, *véase* transformada wavelet *lenta*
LBG, algoritmo para cuantificación vectorial, 398–402, 490, 941
Lempel, Abraham (1936–), 45, 176, 1156
patente LZ78, 265
Lempereur, Yves, 37, 1144

Lena (imagen), 291, 338, 339, 386, 564, 997, 1100
    difuminada (*blurred*), 1099
lenta, transformada wavelet, 600
Les Revenentes (novela corta), 148
Levinson, Norman (1912–1975), 769
Levinson-Durbin, algoritmo, 764, 769, 782, 1196
lexicográfico, orden, 183, 847
LHA, 235, 1156
LHArc, 235, 1156
libre, compresión de audio sin pérdidas (FLAC), XLI, XLII, 726, 759–769, 782, 1151
Liebchen, Tilman, 780
LIFO, 213
lifting, esquema, 598–608, 1156
Ligtenberg, Adriaan (1995–), 327
línea (como curva de relleno de espacio), 1092
línea (descomposición de imagen wavelet), 592
lineal, codificación predictiva (LPC), XLI, 764, 767–769, 779, 1143
    datos hiperespectrales, 935, 936
lineal, predicción
    ADPCM, 739
    ALS, 781–783
    audio MLP, 746
    cuarto orden, 767
    FLAC, 763
    shorten, 754
lineales, sistemas, 1148
lipograma, 147
lista (estructura de datos), 1148
little endian (orden de bytes), 249
little endian (orden de bytes) en formato Wave, 731
LLM, algoritmo para la DCT, 327
lockstep (en sincronía), 94, 137, 206
LOCO-I (predecesor de JPEG-LS), 360
Loeffler, Christoph, 327
logarítmico, árbol (en descomposición wavelet), 571
logaritmo
    como función de información, 53
    usado en métricas, 13, 288
lógica, compresión, 12, 175
loros (imagen), 280
lossless (sin pérdidas), compresión, 10, 21, 1156
lossy (con pérdidas), compresión, 10, 1156
LPC, *véase* codificación predictiva *lineal*
LPC (compresión de voz), 750–753
LPVQ, 1156
Luminancia, 1156
luminancia, componente del color, 276, 278, 344, 347–350, 353, 562, 675
    uso en PSNR, 287
luz, 542
    visible, 348
LZ1, *véase* LZ77
LZ2, *véase* LZ78
LZ77, XLV, 176, 180–183, 199, 203, 408, 855, 858, 859, 900, 1145, 1149, 1152, 1156, 1157, 1162, 1166
    deficiencias, 186
    y Deflate, 237
    y LZRW4, 202
    y tiempos de repetición, 188
LZ78, 176, 186, 193–196, 203, 859, 1157
    patente, 265
LZAP, 215, 217, 1157
LZARI, XL, 186, 1157
LZC, 200, 230
LZEXE, 260, 261, 1150
LZFG, 60, 196–200, 1157
    patente, 199, 265
LZH, 180
LZMA, XLI, XLII, 136, 248–254, 1157
LZMW, 214, 215, 1157
LZP, 220–227, 268, 411, 852, 853, 1157
LZRW1, 199–201
LZRW4, 202, 203
LZSS, XL, 183–186, 235, 1156, 1157
    usado en RAR, XL, 233
LZW, 203–214, 790, 1111, 1157, 1164
    basado en palabras, 880
    decodificación, 206, 207
    patente, 203, 264–266, 1159
    UNIX, 230
LZX, XLV, 191–193, 1157
LZY, 217–220, 1157

m4a, *véase* codificación *avanzada* de audio
m4v, *véase* codificación *avanzada* de audio
Mackay, Alan Lindsay (1926–), 11
MacWrite (compresión de datos en), 24
Mahler, tercera sinfonía, 815
mallas (compresión de), edgebreaker, XLIX, 846, 902, 903, 905–913
Mallat, Stephane (y descomposición multirresolución), 590
Malvar, Henrique (Rico), 703
Manber, Udi, 928
mandril (imagen), 338, 340, 597, 997

Manfred, Eigen (1927–), 122
Marcellin, Michael W. (1959–), 1167
Markov, modelo, XLIX, 35, 106, 145, 279, 376, 897, 1085, 1112
Marx, Julius Henry (Groucho, 1890–1977), 715
matemáticas
  álgebra vectorial, 1035
    escalar por vector, 1035
    producto escalar, 1035
    producto mixto de tres vectores, 1037
    producto vectorial, 1036
    propiedades del producto vectorial, 1036
    proyección de un vector, 1037
    suma vectorial, 1035
  alfabeto griego, 1049
  convolución, 1040–1044
  identidades trigonométricas, 1033–1035
  L-sistemas, 1045–1048
    notación BNF, 1045
  matrices, 1030
    identidad, 1031
    inversa, 1031
    operaciones con, 1033
    ortogonales, 1032
    ortonormales, 1032
    productos tensoriales, 1031
    suma y resta, 1030
    valor propio, 1032
    vector propio, 1032
  números complejos, 1038
    coordenadas cartesianas, 1038
    coordenadas polares, 1038
    diagrama de Argand, 1038
    fórmula de Euler, 1039
    multiplicación, 1038
    suma y diferencia, 1038
    valor absoluto, 1038
    visualización de los, 1039
  placa de Petri, 1044
  regiones de Voronoi, 1044
  sumas útiles
    desarrollo de una exponencial, 1029
    potencias de 2, 1030
    primeros n cuadrados, 1030
    primeros n enteros, 1030
    serie geométrica, 1029
    teorema del binomio, 1029
  teselas de Dirichlet, *véase* regiones de Voronoi
Matisse, Henri (1869–1954), 921
Matlab, software, propiedades de, 291, 564, 566
matrices
  descomposición QR, 319, 330
  secuencia de, 299
  valores propios, 303
matriz, *véase* arreglo
Matsumoto, Teddy, 261
Maugham, William Somerset (1874–1965), 529
MDCT, *véase* transformada *discreta* del coseno, modificada
mean square error (MSE), *véase* error cuadrático *medio*
media, diferencia absoluta, 490, 667
media, diferencia cuadrática, 667
mediana, definición de, 371
medidas de distorsión, 397
medidas de la eficiencia de la compresión, 12–14
medio, error absoluto, 667
medio, error cuadrático (MSE), 13, 287, 615
mejora de cuantificación de la escala de grises, *véase* IGS
memoria (sin), fuente, 44, 168, 169, 171, 172
  definición de, 4
meridian, empaquetamiento sin pérdidas, *véase* MLP (audio)
métrica, definición de, 526
Mexicano (sombrero), wavelet, 547, 548, 550, 552
Meyer, Carl D., 330
Meyer, Yves (y descomposición multirresolución), 590
Microcom, Inc., 28, 100, 1158
midriser, cuantificación, 737
midtread, cuantificación, 737
  en audio MPEG, 801
Millan, Emilio, LII
miniatura, 1163
mirroring (reflejo), *véase* lockstep
MLP, 277, 419, 426–440, 442, 443, 726, 1156, 1157, 1159
MLP (audio), XLV, 14, 426, 744–747, 1158
MMR, codificación, 115, 384, 387, 388, 390
  en JBIG2, 385
MNG, *véase* multiple-image network format
MNP clase 5, 28, 101, 102, 104, 105, 1158
MNP clase 7, 105, 106, 1158
modelo
  adaptativo, 147, 370
  basado en el contexto, 146
  de DMC, 889, 891
  de JBIG, 376
  de MLP, 429

de orden *N*, 148
de PPM, 135
de probabilidad, 122, 145
en MLP, 434
estático, 146
estadístico, 1163
máquina de estados finitos, 888
Markov, XLIX, 35, 106, 145, 279, 376, 897, 1085, 1112
probabilidad, 132, 170, 375, 376
problema de la probabilidad cero, 146
módem, 7, 54, 95, 100, 110, 234, 235, 1158, 1164
Moffat, Alistair, 417
Molnár, László, 261
Monk, Ian, 148
monkey (audio de), XLI, 778, 1158
monocromática (imagen), *véase* imagen *binivel*
Montesquieu (Charles de Secondat, 1689–1755), 918
Morlet, wavelet, 547, 552
Morse, código, 19, 51
Morse, Samuel Finley Breese (1791–1872), 3, 51
Moschytz, George S. (método LLM), 327
Motil, John Michael (1938–), 85
Motta, Giovanni (1965–), XLII, XLVI, 846, 920, 931, 940, 1156
mover-al-frente, método, 10, 39–43, 847, 849, 1145, 1156, 1158
inverso de, 851
y wavelets, 554, 562
movimiento, compensación (en compresión de vídeo), 666
movimiento, vectores (en compresión de vídeo), 666, 707
Mozart, Joannes Chrysostomus Wolfgangus Theophilus (1756–1791), XLIII
mp3, 790–814
comparado con AAC, 820, 821
madre de, *véase* Vega, Susan
y *Tom's Diner*, 839
mp4, *véase* codificación de audio *avanzada*
MPEG, XLVII, 348, 656, 668, 1154, 1158
en cuantificación, 679–684
en DCT, 678–684
en IDCT, 679–694
imagen D, 685
similar a JPEG, 677
MPEG-1, audio, XLII, XLVII, 12, 298, 790–814, 816
MPEG-1, vídeo, 675–695
MPEG-2, audio, XLII, 815–840
MPEG-3, 816
MPEG-4, 695–700, 816
AAC, 835–837
codificación de audio sin pérdidas (ALS), XLI, 779, 780, 782–789, 836, 1143
codificadores de audio, 836
extensiones de AAC, 835–837
MPEG-7, 816
MQ, codificador, 136, 385, 640, 641, 646
MSE, *véase* mean square error
MSZIP (deflate), 191
Muestra, see Token1163
μ-law companding, 734–739, 750
multianillo, codificación de cadena, 895
multiple-image network format (MNG), 258
multiring chain coding, *véase* codificación de cadena *multianillo*
multirresolución, árbol (en descomposición wavelet), 572
multirresolución, descomposición, 590, 1158
multirresolución, imagen, 503, 1158
Murray, James, 49
musical, notación, 545
musicales, notaciones, 385
Musset, Alfred de (1810–1857), XV
Muth, Robert, 928

N-trees, 474–480
nanómetro (definición de), 347
negación e intercambio, regla, 518
Nelson, Mark, 17
Newton, Isaac (1643–1727), 130
no adaptativa, compresión, 10
no diferenciables, funciones, 590
no estándar (descomposición de imágenes wavelet), 592
normal, distribución, 472, 1151, 1158
NSCT (never the same color twice), 661
NTSC (estándar de televisión), 655, 656, 661, 687, 841
numérica, historia (bruma de la), 440
Nyquist, Harry (1889–1976), 544
Nyquist, tasa o frecuencia de, 544, 580, 722
Nyquist-Shannon, teorema de muestreo, 723

Oberhumer, Markus Franz Xaver Johannes, 261
oclusivos, sonidos, 748
OCR (reconocimiento óptico de caracteres), 882
octasección, XLVIII, 486, 488, 1160

octasection, *véase* octasección
octava, 572
  en análisis wavelet, 572, 594
octree (en compresión prefija), 876
octress, 475
Ogg Squish, 759
Ohm (ley de), 720
oído (humano), 726–730
ojo
  e integración espacial, 657, 675
  resolución del, 348
  respuesta en frecuencia de, 680
  y percepción de bordes, 272
  y sensibilidad al brillo, 276
Okumura, Haruhiko (LZARI), XL, LII, 186, 235, 1156, 1157
óptimo, método de compresión, 12, 188
ortogonal
  proyección, 452
  transformada, 290, 556–562
ortogonales, filtros, 571
ortonormal, matriz, 291, 294, 313, 314, 319, 568, 579

packing (compactación), 22
PAL (estándar de televisión), 654, 656, 659, 687, 841
palabras (basado en), compresión, 878–881
Pandit, Vinayaka D., 510
paramétrico, polinomio *cúbico* (PC), 436
parcial (correlación), coeficientes, *véase* parcor
parcor (en MPEG-4 ALS), 782
pares, funciones, 335
paridad, 261
  de las funciones, 335
  horizontal, 262
  vertical, 262
parsimonia (principio de), 25
Pascal
  curva de Hilbert, 963, 964
  curva de Sierpiński, 965
Pascal, Blaise (1623–1662), 4, 67, 174
patentes de algoritmos, XLVIII, 248, 264–266, 1159
patrón de substitución, 29
Pavlov, Igor (creador de 7z y LZMA), XLI, XLII, 248, 254, 1143, 1157
PDF, *véase* Formato de Documento *Portátil*
PDF (formato de documento portátil de Adobe)
  y DjVu, 630
Peak Signal to Noise Ratio (PSNR), 13, 287, 288
Peano, curva, 495
  recorrido de la, 499, 500
  usada para errores, 490
pecados (los siete capitales), 817
pel, *véase también* píxeles
  clasificación por diferencia (PDC), 668
  compresión de facsímiles, 110, 269
  relación de aspecto, 656, 687
pequeños, números (fáciles de comprimir), 41, 352, 356, 370, 448, 454, 458, 459, 849, 1111
percepción, compresión de, 11
Percival, Colin (creador de BSDiff), 930, 931, 1145
Perec, Georges (1936–1982), 148
permutación, 847
petri, plato o placa de, 1165
phased-in (por etapas), códigos binarios, 64, 96, 230
Phong, sombreado (para superficies poligonales), 902
Picasso, Pablo Ruiz (1881–1973), 321, 1008
PIFS (compresión IFS), 525
pila (estructura de datos), 92, 93, 213, 1148
pimientos (imagen), 338, 997
pirámide (descomposición de imágenes wavelet), 556, 592
pirámide (Laplaciana), XLVII, 533, 591, 610–613, 1156
pirámide, codificación (en compresión progresiva), 367, 458, 460, 463, 591, 610–613, 1156
píxeles, 30, 375, *véase también* pel
  altamente correlacionados, 274
  background (fondo), 375, 380, 1144
  correlacionados, 290
  definición de, 269
  descorrelacionados, 275, 278, 290, 298, 301, 336
  foreground (primer plano), 375, 380, 1144
PKArc, 235, 1159
PKlite, 235, 261, 1159
PKunzip, 235, 1159
PKWare, 235, 1159
PKZip, 235, 1159
Planck, constante de, 543
plano de bits, *véase* bitplane
PMMA, 155
PNG, *véase* gráficos de red *portátil*es
pod, código, XL, 758
Poe, Edgar Allan (1809–1849), 10
Poisson, distribución, 155
poligonales (superficies), compresión

edgebreaker, XLIX, 846, 902, 903, 905–913
polinomio
bicúbico, 439
cúbico paramétrico, 436
definición de, 263, 435
representación paramétrica, 435
polinomios (de interpolación), XLVI, XLIX, 429, 435–440, 605–608, 613, 1153
grado 5, 607
ponderación del árbol de contexto, *véase contexto* (árbol de ponderación)
Porta, Giambattista della (1535–1615), 3
Portátil, Formato de Documento (PDF), XLII, 846, 918–920, 1159
Portátiles, gráficos de red (PNG), 254–258, 264, 1159
PostScript (y patente LZW), 264
Poutanen, Tomi (LZX), 191
Powell, Anthony Dymoke (1905–2000), 18, 554
PPM, XLV, 45, 146–159, 161–165, 442, 852, 861, 884, 895, 900, 1159
basado en palabras, 881
exclusión, 154
trie, 156
vine pointers, 158
PPM (rápido), XLV, 165, 166
PPMB, 155, 443
PPMC, 152, 155
PPMD (en RAR), 233
PPMdH (de Dmitry Shkarin), 248
PPMP, XLV, 155
PPMX, XLV, 155
PPMZ, XLV, 163–165
PPM*, XLV, 161–163
PPPM, 442, 443, 1159
predicción, 1159
a largo plazo, 787
AAC, 834, 835
ADPCM, 739–743
BTPC, 460, 462, 463
CALIC, 445, 446
CELP, 754
compresión de imágenes, 277
definición de, 146
determinista, 375, 382
FELICS, 1085
JPEG-LS, 347, 356, 360, 362
LZP, 220
LZRW4, 202
MLP, 426–429
MLP audio, 745
monkey, audio, 778
n-ésimo orden, 451, 755, 781
Paeth, 257
PNG, 254, 256
PPM, 150
PPMZ, 163
PPM*, 161
PPPM, 442
probabilidad, 145
progresiva, 786
vídeo, 664, 669
preecos
en AAC, 833
en audio MPEG, 810–814
prefija, compresión
cadenas dispersas, XLIX, 873–877
imágenes, XLVIII, 480–482, 1159
prefija, propiedad, 107, 421, 1159, 1164
definición de, 59
prefijo, códigos, 37, 59–65, 188, 229, 276, 277, 414
y compresión de vídeo, 668
probabilidad
conceptos, XLVI, XLIX
condicional, 1147
conjunta, 952
correlación, 951
covarianza, 951
desviación estándar, 950
distribución geométrica, 68
esperanza, 949
eventos independientes, 952
media, 949
modelo, 17, 121, 132, 170, 370, 375, 1158
adaptativo, 147
variable aleatoria discreta, 949
varianza, 950
Prodigy (y patente LZW), 264
progresiva, compresión, 374, 378–382
progresiva, compresión de imágenes, 279, 366–373, 1160
codificación de la geometría de crecimiento, 371–373
mediana, 371
MLP, 277
opción de pérdida, 366, 427
SNR, 367, 650
progresiva, predicción, 786
progresivo, FELICS, 423–428, 1160
propiedades de la voz, 747–749

Proust, Marcel Valentin Louis George Eugene (1871–1922), 1155, 1195
Prowse, David (Darth Vader, 1935–), 95
psicoacústico, 11, 726–730
psicoacústico, modelo (en audio MPEG), 790, 792, 796, 800, 803, 807, 808, 813, 1107, 1160
PSNR, *véase* Pak Signal to Noise Ratio287
pulsos (codificados), modulación (PCM), 723
punting (definición de), 454
punto de equilibrio en LZ, *véase break* even point en LZ
puntos (correlación cruzada de), 291

QIC, 108
QIC-122, 188–191, 1160
QM, codificador, XLVIII, 136–145, 167, 345, 356, 1160
QMF, *véase* quadrature mirror filters
QR, descomposición de matrices, 319, 330
QTCQ, *véase* quadtree classified trellis coded quantized
quadrature mirror filters, 579
quadrisection, see cuatrisección1160
quadtree classified trellis coded quantized
  compresión de imágenes wavelet (QTCQ), 624, 625
Quadtrees, 1160
quadtrees, XLV, XLVIII, 276, 402, 464–482, 489, 501, 1144, 1152, 1159, 1160, 1165
  e IFS, 527
  y árboles de orientación espacial, 624
  y compresión prefija, XLVIII, 480–482, 873, 1159
  y curvas de Hilbert, 489
  y numeración de cuadrantes, 465, 501
  y quadrisection (cuatrisección), 482, 484
Quantum (codificador basado en diccionarios), 191
Quayle, James Danforth (Dan, 1947–), 663
quincunce (descomposición de imágenes wavelet), 592

Racha, 1161
rango, codificación, 135, 136, 779
  en LZMA, 250
Rao, Ashok, 510
rápido, PPM, XLV, 165, 166
RAR, XL, 231–234, 1160
Rarissimo, 234, 1160
Raster, 1161
raster (orden), escaneo, 36, 276, 278, 290, 366, 408, 423, 428, 443, 444, 454, 456, 671, 677, 690
raster (origen de la palabra), 655
ratio o razón de compresión, 12, 1147
razones para la compresión de datos, 3
RDSI, 1153
recursiva, reducción de rango (3R), XL, XLII, 46, 47, 1161
Reducción recursiva del rango, XL, XLII, 46, 47, 1161
redundancia, 58, 78, 107
  alfabética, 4
  definición de, 55, 1077
  dirección de más alta, 592
  espacial, 274, 664, 1165
  temporal, 664, 1165
  y compresión de datos, 4, 232, 271, 889
  y contextual, 4
  y fiabilidad, 107
Reed-Solomon, código corrector de errores, 232
reflection (reflexión), 517
reflejado, código de Gray, XLVIII, 277, 279–285, 375, 453, 499, 1152
  curva de Hilbert, 489
reflexión (reflection), 517
reflexión, coeficientes de, *véase* parcor
refresco, frecuencia (de películas y TV), 653–656, 662, 675, 686
relativa, codificación, 30, 213, 448, 665, 1150, 1161
  en JPEG, 347
renormalización (en el codificador QM), 138–145, 1066
repetición, buscador de, 227–229
repetición, tiempos de, 187, 188
representación (de funciones), 580, 582–584
resolución
  de imágenes (definición), 269
  de la HDTV, 661–663
  de la televisión, 655, 657
  del ojo, 348, 1007
Reynolds, Paul, 177
RGB
  espacio de color, 347
  razones para utilizarlo, 348
RGC, *véase* código de Gray *reflejado*
Ribera, Francisco Navarrete y (1600–?), 148
Rice (códigos), 47, 64, 426, 1143, 1151, 1161, 1162
  código subexponencial, 426
  definición de, 71
  en datos hiperespectrales, 937

FLAC, 759, 764
no usados en WavPack, 773
PPM rápido, 167
Shorten, 757
Rice, Robert F. (desarrollador de los códigos Rice), 71, 757
Richardson, Iain, 703
Richter, Jason James (1980–), 843
Riding the Bullet (novela), 919
RIFF (Resource Interchange File Format), 732
Rijndael, *véase* estándar de encriptación avanzada
Rizzo, Francesco, 940, 1156
RLE, *véase también* codificación *run-length*, 25–49, 111, 276, 277, 1161
archivos de imagen BMP, XLIV, 39, 1145
BinHex4, 37, 38
compresión de imágenes, 31–34
en JPEG, 345
QIC-122, 188–190, 1160
y método BW, 847, 848, 1145
y sonido, 729
y wavelets, 554, 562
RMSE, *véase* root mean square error
Robinson, John Allen, LII
Romanos, números, 534
root mean square error (RMSE), 287
Roshal, Alexander Lazarevitch, 231
Roshal, Eugene Lazarevitch (1972–), XL, XLII, 231, 233, 1160
rotación, 517
90º, 518
matriz de, 319, 322
rotaciones
en tres dimensiones, 334, 335
Givens, 319, 322–325, 327–330
impropias, 319
ruleta, juego (y distribución geométrica), 67, 68
Run, *véase* Racha
Run length, see Racha1161
run-length encoding, *véase* RLE
run-length, codificación, 9, 25–49, 82, 272, 1158
BTC, 414
en imágenes, 277
FLAC, 763
MNP5, 101
y códigos Golomb, *véase también* RLE, 67–75
y EOB, 351
Ryan, Abram Joseph (1839–1886), 556, 562, 569

sólido, archivo, *véase* RAR
símbolos, clasificación de, XLIX, 146, 845, 851–853, 855, 861, 862, 1163
Sagan, Carl Edward (1839–1886), 44, 658
sampling (muestreo de audio), 721–726
Samuelson, Paul Anthony (1915–), 153
Saravanan, Vijayakumaran, LII, 326, 1081
Sayood, Khalid, 447
SBC (compresión de voz), 750
scaling (escalado), 517
Scott, Charles Prestwich (1846–1932), 674
SCSU, 1161
SECAM (estándar de televisión), 654, 659, 687
secuencia, see cadena1163
secuencia (definición de), 299
semiadaptativa, codificación de Huffman, 94
semiadaptativa, compresión, 10, 94, 1161
semiestructurado, texto, XLIX, 846, 902, 1162
semitonos, 378, 380, 381, 1152
y compresión de faxes, 114
sequitur, XLIX, 29, 846, 897–902, 1147, 1162
y métodos basados en diccionarios, 902
sesgadas, probabilidades, 124
sesgado, código de Elias Gamma, 758
Set Partitioning In Hierarchical Trees (SPIHT), XLVII, 533, 614–625, 1151, 1162
y CREW, 625
SHA-256, algoritmo hash, 249
Shanahan, Murray, 167
Shannon, Claude Elwood (1916–2001), 54, 77, 851, 982, 1150, 1153
Shannon-Fano, método de, 51, 59, 77, 78, 1153, 1162, 1163
shearing (extrusión), 516
Shkarin, Dmitry, 233, 248
shorten (compresión de voz), XLI, 71, 754–759, 764, 935, 1151, 1162
Sierpiński
curva, 489, 494, 495
gasket, 522–524, 1095, 1097
triángulo, 520, 522, 1095
Sierpiński, Waclaw (1882–1969), 489, 494, 522
siete pecados capitales, 817
Signal to Noise Ratio (SNR), 288
Signal to Quantization Noise Ratio (SQNR), 288
signo-magnitud (representación de enteros), 647
silencio, compresión, 730
simétrica (descomposición de imágenes wavelet), 591, 634, 635
simétrica, compresión, 11, 176, 190, 345, 634
simétrica, transformada wavelet discreta, 634

simétrico, contexto (de un píxel), 419, 429, 443, 446, 646
Símbolo, 1163
similitud de contextos basados en el orden, XLIX, 845, 862–867
simples (imágenes), EIDAC, XLVIII, 395, 1162
sin voz, sonido, 748
sintética, imagen, 270
sistema visual humano, 999
  colores aditivos y sustractivos, 1004, 1005
  colores complementarios, 1006, 1007
  colores complementarios sustractivos, 1007
  densidad espectral, 1012–1015
  difuminado, 1019
  difuminado ordenado, 1020–1022
  difuminado por difusión, 1023–1026
  difuminado promedio restringido, 1022
  difusión de punto, 1026–1028
  el color y los ojos, 999–1001
  el cubo RGB, 1003, 1004
  estándar CIE, 1015–1017
  luminancia y crominancia, 1008, 1009
    corrección gamma, 1010, 1011
    RGB, YUV e YCbCr (equivalencia), 1012
  modelo de color
    HLS, 1001, 1002
    HSV, 1001–1003
    RGB, 1002, 1003
  modelos de color sustractivo, 1005, 1006
  semitonos (halftoning), 1017–1019
  visión humana, 1007
    bastones, 1007
    colores calientes, 1007
    colores fríos, 1007
    conos, 1007
SNR, *véase* Signal to Noise Ratio
SNR, compresión de imágenes progresiva, 367, 639, 650
Soderberg, Lena (de imagen famosa, 1951–), 339, 997
sombra, máscara de, 656
sonido
  de voz, 747
  fricativo, 748
  oclusivo, 748
  propiedades, 718–720
  sampling (muestreo), 721–726
  sin voz, 748
soporte (de una función), 553
sorpresa (como una medida de la información), 52, 58
source coding (nombre formal de la compresión de datos), 3
SourceForge.net, 759
SP, teoría (sencillez y poder), 8
space-filling (curvas), XLVI, XLIX, LI, 276, 488–500, 1162
  Hilbert, 490–494
  Peano, 495
  Sierpiński, 494, 495
Sperry Corporation
  patente de LZ78, 265
  patente de LZW, 264
SPIHT, *véase* Set Partitioning In Hierarchical Trees
SQNR, *véase* Signal to Quantization Noise Ratio
Squish, 759
Stafford, David (compresión basado en diccionarios quantum), 191
star-step-stop (inicio-paso-parada), códigos, 60
Storer, James Andrew, 183, 940, 1119, 1156, 1157, 1167
stream, see cadena1163
streaming, modo continuo, 12, 846
subbanda (tamaño mínimo de), 590
subbanda, transformada, 290, 556–562, 569
subexponencial, código, 64, 426, 427, 757
Submuestreo, 1163
submuestreo, XLVIII, 289, 1163
  en compresión de vídeo, 665
sucesivas, aproximaciones (in JPEG), 345
Swift, Jonathan (1667–1745), LII
SWT, *véase* transformada wavelet discreta *simétrica*
Szymanski, Thomas G., 183, 1157

T/F, codificador, *véase* codificador en *tiempo/frecuencia* (T/F)
taps (filtros de coeficientes wavelet), 573, 586, 587, 591, 625, 634, 1163
TAR (Tape ARchive de UNIX), 1163
tasa de bits, *véase* bitrate
tasa de distorsión, teoría de la, 177
televisión
  estándares usados en, 562, 654–659, 841
  relación de aspecto, 642, 655–658
  resolución de la, 655, 657
televisión de alta definición, *véase* HDTV
temporal, enmascaramiento, 726, 728, 792
temporal, redundancia, 664, 1165

tera (= 2^{40}), 633
texto
aleatorio, 7, 81, 1057
archivos de, 10
case flattening, 21
inglés, 4, 5, 16, 17, 176
lenguaje natural, 106
semiestructurado, XLIX, 846, 902, 1162
texto, compresión, 9, 17, 25
clasificación de símbolos, 851–853, 855, 861, 862, 1163
LZ, 177, 1156
QIC-122, 188–190, 1160
RLE, 25–30
textual, substitución, 403
textuales, compresión de imágenes, 845, 882–887, 1163
Thomson, William (Lord Kelvin 1824–1907), 544
tiempo/frecuencia (T/F), codificador, 819, 1143, 1163
TIFF
y JBIG2, 1154
y patente LZW, 264
título de este libro, 18
Tjalkins, Tjalling J., LII
Toeplitz, Otto (1881–1940), 768, 769
token (definición de), 1163
tokens
en compresión prefija, 481, 482
en emparejamiento de bloques, 410, 411
en LZ77, 180, 181, 237
en LZ78, 193, 194
en LZFG, 196, 197
en LZSS, 183, 185
en LZW, 203
en MNP5, 101, 102
en QIC-122, 188
métodos de diccionario, 175
Transformaciones afines, 1143
Transformada
discreta del coseno, 1149
transformadas, 9
coeficiente DC, 294, 304, 307, 336, 345, 351–353, 356
coeficientes AC, 294, 304, 307
definición de, 534
discreta de Fourier, 349
discreta del coseno, XLVIII, 298, 303–335, 344, 349, 350, 560, 707, 711, 1149
3D, 303, 937–939
discreta del seno, XLVIII, 335–337
Fourier, LI, 290, 534–545, 560, 572, 574, 596
Haar, XLVII, XLVIII, 298, 300, 332, 552–568
Hotelling, *véase* transformada de *Karhunen-Loève*
imágenes, XLVIII, 290–337, 490, 556–562, 1164
inversa de Walsh-Hadamard, 298, 299
inversa discreta del coseno, 304–311, 313, 315–329, 331, 333, 335, 349, 350, 1081
inversa discreta del seno, 335–337
Karhunen-Loève, XLVIII, 298, 301, 303, 334, 560
ortogonal, 290, 556–562
subbanda, 290, 556–562, 569
Walsh-Hadamard, XLVIII, 298, 299, 560, 711, 712, 1079, 1080
trends (en una imagen), 546
tresbolillo, *véase* quincunce
triángulo, compresión de mallas de, edgebreaker, XLIX, 846, 902, 903, 905–913, 1164
triángulo (Sierpiński), 520, 522, 1095
trie
definición de, 156, 195
LZW, 207
Patricia, 253
trigrama, 146, 147
y redundancia, 4
trit (dígito ternario), 53, 65, 87, 500, 788, 1058, 1163, 1164
Truţa, Cosmin, XLII, XLIV, XLVI, 75, 258
TSVQ (Tree-Structured Vector Quantization), *véase* cuantificación vectorial con *estructura de árbol*
Tunstall, código, 66, 67, 1164
Twain, Mark (1835–1910), 28
two-pass compression, *véase* compresión de *2 pasadas*

Udupa, Raghavendra, XLIX, LII, 510
UIT, see ITU1154
Ulysses (novela), 21
unario, código, 60–65, 226, 421, 426, 482, 1054, 1151, 1164
general, *véase también* edad de piedra, binario, 61, 197, 410, 1084, 1164
Unicode, 183, 1084, 1146, 1164
Unicode, compresión, XLV, 846, 913–917, 1161
unificación de métodos estadísticos y de diccionario, 266–268

uniforme (descomposición de imágenes wavelet), 596
unirse a la DC community, 1119, 1167
Unisys (y patente LZW), 264
universal, método de compresión, 12, 188
univocalic, 148
UNIX
  compact, 94
  compress, 196, 200, 230, 231, 264, 1146, 1159
  Gzip, 230, 264
UPX (compresor exe), 261

V.32, 95
V.32bis, 234, 1164
V.42bis, 7, 234, 235, 1164
Valens, Clemens, 651
Valenta, Vladimir, 500
valores propios de una matriz, 303
Vanryper, William, 49
variable, códigos de tamaño, 4, 5, 51, 58–65, 82, 94, 99, 101, 106, 120, 175, 177, 1057, 1159, 1162, 1164
  definición de, 58
  diseño de, 59
  en compresión de facsímiles, 111
  no ambiguos, 76, 1155
  y cadenas dispersas, 869–872
  y fiabilidad, 107, 1161
varianza, 276
  como energía, 294
  de códigos de Huffman, 80
  de diferencias, 448
  definición de, 432
  y MLP, 430–432, 434
VCDIFF (diferenciación de archivos), 922–924, 1165
vectorial, cuantificación, 290, 367, 396–402, 515, 1165
  AAC, 836
  adaptativa, XLVIII, 403, 405–407
  algoritmo LBG, 398–402, 490, 941
  datos hiperespectrales, 939–942
  de particionado localmente óptimo (LPVQ), 940–942
  estructurado en árbol, 402
  ruido de cuantificación, 450
vectoriales (espacios), 331–334, 452
Vega, Susan (madre del mp3), 839
Veldmeijer, Fred, XLVII
ventana deslizante, compresión, XLVIII, 180–186, 408, 902, 1145, 1162
  tiempos de repetición, 187, 188
vídeo
  alta definición, 661–663
  analógico, 653–659
  digital, 660, 661, 1149
vídeo, compresión, XLVII, 664–714, 1165
  compensación de movimiento, 666
  diferenciación, 665
  diferenciación de bloque, 665
  H.261, XLVII, 700, 701, 1152
  H.264, XLI, 356, 702–714, 1152
  inter frame, 664
  intra frame, 664
  medidas de distorsión, 667, 668
  MPEG-1, 668, 675–695, 1158
  MPEG-1 de audio, XLVII, 12, 298, 790–814, 816
  submuestreo, 665
  vectores de movimiento, 666, 707
vine pointers, 158
visión (humana), 348, 932
vmail (email con vídeo), 660
vocoders, codificadores de voz, 748, 750
Voronoi, diagramas de, 402, 1165
voz (propiedades de la), 747–749
voz, compresión, 585, 718, 747, 748, 750–758
  A-law, 750
  AbS, 753
  ADPCM, 750
  ATC, 750
  códigos híbridos, 748, 753, 754
  CELP, 754
  codificadores de fuente, 750–753
  CS-CELP, 754
  de forma de onda, 748, 750
  DPCM, 750
  LPC, 750–753
  μ-law, 750
  SBC, 750
  shorten, XLI, 71, 754–759, 764, 935, 1151
  vocoders, 748, 750
voz, sonidos de, 747

Wagner, ciclo del anillo de, 815
Walsh-Hadamard, transformada, XLVIII, 296, 298, 299, 560, 711, 712, 1079, 1080
Warnock, John (1940–), 919
WAVE, formato de audio, XLI, 731–733
wavelet, descomposición de imágenes
  completa, 596

descomposición de paquetes wavelet adaptativa, 597
estándar, 556, 592, 594
línea, 592
no estándar, 592
pirámide, 556, 594
pirámide Laplaciana, 591
quincunce, 592
simétrica, 591, 634, 635
uniforme, 596
wavelet, transformada de paquetes, 596
wavelets, 278, 279, 298, 545–651
bancos de filtros, 569–577
biortogonales, 571
decimación, 570
derivación de coeficientes de filtro, 576, 577
ortogonales, 571
Beylkin, 580
Coifman, 585
compresión de huellas dactilares, XLVII, 590, 632, 634–638, 1165
Daubechies, 580, 585–590
D4, 571, 577, 578
D8, 1099
descomposición multirresolución, 590, 1158
descomposiciones de imágenes, 590–598
esquema lifting, 598–608, 1156
filtros espejo de cuadratura, 579
Haar, 552–568, 580
Mexicano (sombrero), 547, 548, 550, 552
Morlet, 547, 552
origen del nombre, 547
simétricas, 585
transformada continua, 349, 546–552, 1148
transformada discreta (DWT), XLVII, 349, 578–590, 1149
transformada entera (IWT), 608–610
transformada lenta (lazy), 600
usadas para representación de funciones, 580–584
Vaidyanathan, 585
Wavelets Scalar Quantization (WSQ), XLVII, 533, 632–638, 1165
WavPack, compresión de audio, XLI, 769–778, 1165
Web (sitio) de este libro, XLIII, XLVI, XLIX
web, navegadores
y DjVu, 630
y FABD, 456
y GIF, 231, 264
y PDF, 919
y PNG, 254
y XML, 259
Weierstrass, Karl Theodor Wilhelm (1815–1897), 590
Weighted Finite Automata (WFA), XLVI, 468, 500–512, 1165
Weisstein, Eric W., 490
Welch, Terry A., 203, 264, 1157
WFA, *véase* Weighted Finite Automata
Wheeler, Wayne, V, XL
Whitman, Walt (1819–1892), 260
Whittle, Robin, 758
WHT, *véase* transformada de Walsh-Hadamard
Wilde, Erik, 360
Willems, Frans M. J., LII, 187
Williams, Ross N., 199, 202, 265
Wilson, Sloan (1920–2003), 148
winRAR, XL, 231–234
Wirth, Niklaus (1934–), 493
Wister, Owen (1860–1938), 217
Witten, Ian A., 146
Wolf, Stephan, XLI, XLII, 479
Wright, Ernest Vincent (1872–1939), 147
WSQ, *véase* Wavelets Scalar Quantization
www (web), 264, 344, 747, 1155

Xerox Techbridge (software OCR), 882
xilografía (tallar un grabado en madera), 442
XML, compresión, XMill, XLV, 258–260, 1165

YCbCr, espacio de color, 276, 326, 348, 349, 359, 642, 659
YIQ, modelo de color, 512, 528
Yokoo, Hidetoshi, LII, 227, 229, 867
Yoshizaki, Haruyasu, 180, 235, 1156
YPbPr, espacio de color, 326

zdelta, 925, 926, 1166
Zelazny, Roger (1937–1995), 243
zigzag, secuencia en, 276
en H.261, 701
en H.264, 712
en JPEG, 350, 1081
en MPEG, 682, 695, 1105
en RLE, 36
tridimensional, 938, 939
Zip (software de compresión), 236, 1149, 1166
Ziv, Jacob (1931–), 45, 176, 1156
y patente LZ78, 265
Zurek, Wojciech Hubert (1951–), 58

No existe un indicador de carácter tan seguro como la voz.

—Benjamin Disraeli

Colofón

La primera edición de este libro fue escrita en una explosión de actividad durante el corto período de tiempo: de Junio de 1996, hasta febrero de 1997. La segunda edición fue el resultado de un intenso trabajo durante el segundo semestre de 1998 y el primer semestre de 1999. La tercera edición incluye material escrito, sin prisa, sobre todo a finales de 2002 y principios de 2003. La cuarta edición fue escrita, a un ritmo pausado, durante el primer semestre de 2006. Este libro es la traducción de la cuarta edición, corregida y espero que mejorada. Empezó a ser proyecto serio en abril de 2011, cuando David Salomon tuvo conocimiento del mismo. Es bastante fiel al original, pero no idéntico. El libro original en inglés, fue diseñado por el autor e impreso por él con el sistema de escritura TEX desarrollado por D. Knuth. El texto y las tablas fueron hechas con Textures, una implementación comercial de TEX para el Macintosh. Los diagramas fueron realizados con Adobe Illustrator, también en el Macintosh. Los diagramas que necesitaban cálculos se realizaron con *Mathematica* o *Matlab*, pero incluso ésos fueron "pulidos" en Adobe Illustrator. Los siguientes puntos ilustran la cantidad de trabajo que entró en el libro:

- El libro (incluido el material auxiliar ubicado en el sitio web del libro) contiene alrededor de 523.000 palabras, que suponen alrededor de 3.081.000 caracteres (grande, incluso para los estándares de Marcel Proust). Sin embargo, el tamaño del material auxiliar recogido en el ordenador del autor y en sus estanterías mientras trabajaba en el libro es unas 10 veces más grande que el libro entero. Este material incluye artículos y códigos fuente disponibles en Internet, así como muchas páginas de la información recopilada de varias fuentes.
- El texto está escrito principalmente en fuente cmr10, pero se han utilizado cerca de otras 30 fuentes.
- El archivo de índice bruto tiene alrededor de 4.500 ítems.
- Hay cerca de 1150 referencias cruzadas en el libro.

Los recursos de traductor han sido limitados; pero más que suficientes para llevar a cabo su tarea:

- Una copia en pdf del original (en inglés).
- El paquete gratuíto MiKTEX (versión 2.9, instalación de febrero de 2011), que dispone de lo necesario (incluidos manuales de utilización de los diversos paquetes) para desarrollar textos en TEX, desarrollado por Donald E. Knuth. Se puede obtener vía internet en `http://miktex.org/`.
- El excelente programa LyX —de Matthias Ettrich y el equipo LyX—, también gratuito. Éste permite usar de forma transparente al usuario MiKTEX, y permite la inclusión de código TEX puro. Se puede obtener en la URL `http://www.lyx.org/`.

- El programa *Recortes* que viene con Windows para la captura de las imágenes del pdf original que no podía realizar en TEX, sin falsear los datos originales.
- Los programas: *Paint* (de windows) y *Microsoft Office Picture Manager* (de la office 2007 para hogar y estudiantes), para retocar los recortes de imágenes..
- En muy raras ocasiones, el programa *Gimp* para manipulación de imágenes, disponible de forma gratuita en `http://www.gimp.org/`.
- El programa *FreePascal* —de Bérczi Gábor, Pierre Muller y Peter Vreman— disponible en `http://www.freepascal.org/`. Sólo usado para escribir la mayor parte del código TEX del histograma de la introducción.
- Programas similares al *Matlab* (*wxMaxima*, disponible en `http://andrejv.github.com/wxmaxima/`) y *Mathematica* (Octave, disponible en `http://www.gnu.org/software/octave/`)
- El diccionario de la Real Academia Española (RAE), tanto en su versión escrita, como en internet: `http://www.rae.es/rae.html`.
- El diccionario Collins español-inglés, inglés-español en versión escrita.
- El diccionario de informática y tecnologías afines, de Michel Ginguay.
- Diversos diccionarios en la red:
  - SpanishDict (`http://www.spanishdict.com/`),
  - Urban dictionary (`http://www.urbandictionary.com/`).
  - The free Dictionary (`http://www.thefreedictionary.com/`),
  - WordReference (`http://www.wordreference.com/es/`),
  - Linguee (`http://www.linguee.es/espanol-ingles/`),
  - Dictionary (`http://dictionary.reference.com/`),
- De inestimable ayuda es el traductor de Google (`http://translate.google.es/?hl=es#`), que sirve tanto como para escribir con menos errores, como para obtener sinónimos de las palabras buscadas. (No es recomendable usar sus traducciones sin revisarlas antes.)
- El buscador de Google (`http://www.google.es/`) ha resultado muy útil a la hora de encontrar documentos en internet.

No se puede iniciar un nuevo proyecto en Visual Studio/Delphi/lo que sea, luego añadir en un codificador ADPCM, el mejor modelo psicoacústico, algunos elementos DSP, Levinson Durbin, descomposición de sub-banda, MDCT, un generador de números aleatorios Blum-Blum-Shub, un simulador de movimiento browniano basado en wavelets y un sistema de cifrado de una red de Feistel utilizando un hash criptográfico de la serie Matrix, y esperar echarlo todo por tierra, ¿ahora puede hacerlo?

Anónimo, encontrado en [hydrogenaudio 06]