



UNIVERSIDAD DE ALMERÍA

Departamento de Arquitectura de Computadores y Electrónica

TESIS DOCTORAL

**Compresión Reversible
y
Transmisión de Imágenes**

Vicente González Ruiz

Julio 2000

Dr. Inmaculada García Fernández
Catedrática de Arquitectura de Computadores
de la Universidad de Almería.

CERTIFICA:

Que la memoria titulada “**Compresión Reversible y Transmisión de Imágenes**”, ha sido realizada por D. **Vicente González Ruiz** bajo mi dirección en el Departamento de Arquitectura de Computadores de la Universidad de Almería y concluye la Tesis que presenta para optar al grado de Doctor en Ingeniería Informática.

Almería, 21 de Julio de 2000

Dr. Inmaculada García Fernández
Director de la Tesis

A mi Susana con cariño

*Ciento es que una imagen vale más que mil palabras,
aunque no menos cierto es que ocupa mucha más memoria.*

Agradecimientos

Me hubiera sido del todo imposible sacar adelante este trabajo, sin la ayuda y el apoyo que muchas personas me han brindado.

- A mi esposa Susana, por su amor, comprensión y fe ciega en mí y en todo lo que hago.
A toda mi familia por hacer tres cuartos de lo mismo, especialmente a mis padres, hermanos, Fátima, Pepe, Antonia, Jose y Julia.
 - A mi directora de tesis y compañera de trabajo Inmaculada García, por darme la oportunidad de comenzar a investigar cuando tan sólo era un despistado estudiante. Desde entonces me ha dedicado (casi siempre sin tenerlo) todo el tiempo que ha sido necesario. Por la confianza depositada en mi trabajo y sobre todo, por sus siempre acertadas ideas y consejos.
 - A mis compañeros de trabajo por hacer que el mío sea más llevadero y ameno. Especialmente a Jose Jesús y a Leo, por haberme sacado de tantos “apuros computacionales”. A Jose Jesús además por su amistad, por ser tan buen compañero de despacho y por todo el esfuerzo que ha puesto en que esta tesis sea lo que es.
 - A Roberto Rodríguez y a Javier Díaz, por su ayuda y dedicación durante mi estancia en la Universidad de Santiago de Compostela.
 - A Roland Gredel, a Ulli Thiele y a Enrique de Guindos por enseñarme tantas cosas interesantes acerca de Calar Alto y por hacer posible que mi trabajo tenga una utilidad práctica.
 - A Manu por su desinteresada entrega, por escribir gran parte del software de transmisión progresivo de imágenes y por las largas discusiones científicas que hemos mantenido.
 - A Charo porque me ayudó con el trocito de biología que aparece en la memoria.
 - A Manolo, Jose Luis, Camacho y Rafa por todo cuanto hicieron.
- A todos, mi más sincero agradecimiento.

Glosario de Términos

AOE	Árbol de Orientación Espacial
BPN	<i>Back Propagation Network</i>
bpp	Bits por punto o también <i>bits per pixel</i>
BPS	Bits por símbolo
BWT	<i>Burrows-Wheeler Transform</i>
CALIC	<i>Context-based, Adaptive, Lossless Image Coding</i>
CCITT	<i>International Telephone and Telegraph Consultative Committee</i>
codec	<i>COmpressor/DECompressor</i> o también <i>COder/DECoder</i>
dB	<i>Decibelio</i>
DCT	<i>Discrete Cosine Transform</i>
DWT	<i>Discrete Wavelet Transform</i>
EZW	<i>Embedded Zerotree Wavelet</i>
FCT	<i>Fast Cosine Transform</i>
FELICS	<i>Fast, Efficient, Lossless Image Compression System</i>
FWHT	<i>Fast Walsh-Hadamard Transform</i>
HPF	<i>High Pass Filter</i>
HT	<i>Haar Transform</i>
ITU	<i>International Telecommunications Union</i>
JBIG	<i>Joint Binary Image Group</i>
JPEG	<i>Joint Photographic Experts Group</i>
KLT	<i>Karhunen-Loève Transform</i>
LOCO	<i>LOw COmplexity, context-based, lossless image compression algorithm</i>
LPF	<i>Low Pass Filter</i>
LZ	<i>Lempel-Ziv</i>
LZSS	<i>Lempel-Ziv-Storer-Szymanski</i>
LZW	<i>Lempel-Ziv-Welch</i>
MSE	<i>Mean Square Error</i>
MTF	<i>Move To Front</i>
PBT	<i>Prediction Based Transform</i>
PPM	<i>Prediction by Partial Match</i>
PSNR	<i>Peak Signal to Noise Ratio</i>

RLE	<i>Run Length Encoding</i>
RMSE	<i>Root Mean Square Error</i>
SDPG	Secuencia con Distribución de Probabilidad Geométrica
SNR	<i>Signal to Noise Ratio</i>
SPIHT	<i>Set Partitioning In Hierarchical Trees</i>
(S+P)T	<i>(Sequential + Prediction) Transform</i>
ST	<i>Sequential Transform</i>
TC	Tasa de Compresión
TT	Tasa de Transferencia
U(S+P)T	<i>Unitary (Sequential + Prediction) Transform</i>
UST	<i>Unitary Sequential Transform</i>
WHT	<i>Walsh-Hadamard Transform</i>

Índice General

Prólogo	1
1 Un Marco Teórico para la Compresión de Datos	7
1.1 Teoría de la información	8
1.1.1 Una definición de información	8
1.1.2 Bit de información	9
1.1.3 Fuentes de información	10
1.1.4 Entropía de una fuente de información	10
1.1.5 Eficiencia y redundancia de una fuente de información	11
1.1.6 Extensión de una fuente de información	12
1.2 Teoría de la codificación	12
1.2.1 Elementos de representación	12
1.2.2 Codificación	13
1.2.3 Códigos usados en compresión	13
1.2.4 Códigos de longitud variable	14
1.2.5 La desigualdad de Kraft	15
1.2.6 Códigos prefijo óptimos	16
1.2.7 Eficiencia y redundancia de un código	16
1.3 Comunicación a través de canales discretos sin ruido	17
1.3.1 Capacidad de un canal discreto sin ruido	17
1.3.2 Teorema de la codificación en canales discretos sin ruido	18
1.4 Comprimir = modelar + codificar	19
1.5 Medidas de compresión	20
1.5.1 Bits por símbolo	21
1.5.2 Factor de compresión (X:1)	21
1.5.3 El espacio del rendimiento	21
1.6 Otras aplicaciones de un comprresor de datos	24
1.6.1 Generación de secuencias pseudo-aleatorias	24
1.6.2 Encriptación de datos	25
1.6.3 Detección de fuentes de información	25
1.7 Limitaciones de la compresión de datos	26

1.7.1	El argumento del recuento	26
1.7.2	Los compresores iterativos no existen	26
1.8	El ciclo de la información	27
1.9	Resumen	28
2	Compresión de Texto	31
2.1	La compresión de series	32
2.1.1	RLE estándar	32
2.1.2	RLE binario	33
2.1.3	RLE MNP-5	33
2.2	La compresión basada en diccionarios	33
2.2.1	LZ77	34
2.2.2	LZSS	36
2.2.3	LZ78	36
2.2.4	LZW	40
2.2.5	Consideraciones sobre el uso y la eficiencia	46
2.3	La compresión entrópica	46
2.3.1	Un codificador entrópico universal	47
2.3.2	La codificación de Huffman	49
2.3.3	Compresión adaptativa	51
2.3.4	Modelos inicialmente vacíos	54
2.3.5	El problema de la frecuencia cero	55
2.3.6	Modelos con memoria y la necesidad de códigos mejores	57
2.3.7	La codificación aritmética	58
2.3.8	Codificación de secuencias con una distribución de probabilidad geométrica (SDPG)	67
2.3.9	La codificación binaria ajustada	68
2.3.10	La codificación de Golomb	69
2.3.11	La codificación de Rice	72
2.3.12	Golomb y Rice frente a Huffman	74
2.3.13	Codificación aritmética de una SDPG	74
2.3.14	PPM: predicción usando búsqueda parcial de cadenas	74
2.3.15	BWT: La transformada de Burrows-Wheeler	77
2.3.16	La codificación mover-al-frente: MTF	81
2.3.17	PBT: La transformada de texto basada en predicción	83
2.3.18	BWT frente a PBT	92
2.3.19	PPM frente a BWT y PBT	93
2.4	Evaluación	94
2.5	Resumen	107

3 Compresión de Imágenes	109
3.1 JPEG	110
3.2 FELICS	116
3.3 LOCO-I	119
3.4 CALIC	123
3.5 BPN+aritha	129
3.6 Evaluación	132
3.7 Resumen	136
4 Transmisión de Imágenes	139
4.1 Requisitos del codec progresivo	140
4.2 Dominios de representación	141
4.3 La transformada de Walsh-Hadamard	142
4.4 La transformada coseno	144
4.5 Selección de los coeficientes espectrales	145
4.6 Codificación de los espectros	157
4.7 Requisitos de una nueva transformación	164
4.8 La transformada wavelet discreta	170
4.9 La transformada S	175
4.9.1 La transformada S unitaria	177
4.10 La transformada S+P	177
4.11 Evaluación de las transformadas en transmisión progresiva	180
4.12 La transformada wavelet para un número de muestras distinto de 2^n	183
4.13 Transmisión por planos de bits	185
4.14 Compresión de los planos de bits	188
4.14.1 Un método de compresión	189
4.14.2 Bits de significancia y de refinamiento	190
4.14.3 Acerca del bit de signo	191
4.15 Un codec progresivo de coeficientes wavelet: SPIHT	192
4.15.1 Un poco de historia	192
4.15.2 Funcionamiento de SPIHT	193
4.15.3 Detalles de la implementación	194
4.15.4 SPIHT + codificación aritmética	198
4.15.5 SPIHT sobre imágenes con cualquier tamaño	199
4.16 Evaluación	200
4.16.1 Evaluación como compresor	200
4.16.2 Evaluación como transmisor	202
4.17 Sobre la implementación <i>software</i> del transmisor	236
4.18 Resumen	236
Conclusiones y Principales Aportaciones	239

Perspectivas futuras	241
A Imágenes de Prueba	243
B Demostraciones	253
B.1 Algoritmo rápido para el cálculo de la WHT	253
B.2 Algoritmo rápido para el cálculo de la DCT	254
Bibliografía	257

Índice de Figuras

1.1	Paradigma del compresor/descompresor de Shannon	20
1.2	El espacio del rendimiento de los compresores de datos	23
1.3	El ciclo de la información	28
2.1	Codificación RLE-MNP5: algunos ejemplos	34
2.2	Ejemplo de codificación usando LZ77	35
2.3	Ejemplo de descodificación usando LZ77	35
2.4	Ejemplo de codificación usando LZSS	36
2.5	Ejemplo de descodificación usando LZSS	37
2.6	Ejemplo de codificación usando LZ78	38
2.7	Ejemplo de diccionario LZ78	39
2.8	Ejemplo de descodificación usando LZ78	40
2.9	Ejemplo codificación LZW	41
2.10	Ejemplo de contenido de diccionario LZW	42
2.11	Ejemplo de descodificación usando LZW	45
2.12	Modelo de codificación/descodificación entrópica	47
2.13	Ejemplo de construcción de un árbol de Huffman	50
2.14	Ejemplo de actualización de un árbol de Huffman	53
2.15	Ejemplo de inserción en un árbol de Huffman	56
2.16	Comparativa Huffman vs código ideal	58
2.17	Intervalos generados por la codificación del fichero <code>aab</code>	60
2.18	Ejemplo de codificación aritmética	62
2.19	Ejemplo de histograma de una SDPG	67
2.20	Modelo de compresión multietapa	68
2.21	Ejemplo de codificación PPMC	77
2.22	Ejemplo de descodificación PPMC	78
2.23	Ejemplo de matriz inicial para la BWT	79
2.24	Ejemplo de matriz ordenada para la BWT y selección del índice i	80
2.25	Ejemplo de codificación MTF	82
2.26	Ejemplo de descodificación MTF	83
2.27	Ejemplo de codificación MTF mejorada	85
2.28	Ejemplo de descodificación MTF mejorada	85

2.29 Ejemplo de codificación PBT	87
2.30 Ejemplo de descodificación PBT	88
2.31 Entropías parciales de “lena” y aplicando la PBT	90
2.32 Detalle de las entropías parciales para “lena” aplicando la PBT	91
2.33 Rendimiento de los algoritmos de texto: imágenes de 8 bpp	106
2.34 Rendimiento de los algoritmos de texto: imágenes de 16 bpp	107
 3.1 Codec sin pérdida de información propuesto por JPEG	110
3.2 Contexto de predicción espacial usado por JPEG	111
3.3 Histogramas de las imágenes residuo de 8 bits	112
3.4 Histogramas de las imágenes residuo de 16 bits	112
3.5 Contexto de predicción espacial usado por FELICS	116
3.6 Modelo probabilístico usado en FELICS	117
3.7 Contexto de predicción espacial usado por LOCO-I	120
3.8 Contexto de predicción espacial usado por CALIC	124
3.9 Contexto de predicción espacial usado por BPN	129
3.10 Arquitectura de la BPN	131
3.11 Rendimiento de los compresores para las imágenes de 8 bpp	134
3.12 Rendimiento de los compresores para las imágenes de 16 bpp	135
 4.1 Espectros WHT y DCT de “lena” y “B0100”	146
4.2 Políticas de transmisión incremental de coeficientes	147
4.3 Transmisión progresiva de “lena” usando la WHT	148
4.4 Transmisión progresiva de “lena” usando la WHT (cont.)	149
4.5 Transmisión progresiva de “lena” usando la DCT	150
4.6 Transmisión progresiva de “lena” usando la DCT (cont.)	151
4.7 Transmisión progresiva de “B0100” usando la WHT	152
4.8 Transmisión progresiva de “B0100” usando la WHT (cont.)	153
4.9 Transmisión progresiva de “B0100” usando la DCT	154
4.10 Transmisión progresiva de “B0100” usando la DCT (cont.)	155
4.11 Coeficientes WHT enviados con el filtrado progresivo de “lena”	159
4.12 Coeficientes DCT enviados con el filtrado progresivo de “lena”	160
4.13 Coeficientes WHT enviados con el filtrado progresivo de “B0100”	161
4.14 Coeficientes DCT enviados con el filtrado progresivo de “B0100”	162
4.15 Signos de los coeficientes WHT y DCT	163
4.16 Magnitud de los coeficientes WHT y DCT para “lena” (filtro triangular) . .	165
4.17 Magnitud de los coeficientes WHT y DCT para “lena” (filtro cuadrado) . .	166
4.18 Magnitud de los coeficientes WHT y DCT para “B0100” (filtro triangular)	167
4.19 Magnitud de los coeficientes WHT y DCT para “B0100” (filtro cuadrado) .	168
4.20 Descomposición frecuencial usando QMFs	171
4.21 Respuesta en frecuencia de los filtros de Haar	172
4.22 Cálculo de la DWT (<i>dyadic</i>)	172

4.23 Wavelets de Haar	173
4.24 Funciones base (wavelets) de la transformada de Haar	174
4.25 Descomposición espectral de la DWT (<i>dyadic</i>)	175
4.26 Cálculo de la DWT (<i>dyadic</i>) bidimensional	176
4.27 ST frente a UST	178
4.28 Factores de la transformada S unitaria	179
4.29 Transmisión progresiva de “lena” usando la UST y la $U(S+P)T$	181
4.30 Transmisión progresiva de “B0100” usando la UST y la $U(S+P)T$	182
4.31 Transmisión progresiva de “lena” por planos	186
4.32 Transmisión progresiva de “B0100” por planos	187
4.33 Árboles creados en la DWT-2D	189
4.34 Un ejemplo de planos de bits	190
4.35 Cuarto plano de significancia y de refinamiento para “lena”	191
4.36 Bits de signo de “lena” y “B0100”	192
4.37 Ejemplo de descomposición wavelet 2D de lado cualquiera	199
4.38 Rendimiento de los transmisores para las imágenes de 8 bpp	203
4.39 Rendimiento de los transmisores para las imágenes de 16 bpp	204
4.40 Transmisión de “lena” en PSNR	205
4.41 Transmisión de “bárbara” en PSNR	205
4.42 Transmisión de “boats” en PSNR	206
4.43 Transmisión de “zelda” en PSNR	206
4.44 Transmisión de “B0100” en PSNR	207
4.45 Transmisión de “29jul24” en PSNR	207
4.46 Transmisión de “29jul25” en PSNR	208
4.47 Transmisión de “29jul26” en PSNR	208
4.48 Comparación entre refinar antes o después (“lena”)	210
4.49 Comparación entre refinar antes o después (“B0100”)	211
4.50 Transmisión progresiva de “lena”	212
4.51 Transmisión progresiva de “bárbara”	213
4.52 Transmisión progresiva de “boats”	214
4.53 Transmisión progresiva de “zelda”	215
4.54 Transmisión progresiva de “B0100”	216
4.55 Transmisión progresiva de “29jul24”	217
4.56 Transmisión progresiva de “29jul25”	218
4.57 Transmisión progresiva de “29jul25” (cont.)	219
4.58 Transmisión progresiva de “29jul25” (cont.)	220
4.59 Transmisión progresiva de “29jul25” (cont.)	221
4.60 Transmisión progresiva de “29jul25” (cont.)	222
4.61 Transmisión progresiva de “29jul25” (cont.)	223
4.62 Transmisión progresiva de “29jul26”	224
4.63 Transmisión progresiva de “29jul26” (cont.)	225
4.64 Transmisión progresiva de “29jul26” (cont.)	226

4.65 Transmisión progresiva de “29jul26” (cont.)	227
4.66 Transmisión progresiva de “29jul26” (cont.)	228
4.67 Transmisión progresiva de “29jul26” (cont.)	229
4.68 Comparativa entre JPEG y SPIHT. “lena”	231
4.69 Comparativa entre JPEG y SPIHT. “bárbara”	232
4.70 Comparativa entre JPEG y SPIHT. “boats”	233
4.71 Comparativa entre JPEG y SPIHT. “zelda”	234
4.72 Comparativa entre JPEG y SPIHT (16 bpp)	235
 A.1 Imagen “lena”	244
A.2 Imagen “bárbara”	245
A.3 Imagen “boats”	246
A.4 Imagen “zelda”	247
A.5 Imagen “B0100”	248
A.6 Imagen “29jul24”	249
A.7 Imagen “29jul25”	250
A.8 Imagen “29jul26”	251
 B.1 Algoritmo <i>bit-reversal</i>	255

Índice de Tablas

1.1	Ejemplos con diferentes tipos de códigos descodificables	14
2.1	Probabilidades asociadas a los símbolos del fichero <i>aab</i>	59
2.2	Códigos de Golomb y de Rice para los primeros símbolos	71
2.3	Entropía del error de predicción usando MTF y PBT	90
2.4	Número de contextos generados y memoria consumida por PBT	92
2.5	Entropías arrojadas por BWT+MTF vs PBT para las imágenes de prueba .	93
2.6	Tasas de compresión texto para las imágenes de 8bpp	102
2.7	Tasas de transferencia texto para las imágenes de 8 bpp	103
2.8	Tasas de compresión texto para las imágenes de 16 bits	104
2.9	Tasas de transferencia texto para las imágenes 16 bpp	105
3.1	Predictores utilizados por JPEG	110
3.2	Categorías de diferencia usadas en la codificación de Huffman (JPEG) . .	113
3.3	Tabla de códigos de Huffman para luminancia DC	113
3.4	Entropías del error de predicción de JPEG, CALIC y BPN	131
3.5	Tasas de compresión secuencial de imágenes (8 bpp)	133
3.6	Tasas de transferencia secuencial de imágenes (8 bpp)	134
3.7	Tasas de compresión secuencial de imágenes (16 bpp)	135
3.8	Tasas de transferencia secuencial de imágenes (16 bpp)	136
4.1	WHT vs DCT. Reconstrucciones de “lena”	156
4.2	WHT vs DCT. Reconstrucciones de “B0100”	157
4.3	WHT vs DCT vs UST vs U(S+P)T. Reconstrucciones de “lena”	183
4.4	WHT vs DCT vs UST vs U(S+P)T. Reconstrucciones de “B0100”	184
4.5	Tasas de compresión de SPIHT para las imágenes de 8 bpp	201
4.6	Tasas de transferencia de SPIHT para las imágenes de 8 bits	201
4.7	Tasas de compresión de SPIHT para las imágenes de 16 bits	201
4.8	Tasas de transferencia de SPIHT para las imágenes de 16 bits	202
A.1	Algunos parámetros interesantes de las imágenes de prueba	243

Prólogo

La compresión de datos es el proceso mediante el cual se consigue representar cierto volumen de información usando la mínima cantidad de datos. Este tipo de transformaciones son útiles porque permiten ahorrar en recursos de almacenamiento y de transmisión. Los compresores de datos son codificadores universales, capaces de comprimir cualquier secuencia de datos si en ella se detecta redundancia estadística.

La compresión de imágenes es un proceso semejante, pero los datos comprimidos siempre son representaciones digitales de señales bidimensionales. Los compresores de imágenes explotan, además de la redundancia estadística, la redundancia espacial. De esta forma, los niveles de compresión que se pueden conseguir aumentan sensiblemente.

Incluso comprimidas, las imágenes digitales requieren grandes cantidades de memoria. Debido a este factor, existen básicamente dos tipos de compresores de imágenes: (1) compresores reversibles o compresores sin pérdidas (*lossless compressors*) y (2) compresores irreversibles o compresores con pérdidas (*lossy compressors*). Estos últimos, a costa de eliminar la información menos relevante para el observador, alcanzan cotas de compresión muy superiores. El mejor ejemplo lo tenemos en el estándar de compresión de imágenes JPEG.

Sin embargo, existen muchas situaciones en las que no es aceptable ningún deterioro de la imagen, por ejemplo en medicina o en astronomía. Esta restricción viene impuesta porque a priori toda la información contenida, incluso la que es considerada como ruido, se considera imprescindible. Precisamente por la existencia de este ruido, los niveles de compresión sin pérdidas esperables son en general bastante modestos. Típicamente, los ficheros comprimidos son la mitad de largos que el fichero original.

En muchos de los marcos de trabajo, las necesidades de compresión provienen principalmente de problemas de transmisión y manipulación de la información, y en menos ocasiones por problemas de almacenamiento. Esta situación es especialmente cierta en el caso de las imágenes. Evidentemente, almacenar el doble de imágenes ya es una gran ventaja, pero donde realmente podemos sacar provecho de su compresión es en la transmisión ya que transmitir una imagen comprimida puede dividir el tiempo por la mitad, lo que sin duda constituye un gran ahorro. Esto es especialmente cierto cuando se transmite a través de canales lentos como puede ser *Internet*.

La teleobservación astronómica es un buen ejemplo de ésto. Los lugares de obtención de las imágenes están situados en altas montañas o en el espacio (telescopio Hubble), por

tanto, casi siempre alejados de los naturales centros de trabajo e investigación. El tamaño de las imágenes astronómicas provoca que prácticamente sea imposible ver en tiempo real lo que el telescopio está captando en ese instante, lo que dificulta enormemente las tareas de telecontrol.

Usar algoritmos *lossy* puede resolver el problema, pero sólo parcialmente. En primer lugar, la mayoría de ellos están orientados a conseguir la mínima distorsión posible a un determinado nivel de compresión, sin embargo, a pesar de que una compresión de datos *lossy* disminuye el tiempo total de transmisión, en general los algoritmos existentes no permite visualizar globalmente la imagen transmitida hasta que el proceso de descompresión ha finalizado completamente. En segundo lugar, la imagen nunca es idéntica a la original, lo que puede ser totalmente inaceptable dependiendo del uso que se vaya a hacer de ella.

La mejor forma de resolver estos problemas es usar algoritmos de compresión que sean progresivos y reversibles. Con este tipo de compresores/descompresores una imagen aproximada llega hasta el receptor en el mínimo tiempo posible (en general muy inferior al tiempo empleado por un compresor *lossy*) y aunque ésto ya es una gran ganancia, la principal ventaja proviene de que la imagen mejora en calidad con el transcurso del tiempo. Así, en cualquier instante, la transmisión puede interrumpirse si se considera que la calidad de la imagen recibida es suficiente. Si la interrupción no se produce, finalmente la imagen original es reconstruida.

Es evidente que este tipo de tecnología tiene grandes e importantes aplicaciones. En este sentido cabe destacar que el nuevo estándar de compresión de imágenes estáticas JPEG 2000 (aún en proceso de diseño) es un compresor progresivo. Cuando esta tecnología haya sido implantada en *Internet*, la descarga de las páginas *html* con imágenes del tipo “mapa de bits” será mucho más rápida. De hecho las imágenes se irán haciendo reconocibles simultáneamente con el texto. Además, durante el transcurso de la visita las imágenes ganarán en calidad. De esta forma si la página no es interesante no tenemos que esperar a que toda la información visual sea descargada, lo que generará un ahorro importante de ancho de banda e *Internet* será más rápida.

En dicha línea, este trabajo trata de hacerse eco de las principales tendencias en el campo de la compresión de datos, la compresión *lossless* de imágenes y la transmisión progresiva de imágenes. Con ese objetivo se ha dedicado un esfuerzo considerable a realizar un “review” lo más amplio posible de todas las técnicas de compresión sin pérdidas que pueden ser aplicadas en el campo de la compresión y transmisión de imágenes. Consideramos que una visión general es fundamental de cara a elegir la línea de trabajo actual y futuro más correcta. Consecuentemente, la tesis se ha dividido en cuatro capítulos:

- En el Capítulo 1, titulado “Un Marco Teórico para la Compresión de Datos”, se introduce un número importante de conceptos y definiciones que van a ser usados en el resto de los capítulos. Sitúa la compresión de datos en el contexto más general de la teoría de la información y de la codificación. En él, se exponen los principios en los que se fundamenta la compresión de datos, el porqué es posible comprimir, cuál debe ser la arquitectura del proceso de compresión y de descompresión, cuánto es

posible comprimir y cómo medir la bondad de los diferentes sistemas de compresión. Se razona porqué no existen medidas absolutas y se propone una medida llamada “espacio del rendimiento” que se utilizará en el resto de la tesis para comparar los algoritmos de compresión estudiados.

- El Capítulo 2, titulado “Compresión de Texto” se centra en la exposición y evaluación de las técnicas más importantes de compresión de datos, donde “datos” puede ser cualquier representación de la información. Los compresores de texto tratan la secuencia de datos a comprimir como una secuencia unidimensional, sobre la que no se hacen suposiciones excepto que puede contener redundancia estadística. Este tipo de redundancia se refiere a que la ocurrencia de un símbolo, dentro de la secuencia, sea en general más probable que la de otro símbolo y que en la mayoría de los casos se puede aplicar algún método que utilice el contexto (anterior o posterior) para efectuar una predicción acerca de dicho símbolo. En general, después de un proceso de predicción se usa un codificador entrópico para representar el error de predicción.

Aunque la mayoría del capítulo se dedica a presentar algoritmos que explotan la idea de “la compresión entrópica”, existen dos grandes apartados más en los que se estudian los métodos para “la compresión de series” (RLE) y “la compresión basada en diccionarios” o “métodos Lempel-Ziv” (LZ). De esta forma se han analizado la mayoría de los algoritmos de compresión de texto conocidos.

Finalmente, se evalúan todos los métodos de compresión que aparecen descritos a lo largo del capítulo. Entre ellos cabría destacar al algoritmo “BPT” que aunque no puede decirse que es una idea original (porque ya Shannon la usó en sus famosos experimentos de la medida de la entropía del idioma inglés), sí es la primera vez que se implementa y se evalúa como compresor de imágenes, resultando que es el compresor de texto que mejor se adapta a este propósito para una determinada clase de imágenes. Se ha usado una batería de imágenes formada por 4 imágenes estándar de tono continuo de 8 bits/punto y por 4 imágenes astronómicas de 16 bits/punto. Estas últimas no pueden ser consideradas como ejemplos típicos de imágenes de tono continuo debido a la baja correlación espacial que presentan. Todas las imágenes aparecen descritas en el Apéndice A.

La razón por la que se aplican los compresores de texto directamente a las imágenes de prueba (a pesar de no estar especializados en este tipo de fuentes) es que la correlación espacial provoca también una correlación estadística. Además, como veremos en los siguientes capítulos, el resto de compresores específicos son muy semejantes a los compresores de texto y por este motivo deben ser estudiados previamente.

- El Capítulo 3, llamado “Compresión de Imágenes” analiza los compresores reversibles de imágenes que son “el estado del arte” en compresión secuencial de imágenes. La misión de estos algoritmos es la de alcanzar la tasa de compresión más alta en el mínimo tiempo posible. Básicamente son adaptaciones de algún algoritmo de

compresión de texto (PPM casi siempre) en el que el modelo probabilístico se ha sustituido por un predictor espacial.

Los métodos analizados son: JPEG, FELICS, LOCO-I, CALIC y BPN. Este último es un desarrollo original que trata de estudiar la capacidad de aprendizaje de una red neuronal cuando se entrena como predictor espacial. El algoritmo de aprendizaje es el conocido método de retropropagación del error y de ahí el nombre de nuestro método: *Back Propagation Network*. Los resultados (usando las mismas imágenes de prueba que fueron usadas en el capítulo anterior) confirman que una red neuronal puede ser un predictor bastante bueno, especialmente si la imagen es suficientemente compleja (existe correlación aunque no fundamentalmente lineal).

Como era de esperar, los compresores de imágenes mejoran las tasas de compresión de los compresores de texto, aunque no espectacularmente. Quizás el avance más importante se dirige en el camino de la reducción de los tiempos de compresión y de descompresión.

- El Capítulo 4, titulado “Transmisión de Imágenes”, describe el trabajo desarrollado para adaptar los conceptos y modelos descritos, al problema de la transmisión progresiva de imágenes. Dicho trabajo ha sido en parte motivado por la colaboración científica entre el grupo de investigación **Supercomputación: Algoritmos** de la Universidad de Almería y el **Centro Astronómico Hispano-Alemán de Calar Alto**.

La resolución del problema de transmisión progresiva de imágenes desemboca en la implementación de un codificador/descodificador progresivo de imágenes que además es un excelente compresor. La clave de esta tecnología está en aplicar algún tipo de transformación que nos permita encontrar aquella información visual que sea más importante, que será la que se transmitirá primero. En este contexto se analiza el uso de cuatro transformadas: la transformada de Walsh-Hadamard, la transformada coseno, la transformada S y la transformada S+P. Las dos primeras son transformaciones clásicas, diseñadas a partir de bases periódicas. Las dos últimas son transformadas wavelet reversibles.

En la segunda parte del capítulo se analiza un codificador progresivo de coeficientes wavelet llamado SPIHT. Este método tiene varias alternativas de diseño y todas son estudiadas. La eficiencia conjunta de SPIHT y la transformada S+P es excelente. Los resultados, especialmente en el campo de la televisualización de imágenes, son bastante alentadores. Por ejemplo, usando imágenes astronómicas típicas podemos esperar tasas útiles de compresión de 50000:1, lo que significa que el astrónomo puede comenzar a considerar si la imagen captada es reconocible o utilizable en un tiempo 50000 veces inferior al tiempo que emplearía si la imagen no fuera codificada. Éste es realmente el factor determinante en el diseño de sistemas de telecontrol de telescopios. Durante todo el proceso de transmisión, la imagen mejora en calidad de forma continua. Además, el algoritmo de compresión es totalmente reversible, lo

que significa que si el astrónomo decide que la imagen es interesante, sólo tiene que esperar hasta el final de su transmisión para obtenerla. Como la imagen transmitida está comprimida, el tiempo total de transmisión también es minimizado.

Este documento finaliza con un resumen de las principales aportaciones realizadas, las conclusiones y las líneas de trabajo futuro más interesantes.

Indicar por último que se han incluido dos apéndices que contienen información sobre las imágenes que han servido para evaluar los algoritmos y algunos aspectos algorítmicos relacionados con las transformadas de Walsh-Hadamard y coseno.

Capítulo 1

Un Marco Teórico para la Compresión de Datos

Claude E. Shannon trabajaba como matemático a finales de los años 40 en los laboratorios Bell cuando se planteó, desde un punto de vista teórico, el problema de la transmisión de información a través de canales de comunicación digitales. Su trabajo proporcionó un gran avance a las teorías de la comunicación y de la información, las cuales tienen dos objetivos principales: (1) la transmisión de la máxima cantidad de información posible a través de un canal y (2) la detección y corrección de errores de transmisión debido a ruidos en dicho canal. Claramente, ambos objetivos persiguen a su vez el abaratamiento de los costes y la seguridad en la transmisión de la información.

El interés por la teoría de la comunicación, desde el punto de vista de la compresión de datos, se centra principalmente en la codificación eficiente de la información en ausencia de ruido. En estas circunstancias, la meta es representar la información de la forma más compacta posible. Shannon definió numerosos conceptos y medidas que se utilizan en todos los compresores de datos actuales, y de ellos habla este capítulo. Además, entre otros, desarrolló un importante teorema matemático conocido con el nombre de primer teorema de Shannon o teorema de la codificación sin ruido [109] que básicamente define que siempre es posible encontrar un código que aproveche al máximo la capacidad de un canal de transmisión en ausencia de ruido.

Dicho teorema junto con la codificación de Huffman [49], constituyeron el estado del arte hasta principios de los 80,¹ momento en el que Rissanen y Langdon diseñaron una implementación asequible de la codificación aritmética para alfabetos binarios [88]. Poco tiempo después, Witten, Bell y Cleary hicieron lo propio para alfabetos de 256 símbolos [128]. Con este nuevo código, el teorema de la codificación sin ruido perdió cierta relevancia en el mundo de la compresión de datos, aunque todavía es un aspecto muy

¹Aunque ya desde los años 60 se conocía que podría calcularse un código 100% eficiente. De hecho, Solomon W. Golomb, en su artículo titulado *Run-Length Encodings* [32] y publicado en 1965 se refería a este aspecto.

importante a tener en cuenta si se desean utilizar codificadores rápidos.

1.1 Teoría de la información

Probablemente, la teoría de la información sea una de las teorías (matemáticas o no) más complejas de definir y delimitar, debido a que es muy complicado definir qué es la información. Así, podemos encontrar autores (entre ellos a Cover [21]) que dentro de ella entienden, por ejemplo, la teoría de la comunicación. Sin embargo, hay otros autores (sobre todo, los ingenieros en comunicaciones y Shannon el primero) que piensan lo contrario. Nosotros vamos a tomar la segunda alternativa, separando la información de su transmisión, con la idea de estructurar mejor este documento.

En general, podemos decir que la teoría de la información proporciona una formulación precisa para un gran número de problemas de transmisión. Sus repercusiones alcanzan la física, la lingüística, la psicología, la medicina, la informática, etc. y uno de sus principales resultados prácticos fue el análisis de la redundancia en los idiomas [108] y la síntesis del lenguaje escrito [109]. Junto con la creación de códigos compresores de datos, la teoría de la información ha contribuido también a diseñar códigos detectores y correctores de errores, utilizados en entornos de transmisión ruidosos, y códigos encriptadores.

1.1.1 Una definición de información

La información es un concepto primitivo y de complicada definición. En general, puede decirse que es todo aquello que tiene asociado algún significado. Aparte de esta definición y de poner un sinfín de ejemplos, sólo podemos añadir que es necesario representar la información de alguna forma para poder almacenarla o transmitirla. Decimos entonces que la información se almacena o transmite mediante mensajes. Por ejemplo, ahora, durante la lectura de este documento, se está produciendo una transmisión de cierto volumen de información desde el documento hasta su mente. El texto y figuras presentadas constituyen el mensaje y lo que usted comprende durante su lectura es (al menos una parte de) la información contenida.

Para que un mensaje contenga información, es preciso que aporte algo nuevo o imprevisto. El nivel de significado depende del contexto y del sujeto que asimila la información. Por ejemplo, si una persona a las 12 del medio día nos dice: “ahora no es de noche”, seguramente no nos proporcionará ninguna sorpresa pues sólo ha dicho algo que es evidente. Sin embargo si dijera: “ahora es de noche”, nos estaría proporcionando suficiente información como para intuir que ha ocurrido, por ejemplo, un eclipse solar y esto sí es algo imprevisto (si no lo esperábamos, claro).

Por tanto, la información aportada por un mensaje parece medirse por el carácter más o menos previsible del acontecimiento expresado. En este sentido entra en juego un importante factor subjetivo: un mensaje puede producir en diferentes sujetos reacciones distintas y, por consiguiente, diferente nivel de información. Sin embargo, considerar estos factores desde un punto de vista matemático complicaría enormemente la teoría y por

esta razón, prescindiremos del significado de los mensajes y de la subjetividad de los sujetos destino, para fijarnos exclusivamente en la probabilidad de que se produzca un determinado mensaje o suceso.

En un proceso de transmisión de información existen tres elementos básicos: (1) el emisor de información, (2) el canal de comunicación y (3) el receptor de información. La información que fluye desde el emisor hasta el receptor, pasando por el canal, puede verse alterada en el caso de que el canal de comunicación no esté exento de ruido.

En el contexto de la compresión de datos, como ya hemos indicado, el canal se supone sin ruido y por tanto, definiremos la información $I(a)$ asociada al suceso a como el logaritmo del inverso de la probabilidad de que el suceso a ocurra. Por tanto,

$$I(a) = \log \frac{1}{p(a)} = -\log p(a). \quad (1.1)$$

Como podemos apreciar, la definición dada de información es coherente si tenemos en cuenta que:

1. Si $p(a) = 1$ (el suceso a siempre ocurre), entonces $I(a) = 0$ (la ocurrencia del proceso no proporciona al sujeto ninguna información).
2. Si $p(a) \rightarrow 0$ (el suceso a casi nunca ocurre), entonces $I(a) \rightarrow \infty$ (la ocurrencia del proceso proporciona mucha información).

1.1.2 Bit de información

La información se puede cuantificar. Cuando la base del logaritmo es 2 en la Expresión (1.1), la información se mide en bits de información. Por tanto, un bit de información se define como la cantidad de información que se genera al realizarse una elección entre dos posibles alternativas equiprobables. Si llamamos a estos sucesos a_0 y a_1 , tenemos que $I(a_0) = I(a_1) = 1$ bit de información. Nótese que es imposible definir una cantidad de información más elemental. Por ejemplo, un proceso de generación de información en el que en cada iteración se produce aproximadamente un bit de información es el nacimiento de un niño o de una niña.

Debe tenerse en cuenta que la elección de la base del logaritmo no determina el número de sucesos que tenemos para seleccionar. Cuando el número de alternativas sea mayor, el volumen de información producida en una elección puede superar la cantidad de 1 bit de información. De igual manera, si la probabilidad asociada a un evento es muy baja, independientemente del número de elecciones posibles, la cantidad de información asociada (a su elección) puede superar la cantidad de 1 bit de información.

En el resto del documento mediremos la información en bits de información y por lo tanto, la base de todos los logaritmos relacionados con esta medida será dos.

1.1.3 Fuentes de información

Una fuente de información discreta es aquella que produce mensajes como sucesiones de símbolos-fuente (a las que llamaremos en general secuencias-fuente) extraídos de entre una colección finita de símbolos-fuente que conforman un alfabeto-fuente. La notación

$$A = \{a_1, \dots, a_r\},$$

indica un alfabeto-fuente A compuesto de r símbolos-fuente a_i . A r se le llama base de la fuente de información discreta. Nótese que hablar de bits de información no significa tomar $r = 2$.

En el ámbito de la teoría de la información, la emisión de símbolos para la construcción de mensajes se considera gobernada por un proceso estocástico, lo que quiere decir que la fuente de información produce los símbolos-fuente de acuerdo con unas ciertas probabilidades

$$P = \{p(a_1), \dots, p(a_r)\}.$$

La probabilidad asociada a un símbolo-fuente debe ser mayor que cero, para que el símbolo-fuente tenga razón de existir como tal en el alfabeto-fuente (o de lo contrario nunca formaría parte de ninguna secuencia-fuente) e inferior a uno, para que existan al menos dos símbolos-fuente diferentes en el alfabeto-fuente.

Además se cumple que, teniendo una muestra (mensaje) razonablemente grande de la fuente de información, dicha muestra es representativa (estadísticamente hablando) del conjunto de todas las cadenas o secuencias de Markov que pueden ser producidas por la fuente, y por esta razón, diremos que se trata de un “proceso ergódico”.² Gracias a esta propiedad, el famoso experimento realizado por Shannon que consistió en sintetizar mensajes escritos en inglés utilizando una fuente de este tipo, tras haber analizado suficientes secuencias de texto escrito, fue un éxito.

Los procesos de Markov nos hablan acerca de fuentes de información que generan los símbolos-fuente con una probabilidad que puede depender o no de los símbolos anteriormente producidos. Cuando, la probabilidad de un símbolo-fuente no depende del previamente emitido, hablamos de fuentes sin memoria, de fuentes de memoria nula o de fuentes de Markov de orden 0. Cuando ocurre lo contrario, hablamos de fuentes con memoria o de fuentes de Markov de orden K , donde K es el número de símbolos-fuente que forman el contexto que determina la probabilidad de generación del siguiente símbolo-fuente.

1.1.4 Entropía de una fuente de información

La entropía $H(A)$ de una fuente de información discreta sin memoria A es una medida de la cantidad de información que produce la fuente de información. Concretamente, la entropía es la cantidad de información producida por la fuente en promedio. Si tenemos

²Una fuente de información se dice ergódica si una muestra suficientemente corta de ella es representativa (estadísticamente hablando) de toda la información que es capaz de emitir.

en cuenta que la probabilidad (información) asociada a cada símbolo-fuente puede ser distinta, la media puede calcularse a partir de las probabilidades de los símbolos-fuente mediante la expresión

$$H(A) = \sum_{i=1}^r p(a_i) \times I(a_i) = - \sum_{i=1}^r p(a_i) \times \log_2 p(a_i). \quad (1.2)$$

Debido a la base tomada en el logaritmo (dos), la entropía se calcula en bits de información/símbolo-fuente. Cuando en general, la base del logaritmo es x , hablamos de $H_x(A)$ y ya no se mide en bits de información/símbolo-fuente. Concretamente, cuando $x = e$ (el número e), la información se mide en Nats y cuando $x = 10$ se mide en Hartleys.

1.1.5 Eficiencia y redundancia de una fuente de información

La entropía $H(A)$ de una fuente A con r símbolos-fuente se maximiza cuando

$$p(a_i) = 1/r, \quad \forall a_i \in A.$$

En este caso

$$H(A) = \log_2 r.$$

Se define la entropía relativa o eficacia de la fuente como la razón entre la entropía real y la máxima entropía de la fuente

$$E(A) = \frac{H(A)}{\log_2 r}. \quad (1.3)$$

Nótese que $0 < E(A) \leq 1$.

Por ejemplo, si la entropía relativa de una fuente es de 0.8, esto puede interpretarse como que ha existido sólo un 80% de libertad de elección de los símbolos de la fuente, si la comparamos con el máximo grado de libertad posible.

Se define además la redundancia de una fuente de información como

$$R(A) = 1 - E(A). \quad (1.4)$$

El concepto de redundancia de la fuente está ligado al exceso de terminología por parte de una fuente para producir mensajes que contienen una cierta cantidad de información. Una fuente más redundante que otra necesitará producir mensajes más largos para transmitir la misma cantidad de información. Shannon, en su trabajo acerca de la redundancia de los idiomas, encontró que aproximadamente el 50% de los caracteres en el idioma inglés son superfluos. En consecuencia, la mitad de las letras o palabras que se utilizan en éste lenguaje se escogen libremente y el resto están determinadas por la estructura estadística del lenguaje. En otras palabras, podrían expresarse las mismas ideas con la mitad de texto escrito aproximadamente.

1.1.6 Extensión de una fuente de información

La extensión de una fuente es el nombre que recibe el proceso de construir una fuente llamada fuente extendida a partir de una fuente de información, generando cada símbolo-fuente extendido como la concatenación de dos o más símbolos-fuente. Como consecuencia, la base de la fuente extendida es r^N , donde N es el orden de la extensión y r es el tamaño del alfabeto. El objetivo de extender una fuente es aumentar suficientemente el número de símbolos-fuente. Como veremos, haciendo esto es posible encontrar representaciones más eficientes para las secuencias-fuente originales. Por ejemplo, si el alfabeto-fuente $A = \{a_1, a_2, a_3\}$, el alfabeto-fuente extendido A^2 de orden 2 sería

$$A^2 = \{a_1a_1, a_1a_2, a_1a_3, a_2a_1, a_2a_2, a_2a_3, a_3a_1, a_3a_2, a_3a_3\}.$$

1.2 Teoría de la codificación

Información y codificación son las dos caras de una misma moneda. Ya se dijo al tratar de definir la información que ésta es imposible de manipular en la práctica si no está representada de alguna forma y en este punto es donde entra en juego el proceso de la codificación. El problema de la compresión de datos puede ser separado en dos fases [42] fundamentales: (1) la descomposición de un conjunto de datos (por ejemplo, un fichero de texto en español o una imagen) en una secuencia de eventos (porciones de información) cuya representación actual no es suficientemente compacta y (2) la codificación de estos eventos usando tan pocos elementos de representación como sea posible. En la compresión de datos existe por tanto un proceso de recodificación con el objetivo de encontrar una representación más corta.

1.2.1 Elementos de representación

Un alfabeto-código es un conjunto B de elementos de representación utilizados en un proceso de codificación de la información. Los elementos de B notados por b_i (a los que llamaremos símbolos-código) se usan para representar símbolos-fuente a_i . Al número s de símbolos-código distintos en B se llama base del código. Por tanto, un alfabeto-código se define como

$$B = \{b_1, \dots, b_s\}.$$

Cuando $s = 2$, los elementos del alfabeto-código se llaman bits de código. Es importante diferenciar entre un bit de código y un bit de información. Los símbolos 0 y 1 representan generalmente dos niveles de tensión, dos estados magnéticos, etc. El término bit (de código), por tanto, se utiliza en las computadoras digitales para designar uno de estos dos niveles lógicos, pero debe tenerse en cuenta que un bit de código puede representar o no un bit de información. Esto en realidad depende del nivel de compresión conseguido.³ Por

³El nivel de representación más compacto es aquél en el que un bit de información está representado por un bit de código.

ejemplo, un bit de código representaría sin redundancia cada nacimiento de un ser humano ya que cada bit de código representaría exactamente a un bit de información.

1.2.2 Codificación

La función codificación o simplemente código es una función que asigna a cada símbolo-fuente a_i un conjunto de uno o más símbolos-código b_j , generando una cadena o bloque c_i . Sea C el conjunto de todos los c_i . Cuando r (la base del alfabeto-fuente) es igual a s (la base del alfabeto-código), cada símbolo-fuente se representa mediante un símbolo-código. Sin embargo, como suele ser habitual debido a requerimientos tecnológicos, $r > s$ y por tanto es necesario usar bloques de símbolos-código para designar a cada símbolo-fuente. A un bloque c_i de símbolos-código que realiza tal función lo llamaremos palabra-código. El número de símbolos-código que forman una palabra-código c_i es su longitud y la notaremos matemáticamente por $l_s(c_i)$, donde s es la base del código. La secuencia de palabras-código que codifican una secuencia-fuente se llama secuencia-código o corriente-código (*code-stream*).⁴

Por ejemplo, una fuente de información muy importante para nosotros es la almacenada en nuestro código genético. La información genética utiliza un alfabeto-fuente con 4 símbolos-fuente, formado por los 4 nucleótidos (desoxiadenilato o adenina (*a*), desoxiguanilato o guanina (*g*), desoxitimidilato o timina (*t*) y desoxicitidilato o citosina (*c*)). El alfabeto-código utilizado es por tanto, el $\{a, g, t, c\}$. En este ejemplo, $r = s = 4$ y la longitud de cada palabra-código es 1.

Otro ejemplo (de código binario en este caso) es el siguiente: una fuente de información ASCII puede generar hasta 256 eventos distintos. Para representar un alfabeto-fuente con 256 símbolos-fuente se utilizan 8 bits de código/símbolo-fuente. Por ejemplo, la palabra-código asociada al símbolo-fuente ASCII **a** es la 0110 0001 (97 en decimal). En este ejemplo, $r = 256$, $s = 2$ y $l(c_i) = 8$, $0 \leq i < r$.

1.2.3 Códigos usados en compresión

La definición presentada de código es muy amplia. En general, para que un código sea útil en compresión de datos (y sea por tanto un código compresor) es preciso imponer al menos dos condiciones: (1) que sea de longitud variable y (2) que sea decodificable. Lo primero implica que existan palabras-código de diferente longitud. Lo segundo que cualquier símbolo-fuente pueda ser decodificado sin incertidumbre.

Junto a estas dos propiedades, existe otra muy conveniente aunque no indispensable: un código compresor además debiera ser instantáneamente decodificable, lo que significa que los símbolos-fuente pueden ser decodificados sin necesidad de conocer la siguiente palabra-código de la secuencia-código. Los códigos instantáneos son muy fáciles de distinguir porque todos ellos cumplen la propiedad del prefijo [8], que dice que ninguna palabra-código es

⁴Este término se usa especialmente cuando los símbolos se reciben y se codifican de forma inmediata, sin necesidad de ser almacenados.

prefijo de cualquier otra. La Tabla 1.1 presenta un ejemplo de código descodificable aunque no instantáneo y otro instantáneo.

Tabla 1.1: Ejemplos con diferentes tipos de códigos descodificables.

símbolo-fuente	palabra-código	
	no instantáneo	instantáneo
1	10	0
2	00	10
3	11	110
4	110	111

Para ver cómo se complica la descodificación usando un código no instantáneo, valga el siguiente ejemplo. Supongamos la secuencia de código 11000. El primer bit indica que el primer símbolo-fuente es 1, 3 o 4. El siguiente reduce las posibilidades al 3 o al 4. Con el tercero se sabe que se trata de la secuencia que comienza por 32 o por 4. El cuarto bit no nos permite distinguir aún entre ambas posibilidades y el quinto bit, finalmente indica que se trataba de la secuencia 42.

El código instantáneo cumple la propiedad del prefijo, lo que lo convierte automáticamente en descodificable y además, no es necesario conocer los bits del siguiente código para descodificar un símbolo anterior. Por lo tanto un código prefijo es descodificable e instantáneo.

1.2.4 Códigos de longitud variable

Atendiendo a la propiedad de que todas las palabras-código utilizadas en una codificación tengan o no la misma longitud, podemos dividir los códigos en (1) códigos de longitud variable y (2) códigos uniformes, códigos bloque o códigos de longitud fija.

Debido a que la longitud de una palabra-código puede no ser fija, se define la longitud media de un código como

$$L_s(C) = \sum_{i=1}^r l_s(c_i) \times p(a_i), \quad (1.5)$$

donde s es la base del alfabeto-código utilizado y r la base del alfabeto-fuente. La longitud media de un código se expresa en símbolos-código/símbolos-fuente o símbolos-código/palabra-código.⁵ Como es lógico, independientemente de la probabilidad asociada al símbolo-fuente, si s es grande, las longitudes son pequeñas y viceversa. Cuando $s = 2$ (lo mínimo posible),

⁵Debe tenerse en cuenta que cada símbolo-fuente se expresa mediante una palabra-código.

estamos utilizando un alfabeto-código binario y entonces

$$L_2(C) = L(C) = \sum_{i=1}^r l(c_i) \times p(a_i), \quad (1.6)$$

siendo

$$l(c_i) = l_2(c_i), \quad (1.7)$$

donde hablaremos forzosamente de bits de código.

Nótese que por definición de codificación se cumple que $p(a_i) = p(c_i)$, siendo $p(c_i)$ la probabilidad de utilizar la palabra-código c_i en la codificación de una determinada secuencia-fuente.

Típicamente, en un proceso de compresión de datos, una secuencia-fuente está inicialmente codificada mediante una secuencia de palabras-código de longitud fija y se trata de encontrar una codificación de longitud variable que asigne a los símbolos-fuente más frecuentes nuevas palabras-código más cortas (y viceversa). Por tanto, la longitud media de un código de longitud fija coincide con la longitud de una palabra-código y es $\log_s r$ donde s es el número de símbolos-código y r es el número de símbolos-fuente a codificar.

1.2.5 La desigualdad de Kraft

Tal y como se indicó en la Sección 1.2.3, una propiedad interesante en los códigos compresores es que cumplan la propiedad del prefijo y sean así decodificables de forma instantánea. Existe una condición matemática sencilla que permite saber si un código es instantáneo conociendo simplemente las longitudes de las palabras-código. Fue demostrada por Kraft [58] en 1949 y dice que un código es instantáneo si cumple la desigualdad

$$\sum_{i=1}^r s^{-l_s(c_i)} \leq 1. \quad (1.8)$$

Kraft demostró que su desigualdad es suficiente, pero no demostró que fuera necesaria. Esta se la debemos a McMillan's [65, 66] en 1956.

La desigualdad de Kraft permite además conocer cuál debe ser la longitud promedio que debe tener un código para que éste sea óptimo desde el punto de vista de la compresión de datos. Basta con minimizar la Ecuación (1.5) utilizando como condición la Desigualdad (1.8). El resultado es que las longitudes de las palabras-código tienen que ser igual a la información asociada al símbolo-fuente representado

$$l_s(c_i) = -\log_s p(a_i). \quad (1.9)$$

Sustituyendo la Ecuación (1.9) en la Ecuación (1.5), se determina que

$$L_s(\hat{C}) = \sum_{i=1}^r -\log_s p(a_i) \times p(a_i) = H_s(A), \quad (1.10)$$

o lo que es lo mismo, la longitud media de un código óptimo \hat{C} es igual a la entropía de la fuente de información codificada.

Si la base del alfabeto-código es dos ($s = 2$), la longitud se expresa en bits de código. Si además, la información se mide en bits de información, diremos que el código es óptimo cuando el número de bits de código utilizados coincide con el número de bits de información representados.

1.2.6 Códigos prefijo óptimos

Los códigos prefijo son utilizados en compresión de datos porque son instantáneamente decodificables. Sin embargo, poseen una restricción que los puede convertir en muchos casos en no óptimos para comprimir: la longitud de una palabra-código debe ser un número entero. Debido a esto, si las entropías de los símbolos-fuente codificados no son números enteros, no se conseguirá que cada bit de código represente a un bit de información exactamente.

Observando la Expresión (1.9) es fácil deducir que, para que un código prefijo sea óptimo, las probabilidades de ocurrencia de cada uno de los símbolos-fuente codificados deben ser una potencia inversa del número de símbolos-código s . En la práctica, al utilizar códigos binarios, todas las probabilidades deben poder calcularse según la expresión

$$p(a_i) = \frac{1}{2^x}, \quad (1.11)$$

donde x es un número natural. El código de Huffman [49] es un claro ejemplo de código prefijo óptimo bajo éstas circunstancias.

1.2.7 Eficiencia y redundancia de un código

Ya que es posible conocer (utilizando la Ecuación (1.10)) cómo de bueno puede llegar a ser un código para la compresión de datos, podemos definir la eficiencia de un código como la razón existente entre la longitud óptima (determinada por la entropía) y la longitud actual conseguida por el código

$$E_s(C) = \frac{H_s(A)}{L_s(C)}. \quad (1.12)$$

La eficiencia de un código se expresa en bits de información/bits de código. Nótese que la eficiencia es un valor real comprendido entre cero y uno, y cuanto mayor sea, más eficiente será el código. Así, podemos definir la redundancia de un código calculada como

$$R_s(C) = 1 - E_s(C). \quad (1.13)$$

La redundancia de un código se expresa en las mismas unidades que la eficiencia.

1.3 Comunicación a través de canales discretos sin ruido

Una de las aplicaciones más claras de la compresión de datos consiste en maximizar la tasa de transmisión de datos a través de un cierto canal de comunicación. Para hacer esto, hay que enviar aquellos símbolos más probables usando los códigos de longitud más pequeña y viceversa, según la Expresión (1.9).

Shannon demostró que extendiendo la fuente suficientemente, es posible encontrar (asintóticamente) una codificación óptima, aunque no formuló ningún algoritmo para calcular los códigos [109]. Posteriormente, Huffman encontró un método de codificación óptimo para algunos casos (ver Expresión (1.11)) sin necesidad de extender la fuente [49].

En aquellas situaciones en las que el código de Huffman no es óptimo, se puede aplicar la idea original de Shannon y extender la fuente para encontrar una codificación más eficiente. El problema es que la extensión provoca un aumento exponencial del número de símbolos a codificar (ver Sección 1.1.6) y por tanto, del tiempo de codificación. A principios de los 80, Pasco [75], y posteriormente Rissanen y Langdon [88] encontraron una codificación (que se llamó codificación aritmética) que consigue, independientemente de la condición expresada en la Ecuación (1.11) y al menos en la teoría, el límite teórico de la entropía. Sin embargo, en el fondo debemos darle la razón a Shannon: como más tarde se mostrará, la codificación aritmética asigna una gran palabra-código a cada secuencia de símbolos-fuente y esto es en definitiva una extensión de la fuente tan grande, que la secuencia de datos originales a codificar es tratada como un único gran símbolo-fuente.

1.3.1 Capacidad de un canal discreto sin ruido

En la teoría de la información, el canal de comunicación constituye el enlace entre el codificador del emisor y el descodificador del receptor. Un canal discreto transmite símbolos-código (bits de código cuando sea binario, como casi siempre ocurre). Si además no existe ruido de transmisión, a la salida tenemos los mismos símbolos-código y en la misma secuencia (orden) que a la entrada, transcurrido un cierto tiempo. Puesto que no existen alteraciones durante la transmisión, podemos afirmar que la cantidad de información de salida es igual a la de entrada.

En el contexto de la compresión de datos, el canal simboliza el dispositivo de transmisión o de almacenamiento. Por tanto, la compresión de datos va a ser óptima si aprovechamos al máximo la capacidad de transmisión del canal.

Se define la capacidad de un canal como la máxima cantidad de información que puede transmitirse en cierto intervalo de tiempo. Todos los canales, incluso los ruidosos, pueden transmitir una cantidad de información infinita durante un periodo de tiempo ilimitado, así que lo normal es hablar de cantidad de información transmitida por unidad de tiempo. Éste es el concepto de capacidad de un canal manejado en muchos ámbitos de transmisión de datos, independientemente del tipo de canal (analógico o digital).

Sin embargo, para definir la capacidad de un canal discreto, se puede hablar de capacidad en términos de la cantidad de información transmitida por símbolo-código o palabra-

código enviada. Cuando la duración o longitud de los símbolos-código es constante y tenemos un total de s símbolos-código, la capacidad de un canal puede calcularse como

$$W = \log_2 s, \quad (1.14)$$

cantidad medida en bits de información/bits de código.⁶ En el caso de trabajar con códigos y fuentes binarias, la capacidad de un canal (sin ruido) es de 1 bit de información/bit de código transmitido. En este caso, se está utilizando un código óptimo desde el punto de vista de la compresión de datos.

1.3.2 Teorema de la codificación en canales discretos sin ruido

Usando códigos prefijo, rara vez se satisface la Condición (1.9) y se encuentra así una longitud promedio óptima de las palabras-código (ver Sección 1.2.6). Por consiguiente, ya que las longitudes deben ser números enteros, debemos tomar una longitud para las palabras-código c_i tal que

$$-\log_s p(a_i) \leq l_s(c_i) < -\log_s p(a_i) + 1. \quad (1.15)$$

Si multiplicamos la Expresión (1.15) por $p(a_i)$ y sumamos sobre s , obtenemos entropías y longitudes promedio⁷

$$H_s(A) \leq L_s(C) < H_s(A) + 1. \quad (1.16)$$

De esta ecuación, Shannon dedujo que extendiendo la fuente de información podemos mejorar la codificación. Si N es el número de símbolos-fuente/símbolo-extendido, la entropía de la fuente extendida es $N \times H_s(A)$ y la longitud media de las palabras-código que representan los símbolos-extendidos es $N \times L_s(C)$. Por tanto la Expresión (1.16) se transforma en

$$N \times H_s(A) \leq N \times L_s(C) < N \times H_s(A) + 1. \quad (1.17)$$

Dividiendo la Expresión (1.17) entre N se obtiene que

$$H_s(A) \leq L_s(C) < H_s(A) + \frac{1}{N}. \quad (1.18)$$

Obsérvese que si en la Desigualdad (1.18) aumentamos N , podemos hacer $L_s(C)$ y $H_s(A)$ tan próximos como deseemos. Matemáticamente hablando, el

$$\lim_{N \rightarrow \infty} L_s(C) = H_s(A). \quad (1.19)$$

Supongamos una fuente de información de alfabeto-fuente $A = \{a_1, a_2\}$ tal que $p(a_1) = 2/3$ y $p(a_2) = 1/3$. La entropía de la fuente calculada según la Expresión (1.2) es de 0.918

⁶Nótese que las dimensiones de la capacidad de un canal y de la eficiencia de un código coinciden.

⁷Recuérdese que $\sum_s p(a_i) = 1$.

bits de información/símbolo-fuente. Una codificación binaria (de las dos posibles) podría ser $c_1 = 0$ y $c_2 = 1$, de lo que se deduce inmediatamente que la longitud media del código es $L(C) = 1$ bit de código/símbolo-fuente. La eficiencia del código empleado es según la Expresión (1.12) de 0.918 bits de información/bit de código.

Extendiendo la fuente para $N = 2$, el nuevo alfabeto-fuente va a ser $A_2 = \{\alpha_1, \alpha_2, \alpha_3, \alpha_4\}$ donde $\alpha_1 = a_1a_1$, $\alpha_2 = a_1a_2$, $\alpha_3 = a_2a_1$ y $\alpha_4 = a_2a_2$. Si consideramos que la fuente de información no posee memoria, las probabilidades de los símbolos-extendidos se calculan multiplicando las probabilidades de los símbolos-fuente, obteniéndose que $p(\alpha_1) = 4/9$, $p(\alpha_2) = 2/9$, $p(\alpha_3) = 2/9$ y $p(\alpha_4) = 1/9$. Ahora, utilizaremos un código binario de longitud variable que cumpla la propiedad del prefijo, capaz de codificar con menos bits los símbolos-extendidos más probables a costa de codificar con más bits los menos probables. Sea este nuevo código $C = \{0, 10, 110, 111\}$. La longitud media del código se calcula según la Expresión (1.5) y resulta 1.89 bits de código/símbolo-extendido. Puesto que cada símbolo-bloque contiene 2 símbolos-fuente, la longitud media del código es $1.89/2 = 0.94$ bits de código/símbolo-fuente. La eficiencia por tanto, según la Expresión (1.12) resulta ser igual al

$$\frac{0.918 \text{ bits de información/símbolo-fuente}}{0.94 \text{ bits de código/símbolo-fuente}} =$$

$$0.977 \text{ bits de información/bit de código.}$$

Hemos mejorado la representación en un 5.9%.

1.4 Comprimir = modelar + codificar

Para muchos investigadores del campo de la compresión de datos, el trabajo más interesante de Shannon es el de la medida de la entropía del idioma inglés [108]. Para ello, Shannon utilizó hablantes nativos (de habla inglesa) como predictores y midió la entropía del error de predicción, calculando de esta forma que aproximadamente el 50% de los caracteres ingleses son redundantes. Esta idea tan brillante ha sido luego utilizada por Rissanen y Langdon [89] para indicar que cualquier compresor de datos puede ser descompuesto en dos partes independientes: (1) un modelo de la fuente a comprimir utilizado como predictor y (2) un codificador (ver Figura 1.1).

La función del predictor usado en el emisor consiste en generar una secuencia de símbolos-fuente (la predicción) lo más parecida posible a la secuencia de símbolos-fuente que forman el mensaje y para ello, si el modelo posee memoria (que es lo más frecuente), utiliza la historia conocida hasta ese instante (o contexto).⁸ Desde este punto de vista, el modelo es una fuente de información que trata de imitar a la fuente de información a comprimir. Puesto que en el receptor existe un modelo idéntico, la información que éste es capaz de reproducir no tiene por qué ser enviada a través del canal.

⁸En el experimento de Shannon, el contexto estaba formado por los últimos 100 caracteres.

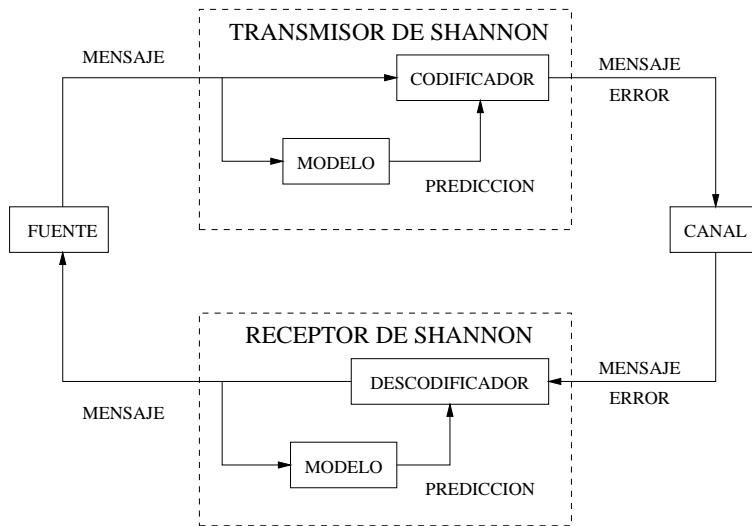


Figura 1.1: El paradigma de la compresión/descompresión de datos formulado por Shannon.

La misión del codificador es encontrar una representación eficiente de la información residual, que es aquella que el modelo no puede predecir. Básicamente, lo que ocurre es que por cada símbolo-fuente emitido por la fuente, el modelo genera otro símbolo-fuente, y la discrepancia entre ambos es lo que el codificador representa de forma compacta y envía a través del canal.

Desde un punto de vista probabilístico, la predicción consiste en asignar o determinar la probabilidad correcta a cada símbolo-fuente y el codificador asigna un código de longitud variable a cada símbolo en función de dicha probabilidad. En concreto, el codificador tratará de asignar al símbolo-fuente s codificado $-\log_2 p(s)$ bits de código, donde $p(s)$ es la probabilidad del símbolo-fuente a comprimir (la predicción) generada por el modelo. Nótese que, para que la codificación sea posible, debe ocurrir que $0 < p(s) < 1$:

1. Si $p(s) = 0$, el símbolo-fuente sería codificado con una palabra-código de longitud infinita. Además, s no pertenecería al alfabeto-fuente.
2. Si $p(s) = 1$, la palabra-código generada obtendría una longitud de 0 bits y el descodificador sería incapaz de detectarla. Por otra parte, el alfabeto fuente sólo contendría un símbolo-fuente.

1.5 Medidas de compresión

Existe un gran número de métricas usadas en la medición de la eficiencia de un sistema de compresión [42]. El principal defecto de todas ellas es que dicha medición es siempre

relativa a los datos que están siendo comprimidos, por lo que si no se indican cuales fueron, la medida es inútil.

En esta sección se estudian algunas de las medidas más usadas y además se presenta una propuesta nueva: “el espacio del rendimiento” que trata de situar a los sistemas de compresión y descompresión dentro de un plano, lo que facilita una comprensión rápida de su eficiencia.

1.5.1 Bits por símbolo

Las medidas relativas siempre se calculan expresando una relación entre las longitudes de las secuencias-código antes y después del proceso de compresión (α y β respectivamente). Una métrica bastante intuitiva es el número de bits de código/símbolo-fuente (BPS — Bits Por Símbolo —) obtenidos tras comprimir. Este valor se calcula como

$$\text{BPS} = l \times \frac{\beta}{\alpha}, \quad (1.20)$$

donde l es el tamaño en bits (de código) de la representación no comprimida de los símbolos-fuente.

Por ejemplo, si estamos comprimiendo ficheros ASCII el número de bits de código/símbolo-fuente o carácter ASCII es de 8 bits/carácter. Si obtenemos un fichero comprimido de 500 bytes de tamaño a partir de un fichero sin comprimir de 1000 bytes, el número de bits de código por símbolo-fuente después de la compresión es de

$$8 \times \frac{500}{1000} = 4.$$

1.5.2 Factor de compresión (X:1)

El factor de compresión es otra medida acerca del nivel de compresión conseguido que será usada en este trabajo. Indica la proporción existente entre el tamaño del fichero sin comprimir α y el tamaño del fichero comprimido β y se expresa típicamente de la forma (X:1) donde

$$X = \frac{\alpha}{\beta}. \quad (1.21)$$

Por ejemplo, cuando el fichero comprimido es de tamaño igual a la mitad del fichero original, hablamos de un factor de compresión de (2:1).

1.5.3 El espacio del rendimiento

Indicar únicamente el nivel de compresión que se alcanza resulta muchas veces insuficiente cuando tratamos de comparar dos algoritmos, pues a veces un factor crítico puede ser el tiempo de compresión y descompresión. Para relacionar ambas medidas: nivel de compresión y tiempo de ejecución, vamos a introducir una medida que las relaciona y que llamaremos rendimiento.

Tasa de compresión

Con el objetivo de hacer esta medida independiente de la máxima cantidad de parámetros, en lugar de usar los BPS para calcular el nivel de compresión usaremos una medida llamada tasa de compresión (TC) definida por

$$TC = 1 - \frac{\beta}{\alpha}. \quad (1.22)$$

Como puede comprobarse:

1. Si $\alpha = \beta$, $TC = 0$ y por lo tanto, no se ha podido comprimir nada.
2. Si $\alpha < \beta$, $TC < 0$ lo que significa que se ha producido una expansión al comprimir los datos.
3. Si $\alpha > \beta$, $0 < TC < 1$ lo que indica que los datos han sido comprimidos. TC no puede ser nunca igual a 1 ya que eso implicaría que $\beta = 0$, lo que es imposible por definición (ver Sección 1.4).

Tasas de transferencia

Para que el tiempo de ejecución sea lo más independiente posible de la máquina en la que se ejecutan los algoritmos y del tamaño de las secuencias de datos, será expresado en proporción al tiempo que emplea la utilidad *cp* del sistema operativo UNIX en copiar los datos desde un punto a otro en el sistema de ficheros. Todos los compresores y descompresores de datos funcionan básicamente igual que este comando, la única diferencia es que modifican la representación de los datos. Como *cp* es el transmisor de datos más elemental que es posible utilizar en la mayoría de entornos computacionales, es esperable que los compresores y expansores gasten más tiempo en realizar las transferencias. Por lo tanto, *cp* marca el tiempo mínimo (al menos en teoría) que un compresor o descomprimidor puede llegar a consumir. Sean las proporciones

$$\begin{aligned} TT_c &= \frac{t_{cp}}{t_c} \\ TT_d &= \frac{t_{cp}}{t_d} \end{aligned} \quad (1.23)$$

donde TT_c denota a la tasa de transferencia cuando se comprime y TT_d es la tasa de transferencia cuando se descomprime.

Por lo tanto, cuando estemos comprimiendo, la tasa de transferencia es la relación entre el tiempo que tarda *cp* en hacer una copia de los datos (t_{cp}) y el tiempo empleado en comprimir esos mismos datos (t_c). Cuando estemos descomprimiendo, la tasa se calcula dividiendo t_{cp} entre el tiempo gastado en descomprimir los datos (t_d).

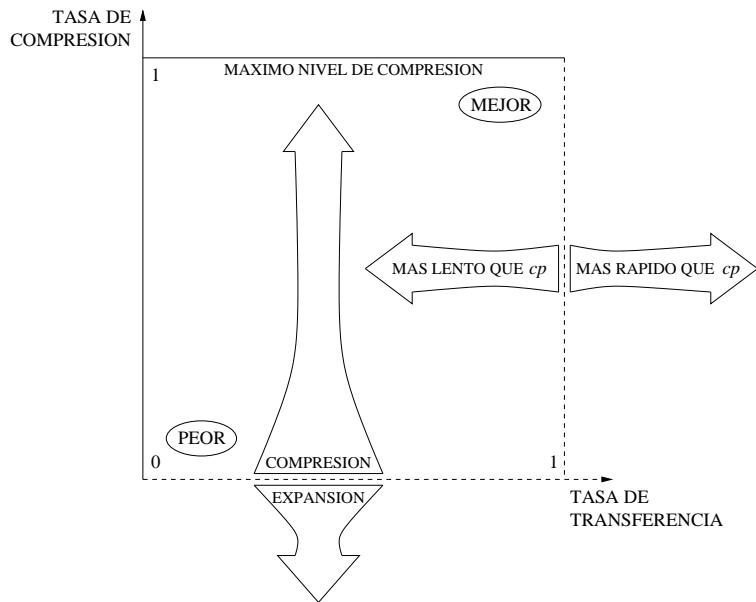


Figura 1.2: El espacio del rendimiento de los compresores de datos.

Simetría

La diferencia de velocidad entre el compresor y el descompresor (simetría) es un aspecto que hay que tener en cuenta en ciertas situaciones. Típicamente, la tasa de transferencia del descompresor supera la del compresor. En aquellos casos en los que los datos pueden comprimirse una vez y se descomprimen muchas veces, no importa que el sistema de compresión usado sea altamente asimétrico, pues lo que interesa es que las descompresiones sean rápidas a costa incluso de que la compresión sea lenta. Un ejemplo típico puede ser la compresión de sonido o de vídeo. Sin embargo, cuando la compresión y la descompresión se producen simultáneamente (por ejemplo, sobre dos sistemas remotos que se transmiten datos), son interesantes los procesos de compresión simétricos (para que un sistema no tenga que esperar al otro). Nótese que la simetría nada dice acerca de las tasas de transferencia que pueden ser conseguidas. Es simplemente una medida relativa entre los tiempos de compresión y descompresión.

El espacio del rendimiento

En este contexto, se propone representar para cualquier compresor o descompresor como un par de puntos en el espacio del rendimiento (ver Figura 1.2) de forma que para todo *codec* (*compressor/decompressor*) evaluado aparecerá una línea horizontal que une los puntos (TT_c, TC) y (TT_d, TC) . Nótese que:

- La línea se hallará por debajo de la tasa de compresión 0 si la compresión ha expan-

dido los datos y estará más cerca del 1 cuanto mejor sea la compresión.

- La capacidad compresora será superior cuanto más arriba se sitúe la línea.
- La velocidad del compresor o del descompresor será mayor cuanto más a la derecha pintemos los extremos de la línea. De hecho, si se supera la tasa de transferencia igual a 1, entonces supera a la velocidad con la que la utilidad *cp* copia los ficheros.
- El tamaño de la línea será mayor cuanto más asimétrico es el codec.

En definitiva, el rendimiento del compresor o del descompresor será mayor cuanto más en la esquina superior derecha del espacio esté situado. Las líneas continuas no pueden ser nunca sobrepasadas, pero las discontinuas sí.

El espacio del rendimiento posee unas determinadas cualidades como medida que son resaltadas a continuación:

- Las tasas de transferencia son dependientes de la complejidad del algoritmo pero independientes del volumen de datos. Ahora, la longitud del fichero no afecta puesto que también el tiempo de copia del fichero (t_{cp}) se ve afectado en la misma proporción.
- La tasa de compresión es independiente del formato de los datos y está normalizada puesto que siempre es menor que 1.
- Muestra rápidamente la simetría del esquema de compresión.

1.6 Otras aplicaciones de un compresor de datos

Un compresor de datos puede usarse para otro tipo de objetivos diferente de la compresión. Estas aplicaciones especiales son presentadas porque pueden ayudar a comprobar la calidad de un sistema de compresión.

1.6.1 Generación de secuencias pseudo-aleatorias

Para que una secuencia de números sea considerada aleatoria, la relación entre cada dos números cualesquiera de la secuencia debe ser ninguna. En otras palabras, que aún conociéndose toda la secuencia generada, sea imposible predecir el valor del siguiente número.

Es sabido que no se puede generar una secuencia con esta característica mediante procedimientos algorítmicos porque los programas de ordenador son absolutamente deterministas. Lo que se hace entonces es tratar de simular el comportamiento de una fuente absolutamente aleatoria mediante métodos que nos garanticen cierto nivel de descorrelación en la secuencia, métodos que se llaman pseudo-aleatorios.

En este sentido, los compresores de datos son excelentes generadores de números pseudo-aleatorios ya que su objetivo principal es la eliminación de cualquier tipo de redundancia (que dependerá del modelo utilizado) en el fichero comprimido. De hecho, cuanto

más aleatoriedad sea su salida, menor va a ser el nivel de redundancia y mejores niveles de compresión van a obtenerse. En consecuencia, una forma de medir la calidad de un compresor de datos es pasar un *test* de aleatoriedad a su salida comprimida.

Un *test* sencillo consiste en medir las diferencias de ocurrencias de cada uno de los patrones que podemos formar con el fichero de salida. En el primer paso, contaría los el número de ceros y de unos que existen. Si el compresor es bueno, ambas cantidades deben ser similares. A continuación contaría el número de ocurrencias de los patrones 00, 01, 10 y 11, y así sucesivamente. El compresor de datos perfecto generaría una salida pseudo-aleatoria que sea cual fuere el patrón buscado, la probabilidad de ser encontrado es igual a la de encontrar cualquier otro de idéntica longitud.

1.6.2 Encriptación de datos

Una de las principales estrategias que utilizan los algoritmos de desencriptación consiste en lanzar ataques estadísticos contra la secuencia encriptada con el fin de identificar los símbolos-fuente más frecuentes dentro de la secuencia. Conocidos estos, se facilita enormemente la extracción de la información contenida en el mensaje.

Justamente para evitar este tipo de ataques, la compresión de datos puede ser usada para eliminar la máxima cantidad de redundancia estadística dentro del mensaje. Por lo tanto, los compresores de datos son buenos encriptadores de datos [125] y una forma de estudiar la calidad de un esquema de compresión consiste en encontrar cuánto cuesta recuperar la información almacenada en la secuencia comprimida usando algoritmos de desencriptación.

1.6.3 Detección de fuentes de información

Existen situaciones donde se debe conocer de forma automática si una determinada fuente de información es aleatoria (ruido electrónico, por ejemplo) o no. Un problema de este tipo ocurre en la búsqueda de inteligencia extraterrestre usando radiotelescopios.

Los compresores de datos, por definición, son incapaces de comprimir aquellas secuencias de datos que no posean algún tipo de redundancia y viceversa: para que exista cierto nivel de compresión, debe existir un mínimo de redundancia.

En este marco, un compresor de datos será mejor que otro si consigue distinguir con mayor porcentaje de éxito una secuencia aleatoria de datos de otra que no lo es.

1.7 Limitaciones de la compresión de datos

1.7.1 El argumento del recuento

El fenómeno de la compresión de datos puede ser considerado como un mapeo o proyección entre conjuntos. Sea A el conjunto formado por todos los ficheros⁹ de hasta n bits de longitud¹⁰, es decir

$$A = \{A_1, A_2, \dots, A_n\}$$

donde A_i es el conjunto de todos los ficheros de i bits que tienen 2^i elementos. Sea B el conjunto formado por todos los ficheros resultantes de comprimir todos los ficheros de A . Para que la compresión sea sin pérdida de información, cualesquiera dos ficheros de A deberán ser comprimidos obteniéndose dos ficheros distintos de B . En otras palabras, la compresión de datos sin pérdida es una proyección biyectiva y en consecuencia A y B deben tener el mismo número de elementos (ficheros), que asciende a

$$N = 2^n + 2^{(n-1)} + \dots + 2. \quad (1.24)$$

Afirmar que es posible comprimir sin pérdida todos los ficheros de A significa, como mínimo, que todos ellos obtienen un fichero de B cuya longitud es de a lo sumo $n - 1$ bits, lo cual es imposible ya que no podemos enumerar N ficheros de hasta n bits de longitud usando un índice de hasta $n - 1$ bits. Por lo tanto, ningún compresor de datos es capaz de comprimir reversiblemente todos los ficheros. De hecho, cualquier compresor de datos expandirá el 50% de los ficheros. El arte de la compresión de datos sin pérdida radica precisamente en idear compresores que compriman lo máximo posible aquellos ficheros que más frecuentemente deben ser comprimidos. En el FAQ de compresión [30] es posible encontrar algunos ejemplos de personas y compañías que afirmaron (erróneamente) haber diseñado un compresor capaz de comprimir sin ninguna pérdida de información cualquier tipo de fichero, incluidos los formados por números aleatorios que son por definición incompresibles.

1.7.2 Los compresores iterativos no existen

Los compresores de datos iterativos (o recursivos), en teoría serían aquellos que son capaces de comprimir un fichero en sucesivas pasadas. En cada iteración el fichero se recomprime obteniéndose una longitud menor que en la iteración anterior. Este tipo de algoritmos han sido en ocasiones propuestos como compresores ideales, capaces de comprimir (sin límite aparente) cualquier fichero. La trampa está en pensar que es posible construir un compresor que sea capaz de comprimir, aunque sólo sea en una proporción muy pequeña (un bit de

⁹Es común llamar “fichero” a las secuencias-código de longitud conocida, aunque éste sea también el nombre que recibe la estructura de datos proporcionada por muchos sistemas operativos.

¹⁰Sin pérdida de generalidad, trataremos con alfabetos-código binarios.

código, por ejemplo) cualquier fichero, incluyendo por supuesto el fichero comprimido en la iteración anterior.

Estos compresores no existen (o al menos no son útiles) por dos razones: (1) porque su existencia contradice el argumento del conteo, y (2) porque si un compresor de datos puede ser utilizado iterativamente es porque en cada iteración introduce cierto nivel de redundancia y esto implica que tarde o temprano generará una expansión. Además, existe una máxima (muy lógica) en el diseño de compresores de datos [30] que dice que si un compresor (formado por un modelo y un codificador) consigue en dos pasos más compresión que en un único paso, para algún fichero, entonces existe un modelo tal que aplicando el mismo codificador obtendrá el mismo resultado y en un solo paso. El modelo buscado debe explotar la redundancia que el otro modelo elimina en dos o más pasadas.

Tratando de justificar la segunda razón no olvidemos que, si mantenemos fijo el modelo (cosa que debe ocurrir en la compresión iterativa) y utilizamos un codificador imperfecto (cosa que ocurre en cualquier caso, ya que ninguno de los que existen es capaz de representar en todos los casos cada bit de información mediante un bit de código), la introducción de redundancia por parte del codificador es inevitable, incluso en la primera iteración. Por tanto, la compresión iterativa es imposible o ineficaz.

Sin embargo (y por esta razón se propone aquí) la compresión iterativa puede utilizarse para averiguar la cantidad absoluta de redundancia generada en el compresor. Si el compresor fuera perfecto, no se produciría nunca una expansión, y sólo en la primera iteración se produciría una compresión. Como los compresores son en general imperfectos, en la primera iteración debe alcanzarse la salida más compacta posible y en las siguientes siempre van a producirse expansiones. Así, es posible calcular empíricamente la cantidad absoluta de redundancia introducida por el compresor en una iteración, observando el aumento de longitud producido.

Por ejemplo, si un fichero de 1000 bytes al ser comprimido por primera vez genera otro de 500 bytes y al ser recomprimido (por el mismo compresor) provoca otro de 510 bytes, podemos decir que el compresor genera 10 bytes redundantes por cada 500 bytes procesados.

Es de esperar que, a mayor redundancia introducida por el codificador, menores niveles de compresión van a ser obtenidos en cualquier iteración (incluida la primera), aunque esta afirmación también depende fuertemente del modelo usado.

1.8 El ciclo de la información

Ahora intentaremos relacionar el concepto de compresor y descompresor de datos, el de información y el de codificación, tratando además de aclarar un poco el gran número de definiciones y conceptos teóricos introducidos hasta ahora.

La Figura 1.3 resume el proceso de la compresión y descompresión de datos. El compresor recodifica la actual representación de un cierto volumen de información, tratando de generar una representación más corta. Puesto que en la mayoría de las ocasiones se

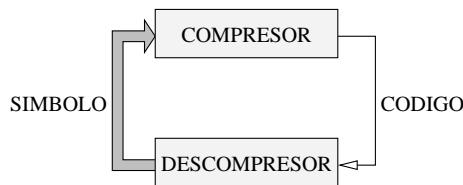


Figura 1.3: El ciclo de la información

trabaja con códigos cuyos alfabetos-código son binarios, llamaremos en adelante a los símbolos-código, bits de código y a las palabras-código, simplemente códigos. Llamaremos a los símbolos-fuente simplemente símbolos y en consecuencia, el papel del descompresor en el ciclo de la información es el de recuperar la representación original de los símbolos. Diremos que, en general, descodifica símbolos a partir de códigos.

En el ciclo de la información, los procesos de compresión y descompresión no crean ni destruyen información, solamente alteran su representación. Un símbolo es por tanto una representación cualquiera de una cierta cantidad de información, y un código es una representación, en general más corta, de la misma cantidad de información. Para enfatizar gráficamente esta idea, en la Figura 1.3 se ha pintado la flecha que pasa desde del descompresor al compresor más ancha que la que va en sentido contrario.

Las notaciones simplificadas presentadas en esta sección serán usadas cuando por el contexto no exista posibilidad de confusión entre símbolos-fuente, símbolos-código, palabras-código, etc.

1.9 Resumen

El propósito de esta introducción general ha sido definir y analizar desde un punto de vista teórico, los conceptos básicos sobre los que se fundamentan los compresores de datos.

En el fondo, todos los procesos de compresión son recodificaciones de información que una fuente ha generado y que un codificador ha representado. La información es despojada de su cuerpo actual y reencarnada mediante otra codificación que utiliza menos bits de código (al menos en promedio). En este sentido, hablamos de codificadores perfectos cuando se consigue representar (en promedio) un bit de información utilizando un bit de código.

Nótese que la utilización de un codificador óptimo poco nos dice realmente acerca de la calidad del compresor. Todos los compresores de datos pueden descomponerse en un codificador de longitud variable y en uno o varios modelos. La información se segmenta formando secuencias de símbolos-fuente que son utilizados por el modelo para expresar una predicción (generalmente en forma de probabilidad). Entonces, el codificador utiliza dichas probabilidades para codificar los símbolos-fuente más probables usando las palabras-código más cortas. Si la predicción no es adecuada, es evidente que la compresión puede degenerar en una expansión.

Actualmente, la búsqueda de codificadores más eficientes que la codificación de Huffman o la codificación aritmética se ha enfriado. Esto es debido fundamentalmente a que son prácticamente óptimos, pues alcanzan eficiencias muy próximas al límite teórico de la entropía de Shannon. El trabajo más grueso se realiza en la búsqueda de modelos que permitan predecir con mayor precisión el comportamiento de la fuente codificada. Varios modelos independientes, que explotan redundancias distintas, podrían ser utilizados para construir predictores mejores.

Para finalizar, en la última parte de este capítulo de introducción se ha tratado de transmitir dos importantes aspectos: (1) que no existe un compresor universal, capaz de comprimir cualquier secuencia y (2) que el cálculo de la eficiencia de un compresor es siempre relativo a los niveles de compresión conseguidos por otros compresores pues, al no existir el compresor perfecto, no tenemos una medida del máximo nivel de compresión alcanzable para una secuencia dada. En este sentido se ha definido una medida del rendimiento de un compresor/descompresor de datos que relaciona la tasa de compresión con la tasa de transferencia. Esta métrica será utilizada en el resto de la tesis para comparar las eficiencias arrojadas por los diferentes métodos de compresión estudiados.

Capítulo 2

Compresión de Texto

Entendemos por “texto” cualquier secuencia de símbolos (caracteres, números, puntos de una imagen, etc.) que es tratada como una secuencia unidimensional, y sobre la que no se presupone nada, excepto que puede contener redundancia estadística. Al ser ésta la fuente de redundancia más general que existe, los compresores de texto son universales y pueden usarse para intentar comprimir de forma reversible cualquier fuente de información.

Históricamente, el desarrollo de un compresor de texto ha sido más un arte que una ciencia. El porqué de esta evolución *ad hoc* se debe fundamentalmente a que los algoritmos de compresión de texto se basan en conceptos e ideas ciertamente evidentes y que son básicamente expuestas a continuación:

1. La expresión más sencilla de la redundancia en secuencias de texto se debe a que en ocasiones es posible encontrar “series” formadas por un único tipo de símbolo o carácter (por ejemplo, como pasa en la cadena `aaa...a`). Estas series de caracteres pueden ser representadas de forma compacta indicando el carácter usado y el número de veces que éste se repite. Cuando la representación de estas indicaciones (códigos) ocupe menos bits de código que la cadena, estaremos provocando un proceso de compresión de datos. Este tipo de compresores serán descritos bajo el nombre de “compresores de series”.
2. Por desgracia, la frecuencia de ocurrencia de series en las secuencias de texto que es bastante baja y pronto se desarrollaron compresores que explotan un tipo de redundancia semejante pero mucho más común. En general, las secuencias de texto se forman a partir de un conjunto relativamente reducido de cadenas básicas o palabras, que se repiten muchas veces y que podrían ser sustituidas por referencias a un diccionario donde figuran todas ellas, o por referencias a una posición anterior en la secuencia donde aparece la palabra. Si estas referencias ocupan menos bits de código que la palabra, estaremos comprimiendo. Este tipo de compresores se describen como “compresores basados en diccionarios” o “compresores de cadenas”.
3. Previamente al desarrollo de los compresores basados en diccionarios, en los trabajos

de Shannon [109] y Huffman [49] ya se había planteado una forma alternativa de realizar la compresión de texto que se llamó de forma genérica “codificación entrópica” o “codificación de símbolos”.

La idea en la que se fundamentan este tipo de compresores es sencilla y ya había sido aplicada con éxito por ejemplo en el Código Morse, que representa las letras del alfabeto usando un número variable de puntos y rayas. Una secuencia de texto puede ser comprimida si usamos un código de longitud variable para representar los símbolos más probables de forma que, en promedio, la longitud de los nuevos códigos sea menor que la longitud de la representación de la secuencia de símbolos original. El tratamiento que se hace de los símbolos es probabilístico y en este punto es donde Rissanen y Langdon [88, 89] ayudaron al diseño de mejores compresores de datos indicando que cualquier método de compresión puede ser descrito como un codificador de longitud variable y un modelo probabilístico que explote la redundancia estadística presente en la secuencia a comprimir.

En este capítulo se van a describir los compresores de texto más conocidos, basados en la codificación de series, de cadenas y entrópica. Las ideas sobre las que se basan los compresores de texto forman parte de cualquier compresor más específico y por esta razón deben ser analizados previamente. Además, son compresores de datos genéricos y por lo tanto también pueden ser utilizados para comprimir imágenes. Al final del capítulo se hace una evaluación de ellos mediante su representación en el espacio del rendimiento descrito en la Sección 1.5.3.

2.1 La compresión de series

Las cadenas construidas a partir de repeticiones de caracteres se generan con frecuencia en determinados tipos de ficheros, por ejemplo, en secuencias ASCII que contienen datos formateados (tablas) o en secuencias formadas por puntos de imágenes simples, generadas por regiones de valor constante. La forma en que se elimina este tipo de redundancia depende fuertemente de la naturaleza de la fuente y debido a esto, existe una gran variedad de algoritmos de codificación de series. Los más típicos se describen de forma resumida en esta sección.

2.1.1 RLE estándar

RLE (*Run Length Encoding*) propone codificar cada serie indicando el carácter que se repite y el número de repeticiones (típicamente, número de ocurrencias menos 1). Por tanto, cada serie se sustituye por una pareja de códigos xy , donde x representa al carácter e y indica el número de repeticiones. Así por ejemplo, la cadena **aaaab** se puede codificar como **a3b0**. El decodificador recibe el código **a3** y decodifica la serie **aaaa**, luego recibe **b0** y decodifica la serie **b**.

RLE permite construir compresores y descompresores muy rápidos, tanto en *software* como en *hardware*. Sin embargo, la tasa de compresión puede ser muy variable ya que depende de varios factores importantes:

- De que existan suficientes cadenas de longitud adecuada. Si son insuficientes, la representación actual será expandida.
- Del número de bits empleados para expresar el campo y . Si se emplean pocos bits, la codificación de series muy largas va a realizarse usando muchos códigos, mientras que si el número de bits es muy grande, este problema desaparece, aunque ahora la longitud del campo y aumenta. Hay por tanto que llegar siempre a una situación de compromiso, la cual es imposible de determinar a priori sin conocer el contenido total del fichero. Soluciones adaptativas parecen ser las más razonables.

2.1.2 RLE binario

Cuando la fuente de información es binaria, sólo existen dos caracteres diferentes y por tanto, no es necesario indicar el carácter x que forma la serie. Por ejemplo, una secuencia formada por 4 unos, 1 cero y 20 unos puede ser codificada por la secuencia de códigos 4 1 15 0 ¹, suponiendo que se han usado 4 bits para representar las longitudes de las series y suponiendo además que la primera serie está formada por unos. En este ejemplo, hemos representado una secuencia de 25 bits mediante otra de 20.

La descodificación es muy sencilla: la recepción del 4 indica al descomprimidor que hay que emitir cuatro unos. No hay confusión posible porque si se hubiera tratado de una serie de ceros, el primer código de compresión recibido habría sido cero. A continuación se recibe el código 1 y se emite un cero. Se recibe 15 y se emiten 15 unos. Se recibe 0 y no se emite ningún 0. Por último se recibe 5 y se emiten 5 unos.

2.1.3 RLE MNP-5

En el protocolo de transmisión de datos a través de modem MNP-5 (Microcom Networking Protocol) [39] se usa una técnica RLE para la compresión de secuencias generadas a partir de alfabetos con 256 símbolos. El algoritmo codifica series de al menos 3 caracteres según la Figura 2.1, en la que se ha supuesto que el campo que indica la longitud de la serie es de 8 bits. De esta forma, todos los códigos x e y que aparecen en la secuencia comprimida se representan usando 8 bits.

2.2 La compresión basada en diccionarios

En las secuencias de texto, en general, la ocurrencia de un determinado símbolo condiciona la aparición del símbolo siguiente. Cuando esto ocurre, la fuente tiene memoria (ver Sec-

¹Usaremos el espacio en blanco para delimitar códigos sólo cuando sea estrictamente necesario como pasa en este ejemplo.

entrada	salida	comentario
ab	ab	no hay codificación
aab	aab	no hay codificación
aaab	aaa0b	expansión
aaaab	aaa1b	ni compresión, ni expansión
aaaaab	aaa2b	compresión
$a^n b$	$aaa(n-3)b$	en general, donde a^n representa una serie de longitud $n \geq 3$

Figura 2.1: Codificación RLE-MNP5: algunos ejemplos.

ción 1.1.3) y consecuentemente, la cantidad de palabras que son generadas es menor que el número total de palabras que podrían ser formadas si no existiera correlación estadística. En este marco de trabajo, los métodos de compresión basados en diccionarios construyen estas estructuras de datos con las cadenas más frecuentes. Así, cuando de nuevo son encontradas, se sustituyen por referencias al diccionario. La compresión se produce cuando la longitud de las cadenas referenciadas es superior a la longitud de las referencias.

2.2.1 LZ77

En 1977, Jacob Ziv y Abraham Lempel [135] publicaron un método de sustitución de cadenas que habían aparecido previamente en un fichero de texto. LZ77 es la base de compresores de datos tan famosos como *gzip*, *pkzip* o *arj*.

El compresor

El codificador LZ77 es básicamente un buscador de cadenas. Define una estructura de datos llamada ventana deslizante (*sliding window*) que se desplaza a lo largo del fichero a comprimir (ver Figura 2.2). La ventana se divide en dos partes de tamaño desigual. La más grande se llama diccionario y almacena la parte del fichero que ya ha sido comprimido. La otra se llama *buffer* de anticipación (*look-ahead buffer*) y almacena la parte del fichero que se intenta comprimir.

En un régimen normal de funcionamiento, existe suficiente correlación y el diccionario es lo bastante grande para que sea posible encontrar en el diccionario al menos una ocurrencia de la cadena que está al comienzo del *buffer* de anticipación. Cuando esto ocurre, el compresor emite un código de compresión formado por la terna de códigos ijk , donde i es la posición de la cadena en el diccionario, j es la longitud de la cadena y k es el carácter situado a la derecha de la cadena en el *buffer* de anticipación (el carácter que provoca que la longitud de la cadena encontrada no sea mayor). A continuación, la ventana se desplaza $j+1$ posiciones (caracteres) sobre el fichero y el proceso se repite.² Si la búsqueda no tiene

²Nótese que el desplazamiento provoca que $j+1$ caracteres se pierdan por la izquierda del diccionario y no puedan ser usados para buscar en la siguiente iteración, y que otros tantos pasen desde el *buffer* al diccionario.

	dicc	buffer		salida	comentario
		abab	cbababaaaaaa	0 0 a	diccionario vacío
a	a	babc	bababaaaaaa	0 0 b	b no encontrado
	ab	abcb	ababaaaaaa	2 2 c	ab encontrado
ababc	babc	baba	baaaaaaa	0 3 a	bab encontrado
ababcbab	baba	baaa	aaa	0 2 a	ba encontrado
	abaa	aaaa		2 3 a	aaa encontrado
				0123	

Figura 2.2: Ejemplo de codificación de la cadena ababcbababaaaaaa usando LZ77.

entrada	salida	dicc	buffer
0 0 a	a		a
0 0 b	b		b
2 2 c	abc		ab
0 3 a	baba	a	abc
0 2 a	baa	babc	baba
2 3 a	aaaa	baba	baa
		abaa	aaaa
		0123	

Figura 2.3: Ejemplo de descodificación de la cadena ababcbababaaaaaa usando LZ77.

éxito (porque el carácter que está al principio del *buffer* no aparece en el diccionario) se emite un código de compresión $0\ 0\ k$, donde k es el carácter no encontrado. El proceso de codificación finaliza cuando ya no existen más caracteres que codificar.

El descompresor

El descodificador utiliza un diccionario y un *buffer* de anticipación idénticos y es mucho más rápido ya que no tiene que efectuar las búsquedas. Se limita a emitir los j caracteres extraídos a partir de la posición i del diccionario y por último, concatena a esta salida el carácter k . Con cada descodificación la ventana se desplaza $j + 1$ posiciones y $j + 1$ caracteres pasan desde el *buffer* al diccionario (ver Figura 2.3).

La principal ventaja del codificador LZ77 es el bajo coste de memoria y tiempo. Si el contenido del diccionario se mantiene ordenado, puede ejecutarse en un tiempo proporcional a $\log_2(N)$ donde N es el tamaño del diccionario (medido en caracteres). Por el contrario, el descodificador es mucho más veloz porque como ya se ha indicado, no tiene que realizar búsquedas. Prácticamente es una operación de copia de cadenas.

El principal inconveniente de LZ77 (lo que provocó que pronto apareciera una versión mejorada que a continuación veremos) es que cuando el diccionario no contiene la cadena buscada, se produce una expansión apreciable. Téngase en cuenta que:

- El tamaño del diccionario debe ser lo bastante grande para almacenar el máximo

	dicc	buffer		salida	comentario
a	abab	abab	cbababaaaaaa	1 a	diccionario vacío
	a	babc	bababaaaaaa	1 b	b no encontrado
	ab	abcb	ababaaaaaa	0 2 2	ab encontrado
	abab	cbab	abaaaaaa	1 c	c no encontrado
	babc	baba	baaaaaaa	0 0 3	bab encontrado
	abab	cbab	aaaa	0 2 3	aba encontrado
	ababcba	baba	aaaa	0 3 4	aaaa encontrado
	ababcbababa	aaaa	a	0 0 1	a encontrado
	0123				

Figura 2.4: Ejemplo de codificación de la cadena ababcbababaaaaaa usando LZSS.

número de cadenas y lo bastante pequeño para que los índices i no sean muy grandes. Un tamaño típico es $N = 4096$ lo que implica usar índices i de 12 bits.

- El tamaño del *buffer* de anticipación también es importante. Un *buffer* muy grande permite codificar cadenas muy largas, pero también provoca que el campo j aumente de tamaño. Típicamente se usan 3 bits para codificar j .

Con esta configuración, un carácter (8 bits) puede llegar a ser codificado por $12+3+8 = 23$ bits.

2.2.2 LZSS

En 1982, Storer y Szymanski [111] propusieron una variación de LZ77 llamada LZSS que soluciona la expansión excesiva de las codificaciones de las cadenas de un único carácter.

El codec LZSS es idéntico al de LZ77 salvo en la forma en que se codifican las ternas. LZSS propone dos códigos de compresión diferentes:

- $1k$ se usa para codificar el carácter k si éste no aparece en el diccionario. El primer campo sólo ocupa un bit y el segundo 8.
- $0ij$ se usa en caso contrario, donde i es la dirección de la cadena en el diccionario y j su longitud. El primer campo ocupa un bit y el tamaño de los siguientes se escoge siguiendo los mismos criterios que para LZ77.

La Figura 2.4 recoge un ejemplo de codificación y la Figura 2.5 su descodificación.

2.2.3 LZ78

En 1978, Ziv y Lempel [136] publicaron otra técnica de compresión de datos que, aunque sigue estando basada en un diccionario, es radicalmente diferente de LZ77. El concepto de ventana deslizante ya no se usa y el diccionario es una estructura más parecida a lo que

entrada	salida	dicc	buffer
1 a	a		a
1 b	b		b
0 2 2	ab		ab
1 c	c		ab
0 0 3	bab	a	ab
0 2 3	aba	abab	ab
0 3 4	aaaa	ababcba	c
0 0 1	a	ababcbababa	bab
			bab
			aaa
			a
			0123

Figura 2.5: Ejemplo de descodificación de la cadena `ababcbababaaaaaaaa` usando LZSS.

nosotros entendemos por diccionario: una colección de palabras o cadenas. La ventaja de LZ78 frente a LZ77 consiste en que ahora la longitud máxima de la cadena encontrada no está limitada por el tamaño del *buffer* de anticipación, sino que puede ser tan grande como el número de cadenas almacenadas en el diccionario, que coincide con su tamaño (medido en caracteres).

El compresor

El codificador trabaja descomponiendo las cadenas de símbolos en 2 partes. La primera se llama prefijo (w) y contiene todos los caracteres de la cadena excepto el último que se llama sufijo (k). w es en realidad un índice y no una cadena, aunque en este contexto ambos conceptos son intercambiables, con lo que podemos decir que la cadena de entrada es la concatenación wk . El diseño del diccionario es tal que si wk está en el diccionario, entonces w también está y además se encuentra en la dirección w .

Inicialmente el diccionario del comprresor y del descompresor se encuentran vacíos. Conforme transcurre la compresión, ambas estructuras se van llenando exactamente con idénticas cadenas y en el mismo orden, de forma que la descodificación es posible a partir de los índices w y k que forman la corriente de códigos. El algoritmo de codificación es el siguiente:

1. $w \leftarrow 0$ (dirección de la cadena vacía).
2. Mientras existan caracteres de entrada:
 - (a) $k \leftarrow$ siguiente carácter de entrada.
 - (b) Si wk existe en el diccionario:
 - i. $w \leftarrow$ dirección de wk en el diccionario.
 - (c) Si no:
 - i. Emitir w y k , usando tantos bits como sean necesarios.

entrada	salida	comentario
a	0a	$D[1] \leftarrow a$
b	0b	$D[2] \leftarrow b$
a		$D[1] = a$
b	1b	$D[3] \leftarrow ab$
c	0c	$D[4] \leftarrow c$
b		$D[2] = b$
a	2a	$D[5] \leftarrow ba$
b		$D[2] = b$
a		$D[5] = ba$
b	5b	$D[6] \leftarrow bab$
a		$D[1] \leftarrow a$
a	1a	$D[7] \leftarrow aa$
a		$D[1] \leftarrow a$
a		$D[7] = aa$
a	7a	$D[8] \leftarrow aaa$
a		$D[1] = a$
a		$D[7] = aa$
a		$D[8] = aaa$
a	8a	$D[9] \leftarrow aaaa$

Figura 2.6: Codificación de la cadena `ababcbababaaaaaaa` usando LZ78. En la columna de comentario usamos una flecha para indicar una inserción de una cadena dentro del diccionario y el símbolo igual para indicar que la cadena está en el diccionario. El diccionario $D[]$ se gestiona como un vector de cadenas.

- ii. Insertar la cadena wk en el diccionario.
- iii. $w \leftarrow 0$.

Sobre el compresor LZ78 podemos realizar los siguientes comentarios:

- La búsqueda de la cadena wk dentro del diccionario puede realizarse en un tiempo proporcional a $\log_2(N)$, donde N es el número de cadenas almacenadas en el diccionario. Por lo tanto, el tiempo de ejecución del compresor LZ78 es similar al del compresor LZ77.
- Puesto que el diccionario está inicialmente vacío, el número de bits que es necesario usar para codificar la cadena w (paso 2.c.i) aumenta desde 1 progresivamente, en función del número de cadenas almacenadas en el diccionario. Por tanto, inicialmente es posible usar menos bits que al final, cuando el diccionario está más lleno.
- La inserción de la primera cadena wk en el diccionario (paso 2.c.ii) se produce en la posición 1 y no en la 0. Esta dirección se reserva para poder codificar la cadena vacía.

dirección	<i>w</i>	<i>k</i>	cadena
1	0	a	a
2	0	b	b
3	1	b	ab
4	0	c	c
5	2	a	ba
6	5	b	bab
7	1	a	aa
8	7	a	aaa
9	8	a	aaaa

Figura 2.7: Ejemplo de diccionario para LZ78 generado tras codificar la cadena ababcbababaaaaaaaa. En cada entrada del diccionario se almacenan un par de códigos *wk*.

La Figura 2.6 muestra un ejemplo de compresión y la Figura 2.7 el contenido del diccionario. Como puede apreciarse, la principal aportación de LZ78 radica en la forma en la que el diccionario es almacenado. Como ya se ha indicado anteriormente, siempre se cumple que si la cadena *wk* está en el diccionario, entonces la cadena *w* también lo está. Esto hace posible que el diccionario tenga una representación muy compacta.

El descompresor

Únicamente a partir de los códigos *w* y *k*, el descompresor LZ78 reconstruye el diccionario y recupera la secuencia original. Esto se debe a que cuando el comprimido emite un código, sólo tiene en cuenta la información almacenada en el diccionario que es conocida por el descompresor hasta ese instante. El algoritmo de descompresión es el siguiente:

1. Mientras existan códigos a la entrada:
 - (a) $wk \leftarrow$ siguiente código.
 - (b) Emitir la cadena *w*. El diccionario se recorre en sentido descendente hasta alcanzar una entrada que sólo almacena un carácter. La cadena formada se emite en orden inverso a este recorrido.
 - (c) Emitir *k*.
 - (d) Insertar la cadena *wk* en el diccionario.

Al igual que ocurre con LZ77, el descompresor es un proceso muy veloz ya que no es necesario efectuar las búsquedas. Un ejemplo de descodificación se muestra en la Figura 2.8. Debe tenerse en cuenta que el diccionario generado por el descompresor es de nuevo el representado en la Figura 2.7.

entrada	salida	comentario
0a	a	$D[1] \leftarrow a$
0b	b	$D[2] \leftarrow b$
1b	ab	$D[3] \leftarrow ab$
0c	c	$D[4] \leftarrow c$
2a	ba	$D[5] \leftarrow ba$
5b	bab	$D[6] \leftarrow bab$
1a	aa	$D[7] \leftarrow aa$
7a	aaa	$D[8] \leftarrow aaa$
8a	aaaa	$D[9] \leftarrow aaaa$

Figura 2.8: Ejemplo de descodificación de la cadena `ababcbababaaaaaaaaaa` usando LZ78. $D[]$ representa al diccionario que se gestiona como un vector de cadenas y la flecha simboliza la operación de inserción de una cadena en el diccionario.

2.2.4 LZW

El artículo original de LZ78 posee un enfoque marcadamente matemático y no recibió la atención debida hasta que en 1984, Terry A. Welch [121] lo usara como base para escribir una versión diferente llamada LZW (Lempel Ziv Welch). Welch propuso su algoritmo con la idea de construir un compresor *hardware* que disminuyera el tiempo de entrada/salida en discos duros, aumentando además su capacidad efectiva.

LZW también ha sido implementado mediante *software*. De hecho, se usa en el formato estándar para imágenes GIF [18] (*Graphics Interchange Format*) y en el famoso programa *compress*³ [36, 112] usado como compresor de datos por excelencia en los sistemas UNIX durante mucho tiempo.

El compresor

El codificador LZW evita tener que enviar un carácter sin codificar con cada código de compresión y aumenta así su eficiencia respecto a LZ78. La estructura del diccionario es idéntica, pero ahora inicialmente no está vacío, sino que almacena en sus 256 primeras posiciones las 256 posibles cadenas de 1 carácter (suponiendo un alfabeto de 256 símbolos). A estas cadenas Welch las llama “raíces” y éste es su algoritmo:

1. Insertar en el diccionario las posibles raíces.
2. $w \leftarrow$ primer carácter de entrada.
3. Mientras existan caracteres de entrada:
 - (a) $k \leftarrow$ siguiente carácter de entrada.

³El código fuente de este programa puede encontrarse como fichero de *test* en el *Calgary Text Compression Corpus* descrito en el libro *Text Compression* de Bell *et al.* [9].

entrada	w	k	salida	comentario
ab	97	b	97	$D[257] \leftarrow ab$
a	98	a	98	$D[258] \leftarrow ba$
b	97	b		$w = 257$
c	257	c	257	$D[259] \leftarrow abc$
b	99	b	99	$D[260] \leftarrow cb$
a	98	a		$w = 258$
b	258	b	258	$D[261] \leftarrow bab$
a	98	a		$w = 258$
b	258	b		$w = 261$
a	261	a	261	$D[262] \leftarrow baba$
a	97	a	97	$D[263] \leftarrow aa$
a	97	a		$w = 263$
a	263	a	263	$D[264] \leftarrow aaa$
a	97	a		$w = 263$

Figura 2.9: Ejemplo de codificación de la cadena ababcbababaaaa usando LZW. El diccionario $D[]$ se gestiona como un vector de cadenas. La flecha simboliza a la operación de inserción de una cadena en el diccionario.

- (b) si wk está en el diccionario:
 - i. $w \leftarrow$ dirección de wk en el diccionario.
- (c) Si no:
 - i. Emitir w .
 - ii. Insertar la cadena wk en el diccionario.
 - iii. $w \leftarrow k$.

En la Figura 2.9 aparece un ejemplo de codificación. El diccionario usado contiene 257 raíces. Una raíz especial (la 256 o lo que es lo mismo, la que se almacena en esta dirección del diccionario) se usa para codificar un código especial llamado “escape” (al que notaremos con **ESC**) que cuando es emitido, precede a otro código que indica una acción que debe ser realizada por el descompresor. En general estos comandos son los siguientes:

- Incrementar el número de bits usados para codificar el campo w (téngase en cuenta que el diccionario crece e inicialmente es posible usar menos bits para codificar este campo).
- Vaciar el diccionario porque almacena cadenas diferentes de las que están siendo comprimidas en ese instante.
- Finalización de la descodificación porque ya no existen más símbolos que codificar.

dirección	cadena	w	k	\leq	$>$
0		0	NULL	ESC	ESC
:	:	:	:	:	:
97		0	a	263	257
98		0	b	258	ESC
99		0	c	260	ESC
:	:	:	:	:	:
256	reservado	ESC			
257	ab	97	b	ESC	259
258	ba	98	a	ESC	261
259	abc	257	c	ESC	ESC
260	cb	99	b	ESC	ESC
261	bab	258	b	262	ESC
262	baba	261	a	ESC	ESC
263	aa	97	a	264	ESC
264	aaa	263	a	?	?

Figura 2.10: Contenido del diccionario de LZW tras codificar la cadena `ababcbababaaaaaa`. Las columnas \leq y $>$ se usan para implementar el árbol binario de búsqueda.

Puesto que la entrada 256 del diccionario está ocupada, las cadenas encontradas en la secuencia a comprimir se almacenan a partir de la dirección 257. El contenido del diccionario tras codificar la cadena `ababcbababaaaaaa` es presentado en la Figura 2.10.

Dado que el compresor emplea la mayor parte de su tiempo en buscar las cadenas wk en el diccionario, es importante que los algoritmos de búsqueda sean eficaces. Los más frecuentemente usados son:

Búsqueda multicamino : Es la forma de búsqueda más común y se utiliza en implementaciones como la de *compress* o las de Nelson y Gailly [64]. Estos últimos autores han proporcionado como dominio público dos implementaciones diferentes del algoritmo LZW llamadas *lzw12* y *lzw15v* (que serán evaluadas al final del capítulo como compresores de imágenes).

Esta técnica de búsqueda consiste en acceder al diccionario usando un árbol multicamino, en el que cada nodo puede tener hasta 256 hijos (suponiendo, como es lo habitual, un alfabeto con 256 símbolos). El número de nodos recorridos para encontrar una cadena coincide con la longitud de la cadena. Dentro de cada nodo se usa *hashing* para reducir el tiempo de búsqueda, aunque también es posible utilizar búsqueda binaria si estos se mantienen ordenados. En este último caso, el tiempo de búsqueda es proporcional a $\log_2(N)$ donde N es el número de cadenas almacenadas en el diccionario.

Búsqueda binaria : Se basa en mantener siempre el diccionario ordenado y arroja tam-

bién una complejidad de $\log_2(N)$ donde N es el número de cadenas en el diccionario. Nuestra implementación del algoritmo LZW emplea esta técnica. En adelante conoceremos esta implementación por *lzw* [98].

Generalmente, el diccionario no se almacena físicamente ordenado, sino que se define una estructura de datos adicional que puede ser considerada como un árbol binario y que se emplea para acelerar las búsquedas. Dicha estructura se presenta en la Figura 2.10 (a la derecha), para un ejemplo concreto. Puede comprobarse que en realidad existen 256 árboles binarios, uno para cada raíz. La columna “ \leq ” (en la Figura 2.10) indica para cada nodo del árbol (cada cadena en el diccionario) qué direcciones del diccionario almacenan cadenas lexicográficamente menores o iguales que la contenida en ese nodo, y la columna “ $>$ ” indica dónde encontrar las cadenas lexicográficamente mayores. Las hojas se representan mediante el código especial **ESC** ya que esta entrada del diccionario está reservada y nunca se almacena ninguna cadena en ella.

Para comprender el funcionamiento del árbol de búsqueda vamos a ver un ejemplo de cómo se buscaría la cadena **baba**. Antes de comenzar el ejemplo, debe tenerse en cuenta que el codificador LZW trata de formar cadenas, partiendo siempre de una raíz (ver paso 3.c.iii) a la que se concatena un nuevo carácter en cada iteración. Por tanto, la primera cadena que se busca en la creación de la cadena **baba** es $wk = ba$, donde $w = b = 98$ y $k = a = 97$. Para saber si esta cadena está en el diccionario, miramos en la entrada 98. Como $k \leq D[98].k$ ($a \leq b$, $97 \leq 98$), saltamos a la posición 258 haciendo $w \leftarrow 258$ ($w \leftarrow ba$, ver paso 3.b.i). Ahora buscamos en la entrada 258 y vemos que $k > D[258].k$ ($b > a$). Por tanto, saltamos a la posición 261 haciendo $w \leftarrow 261$ ($w \leftarrow bab$). Por último, buscamos la cadena $wk = baba$ donde $w = bab$ y $k = a$. Vemos que $k \leq D[261].k$ ($a \leq b$), por lo que hacemos $w \leftarrow 262$, donde está almacenada la cadena **baba**.

Búsqueda usando hashing : La cadena wk puede ser utilizada como una clave para acceder de forma aleatoria al diccionario de cadenas. Sin embargo, el rendimiento (temporal) de esta búsqueda es similar al rendimiento del árbol binario o del árbol multicamino, puesto que es imposible definir una función *hash* perfecta y sea necesario emplear algún método de manejo de colisiones. Las tasas de compresión tampoco son diferentes.

Sin embargo, en [99] proponemos el siguiente procedimiento: si tenemos una colisión y la cadena buscada no es hallada (porque existe otra en su lugar) podemos suponer que la cadena no está en el diccionario y emitir el código de compresión correspondiente. Como es lógico pensar, la tasa de compresión se ve negativamente afectada, pero el tiempo de compresión es menor. Además, en [99] se propone el uso de una clave de *hashing* cuantificada, que no tiene en cuenta los bits menos significativos de la concatenación de los números w y k (siendo este último el byte menos significativo). De esta forma, el compresor LZW se transforma en *lossy* y permite alcanzar tasas de compresión significativamente superiores a costa de una pérdida controlada.

da de información que podría permitirse en el caso de la compresión de imágenes. Designaremos a esta implementación *hlzw*.

Control del tamaño de las palabras

Para finalizar con la descripción de las tareas que efectúa el compresor, entre los posibles códigos que el compresor puede usar está el 256 (**ESC**), que se emite para sincronizar los procesos de compresión y descompresión. La implementación *lzw12* usa un diccionario con 4096 entradas y emite siempre códigos de 12 bits. Sin embargo, existe una forma fácil de mejorar notablemente el rendimiento del compresor si tenemos en cuenta que es posible emplear sólo el número de bits necesarios para referenciar una cadena dentro de él en función de su tamaño actual. Por ejemplo, al comienzo el número de cadenas es 257 y hasta que almacenamos 1024 cadenas en el diccionario podemos usar códigos de 9 bits. Cuando la cadena 1024 es insertada, el compresor emite un código **ESC** seguido de un número (0 por ejemplo) que indica esta circunstancia y tanto el compresor como el descompresor pasan a usar 10 bits/código de compresión. Realmente, esta situación no hay por qué comunicarla al descompresor, pero se comprobó que se disminuía significativamente el tiempo de descompresión sin que la tasa de compresión se viera perjudicada seriamente. El uso de este símbolo de sincronización es usado en *lzw*, *compress* y *lzw15v*.

Control del vaciado del diccionario

El símbolo especial **ESC** se usa también para indicar al descompresor además dos circunstancias especiales:

- Que ya no existen más caracteres que descomprimir y por lo tanto el proceso de descompresión debe finalizar.
- Que el diccionario debe ser vaciado. Esto último ocurre en nuestra implementación (*lzw*) siempre que el tamaño máximo del diccionario es excedido. Sin embargo, existe una forma más refinada de determinar esta situación (usada en *compress* y *lzw15v*). Consiste en comprobar si la tasa de compresión está siendo baja debido a que las propiedades probabilísticas de la fuente han variado.

Nosotros usamos en [98] la primera opción porque deseábamos evaluar también la alternativa más rápida. Puesto que solamente tres posibles órdenes pueden suceder al símbolo **ESC**, sólo dos bits son suficientes para codificar una de estas órdenes.

El descompresor

Pasemos ahora a describir el descodificador LZW. Dos nuevos registros llamados *code* y *old_code* son usados. Su algoritmo a grandes rasgos es el siguiente:

1. Insertar las posibles raíces en el diccionario.

entrada	code	w	k	salida	comentario
97	97			a	Iniciación
98	98	97	b	b	code < 257, $D[257] \leftarrow ab$
257	257	98	a	ab	code < 258, $D[258] \leftarrow ba$
99	99	257	c	c	code < 259, $D[259] \leftarrow abc$
258	258	99	b	ba	code < 260, $D[260] \leftarrow cb$
261	261	258	b	bab	code = 261, $D[261] \leftarrow bab$
97	97	261	a	a	code < 262, $D[262] \leftarrow baba$
263	263	97	a	aa	code = 263, $D[263] \leftarrow aa$

Figura 2.11: Ejemplo de descodificación de la cadena `ababcbababaaaaaa` usando LZW.

2. $code \leftarrow$ primer código de entrada.
3. Emitir $code$ pues se trata de un único carácter.
4. $old_code \leftarrow code$.
5. Mientras existan códigos de entrada:
 - (a) $code \leftarrow$ siguiente código.
 - (b) $w \leftarrow old_code$.
 - (c) si $code$ está en el diccionario:
 - i. Emitir la cadena almacenada en $code$, como se realiza en el paso 1.b del descompresor LZ78.
 - (d) Si no:
 - i. Emitir la cadena almacenada en wk .
 - (e) $k \leftarrow$ primer carácter emitido en la salida anterior.
 - (f) Insertar la cadena wk en el diccionario.
 - (g) $old_code \leftarrow code$.

El descompresor LZW es más rápido que el compresor porque buscar la cadena $code$ en el diccionario es muy sencillo. Basta con ver si $code$ es menor que el número de cadenas insertadas en ese momento. Sólo si lo es, entonces $code$ está en el diccionario [98].

La Figura 2.11 expone la descompresión de la cadena comprimida anteriormente. Como puede comprobarse en las Figuras 2.10 y 2.11, el descompresor reconstruye un diccionario idéntico al usado por el compresor sin que exista un paso explícito del diccionario en la secuencia de códigos.

2.2.5 Consideraciones sobre el uso y la eficiencia

Las excelentes velocidades de compresión y descompresión han provocado que los compresores basados en diccionario sean los más usados en compresión de texto. Sin duda, la posibilidad de reducir los ficheros en promedio un 50% a costa de muy poca CPU y memoria los hace muy atractivos.

La familia de compresores LZ78 posee además otra interesante característica que descubrimos durante la implementación de *hlzw*. En principio, puesto que usamos *hashing* sin manejo de colisiones y los códigos w eran diferentes, parecía lógico pensar que el descomprimidor iba también a ser distinto. Bien, pues esto no ocurre así.

El descomprimidor es independiente del algoritmo usado para la búsqueda de las cadenas porque en realidad no se efectúan búsquedas y lo que es más importante, también es independiente de la efectividad de dicho algoritmo de búsqueda. Esto significa que, independientemente de la complejidad del proceso de compresión, la velocidad de descompresión no está relacionada con este factor.

En consecuencia, las técnicas basadas en diccionario son ideales en aquellos casos en los que la velocidad de compresión no es el factor más importante y la velocidad de descompresión debe ser lo más elevada posible. Además, es posible modificar el algoritmo de búsqueda en el compresor y el descomprimidor no tiene que ser modificado. Por esta razón, el descomprimidor es un algoritmo universal capaz de descomprimir cualquier fichero comprimido usando un algoritmo de la familia LZ78.

Por desgracia, las velocidades de compresión que son alcanzadas tienen un precio. Todos los métodos de compresión expuestos son algoritmos del tipo “glotón” (*greedy*), que tratan de encontrar las cadenas más largas posibles en un instante determinado, pero carecen de “visión de futuro”, en el sentido de que no construyen diccionarios que permitieran alcanzar tasas de compresión absolutamente óptimas. Esta pérdida de optimidad está permitida porque las tasas de compresión que se alcanzarían no justifican en muchos casos los descensos de velocidad [37, 111]. Además, está demostrado que las tasas de compresión serían óptimas si la longitud de la secuencia de datos fuera infinita [136].

2.3 La compresión entrópica

Como se comentó al principio de este capítulo, la compresión entrópica asigna a cada símbolo un código de longitud promedio igual a la entropía de la fuente (ver Sección 1.1.4). Los compresores entrópicos constan de dos partes fundamentales [25, 89] cuya interacción se describe gráficamente en la Figura 2.12:

- Un modelo probabilístico M que asigna a cada símbolo de entrada s una probabilidad de ocurrencia $p(s)$, bajo las restricciones indicadas en la Sección 1.4.
- Un codificador C que realiza las funciones de traductor, capaz de asignar a cada símbolo de entrada s un código c de longitud ideal igual a la expresada por la Ecación (1.9). En otras palabras, la longitud debe ser igual a la entropía de la fuente

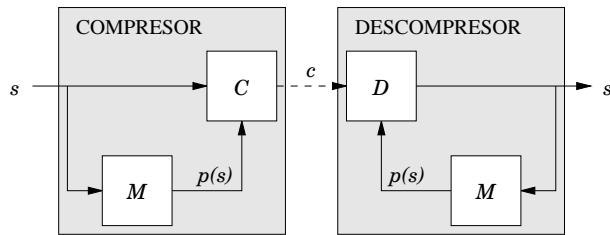


Figura 2.12: Modelo de codificación/descodificación entrópica.

expresada en bits de código/símbolo. El traductor inverso o descodificador D recupera la representación original de los símbolos, conociendo la misma información de contexto que el traductor directo, que es proporcionada por un modelo probabilístico idéntico al del descompresor.

La clave para obtener el máximo rendimiento en un compresor entrópico es la estimación exacta de las probabilidades de los símbolos. Si es demasiado baja, el código asignado aumenta de longitud y como el símbolo asociado ocurre en más ocasiones de las esperadas, la longitud de la secuencia-código aumenta. Por otra parte, si la probabilidad es demasiado alta, estaremos codificando un símbolo que no es tan frecuente como creemos mediante un código muy corto, a costa de alargar la longitud de los códigos asignados al resto de símbolos que tendrán asignada una probabilidad menor que la real.

El rendimiento de los compresores entrópicos depende tanto de los codificadores de longitud variable como de los modelos probabilísticos usados, pero su independencia es tal que pueden ser estudiados por separado. Por otra parte, los compresores entrópicos en general son más lentos que los basados en diccionarios. Esto se debe a que ahora se busca la máxima compresión posible, incluso a costa de desarrollar algoritmos pesados que permitan aprovechar ese “grado de optimalidad” del que hablamos en la sección anterior y que los compresores basados en diccionarios no explotan sobre secuencias de longitud finita.

Los algoritmos e ideas introducidos en esta sección son fundamentales para el diseño de los compresores de imágenes reversibles que serán analizados en el siguiente capítulo. Sin embargo, en éste se analizarán tres métodos de compresión que pueden ser considerados como el “estado del arte” en compresión de texto: PPM, BWT y PBT.

2.3.1 Un codificador entrópico universal

Antes de comenzar a estudiar los diferentes codificadores entrópicos usados en compresión de datos, vamos a adelantar la idea que es la esencia de todos ellos. Esto nos va a permitir comprender con más facilidad cómo funcionan, qué esperar de ellos y en qué se parecen todos los compresores entre si.

Como fue expuesto en la Sección 1.3.1, un código no es redundante si cada bit de información es representado exactamente por un bit de código. Los compresores basados

en diccionarios no estiman las probabilidades de los símbolos y por lo tanto es difícil entender que la afirmación anterior también es cierta en este caso, pero así es. Cuando las cadenas son sustituidas por índices, lo que en realidad se hace es codificar un gran símbolo-fuente compuesto por varios caracteres usando un código de longitud constante de al menos 1 bit.

El mismo efecto puede obtenerse en el caso de la codificación entrópica porque cada carácter se codifica usando un código de longitud variable. Ahora estimamos las probabilidades de los símbolos (caracteres) y construimos un código de longitud igual al número de bits de información correspondientes.

Por lo tanto, el objetivo de todo codificador es encontrar una representación lo más compacta posible para un determinado símbolo s . En teoría, debería ser posible asignar fracciones de bit de código dependiendo de su probabilidad de ocurrencia.⁴ Bajo estas condiciones, sea el siguiente algoritmo de codificación:

1. Mientras s no esté determinado sin incertidumbre (por el descodificador):
 - (a) Realizar una afirmación sobre s que permita al descodificador averiguar algo acerca de la identidad de s . Intentar que dicha afirmación tenga tantas posibilidades de ser cierta como de ser falsa.
 - (b) Emitir un bit de código indicando el resultado de la afirmación.

El descodificador asociado debería constar de los siguientes pasos:

1. Mientras s no esté determinado sin incertidumbre:
 - (a) Realizar la misma afirmación que el codificador.
 - (b) Recibir un bit de código indicando el resultado de la afirmación.

No es difícil apreciar que si se cumple la premisa de que la afirmación sea equiprobable, estaremos encontrando un código 100% eficiente. Sin embargo, lo que ocurre en la práctica es que formular afirmaciones ciertas exactamente en un 50% es muy complicado y por tanto el código contiene cierto nivel de redundancia.

Planteemos una implementación sencilla de los anteriores algoritmos. Supongamos que trabajamos con un alfabeto de 256 símbolos y que conocemos sus probabilidades ya que existe un modelo que las calcula. El paso no trivial que hay que describir es el de la formulación de la afirmación equiprobable. Supongamos que ordenamos los símbolos por su probabilidad de forma decreciente. Una afirmación lo más equiprobable posible sería: “el símbolo a codificar pertenece al conjunto formado por uno o más símbolos que se forma cuando recorremos la lista de símbolos y la suma de las probabilidades de todos ellos alcanza el valor más próximo a 0.5”.

⁴La ventaja principal de los compresores basados en diccionarios sobre los entrópicos es que al extender la fuente no es necesario trabajar con fracciones de bits lo que produce el correspondiente aumento en velocidad.

El descodificador puede hacer exactamente la misma predicción porque para formarla no se usa el símbolo codificado. Así, con la llegada del bit, el descodificador conoce si el símbolo está en el primer conjunto o en el segundo. El símbolo es encontrado cuando en el conjunto seleccionado sólo existe un símbolo.

Una forma eficiente de realizar las particiones (binarias) en nuestra lista ordenada es representarla en como un árbol binario en el que los símbolos más probables estén más cerca de la raíz y los más improbables estén lo más alejados de ésta como sea posible. En realidad, justamente esto es lo que hace la codificación de Huffman.

2.3.2 La codificación de Huffman

El código de Huffman [49] fue inventado por David A. Huffman en 1952 y desde entonces ha sido intensivamente utilizado en compresión de datos, porque genera un código de longitud entera instantáneo y óptimo.

El codificador

Huffman ideó un método para construir un árbol binario donde cada hoja representa a un símbolo, con la propiedad de que la distancia de una hoja s a la raíz del árbol es exactamente

$$\lceil -\log_2 p(s) \rceil. \quad (2.1)$$

A continuación asignó un dígito binario a cada rama del árbol y diseñó así un código de longitud variable que representa a los símbolos más probables con un código más corto y viceversa. El algoritmo de construcción del árbol de Huffman es el siguiente:

1. Crear una lista con tantos nodos como símbolos diferentes vayamos a codificar. Cada nodo representa a un símbolo diferente y almacena su probabilidad.
2. Mientras existan al menos dos nodos en la lista:
 - (a) Extraer de la lista los dos nodos con menor probabilidad. Si existen más de dos, la elección es irrelevante.
 - (b) Insertar en la lista un nodo que sea padre de los dos nodos extraídos, formando un árbol binario equilibrado de tres nodos. Este nuevo nodo tiene una probabilidad que es la suma de las probabilidades de los hijos y no representa a un símbolo en concreto.

La Figura 2.13 muestra un ejemplo de construcción de un árbol de Huffman. Por comodidad trabajaremos con pesos enteros y no con probabilidades reales. Supongamos que codificamos 5 símbolos representados por A, B, C, D y E con pesos 15, 7, 6, 6 y 5 respectivamente. El primer paso (Figura 2.13-a) consiste en crear una lista de 5 nodos. Gráficamente esta lista estará formada por los nodos que queden en el nivel superior.

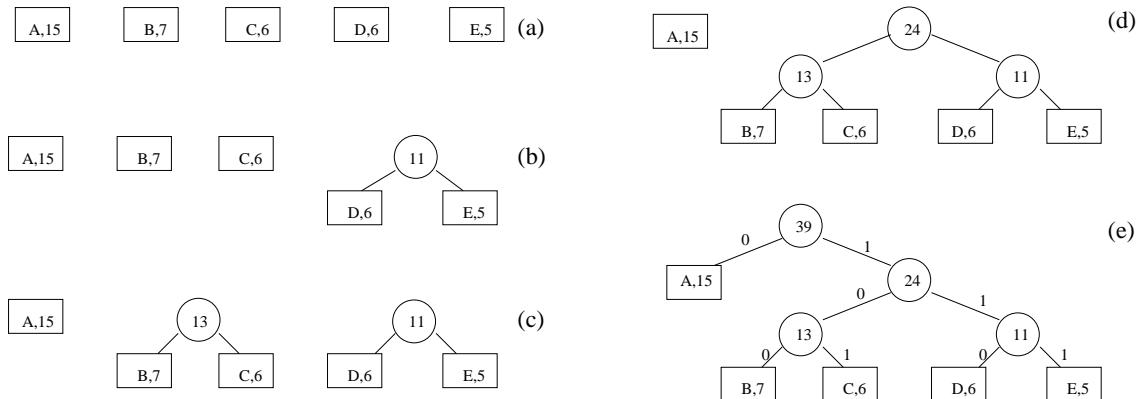


Figura 2.13: Ejemplo de construcción de un árbol de Huffman.

A continuación extraemos los nodos con símbolos D y E, que son los de menor peso. También podríamos seleccionar el nodo C y el árbol de Huffman resultante sería distinto pero equivalente desde el punto de vista de la compresión de datos. Los pesos de D y E son 6 y 5 y por lo tanto, insertamos un nuevo nodo en la lista con peso 11 (Figura 2.13-b). Los dos siguientes nodos utilizados para construir un nuevo subárbol son el B y el C (Figura 2.13-c). A continuación, los dos subárboles forman un árbol mayor (Figura 2.13-d) y por último, el nodo A se procesa quedando el árbol de Huffman completo (Figura 2.13-e). El número de iteraciones necesarias para crear un árbol de Huffman que codifique N símbolos es N , ya que en cada iteración extraemos dos nodos de la lista e insertamos uno.

Una vez empleada la información que proporciona el modelo para construir el árbol es posible codificar y decodificar. La codificación de un símbolo consiste en emitir el código formado por la concatenación de las etiquetas de cada una de las ramas que es necesario recorrer desde la raíz hasta la hoja que representa al símbolo. Por ejemplo, si las ramas izquierdas se etiquetan con el bit de código 0 y las derechas con el 1 (ver Figura 2.13-e), el código de Huffman calculado es $c_A = 0$, $c_B = 100$, $c_C = 101$, $c_D = 110$ y $c_E = 111$.

El descodificador

La decodificación es justamente el proceso inverso. Un código describe con los bits que lo forman un camino desde la raíz hasta una hoja del árbol de Huffman. El símbolo decodificado es el asociado a la hoja.

Por ejemplo, supongamos que codificamos la secuencia de símbolos ABACDE. La secuencia de códigos resultante es (teniendo en cuenta el árbol de Huffman anteriormente construido): 01000101110111. Para decodificarla, inicialmente nos situamos en la raíz del árbol. Como el primer bit es un 0, el primer símbolo decodificado es una A (ver Figura 2.13). Para decodificar el siguiente símbolo nos situamos de nuevo en la raíz y utilizamos los bits 100 para llegar hasta el símbolo B. Este proceso se repite hasta que no existen más bits de código que decodificar.

2.3.3 Compresión adaptativa

La longitud de los códigos de Huffman depende de la correcta estimación de las probabilidades de los símbolos. Existen fuentes que emiten los símbolos cuyas probabilidades permanecen constantes (fuentes ergódicas, ver Sección 1.1.3) y por lo tanto puede usarse un único árbol de Huffman para codificarla eficientemente. El modelo probabilístico se calcula una única vez o es conocido de antemano tal como ocurre en la compresión de imágenes [67, 93, 117]. Los modelos que permanecen inalterables durante el proceso de compresión se llaman modelos estáticos.

Cuando las distribuciones de probabilidad no son conocidas de antemano, la utilización de modelos estáticos requiere que la secuencia a comprimir sea recorrida dos veces. En la primera pasada se calculan las probabilidades de los símbolos y en la segunda se codifica. El modelo debe ser comunicado de forma explícita al descompresor, típicamente en forma de cabecera previa a la secuencia de códigos.

El otro tipo de modelos probabilísticos que existen se llaman modelos dinámicos o adaptativos. En general proporcionan mejores tasas de compresión que los estáticos debido fundamentalmente a que las fuentes de información no son ergódicas. Sin embargo, también provocan procesos de compresión y descompresión más lentos, pues hay que emplear recursos para gestionar el modelo.

La ventaja más interesante de usar modelos adaptativos es que pueden estimar las probabilidades de los símbolos sin necesidad de realizar dos pasadas,⁵ sin embargo, cuando la fuente es ergódica, los modelos estáticos son realmente eficientes, su rendimiento es comparable al de los no adaptativos [17, 44, 119].

La idea que hay detrás de la compresión entrópica adaptativa es la misma que la que usan los compresores basados en diccionarios: codificar usando sólo la información que hasta ese instante conoce el descompresor. Un algoritmo de compresión adaptativo realiza los siguientes pasos:

1. Inicialmente todos los símbolos son equiprobables y su probabilidad es mayor que 0.
2. Mientras existan símbolos que codificar:
 - (a) Codificar el siguiente símbolo.
 - (b) Actualizar la probabilidad del símbolo codificado.

Mientras que el algoritmo de descompresión consiste en:

1. Idéntico al paso 1 del compresor.
2. Mientras existan símbolos que descodificar:
 - (a) Descodificar el siguiente símbolo.
 - (b) Actualizar la probabilidad del símbolo descodificado.

⁵Diremos en estos casos que la secuencia puede ser tratada como una corriente (*stream*).

Actualización en el árbol de Huffman

La actualización de un símbolo en el árbol provoca que la probabilidad del símbolo actualizado y de todos los nodos internos que son antecesores (hasta llegar a la raíz) del símbolo sean incrementadas. La modificación de los pesos de los nodos puede provocar que los nodos afectados tengan que ascender por el árbol en dirección a la raíz, lo que acarrea una modificación de su estructura.

Usar el algoritmo de construcción presentado en la Sección 2.3.2 es demasiado costoso pues su complejidad es de $O(n^2)$ si n es el número de símbolos contemplados. Para acelerar este proceso, el árbol se representa de forma ordenada atendiendo a los pesos de los nodos usando un *array* $A[]$ con $2 \times n - 1$ elementos [57, 64, 101, 116]. El nodo raíz se almacena en $A[0]$, su hijo de mayor peso en $A[1]$ y el de menor en $A[2]$. A continuación se hace lo mismo con los hijos, comenzando por el de más peso, siempre que los nodos no sean hojas. Como resultado el *array* se ordena de forma decreciente y siempre deberá cumplir esta propiedad.

Usando $A[]$, es sencillo comprobar si la actualización de un símbolo provoca una modificación de la estructura del árbol porque esto sólo ocurre cuando el *array* deja de estar ordenado. En dicho caso, el nodo cuyo peso supera al que está por encima de él en el *array* debe ser intercambiado con el actualizado. Para comprender mejor este proceso, en la Figura 2.14-(A) se muestra un ejemplo.

Cuando incrementamos en una unidad el peso del símbolo A (lo que llamamos una actualización en un algoritmo de compresión adaptativo), tenemos que incrementar el peso también de su nodo padre, de su nodo abuelo y así sucesivamente hasta llegar a la raíz, tal y como se describe en la Figura 2.14-(B). Si incrementamos de nuevo el símbolo A se viola la propiedad de que se puedan recorrer los nodos en orden decreciente (se degenera el árbol). Por esta razón, antes de continuar con la actualización de los pesos hasta la raíz, los nodos 8 y 5 deben ser intercambiados (Figura 2.14-(C)). Después del intercambio el proceso de actualización continua, comprobándose en cada actualización el cumplimiento de la propiedad del recorrido ordenado. De esta forma se consigue el árbol de Huffman de la Figura 2.14-(D).

Hasta ahora, la forma del árbol no ha variado ya que el incremento del peso de una de las hojas es insuficiente para mover un nodo interno. Para que esto ocurra vemos que por ejemplo, el símbolo A debe ser incrementado al menos dos veces más. Tras el primer incremento las cosas quedan como se indica en la Figura 2.14-(E). Como puede verse no se incumple la propiedad del recorrido ordenado. Pero el segundo incremento de A provoca varios intercambios. En primer lugar, al acumular un peso igual a 5, debe ser intercambiado por el nodo almacenado en 4. Dicha circunstancia es la que se muestra en la Figura 2.14-(F). Tras el intercambio continua la propagación del incremento del nodo padre del símbolo A situado en la posición 2 (Figura 2.14-(G)). El incremento de este nodo provoca de nuevo la violación del recorrido ordenado y los nodos 1 y 2 deben ser intercambiados (Figura 2.14-(H)). Cuando éste ya se ha realizado, se produce el incremento del nodo raíz (Figura 2.14-(I)) y el árbol de Huffman queda construido (Figura 2.14-(J)).

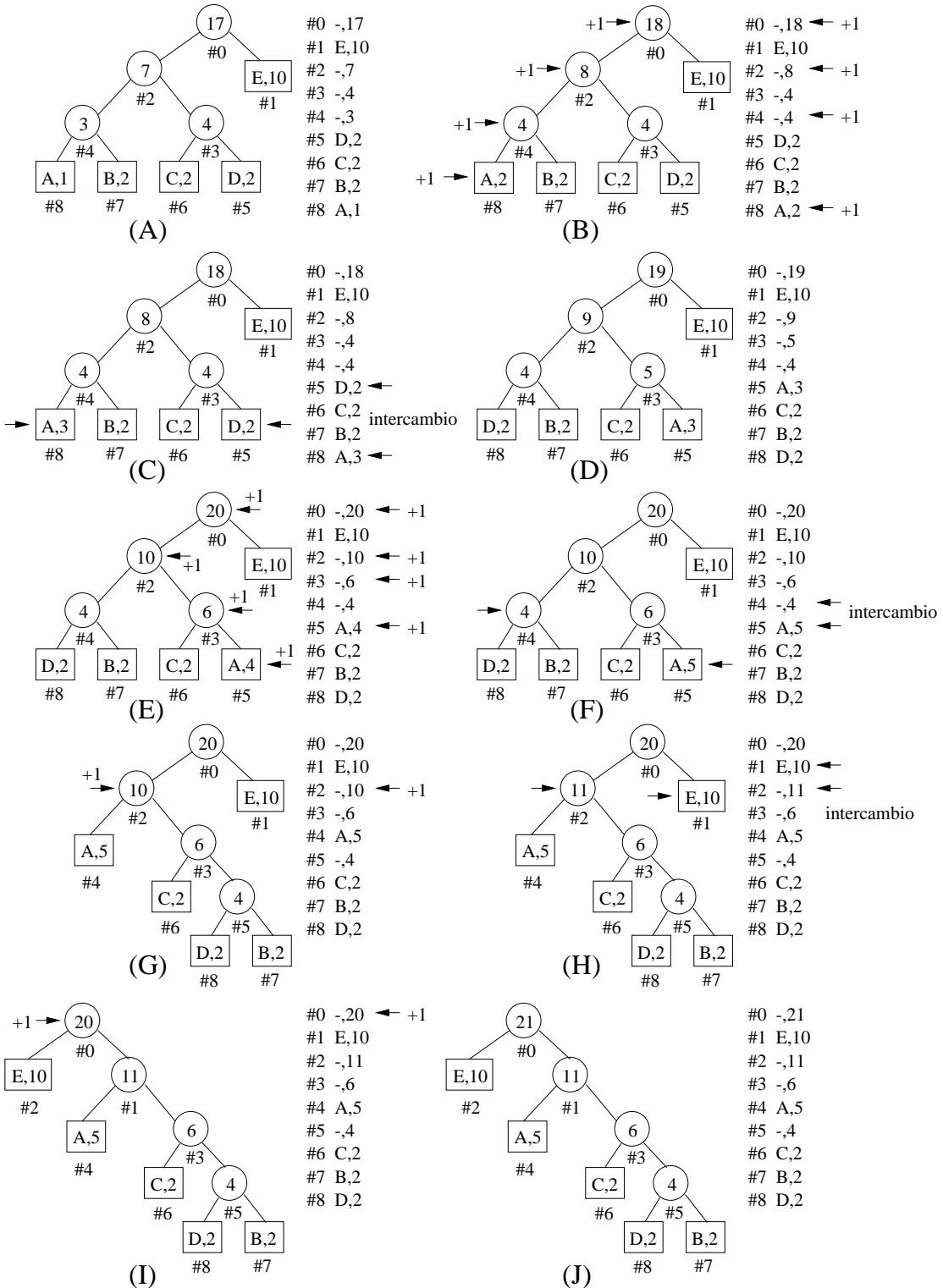


Figura 2.14: Ejemplo de actualización de un árbol de Huffman.

Debido a que normalmente se utilizan datos de tipo entero (con pocos bits de precisión) para almacenar la probabilidad de un símbolo, es necesario ejecutar algún proceso de escalado de todas las probabilidades de los nodos del árbol de Huffman y así evitar desbordamientos. Dicho proceso consiste típicamente en dividir (truncando) todos los recuentos entre 2. Esta operación no modifica la forma del árbol y además tiene un efecto beneficioso. Es frecuente encontrar secuencias de símbolos en las que muchos de ellos aparecen en zonas determinadas de la secuencia, como consecuencia de que las fuentes no son realmente ergódicas. La operación de escalado provoca que debido a los truncamientos, las probabilidades de estos símbolos sean menores que si no se produjera escalado, y esto ocurre precisamente cuando no son utilizados. Visualmente lo que ocurre es que dichos símbolos se “hunden” en el árbol más rápidamente y dejan que los símbolos más usados puedan codificarse con menos bits.

2.3.4 Modelos inicialmente vacíos

En la mayoría de las secuencias, la utilización de todos los símbolos que forman el alfabeto no se produce hasta que ya han sido codificados muchos de ellos. En estas circunstancias sería beneficioso para la tasa de compresión que el modelo contemplase únicamente los símbolos que han aparecido hasta ese instante, ya que las probabilidades de estos podrían ser mayores y por lo tanto, las longitudes de los códigos menores.

Para manejar la situación que se produce cuando aparece un nuevo símbolo, los modelos inicialmente vacíos hacen uso del símbolo especial escape (ESC) que jamás ocurre en la secuencia a comprimir. Así, cuando aparece un símbolo nuevo, el compresor codifica previamente un ESC, indicando al descompresor que debe insertar en el modelo probabilístico el símbolo que a continuación va a ser codificado.

Típicamente, al comienzo de la compresión el modelo probabilístico contempla además otro símbolo especial llamado fin de fichero (EOF) que se usa para indicar al descompresor que ya no existen más símbolos que descodificar. En el caso de usar la codificación de Huffman, inicialmente el árbol sólo posee dos hojas que contienen dichos símbolos.

Un algoritmo de compresión que utilice un modelo inicialmente vacío se compone de los siguientes pasos:

1. Contemplar un alfabeto inicial sólo con los símbolos ESC y EOF. Provocar que ESC sea mucho más probable que EOF.
2. Mientras existan símbolos que codificar:
 - (a) Si el símbolo ha ocurrido antes:
 - i. Codificarlo.
 - (b) Si no:
 - i. Codificar un ESC.
 - ii. Codificar el nuevo símbolo.

- iii. Añadir el símbolo al modelo probabilístico.
- (c) Actualizar la probabilidad del símbolo.
- 3. Codificar EOF.

El algoritmo de descompresión consiste en:

1. Idéntico al paso 1 del comprresor.
2. Mientras el símbolo EOF no sea descodificado:
 - (a) Descodificar el siguiente símbolo.
 - (b) Si el símbolo es ESC:
 - i. Descodificar el siguiente símbolo.
 - ii. Idéntico al paso 2.b.iii del comprresor.
 - (c) Idéntico al paso 2.c del comprresor
 - (d) Emitir el símbolo descodificado.

Los algoritmos de compresión y descompresión presentados son genéricos e independientes del codificador usado. En el caso de utilizar la codificación de Huffman, la inserción de un nuevo símbolo en el árbol se lleva a cabo como sigue: El nodo que lo representa se crea con peso 0, ya que por la semántica del comprresor y del descomprresor, su peso se incrementa tras la operación de inserción. Como el nuevo nodo tiene peso 0, es muy fácil encontrar el lugar dentro del array donde hay que colocarlo porque éste se encuentra ordenado. En concreto, el algoritmo de inserción es el siguiente:

1. Encontrar el nodo con peso menor. Sea i su posición.
2. Crear dos nuevos nodos $i + 1$ y $i + 2$.
3. Convertir i en un nodo interno (que no representa a un símbolo).
4. Hacer que $i + 1$ (por ejemplo) represente al nodo que antes representaba el nodo i .
5. Hacer que $i + 2$ represente al nodo insertado.

La Figura 2.15 muestra cómo habría que insertar el símbolo E en el árbol de Huffman de la izquierda de la figura.

2.3.5 El problema de la frecuencia cero

El uso del símbolo especial ESC para anunciar la llegada de nuevos símbolos también acarrea un nuevo problema para el modelo probabilístico llamado “el problema de la frecuencia cero” [92]. Cada vez que aparece un símbolo por primera vez, es necesario codificar un símbolo ESC, por lo que es importante estimar adecuadamente su probabilidad. Si es

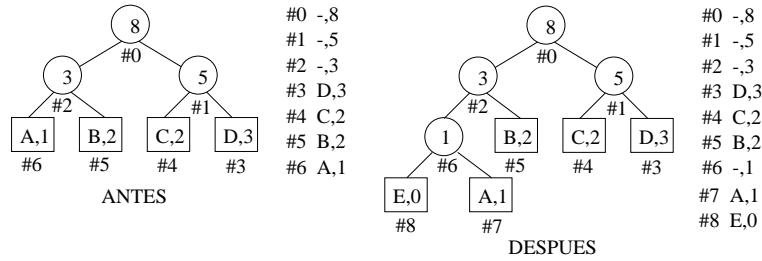


Figura 2.15: Ejemplo de inserción del símbolo E en un árbol de Huffman.

demasiado baja, el código asignado al símbolo ESC será excesivamente largo y sin embargo va a ser usado con frecuencia, mientras que si es demasiado alta, estaremos disminuyendo la probabilidad relativa de los demás símbolos respecto del ESC, lo que aumenta la longitud de los códigos asignadas a estos.

La estimación de la probabilidad del ESC es actualmente un problema abierto pues no existe una solución óptima para todos los casos. Realmente es imposible saber su probabilidad exacta, ya que eso implicaría conocer de antemano los símbolos novedosos que van a ocurrir en el resto de la secuencia. Sin embargo, se han propuesto básicamente tres métodos que han demostrado [124] una eficiencia generalizada:

Método A : Recibe este nombre porque fue el primero en ser probado [17]. Sea n el número de símbolos que forman la secuencia comprimida. La probabilidad de que un nuevo símbolo ocurra, o lo que es lo mismo, la probabilidad de emitir un ESC es

$$P(\text{ESC}) = \frac{1}{n+1}. \quad (2.2)$$

Este método funciona porque si n es pequeño (se ha recorrido una sección pequeña de la secuencia a comprimir), la probabilidad de encontrar un símbolo nuevo es alta y viceversa.⁶

Método B : Fue introducido en [16] y se basa en la siguiente idea. Si ya se ha codificado una buena parte de la secuencia y han ocurrido pocos símbolos diferentes (sea esta cantidad q), entonces la probabilidad de que el símbolo ESC sea utilizado es baja. Por el contrario, si se han usado muchos símbolos diferentes (q es grande), entonces la probabilidad de encontrar más símbolos nuevos también lo es. Por esta razón, el método B define la probabilidad del símbolo ESC como

$$P(\text{ESC}) = \frac{q}{n}. \quad (2.3)$$

Método C : Este método fue diseñado como un compromiso entre los métodos A y B [69, 71]. Considera que la probabilidad del ESC debe aumentar si el número de

⁶Dividimos entre $n + 1$ porque con el nuevo símbolo se han procesado $n + 1$ símbolos.

símbolos nuevos (q) es alto, pero también tiene en cuenta que cada vez que ocurre un símbolo nuevo (q es incrementado) la probabilidad de encontrar más se reduce, porque el tamaño del alfabeto es finito. Así, cuando ocurre un nuevo símbolo, se aumenta la probabilidad del ESC y del nuevo símbolo, obteniéndose que

$$P(\text{ESC}) = \frac{q}{n+q}. \quad (2.4)$$

Nótese que si la secuencia de símbolos es suficientemente larga entonces $n \gg q$ y por lo tanto $n \approx n + q$. En consecuencia, el método B y el C son muy parecidos entre sí cuando ya se han codificado muchos símbolos.

Con posterioridad han aparecido más métodos de estimación de la probabilidad del símbolo ESC [126], llamados D, P, X, XC, Z, etc. Todos ellos mejoran en general los propuestos inicialmente, aunque ninguno de ellos es el más eficiente en todos los casos. Bajo estas circunstancias, en la mayoría de las ocasiones el factor determinante a la hora de estimar la probabilidad del ESC es el tiempo y la cantidad de memoria que es necesario emplear.

2.3.6 Modelos con memoria y la necesidad de códigos mejores

La mayoría de las fuentes de información poseen memoria (ver Sección 1.1.3) y en consecuencia, la probabilidad de encontrar un determinado símbolo depende de los símbolos que han sido generados inmediatamente antes. Por ejemplo, en el idioma español, la ocurrencia de la cadena “qu” antecede frecuentemente al símbolo “e”. Este tipo de redundancia puede ser usada para calcular con mayor exactitud la probabilidad de un determinado símbolo.

Los modelos con memoria también se llaman modelos basados en el contexto y son útiles cuando existe correlación (estadística por ejemplo, aunque también es posible eliminar otros tipos de correlaciones) entre símbolos. Por el contrario, los modelos sin memoria (o modelos libres de contexto) son usados si no existe correlación entre símbolos. La mayoría de los modelos probabilísticos utilizados en compresión de texto explotan la correlación estadística.

El uso de modelos adaptativos basados en el contexto genera distribuciones de probabilidad que casi nunca verifican la Expresión (1.11) y por lo tanto, el codificador de Huffman introduce cierto nivel de redundancia en la fase de codificación de cada símbolo. La cantidad de redundancia introducida depende de la probabilidad del símbolo y aumenta a medida que crece su probabilidad. Para apreciar mejor este problema, la Figura 2.16 muestra la longitud de un código de Huffman en función de la probabilidad del símbolo s

$$L_{\text{Huff}}(s) = \lceil -\log_2 p(s) \rceil \quad (2.5)$$

y cual sería la longitud ideal

$$L_{\text{ideal}}(s) = -\log_2 p(s). \quad (2.6)$$

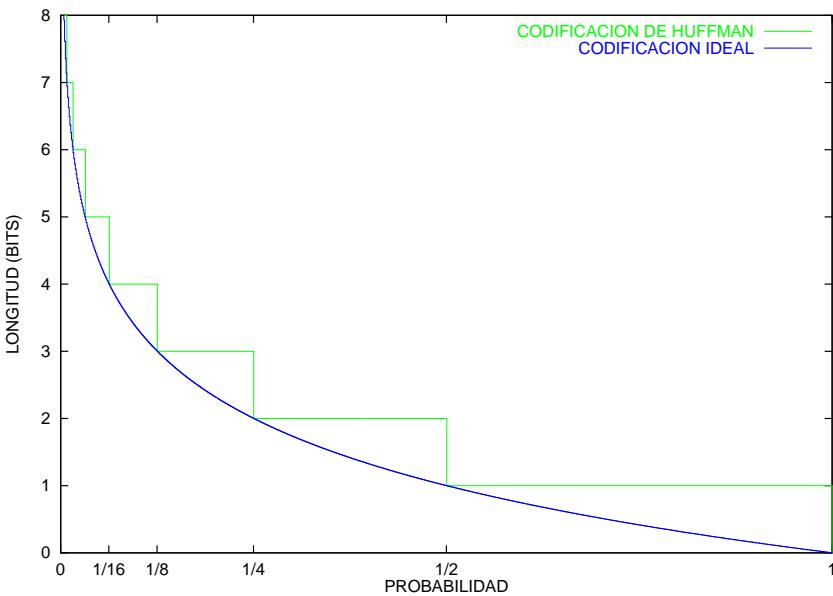


Figura 2.16: Longitud de un código generado por un codificador de Huffman y un codificador ideal en función de la probabilidad del símbolo asociado.

Por ejemplo cuando la probabilidad de un símbolo es 0.99, casi la mitad del código emitido es redundante en el caso de usar la codificación de Huffman.

La solución a este problema consiste en usar un código de longitud variable que fuera capaz de asignar fracciones de bit de código a los símbolos. Justamente esto es lo que consigue hacer la codificación aritmética.

2.3.7 La codificación aritmética

La codificación aritmética fue introducida por Abramson [2] en 1963 y desarrollada por Pasco [75] en 1976. Desde entonces, ha recibido una atención especial porque es una forma de representación de la información muy eficiente y permite una independencia máxima entre el modelo probabilístico y el codificador.

Supongamos que necesitamos codificar una fuente de información binaria (por ejemplo, una imagen de fax) que emite símbolos con probabilidades desiguales. La codificación de Huffman es incapaz de expresar eficientemente esta fuente de información porque representaría cada símbolo con 1 bit de código al menos que es el tamaño de la representación actual. La solución dada a este problema durante mucho tiempo ha consistido en extender primero la fuente (ver Sección 1.1.6) y luego usar la codificación de Huffman sobre un alfabeto mayor. La eficiencia de la solución sólo es ideal cuando el orden de la extensión es infinito y por tanto, nunca podemos representar la fuente sin redundancia.

Abramson, Pasco y más tarde Rissanen y Landon [61, 88] desarrollaron un método de

Tabla 2.1: Probabilidades asociadas a los símbolos del fichero **aab**.

símbolo	probabilidad
a	2/3
b	1/3

codificación que permitía codificar fuentes binarias sin necesidad de extender la fuente. El método se llamó codificación aritmética binaria. La idea es muy sencilla, aunque sobre su implementación *software* y *hardware* se han escrito decenas de artículos [34, 44, 45, 46, 61, 68, 78, 88, 89, 90, 91, 95, 128].

Consiste en asignar a cada posible secuencia de símbolos un subintervalo dentro del intervalo $[0, 1)$ cuyo tamaño es igual al producto de todas las probabilidades de los símbolos que forman la secuencia. La posición del subintervalo dentro del intervalo real $[0, 1)$ está determinado por la secuencia concreta de símbolos. El algoritmo básico de codificación consiste en emitir un número cualquiera perteneciente al subintervalo final. El número de bits de código necesarios para representar el código aritmético, es igual a la entropía de la secuencia de símbolos y por esta razón, la codificación aritmética se considera 100% eficiente.

El codificador ideal

A continuación se presenta el algoritmo básico de codificación que es una primera aproximación a la idea de la codificación aritmética.

1. Sea $[L, H) \leftarrow [0, 1)$ el intervalo inicial.
2. Mientras existan símbolos de entrada:
 - (a) Dividir el intervalo $[L, H)$ en tantos subintervalos como símbolos diferentes existen en el alfabeto. El tamaño de cada subintervalo es proporcional a la probabilidad del símbolo asociado.
 - (b) Seleccionar de entre todos los subintervalos, el que corresponde al símbolo codificado en la iteración actual. Sea el intervalo elegido $[L', H')$.
 - (c) Hacer $[L, H) \leftarrow [L', H')$.
3. Emitir un número $x \in [L, H)$ como código aritmético. El número de cifras deberá permitir distinguir el intervalo final $[L, H)$ de cualquier otro.

Supongamos el fichero **aab** del que se desprenden las probabilidades de la Tabla 2.1 que se calculan en función del número de veces que ocurre cada símbolo. Inicialmente $L = 0$ y $H = 1$. Dividimos (el proceso de divisiones se muestra en la Figura 2.17) el intervalo $[0, 1)$ en dos subintervalos: el $[0, 2/3)$ asociado al símbolo **a** y el $[2/3, 1)$ asociado

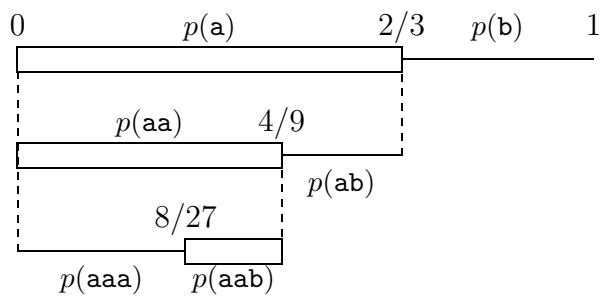


Figura 2.17: Intervalos generados por la codificación del fichero **aab**.

al símbolo **b**. Los intervalos son abiertos por arriba para evitar que se solapen, o de lo contrario la descodificación sería imposible si como código aritmético tuviéramos un límite de un intervalo. En la primera iteración codificamos una **a** por lo que hacemos $L = 0$ y $H = 2/3$. En la segunda iteración dividimos el intervalo actual $[0, 2/3)$ en los subintervalos $[0, 4/9)$ y $[4/9, 2/3)$. Ya que codificamos otra **a**, hacemos $L = 0$ y $H = 4/9$. En la tercera y última iteración dividimos el intervalo $[0, 4/9)$ en $[0, 8/27)$ y $[8/27, 4/9)$. Al codificar una **b** hacemos $L = 8/27$ y $H = 4/9$. Ahora buscamos un real $x \in [8/27, 4/9)$ de forma que tenga un número de cifras mínimo. Si utilizamos la base decimal, $x = 0.3 \in [0.2962929 \dots, 0.444 \dots]$. De igual forma x podría haber sido 0.4. Para encontrar el código aritmético expresado en binario (trabajando con 16 bits de precisión) efectuamos las operaciones

$$\lfloor 8/27 \times 2^{16} \rfloor = 19417 = 0100\ 1011\ 1101\ 1010$$

$$\lfloor 4/9 \times 2^{16} \rfloor = 29127 = 0111\ 0001\ 1101\ 0111.$$

El intervalo final expresado en binario sería

$$[0.0100\ 1011\ 1101\ 1011 \dots, 0.0111\ 0001\ 1101\ 0111).$$

Un real perteneciente a este intervalo y con un número de cifras mínimo es $x = 0.011$. Al tratarse siempre de un número menor que la unidad, la parte entera de la representación puede omitirse, lo que nos da que el fichero **aab** puede codificarse en decimal por 3 y en binario por 011.

El descodificador ideal

A continuación se expone el algoritmo de descodificación y un ejemplo:

1. $[L, H) \leftarrow [0, 1)$.
2. Mientras sea posible descodificar símbolos:
 - (a) Dividir el intervalo $[L, H)$ en subintervalos de tamaño proporcional a las probabilidades de los símbolos.

- (b) Seleccionar el subintervalo $[L', H']$ al que el código aritmético x pertenece.
- (c) Emitir el símbolo asociado al intervalo $[L', H']$.
- (d) $[L, H] \leftarrow [L', H']$.

A partir de las probabilidades de la Tabla 2.1 y del código aritmético 3 (0.3 en realidad, pues sabemos que se trata de un número mayor o igual que cero y menor que uno), vamos a realizar la descodificación del fichero. Inicialmente $L = 0$ y $H = 1$. Dividimos este intervalo en 2 subintervalos $[0, 2/3)$ y $[2/3, 1)$ (ya anticipamos que la secuencia de intervalos es idéntica a la obtenida durante el proceso de codificación). Seleccionamos el primer intervalo pues $0.3 \in [0, 2/3)$. Así, el primer símbolo descodificado es una **a**. Tomamos como intervalo actual dicho intervalo que es dividido en $[0, 4/9)$ y $[4/9, 2/3)$. Como $0.3 \in [0, 4/9)$, el segundo símbolo descodificado es una **a**. De nuevo dividimos este intervalo en $[0, 8/27)$ y $[9/27, 4/9)$. Como $0.3 \in [8/27, 4/9)$, el último símbolo codificado fue una **b**. En total, la secuencia descodificada es **aab**.

Los algoritmos reales

Los algoritmos ideales presentados tienen graves problemas de implementación. El principal es que el codificador genera el código aritmético cuando se ha encontrado el subintervalo final. Trabajando con registros de tamaño finito, la precisión que estos son capaces de proporcionar es insuficiente y rápidamente L y H se hacen virtualmente iguales. Una solución consistiría en usar aritmética de precisión infinita, pero los algoritmos serían tremadamente lentos.

La mejor solución propuesta hasta ahora se debe a Pasco [75] y se llamó transmisión incremental. Ya que en cada iteración L y H se parecen más entre si, es inútil almacenar los bits más significativos de ambos registros pues estos son inalterables durante el tiempo que dura la codificación. De hecho, estos bits forman parte del código aritmético y por lo tanto, en el modelo de transmisión incremental se transmiten en cuanto son conocidos.

El segundo gran problema está provocado por el uso de aritmética en punto flotante, cuando en realidad es posible usar aritmética entera que es más rápida. Las modificaciones necesarias en el codificador y el descodificador son mínimas. Realmente, sólo hay que imaginar que existen N bits de resolución para realizar los cálculos y que las divisiones necesarias para seleccionar el siguiente subintervalo son enteras. Esto provoca que exista cierto error, debido a los truncamientos, que puede alterar la longitud de los subintervalos y por lo tanto a la longitud del código aritmético. El problema del error por truncamiento será analizado posteriormente en esta sección.

El codificador de Witten

La filosofía de cálculo empleada en el codificador que vamos a estudiar está inspirada más en el trabajo de Witten *et al.* [128] que en el de Rissanen y Langdon (aunque ambas son equivalentes). Se ha seleccionado la versión de Witten porque desde el punto de vista de

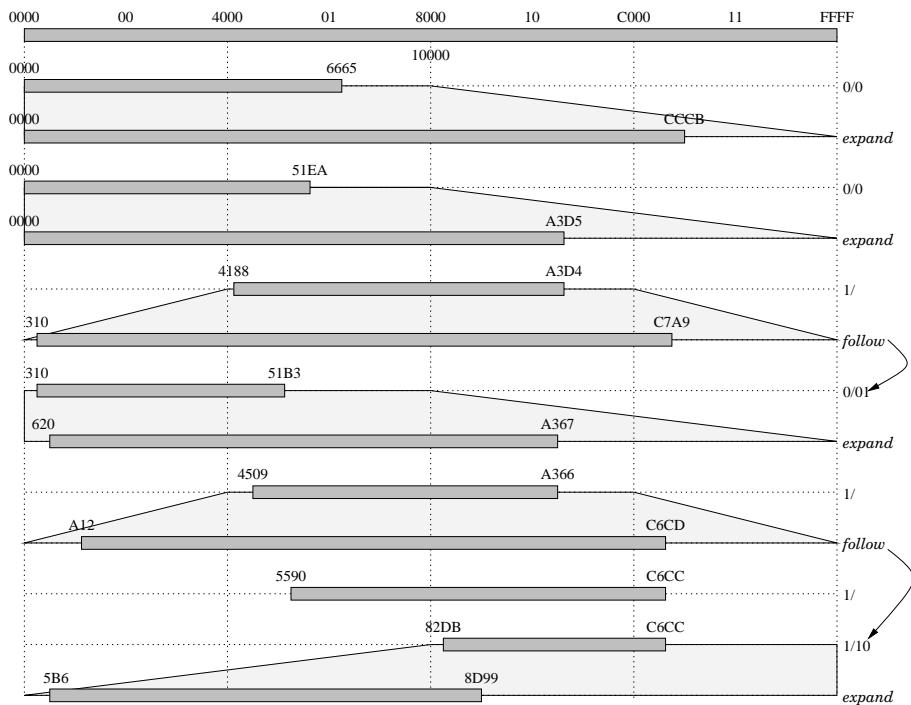


Figura 2.18: Ejemplo de codificación aritmética de la secuencia 0010111 donde $p(0) = 0.4$.

una implementación *software* es más sencilla y además permite la codificación de alfabetos con hasta 256 símbolos.

En la Figura 2.18 presentamos un ejemplo de codificación aritmética para el caso de un alfabeto binario en el que la probabilidad del símbolo 0 es 0.4 (y en consecuencia la probabilidad del símbolo 1 es 0.6). Se ha supuesto también que $N = 16$ (la precisión de los registros enteros es 16 bits). Usaremos la notación hexadecimal para acortar las expresiones numéricas y siguiendo con la especificación de Witten, el valor de H será inferior en una unidad a su valor real, con el fin de aprovechar al máximo la capacidad de los registros enteros.

Inicialmente $L = 0000$ y $H = \text{FFFF}$ y el intervalo actual es $[0000, 10000]$ o $[0000, \text{FFFF}]$. Codificamos un 0 y seleccionamos el subintervalo $[0000, 6665]$. El bit más significativo de L y H es 0 y es emitido, desplazando a la izquierda ambos registros. En L insertamos un 0 por la derecha y en H un 1 (debe tenerse en cuenta que a H siempre tenemos que restarle 1 al trabajar con intervalos cerrados). Las operaciones de desplazamiento y relleno con nuevos bits es lo que se ha llamado operación de expansión (*expand* en la Figura 2.18), provocando que $L = 0000$ y $H = \text{CCCB}$. A continuación codificamos un 0 y seleccionamos el intervalo $[0000, 51EA]$. El bit más significativo de L y H vuelve a ser 0 y se produce otra operación de expansión que genera el intervalo $[0000, A3D5]$. Ahora codificamos un 1 y seleccionamos el intervalo $[4188, A3D4]$ (realmente el intervalo seleccionado debiera haber

sido [4188, A3D5], pero por problemas de truncamiento de la división entera, el intervalo se hace ligeramente más pequeño). Los bits más significativos de L son 01 y los de H 10. En esta situación, el proceso de codificación sufre el peligro de *underflow* [64, 128]. L y H podrían acercarse tanto que fuera imposible representar el siguiente subintervalo por falta de precisión.

Witten *et al.* propusieron un algoritmo llamado *follow* que consiste en restar a L y a H el valor $10000/4 = 4000$ (el tamaño del intervalo inicial dividido entre 4), con lo cual el intervalo actual que se coloca alrededor del valor 8000 ahora se coloca alrededor del valor 4000 sin que su tamaño se vea afectado. A continuación se produce una expansión pero no se emite ningún bit de código. El intervalo generado tras la operación *follow* es el [310, C7A9]. El siguiente 0 de entrada selecciona el intervalo [310, 51B3]. L y H comienzan por 0 y éste es emitido (lo que provoca una expansión) y a continuación se emiten tantos bits contrarios a éste como iteraciones se ha estado en peligro de *underflow* tras efectuar la última operación *follow*.

Como sólo hemos estado en situación de *underflow* durante una iteración, se emite un único 1. Tras la expansión provocada por la emisión del último 0, se consigue el intervalo [620, A367]. Ahora codificamos un 1 y entramos de nuevo en situación de peligro de *underflow*. Restamos 4000 a L y a H y expandimos (*follow*), quedando $L = A12$ y $H = C6CD$. A continuación codificamos otro 1 y seleccionamos el subintervalo [5590, C6CC]. No se emite nada porque los bits más significativos de L y H son distintos. Codificamos otro 1 y seleccionamos el intervalo [82DB, C6CC]. Ahora si emitimos un 1 seguido de un 0 debido a la situación de *underflow* anterior. Tras la expansión por el 1 emitido, se alcanza el intervalo [5B6, 8D99].

Seguidamente se presenta el algoritmo de Witten *et al.* [128], que fue desarrollado para alfabetos con 256 símbolos y para $N = 16$, por lo que L y H tienen 16 bits de precisión. El codificador consiste en:

1. Sea $L \leftarrow 0000$, $H \leftarrow FFFF$ y $follow \leftarrow 0$.
2. Mientras existan símbolos de entrada:
 - (a) Sea s el símbolo y sea $p(s)$ su probabilidad. La selección del intervalo asociado se consigue a partir de las probabilidades acumuladas $P_L(a) = \sum_{i=0}^{s-1} p(i)$ y $P_H = P_L + p(s)$ haciendo que $H \leftarrow L + P_H \times R - 1$ y $L \leftarrow L + P_L \times R$, donde $R = H - L + 1$ es el rango del subintervalo actual.
 - (b) Ejecutar en un lazo los siguientes pasos:
 - i. Si el bit más significativo de L es igual al bit más significativo de H , entonces:
 - A. Emitir el bit más significativo.
 - B. Emitir tantos bits contrarios al bit más significativo como indique el contador *follow*.
 - C. $follow \leftarrow 0$.

- ii. Si no (si los bits más significativos no coinciden):
 - A. Si el segundo bit más significativo de L es distinto al segundo bit más significativo de H , entonces:
 - (α) Incrementar el contador *follow*.
 - (β) Desplazar el intervalo actual haciendo $L \leftarrow L - 4000$ y $H \leftarrow H - 4000$.
 - B. Si no (si los bits más significativos no son iguales, ni los segundos):
 - (α) Ir al paso 2.
- iii. Expandir el intervalo haciendo $L \leftarrow L \times 2$ y $H \leftarrow H \times 2 + 1$.

El descodificador de Witten

El proceso de descodificación aritmética funciona “en simpatía” con el codificador. Se genera la misma secuencia de subintervalos y en cada iteración se emite el símbolo cuyo intervalo contiene la parte del código aritmético que se está descodificando.

Comencemos descodificando el ejemplo anterior. Inicialmente partimos del intervalo [0000, FFFF]. Como conocemos las probabilidades de los símbolos y sólo existen dos (en nuestro ejemplo), sabemos que el intervalo asociado al símbolo 0 es el [0000, 6665] y el intervalo asociado al símbolo 1 es el [6666, FFFF]. Los dos primeros bits del código aritmético son 00 (almacenados en un registro de 16 bits que llamaremos V y que contiene inicialmente los 16 primeros bits de código aritmético) lo cual indica sin lugar a dudas que el primer símbolo a descodificar es el 0 ya que es el único intervalo (de los dos posibles) que fue capaz de emitir dos ceros consecutivos. Se expande el primer intervalo obteniéndose el [0000, CCCB], lo cual genera la pérdida del bit más significativo de L , H y V debido a una operación de desplazamiento de bits a la izquierda. Los bits más significativos de L y H deben ser descartados para que puedan ser obtenidos los mismos intervalos generados por el codificador. El bit más significativo de V se olvida porque ya ha sido utilizado para descodificar el primer símbolo. Por lo tanto, el contenido actual de V es 00 \dots , ya que el tercer bit de código aritmético es otro 0. A continuación se partitiona el intervalo actual en los subintervalos [0000, 51EA] y [51EB, CCCC]. Como $V = 00\cdots$, el intervalo seleccionado es el primero y el siguiente símbolo descodificado es un 0. Expandimos y provocamos que $L = 0000$, $H = A3D5$ y $V = 0110\cdots$. Se partitiona de nuevo el intervalo actual y obtenemos los subintervalos [0000, 4188] y [4189, A3D4]. El valor de V indica que el intervalo usado por el codificador fue el segundo con lo que se descodifica un 1.

Del ejemplo se deduce que el algoritmo de descodificación es similar al de codificación, aunque no hay que tener en cuenta las situaciones de *underflow*. Por otra parte, es necesario utilizar un registro extra llamado V que en cualquier instante contiene los bits de código aritmético que van a ser usados para seleccionar un subintervalo y descodificar así el siguiente símbolo. Como $N = 16$, el número de bits necesarios para discriminar entre cualquier posible intervalo es 16 y éste es el tamaño del registro V . Por lo tanto, el algoritmo de descodificación aritmética multisímbolo propuesto por Witten es el siguiente:

1. Sea $L \leftarrow 0000$, $H \leftarrow FFFF$ y $V \leftarrow 16$ primeros bits de entrada.

2. Mientras existan símbolos que descodificar:
 - (a) Encontrar el símbolo cuyo intervalo contiene a V .
 - (b) Calcular el siguiente intervalo de forma análoga al paso 2.a del codificador.
 - (c) Ejecutar en un lazo los siguientes pasos:
 - i. Si el bit más significativo de L y H no son iguales, entonces:
 - A. Si el siguiente bit más significativo de L y H no son iguales:
 - (a) Desplazar el intervalo actual haciendo $L \leftarrow L - 4000$ y $H \leftarrow H - 4000$.
 - B. Si no:
 - (a) Ir al paso 2.
 - ii. Expandir el intervalo haciendo $L \leftarrow L \times 2$ y $H \leftarrow H \times 2 + 1$.
 - iii. Insertar por la izquierda el siguiente bit de código aritmético en V haciendo $V \leftarrow V \times 2 + x$ donde x es el siguiente bit de código aritmético.

Independencia entre el modelo y codec

Como se deduce de los algoritmos de codificación y descodificación aritmética, la libertad de cambio de las probabilidades de los símbolos es absoluta (ver pasos 2.(a) del codificador y 2.(b) del descodificador). El tamaño y situación del siguiente subintervalo depende de las probabilidades de los símbolos que es la única información que debe proporcionar el modelo. Si el modelo es adaptativo, los pasos a realizar en la codificación y la descodificación son idénticos a los descritos. Esto también es cierto si el modelo está basado en el contexto.

La independencia total entre el modelo y el *codec* es una de las principales ventajas de la codificación aritmética frente a la codificación de Huffman.

Análisis del error por truncamiento

El paso 2.a del codificador necesita realizar una operación de división en punto fijo porque las probabilidades de los símbolos se calculan como el cociente entre dos recuentos: (1) el recuento del símbolo y (2) el recuento total de todos los símbolos (el tamaño del fichero). El porqué se usan recuentos y no probabilidades reales se debe (1) a que trabajar con números reales aumentaría significativamente el tiempo que el modelo dedica a la operación de actualización (ver Sección 2.3.4), (2) a que complicaría la operación de escalado que tan interesante es para la codificación de fuentes no ergódicas y (3) a que se alargaría el tiempo de selección del siguiente subintervalo.

La cantidad de error debido al truncamiento en la operación de división depende de la precisión de los registros N y es inevitable (incluso si se usara aritmética en punto flotante, que involucra redondeos). En consecuencia, los registros L y H siempre van a tomar valores iguales o más pequeños que los ideales. Estas diferencias van a afectar al tamaño, a la posición del intervalo o a ambas simultáneamente. Cuando sólo la posición

sea afectada (porque L y H se truncan exactamente en la misma cantidad), el código aritmético se verá afectado en su composición, pero no en su longitud. Sin embargo (y como es probable que ocurra) si el tamaño es menor, la longitud del código aumentará.

Cuando el intervalo seleccionado es el de más a la derecha, el registro H no debería variar. Sin embargo, como puede apreciarse en la Figura 2.18, hay ocasiones en las que debido a errores por truncamiento en las divisiones, el valor de H es decrementado en 1. En consecuencia, el intervalo calculado es más pequeño que el ideal. Sin embargo, cuando el intervalo seleccionado es el que está más a la izquierda, L no se mueve a la izquierda y H sigue sufriendo el problema anterior. Este comportamiento del codificador aritmético provoca que en general los intervalos calculados sean de menor longitud que los ideales. En general, cuanto menor es N , mayores son los acortamientos. Witten *et al.* estimaron empíricamente que para $N = 16$, la longitud del código aritmético aumenta a lo sumo en 10^{-4} bits de código/símbolo codificado, por lo que el rendimiento del codificador a pesar de usar sólo 16 bits de precisión en las operaciones aritméticas es excelente.

La codificación aritmética binaria

Históricamente la codificación aritmética binaria ha recibido una atención especial debido a las siguientes razones:

- El cálculo del intervalo siguiente sólo afecta en una iteración a L o a H , pero nunca a ambos: si el símbolo a codificar es un 0, sólo hay que calcular H y viceversa. El tiempo de cálculo del intervalo siguiente se divide por tanto por la mitad aproximadamente.
- Se puede utilizar para codificar fuentes de información multisímbolo si construimos un árbol de Huffman en el que cada nodo tiene asociada una probabilidad de transición hacia su rama izquierda o derecha [42]. Se usa un árbol de Huffman porque esta estructura minimiza el número de decisiones para codificar un símbolo en función de su probabilidad. Cada vez que atravesamos un nodo interno, se codifica aritméticamente la probabilidad de la rama escogida que depende del nodo asociado.
- Los modelos probabilísticos para dos símbolos son muy sencillos ya que con una única probabilidad se describe a la fuente. Además, en el proceso de descodificación la búsqueda del subintervalo que contiene el código aritmético se trivializa porque sólo existen dos alternativas.

Estas circunstancias (y en especial la última) han provocado que la mayoría de las implementaciones físicas sean para codificadores y descodificadores binarios y el ejemplo más claro lo tenemos en el Q-Coder [7, 60, 68, 76, 77, 78]. Sin embargo, cuando los alfabetos son multisímbolo (256 típicamente), el codificador aritmético multisímbolo supera en velocidad a la versión binaria. Esto es especialmente cierto en el caso de una implementación *software* en la que es mucho más determinante el número de instrucciones ejecutadas que la complejidad aritmética de éstas. Incluso en el caso de una codificación de fuentes

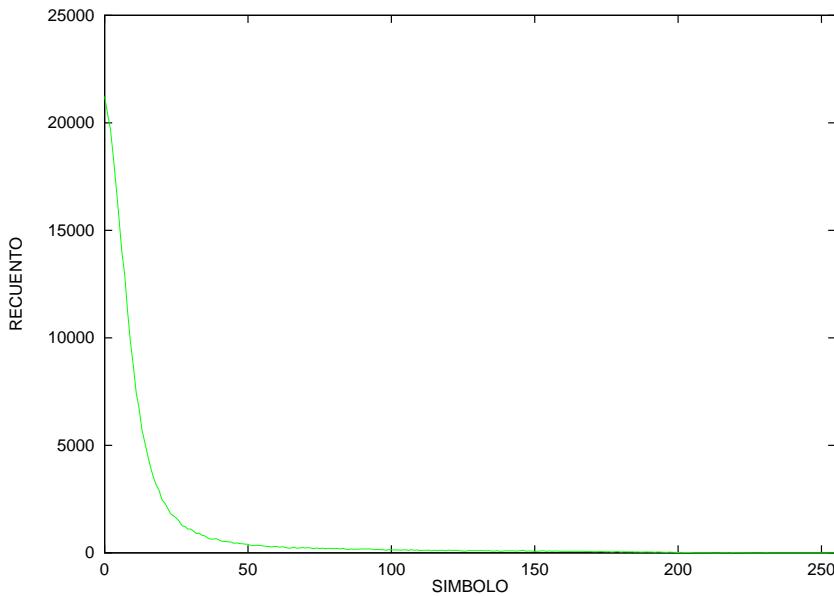


Figura 2.19: Histograma de una secuencia de símbolos que sigue una distribución de probabilidad geométrica (SDPG).

binarias, la menor velocidad de funcionamiento del algoritmo multisímbolo puede ser superada procesando la fuente de forma extendida, incluso tratándose de implementaciones hardware [74].

2.3.8 Codificación de secuencias con una distribución de probabilidad geométrica (SDPG)

En compresión de datos y especialmente en compresión de imágenes, es muy frecuente encontrar secuencias de símbolos que se distribuyen siguiendo aproximadamente una función exponencial decreciente (ver Figura 2.19). Decimos que se trata de secuencias con Distribuciones de Probabilidad Geométrica (SDPG).

Las SDPGs son útiles en compresión de texto debido a permiten diseñar esquemas de compresión con gran flexibilidad de diseño. Las principales ventajas son:

- Permite un aislamiento absoluto entre el modelo probabilístico y el codec entrópico, hasta el punto que pueden ser programas diferentes que se pasan una SDPG a nivel de archivos. Como consecuencia, cualquier codec entrópico puede ser usado sin necesidad de modificar el modelo.
- El tamaño de la representación de los símbolos no es modificado tras obtener una SDPG. La longitud del fichero comprimido es igual a la del descomprimido. Esto ayuda notablemente a la manipulación de las SDPGs.

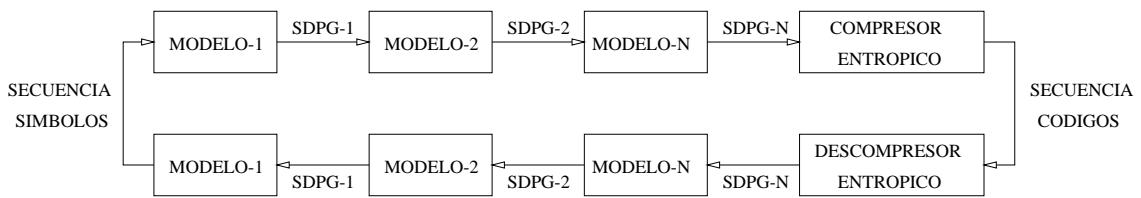


Figura 2.20: Modelo de compresión multietapa usando modelos independientes y un *codec* entrópico.

- Una SDPG puede ser usada por otro modelo que explote una redundancia diferente de la correlación estadística (por ejemplo, la espacial), si ambas fuentes de redundancia están presentes simultáneamente. Así, los compresores de datos pueden diseñarse como una serie de modelos que sustraen de las secuencias de entrada las respectivas fuentes de información que son capaces de predecir y que pueden ser aplicados secuencialmente. Para hacer efectiva esta reducción del nivel de información y como paso final, aplicamos un compresor entrópico (ver Figura 2.20). La descompresión consiste justamente en el proceso inverso.

El esquema es similar al que se usa con fuentes de información complejas como puede ser una secuencia de vídeo. Un primer modelo se usa para eliminar la redundancia temporal, la cual indica que los diferentes cuadros de la secuencia son parecidos entre si. A la técnica que explota dicha fuente se le conoce con el nombre de estimación del movimiento. Un segundo modelo se utiliza para eliminar la correlación dentro de cada cuadro que es debida a la correlación espacial. Ésta nos habla de que localmente, las imágenes (especialmente las de tono continuo) son fundamentalmente señales bidimensionales suaves y consecuentemente, puntos vecinos tienen niveles de intensidad parecidos.

Una SDPG puede ser comprimida usando cualquier código de longitud variable ya que existen unos símbolos más frecuentes que otros. Sin embargo, existe un tipo especial de códigos de longitud variable cuya composición es independiente del número de símbolos del alfabeto y que están especialmente indicados para la compresión de SDPGs porque son mucho más fáciles de determinar que la codificación de Huffman o aritmética. Estos códigos son: el código binario ajustado, el código de Golomb y el código de Rice.

Describiremos una serie de códigos de longitud variable, que son apropiados para la codificación de SDPGs, empezando por el más sencillo de todos ellos: la codificación binaria ajustada.

2.3.9 La codificación binaria ajustada

Un código binario ajustado [42] puede ser utilizado cuando los símbolos de índice menor son ligeramente más probables que los otros. La pendiente de la SDPG se espera que sea muy poco pronunciada, en realidad, casi plana. El tamaño del alfabeto debe ser finito y conocido.

El codificador

Sea $n = 2^k + b$ el tamaño del alfabeto. La codificación del símbolo $s = 0, 1, \dots, n - 1$ se realiza de la siguiente forma:

1. Si $s < n - 2b$:
 - (a) Emitir s usando un código binario de k bits.
2. Si no:
 - (a) Emitir $s + n - 2b$ usando un código binario de $k + 1$ bits.

La codificación de un símbolo es muy simple como puede comprobarse. Supongamos que $k = 3$ y $b = 2$, con lo que $n = 10$. Si por ejemplo $s = 0$, emitiríamos el código 000, ya que $s < 10 - 4 = 6$. Si $s = 1$, emitiríamos 001. Si $s = 5$, emitiríamos 101. Si $s = 6$, el código asignado sería $6 + 10 - 4 = 12$ representado en binario usando 4 bits: 1100. Si $s = 7$ emitiríamos 1101. Con $s = 8$, el código es 1110 y si $s = 9$, emitimos 1111.

El descodificador

Conociendo n , la descodificación del símbolo s se calcula así:

1. Leer sobre s los siguientes k bits de entrada.
2. Si $s < n - 2b$:
 - (a) Emitir s .
3. Si no:
 - (a) $s \leftarrow 2s + \text{siguiente bit de entrada.}$
 - (b) $s \leftarrow s - n + 2b.$

La descodificación es casi trivial. Si los k bits leídos no forman un número que supera la cantidad $n - 2b$ ($= 6$ en el ejemplo anterior), ya tenemos el símbolo descodificado. Si la superan, sólo hay que leer un bit más y restar la cantidad sumada en el codificador. Por ejemplo, si el código es 1110, los 3 primeros bits forman el valor 7. Como $7 > 6$, leemos el siguiente bit y tenemos el número 14. Ahora restamos 6 y obtenemos el símbolo 8.

2.3.10 La codificación de Golomb

Solomon W. Golomb desarrolló en 1966 una forma rápida y compacta de codificación de SDPGs [32]. Encontró un algoritmo de codificación y de descodificación que puede ser usado fácilmente con modelos adaptativos y basados en el contexto. Esta propiedad junto a que la ocurrencia de SDPGs es bastante corriente en los compresores de datos están provocando que la codificación de Golomb sea muy utilizada en el diseño de codec modernos y rápidos.

El codificador

El codificador de Golomb necesita conocer a priori cómo de probable es el símbolo más frecuente ya que esto define la pendiente de la distribución de probabilidad que sigue la SDPG. Si p es su probabilidad, Golomb definió un parámetro llamado

$$m = -\log(2) / \log(p)$$

que es directamente proporcional a la longitud de los códigos asignados de forma que cuanto más probable es el símbolo más frecuente, más pequeña es m (mayor es la pendiente). El algoritmo de generación de la palabra asociada al símbolo s es el siguiente:

1. Sea $k \leftarrow \lceil \log_2(m) \rceil$.
2. Sea $r \leftarrow s \bmod m$ (el resto de la división entera).
3. Sea $t \leftarrow 2^k - m$.
4. Emitir $(s \bmod m)$ usando un código unario de $(s \bmod m) + 1$ bits.
5. Si $r < t$:
 - (a) Emitir r usando un código binario de $k - 1$ bits.
6. Si no:
 - (a) $r \leftarrow r + t$.
 - (b) Emitir r usando un código binario de k bits.

Veamos un ejemplo de codificación. Supongamos que $m = 3$ cuando vamos a codificar el símbolo 0 lo que provoca que $k \leftarrow 2$, $r \leftarrow 0$ y $t \leftarrow 1$. Primero emitimos 0 ($0 \bmod 3$) usando un código unario de 1 bit. Por lo tanto la primera parte del código de Golomb es 0. Como $r < t$, emitimos 0 usando un código binario de 1 bit. Por lo tanto, la segunda parte del código es otro 0. El código de Golomb asignado al símbolo 0 para $m = 3$ es el 00.

Supongamos ahora que codificamos el símbolo 1. $k \leftarrow 2$, $r \leftarrow 1$ y $t \leftarrow 1$. Emitimos 0 ($1 \bmod 3$) usando un código unario: 0. Como $r = t$, hacemos $r \leftarrow 1 + 1 = 2$ y codificamos 2 usando un código binario de 2 bits. El código de Golomb asignado al símbolo 1 para $m = 3$ es el 010. Etc.

El descodificador

El descodificador de Golomb recupera el símbolo a partir de un código y del valor del parámetro m . Este es su algoritmo:

1. Sea $k \leftarrow \lceil \log_2(m) \rceil$ y $t \leftarrow 2^k - m$.

Tabla 2.2: Códigos de Golomb y de Rice para los primeros símbolos.

Golomb Rice	$m = 1$ $k = 0$	$m = 2$ $k = 1$	$m = 3$ $k = 2$	$m = 4$ $k = 2$	$m = 5$	$m = 6$	$m = 7$	$m = 8$ $k = 3$
$s = 0$	0	00	00	000	000	000	000	0000
1	10	01	010	001	001	001	0010	0001
2	110	100	011	010	010	0100	0011	0010
3	1110	101	100	011	0110	0101	0100	0011
4	11110	1100	1010	1000	0111	0110	0101	0100
5	$1^5 0$	1101	1011	1001	1000	0111	0110	0101
6	$1^6 0$	11100	1100	1010	1001	1000	0111	0110
7	$1^7 0$	11101	11010	1011	1010	1001	1000	0111
8	$1^8 0$	111100	11011	11000	10110	10100	10010	10000
:	:	:	:	:	:	:	:	:

2. Sea $s \leftarrow$ el número de unos consecutivos de entrada.
3. Sea $x \leftarrow$ los siguientes $k - 1$ bits de entrada.
4. Si $x < t$:
 - (a) $s \leftarrow s \times m + x$.
5. Si no:
 - (a) $x \leftarrow x \times 2 +$ siguiente bit de entrada.
 - (b) $s \leftarrow s \times m + x - t$.

Con $m = 3$ descodificaremos el código 00. Según el paso 1, $k \leftarrow 2$ y $t \leftarrow 1$. $s \leftarrow 0$ pues no existe ningún 1 consecutivo a la entrada. $x \leftarrow 0$. Como $x < t$ hacemos $s \leftarrow 0 \times 3 + 0 = 0$ y 0 es el símbolo descodificado.

Con $m = 3$ descodificaremos el código 010. $k \leftarrow 2$ y $t \leftarrow 1$. $s \leftarrow 0$ pues no existe ningún 1 consecutivo a la entrada. $x \leftarrow 1$. Como $x = t$, hacemos $x \leftarrow x \times 2 +$ siguiente bit de entrada que es 0, quedando que $x = 10$. $s \leftarrow 0 \times 3 + 2 - 1 = 1$ y 1 es el símbolo descodificado.

En la Tabla 2.2 se muestran los primeros códigos de Golomb para diferentes valores de m . Veamos un ejemplo más, para $m = 7$ y un código igual a 10010, vamos a encontrar cuál es el símbolo codificado. $k \leftarrow 3$ y $t \leftarrow 1$. $s \leftarrow 1$. $x \leftarrow 01$. Como $x = t$, $x \leftarrow x \times 2 + 0 = 2$. $s \leftarrow 1 \times 7 + 2 - 1 = 8$.

Estimación del parámetro m

El valor de m decide el número de bits de código asignados a los símbolos, por lo que su estimación adecuada es muy importante. Podemos distinguir dos situaciones básicas:

- Cuando la pendiente de la SDPG sea constante, m puede estimarse calculando para los valores más probables de m , qué valor de m genera una secuencia de palabras más corta. Ya que la pendiente es invariable, basta calcular la anterior longitud para un número suficientemente grande de símbolos (en la práctica unas decenas bastan).
- En general, la pendiente de la SDPG no es constante y además, inicialmente es menor ya que los modelos de Markov usados tardan un cierto tiempo en aprender. En este caso conviene comenzar con un valor grande de m e ir comprobando dinámicamente si para $m - 1$ se consigue una secuencia de palabras más corta atendiendo a los últimos símbolos codificados.

Por otra parte, para resolver el problema de que ocurra una reducción en la correlación estadística entre los símbolos y en consecuencia, la pendiente pueda ser menos acusada (el modelo está prediciendo peor), convendría además probar con cada símbolo codificado si para $m + 1$ se consigue un acortamiento en la secuencia comprimida. Si estimar m para cada símbolo codificado es demasiado costoso puede realizarse la estimación cada cierto número de símbolos.

2.3.11 La codificación de Rice

Al igual que la codificación de Golomb, el código de Rice [87] utiliza el índice asociado a un símbolo para encontrar su código. En realidad se trata de un caso particular del código de Golomb para $m = 2^k$, donde $k = 0, 1, \dots$ expresa la pendiente de la distribución de probabilidad geométrica. Justamente para estos valores de m , el código de Golomb es especialmente fácil de calcular y esto es lo que se conoce como código de Rice.

El codificador

El codificador de Rice puede deducirse del codificador de Golomb teniendo en cuenta las restricciones impuestas al parámetro m . El algoritmo de codificación es extremadamente simple (especialmente usando hardware):

1. Sea $m \leftarrow 2^k$.
2. Emitir $\lfloor s/m \rfloor$ ⁷ usando un código unario (de $\lfloor s/m \rfloor + 1$ bits).
3. Emitir $s - \lfloor s/m \rfloor$ (el módulo) usando un código binario de k bits.

⁷ $\lfloor \cdot \rfloor$ denota “al menor entero más grande que”.

Supongamos que $k = 0$ y $s = 7$. Entonces $m \leftarrow 1$. Emitimos $\lfloor s/m \rfloor = 7$ como un número unario de 8 bits generando a la salida 11111110 y seguidamente emitimos $s - \lfloor s/m \rfloor \times m = 7 - 7 \times 1 = 0$ usando un código binario de 0 bits. Por lo tanto, el código asociado al símbolo 7 es el 11111110. La división entera puede realizarse desplazando s , m bits a la derecha. Notar que el cálculo del módulo consiste simplemente en extraer los m bits menos significativos de s .

Un segundo ejemplo. Sea $k = 3$ y $s = 11$. $m \leftarrow 8$. Emitimos 1 como un número unario: 10 y emitimos 3 como un número binario de 3 bits: 011. El código de Rice asignado es: 10011.

La Tabla 2.2 presenta los códigos de Rice para los primeros símbolos, en función del parámetro k . Como puede deducirse fácilmente, el código de Rice puede expresar con menos precisión las distribuciones de probabilidad de los modelos probabilísticos que el código de Golomb.

El descodificador

Dado un código y un valor de k , el descodificador encuentra el símbolo s asociado con una sencillez asombrosa:

1. Sea s el número de unos consecutivos a la entrada.
2. Sea x los siguientes $k + 1$ bits de entrada.
3. $s \leftarrow s \times 2^k + x$.

Descodifiquemos el código 11111110 para $k = 0$. $s \leftarrow 7$. Sobre x leemos los siguientes $0 + 1$ bits de entrada con lo que $x = 0$ (notar que el primer bit leído sobre x siempre es 0). Efectuamos la operación $s \leftarrow s \times 2^k + 0 = 7 \times 2^0 = 7$ (el producto $s \times 2^k$ equivale a desplazar s k bits a la izquierda, introduciendo ceros por la derecha y la suma con otros k bits se convierte en una asignación de bits).

Descodifiquemos el segundo ejemplo: 10011 con $k = 3$. $s \leftarrow 1$. $x \leftarrow 3$. $s \leftarrow 1 \times 8 + 3 = 11$.

Estimación del parámetro k

El parámetro k se rige básicamente por las mismas normas que el parámetro m en el código de Golomb. Hay que estimarlo de acuerdo con el valor de la pendiente de la SDPG. Debido a que el número de posibles valores de k es pequeño, su estimación es menos costosa que la del parámetro m en la codificación de Golomb. Por esta razón, los métodos más eficientes son los mismos que para la codificación de Golomb, pero además es posible calcular la longitud de la secuencia comprimida, incluso para cualquier valor de k . De esta forma, cada vez que se codifica un símbolo se elige el valor que provoca una longitud menor de forma dinámica (ya que valores de k mayores que 7 son muy raros en el caso de codificar alfabetos con 256 símbolos) [42].

2.3.12 Golomb y Rice frente a Huffman

Los códigos de Golomb y Rice son muy útiles si el tamaño del alfabeto es muy grande o infinito. Sólo en estos casos el código de Golomb tiene la misma longitud que el de Huffman y se puede afirmar que Golomb es un caso particular de Huffman, pero mucho más fácil de calcular. Además, si las probabilidades de la SDPG son muy variables, las velocidades de cálculo del código Golomb y especialmente del código Rice, pueden ser un factor determinante. Golomb y Rice son buenos códigos de longitud variables para modelos adaptativos.

Sin embargo, cuando el tamaño del alfabeto es pequeño (256 o menos símbolos) y las probabilidades varían lentamente, entonces Golomb es inferior a la versión adaptativa del código de Huffman. Supongamos por ejemplo que el alfabeto tiene sólo 2 símbolos. Huffman asignaría un bit a cada uno. Sin embargo, con $m = 1$ Golomb asignaría un bit a uno y dos bits a otro (ver Tabla 2.2) y cuando $m = 8$, cada símbolos se representaría usando 4 bits.

2.3.13 Codificación aritmética de una SDPG

Uno de los principales inconvenientes de la codificación aritmética multisímbolo es la lentitud del descodificador. Cuando hay que encontrar el intervalo que contiene a la sección actual de código aritmético que se está descodificando, no queda más remedio que probar con cada uno de los intervalos posibles hasta que encontramos el buscado. Una solución sencilla (aunque lenta) es comenzar por el más a la izquierda (el más próximo a 0000) y probar secuencialmente hasta que el intervalo sea encontrado. La mejor opción suele consistir en comenzar siempre la búsqueda por el intervalo más grande ya que es el que con más probabilidad puede contener el código aritmético. Esta solución minimiza el número de iteraciones de búsqueda, pero complica el descodificador pues es necesario mantener una estructura de datos que mantenga ordenados los intervalos por tamaño [128]. Sin embargo, existe una solución que se usa en implementaciones *hardware* y que consiste en hacer muchas comparaciones secuenciales en paralelo, dividiendo así el tiempo de búsqueda entre el número de comparadores [74].

Cuando estamos trabajando con SDPGs, esta estructura de datos no es necesaria ya que una búsqueda secuencial comenzando por el intervalo más a la izquierda equivale a probar primero con el intervalo más grande. El descodificador no invierte por tanto memoria ni CPU en mantener esta lista ordenada y el proceso de descodificación se acelera.

2.3.14 PPM: predicción usando búsqueda parcial de cadenas

PPM (*Prediction by Partial Matching*) es el resultado de unir un modelo probabilístico adaptativo, inicialmente vacío y basado en el contexto, a un codificador aritmético multisímbolo. Explota al 100% cualquier indicio de correlación probabilística que exista en la secuencia de símbolos comprimidos.

Los autores definen a PPM [17] como una técnica de modelado estadístico de contexto finito. El modelo calcula las probabilidades de los símbolos contando el número de veces que ha ocurrido cada símbolo para cada contexto aparecido en la secuencia, hasta un orden máximo finito k prefijado de antemano. Por lo tanto, el modelo utiliza los últimos k símbolos para calcular la probabilidad del siguiente.

Esta idea ya había sido (y será posteriormente) usada en muchos otros algoritmos de compresión basados en modelos de Markov. Entre ellos es posible mencionar a compresores como DAFC [59], FBA [92], LOEMA [92], UMC [91], DHPC [123], DMC [20] y WORD [70], pero fue PPM la primera implementación software útil en cuanto a tiempo de ejecución y tasa de compresión alcanzada.

El problema de todos los modelos con memoria (aparte de la cantidad de memoria que consumen) radica en que necesita cantidades enormes de datos para que sean capaces de predecir adecuadamente las probabilidades de los símbolos cuando k es grande (más de 3 en general, aunque esto depende también de la cantidad de correlación en la fuente). PPM resolvió el problema de minimizar el impacto negativo que se produce en la tasa de compresión cuando k crece, bajo el nombre de “predicción usando búsqueda parcial de cadenas”.

El compresor

A grandes rasgos, el compresor PPM funciona de la siguiente manera: Sea k el máximo orden de predicción (la longitud del contexto más largo) y sea s el símbolo a codificar. Sean $c[i]$ los i últimos símbolos codificados (el contexto actual) y sea $p(s|c[i])$ la probabilidad condicionada que es generada por el modelo para el símbolo s en el contexto $c[i]$. Por tratarse de un modelo inicialmente vacío, se utiliza el símbolo especial ESC para indicar al descompresor la ocurrencia de símbolos nuevos. Debido a esto, inicialmente $p(\text{ESC}|c[i]) = 1$ para todo contexto con $0 \leq i \leq k$ y como es evidente, $p(s|c[i]) = 0$ para cualquier otro símbolo s . Además, existe un contexto especial para $i = -1$ donde todos los símbolos del alfabeto son contemplados, junto con el símbolo especial EOF. En este contexto, $p(s|c[-1]) = \frac{1}{r}$ donde r es igual al tamaño del alfabeto más 1 (debido al símbolo EOF). En este marco de trabajo, el algoritmo de codificación de un símbolo s según PPM consiste en el siguiente proceso que ha de repetirse para cada símbolo codificado:

1. Sea $i \leftarrow k$ el orden de predicción actual.
2. Mientras $p(s|c[i]) = 0$:
 - (a) Codificar un ESC según $p(\text{ESC}|c[i])$. Nótese que este paso sólo tiene efecto si $p(\text{ESC}|c[i]) < 1$. Además, dependiendo del método usado para resolver el problema de la frecuencia cero, su probabilidad se incrementa o no.
 - (b) Actualizar $p(s|c[i])$ (incrementando la probabilidad).
 - (c) $i \leftarrow i - 1$.

3. Codificar s según $p(s|c[i])$. Todos los símbolos que estaban en contextos de orden mayor que i son excluidos del contexto actual ya que es seguro que s no es ninguno de ellos.
4. Actualizar $p(s|c[i])$.

Gracias a cómo se ha definido $p(\cdot|c[-1])$, siempre se alcanza el paso 3. El modelo aprende gracias al paso 2.b y cuando ya han sido codificados suficientes símbolos, el paso 3 con $i = -1$ se alcanza esporádicamente, ya que $p(s|c[i]) \neq 0$ para valores de i cada vez mayores. Nótese también que la probabilidad asignada al símbolo ESC es un factor determinante para la tasa de compresión obtenida, debido a que debe ser codificado cada vez que un símbolo no se encuentra en el contexto actual. Desde un punto de vista probabilístico, la emisión de un símbolo ESC con probabilidad $p(\text{ESC}|c[i])$ provoca que si el siguiente símbolo emitido posee una probabilidad $p(s|c[i-1])$, la probabilidad efectiva de dicho símbolo según el modelo PPM es $p(\text{ESC}|c[i]) \times p(s|c[i-1])$. De esta forma, el uso del símbolo ESC para codificar los símbolos novedosos permite la asignación de probabilidades extremadamente pequeñas.⁸

La Figura 2.21 muestra un ejemplo de codificación según el algoritmo PPMC (PPM + método C para la estimación de la probabilidad del símbolo ESC).

El descompresor

La descompresión de un símbolo es justamente el proceso inverso. Inicialmente todos los contextos se inician de la misma forma que lo hace el compresor. A continuación aplica el siguiente algoritmo:

1. Sea $i \leftarrow k$ el orden de predicción actual.
2. Mientras el símbolo descodificado sea ESC:
 - (a) Actualizar $p(\text{ESC}|c[i])$ si $p(\text{ESC}|c[i]) < 1$, según el método usado para resolver el problema de la frecuencia cero.
 - (b) $i \leftarrow i - 1$.
3. Descodificar el símbolo $s \neq \text{ESC}$ según $p(s|c[i])$. El sistema de exclusión de símbolos realizado en el paso 3 del compresor se realiza también en este paso.
4. Ahora, actualizamos el modelo. Mientras $i \leq k$:
 - (a) Actualizar la probabilidad de $p(s|c[i])$.
 - (b) $i \leftarrow i + 1$.

La Figura 2.22 presenta un ejemplo de descodificación. Ahora, en lugar de indicar las probabilidades de los símbolos descodificados (como ocurre en la Figura 2.21), se presentan las longitudes de los códigos.

⁸De aquí que PPM sólo sea útil si se usa junto con la codificación aritmética.

entrada	salida	probabilidad	contextos afectados
a	$c_a[-1]$	$1/r$	$c[] = \{\text{ESC}, 1 \text{ a}, 1\}$
b	$c_{\text{ESC}}[] c_b[-1]$	$1/2 \cdot 1/(r-1)$	$c[a] = \{\text{ESC}, 1 \text{ b}, 1\},$ $c[] = \{\text{ESC}, 2 \text{ a}, 1 \text{ b}, 1\}$
a	$c_a[]$	$1/4$	$c[b] = \{\text{ESC}, 1 \text{ a}, 1\},$ $c[] = \{\text{ESC}, 2 \text{ a}, 2 \text{ b}, 1\}$
b	$c_b[a]$	$1/2$	$c[a] = \{\text{ESC}, 1 \text{ b}, 2\}$
c	$c_{\text{ESC}}[b] c_{\text{ESC}}[] c_c[-1]$	$1/2 \cdot 2/3 \cdot 1/(r-2)$	$c[b] = \{\text{ESC}, 2 \text{ a}, 1 \text{ c}, 1\},$ $c[] = \{\text{ESC}, 3 \text{ a}, 2 \text{ b}, 1 \text{ c}, 1\}$
b	$c_b[]$	$1/7$	$c[c] = \{\text{ESC}, 1 \text{ b}, 1\},$ $c[] = \{\text{ESC}, 3 \text{ a}, 2 \text{ b}, 2 \text{ c}, 1\}$
a	$c_a[b]$	$1/4$	$c[b] = \{\text{ESC}, 2 \text{ a}, 2 \text{ c}, 1\}$
b	$c_b[a]$	$2/3$	$c[a] = \{\text{ESC}, 1 \text{ b}, 3\}$
a	$c_a[b]$	$2/5$	$c[b] = \{\text{ESC}, 2 \text{ a}, 3 \text{ c}, 1\}$
b	$c_b[a]$	$3/4$	$c[a] = \{\text{ESC}, 1 \text{ b}, 4\}$
a	$c_a[b]$	$3/6$	$c[b] = \{\text{ESC}, 2 \text{ a}, 4 \text{ c}, 1\}$
a	$c_{\text{ESC}}[a] c_a[]$	$1/5 \cdot 2/(8-2)$	$c[a] = \{\text{ESC}, 2 \text{ b}, 4 \text{ a}, 1\},$ $c[] = \{\text{ESC}, 3 \text{ a}, 3 \text{ b}, 2 \text{ c}, 1\}$
a	$c_a[a]$	$1/7$	$c[a] = \{\text{ESC}, 2 \text{ b}, 4 \text{ a}, 2\}$
a	$c_a[a]$	$2/8$	$c[a] = \{\text{ESC}, 2 \text{ b}, 4 \text{ a}, 3\}$
a	$c_a[a]$	$3/9$	$c[a] = \{\text{ESC}, 2 \text{ b}, 4 \text{ a}, 4\}$
a	$c_a[a]$	$4/10$	$c[a] = \{\text{ESC}, 2 \text{ b}, 4 \text{ a}, 5\}$
a	$c_a[a]$	$5/11$	$c[a] = \{\text{ESC}, 2 \text{ b}, 4 \text{ a}, 6\}$

Figura 2.21: Ejemplo de codificación PPMC.

2.3.15 BWT: La transformada de Burrows-Wheeler

La BWT (Burrows-Wheeler Transform) es la base de una nueva generación de compresores de datos muy eficaces, que consiguen tasas de compresión cercanas a las de los algoritmos basados en modelos de Markov adaptativos (tipo PPM), a una velocidad comparable a la de los algoritmos LZ [13].

En realidad, BWT es únicamente un algoritmo de ordenación de cadenas un tanto peculiar. En este sentido, BWT no comprime ni expande la secuencia a comprimir, solamente la transforma (en el más puro sentido de las transformaciones usadas para el tratamiento de señales). La salida está formada por los mismos símbolos que tenemos a la entrada pero en un orden diferente, y es precisamente el orden en que aparecen lo que posibilita comprimir eficientemente la secuencia transformada.

Los autores de la BWT la definen como un algoritmo de codificación de datos sin pérdida basado en la transformación de bloques de símbolos. La secuencia a comprimir debe ser procesada forzosamente de esta forma. Cuanto más grandes son dichos bloques, más compresión es posible conseguir (llegándose a procesar si es necesario toda la secuencia

entrada	bits	salida	contextos afectados
$c_a[-1]$	8	a	$c[] = \{\text{ESC}, 1 \text{ a}, 1\}$
$c_{\text{ESC}}[]$	1		$c[] = \{\text{ESC}, 2 \text{ a}, 1\}$
$c_b[-1]$	7.99	b	$c[a] = \{\text{ESC}, 1 \text{ b}, 1\}, c[] = \{\text{ESC}, 2 \text{ a}, 1 \text{ b}, 1\}$
$c_a[]$	2	a	$c[b] = \{\text{ESC}, 1 \text{ a}, 1\}, c[] = \{\text{ESC}, 2 \text{ a}, 2 \text{ b}, 1\}$
$c_b[a]$	1	b	$c[a] = \{\text{ESC}, 1 \text{ b}, 2\}$
$c_{\text{ESC}}[b]$	1		$c[b] = \{\text{ESC}, 2 \text{ a}, 1\}$
$c_{\text{ESC}}[]$	0.58		$c[] = \{\text{ESC}, 3 \text{ a}, 2 \text{ b}, 1\}$
$c_c[-1]$	7.98	c	$c[b] = \{\text{ESC}, 2 \text{ a}, 1 \text{ c}, 1\}, c[] = \{\text{ESC}, 3 \text{ a}, 2 \text{ b}, 1 \text{ c}, 1\}$
$c_b[]$	2.81	b	$c[c] = \{\text{ESC}, 1 \text{ b}, 1\}, c[] = \{\text{ESC}, 3 \text{ a}, 2 \text{ b}, 2 \text{ c}, 1\}$
$c_a[b]$	2	a	$c[b] = \{\text{ESC}, 2 \text{ a}, 2 \text{ c}, 1\}$
$c_b[a]$	0.58	b	$c[a] = \{\text{ESC}, 1 \text{ b}, 3\}$
$c_a[b]$	1.32	a	$c[b] = \{\text{ESC}, 2 \text{ a}, 3 \text{ c}, 1\}$
$c_b[a]$	0.42	b	$c[a] = \{\text{ESC}, 1 \text{ b}, 4\}$
$c_a[b]$	1	a	$c[b] = \{\text{ESC}, 2 \text{ a}, 4 \text{ c}, 1\}$
$c_{\text{ESC}}[a]$	2.32		$c[a] = \{\text{ESC}, 2 \text{ b}, 4\}$
$c_a[]$	1.58	a	$c[a] = \{\text{ESC}, 2 \text{ b}, 4 \text{ a}, 1\}, c[] = \{\text{ESC}, 3 \text{ a}, 3 \text{ b}, 2 \text{ c}, 1\}$
$c_a[a]$	2.81	a	$c[a] = \{\text{ESC}, 2 \text{ b}, 4 \text{ a}, 2\}$
$c_a[a]$	2	a	$c[a] = \{\text{ESC}, 2 \text{ b}, 4 \text{ a}, 3\}$
$c_a[a]$	1.58	a	$c[a] = \{\text{ESC}, 2 \text{ b}, 4 \text{ a}, 4\}$
$c_a[a]$	1.32	a	$c[a] = \{\text{ESC}, 2 \text{ b}, 4 \text{ a}, 5\}$
$c_a[a]$	1.14	a	$c[a] = \{\text{ESC}, 2 \text{ b}, 4 \text{ a}, 6\}$

Figura 2.22: Ejemplo de descodificación PPMC.

de una vez). Esto plantea el inconveniente de que la compresión no puede realizarse como una corriente (*stream*) y por lo tanto, su uso como algoritmo de compresión en tiempo real está limitado.

Los compresores de datos que utilizan la BWT constan de los siguientes pasos:

1. Transformar la secuencia de entrada usando la BWT directa.
2. Aplicar algún método de compresión de texto.

El proceso de descompresión es justamente el proceso inverso:

1. Descomprimir usando el descompresor de texto correspondiente.
2. Aplicar la BWT inversa.

Estudiaremos a continuación en qué consiste la BWT y cómo comprimir los datos transformados.

<a>babcbababaaaaaaa	0
babcbababaaaaaaa<a>	1
abcbababaaaaaaa<a>b	2
bcbababaaaaaaa<a>ba	3
cbababaaaaaaa<a>bab	4
bababaaaaaaa<a>babc	5
ababaaaaaaa<a>babcb	6
babaaaaaaa<a>babcba	7
abaaaaaaa<a>babcbab	8
baaaaaaaa<a>babcaba	9
aaaaaaaa<a>babcbab	10
aaaaaaaa<a>babcbababa	11
aaaaaa<a>babcbababaa	12
aaaa<a>babcbababaaa	13
aaa<a>babcbababaaaa	14
aa<a>babcbababaaaaaa	15
a<a>babcbababaaaaaaa	16

Figura 2.23: Ejemplo de matriz inicial para la BWT.

La BWT directa

Para un tamaño de bloque N prefijado de antemano, el algoritmo de la transformada BWT realiza los siguientes pasos:

1. Leer la secuencia N de símbolos.
2. Construir una matriz cuadrada de lado igual al tamaño del bloque N , donde la primera fila es la secuencia original, la segunda es la secuencia desplazada un símbolo a la izquierda de forma cíclica, etc.
3. Ordenar lexicográficamente la matriz por filas. Este es el paso pesado de algoritmo y se ejecuta en un tiempo proporcional a $N \times \log_2(N)$.
4. Buscar en la columna $N - 1$ (la de más a la derecha) la fila en la que se encuentra el primer símbolo de la secuencia original. Sea este valor i .
5. La transformada BWT de la secuencia de entrada está formada por el contenido de la columna $N - 1$ (la última) y el índice i .

Como ejemplo vamos a transformar la secuencia `ababcbababaaaaaaa`. La Figura 2.23 presenta la matriz que contiene las secuencias desplazadas. Se ha usado el separador `<>` para delimitar con claridad el primer símbolo de la secuencia. Tras la ordenación se consigue la matriz de la Figura 2.24. En este ejemplo, $i = 14$ y la salida de la BWT es:

aaaaaaaa<a>babcbabab	10	0
aaaaaaa<a>babcbababa	11	1
aaaaaa<a>babcbababaa	12	2
aaaa<a>babcbababaaa	13	3
aaa<a>babcbababaaa	14	4
aa<a>babcbababaaaa	15	5
a<a>babcbababaaaaaa	16	6
abaaaaaaaa<a>babcbab	8	7
ababaaaaaaaa<a>babc	6	8
<a>babcbababaaaaaa	0	9
abcbababaaaaaaa<a>b	2	10
baaaaaaaa<a>babcba	9	11
babaaaaaaaa<a>babc	7	12
babcbababaaaaaa<a>	5	13
bcbababaaaaaaa<a>ba	1	14 $\leftarrow i$
cbababaaaaaaa<a>bab	3	15
	4	16

Figura 2.24: Ejemplo de matriz ordenada para la BWT y selección del índice i .

baaaaaaabbaacaab14. Como es posible apreciar, la secuencia de símbolos de salida tiende a tener largas series de símbolos repetidos. Es justamente esta propiedad lo que provoca que dicha salida sea mucho más comprimible que la secuencia no transformada.

Las series de símbolos repetidos a la salida es consecuencia de que cuando existen patrones que se repiten en la secuencia de entrada (por ejemplo la cadena “que” es muy común en español), al efectuarse las diferentes rotaciones del bloque de cadenas y ordenarse lexicográficamente, es frecuente que encontremos muchos bloques colocados sobre filas consecutivas que comienzan por la misma cadena (“ue” en nuestro ejemplo) y que colocaran el mismo carácter en la última columna, la que forma la secuencia transformada.

Como segundo ejemplo hemos querido transformar la secuencia **ababab** para apreciar mejor qué hace la BWT. El resultado ha sido **bbbaaa3**. Ahora sí se aprecia con claridad que la salida tiende a agrupar caracteres iguales.

La BWT inversa

El *kernel* de la BWT es un producto de matrices de permutación ortonormales que responden a una secuencia de rotaciones y reglas lexicográficas fijas. Por lo tanto, el *kernel* de la transformada inversa existe y puede encontrarse invirtiendo el *kernel* usado en la transformada directa. De esta forma, la secuencia de símbolos original puede ser generada a partir de la secuencia transformada y el índice i .

El algoritmo de la transformación inversa consta básicamente de los siguientes pasos:

1. Sea $L[]$ la secuencia transformada. $L[]$ se ordena lexicográficamente en un tiempo pro-

porcional a $N \times \log_2(N)$. Sea la secuencia ordenada $O[]$, usando el mismo algoritmo de ordenación usado en la transformada directa.

2. Calcular el vector de transformación $T[]$ de forma que si $O[j] = L[l]$ (donde l es el primer carácter de $L[]$ que recorrido secuencialmente comenzando desde el índice 0 cumple dicha condición), entonces $T[j] = l$. El carácter $L[l]$ sólo puede usarse una vez de forma que todos los elementos de $T[]$ son diferentes.
3. Sea $k \leftarrow i$ (i es el índice pasado como entrada).
4. Ejecutar N veces:
 - (a) Emitir el símbolo almacenado en $L[k]$.
 - (b) Hacer $k \leftarrow T[k]$.

Siguiendo con nuestro ejemplo, la secuencia transformada ordenada ($O[]$) es igual a aaaaaaaaaaaaabbbbb. $O[0] = \text{a}$ es el mismo símbolo que $L[1]$ y por lo tanto $T[0] = 1$. $O[1] = \text{a}$ es el símbolo $L[2]$ lo que indica que $T[1] = 2$, etc. El vector de transformación completo es 1, 2, 3, 4, 5, 6, 9, 11, 12, 14, 15, 0, 7, 8, 10, 16 y 13.

Una vez calculado $T[]$ ejecutamos el lazo del paso 4. Emitimos $L[14] = \text{a}$ y $k \leftarrow 10$. Emitimos $L[10] = \text{b}$ y $k \leftarrow 15$. Emitimos $L[15] = \text{a}$ y $k \leftarrow 16$, etc. Finalmente la secuencia original es reconstruida.

2.3.16 La codificación mover-al-frente: MTF

Los autores de la BWT, además de presentar su transformación, dan una idea de cómo es posible comprimir eficientemente la secuencia transformada. Su propuesta consiste en aplicar lo que se conoce con el nombre de codificación mover al frente (*move to front*) o MTF. Esta recodificación de los símbolos no expande ni comprime la secuencia, sino que de nuevo la transforma, para que sea fácilmente comprimida por un compresor entrópico.

El codificador

El algoritmo de codificación MTF es el siguiente:

1. Crear una lista $L[]$ con todos los símbolos del alfabeto.
2. Mientras existan símbolos que codificar:
 - (a) Sea s el siguiente símbolo de entrada.
 - (b) Buscar s en $L[]$ y emitir como código su posición en dicha lista.
 - (c) Mover s al frente de $L[]$ haciendo que $L[0] \leftarrow s$. El resto de símbolos son desplazados una posición hacia posiciones de índice superior.

lista	entrada	salida
...abc...	b	b
b...ac...	a	b
ab...c...	a	0
ab...c...	b	1
ba...c...	b	0
ba...c...	a	1
ab...c...	b	1
ba...c...	a	1
ab...c...	a	0
ab...c...	c	c
cab.....	a	1
acb.....	a	0
acb.....	b	1

Figura 2.25: Ejemplo de codificación MTF de la cadena transformada por la BWT (ver Figura 2.24).

La Figura 2.25 presenta un ejemplo de codificación MTF que codifica la secuencia transformada obtenida mediante la BWT en la sección anterior. Como puede apreciarse claramente, el número de ceros a la salida es alto, aunque más que el número de unos, etc. De hecho, es de esperar que el histograma de la salida del codificador MTF tras aplicar la BWT siga una SDPG.

El descodificador

El algoritmo de descodificación es muy similar al codificador y consiste en:

1. Crear una lista $L[]$ con todos los símbolos del alfabeto (igual al paso 1 del codificador).
2. Mientras existan símbolos que descodificar:
 - (a) Sea c el siguiente código de entrada.
 - (b) Emitir como símbolo $s \leftarrow L[c]$.
 - (c) Idem al paso 2.c del codificador.

La descodificación del ejemplo anterior se muestra en la Figura 2.26 donde se recupera la secuencia original.

lista	entrada	salida
...abc...	b	b
b...ac...	b	a
ab...c...	0	a
ab...c...	1	b
ba...c...	0	b
ba...c...	1	a
ab...c...	1	b
ba...c...	1	a
ab...c...	0	a
ab...c...	c	c
cab.....	1	a
acb.....	0	a
acb.....	1	b

Figura 2.26: Ejemplo de descodificación MTF.

Compresión de la SDPG

Una de las principales ventajas de la BWT es que como resultado final, tras usar la codificación MTF, la secuencia de símbolos sigue una SDPG por lo que cualquiera de los codificadores entrópicos vistos será eficiente. El modelo usado con el codificador es de orden 0, puesto que toda la correlación estadística ha sido eliminada tras aplicar la BWT y la codificación MTF.

2.3.17 PBT: La transformada de texto basada en predicción

La transformada de texto basada en predicción o PBT: (*Predictor Based Transform*) es una herramienta descorrelacionadora que puede ser utilizada para eliminar la redundancia estadística de una secuencia de símbolos. En el esquema mostrado en la Figura 2.20, PBT se colocaría en la caja “MODELO-1”, cuya salida es una SDPG y básicamente sustituye a una BWT+MTF.

Sin embargo, a diferencia de ésta, la PBT puede tratar la secuencia de símbolos como una corriente lo que es muy útil en transmisión de datos, donde el cuello de botella está normalmente en el canal de transmisión y no en los algoritmos de codificación.

Antes de presentar la PBT, vamos a introducir un método de codificación muy semejante a la codificación MTF que llamaremos MTF mejorado, ya que la PBT puede ser considerada como una extensión de éste cuando se usa un modelo basado en el contexto.

Una codificación MTF mejorada: la PBT de orden 0

La idea de la MTF mejorada se basa en que la colocación del símbolo codificado en la cabeza de la lista de símbolos es una apuesta demasiado fuerte para un símbolo que ha podido ocurrir sólo una vez. Piénsese en que los símbolos de la cabeza pueden estar siendo utilizados muy frecuentemente y no tiene sentido relevarlos a posiciones inferiores de la lista a causa de un símbolo que puede ser utilizado esporádicamente.

Una forma de solucionar este problema consiste en mantener junto a cada símbolo de la lista, un número que nos ayude a calcular su probabilidad de ocurrencia. Usando esta estructura de datos, un algoritmo de codificación MTF mejorado consistiría en los siguientes pasos:

1. Idéntico al paso 1 del codificador MTF. Además, cada nodo de la lista contemplará un recuento que indica el número de veces que se ha usado el símbolo.
2. Mientras existan símbolos que codificar:
 - (a) Sea s el siguiente símbolo de entrada.
 - (b) Buscar s en $L[]$ y emitir como código su posición en dicha lista.
 - (c) Incrementar el recuento de s .
 - (d) Ordenar la lista en función del recuento de s . Puesto que la lista está siempre ordenada, este paso puede hacerse en general en un tiempo inferior a $\log_2(r)$ donde r es el número de símbolos en la lista. s se coloca a la cabeza de todos los símbolos que tienen un recuento igual que él.

Un ejemplo de codificación MTF mejorada es mostrado en la Figura 2.27.

El algoritmo de descodificación es descrito por:

1. Idéntico al paso 1 del codificador MTF mejorado.
2. Mientras existan símbolos que descodificar:
 - (a) Sea c el siguiente código de entrada:
 - (b) Emitir como símbolo $s \leftarrow L[c]$.
 - (c) Ídem al paso 2.d del compresor MTF mejorado.

La descodificación de la secuencia codificada se muestra en la Figura 2.28.

La codificación MTF mejorada es un caso particular de la PBT en el que el orden de predicción es 0.

	lista			entrada	salida
...	a, 0	b, 0	c, 0	...	b b
b, 1	...	a, 0	c, 0	...	a b
a, 1	b, 1	...	c, 0	...	a 0
a, 2	b, 1	...	c, 0	...	a 0
a, 3	b, 1	...	c, 0	...	a 0
a, 4	b, 1	...	c, 0	...	a 0
a, 5	b, 1	...	c, 0	...	a 0
a, 6	b, 1	...	c, 0	...	b 1
a, 6	b, 2	...	c, 0	...	b 1
a, 6	b, 3	...	c, 0	...	a 0
a, 7	b, 3	...	c, 0	...	b 1
a, 7	b, 4	...	c, 0	...	a 0
a, 8	b, 4	...	c, 0	...	a 0
a, 9	b, 4	...	c, 0	...	c c
a, 9	b, 4	c, 1	a 0
a, 10	b, 4	c, 1	a 0
a, 11	b, 4	c, 1	b 1

Figura 2.27: Ejemplo de codificación MTF mejorada.

	lista			entrada	salida
...	a, 0	b, 0	c, 0	...	b b
b, 1	...	a, 0	c, 0	...	b a
a, 1	b, 1	...	c, 0	...	0 a
a, 2	b, 1	...	c, 0	...	0 a
a, 3	b, 1	...	c, 0	...	0 a
a, 4	b, 1	...	c, 0	...	0 a
a, 5	b, 1	...	c, 0	...	0 a
a, 6	b, 1	...	c, 0	...	1 b
a, 6	b, 2	...	c, 0	...	1 b
a, 6	b, 3	...	c, 0	...	0 a
a, 7	b, 3	...	c, 0	...	1 b
a, 7	b, 4	...	c, 0	...	0 a
a, 8	b, 4	...	c, 0	...	0 a
a, 9	b, 4	...	c, 0	...	c c
a, 9	b, 4	c, 1	0 a
a, 10	b, 4	c, 1	0 a
a, 11	b, 4	c, 1	1 b

Figura 2.28: Ejemplo de descodificación MTF mejorada.

PBT: La transformada basada en predicción

La PBT se basa en las ideas Shannon cuando realizó su famoso experimento de la medición de la entropía del idioma inglés [108], que consistió básicamente en usar una persona que comprendía el lenguaje para predecir cual iba a ser el siguiente símbolo, dentro de un contexto de 100 de ellos. Calculó la entropía del error de predicción contando el número de veces que se había equivocado en cada predicción. La secuencia resultado o secuencia error de predicción era por lo tanto en una SDPG. La PBT funciona de forma similar, pero en lugar de usar una persona, usa un modelo probabilístico basado en el contexto.

El codificador

Presentamos el algoritmo de codificación de un símbolo s . Sea $c[i]$ el contexto actual de orden i y sea $L[c[i]]$ la lista de predicción asociada al contexto $c[i]$ o lo que es lo mismo, la lista de símbolos que alguna vez han sucedido a la cadena $c[i]$. Cada lista de predicción se gestiona según el algoritmo MTF mejorado expuesto en la Sección 2.3.17 y por tanto, cada símbolo tiene asociado un recuento que indica cuantas veces ha ocurrido en ese contexto:

1. Sea $i \leftarrow k$ el orden de predicción.
2. Sea $H \leftarrow \emptyset$ la lista de símbolos diferentes de predicción probados.
3. Mientras $s \notin L[c[i]]$:
 - (a) $H \leftarrow H + L[c[i]]$, acumulando aquellos símbolos que son probados por primera vez. Esto significa que H no almacenará más de una vez el mismo símbolo en ningún momento.
 - (b) Actualizar s en $L[c[i]]$ según el método MTF mejorado.
 - (c) $i \leftarrow i - 1$.
4. Sea e la posición de s en $L[c[i]]$ (el error de predicción).
5. $e \leftarrow e + \text{size}(H)$ (el número de símbolos en H). Así, sumamos todos los errores de predicción para contextos superiores.
6. Emitir el código correspondiente al error de predicción e .
7. Actualizar s en $L[c[i]]$ según el método MTF mejorado.

En otras palabras, lo que el codificador PBT hace básicamente es encontrar la posición del símbolo codificado en una lista de símbolos formada por la concatenación de todas las listas de predicción posibles para el contexto dado, empezando por el de orden mayor. Como es posible que ocurran en contextos inferiores símbolos ya aparecidos en los superiores, estos no son tenidos en cuenta para calcular el error de predicción.

entrada	salida	contextos afectados
a	a	$L[] = \{a, 1 \dots\}$
b	b	$L[a] = \{b, 1\}, L[] = \{b, 1 a, 1 \dots\}$
a	1	$L[b] = \{a, 1\}, L[] = \{a, 2 b, 1 \dots\}$
b	0	$L[a] = \{b, 2\}$
c	c	$L[b] = \{c, 1 a, 1\}, L[] = \{a, 2 c, 1 b, 1 \dots\}$
b	2	$L[c] = \{b, 1\}, L[] = \{b, 2 a, 2 c, 1 \dots\}$
a	1	$L[b] = \{a, 2 c, 1\}$
b	0	$L[a] = \{b, 3\}$
a	0	$L[b] = \{a, 3 c, 1\}$
b	0	$L[a] = \{b, 4\}$
a	0	$L[b] = \{a, 4 c, 1\}$
a	1	$L[a] = \{b, 4 a, 1\}, L[] = \{a, 3 b, 2 c, 1 \dots\}$
a	1	$L[a] = \{b, 4 a, 2\}$
a	1	$L[a] = \{b, 4 a, 3\}$
a	0	$L[a] = \{a, 4 b, 4\}$
a	0	$L[a] = \{a, 5 b, 4\}$
a	0	$L[a] = \{a, 6 b, 4\}$

Figura 2.29: Ejemplo de codificación PBT.

Como ejemplo vamos a codificar la secuencia `ababcbababaaaaaa` para un orden máximo de predicción igual a uno. Inicialmente sólo existe la lista de predicción de orden 0 ($L[]$) que contiene todos los símbolos que aparecen en el alfabeto ASCII ordenados normalmente según su código ASCII. El recuento de todos ellos es 0. El primer símbolo (`a`) no puede ser codificado atendiendo al símbolo anterior porque es el primero así que se emite sin codificar (de hecho, esto produce el mismo efecto que buscarlo en $L[]$ y emitir su posición, que es realmente lo que se hace). En la siguiente iteración, `a` se convierte en el contexto actual ($c[1] = a$) y tratamos de encontrar en él, el símbolo `b`. Como $L[a] = \emptyset$, H no crece (ver el paso 3.a). Se actualiza el recuento de `b` en $L[a]$ resultando que $L[a] = \{b, 1\}$. Ahora buscamos `b` en $L[]$ y emitimos su posición que es igual a `b`. `b` se convierte en el contexto actual y buscamos en él el siguiente símbolo que es una `a`. No está y como la lista estaba vacía, $L[b]$ se actualiza provocando que $L[b] = \{a, 1\}$. Ahora buscamos en $L[]$ donde encontramos al símbolo `a` en la posición 1, etc. En la Figura 2.29 podemos ver de forma resumida el resto de la codificación.

El descodificador

El proceso de descodificación (la transformación inversa) para un código e puede ser descrita por:

1. Sea $i \leftarrow k$ el orden de predicción.

entrada	salida	contextos afectados
a	a	$L[] = \{a, 1 \dots\}$
b	b	$L[a] = \{b, 1\}, L[] = \{b, 1 a, 1 \dots\}$
1	a	$L[b] = \{a, 1\}, L[] = \{a, 2 b, 1 \dots\}$
0	b	$L[a] = \{b, 2\}$
c	c	$L[b] = \{c, 1 a, 1\}, L[] = \{a, 2 c, 1 b, 1 \dots\}$
2	b	$L[c] = \{b, 1\}, L[] = \{b, 2 a, 2 c, 1 \dots\}$
1	a	$L[b] = \{a, 2 c, 1\}$
0	b	$L[a] = \{b, 3\}$
0	a	$L[b] = \{a, 3 c, 1\}$
0	b	$L[a] = \{b, 4\}$
0	a	$L[b] = \{a, 4 c, 1\}$
1	a	$L[a] = \{b, 4 a, 1\}, L[] = \{a, 3 b, 2 c, 1 \dots\}$
1	a	$L[a] = \{b, 4 a, 2\}$
1	a	$L[a] = \{b, 4 a, 3\}$
0	a	$L[a] = \{a, 4 b, 4\}$
0	a	$L[a] = \{a, 5 b, 4\}$
0	a	$L[a] = \{a, 6 b, 4\}$

Figura 2.30: Ejemplo de descodificación PBT.

2. Sea $H \leftarrow \emptyset$ el conjunto de símbolos aparecidos.
3. Sea $e \leftarrow$ siguiente error de predicción.
4. Mientras $\text{size}(H) < e$:
 - (a) $H \leftarrow H + L[c[i]]$.
 - (b) $i \leftarrow i - 1$.
5. Emitir $s \leftarrow H[e]$.
6. Mientras $i \leq k$:
 - (a) Actualizar s en $L[c[i]]$ según el método MTF mejorado.
 - (b) $i \leftarrow i + 1$.

La Figura 2.30 muestra un ejemplo de descodificación usando la PBT. El primer código (error de predicción) de entrada es **a**, al no existir contexto previo, buscamos el símbolo en $L[]$ que está en la posición **a** y este es precisamente el símbolo **a**, que es emitido por el descodificador. El siguiente código es **b**, lo que provoca que $L[a] = \{b, 1\}$. Como este contexto estaba vacío, es necesario buscar el símbolo en $L[]$. Encontramos el símbolo **b** en la posición **b** y es emitido, etc.

Otras implementaciones de la PBT

Existen implementaciones distintas de la PBT que no usan modelos de Markov para crear las listas de predicción. En [29] se presenta un algoritmo de predicción que utiliza una ventana deslizante al estilo LZ77 para construir las listas. El algoritmo de búsqueda de cadenas usado en LZ77 se utiliza para encontrar todas las ocurrencias del contexto actual en la ventana. La lista de predicción se construye con todos los símbolos que suceden a estas ocurrencias, ordenados primero por orden de predicción y segundo por proximidad a la cabeza de la ventana, donde están los símbolos más próximos al que se trata de codificar. Esta clase de implementación de la PBT es sensiblemente inferior en velocidad y en rendimiento a la presentada en esta sección porque:

- Es necesario buscar todas las ocurrencias de las cadenas y no una de ellas como ocurre en LZ77.
- La cantidad de información que almacena la ventana es muy inferior a la que puede llegar a almacenar el modelo de Markov.

Por el contrario, la ventaja de usar la ventana es que la cantidad de memoria consumida está limitada por su tamaño y es normalmente bastante inferior a la que emplea el modelo de Markov.

MTF frente a PBT

La codificación MTF puede ser también extendida a cualquier orden de predicción y de esta forma, podemos comparar los rendimientos de la PBT y de la codificación MTF basada en el contexto.

La Tabla 2.3 muestra las entropías del error de predicción utilizando ambos algoritmos en función del orden de predicción, para las 8 imágenes de prueba (ver Apéndice A). Se puede apreciar claramente que, excepto para el orden de predicción 0 que normalmente carece de interés, PBT produce valores de la entropía inferiores a los obtenidos mediante MTF.

Consideraciones sobre el modelo de Markov

En este apartado estudiaremos el funcionamiento del modelo utilizado en PBT. Todas las consideraciones expuestas son también válidas para PPM pues usa un modelo idéntico.

La Figura 2.31 trata de mostrar la relación que existe entre la entropía del error de predicción usando PBT y la entropía de los datos originales (en este ejemplo la imagen “lena”). Esta figura también nos da una idea de la cantidad de correlación que se está explotando cada vez que se aumenta el orden máximo de predicción.

La Figura 2.32 es una ampliación de los primeros 4096 códigos de la Figura 2.31 y muestra la inercia que presenta el modelo de Markov. Cuando el orden de predicción es 0 (PBT 0), la entropía del error de predicción está por encima de la entropía de la imagen

Tabla 2.3: Entropía (MTF/PBT) del error de predicción para las imágenes de prueba cuando usamos MTF y PBT en función el orden de predicción.

imagen	orden de predicción					
	0	1	2	3	4	5
lena	6.79/7.31	5.64/5.25	5.61/5.36	5.62/5.38	5.62/5.38	5.62/5.38
bárbara	6.99/7.35	6.53/6.13	6.49/6.26	6.50/6.27	6.50/6.27	6.50/6.27
boats	6.18/6.63	5.70/5.38	5.69/5.46	5.69/5.49	5.69/5.49	5.69/5.49
zelda	6.68/7.13	5.26/4.92	5.31/5.07	5.31/5.09	5.31/5.09	5.31/5.09
29jul24	4.59/4.84	4.44/4.28	4.44/4.31	4.44/4.32	4.46/4.37	4.46/4.37
29jul25	5.62/5.62	5.47/5.36	5.31/5.23	5.31/5.26	5.31/5.26	5.33/5.27
29jul26	5.48/5.53	5.33/5.26	5.18/5.11	5.16/5.11	5.16/5.12	5.16/5.10
B0100	3.50/3.89	3.49/3.88	3.38/3.22	3.38/3.22	3.52/3.38	3.52/3.38

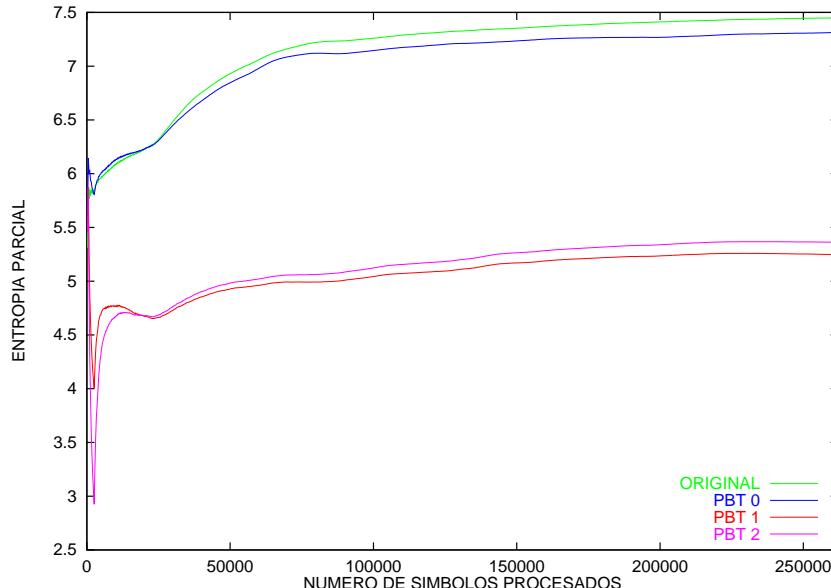


Figura 2.31: Entropías parciales de “lena” y aplicando la PBT.

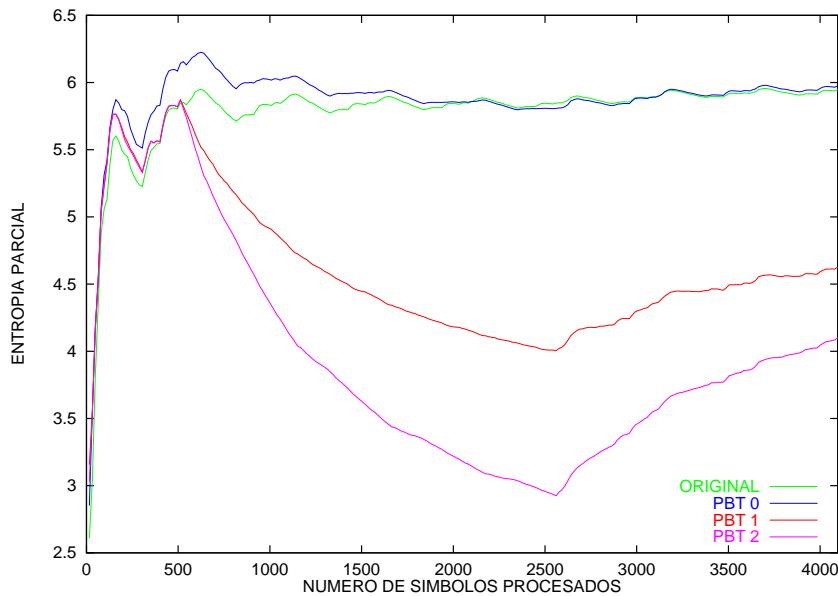


Figura 2.32: Detalle de las entropías parciales para “lena” aplicando la PBT, para mostrar la inercia del modelo.

“lena”, pero cuando ya han sido procesados alrededor de 2000 símbolos, ambas se igualan. Cuando el orden de predicción es 1, el rendimiento del modelo mejora notablemente, aunque todavía presenta algo de inercia, pues la curva está por encima de la entropía original. Justamente a partir del símbolo 512 (“lena” tiene 512×512 puntos) el modelo comienza a predecir muy bien. Cuando el orden de predicción es 2, el modelo funciona aún mejor, pero como puede verse en la Figura 2.31 el modelo de segundo orden se poluciona rápidamente ya que en las imágenes el número de patrones posibles parecidos entre sí es muy alto. Por esta razón, el modelo de orden 1 termina siendo mejor que el de orden 2. Digamos que al aumentarse el orden, el modelo recuerda demasiados contextos que jamás van a aparecer. Estos contextos innecesarios provocan que:

- Las predicciones sean de menor calidad debido a la gran cantidad de símbolos ocurridos en contextos de órdenes que realmente no explotan la correlación entre símbolos. En otras palabras, los errores de predicción son mayores.
- El modelo consume memoria de forma innecesaria para almacenar estos contextos, con el consiguiente descenso de velocidad.

La Tabla 2.4 muestra el número de contextos y la cantidad de memoria, medida en bytes, que el modelo de Markov utiliza para almacenar las listas de predicción. Se han tomado medidas para las 8 imágenes de prueba (ver apéndice A).

Todas las imágenes son de tono continuo y generan un número enorme de contextos, a pesar de que su tamaño no es excesivo (512×512 puntos). Por ejemplo, para “lena” cuando

Tabla 2.4: Número de contextos (cont.) generados y memoria (mem.) consumida por PBT.

imagen	orden 0		orden 1		orden 2		orden 3	
	cont.	mem.	cont.	mem.	cont.	mem.	cont.	mem.
lena	1	2056	216	130025	15969	1177028	127153	4150917
bárbara	1	2056	221	214718	26555	1724587	182371	5321516
boats	1	2056	223	157568	19410	1250991	132103	4009519
zelda	1	2056	187	86692	10556	919103	101646	3736662
B0100	1	2056	256	17817	1938	127430	13431	421694
29jul24	1	2056	256	32649	3792	426514	48488	1656923
29jul25	1	2056	256	67409	8137	1529420	180933	7449466
29jul26	1	2056	256	54129	6477	1368428	162859	6505255

usamos un orden de predicción máximo igual a 3, existen 127153 contextos distintos de orden 0, 1, 2 o 3. En concreto existen $127153 - 15969 = 111184$ contextos de orden 3. La cantidad de memoria empleada en ellos es $4150917 - 1177028 = 2973889$ bytes. En promedio cada contexto usa $2973889/111184 = 26.7$ bytes. Las listas de predicción han sido implementadas mediante listas enlazadas donde cada nodo utiliza 6 bytes (uno para el símbolo, otro para el recuento y 4 para el puntero al siguiente nodo). Esto indica que en promedio, después de conocer el valor de 3 puntos de la imagen tenemos aún casi 5 alternativas, lo que da una idea aproximada de la baja correlación estadística que existe entre puntos tan alejados.

2.3.18 BWT frente a PBT

La principal ventaja de PBT frente a BWT consiste en que la secuencia puede ser tratada como una corriente. Este factor, sin embargo, puede repercutir sensiblemente en la tasa de compresión puesto que:

- Inicialmente un modelo de Markov no sabe nada acerca de la secuencia, de cómo unos símbolos se relacionan con otros y además puede pasar un cierto tiempo hasta que este conocimiento es adquirido. El tiempo que transcurre depende básicamente del orden del modelo que a su vez está relacionado con la cantidad de correlación existente.

Debe tenerse en cuenta que un modelo de Markov de orden n , que aprende de una secuencia generada a partir de un alfabeto con m símbolos, puede llegar a necesitar una cantidad de memoria proporcional a m^n , lo que significa que cantidades enormes de símbolos deben ser procesados hasta que el modelo prediga adecuadamente. Un método de compresión basado en la BWT no padece este tipo de inercia y por lo tanto, pueden esperarse ligeras mejoras en la tasa de compresión.

Tabla 2.5: Entropías arrojadas por BWT+MTF vs PBT para las imágenes de prueba.

	lena	bárbara	boats	zelda	29jul24	29jul25	29jul26	B0100
BWT+MTF	5.45	6.37	5.58	5.14	4.45	5.39	5.25	3.34
PBT	5.25	6.13	5.38	4.92	4.28	5.23	5.11	3.22

- Aunque en ambas técnicas el orden de predicción es fijo, PBT usa uno mucho menor que BWT. Mientras PBT usa órdenes típicos entre 1 y 5, para BWT es tan grande como el tamaño del bloque [15]. Esta ventaja es normalmente muy pequeña porque en la mayoría de los ficheros la correlación de este orden es muy baja, lo cual es especialmente cierto en el caso de las imágenes.

A pesar de estos problemas, PBT mejora a BWT+MTF de forma consistente. La Tabla 2.5 muestra las entropías para BWT+MTF y para la PBT. Los datos numéricos para PBT han sido extraídos de la Tabla 2.3, eligiendo el mejor caso.

2.3.19 PPM frente a BWT y PBT

Desde el punto de vista del tipo de redundancia eliminada, BWT+MTF, PBT y PPM son equivalentes [15]: Todos ellos explotan en lo posible la correlación estadística presente en la mayoría de secuencias de texto. Sin embargo, la compresión “al estilo PPM” alcanza en general, tasas de compresión superiores, incluso contando con el lastre de tener que codificar los símbolos especiales ESC.⁹ Justamente debido a su uso, a la hora de codificar aritméticamente (comprimir) un símbolo, PPM conoce cuántos símbolos son posibles en el contexto actual y calcula la probabilidad de ellos en consonancia. Sin embargo, BWT+MTF o PBT no conocen dicha información de contexto, manteniéndose siempre el tamaño del alfabeto constante. En otras palabras, no es posible explotar el conocimiento que PPM tiene acerca de la existencia de un conjunto de símbolos que no deben ser tenidos en cuenta en la codificación del símbolo actual, y por lo tanto, las tasas de compresión disminuyen.

Todo esto es igualmente cierto si se usa cualquier otra codificación:

- Si usamos la codificación de Huffman, el problema es idéntico: el árbol es en general más pequeño usando PPM. De esta forma, menos bits de código son asignados a cada símbolo.
- Si se utiliza el código de Golomb o el de Rice, aparece un problema relacionado con la estimación del parámetro m y k (recordemos que estos definen la pendiente de la SDPG). Sin la información de contexto es muy costoso determinar con exactitud los valores correctos de los parámetros, y las consecuencias de ello en la tasa de

⁹Recuérdese que BWT y PBT no usan dicho símbolo.

compresión pueden ser importantes. Si m o k son demasiado pequeños, la codificación de un símbolo grande generará un código muy largo. Si son demasiado grandes, la codificación de un símbolo pequeño (que es frecuente) producirá un código demasiado largo.

En lugar de usar las técnicas de estimación descritas en los Apartados 2.3.10 y 2.3.11 lo más natural sería aprovechar el contexto de predicción. Supongamos por ejemplo que utilizamos la PBT. El orden de predicción para el símbolo actual es el más alto posible y además el número de símbolos en la lista de contexto es bajo. En estas circunstancias la generación de errores de predicción pequeños es muy probable por lo que los parámetros debieran ser pequeños. Si el orden de predicción se decrementa porque no ha sido encontrado el símbolo para el contexto actual o las tablas de contexto están muy cargadas de símbolos, la probabilidad de generar un error más grande crece, con lo que deberían también crecer los parámetros. El caso más extremo aparece en la codificación del símbolo especial EOF que, como recordaremos, codifica el fin de la secuencia. Si trabajamos con alfabetos de 256 símbolos, dicho símbolo se representa normalmente con el valor 256. Si por ejemplo, estamos usando la codificación de Rice con $k = 0$ y no es posible usar la información de contexto para conocer que el símbolo que se va a codificar (el que representa el fin de fichero) es muy improbable, construiremos un código de 257 bits.

Para aliviar este problema, existen intentos de mantener el alfabeto con un tamaño lo más reducido posible [27, 28, 29]. Se basan en la idea de que la probabilidad de usar símbolos de índice alto es improbable y por lo tanto, se puede utilizar un alfabeto reducido durante largos períodos de tiempo. Así, cuando llega uno de estos símbolos raros, el codificador entrópico emite un símbolo especial de significado parecido al ESC indicando que es necesario ampliar (por ejemplo hasta la siguiente potencia de dos) el tamaño del alfabeto para codificar el siguiente símbolo. Sin embargo, al no disponer de la información exacta de contexto, nunca es posible estimar con precisión la probabilidad del símbolo a codificar, tal y como PPM es capaz de realizar.

2.4 Evaluación

En esta sección se van a presentar los resultados obtenidos de la evaluación de los algoritmos descritos en este capítulo, usando la batería de imágenes presentadas en el Apéndice A. Para cada uno de los algoritmos existen diferentes implementaciones con particularidades propias. De esta forma, es posible encontrar implementaciones que usan sólo un algoritmo y otras en las que se aplica más de una técnica de compresión. A pesar de este inconveniente, se ha tratado de agrupar a las implementaciones atendiendo al algoritmo dominante que usan. Por esta razón, existen tres grandes grupos: RLE, Lempel-Ziv y Entrópico, que coinciden con las tres grandes secciones de las que se compone este capítulo.

Cada imagen ha sido comprimida de forma independiente a la hora de obtener las tasas de compresión. Las tasas de transferencia están calculadas a partir de tiempos reales de

ejecución (usuario+sistema), teniéndose cuidado de que el sistema operativo no tuviera en ese instante las imágenes en los buffers de entrada/salida, ya que los tiempos no hubieran sido exactos. Todas las medidas fueron realizadas en un Intel Pentium a 200 MHz que monta un disco SCSI y 128 MBytes de RAM y 72 MBytes de swapping en disco. El sistema operativo ha sido Linux y la utilidad usada para tomar los tiempos fue *time*.

Las imágenes de prueba son 8 en total y tienen dos orígenes y características diferentes. El primer grupo está formado por las imágenes “lena”, “bárbara”, “boats” y “zelda”. Todas tienen unas dimensiones de 512×512 puntos y una profundidad de 8 bits, por lo que pueden clasificarse como de tamaño medio. Su característica fundamental es que son imágenes de tono continuo debido a que la mayor parte de las escenas presentan transiciones suaves o relativamente suaves entre puntos vecinos. Más información acerca de estas 4 imágenes puede encontrarse en el Apéndice A.

El segundo grupo lo forman 4 imágenes llamadas “B0010”, “29jul24”, “29jul25” y “29jul26”. Sus dimensiones son 1024×1083 (para “B0010”) y 1024×1148 (para las otras 3). Todas tienen una profundidad de 16 bits. Contiene escenas astronómicas y no pueden considerarse exactamente como imágenes de tono continuo. En realidad pueden ser descritas como un conjunto de puntos blancos de diferente intensidad y grosor, que se colocan arbitrariamente sobre un fondo oscuro que puede llegar a ser, para algunas imágenes, bastante ruidoso (lo que dificulta enormemente su compresión sin pérdidas). Información más detallada puede también ser encontrada en el Apéndice A.

Los detalles particulares de cada implementación son descritos a continuación, donde además, se realiza un comentario acerca de los resultados que se obtienen. Todos ellos se presentan conjuntamente en las Tablas 2.6 y 2.7 para las imágenes de 8 bpp y en las Tablas 2.8 y 2.9 para las imágenes de 16 bpp.

- *RLE-MNP5* [39] es un algoritmo muy poco eficaz en el contexto de la compresión de imágenes de tono continuo. 5 de las 8 imágenes fueron expandidas y las restantes no pudieron comprimirse como consecuencia de que las series son muy cortas. Lo único positivo que es posible resaltar es que las tasas de transferencia son bastante buenas.
- *lzss* usa el algoritmo estándar presentado en la Sección 2.2.1. La implementación evaluada ha sido extraída de [64] y utiliza códigos de longitud constante para representar las direcciones de la ventana deslizante y las longitudes de las cadenas encontradas. En concreto, el tamaño de la ventana fue de 4096 caracteres y el del buffer de anticipación de 16 caracteres.

Para las imágenes de 8 bpp, la tasa de compresión es bastante pobre (0.011) y 2 de las 4 imágenes fueron expandidas. En las imágenes de 16 bpp la tasa de compresión mejora sensiblemente aunque *lzss* es el segundo en la lista de los peores compresores. La tasa de transferencia del compresor es relativamente mala, aunque la del descompresor mejora en un factor de 4 aproximadamente. Esta asimetría es típica en todos los compresores basados en diccionarios.

- *gzip -9* [31] es una mejora notable de LZ77 que usa un código de Huffman para expresar las referencias a la ventana y las longitudes de las cadenas. Ha sido invocado con el flag *-9* que consigue la máxima compresión posible. El tamaño de la ventana deslizante es de 32768 caracteres y el tamaño del *buffer* de anticipación es de 256 caracteres. Si comparamos estos valores con los de *lzss* y tenemos en cuenta que luego usamos la codificación de Huffman, entenderemos porqué *gzip* consigue unas tasas de compresión entre 3 y 15 veces superiores.

El compresor utiliza *hashing* para localizar las cadenas en la ventana (*lzss* usa un árbol binario) consiguiendo 4 veces más tasa de transferencia en el caso de las imágenes de 8 bpp y casi el doble en el caso de las imágenes de 16 bpp. Si tenemos en cuenta las tasas de compresión que se alcanzan, descubrimos que *gzip* es un excelente compresor de imágenes. Además, la tasa de transferencia del descompresor es la mejor de todos los compresores probados, lo que hace que *gzip* sea el compresor ideal para almacenar ficheros permanentemente comprimidos en elementos de almacenamiento rápidos (por ejemplo, discos duros).

- “*ha a*” [40] utiliza LZ77 y la codificación aritmética en lugar de la de Huffman para representar las referencias a la ventana y las longitudes de las cadenas. Esto genera una ganancia aproximada del 2% de la tasa de compresión, pero las tasas de transferencia caen mucho respecto a las de *gzip*. En concreto, el descompresor es más de 10 veces más lento.
- *lzw12* [64] implementa el algoritmo LZW (ver Sección 2.2.4) sin ninguna mejora extra. El número de cadenas que puede llegar a contener el diccionario es 4096. Las tasas de compresión obtenidas son las peores de todos los compresores probados. De las 8 imágenes, sólo 2 fueron comprimidas algo. Aunque el descompresor supera en velocidad al compresor, se puede decir que son bastante simétricos. En la implementación probada, el compresor utiliza una búsqueda multicamino y las tasas de transferencia son bastante buenas.
- *hlzw 16* [99] es otra implementación del algoritmo LZW que usa un diccionario de 2^{16} entradas. La búsqueda se realiza como en una tabla *hash*. Las altas tasas de transferencia alcanzadas (sólo superadas por *compress*) en el compresor se deben a que cuando ocurre una colisión se interpreta que la cadena no está en el diccionario, lo que también provoca que baje las tasas de compresión. En teoría, el descompresor *hlzw 16* debería ser lo mismo de rápido que *lzw 17*. La diferencia radica en que este último usa punteros de 32 bits para construir el diccionario, mientras que *hlzw 16* utiliza punteros de 16 bits.
- *lzw15v* [64] utiliza también el algoritmo LZW, pero en el diccionario caben 32768 cadenas. Además emplea un código de longitud incremental y proporcional al tamaño del diccionario. *lzw15v* monitoriza constantemente la tasa de compresión para ver si cae por debajo de un determinado umbral. Cuando esto ocurre, vacía el diccionario

por considerar que se ha polucionado con datos que no son útiles en el contexto actual de la secuencia a comprimir.

Todas estas mejoras (y en especial el uso de códigos de longitud creciente) provocan un aumento espectacular de la tasa de compresión respecto de *lzw12*. Las tasas de transferencia son iguales o ligeramente inferiores a pesar de usar un diccionario 8 veces superior.

- *compress* [112] es una implementación muy rápida del algoritmo LZW. Utiliza un diccionario de 65536 cadenas y todas las mejoras introducidas en *lzw15v*. La duplicación del tamaño del diccionario proporciona un aumento del 20% de la tasa de compresión en las imágenes de 8 bpp y del 8% en las imágenes de 16 bpp. El compresor *compress* es el más rápido de todos los probados (con diferencia) y el descompresor sólo es superado ligeramente por *gzip*.
- *lzw 17* [98] es una implementación más de LZW que permite trabajar con diccionarios muy grandes. En nuestra evaluación, el diccionario es capaz de almacenar hasta 128K cadenas. A pesar de que no existe monitorización de la tasa de compresión para vaciarlo si se poluciona, la tasa de compresión es superior a la de *compress* en un 6% para las imágenes de 8 bpp y en 0.2% en las imágenes de 16 bpp. Esto prueba que el tamaño del diccionario es un parámetro determinante en este tipo de compresores.

El compresor *lzw* es aproximadamente el doble de lento que el descompresor y las tasas de transferencia pueden calificarse como buenas.

- *huffs* [64] es una sencilla implementación de la codificación de Huffman y un modelo probabilístico estático sin memoria. En la primera pasada calcula las probabilidades de los símbolos y en la segunda realiza la codificación. A pesar de la simplicidad del método, las tasas de compresión son casi tan buenas como los de *lzw15v* lo que nos obliga a pensar que las imágenes tienen poca correlación estadística de orden superior.

Las tasas de transferencia del compresor y del descompresor son muy parecidas, lo que es típico en todos los compresores de texto entrópicos.

- *huffa* [64] utiliza el código de Huffman y un modelo probabilístico adaptativo sin memoria. La compresión se realiza por lo tanto en una sola pasada. La adaptatividad del modelo provoca una mejora del 17% de la tasa de compresión en las imágenes de 8 bpp y del 9% en las imágenes de 16 bpp, lo que indica que las imágenes de 8 bpp son menos “ergódicas”.

Las tasas de transferencia empeoran aproximadamente en un 40% respecto de *huffs* debido al coste extra de usar un algoritmo dinámico.

- *huff1a* es una versión mejorada de *huffa* que usa un modelo probabilístico adaptativo de orden 1. Las mejoras en la tasa de compresión respecto a *huffa* son notables: un 64% para las imágenes de 8 bpp y un 23% para las imágenes de 16 bpp. De

estos resultados concluimos que (1) las imágenes de 8 bpp tienen casi 3 veces más correlación estadística de orden 1 que las de 16 bpp y (2) que la correlación estadística de orden 1 es (en general) bastante alta en las imágenes.

El compresor *huff1a* es bastante asimétrico, resultando un descompresor el doble de rápido que el compresor. El coste temporal de usar un modelo de orden 1 es alto y disminuye las tasas de transferencia en aproximadamente un factor de 4.

- *aritha* [128] utiliza un codificador aritmético de 256 símbolos y un modelo probabilístico adaptativo sin memoria. La mejora introducida por el codificador (respecto de *huffa*) es aproximadamente del 12% para las imágenes de 8 bpp y del 1% para las imágenes de 16 bpp.

Si comparamos las tasas de transferencia con las de *huffa*, vemos que el coste del codificador es importante ya que genera compresores entre 2 y 3 veces más lentos cuando usamos modelos de orden 0.

- *arith1a* [64] es equivalente a *huff1a* pero el codificador es aritmético. Los resultados son un 1% mejores comparados con éste, para las imágenes de 8 bpp y un 6% mejores para las imágenes de 16 bpp. Tal vez pudiera parecer chocante que cuando el modelo probabilístico no tiene memoria (como ocurre en *aritha* y *huffa*), la diferencia entre usar un codificador aritmético y un codificador de Huffman es mayor cuando comprimimos imágenes de 8 bpp (ahora ocurre justo a revés).

Debemos tener en cuenta que las imágenes astronómicas representan cada punto usando 2 bytes (símbolos de 8 bits) y que en la mayoría de los casos, los puntos son prácticamente negros. Como consecuencia, las secuencias de puntos típica analizada como una fuente de 256 símbolos es de la forma: 0X0Y... pues el byte más significativo de cada punto casi siempre es 0 (o un número muy próximo a cero) y el siguiente byte puede ser muy variable debido al ruido presente en la mayoría de las imágenes. El modelo de orden 1 aprende estadísticamente de este tipo de secuencias y cuando se ha comprimido el símbolo que representa al byte menos significativo (el más aleatorio), la probabilidad de codificar un 0 (o un valor cercano) es muy alta. Si ésta supera el 50%, la codificación de Huffman comienza a introducir gran cantidad de redundancia (ver Sección 2.3.6). Como es de esperar, este problema se hace más importante cuando el orden del modelo aumenta. Debido a esto, la mayoría de los compresores entrópicos que usan modelos con memoria utilizan la codificación aritmética como codificador entrópico a pesar del coste computacional añadido.

El coste de usar un modelo de orden 1 (comparando con *aritha*) enlentece el esquema de compresión alrededor de un 30%.

- *arith1ae* [64] es equivalente a la implementación anterior salvo que el modelo está inicialmente vacío. La mejora en la tasa de compresión es prácticamente insignificante en las imágenes de 8 bpp porque casi desde el comienzo de la compresión, prácticamente todos los símbolos posibles han sido vistos. En el caso de las imágenes de 16

bpp, el uso de un modelo inicialmente vacío mejora la tasa de compresión casi en un 3%. Esto tiene sentido si tenemos en cuenta que tras la codificación de un byte menos significativo vamos a tener una lista de contexto bastante reducida, lo que permite estimar con más precisión la probabilidad de los símbolos más probables (0 y próximos).

La utilización del símbolo ESC prácticamente no añade complejidad al descompresor y por lo tanto las tasas de transferencia se mantienen. En el caso del comprresor, el coste computacional adicional es ligero y sólo se aprecia en las imágenes de 8 bpp.

- *PPMZ* [10] es el comprresor de datos que ostenta (actualmente) la máxima tasa de compresión sobre el *Calgary Text Compression Corpus*. Es una versión mejorada del algoritmo PPMC (ver Secciones 2.3.5 y 2.3.14) que usa un orden máximo de predicción variable (entre 0 y 12), dependiendo de la probabilidad del símbolo más probable para el contexto actual.

En general, *PPMZ* es un comprresor extremadamente lento y consume una cantidad de memoria ingente, pero las tasas de compresión son excelentes (al menos, para la compresión de ficheros de texto).

En el caso de la compresión de imágenes el rendimiento no es tan abrumador como cabría esperar. Para las imágenes de 8 bpp, la tasa de compresión es un 3% mejor que la de *arith1ae*, aunque esto es comprensible si tenemos en cuenta la baja cantidad de redundancia que existe debido a la correlación estadística de orden 2 y superior. Para las imágenes de 16 bpp *PPMZ* sólo pudo terminar para “B0010” por falta de memoria.¹⁰

Las tasas de transferencia son muy bajas y se justifican sólo si los ficheros van a ser almacenados de forma comprimida durante largos períodos de tiempo sin hacer uso de ellos, o si se usa en transmisiones sobre canales muy lentos. En concreto, el descompresor *PPMZ* es alrededor de 77 veces más lento que el de *gzip*.

En la implementación evaluada es posible fijar el orden máximo de predicción, por lo que existen 5 evaluaciones más:

- *PPMZ -c8* usa un modelo sin memoria. La referencia anteriormente probada más cercana es *aritha*. En el caso de las imágenes de 8 bpp, *PPMZ -c8* es sensiblemente superior debido probablemente al uso de un modelo inicialmente vacío. Para las imágenes de 16 bpp, *aritha* es mejor. *PPMZ -c8* además es mucho más lento que *aritha*.
- *PPMZ -c7* utiliza un modelo de orden 1. El comprresor más similar es *arith1ae* al que mejora en un 7% en las imágenes de 8 bpp y empeora en un 2% en las imágenes de 16 bpp. Las tasas de transferencia son similares.

¹⁰En este caso se agotaron los 128+72 MBytes de memoria virtual.

- *PPMZ -c6* usa un modelo de orden 2. En el caso de 8 bpp las tasas de compresión empeoran en un 3.4% al aumentar el orden. Para las imágenes de 16 bpp, donde al parecer existe más correlación estadística (lógico ya que los puntos ahora ocupan dos símbolos de 8 bits contiguos) las tasas de compresión mejoran en un 5.1%. Las tasas de transferencia son entre un 20% y un 40% inferiores para las imágenes de 8 bpp y similares en el caso de 16 bpp.
- *PPMZ -c5* y *PPMZ -c4* utilizan respectivamente un modelo de orden 3 y 4. En todos los casos, las tasas de compresión y transferencia empeoran al no existir suficiente redundancia de este orden.
- *PPM** [15] es una implementación de PPMC que, al igual que *PPMZ*, no tiene un límite máximo de predicción. El resultado es un compresor y descomprimidor extremadamente lentos, que en el marco de la compresión de imágenes obtienen unas tasas de transferencia muy pobres y bastante inferiores a los de *PPMZ*, que es el algoritmo más parecido. Además, la cantidad de memoria consumida es tal que las imágenes de 16 bits no pudieron ser comprimidas.
- “*ha a2*” [40] es otra implementación del algoritmo PPMC y un modelo de orden 4, por lo que es equivalente a *PPMZ -c4*. En las imágenes de 8 bpp la tasa de compresión es superior a *PPMZ -c4* en casi un 3%. Las tasas de transferencia son también mejores. En las imágenes de 16 bpp, la tasa de compresión mejora en un 4% lo que convierte a *ha a2* en el mejor comprisor de texto evaluado para este tipo de imágenes. Las tasas de transferencia son también más altas.
- *PBT n+aritha* ejecuta la transformada basada en predicción (ver Sección 2.3.17) seguida de una codificación aritmética adaptativa de orden 0 (modelo inicialmente vacío). El parámetro *n* indica el orden máximo de predicción usado. En las imágenes de 8 bpp la configuración *PBT 1+aritha* es la que mejores tasas de compresión consigue. Las tasas de transferencia para *n = 1* son aceptables. Sin embargo, cuando aumentamos el orden de predicción, las tasas de compresión y de transferencia decaen al no existir redundancia.

Para las imágenes de 16 bpp las tasas de compresión caen aproximadamente un 18% frente a PPM. Estos resultados eran esperados a tenor de los problemas expuestos para este tipo de comprisor en la Sección 2.3.19: el número de símbolos posibles que suceden al byte menos significativo de cada punto es muy bajo y *aritha* mantiene siempre un alfabeto de tamaño constante de 256 símbolos.

- *bzip2 -9* [107] usa el algoritmo BWT (ver Sección 2.3.15) seguido del codificador MTF (ver Sección 2.3.16). A continuación utiliza un código de Huffman para codificar la SDPG generada. Las tasas de compresión y de transferencia son excelentes para ambos tipos de imágenes. Se puede afirmar que *bzip2 -9* arroja tasas de compresión comparables a los de los métodos PPM y una complejidad similar a la de los métodos

LZ77. El único inconveniente es que las secuencias deben ser procesadas por bloques (el flag -9 provoca que el tamaño de estos sea 900000 bytes).

Un aspecto interesante a destacar es que *bzip2* ha resuelto en buena parte el problema de perder la información de contexto para la estimación de la pendiente de la SDPG y del número total de símbolos que el codificador entrópico maneja. Esto se consigue aplicando dos técnicas distintas:

- Utiliza la codificación RLE para minimizar la longitud de las series, especialmente la de los ceros que son las más frecuentes. Tras aplicar RLE, la secuencia ya ha sido parcialmente comprimida. Además, la pendiente de la SDPG ha caído, lo cual ayuda al codificador de Huffman a no ser especialmente redundante en la codificación de los símbolos más frecuentes.
- Usa un alfabeto lo más reducido posible tal y como se expone en la última parte de la Sección 2.3.19. El codificador maneja 8 modelos que contemplan sucesivamente más símbolos. El primero está formado por el símbolo 0, el 1 y el ESC, el segundo por los símbolos 2, 3 y el ESC, el tercero por los símbolos 4, 5, 6, 7 y el ESC, etc.

Para finalizar la sección de evaluación de los resultados y con el objetivo de hacer una comparación rápida con otros presentados en secciones posteriores, las Figuras 2.33 y 2.34 muestran los rendimientos conseguidos por los algoritmos estudiados.

Para ello se ha usado el espacio del rendimiento definido en la Sección 1.5.3. Por cada codec evaluado aparece una línea horizontal. Normalmente, el extremo de la izquierda lo marca la tasa de transferencia del compresor (TT_c) y el extremo de la derecha la tasa de transferencia del descomprimidor (TT_d), porque se tarda más tiempo en comprimir que en descomprimir.

Se ha dedicado un símbolo específico a cada uno de los grupos o familias en los que se han descompuesto los diferentes métodos de compresión. El cuadrado se ha usado para representar al único miembro de la familia RLE, una cruz rotada 45° se ha usado para representar a los compresores basados en diccionarios (Lempel-Ziv) y una cruz sin rotar sitúa a los compresores entrópicos.

Por los resultados obtenidos podemos concluir que las tasas de compresión alcanzadas por los compresores de texto, como compresores de imágenes de tono continuo, son bastante bajas, como consecuencia de la baja correlación estadística presente. En el caso de las imágenes de 8 bpp, la máxima tasa de compresión se consigue aplicando la transformada basada en predicción (PBT), pero rara vez se consigue un factor de compresión 2:1. Además, el coste temporal es alto, especialmente si usamos modelos de Markov para extraer la redundancia estadística. En el caso de las imágenes de 16 bpp el coste computacional es muy alto.

Resaltamos la idea de que las tasas de transferencia no son altas, pero esto no significa que el rendimiento de los algoritmos de compresión de texto es bajo. Sencillamente significa

Tabla 2.6: Tasas de compresión (multiplicados por 1000) alcanzados por los algoritmos probados para las imágenes de 8 bpp.

clase	implementación	imágenes				media
		lena	bárbara	boats	zelda	
RLE	<i>rle-MNP5</i>	-7	-5	-8	-6	-7
Lempel-Ziv	<i>lzss</i>	2	-33	79	-6	11
	<i>gzip -9</i>	149	118	228	165	165
	<i>ha a</i>	148	115	243	166	168
	<i>lzw12</i>	-125	-148	-92	42	-81
	<i>hlzw 16</i>	40	-53	84	73	36
	<i>lzw15v</i>	96	-15	145	140	92
	<i>compress</i>	119	19	144	173	114
	<i>lzw 17</i>	134	35	149	171	122
	<i>huffs</i>	65	63	106	87	89
	<i>huffa</i>	80	75	169	105	107
Entrópico	<i>huff1a</i>	325	176	308	371	295
	<i>aritha</i>	90	85	191	118	121
	<i>arith1a</i>	316	216	318	346	299
	<i>arith1ae</i>	330	182	313	376	300
	<i>PPMZ -c8</i>	94	89	205	126	129
	<i>PPM*</i>	291	178	283	316	267
	<i>PPMZ -c5</i>	336	200	329	366	308
	<i>PPMZ -c4</i>	339	200	328	366	308
	<i>PPMZ</i>	340	201	330	369	310
	<i>PPMZ -c6</i>	338	203	331	370	311
	<i>ha a2</i>	338	225	343	359	317
	<i>PPMZ -c7</i>	349	208	340	390	322
	<i>PBT 0+aritha</i>	90	83	199	115	122
	<i>PBT 4+aritha</i>	331	218	33	370	312
	<i>PBT 3+aritha</i>	331	218	331	370	313
	<i>PBT 2+aritha</i>	333	219	334	370	314
	<i>PBT 1+aritha</i>	346	235	342	388	328
	<i>bzip2 -9</i>	337	235	333	359	316

Tabla 2.7: Tasas de transferencia (multiplicadas por 1000) de los algoritmos (comprimidor/descompresor) probados para las imágenes de 8 bpp.

clase	implementación	imágenes				media
		lena	bárbara	boats	zelda	
RLE	<i>rle-MNP5</i>	400 /400	400 /384	384/384	400/384	396/388
Lempel-Ziv	<i>lzss</i>	46/161	46/163	47/178	40/166	45/167
	<i>gzip -9</i>	181/769	192/769	181/769	181/769	184/769
	<i>ha a</i>	49/62	49/59	49/68	50/62	49/63
	<i>lzw12</i>	138/153	133/156	140/163	140/172	138/161
	<i>hlzw 16</i>	300/387	293/375	324/444	300/375	304/395
	<i>lzw15v</i>	131/161	126/151	128/163	138/172	131/162
	<i>compress</i>	384/625	400/666	454/714	416/769	414/694
	<i>lzw 17</i>	129/294	121/285	135/312	138/294	131/296
	<i>huffs</i>	147/138	147/140	149/147	153/142	149/142
	<i>huffa</i>	96/85	97/86	107/88	101/90	100/87
Entrópico	<i>huff1a</i>	30/63	24/46	30/60	33/71	28/60
	<i>aritha</i>	53/32	50/32	57/33	47/33	52/33
	<i>arith1a</i>	37/25	34/23	41/27	33/25	36/25
	<i>arith1ae</i>	37/24	32/22	40/26	33/24	24/24
	<i>PPMZ -c8</i>	17/16	17/16	18/15	18/17	18/16
	<i>PPM*</i>	7/7	7/6	7/7	12/10	8/8
	<i>PPMZ -c5</i>	30/26	19/18	27/25	31/30	27/25
	<i>PPMZ -c4</i>	17/17	14/14	17/17	18/18	17/17
	<i>PPMZ</i>	9/9	10/10	10/10	9/9	10/10
	<i>PPMZ -c6</i>	42/35	24/22	37/31	51/42	39/33
	<i>ha a2</i>	29/30	22/22	29/27	34/34	29/28
	<i>PPMZ -c7</i>	55/41	29/25	43/34	64/44	48/36
	<i>PBT 0+aritha</i>	70/56	69/56	79/64	73/58	73/59
	<i>PBT 3+aritha</i>	36/31	27/23	33/29	39/33	34/29
	<i>PBT 4+aritha</i>	26/22	20/16	27/21	28/22	25/20
	<i>PBT 2+aritha</i>	48/40	31/28	48/43	54/47	45/40
	<i>PBT 1+aritha</i>	77/70	58/51	75/67	83/75	73/66
	<i>bzip2 -9</i>	194/566	180/517	212/545	214/588	200/554

Tabla 2.8: Niveles de compresión (multiplicados por 1000) alcanzados por los algoritmos (compresor/descompresor) probados para las imágenes de 16 bpp.

clase	implementación	imágenes				media
		B0100	29jul24	29jul25	29jul26	
RLE	<i>rle-MNP5</i>	0	0	-1	0	0
Lempel-Ziv	<i>lzss</i>	399	122	-7	0	128
	<i>gzip -9</i>	601	453	312	330	424
	<i>ha a</i>	590	472	325	344	432
	<i>lzw12</i>	-75	348	233	214	180
	<i>hlzw 16</i>	523	331	154	177	296
	<i>lzw15v</i>	576	407	211	232	356
	<i>compress</i>	592	456	240	261	387
	<i>lzw 17</i>	593	447	244	271	388
	<i>huffs</i>	430	357	286	283	339
	<i>huffa</i>	500	397	294	305	374
Entrópico	<i>huff1a</i>	562	521	422	431	484
	<i>aritha</i>	501	400	300	311	378
	<i>arith1a</i>	616	571	428	440	513
	<i>arith1ae</i>	628	584	443	456	527
	<i>PPMZ -c8</i>	502	399	299	309	377
	<i>PPM*</i>	-	-	-	-	-
	<i>PPMZ</i>	707	-	-	-	-
	<i>PPMZ -c7</i>	629	572	428	441	517
	<i>PPMZ -c4</i>	708	550	417	437	528
	<i>PPMZ -c5</i>	709	563	436	457	541
	<i>PPMZ -c6</i>	710	565	446	461	545
	<i>ha a2</i>	703	578	450	470	550
	<i>PBT 1+aritha</i>	519	464	330	343	414
	<i>PBT 0+aritha</i>	516	395	299	310	380
	<i>PBT 3+aritha</i>	599	460	343	361	440
	<i>PBT 2+aritha</i>	599	461	346	361	441
	<i>PBT 4+aritha</i>	579	453	409	360	450
	<i>bzip2 -9</i>	697	567	455	473	548

Tabla 2.9: Tasas de transferencia (multiplicadas por 1000) de los algoritmos probados para las imágenes de 16 bpp.

clase	implementación	imágenes				media
		B0100	29jul24	29jul25	29jul26	
RLE	<i>rle-MNP5</i>	216/236	330/333	327/312	328/310	300/297
Lempel-Ziv	<i>lzss</i>	27/140	39/152	38/133	40/138	36/140
	<i>gzip -9</i>	14/ 671	48/ 787	93/ 711	84/ 704	59/ 718
	<i>ha a</i>	8/74	20/72	34/63	33/66	23/68
	<i>lzw12</i>	81/308	294/317	305/332	318/375	301/333
	<i>hlzw 16</i>	196/315	241/354	227/301	226/306	223/319
	<i>lzw15v</i>	125/151	135/176	117/150	122/338	124/203
	<i>compress</i>	354 /662	359 /704	395 /732	413 /691	380 /697
	<i>lzw 17</i>	116/326	139/366	111/298	116/290	120/320
Entrópico	<i>huffs</i>	129/125	166/156	160/144	159/141	153/141
	<i>huffa</i>	89/82	106/96	92/83	94/85	95/86
	<i>huff1a</i>	24/64	30/29	25/61	26/65	26/54
	<i>aritha</i>	31/22	36/30	39/28	39/28	36/27
	<i>arith1a</i>	23/18	24/21	26/21	25/21	24/20
	<i>arith1ae</i>	23/18	25/22	24/20	24/21	24/20
	<i>PPMZ -c8</i>	5/3	4/5	4/4	4/4	4/4
	<i>PPM*</i>	-	-	-	-	-
	<i>PPMZ</i>	1/1	-	-	-	-
	<i>PPMZ -c7</i>	11/8	9/8	8/7	8/8	9/8
	<i>PPMZ -c4</i>	1/1	3/3	3/3	3/3	2/2
	<i>PPMZ -c5</i>	15/12	8/7	6/5	6/6	8/7
	<i>PPMZ -c6</i>	16/12	8/8	7/7	7/7	9/8
	<i>ha a2</i>	19/17	13/14	5/5	5/5	10/10
	<i>PBT 1+aritha</i>	26/16	18/20	16/15	17/14	19/16
	<i>PBT 0+aritha</i>	18/17	21/20	20/16	20/17	19/17
	<i>PBT 3+aritha</i>	16/13	11/11	5/4	5/5	9/8
	<i>PBT 2+aritha</i>	16/15	14/13	7/5	6/6	10/9
	<i>PBT 4+aritha</i>	13/11	9/7	4/3	4/3	7/6
	<i>bzip2 -9</i>	48/161	52/188	42/161	44/163	46/168

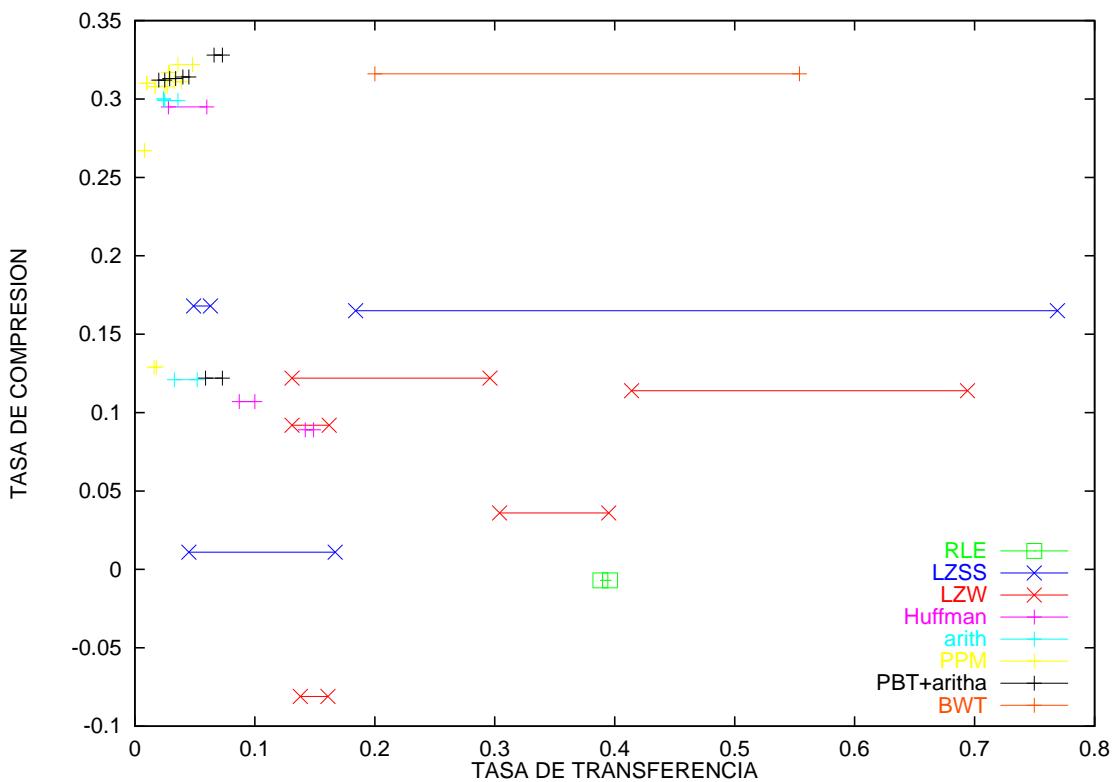


Figura 2.33: Rendimiento de los algoritmos de compresión de texto probados para las imágenes de 8 bpp. Cada línea representa el rendimiento de un compresor. Las tres familias de compresores se diferencian por el símbolo situado en los extremos.

que existe poca correlación estadística en las imágenes que forman nuestra batería de prueba.

bzip2 -9 (BWT+MTF+RLE+Huffman) merece un comentario especial porque ocupa un lugar privilegiado dentro del espacio del rendimiento. Permite alcanzar tasas de compresión próximas a las proporcionadas por PPM arrojando, sin embargo, tasas de transferencia muy superiores.

Los métodos basados en diccionarios (LZ*) presentan unas tasas de compresión medias, aunque siguen siendo los más rápidos con diferencia.

La codificación aritmética mejora el rendimiento alcanzado por el resto de códigos, pero también acarrea mayor coste computacional.

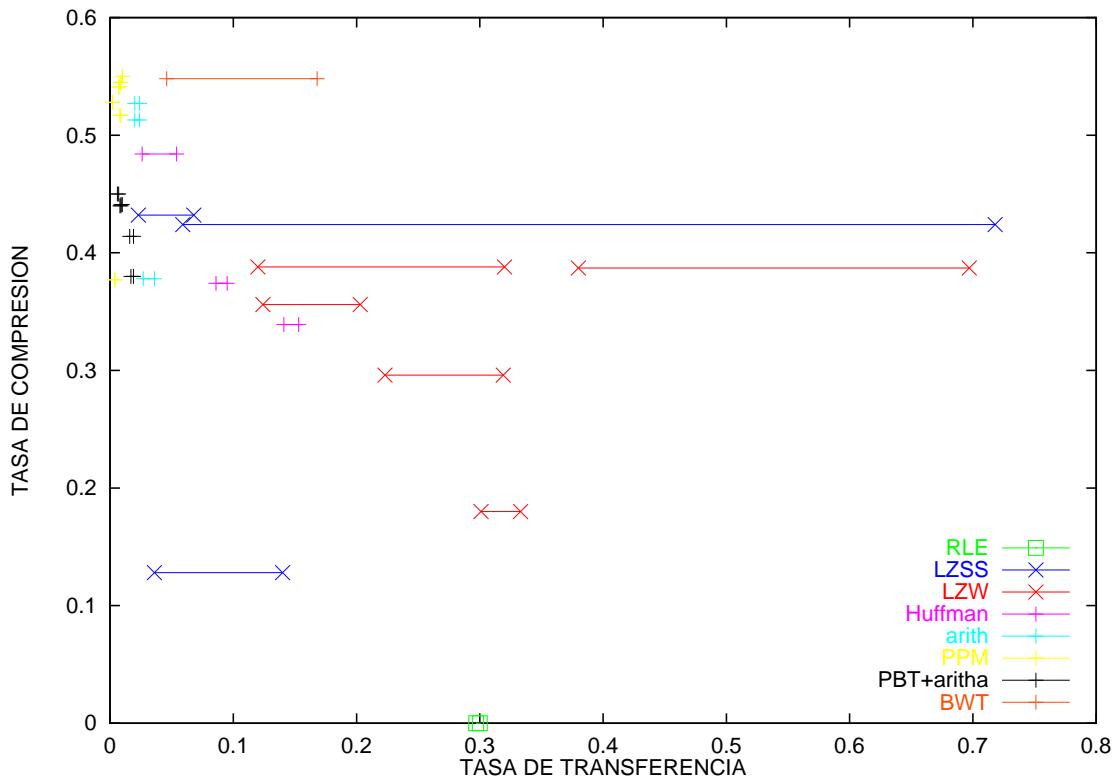


Figura 2.34: Rendimiento de los algoritmos de compresión de texto probados para las imágenes de 16 bpp. Cada línea representa el rendimiento de un compresor. Las tres familias de compresores se diferencian por el símbolo situado en los extremos.

2.5 Resumen

En este capítulo se ha realizado un estudio profundo y extenso de los principales algoritmos de compresión que eliminan la redundancia estadística. Muchos de ellos son algoritmos que constituyen “el estado del arte” en el campo de la compresión de texto. En ese sentido, este trabajo constituye una aportación importante pues presenta una completa clasificación y una evaluación objetiva de todas estas técnicas, llenando el “vacío” que existe en la literatura, en materia de comparativas objetivas entre todas las técnicas.

Un gran número de implementaciones, algunas mezcla de varios algoritmos de compresión independientes, han sido evaluadas como compresores de una batería de imágenes formada por 4 imágenes de tono continuo típicas y por 4 imágenes astronómicas. Estas últimas pueden ser consideradas también como de tono continuo pero con ciertas reservas. Los resultados demuestran que el nivel de correlación estadística en las imágenes de prueba es baja.

La mayor parte de nuestro trabajo investigador se ha centrado fundamentalmente en

las técnicas LZ y en las entrópicas. De él, podemos resaltar lo siguiente:

- En el contexto de la compresión basada en diccionarios, se han desarrollado dos implementaciones diferentes del algoritmo LZW (*lzw* [98] y *hlzw* [99]) que mejoran en velocidad y/o en capacidad de compresión de otras implementaciones estándar (*lzw12*, *lzw15v* y *compress*). Además, *hlzw* puede convertirse fácilmente un compresor de imágenes *lossy* sin que el descompresor tenga que ser modificado, permitiendo descomprimir imágenes comprimidas de forma irreversible y reversible.

Esta línea de trabajo es interesante porque sería posible mejorar el algoritmo de codificación LZW para realizar una construcción más inteligente (con mayor visión de futuro) del diccionario, que permitiera encontrar las cadenas buscadas con mayor acierto. De esta forma, sería posible alcanzar mejores tasas de compresión, sin que la tasa de transferencia del descompresor se viera afectada. Esto es muy útil en aquellas situaciones en las que se comprime una vez y se descomprime muchas veces.

- En el contexto de la compresión entrópica se ha desarrollado e implementado la transformada basada en predicción (PBT). Este algoritmo se ha mostrado (a tenor de los resultados) muy útil como herramienta descorrelacionadora, especialmente en la compresión de imágenes de tono continuo de 8 bits, alcanzando tasas de compresión superiores a los de PPM y BWT, con un coste computacional sensiblemente inferior en el caso de PPM. PBT es más lento que BWT pero permite procesar las secuencias de símbolos como una corriente lo cual es fundamental en los sistemas de transmisión de datos.

El siguiente paso a realizar consistiría en encontrar e implementar un método de codificación más eficiente que el que actualmente se está utilizando para codificar la SDPG resultante, en la misma línea que lo hace *bzip2* con la BWT. Como ya se ha indicado, el principal problema de la PBT es la pérdida de la información de contexto que indica qué símbolos deben ser excluidos. Un codificador que contemple un alfabeto de longitud variable parece ser la mejor opción.

Capítulo 3

Compresión de Imágenes

En este capítulo se analizan algunos de los algoritmos que constituyen “el estado del arte” en compresión de imágenes secuencial sin pérdidas y que son usados en aquellas situaciones en las que no está permitida ninguna modificación del contenido de la imagen.

En general, de todos ellos es posible afirmar que son adaptaciones específicas para el caso de imágenes de tono continuo de los algoritmos de compresión entrópicos. El modelo de inspiración más usado es PPM (ver Sección 2.3.14) que como ya hemos comentado está entre los que mejores tasas de compresión arroja (ver Sección 2.4).

Ahora el modelo probabilístico es un predictor que usa la correlación espacial para generar una predicción por cada punto comprimido. El codificador entrópico expresa, usando un número de bits mínimo, la diferencia entre el valor real del punto y el valor de la predicción espacial.

Los problemas derivados de la inercia del modelo probabilístico (problema de la frecuencia cero) y del consumo excesivo de memoria como consecuencia del alto número de contextos de predicción generados, son resueltos reduciendo significativamente el número de contextos que es posible formar. Básicamente, la idea consiste en mapear muchos contextos similares (estadísticamente hablando) en uno solo, en una operación típicamente llamada “cuantificación de contextos”.

La otra cualidad especial que poseen los compresores de imágenes, y que provoca que las tasas de compresión aumenten, es que se va a explotar la redundancia espacial presente en las dos dimensiones del espacio, cosa que no es posible en los compresores de texto que buscan la correlación estadística en una sola dimensión.

Por último, resaltar una característica común a todos los compresores y descompresores secuenciales: la imagen se procesa por filas (o columnas) como si de una corriente se tratase. Gracias a esta forma de procesamiento de la entrada, los *buffers* de memoria utilizados son muy reducidos (típicamente del tamaño de una de las dimensiones de la imagen). Se trata de compresores conceptualmente simples bastante rápidos y simétricos.

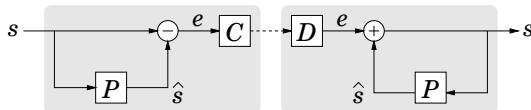


Figura 3.1: Codec sin pérdida de información propuesto por JPEG.

Tabla 3.1: Predictores utilizados por JPEG (ver Figura 3.2).

predictor	predicción
0	$\hat{s} \leftarrow 0$
1	$\hat{s} \leftarrow a$
2	$\hat{s} \leftarrow b$
3	$\hat{s} \leftarrow c$
4	$\hat{s} \leftarrow a + b - c$
5	$\hat{s} \leftarrow a + (b - c)/2$
6	$\hat{s} \leftarrow b + (a - c)/2$
7	$\hat{s} \leftarrow (a + b)/2$

3.1 JPEG

El estándar de compresión de imágenes JPEG (*Joint Photographic Experts Group*) propuesto por la CCITT (*International Telephone and Telegraph Consultative Committee*) en 1992 [55] contempla, entre otros, un método de compresión reversible de imágenes de tono continuo basado en la codificación entrópica del error de predicción.

La Figura 3.1 muestra la relación entre los *codecs* entrópicos y los predictores donde s denota un punto de la imagen a comprimir, \hat{s} la predicción realizada por el predictor P y e el error de predicción. C es el compresor entrópico específico y D el descompresor.

El codificador JPEG

La imagen es recorrida por filas, comenzando por la esquina superior izquierda. La codificación de un punto s se realiza así:

1. Generar una predicción \hat{s} usando alguno de los predictores expuestos en la Tabla 3.1.
2. Calcular el error de predicción $e \leftarrow s - \hat{s}$.
3. Codificar e entrópicamente.

Para realizar la predicción, JPEG usa un máximo de 3 puntos vecinos (ver Figura 3.2). La parte sombreada indica los puntos que ya han sido codificados y por lo tanto son también conocidos por el descompresor.

c	b
a	s

Figura 3.2: Contexto de predicción espacial usado por JPEG.

Los predictores se han diseñado bajo la idea de que los puntos vecinos tienden a parecerse entre si. Así, una predicción lineal a partir de estos puede ser adecuada para averiguar el valor aproximado del punto a codificar. Se está explotando de esta forma la correlación espacial a nivel local.

El error de predicción se calcula como se indica en el paso 2 y su codificación puede realizarse usando la codificación de Huffman o la codificación aritmética.¹ Para esto último es fundamental conocer que e estadísticamente representa una distribución de probabilidad de Laplace. Las Figuras 3.3 y 3.4 muestran los histogramas del error de predicción que resultan de aplicar los predictores 3 y 6 a las imágenes de prueba de 8 bpp y los predictores 4 y 7 a las imágenes de prueba de 16 bpp que son, respectivamente, el mejor y el peor de todos los predictores mostrados en la Tabla 3.1.²

Puede observarse que las imágenes residuo de 8 bpp son altamente modelizables por una Laplaciana. En el caso de las imágenes de 16 bpp, la gran cantidad de ruido Gaussiano desfigura la distribución de Laplace, aunque no de forma importante.

El descodificador JPEG

Conociendo el compresor, el descompresor consiste en:

1. Generar \hat{s} usando el mismo predictor que compresor.
2. Descodificar e usando el mismo código que el compresor.
3. Calcular el punto $s \leftarrow e + \hat{s}$.

A continuación pasamos a presentar con más detalle los codecs entrópicos usados en JPEG: una codificación de Huffman modificada y la codificación aritmética binaria.

El codificador de Huffman

El modelo probabilístico (utilizado en JPEG) siempre es estático y no tiene memoria (el contexto de predicción es de orden 0).

JPEG recomienda una tabla de códigos que en general da buenos resultados, aunque también es posible calcularlas de forma específica para cada imagen en cuyo caso debe ser transferida al descompresor.

¹En realidad no existe ninguna razón por la cual un código de Golomb-Rice no pudiera ser usado. Sin embargo, el estándar no los contempla.

²Empíricamente hablando, para el caso de estas imágenes.

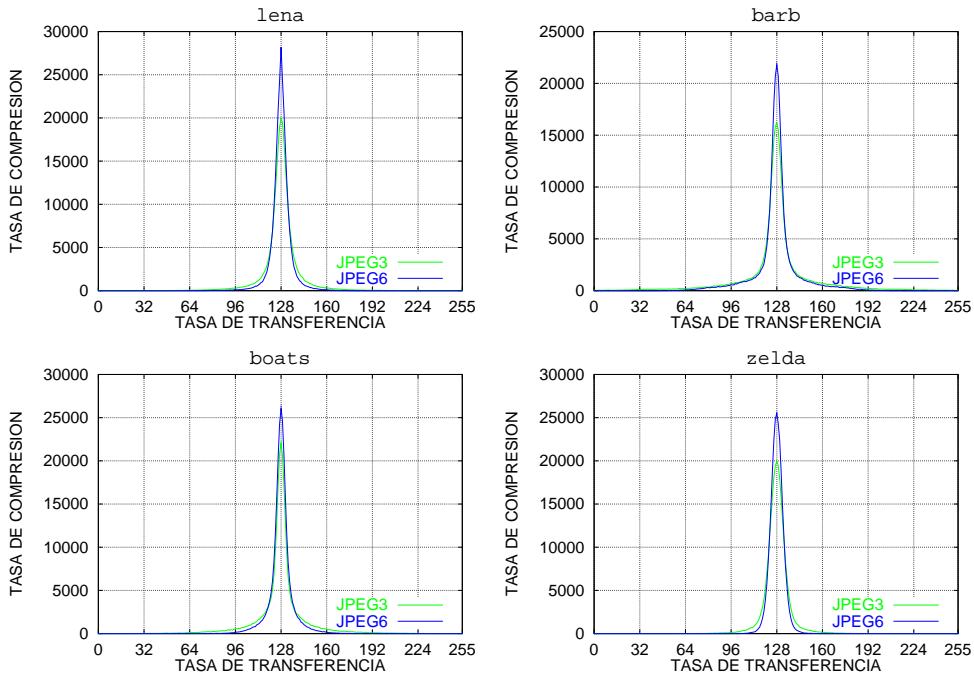


Figura 3.3: Histogramas de las imágenes residuo de 8 bits.

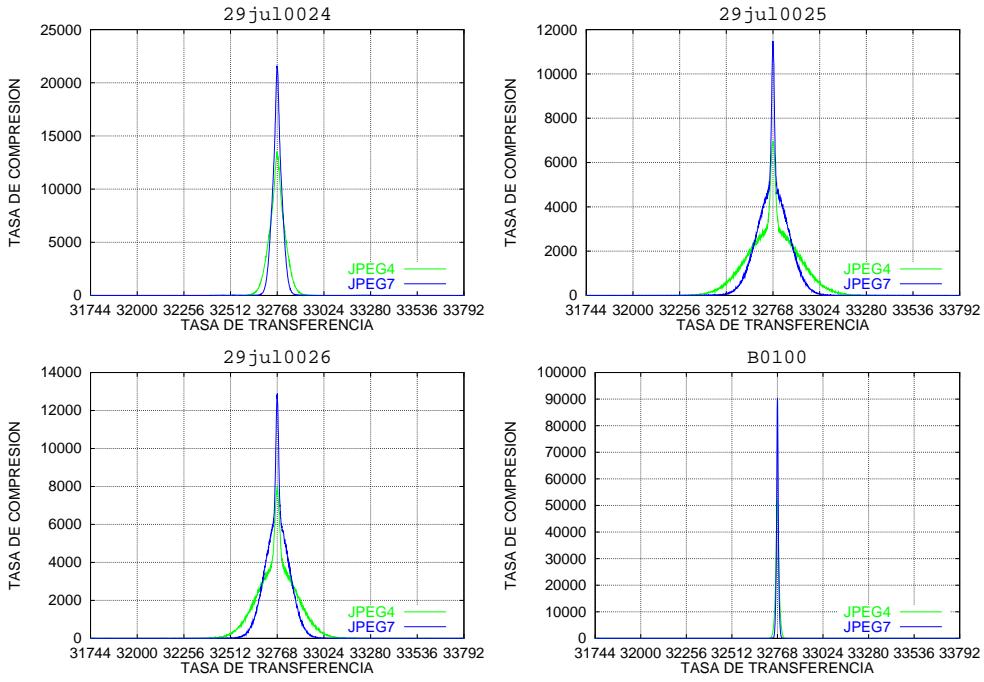


Figura 3.4: Histogramas de las imágenes residuo de 16 bits.

Tabla 3.2: Categorías de diferencia usadas en la codificación de Huffman (tabla H.2 de [55]).

<i>SSSS</i>	<i>DIFF</i>
0	0
1	-1,1
2	-3,-2,2,3
3	-7,⋯,-4,4,⋯,7
4	-15,⋯,-8,8,⋯,15
5	-31,⋯,-16,16,⋯,31
6	-63,⋯,-32,32,⋯,63
7	-127,⋯,-64,64,⋯,127
8	-255,⋯,-128,128,⋯,255
9	-511,⋯,-256,256,⋯,511
10	-1023,⋯,-512,512,⋯,1023
11	-2047,⋯,-1024,1024,⋯,2043
12	-4095,⋯,-2048,2048,⋯,4095
13	-8191,⋯,-4096,4096,⋯,8191
14	-16383,⋯,-8192,8192,⋯,16383
15	-32767,⋯,-16384,16384,⋯,32767
16	-32768

Tabla 3.3: Tabla de códigos de Huffman para luminancia DC (Tabla K.3 de [55]).

<i>SSSS</i>	Longitud	Código base
0	2	00
1	3	010
2	3	011
3	3	100
4	3	101
5	3	110
6	4	1110
7	5	11110
8	6	111110
9	7	1111110
10	8	11111110
11	9	111111110
12	10	1111111110
13	11	11111111110
14	12	111111111110
15	13	1111111111110
16	14	11111111111110

Sólo vamos a exponer la codificación de Huffman usando la tabla propuesta por JPEG, el uso de cualquier otra alternativa es semejante. La codificación del error de predicción e se realiza de la siguiente forma:

1. Buscar e en la segunda columna de la Tabla 3.2. En la primera columna encontramos la categoría de diferencia $SSSS$.
2. Codificar la categoría $SSSS$ según la Tabla 3.3.
3. Codificar la magnitud. Si $e > 0$, entonces:
 - (a) Codificar e usando un código binario de $SSSS$ bits. El bit más significativo de e va a ser siempre 1.
4. Si no:
 - (a) Codificar $e - 1$ usando un código binario de $SSSS$ bits. El bit más significativo de e va a ser siempre 0.

Por ejemplo, si $e = 5$, $SSSS = 3$. Emitimos 100 y a continuación 101 (5 en binario usando 3 bits). Si $e = -9$, $SSSS = 4$. Emitimos 101 seguido de los 4 bits menos significativos de -10 que son 0110.

El descodificador de Huffman

La descodificación de un error de predicción e se realiza según los siguientes pasos:

1. Descodificamos la categoría $SSSS$ utilizando cuantos bits de entrada sean necesarios. Utilizamos para ello la Tabla 3.3.
2. Ahora descodificamos la magnitud. Sea $e \leftarrow$ siguiente bit de entrada.
3. Si $e \neq 0$:
 - (a) $e \leftarrow e \times 2^{SSSS-1} +$ siguientes $SSSS - 1$ bits.
4. Si no:
 - (a) $e \leftarrow -1$ and $(e \times 2^{SSSS-1} +$ siguientes $SSSS - 1$ bits + 1). “and” es la operación lógica AND a nivel de bits.

Cuando el código es 100101, los 3 primeros bits indican que se trata de la categoría $SSSS = 3$. Leemos el siguiente bit de entrada sobre $e \leftarrow 1$ y como es distinto de cero, desplazamos e 2 bits a la izquierda. Leemos los dos siguientes bits de entrada y llegamos a que $e \leftarrow 101 = 5$. Cuando el código es 1010110, los 3 primeros bits indican que $SSSS = 4$. Leemos el siguiente bit de entrada y $e \leftarrow 0$, por lo que (usando 8 bits de precisión para representar los números en complemento a dos) $e \leftarrow 11111111$ and $(0110 + 1) = 11110111 = -9$ (los 4 bits más significativos de 0110 se suponen 1).

El codificador aritmético

JPEG usa un codificador aritmético binario junto con un modelo probabilístico adaptativo basado en el contexto espacial.

El modelo probabilístico usado con el codificador aritmético está basado en el contexto y puede ser adaptativo o no. El contexto se extrae de los errores de predicción obtenidos para el punto que está en la fila superior y en la columna anterior (puntos a y b en la Figura 3.2). Si el modelo es adaptativo, además, tras la codificación de cada símbolo binario se realiza una actualización (ver Sección 2.3.3)

El codificador efectúa básicamente los siguientes pasos:

1. Codificar si $e = 0$.
2. Codificar si $e < 0$.
3. Sea $k \leftarrow 0$.
4. Mientras $(|e| - 1) \geq 2^k$: (codificación de la categoría)
 - (a) Codificar un 1 indicando que $(|e| - 1) \geq 2^k$.
 - (b) $k \leftarrow k + 1$.
5. Mientras $k \geq 0$: (codificación de la magnitud)
 - (a) Codificar $(e \text{ and } 2^k)$, esto es, el k -ésimo bit de e .
 - (b) $k \leftarrow k - 1$.

El codificador aritmético es muy similar al de Huffman, aunque está basado en el contexto y es posible asignar menos de un bit a una decisión binaria (ver Sección 2.3.7). Esto provoca que en promedio, el codificador aritmético supere al de Huffman en un 10% aproximadamente.

El precio que se paga es que el compresor es bastante más lento ya que ahora la mayor parte del tiempo se emplea en la codificación entrópica. Por este motivo, la opción más utilizada, especialmente en las implementaciones software es la de Huffman [48]. Cuando el diseño se hace a nivel hardware, la codificación aritmética suele ser la más usada ya que el codificador es muy fácil de implementar [60, 68, 74, 76, 78] y no es necesariamente la opción más lenta.

El descodificador aritmético

La descodificación es el proceso inverso y parte del mismo contexto de predicción. Así, la descodificación del error de predicción e se realiza de esta forma:

1. Descodificar si $e = 0$.
2. Descodificar si $e < 0$.

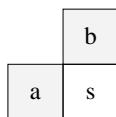


Figura 3.5: Contexto de predicción espacial usado por FELICS.

3. $k \leftarrow 0$.
4. Descodificación de la categoría. Mientras descodifiquemos un 1:
 - (a) $k \leftarrow k + 1$.
5. Descodificación de la magnitud. Sea $e \leftarrow 0$.
6. Mientras $k \geq 0$:
 - (a) $e \leftarrow e \times 2^k +$ el resultado (0 o 1) de la siguiente descodificación.
 - (b) $k \leftarrow k - 1$.

El tiempo que el descodificador aritmético gasta en una implementación *software* es ligeramente superior al del codificador debido a que las descodificaciones binarias son intrínsecamente algo más costosas (ver Sección 2.3.13). En una implementación *hardware* esto no es necesariamente cierto y además suele ser la opción más usada.

3.2 FELICS

El compresor de imágenes FELICS [43] es uno de los compresores de imágenes sin pérdida de información más sencillos, rápidos y ciertamente eficientes que se han diseñado (de hecho, su nombre es el acrónimo de *Fast, Efficient, Lossless Image Compression System*).

La imagen se comprime por filas y conceptualmente, FELICS es idéntico a JPEG en el sentido de que existe un predictor muy simple y un codificador entrópico muy rápido. El contexto espacial de predicción se muestra en la Figura 3.5.

FELICS utiliza un modelo probabilístico especial (ver Figura 3.6). El contexto de predicción se usa para indicar si s está entre el rango de valores contenido entre a y b . Si llamamos L al menor de a y b y a H el mayor, la probabilidad de que $L \leq s \leq H$ es bastante alta y además (al menos en FELICS así se presupone) casi uniforme. Sin embargo, si s está fuera de este rango, FELICS espera una exponencial decreciente (una SDPG) conforme nos alejamos de L y H . Para codificar la probabilidad de s cuando está dentro del intervalo, se usa un codificador binario ajustado (ver Sección 2.3.9) y para codificar la probabilidad fuera se usa un codificador de Rice (ver Sección 2.3.11).

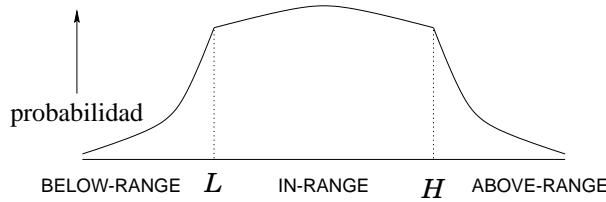


Figura 3.6: Modelo probabilístico usado en FELICS.

El codificador

La estrategia del codificador de FELICS consiste en emitir los códigos suficientes que informen al descodificador acerca de la situación de s respecto de a y b . Su algoritmo es el siguiente:

1. Sea $L \leftarrow \min\{a, b\}$ y sea $H \leftarrow \max\{a, b\}$.
2. Sea $\Delta \leftarrow H - L$.
3. Si $L \leq s \leq H$:
 - (a) Emitir un bit indicando *IN-RANGE*.
 - (b) Emitir un código binario ajustado codificando $s - L$. El tamaño del alfabeto es Δ .
4. Si no:
 - (a) Emitir un bit indicando *OUT-RANGE*.
 - i. Si $s < L$:
 - A. Emitir un bit indicando *BELOW-RANGE*.
 - B. Codificar $L - s - 1$ usando un código de Rice.
 - ii. Si no: ($s > H$)
 - A. Emitir un bit indicando *ABOVE-RANGE*.
 - B. Codificar $s - H - 1$ usando un código de Rice.
5. Estimar el parámetro k necesario para la codificación de Rice según el algoritmo presentado en la Sección 2.3.11.

Por ejemplo, supongamos que $s = 5$, $a = 4$ y $b = 7$. Entonces $L \leftarrow 4$, $H \leftarrow 7$ y $\Delta \leftarrow 3$. Como $3 \leq s \leq 7$, emitimos un bit (1 por ejemplo) indicando *IN-RANGE*. A continuación emitimos el código binario ajustado de la cantidad $s - L = 1$ que es 10 .³

³Si usamos la notación de la Sección 2.3.9, $n = 3$, por lo que $k = 1$ y $b = 1$. El símbolo a codificar s no cumple la condición de que $s < n - 2b = 3 - 2 = 1$, por lo que emitimos $1 + 3 - 2 \times 1 = 2$ usando un código binario de 2 bits: 10.

Supongamos ahora que $s = 3$, $a = 4$ y $b = 7$. $L \leftarrow 4$ y $H \leftarrow 7$. $\Delta \leftarrow 3$. Como $s < L$, emitimos un bit (que en este caso debe ser 0) indicando *OUT-RANGE*. A continuación emitimos otro bit (0 por ejemplo) indicando *BELOW-RANGE* y por ultimo codificamos $L - s - 1 = 4 - 3 - 1 = 0$ usando un código de Rice. Si se ha estimado por ejemplo que $k = 2$, se emiten los bits 000 (ver Tabla 2.2).

El uso de un código de Rice o de un código binario ajustado para expresar el resto del símbolo hace posible que FELICS pueda codificar imágenes de cualquier profundidad de color sin que los algoritmos de codificación y descodificación sean modificados. En el caso de usar la codificación de Huffman o incluso la aritmética, esto hubiera sido mucho más complejo.

El descodificador

A partir de la misma predicción realizada por el codificador y de la información proporcionada con los códigos, el descodificador FELICS recupera el valor original de s aplicando el siguiente algoritmo:

1. Sea $L \leftarrow \min\{a, b\}$ y sea $H \leftarrow \max\{a, b\}$.
2. Sea $\Delta \leftarrow H - L$.
3. Si s está *IN-RANGE*:
 - (a) Leer x descodificando el código binario ajustado sabiendo que el tamaño del alfabeto es Δ .
 - (b) $s \leftarrow L + s$.
4. Si no: (s está *OUT-RANGE*)
 - (a) Leer un bit (que indica si s está *BELOW-RANGE* o *ABOVE-RANGE*).
 - (b) Si s está *BELOW-RANGE*:
 - i. Descodificar x usando el descodificador de Rice.
 - ii. $s \leftarrow L - x - 1$.
 - (c) Si no: (s está *ABOVE-RANGE*)
 - i. Descodificar x usando el descodificador de Rice.
 - ii. $s \leftarrow H + x + 1$.
5. Estimar el parámetro k necesario para la codificación de Rice según el algoritmo presentado en la Sección 2.3.11.

Como ejemplo, descodificaremos los símbolos de los ejemplos anteriores. En primer lugar tenemos el código 110. El descodificador conoce el contexto de predicción y calcula que $L \leftarrow 4$, $H \leftarrow 7$ y $\Delta \leftarrow 3$. La llegada de un 1 indica que el $4 \leq s \leq 7$ y pasamos

a descodificar el código binario ajustado provocando $x \leftarrow 1$. Finalmente, $s \leftarrow L + x = 4 + 1 = 5$.

El siguiente código es el 00000. Por el contexto se conoce que $L \leftarrow 4$, $H \leftarrow 7$ y $\Delta \leftarrow 3$. El primer bit (0) indica que s no está *IN-RANGE*. El siguiente bit (0) indica que s está *BELLOW-RANGE*. Puesto que la estimación del parámetro k es idéntica a la del codificador, $k = 2$. Esto provoca que leamos 3 bits más de entrada para determinar que el número codificado es el 0, con lo que el símbolo descodificado es $s \leftarrow 4 - 0 - 1 = 3$.

FELICS y el codificador entrópico universal

Si observamos con detenimiento los pasos 3.a, 4.a.i.A y 4.a.ii.A del codificador, podemos apreciar que FELICS constituye un buen ejemplo de lo que puede ser un algoritmo de codificación que siga la filosofía de diseño introducida con el codificador entrópico universal (ver Sección 2.3.1). FELICS usa un bit de código para indicar si el símbolo está *IN-RANGE*. Los diseñadores comprobaron empíricamente que la probabilidad de que esta afirmación es cierta es del 50% aproximadamente.

Sin embargo, para eliminar el resto de la incertidumbre en la descodificación del símbolo, FELICS usa un código de longitud variable tanto si el símbolo está *IN-RANGE* como si no. Por esta razón, es posible encontrar en este compresor las dos filosofías de diseño: (1) codificar predicciones equiprobables usando códigos de longitud constante y (2) codificar predicciones no equiprobables usando códigos de longitud variable.

3.3 LOCO-I

LOCO-I [120] (*LOW COmplexity, context-based, lossless image compression algorithm*) es similar a FELICS en que usa un codificador de Rice como codec entrópico, pero a diferencia de éste, el modelo probabilístico está basado en el contexto.

Otra mejora importante introducida en LOCO-I (no usada en JPEG o FELICS) es que el predictor espacial trata de usar la información global de la imagen para mejorar la fase de predicción y generar una distribución de Laplace más angosta. La información global es una fuente importante de redundancia espacial ya que en las imágenes es común encontrar regiones distantes que son similares en textura o composición. Un predictor local es incapaz de extraer este tipo de redundancia.

En este sentido, existe mucha semejanza entre LOCO-I y PPM (ver Sección 2.3.14). Sin embargo, LOCO-I utiliza un proceso de cuantificación de contextos espaciales que minimiza el problema acarreado por el número excesivo de contextos generados en PPM cuando comprime imágenes de tono continuo.⁴

⁴Recuérdese que el principal inconveniente de usar un compresor de texto en compresión de imágenes de tono continuo es que existen tantos patrones diferentes que ningún contexto de orden superior es capaz explotar la correlación espacial. Desde el punto de vista probabilístico lo que ocurre es que el problema de la frecuencia cero es patente durante todo el tiempo que dura la compresión. Este problema dejaría de existir si la imagen fuese muy grande, pero entonces la cantidad de memoria consumida por el modelo de

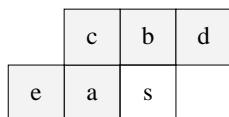


Figura 3.7: Contexto de predicción espacial usado por LOCO-I.

Los autores de LOCO-I se percataron de que había mucha redundancia que podía ser explotada teniendo en cuenta el contexto espacial, a un coste muy bajo en comparación con el uso de codificadores más complejos como es la codificación aritmética. Por este motivo, decidieron usar la codificación de Rice y diseñar así un compresor y descompresor muy rápidos. Todo esto ha provocado que LOCO-I haya sido adoptado casi sin modificación alguna como un nuevo estándar para la compresión de baja complejidad (en términos de memoria y CPU) y sin pérdida de información de imágenes llamado JPEG-LS (JPEG-Lossless) [56].

El codificador

La imagen se recorre por filas comenzando por la esquina superior derecha. El algoritmo del compresor es el siguiente:

1. Inicialización de los contextos de predicción:

- Sea Q el contexto actual. LOCO-I considera un máximo de 1094 contextos distintos.
- Sea $N[Q]$ el número de ocurrencias de cada contexto. Inicialmente $N[Q] \leftarrow 0 \forall Q$.
- Sea $B[Q]$ el error de predicción acumulado en cada contexto. Inicialmente $B[Q] \leftarrow 0 \forall Q$.
- Sea $A[Q]$ la suma de los valores absolutos de los errores de predicción para cada contexto. Inicialmente $A[Q] \leftarrow 0 \forall Q$.
- Sea $C[Q]$ los valores de cancelación del *bias*. El *bias* es un valor que sumado a la predicción espacial provoca que su media sea 0. Inicialmente $C[Q] \leftarrow 0 \forall Q$.

2. Determinación del contexto de predicción Q :

- Calcular el gradiente local. Para ello se efectúan las 4 diferencias (ver Figura 3.7):

$$\begin{aligned} g_1 &\leftarrow d - a \\ g_2 &\leftarrow a - c \\ g_3 &\leftarrow c - b \\ g_4 &\leftarrow b - e \end{aligned}$$

Markov es enorme.

(b) Cuantificar los gradientes según:

$$q_i \leftarrow \begin{cases} 0 & \text{si } g_i = 0 \\ 1 & \text{si } 1 \leq |g_i| \leq 2 \\ 2 & \text{si } 3 \leq |g_i| \leq 6 \\ 3 & \text{si } 7 \leq |g_i| \leq 14 \\ 4 & \text{en otro caso} \end{cases}$$

para $i = 1, 2, 3$ y

$$q_4 \leftarrow \begin{cases} 0 & \text{si } |g_4| < 0 \\ 1 & \text{si } 5 \leq g_4 \\ 2 & \text{en otro caso.} \end{cases}$$

3. Cálculo del error de predicción $M(e)$:

(a) Construir la predicción inicial:

$$\hat{s} \leftarrow \begin{cases} \min(a, b) & \text{si } c \geq \max(a, b) \\ \max(a, b) & \text{si } c \leq \min(a, b) \\ a + b - c & \text{en otro caso.} \end{cases}$$

(b) Cancelar el bias:

$$\hat{s} \leftarrow \begin{cases} \hat{s} + C[Q] & \text{si } g_1 > 0 \\ \hat{s} - C[Q] & \text{en otro caso.} \end{cases}$$

(c) Calcular el error de predicción:

$$e \leftarrow (s - \hat{s}) \bmod \beta,$$

donde β es el número de bits por punto. Esto provoca que el error de predicción sea proyectado desde el intervalo $[-\alpha + 1, \alpha - 1]$ al intervalo $[-\alpha/2, \alpha/2 - 1]$ donde $\alpha = 2^\beta$ es el tamaño del alfabeto.

(d) Barajar los errores de predicción negativos y positivos generando una SDPG. Esto se realiza según el siguiente mapeo:

$$M(e) \leftarrow \begin{cases} 2e & \text{si } e \geq 0 \\ 2|e| - 1 & \text{en otro caso.} \end{cases}$$

Tras dicho mapeo, los errores de predicción se ordenan según: $0, -1, +1, -2, +2, \dots, 2^\beta - 1$.

4. Codificación entrópica de $M(e)$ en el contexto Q :

(a) Se emite un código de Rice que codifica el símbolo $M(e)$ para $k = \lceil \log_2(A[Q]) \rceil$.

5. **Actualización del contexto Q :**

- (a) $B[Q] \leftarrow B[Q] + e.$
- (b) $A[Q] \leftarrow A[Q] + |e|.$
- (c) Si $N[Q] = \text{RESET}$, entonces: (donde $64 \leq \text{RESET} \leq 256$)
 - i. $A[Q] \leftarrow A[Q]/2.$
 - ii. $B[Q] \leftarrow B[Q]/2.$
 - iii. $H[Q] \leftarrow N[Q]/2.$
- (d) $N[Q] \leftarrow N[Q] + 1.$
- (e) La actualización del valor para la cancelación del *bias* es algo más compleja.
Si el error de predicción inicial en el contexto Q no tiene media 0, el nivel de compresión decae severamente porque las medias de la distribución de Laplace real y la modelizada no coinciden. Para evitar esto, $C[Q]$ almacena un valor proporcional a $B[Q]/N[Q]$ que es sumado a la predicción inicial para cancelar el *bias*.

Además, $C[Q]$ es el encargado de solucionar otro problema derivado del barajamiento producido por $M(e)$ debido al cual, se tiende a asignar un código más corto al error negativo que al respectivo error positivo.

Con todo esto, el algoritmo para la actualización de $C[Q]$ es el siguiente [56]:

- i. Si $B[Q] \leq -N[Q]$, entonces:
 - A. $B[Q] \leftarrow B[Q] + N[Q].$
 - B. Si $C[Q] > -128$, entonces:
 - $C[Q] \leftarrow C[Q] - 1.$
 - C. Si $B[Q] \leq -N[Q]$, entonces:
 - $B[Q] \leftarrow -N[Q] + 1.$
- ii. Si no:
 - A. Si $B[Q] > 0$, entonces:
 - $B[Q] \leftarrow B[Q] - N[Q].$
 - Si $C[Q] < 127$, entonces:
 - ★ $C[Q] \leftarrow C[Q] + 1.$
 - Si $B[Q] > 0$, entonces:
 - ★ $B[Q] \leftarrow 0.$

El descodificador

El descodificador es muy simétrico y recupera los símbolos s usando el siguiente algoritmo:

1. **Inicialización de los contextos de predicción** (ídem al paso 1 del compresor).
2. **Determinación del contexto Q** : (ídem al paso 2 del compresor).

3. **Descodificación de $M(e)$:** usando un descodificador de Rice con $k = \lceil \log_2(A[Q]) \rceil$.

4. **Determinación de s :**

- (a) Computar la predicción inicial \hat{s} como en el paso 3.a del compresor.
- (b) Añadir el *bias* sobre \hat{s} :

$$\hat{s} \leftarrow \begin{cases} \hat{s} - C[Q] & \text{si } g_1 > 0 \\ \hat{s} + C[Q] & \text{en otro caso.} \end{cases}$$

- (c) Calcular el mapeo inverso generando la distribución de Laplace original:

$$e \leftarrow M^{-1}(M(e)) = \begin{cases} -M(e)/2 - 1 & \text{si } M(e) \text{ es impar} \\ M(e)/2 & \text{en otro caso.} \end{cases}$$

- (d) Calcular el símbolo s como:

$$s \leftarrow (e + \hat{s}) \bmod \beta.$$

5. **Actualización de Q :** (ídem al paso 5 del compresor).

El modo *run-mode*

LOCO-I usa un codificador de Rice que puede ser muy redundante cuando se generan distribuciones de probabilidad muy angostas, donde la probabilidad del error más frecuente (el 0) es mayor que 0.5 (ver Sección 2.3.6). Para evitar esto, LOCO-I tiene un modo especial de funcionamiento llamado *run-mode* en el que entra cuando $a = b = c = d$ (el contexto es constante) y provoca que $[q_1, q_2, q_3] \leftarrow [0, 0, 0]$. Mientras se permanece en este modo, no se emite ningún código.

El modo *run-mode* es abandonado si se encuentra un símbolo $s \neq a$ o si se ha alcanzado el final de una línea. Esto provoca que el número de símbolos r procesados en *run-mode* (la longitud de la serie) sea codificado. En este caso, LOCO-I emplea una tabla precalculada para encontrar el parámetro k .⁵

3.4 CALIC

CALIC [130, 131] (*Context-Based, Adaptive, Lossless Image Coding*) es muy similar a LOCO-I en todos los aspectos excepto en que la complejidad del codec entrópico utilizado es superior. Se trata de otro compresor de imágenes secuencial que realiza una predicción espacial basada en el contexto. Explota tanto la redundancia local como la global. El predictor es adaptativo pues aprende de los errores cometidos en predicciones anteriores.

⁵Ver apartado A.7.1.2 de [56].

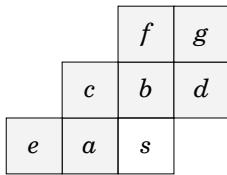


Figura 3.8: Contexto de predicción espacial usado por CALIC.

El modelo probabilístico también está basado en el contexto y es adaptativo, independientemente de si el codificador usado es el aritmético o el de Huffman. Además, CALIC igual que FELICS y LOCO-I, está también preparado para codificar imágenes de hasta 16 bits/punto.

El resultado es que CALIC es uno de los compresores de imágenes que mejores tasas de compresión alcanzan. Como es de esperar, las tasas de transferencia conseguidas son inferiores a las de LOCO-I, pero en situaciones donde esto no sea importante, como puede ser la transferencia de imágenes o el almacenamiento en dispositivos lentos, CALIC puede ser preferible.

El codificador

Como en todos los compresores secuenciales presentados, la imagen se recorre por filas comenzando por la esquina superior derecha. El algoritmo de compresión se describe de la siguiente forma:

1. Inicialización de los contextos de predicción y de probabilidades:

- Sea K el contexto de predicción actual. CALIC contempla hasta 576 contextos de predicción diferentes.
- Sea $N[K]$ el número de ocurrencias en cada contexto K . Inicialmente $N[K] \leftarrow 0 \forall K$.
- Sea $S[K]$ el error de predicción acumulado en cada contexto K . Inicialmente $S[K] \leftarrow 0 \forall K$.
- Sea $Q(\Delta)$ el contexto probabilístico actual. CALIC usa 8 contextos probabilísticos diferentes.
- Sea $N_{\epsilon_1}[Q(\Delta)]$ el número de ocurrencias de cada contexto $Q(\Delta)$. Inicialmente $N_{\epsilon_1}[Q(\Delta)] \leftarrow 0 \forall Q(\Delta)$.
- Sea $S_{\epsilon_1}[Q(\Delta)]$ el error de predicción acumulado en cada contexto $Q(\Delta)$. Inicialmente $S_{\epsilon_1}[Q(\Delta)] \leftarrow 0 \forall Q(\Delta)$.

2. Determinación del contexto de predicción K , del contexto probabilístico $Q(\Delta)$ y de la predicción \tilde{s} :

(a) Calcular el gradiente local horizontal mediante

$$d_h \leftarrow |a - e| + |b - c| + |d - b|$$

y vertical mediante

$$d_v \leftarrow |a - c| + |b - f| + |d - g|.$$

(b) Cuantificar el gradiente:

i. Sea

$$\Delta \leftarrow d_h + d_v + 2|\epsilon_a|,$$

donde ϵ_a es el error de predicción cometido para el punto a (ver Figura 3.8) sin tener en cuenta la cancelación del *bias*. Por lo tanto, $\epsilon_a \leftarrow a - \hat{a}$ donde \hat{a} es la predicción inicial de la iteración anterior.

(c) Cuantificar Δ en 8 niveles según:

$$Q(\Delta) \leftarrow \begin{cases} 0 & \text{si } \Delta \leq 5 \\ 1 & \text{si } 6 \leq \Delta \leq 15 \\ 2 & \text{si } 16 \leq \Delta \leq 25 \\ 3 & \text{si } 26 \leq \Delta \leq 42 \\ 4 & \text{si } 43 \leq \Delta \leq 60 \\ 5 & \text{si } 61 \leq \Delta \leq 85 \\ 6 & \text{si } 86 \leq \Delta \leq 140 \\ 7 & \text{en otro caso.} \end{cases}$$

(d) Construir la predicción inicial para s mediante:

$$\hat{s} \leftarrow \begin{cases} a & \text{si } d_v - d_h > 80 \quad /* \text{ arista horizontal fuerte */} \\ b & \text{si } d_v - d_h < -80 \quad /* \text{ arista vertical fuerte */} \\ \frac{\frac{a+b}{2} + \frac{d-c}{4} + a}{2} & \text{si } d_v - d_h > 32 \quad /* \text{ arista horizontal */} \\ \frac{\frac{a+b}{2} + \frac{d-c}{4} + b}{2} & \text{si } d_v - d_h < -32 \quad /* \text{ arista vertical */} \\ \frac{3\left(\frac{a+b}{2} + \frac{d-c}{4}\right) + a}{4} & \text{si } d_v - d_h > 8 \quad /* \text{ arista horizontal débil */} \\ \frac{3\left(\frac{a+b}{2} + \frac{d-c}{4}\right) + b}{4} & \text{si } d_v - d_h < -8 \quad /* \text{ arista vertical débil */} \end{cases}$$

(e) Calcular el patrón de textura local $B = \{b_0, \dots, b_7\}$ donde

$$b_i \leftarrow \begin{cases} 0 & \text{si } x_i \geq \hat{s} \\ 1 & \text{en otro caso} \end{cases}$$

y donde $x_0 = b$, $x_1 = a$, $x_2 = c$, $x_3 = d$, $x_4 = f$, $x_5 = e$, $x_6 = 2b - f$ y $x_7 = 2a - e$.

- (f) Generar el contexto de predicción compuesto $k = \{\lceil Q(\Delta)/2 \rceil, B\}$.
 (g) Determinar el valor definitivo de la predicción cancelando el *bias*:

$$\tilde{s} \leftarrow \hat{s} + \left\lceil \frac{S[K]}{N[K]} \right\rceil.$$

3. Calcular el error de predicción $M(\epsilon)$:

- (a) Calcular el error de predicción

$$\epsilon = (s - \tilde{s}) \bmod \beta,$$

donde β es el número de bits/punto.

- (b) Hacer una predicción sobre el signo de ϵ haciendo que

$$\epsilon = -\epsilon \text{ si } \lceil S[K]/N[K] \rceil < 0,$$

puesto que el signo del error de predicción suele estar espacialmente correlacionado.

- (c) Barajar los errores de predicción negativos y positivos usando el siguiente mapeo:

$$M(\epsilon) \leftarrow \begin{cases} 2\epsilon - 1 & \text{si } \epsilon > 0 \\ 2|\epsilon| & \text{en otro caso.} \end{cases}$$

Notar que tras el barajamiento, los errores de predicción se ordenan según: $0, +1, -1, +2, -2, \dots, 2^\beta - 1$.

4. Codificación entrópica de $M(\epsilon)$ en el contexto $Q(\Delta)$:

CALIC puede utilizar un codificador aritmético o un codificador de Huffman para codificar el error de predicción $M(\epsilon)$, siendo la primera opción la que mejores tasas de compresión obtiene (alrededor de un 3.5% mejores). Puesto que se trata de codificar una SDPG, de forma independiente al codificador escogido, CALIC incrementa la tasa de compresión truncando las colas de los 8 histogramas y así reducir en lo posible el problema de la frecuencia cero (ver Sección 2.3.5). Empíricamente se puede demostrar que cuando $Q(\Delta) = 0$, el 99% de los errores de predicción pertenecen al intervalo $[-8, 8]$ lo que significa que errores muy cercanos a las colas del histograma son muy poco probables.

Como codificador aritmético, CALIC usa una versión que manipula 256 símbolos y el modelo probabilístico asociado usa 14 bits para almacenar los pesos de los símbolos, lo que implica que la resolución con la que el modelo es capaz de expresar las probabilidades es $1/2^{14} = 1/16384 = 0.000061$.

El símbolo menos probable es el 128 y suponer que va a ocurrir al menos una vez cada 16384 símbolos le asigna realmente una probabilidad demasiado alta (aunque esto depende, por supuesto, de la imagen codificada).

Para estimar mejor las probabilidades de los símbolos raros, CALIC utiliza el símbolo especial **ESC**, de una forma muy similar a como lo hace PPM (ver Sección 2.3.14). Cada histograma truncado codifica su **ESC** con el código que está en la cola del histograma. Los autores han calculado que los mejores símbolos de ESCape son $\text{ESC}_0 = 18$ (el error 18 representa al **ESC** cuando $Q(\Delta) = 0$), $\text{ESC}_1 = 26$, $\text{ESC}_2 = 34$, $\text{ESC}_3 = 50$, $\text{ESC}_4 = 66$, $\text{ESC}_5 = 82$, $\text{ESC}_6 = 114$ y $\text{ESC}_7 = 256$. De esta forma, cuando el símbolo $M(\epsilon)$ no puede ser codificado en $Q(\Delta)$, se codifica $\text{ESC}_{Q(\Delta)}$ y se intenta para $Q(\Delta) + 1$. Este proceso se repite hasta que el modelo probabilístico contempla a $M(\epsilon)$ y cuando esto ocurre, se codifica $M(\epsilon) - (\text{ESC}_{Q(\Delta)} - 1)$. Por ejemplo, si $Q(\Delta) = 0$ y $M(\epsilon) = 30$, codificamos 18, 26 y 30 – (26 – 1) = 5.

Por último, en la fase de codificación entrópica, CALIC también resuelve el problema de representación de errores de predicción muy grandes, superiores a 255, de la siguiente forma. Puesto que el codificador aritmético (o el codificador de Huffman) sólo contempla 256 símbolos diferentes, hay que emplear algún método que permita manipular errores tan grandes como $2^{16} - 1$. Esto se realiza usando una técnica llamada desplazamiento de bits dinámico y que consiste en descomponer $M(\epsilon)$ en dos bloques de bits ϵ_1 y ϵ_2 . El primer bloque contiene los bits más significativos y el segundo los menos significativos. Sea k el número de bits que son considerados como los menos significativos. Primero se codifica ϵ_1 y a continuación ϵ_2 . Sin embargo, ahora existe la posibilidad de emitir ϵ_2 directamente porque la codificación entrópica suele ser inútil debido a la gran cantidad de ruido que presenta en la mayoría de las imágenes. Para encontrar k debe tenerse en cuenta que ϵ_1 debe ser menor que 256 (o de lo contrario no podría ser manipulado por el codec aritmético). Bajo esta restricción, se busca el valor más pequeño de k que verifica que $S_{\epsilon_1}[Q(\Delta)]/N_{\epsilon_1}[Q(\Delta)] < \lceil \text{ESC}_{Q(\Delta)}/2 \rceil$.

5. Actualización de los contextos de predicción y probabilísticos:

- (a) Si $N[K] > 127$, entonces:
 - i. $N[K] \leftarrow 64$.
 - ii. $S[K] \leftarrow S[K]/2$.
 - iii. $N_{\epsilon_1}[Q(\Delta)] \leftarrow 64$.
 - iv. $S_{\epsilon_1}[Q(\Delta)] \leftarrow S_{\epsilon_1}[Q(\Delta)]/2$.
- (b) $N[K] \leftarrow N[K] + 1$.
- (c) $N_{\epsilon_1}[Q(\Delta)] \leftarrow N_{\epsilon_1}[Q(\Delta)] + 1$.
- (d) Actualizar $M(\epsilon)$ en $Q(\Delta)$, incrementando su probabilidad.

El descodificador

El descodificador de CALIC es (como suele ser habitual) muy simétrico y se parece mucho al codificador:

1. **Inicialización de los contextos de predicción y probabilísticos:** ídem al paso 1 del compresor.
2. **Determinar el contexto de predicción K , el probabilístico $Q(\Delta)$ y la predicción \tilde{s} :** ídem al paso 2 del compresor.
3. **Descodificar $M(\epsilon)$ en el contexto $Q(\Delta)$.**
4. **Cálculo de s :**

- (a) Calcular el mapeo inverso recuperando así la distribución de Laplace original:

$$\epsilon \leftarrow M^{-1}(M(\epsilon)) = \begin{cases} M(\epsilon)/2 + 1 & \text{si } M(\epsilon) \text{ es impar} \\ -M(\epsilon)/2 & \text{en otro caso.} \end{cases}$$

- (b) Deshacer la predicción sobre el signo de ϵ : si $\lceil S[K]/N[K] \rceil < 0$ entonces $\epsilon = -\epsilon$.
- (c) Calcular el símbolo s como:

$$s \leftarrow (\epsilon + \tilde{s}) \bmod \beta.$$

5. **Actualizar el contexto de predicción y probabilístico.**

El modo binario

CALIC usa un modo especial llamado “binario” para codificar las imágenes o secciones de imágenes que sólo tienen dos tonos. Este modo se dispara automáticamente si el número de tonos encontrados en los puntos a, b, c, d, e y f (g no se utiliza) es dos a lo sumo. Cuando esto ocurre, se usa un codificador aritmético binario (muy parecido al que se usa en JBIG [53]) basado en el contexto que manipula 3 símbolos. Los dos primeros se usan para codificar los dos tonos y el tercero para indicar la salida del modo binario hacia el modo normal de funcionamiento llamado “continuo”.

Para determinar el contexto de predicción probabilístico se cuantifica el patrón formado por los 6 puntos que conforman el contexto. Sea este patrón de textura $B = b_5b_4b_3b_2b_1b_0$ calculado como

$$b_i \leftarrow \begin{cases} 0 & \text{si } x_i = a \\ 1 & \text{en otro caso,} \end{cases}$$

donde $x_0 = b, x_1 = a, x_2 = c, x_3 = d, x_4 = e$ y $x_5 = f$. Como $b_1 = 0$, este no forma parte realmente del contexto y existen por lo tanto $2^5 = 32$ contextos diferentes.

c	b	d
a	s	

Figura 3.9: Contexto de predicción espacial usado por BPN.

3.5 BPN+aritha

En compresión secuencial de imágenes el elemento que más peso tiene en el nivel de compresión final es el predictor espacial. Todos los modelos espaciales están basados en el contexto que está formado por aquellos puntos que rodean al punto a predecir.

Desarrollar modelos estáticos que sean buenos para todas las imágenes es prácticamente imposible. Sin embargo, puesto que son muy rápidos se han usado en el estándar JPEG.

La diferencia de rendimiento entre modelos se debe, por tanto, al nivel de adaptación que consiguen. Los modelos usados en LOCO-I y CALIC son totalmente adaptativos y además aprovechan la información local y la global para aprender de la imagen que está siendo codificada.

Tratando de seguir esta filosofía de diseño de predictores, en [96] se implementa y evalúa un predictor totalmente adaptativo. La idea consiste en usar una red neuronal (con 2 capas de neuronas ocultas) como predictor (no lineal) totalmente adaptativo llamado BPN (*Back Propagation Network*) que aprende de los errores de predicción cometidos usando el algoritmo de retropropagación del error de predicción [62, 102].

El objetivo del experimento es averiguar qué cantidad de información es capaz de asimilar la red cuando parte de un conocimiento nulo acerca de la forma en que se distribuyen los puntos dentro de una imagen. Por esta razón, la estructura del comprresor y del descomprresor es muy simple. El modelo probabilístico es de orden cero⁶ y por tanto no existe modelización de la distribución estadística de los errores de predicción. Sin embargo, se usa un modelo adaptativo. La adaptación es necesaria porque la distribución de errores de predicción es cambiante. Inicialmente los errores son muy grandes porque la red no sabe absolutamente nada acerca de la imagen. Esta situación cambia cuando la red aprende.

El contexto de predicción usado se muestra en la Figura 3.9 y está formado por los 5 puntos que rodean al punto s a predecir.

El comprresor

Como siempre, la imagen se recorre por filas, de izquierda a derecha. La codificación del punto s consiste en:

1. Construir una predicción \hat{s} usando la BPN y los puntos de alrededor.
2. Calcular el error de predicción $e \leftarrow s - \hat{s}$.

⁶Se supone que no existe correlación estadística ya que ha sido eliminada por el predictor.

3. Codificar entrópicamente e . Para la evaluación se ha usado un codificador aritmético multisímbolo adaptativo y de orden cero.
4. Hacer que la BPN aprenda en función de e usando el algoritmo de retropropagación del error. Con este paso los pesos de las conexiones sinápticas de todas las neuronas de la red son recalculados.

La imagen residuo sigue (como todas las obtenidas en los algoritmos presentados en este capítulo) una distribución de Laplace. En nuestro caso, el codificador aritmético adaptativo de orden 0 (*aritha* estudiado en la Sección 2.4) ha sido utilizado ya que es el más adecuado cuando no hay información de contexto que permita la exclusión de símbolos ni la estimación local de sus probabilidades.

El descompresor

Básicamente consiste en deshacer los pasos del compresor:

1. Construir \hat{s} de la misma forma que en el paso 1 del compresor.
2. Descodificar e .
3. Calcular el punto $s \leftarrow e + \hat{s}$.
4. Hacer que la BPN aprenda en función de e de la misma forma que en el paso 4 del compresor.

Consideraciones sobre el uso de la red neuronal

La forma en que la red ha sido utilizada es adecuada si se desea alcanzar el máximo nivel de compresión posible partiendo de un estado de conocimiento cero, puesto que la red no deja de entrenarse al tiempo que se usa como predictor.

Sin embargo, la reconfiguración dinámica de los pesos de las neuronas es una tarea demasiado costosa para construir en un compresor realmente útil. En nuestros experimentos, la tasa de aprendizaje (α) y el momento de la red (η) son muy altos para que la red aprenda lo antes posible. Además, para minimizar los tiempos, la red es bastante pequeña y sólo contempla 4 neuronas en la primera capa oculta y 8 en la segunda (ver Figura 3.10).

Estos factores podrían provocar que la red cayera en un mínimo local y consecuentemente, que la entropía de las imágenes no fuera la más baja posible. Nuestros experimentos indican lo contrario. La entropía sólo posee un mínimo, aunque es poco profundo. El único riesgo que corremos por tanto es oscilar alrededor del mínimo.

Evaluación del predictor BPN

Para realizar una comparación justa del predictor BPN con alguno de los predictores estudiados en este capítulo no podemos usar las tasas de compresión porque también se

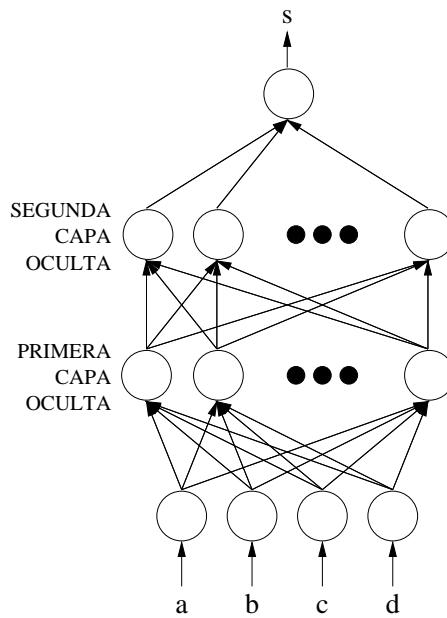


Figura 3.10: Arquitectura de la BPN.

estaría teniendo en cuenta la modelización probabilística que se hace de los errores de predicción (factor que no tiene en cuenta BPN+aritha).

Por este motivo, se han implementado los predictores usados en JPEG y en CALIC, y se han medido las entropías del error de predicción generado por dichos predictores. Estos valores junto con las entropías del error de predicción arrojados por el predictor BPN se muestran en la Tabla 3.4.

Teniendo en cuenta que la red no se encuentra entrenada y que sólo se realiza una única pasada, en general el rendimiento de nuestro predictor es razonablemente bueno. Es mejor que el mejor de los 7 predictores que JPEG propone (excepto para las imágenes “lena”, “B0100” y “29jul24”) y es mejor que CALIC en las imágenes “29jul25” y “29jul26”.

Tabla 3.4: Entropías del error de predicción de los predictores usados en JPEG, CALIC y BPN.

	imágenes de 8 bpp				imágenes de 16 bpp			
	lena	bárbara	boats	zelda	B0100	29jul24	29jul25	29jul26
JPEG6	4.56	5.39	4.75	4.22	5.06	7.17	8.75	8.39
JPEG7	4.61	5.51	4.86	4.24	4.74	6.85	8.43	8.07
BPN	4.61	5.27	4.70	4.20	7.07	7.68	8.00	7.98
CALIC	4.45	5.13	4.54	4.05	4.67	6.76	8.37	8.01

3.6 Evaluación

Pasamos ahora a presentar el rendimiento de los compresores de imágenes descritos en este capítulo. En primer lugar realizaremos una comparación entre ellos y después se discutirán las ganancias obtenidas en comparación con los algoritmos de compresión de texto presentados en el capítulo anterior.

Se han medido las tasas de transferencia⁷ y la tasa de compresión de todos los algoritmos presentes en este capítulo. A continuación comentamos los resultados que se han obtenido y que aparecen en las Tablas 3.5, 3.6, 3.7 y 3.8.

JPEG La implementación usada para JPEG [48] sólo permite la compresión de imágenes de 8 bits y el uso de la codificación de Huffman. El algoritmo selecciona para cada imagen el mejor de los 8 predictores que contempla el estándar. La tabla de Huffman usada es la estándar.

Los niveles de compresión que se obtienen son los más bajos de todos los compresores de imágenes probados, aunque las diferencias no son notables. El codec es bastante asimétrico. El compresor es rápido comparado con el resto de compresores y el descompresor es el más rápido de todos.

FELICS La implementación usada para FELICS [127] (al igual que la de JPEG) sólo permite la compresión de imágenes de 8 bpp. FELICS mejora a JPEG en un 2% aproximadamente en tasa de compresión. El codec es bastante simétrico y muy rápido.

BPN+aritha En tercer lugar aparece nuestra sencilla implementación de un compresor al estilo JPEG [96], que gracias a la adaptatividad del predictor consigue mejorar incluso las tasas de compresión de FELICS. Sin embargo, las tasas de transferencia alcanzadas son pobres pues como ya hemos comentado, incluso usando una red neuronal relativamente pequeña (4+8+1 neuronas), el algoritmo de retropropagación del error es bastante más pesado.

Es interesante comprobar cómo BPN+arith alcanza la tasa de compresión más alta para las imágenes “29jul25” y “29jul26”. Si miramos en el Apéndice A comprobaremos que estas imágenes son las que más entropía presentan de todas. Esto significa que son las imágenes que menores tasas de compresión va a alcanzar en general. El que BPN+arith haga un buen trabajo aquí se debe sin duda a que la red es capaz de detectar la correlación espacial, por compleja que esta sea. Este es un factor interesante en la compresión de imágenes astronómicas que no pueden ser realmente descritas como una señal bidimensional de tono continuo.

LOCO-I La implementación de LOCO-I [118] ha sido escrita y donada al *public domain* por HP con propósitos de evaluación. El compresor mejora a FELICS en un 5% y

⁷Los tiempos han sido tomados en la máquina descrita en la Sección 2.4

Tabla 3.5: Tasas de compresión (multiplicadas por 1000) de los algoritmos de compresión de imágenes secuenciales probados para las imágenes de 8 bpp.

método	imágenes				media
	lena	bárbara	boats	zelda	
JPEG	413	314	388	452	392
FELICS	426	333	417	464	410
BPN+arith	424	341	412	475	413
LOCO-I	470	409	469	500	462
CALICh	478	429	471	504	471
CALICA	487	444	482	518	483

las tasas de transferencia mejoran significativamente (en las imágenes de 8 bpp). El codec es muy simétrico y ocupa un lugar privilegiado en el espacio del rendimiento. La implementación probada también permite comprimir imágenes de 16 bpp. En este caso, LOCO-I arroja una tasa de compresión ligeramente inferior a la de CALIC (el mejor de todos), pero las tasas de transferencia siguen siendo excelentes.

CALIC Una implementación para CALIC puede encontrarse en [132]. Existe una versión (acabada en “h”) que usa un codificador de Huffman y otra versión (acabada en “a”) que usa la codificación aritmética. Ambas consiguen los mejores niveles de compresión de todos los algoritmos probados, resultando en este aspecto (y en el caso de usar la codificación aritmética) para las imágenes de 8 bpp hasta un 2.1% mejor que LOCO-I y en el caso de las imágenes de 16 bpp hasta un 2.5%. Sin embargo, las tasas de transferencia decaen notablemente. Por ejemplo, para las imágenes de 16 bpp, CALIC puede llegar a ser hasta un orden de magnitud más lento que LOCO-I.

En las Figuras 3.11 y 3.12 es posible ver el rendimiento de los algoritmos de compresión de imágenes comparado con el de los algoritmos de compresión de texto estudiados en el capítulo anterior. En el caso de las imágenes de 8 bpp, las tasas de compresión mejoran entre un 15% y un 20% aproximadamente sin que las tasas de transferencia se vean negativamente afectadas. En el caso de las imágenes de 16 bpp, sólo LOCO-I parece ser una solución aceptable pues consigue unas tasas de compresión similares a las arrojadas por PPM pero con un coste muy inferior. BWT sigue siendo una alternativa real, que comprime menos pero es más rápida.

La suave mejora de las tasas de compresión para las imágenes de 16 bpp se debe a la baja correlación espacial reinante. El uso de CALIC no estaría suficientemente justificado respecto a BWT. Sólo LOCO-I, debido a las altas tasas de transferencia conseguidas es una alternativa real a BWT.

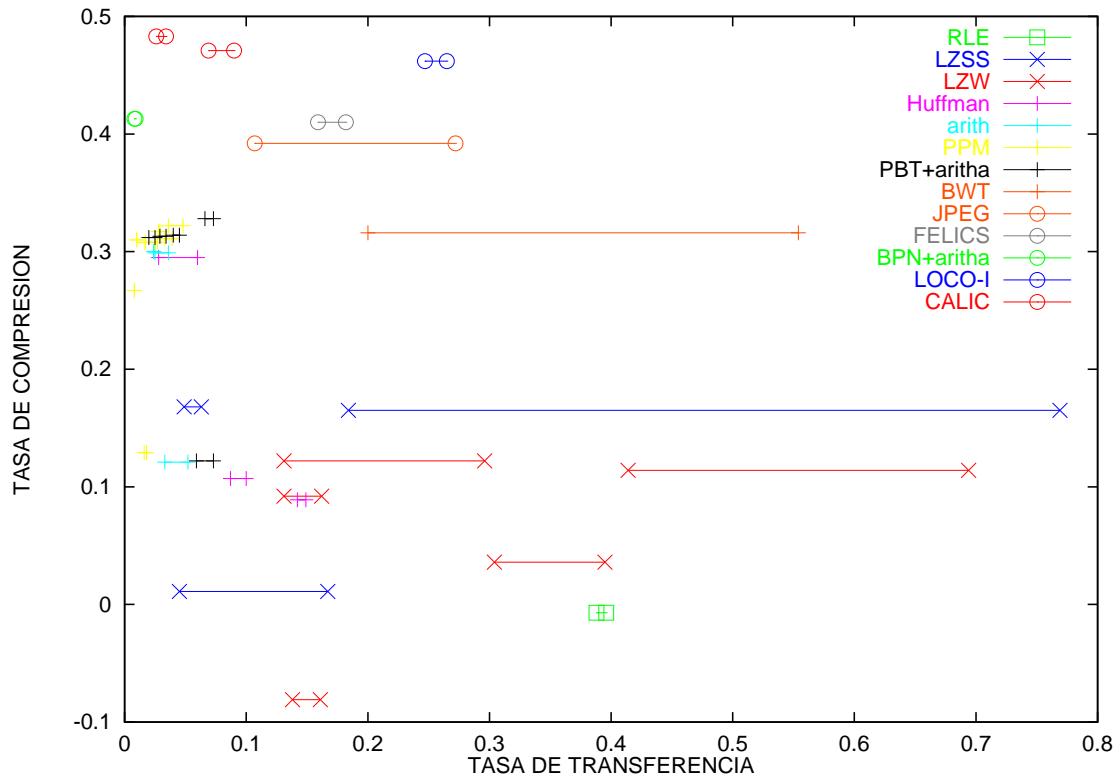


Figura 3.11: Rendimiento de los algoritmos de compresión de imágenes probados para las imágenes de 8 bpp. Los compresores de imágenes tienen asignado un círculo.

Tabla 3.6: Tasas de transferencia (multiplicadas por 1000) de los algoritmos de compresión/descompresión de imágenes secuenciales probados para las imágenes de 8 bpp.

método	imágenes				media
	lena	bárbara	boats	zelda	
JPEG	108/ 294	105/214	107/ 286	109/ 294	107/ 272
FELICS	185/161	169/147	185/161	187/167	182/159
BPN+aritha	8/9	8/10	8/9	8/9	8/9
LOCO-I	270 /238	250 / 244	270 /250	270 /256	265 /247
CALICh	89/69	80/67	94/68	97/71	90/69
CALICA	36/25	35/25	32/24	34/28	34/26

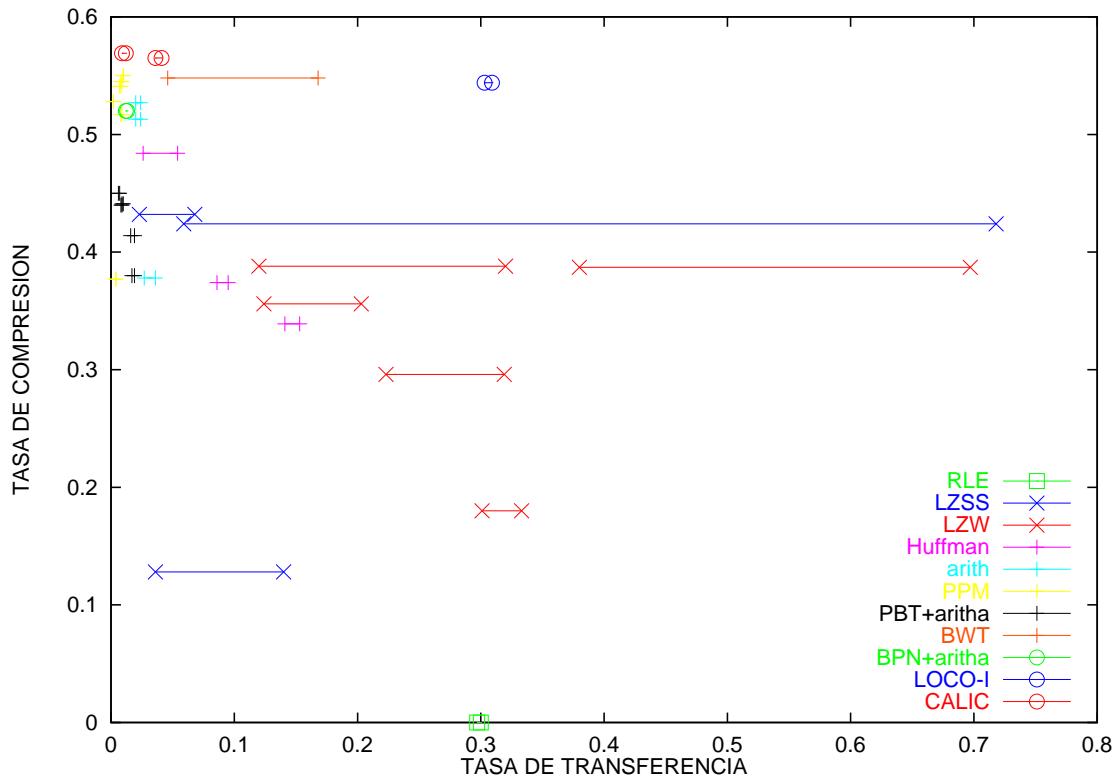


Figura 3.12: Rendimiento de los algoritmos de compresión de imágenes probados para las imágenes de 16 bpp. Los compresores de imágenes tienen asignado un círculo.

Tabla 3.7: Tasas de compresión (multiplicadas por 1000) de los algoritmos de compresión de imágenes secuenciales probados para las imágenes de 16 bpp.

método	imágenes				media
	B0010	29jul24	29jul25	29jul26	
BPN+aritha	558	520	500	501	520
LOCO-I	689	560	450	477	544
CALICH	708	575	476	499	565
CALICa	713	580	476	505	569

Tabla 3.8: Tasas de transferencia (multiplicadas por 1000) de los algoritmos de compresión/descompresión de imágenes secuenciales probados para las imágenes de 16 bpp.

método	imágenes				media
	B0010	29jul24	29jul25	29jul26	
BPN+aritha	9/10	13/14	13/14	13/14	12/13
LOCO-I	280/269	324/331	299/317	309/320	303/309
CALICh	37/40	36/36	45/34	45/34	41/36
CALICa	18/12	9/8	9/7	10/7	12/9

3.7 Resumen

En este capítulo se han analizado cuatro de los algoritmos de compresión *lossless* de imágenes secuenciales más usados actualmente: JPEG, FELICS, LOCO-I y CALIC. En ese orden, JPEG es el más rápido y CALIC el que más comprime. Según nuestras pruebas, mejoran aproximadamente las tasas de compresión de los compresores de texto en entre un 15% y un 20%.

Además, se ha presentado predictor diferente basado en una red neuronal entrenada mediante el algoritmo de retropropagación del error (BPN). Los resultados muestran que la adaptación del predictor es bastante buena puesto que parte de un desconocimiento total del contenido de una imagen. De hecho, para algunas de las imágenes, la predicción es mejor que la de CALIC, que implementa el predictor más sofisticado de todos los algoritmos descritos.

Sin duda, la idea de usar una BPN como predictor espacial adaptativo es interesante y puede dar buenos resultados especialmente en aquellas situaciones donde el contenido de la imagen sea bastante incierto o aleatorio (cosa que parece ocurrir en muchas imágenes de origen astronómico).

El problema principal derivado del uso de la red radica en que su aprendizaje continuado conlleva mucho tiempo. Para disminuirlo proponemos varias alternativas:

- Tener la red pre-entrenada usando una batería de imágenes características y sólo si se comprueba que la entropía de la imagen residuo no está siendo suficientemente baja comparada con la entropía de la imagen original, la red permanece sin entrenarse.
- Nuestra red usa aritmética en punto flotante para representar los pesos entre las conexiones sinápticas. Una mejora importante en el tiempo de predicción (sin contar con la mejora en el tiempo de aprendizaje) puede obtenerse si trabajamos con pesos enteros.
- A pesar de usar la red tal y como ha sido presentada, el proceso de descompresión puede ser mucho más rápido si junto con la imagen comprimida adjuntamos en forma

de cabecera los pesos de las conexiones encontradas por el compresor. Este esquema puede ser útil en aquellas situaciones donde no es necesario comprimir las imágenes como corrientes y el número de descompresiones es mucho mayor.

A parte de mejorar las tasas de transferencia, también convendría aumentar las tasas de compresión. Tanto LOCO-I como CALIC (los dos mejores métodos de compresión secuencial de imágenes conocidos) además de usar un predictor lo más adaptativo posible usan un codificador entrópico basado en el contexto. Este factor mejora significativamente las tasas de compresión y hace que las tasas arrojadas por nuestro compresor no sean realmente comparables con las de estos algoritmos.

Una de las principales aplicaciones de las redes neuronales es la clasificación de patrones. Sería lógico pensar en que si usáramos otra red neuronal para estimar la pendiente de la distribución de Laplace de la imagen residuo en función del contexto de predicción o del contexto del error de predicción, sería posible obtener una mejora apreciable de la tasas de compresión.

Capítulo 4

Transmisión de Imágenes

Uno de los mayores retos en los sistemas de telecontrol es la transmisión de las imágenes (en general señales) que un proceso remoto genera y que deben ser conocidas lo antes posible por el elemento controlador (normalmente una persona). En este contexto, la compresión de imágenes puede ser una gran ayuda de cara a disminuir los tiempos de transmisión. Debido a lo crítico que es el tiempo de respuesta, los algoritmos de transmisión más usados durante mucho tiempo han sido *lossy*. La imagen, antes de ser enviada se comprimía y se enviaba como un único paquete lógico. En el receptor la imagen se descomprimía en su totalidad y era presentada. Ya que la compresión de imágenes *lossy* podía alcanzar (dependiendo de la imagen y de la cantidad de información eliminada) niveles de compresión del orden de 100:1, el tiempo total de transferencia era decrementado a pesar de tener que gastar un cierto tiempo en comprimir primero la imagen.

El principal problema de esta aproximación es que la imagen recibida siempre es una aproximación de la original, factor que en muchos casos es inaceptable. Por ejemplo, en telemedicina el diagnóstico puede ser erróneo si la imagen no es exacta; en teleastronomía, la observación puede ser engañosa si se buscan objetos muy débiles que hayan podido desaparecer por ser confundidos con el ruido de fondo que ha sido eliminado por el compresor; etc.

En este capítulo se aborda el problema de la transmisión progresiva de imágenes. La gran diferencia entre la transmisión de imágenes comprimidas y la transmisión progresiva de imágenes (comprimidas o no) radica en (1) que el tiempo de espera por parte del controlador es inferior y (2) que la imagen recibida mejora en calidad a lo largo del tiempo. Así, en muy poco tiempo podemos ver una versión aproximada de la imagen final que será idéntica a la original si esperamos lo suficiente. Es la persona la que decide hasta cuándo hay que esperar para tomar la decisión (realizar el diagnóstico o controlar el telescopio).

Este capítulo se ha organizado en dos partes fundamentales. En la primera (que comprende hasta la Sección 4.12) se estudian diferentes transformaciones que pueden ser útiles en la transmisión progresiva de imágenes. En concreto se describen y analizan la transfor-

mada de Walsh-Hadamard, la transformada coseno, la transformada S y la transformada S+P. La segunda parte investiga un algoritmo de codificación de planos de bits llamado SPIHT, especialmente diseñado para la transformada wavelet discreta. El capítulo finaliza con una evaluación de los diferentes algoritmos sobre las imágenes de prueba.

4.1 Requisitos del codec progresivo

Un compresor reversible y progresivo de imágenes debe verificar las siguientes propiedades (las más importantes se enumeran primero):

- Finalizada la transmisión de los datos, la reconstrucción debe ser perfecta (sin pérdidas). Esto equivale a decir que si descomprimimos o transmitimos todo el *code-stream*, la imagen recuperada es idéntica a la original.
- La información visualmente más importante debe ser transmitida (y presentada en el receptor) primero. Este aspecto es fundamental en aplicaciones de *tele-browsing*¹ donde el usuario debe poder conocer el contenido más aproximado del resultado final en el mínimo tiempo posible. Así, se ahorra el tiempo de transmisión y el ancho de banda que sería necesario si la imagen no fuera la deseada.

Por otra parte, al enviar primero la información más relevante, la codificación progresiva de imágenes está especialmente recomendada en aquellos casos en los que la transmisión se realiza a través de canales que presentan ruido. Puesto que los primeros datos son los más importantes, son también estos los que más deben protegerse contra el ruido usando códigos de detección y/o corrección de errores.

De esta forma, si a partir de un cierto instante todos los bits son alterados por el ruido y además no existe ningún método de corrección de errores, la imagen todavía puede ser visible puesto que la información alterada es visualmente la menos importante.

- La reconstrucción de la imagen por parte del receptor debe realizarse de forma progresiva en fidelidad, no en resolución. Por lo tanto, en cualquier instante, el tamaño (en puntos reales) de la imagen reconstruida coincide con el tamaño de la imagen original. Dicho requisito es necesario en muchas situaciones de telecontrol donde se debe conocer con exactitud la situación de un determinado objeto dentro de la escena.
- El *code-stream* debe estar lo más comprimido posible. Esta condición es interesante ya que minimiza tanto el tiempo de visualización progresiva como el de transmisión total de la imagen. De esta manera, además podemos ahorrar espacio de almacenamiento.

¹Este término se usa para referirse al proceso de observar una colección grande de imágenes que son una aproximación o resumen (normalmente de menor resolución que las imágenes a las que representan) y que están almacenadas en un punto distante.

- El *codec* debe ser lo más simétrico posible. Así, un posible cuello de botella en un sistema de transmisión queda minimizado puesto que si el canal fuera el elemento más rápido (cosa normalmente improbable) entonces tanto el compresor como el descompresor estarían trabajando al 100% de su rendimiento. Si lo que ocurre es que el canal es el elemento más lento, entonces el compresor y el descompresor esperan en el mismo porcentaje.
- El *code-stream* debe ser lo más seccionable posible. Esto significa que el receptor debe ser capaz de usar cada bit de código que le llega para mejorar la reconstrucción. Como consecuencia, las mejoras que se producen en la imagen cuando se reciben nuevos datos son de la forma más suave posible.

Esta característica es especialmente interesante en el caso, como ya hemos comentado, de transmisiones a través de canales con ruido ya que el rebobinado para retransmitir se puede hacer a nivel de bit.

También se cumple por tanto que cualquier imagen comprimida puede separarse en dos partes por cualquier bit del *code-stream*. Así es posible construir sistemas de consulta de imágenes voluminosas. Si la información visualmente más importante se codifica primero, esta parte puede almacenarse en disco (un elemento de almacenamiento rápido) mientras que el resto del fichero comprimido puede almacenarse en cinta o CD-ROM (elementos de almacenamiento lentos y baratos). Sólo en el caso de que el usuario demandase el contenido exacto de la imagen habría que acceder al sistema de almacenamiento secundario.

4.2 Dominios de representación

Las imágenes de tono continuo son señales bidimensionales y como tales, existen básicamente dos formas de representarlas: (1) en el dominio del espacio y (2) en un dominio transformado, usualmente, el de la frecuencia.

En el dominio espacial, cada punto de la imagen representa la intensidad de un punto de la escena digitalizada. Sin embargo, en el dominio de la frecuencia cada punto (banda de frecuencia o coeficiente) representa a una determinada señal base. Por ejemplo, en el dominio de Fourier, cada coeficiente espectral complejo indica la potencia relativa de una función exponencial compleja cuya frecuencia depende de la posición del coeficiente en el espacio de frecuencias. El análisis de Fourier es muy útil porque permite representar la información contenida en cualquier señal como una sumatoria de señales de tipo senoidal y realizar así un análisis frecuencial de la señal.

En muchas situaciones, y la transmisión progresiva de imágenes es una de ellas, determinadas componentes de frecuencia son visualmente más importantes que otras, de forma que, si decidimos eliminar las menos importantes (filtramos), todavía podemos tener una versión muy aproximada de la imagen original. Por lo tanto, una representación espectral resuelve el problema de encontrar cuál es la información visualmente más relevante. En el

campo de la transmisión progresiva de imágenes este aspecto es esencial, ya que determina qué información hay que transmitir primero.

Otra propiedad muy interesante de la representación espectral es que si las funciones base son las adecuadas, es posible explotar la correlación entre los puntos de la imagen, de forma que si ésta es alta, muy pocos coeficientes serán capaces de representar una versión muy aproximada de la imagen original. En este sentido, una transformada puede ser considerada como un predictor que explota tanto la redundancia local como la global, dependiendo, como veremos, de la función base usada para correlacionar la señal. Este es el principio que soporta a todos los compresores de imágenes basados en transformadas, de los que el más conocido es el estándar JPEG [55, 117], que usa como función base la función coseno.²

En general, este planteamiento descarta a las alternativas de codificación sobre el dominio del espacio como son la codificación por planos de bits [115] o la codificación por intensidad de los puntos.³ Dichos métodos pueden llegar a cumplir todos los requisitos pero sólo parcialmente. Por ejemplo, si codificamos por planos de bits una imagen con 8 bits de profundidad, el problema está en que el *code-stream* sólo es realmente seccionable por 7 puntos, justo cuando comienza a codificarse el siguiente plano. Además, aunque se sabe que es más importante transmitir primero los planos de bits más significativos, no está claro qué bit transmitir primero dentro de cada plano.

4.3 La transformada de Walsh-Hadamard

La transformada de Walsh-Hadamard (WHT: Walsh-Hadamard Transform) [33, 52, 79, 94, 100] puede ser utilizada para representar una señal muestreada en el dominio frecuencial. Aunque no es la transformación más eficaz en lo que a compactación espectral se refiere (acumular la máxima cantidad de energía en el mínimo número de coeficientes), sí es totalmente reversible y por eso es de gran interés en este trabajo. A continuación se presenta una breve descripción.

La mayoría de las transformadas pueden ser expresadas como un producto de la señal a transformar por un *kernel* o función base, que es dependiente de la transformada. Sea⁴ $f[x]$ la señal a transformar y sea $F[u]$ su transformada (x denota espacio y u frecuencia). Si existen N muestras de la señal, la expresión genérica de cualquier par de transformadas

²El estándar JPEG analizado en el capítulo anterior es la versión *lossless*. Ahora nos referimos a la otra parte del estándar que contempla la compresión *lossy* y que se fundamenta en la transformada coseno.

³Aunque en algunas situaciones como pasa con las imágenes astronómicas, esta opción sea ya mucho más útil que la transmisión por filas o columnas, ya que las estrellas (lo visualmente más importante) son transmitidas primero.

⁴Notaremos al argumento de las funciones entre paréntesis y al argumento de las señales entre corchetes. Con esto se trata de enfatizar la idea de que la señal es para nosotros en general un conjunto de muestras que están determinadas por un índice.

responde a

$$\begin{aligned} F[u] &= \sum_{\substack{x=0 \\ N-1}}^{N-1} f[x]k(x, u) \\ f[x] &= \sum_{u=0}^{N-1} F[u]h(x, u). \end{aligned} \quad (4.1)$$

En el caso de la transformada de Walsh-Hadamard, el núcleo de la transformación directa es

$$k(x, u) = (-1)^{\sum_{i=0}^{n-1} b_i(x)b_i(u)} \quad (4.2)$$

y el de la inversa es

$$h(x, u) = \frac{k(x, u)}{N}, \quad (4.3)$$

donde $N = 2^n$ y $b_i(x)$ es el valor del i -ésimo bit del número x escrito en binario. La expresión de la transformada directa de Walsh-Hadamard es por tanto

$$F[u] = \sum_{x=0}^{N-1} f[x](-1)^{\sum_{i=0}^{n-1} b_i(x)b_i(u)} \quad (4.4)$$

y la expresión de la transformada inversa es

$$f[x] = \frac{1}{N} \sum_{u=0}^{N-1} F[u](-1)^{\sum_{i=0}^{n-1} b_i(x)b_i(u)}. \quad (4.5)$$

Como puede apreciarse, la transformada directa de Walsh-Hadamard es igual a la inversa salvo por un factor de escala.

Nótese que para calcular la transformada no es necesario efectuar los productos, ya que precalculada la matriz de transformación, sólo tenemos que cambiar los signos de las muestras. La matriz de transformación de Walsh-Hadamard puede ser calculada a partir de la siguiente expresión recursiva [97]

$$H_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}; H_4 = \begin{bmatrix} H_2 & H_2 \\ H_2 & -H_2 \end{bmatrix}; \dots; H_{2N} = \begin{bmatrix} H_N & H_N \\ H_N & -H_N \end{bmatrix}. \quad (4.6)$$

El número de operaciones en ambas transformadas es proporcional a N^2 pues existen N sumas para cada una de las N muestras.

El par de transformadas de Walsh-Hadamard bidimensionales, para matrices de $N \times M$ elementos (muestras) responde a

$$\begin{aligned} F[u, v] &= \sum_{x=0}^{N-1} \sum_{y=0}^{M-1} f[x, y](-1)^{\sum_{i=0}^{n-1} b_i(x)b_i(u)+b_i(y)b_i(v)} \\ f[x, y] &= \frac{1}{N \times M} \sum_{u=0}^{N-1} \sum_{v=0}^{M-1} F[u, v](-1)^{\sum_{i=0}^{n-1} b_i(x)b_i(u)+b_i(y)b_i(v)}. \end{aligned} \quad (4.7)$$

La transformación multidimensional de Walsh-Hadamard es separable. Por tanto, podemos calcular la transformada bidimensional aplicando la transformación unidimensional sobre las filas (o las columnas) y a continuación aplicar la misma transformación sobre las columnas (o filas).

Para finalizar, conviene destacar que existe un algoritmo rápido de cálculo de la WHT llamado FWHT (*Fast WHT*) que arroja un tiempo proporcional a $N \times \log_2(N)$ [79]. En el apéndice B se muestra dicho algoritmo.

4.4 La transformada coseno

La transformada discreta del coseno (DCT: *Discrete Cosine Transform*) [4, 33, 79, 85, 86] arroja mayores índices de compactación espectral que la transformada de Walsh-Hadamard.⁵ Por esta razón se usa en compresión *lossy* de imágenes [55, 117] y también se ha decidido estudiar su comportamiento como herramienta de transmisión progresiva continua, aunque no es reversible porque utiliza aritmética en punto flotante.

La DCT fue introducida por Ahmed *et al* en 1974 y tiene al menos 4 variantes diferentes [4]. La aquí presentada se conoce como tipo II y posee como expresión

$$\begin{aligned} F[u] &= \sqrt{\frac{2}{N}} e(u) \sum_{x=0}^{N-1} f[x] \cos\left(\pi \frac{(2x+1)u}{2N}\right) \\ f[x] &= \sqrt{\frac{2}{N}} \sum_{u=0}^{N-1} F[u] e(u) \cos\left(\pi \frac{(2x+1)u}{2N}\right), \end{aligned} \quad (4.8)$$

donde $e(0) = 1/\sqrt{2}$ y $e(u) = 1$ para $0 < u < N$.

La matriz de transformación de la DCT es de la forma

$$C(u, x) = \begin{cases} \frac{1}{\sqrt{2N}} & \text{si } u = 0 \\ \sqrt{\frac{2}{N}} \cos\left(\pi \frac{(2x+1)u}{2N}\right) & \text{si } 0 < u < N. \end{cases} \quad (4.9)$$

El cálculo de la DCT arroja algoritmos con complejidad $O(N^2)$ pues existen N productos escalares de vectores de tamaño N . El par de transformadas coseno bidimensionales

⁵De hecho, la DCT es, por debajo de la transformada de Karhunen-Loëve (KLT: Karhunen-Loëve Transform) [51, 79] la transformada que mejor descorrelaciona. La KLT no es usada porque es dependiente de los datos y no existe una transformada rápida. De hecho, el cálculo de la KLT requiere $O(N^3)$ operaciones donde N es el número de puntos de la imagen [122].

es

$$\begin{aligned} F[u, v] &= \frac{2}{\sqrt{N \times M}} e(u)e(v) \sum_{x=0}^{N-1} \sum_{y=0}^{M-1} f[x, y] \cos\left(\pi \frac{(2x+1)u}{2N}\right) \cos\left(\pi \frac{(2y+1)v}{2M}\right) \\ f[x, y] &= \frac{2}{\sqrt{N \times M}} \sum_{u=0}^{N-1} \sum_{v=0}^{M-1} F[u, v] e(u)e(v) \cos\left(\pi \frac{(2x+1)u}{2N}\right) \cos\left(\pi \frac{(2y+1)v}{2M}\right). \end{aligned} \quad (4.10)$$

La DCT multidimensional es separable y tiene un algoritmo rápido llamado FCT (*Fast Cosine Transform*) que la calcula en $O(N \times \log_2(N))$ (ver Apéndice B).

4.5 Selección de los coeficientes espectrales

En esta sección vamos a estudiar las diferentes alternativas que existen a la hora de transmitir los coeficientes espectrales generados por la WHT y la DCT para determinar la política de selección más adecuada.

De la observación de la distribución de amplitudes de los coeficientes se puede deducir el orden más conveniente para una transmisión progresiva. La Figura 4.1 muestra los espectros WHT y DCT de dos imágenes con dimensiones 256×256 . Los puntos más brillantes indican coeficientes más potentes (en el logaritmo de su valor absoluto).

Tanto la WHT como la DCT son transformaciones unitarias, lo que significa que muchas de las medidas que sean realizables en el dominio del espacio son también válidas en el dominio de la frecuencia [79]. En concreto, la norma Euclídea es invariantes y por tanto, el error cuadrático medio (MSE: *Mean Squared-Error*) entre dos señales o imágenes puede ser calculado directamente sobre el dominio frecuencial.⁶ Como es lógico pensar, los coeficientes más importantes (visualmente hablando) son los que tienen mayor potencia, pues contienen mayor cantidad de información. De hecho, si c es un coeficiente espectral y es transmitido (filtrado), el MSE entre la imagen original y la imagen transmitida (filtrada) se decrementa en c^2/N donde N es el número de coeficientes espectrales [24]. Por tanto, lo más eficiente es enviar primero los coeficientes más grandes en valor absoluto pues son los que más energía almacenan [103, 105].

Por otro lado, como se puede apreciar en la Figura 4.1, tanto en la transformada de Walsh-Hadamard como en la del coseno, dichos coeficientes tienden a concentrarse sobre las bandas de menor frecuencia (la esquina superior izquierda aparece más clara) cuando el algoritmo *bit-reversal* ha sido aplicado (ver apéndice B).

A la vista a estos resultados, hemos estudiado el rendimiento de tres políticas de selección (ver Figura 4.2):

- Emitir primero los coeficientes de mayor valor absoluto con el objetivo de mandar lo antes posible la mayor cantidad de información acerca de la imagen. Nótese que

⁶Se ha elegido esta medida de la discrepancia entre imágenes porque en general, y desde un punto de vista visual, es acertada: dos imágenes muy semejantes tienen un MSE muy pequeño y viceversa.

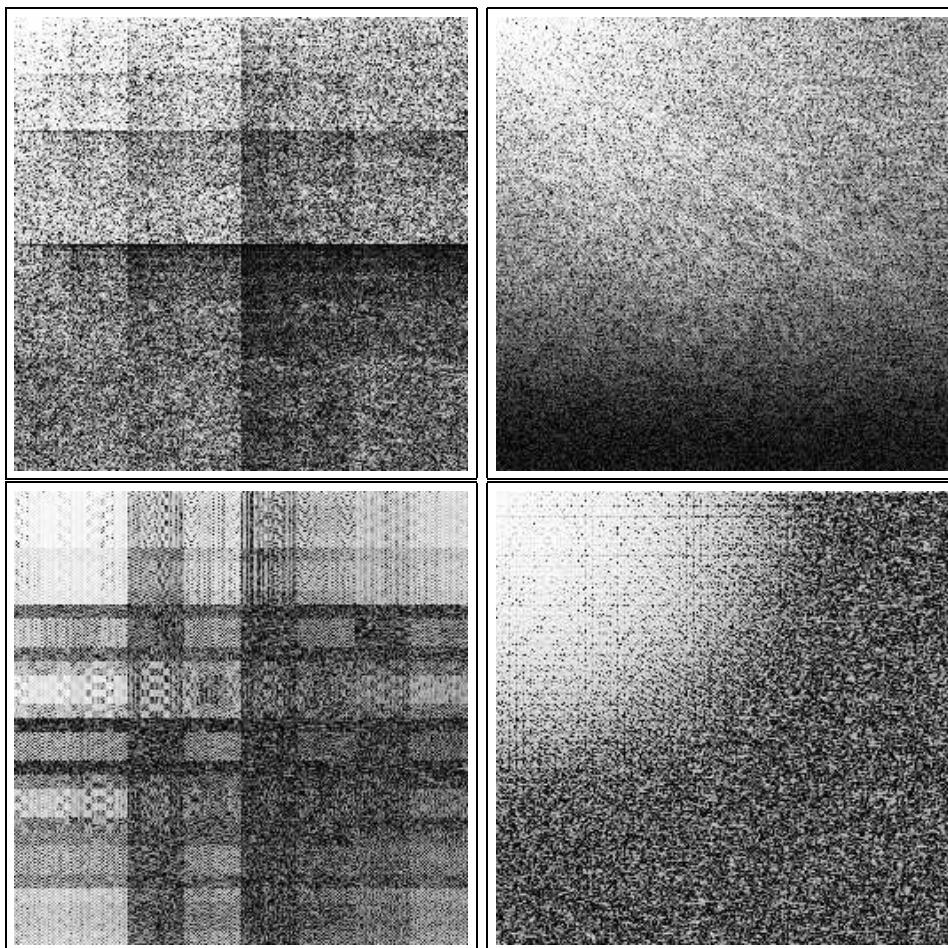


Figura 4.1: Espectro WHT (izquierda) y DCT (derecha) de la imagen “lena” (arriba) y de un detalle de 256×256 puntos (sección (512,512,768,768)) de “B0100” (abajo).

para hacer esto, no es necesario aplicar el algoritmo de reordenación *bit-reversal* tras la transformación rápida y por consiguiente los coeficientes de mayor potencia no se van a concentrar en la esquina superior izquierda. Con este filtro, habría que enviar tanto las coordenadas de los coeficientes como el valor de los mismos.

- Enviar los coeficientes de forma ordenada, atendiendo a su posición dentro del espectro de forma que primero se envía el coeficiente de menor frecuencia, formando un triángulo isósceles situado en la esquina superior izquierda. Con este tipo de filtrado, no habría que enviar las coordenadas de los coeficientes.
- Ídem, pero formando un cuadrado con su vértice superior izquierdo situado en la parte superior izquierda del espectro.

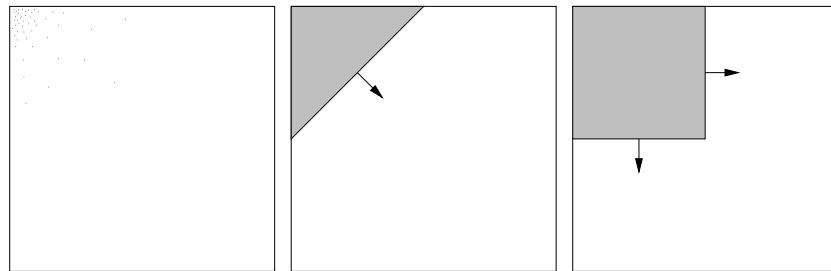


Figura 4.2: Políticas de ordenación de componentes frecuenciales para transmisión incremental: máxima potencia (izquierda), crecimiento triangular (centro) y crecimiento cuadrado (derecha).

Se han estudiado estas tres políticas de transmisión con el objetivo de determinar la más adecuada. El análisis se ha realizado primero desde un punto de vista visual y después atendiendo a la diferencia entre las imágenes transmitidas y la original. Debido al gran volumen de imágenes que se generan, sólo se presentarán resultados para dos imágenes: “lena” en su versión de 256×256 puntos y la sección de la imagen “B0100” también de 256×256 puntos, concretamente la que comienza en el punto de coordenadas (512,512).⁷

Las Figuras 4.3 y 4.4 muestran la transmisión progresiva de “lena” usando la WHT, las Figuras 4.5 y 4.6 muestran la transmisión progresiva de “lena” usando la DCT, las Figuras 4.7 y 4.8 muestran la transmisión progresiva de “B0100” usando la WHT y las Figuras 4.9 y 4.10 muestran la transmisión progresiva de “B0100” usando la DCT.

⁷Los datos también se corresponden con la transmisión de dicha sección y no para toda la imagen.



Figura 4.3: Transmisión progresiva de “lena” usando la WHT, para (de arriba a abajo) 1000, 2000, 3000 y 4000 coeficientes espectrales transmitidos de un total de 65536. En la columna de la izquierda: filtro de máxima potencia, en la central: filtro triangular y a la derecha: filtro cuadrado.



Figura 4.4: Transmisión progresiva de “lena” usando la WHT, para (de arriba a abajo) 5000, 6000, 7000 y 8000 coeficientes espectrales transmitidos de un total de 65536. En la columna de la izquierda: filtro de máxima potencia, en la central: filtro triangular y a la derecha: filtro cuadrado.



Figura 4.5: Transmisión progresiva de “lena” usando la DCT, para (de arriba a abajo) 1000, 2000, 3000 y 4000 coeficientes espectrales transmitidos de un total de 65536. En la columna de la izquierda: filtro de máxima potencia, en la central: filtro triangular y a la derecha: filtro cuadrado.



Figura 4.6: Transmisión progresiva de “lena” usando la DCT, para (de arriba a abajo) 5000, 6000, 7000 y 8000 coeficientes espectrales transmitidos de un total de 65536. En la columna de la izquierda: filtro de máxima potencia, en la central: filtro triangular y a la derecha: filtro cuadrado.

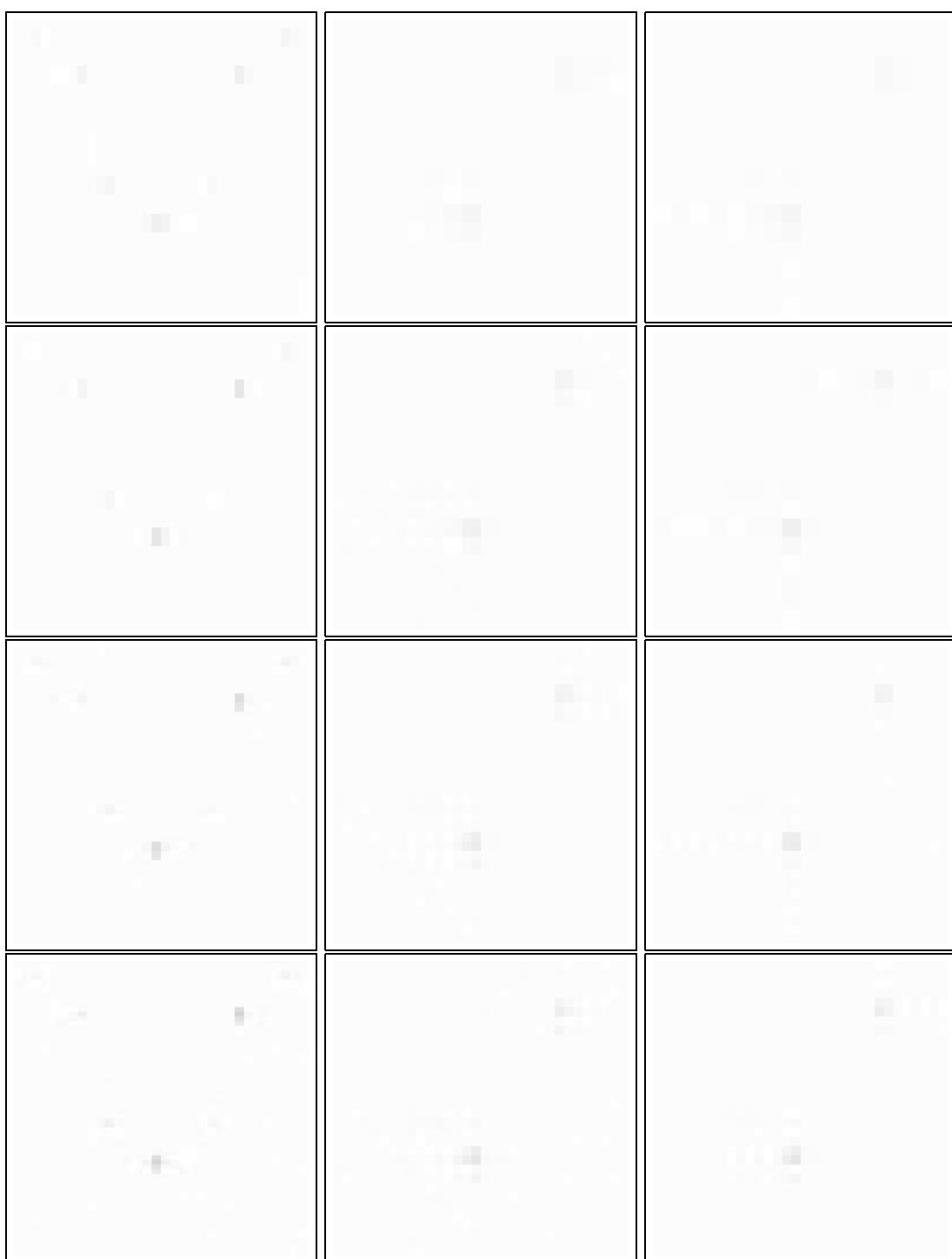


Figura 4.7: Transmisión progresiva de la región (512,512,768,768) de 256×256 puntos de “B0100” usando la WHT, para (de arriba a abajo) 100, 200, 300 y 400 coeficientes espectrales transmitidos de un total de 65536. En la columna de la izquierda: filtro de máxima potencia, en la central: filtro triangular y a la derecha: filtro cuadrado.



Figura 4.8: Transmisión progresiva de la región (512,512,768,768) de 256×256 puntos de “B0100” usando la WHT, para (de arriba a abajo) 500, 600, 700 y 800 coeficientes espectrales transmitidos de un total de 65536. En la columna de la izquierda: filtro de máxima potencia, en la central: filtro triangular y a la derecha: filtro cuadrado.

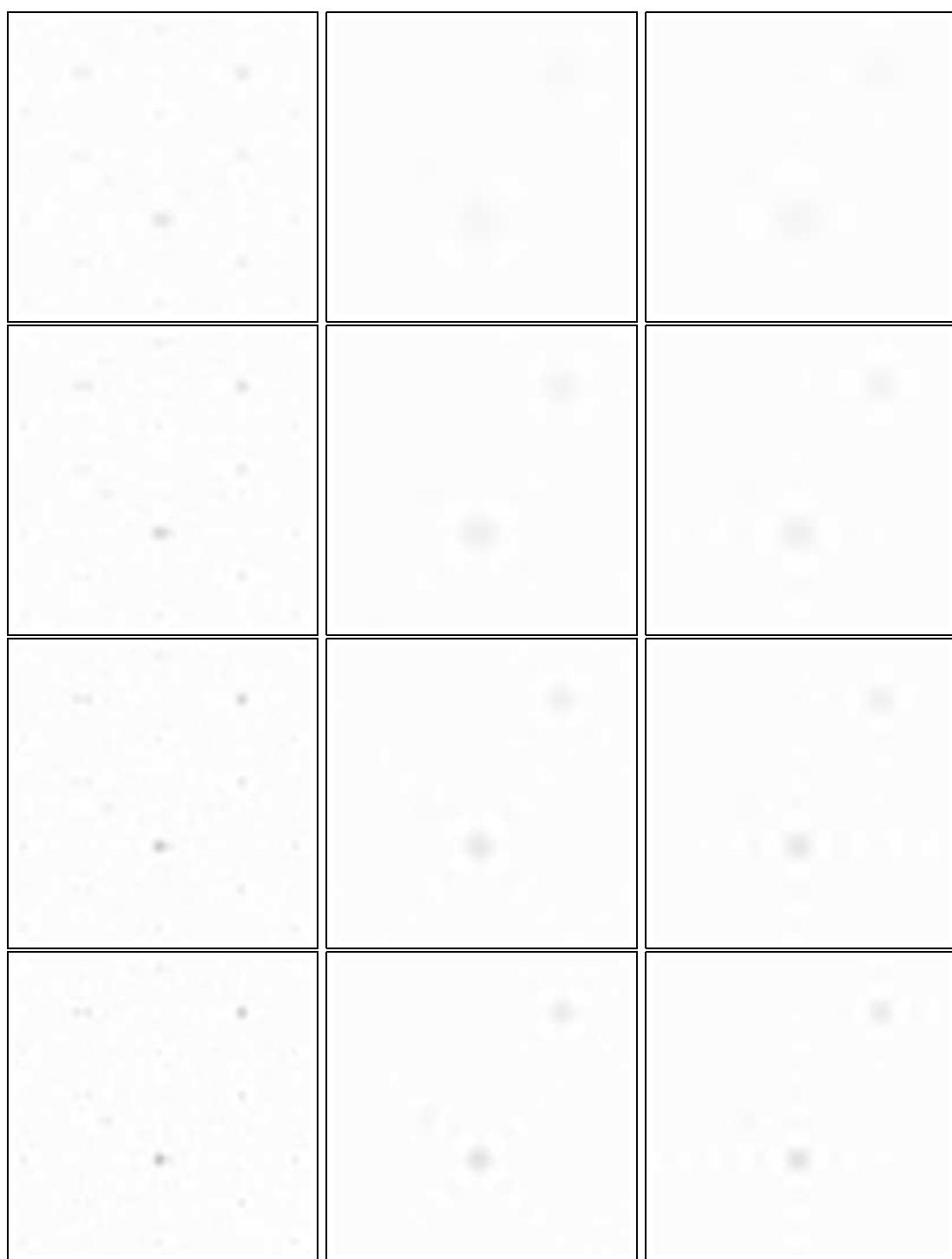


Figura 4.9: Transmisión progresiva de la región (512,512,768,768) de 256×256 puntos de “B0100” usando la DCT, para (de arriba a abajo) 100, 200, 300 y 400 coeficientes espectrales transmitidos de un total de 65536. En la columna de la izquierda: filtro de máxima potencia, en la central: filtro triangular y a la derecha: filtro cuadrado.

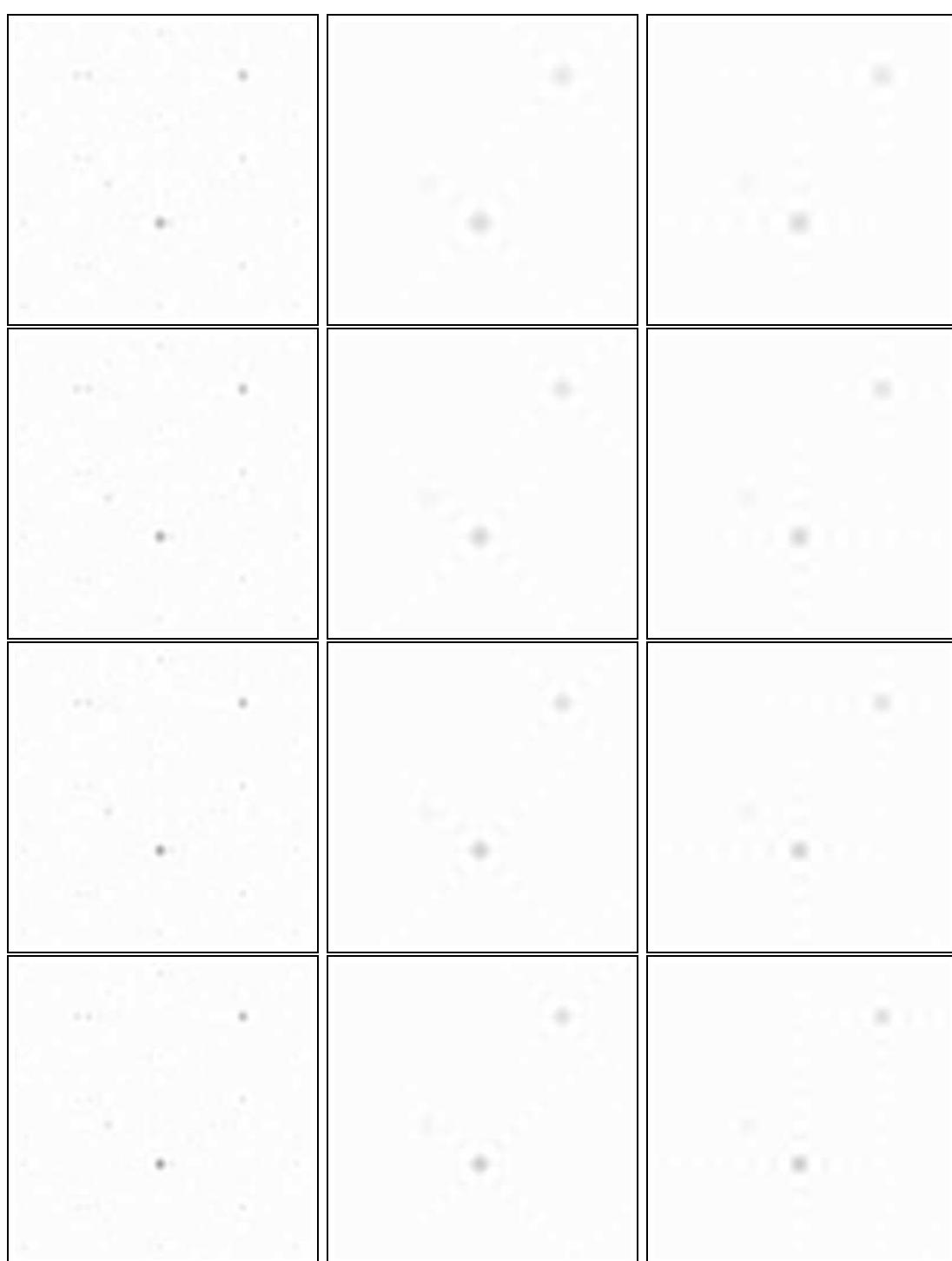


Figura 4.10: Transmisión progresiva de la región (512,512,768,768) de 256×256 puntos de “B0100” usando la DCT, para (de arriba a abajo) 500, 600, 700 y 800 coeficientes espectrales transmitidos de un total de 65536. En la columna de la izquierda: filtro de máxima potencia, en la central: filtro triangular y a la derecha: filtro cuadrado.

Tabla 4.1: Calidad de la reconstrucción en PSNR(dB) para “lena”. En cada columna se presenta a la izquierda el valor para la transformada de Walsh-Hadamard y a la derecha el valor para la transformada del coseno.

coeficientes transmitidos	filtro		
	max. pot.	triangular	cuadrado
1000	22.20 / 24.29	20.28 / 22.32	20.38 / 22.28
2000	23.88 / 26.24	21.72 / 24.22	21.09 / 24.12
3000	24.99 / 27.58	22.73 / 25.36	22.10 / 25.23
4000	25.87 / 28.67	23.37 / 26.13	23.37 / 26.09
5000	26.63 / 29.61	23.90 / 26.79	23.72 / 26.79
6000	27.30 / 30.45	24.42 / 27.43	23.99 / 27.45
7000	27.91 / 31.24	24.92 / 28.03	24.28 / 28.09
8000	28.49 / 31.97	25.37 / 28.57	24.62 / 28.68
9000	29.03 / 32.67	25.80 / 29.05	24.93 / 29.20
10000	29.54 / 33.34	26.17 / 29.51	25.28 / 29.68
20000	33.95 / 39.47	28.91 / 33.33	28.43 / 33.28
30000	38.22 / 45.49	31.15 / 36.66	29.78 / 36.39

Los resultados numéricos de dichas transmisiones expresados como la relación señal ruido pico (PSNR: *Peak Signal to Noise Ratio*)⁸ se exponen en las Tablas 4.1 y 4.2. De estos resultados se puede concluir que:

- El mejor criterio de selección es por máxima potencia, tanto para la transformada de Walsh-Hadamard como para la del coseno. Las otras dos alternativas tienen un rendimiento similar, aunque el filtro triangular parece ser el más adecuado de los dos. Este resultado es coherente si tenemos en cuenta que por ejemplo el estándar JPEG usa un recorrido en Zig-Zag muy semejante al filtro triangular para recorrer los coeficientes. Dicho recorrido comienza por la esquina superior izquierda y termina por la inferior derecha, cumpliéndose que la varianza del valor absoluto de los coeficientes

⁸Para el cálculo de los PSNR en decibelios (dB) hemos usado la expresión

$$\text{PSNR(dB)} = 20 \log_{10} \left(\frac{2^{\text{bpp}} - 1}{\text{RMSE}} \right), \quad (4.11)$$

donde

$$\text{RMSE} = \sqrt{\frac{1}{NM} \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} (f[i, j] - \hat{f}[i, j])^2}, \quad (4.12)$$

donde N y M son las dimensiones de la imagen, bpp es el número de bits/punto, f es la imagen original y \hat{f} la transmitida (filtrada).

Tabla 4.2: Calidad de la reconstrucción en PSNR(dB) para una sección de 256×256 puntos situada en la región (512,512,768,768) de “B0100”. En cada columna se presenta a la izquierda el valor para la transformada de Walsh-Hadamard y a la derecha el valor para la transformada del coseno.

coeficientes transmitidos	filtro		
	max. pot.	triangular	cuadrado
100	37.75 / 38.00	37.36 / 37.39	37.39 / 37.39
200	38.17 / 38.60	37.54 / 37.64	37.52 / 37.66
300	38.58 / 39.11	37.73 / 37.88	37.68 / 37.92
400	38.95 / 39.59	37.89 / 38.16	37.94 / 38.20
500	39.36 / 40.02	38.17 / 38.42	38.03 / 38.44
600	39.72 / 40.44	38.32 / 38.68	38.25 / 38.70
700	40.05 / 40.83	38.51 / 38.94	38.46 / 38.92
800	40.39 / 41.19	38.68 / 39.17	38.74 / 39.18
900	40.75 / 41.54	38.90 / 39.44	39.01 / 39.49
1000	41.04 / 41.87	39.02 / 39.66	39.23 / 39.70
2000	43.09 / 44.48	40.88 / 41.80	40.59 / 41.85
3000	44.35 / 46.35	42.14 / 43.63	42.46 / 43.70

es decreciente.

- La transformada más adecuada es la del coseno, debido probablemente a que las funciones base que usa esta transformación (funciones senoidales) son más adecuadas para la reconstrucción de imágenes de tono continuo que las funciones cuadradas. El problema, como ya indicamos anteriormente, reside en que la transformada coseno no garantiza la reconstrucción exacta de cualquier imagen.

4.6 Codificación de los espectros

Analizaremos ahora qué coste en términos de memoria arrojan las dos alternativas básicas: transmitir por máxima potencia o transmitir usando un orden preestablecido.

Máxima potencia

Aunque el filtro de máxima potencia es el más adecuado, cuando nos planteamos la codificación eficiente de los espectros de Walsh-Hadamard y de la transformada coseno, es necesario tener en cuenta otros factores.

En primer lugar, hay que transmitir las coordenadas de los coeficientes. Para hacernos una idea de cómo se ordenan los coeficientes de mayor a menor (en valor absoluto) cuando

el algoritmo de ordenación *bit-reversal* ha sido utilizado, las Figuras 4.11, 4.12, 4.13 y 4.14 muestran qué coeficientes son transmitidos para conseguir las secuencias de imágenes anteriormente expuestas de “lena” y “B0100”. Como puede apreciarse, no existe un patrón claro que permita extraer una gran cantidad de redundancia de la secuencia de coordenadas, excepto que primero tienden a enviarse los coeficientes de baja frecuencia. Esto es consecuencia de que la WHT y la DCT tienen como misión principal descorrelacionar espacialmente los coeficientes.



Figura 4.11: Coeficientes enviados con el filtrado progresivo para “lena” usando la WHT, para 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000, 20000 y 30000 coeficientes espectrales transmitidos de un total de 65536, usando el criterio de máxima potencia. En cada imagen sólo aparecen los coeficientes que no han sido filtrados anteriormente.

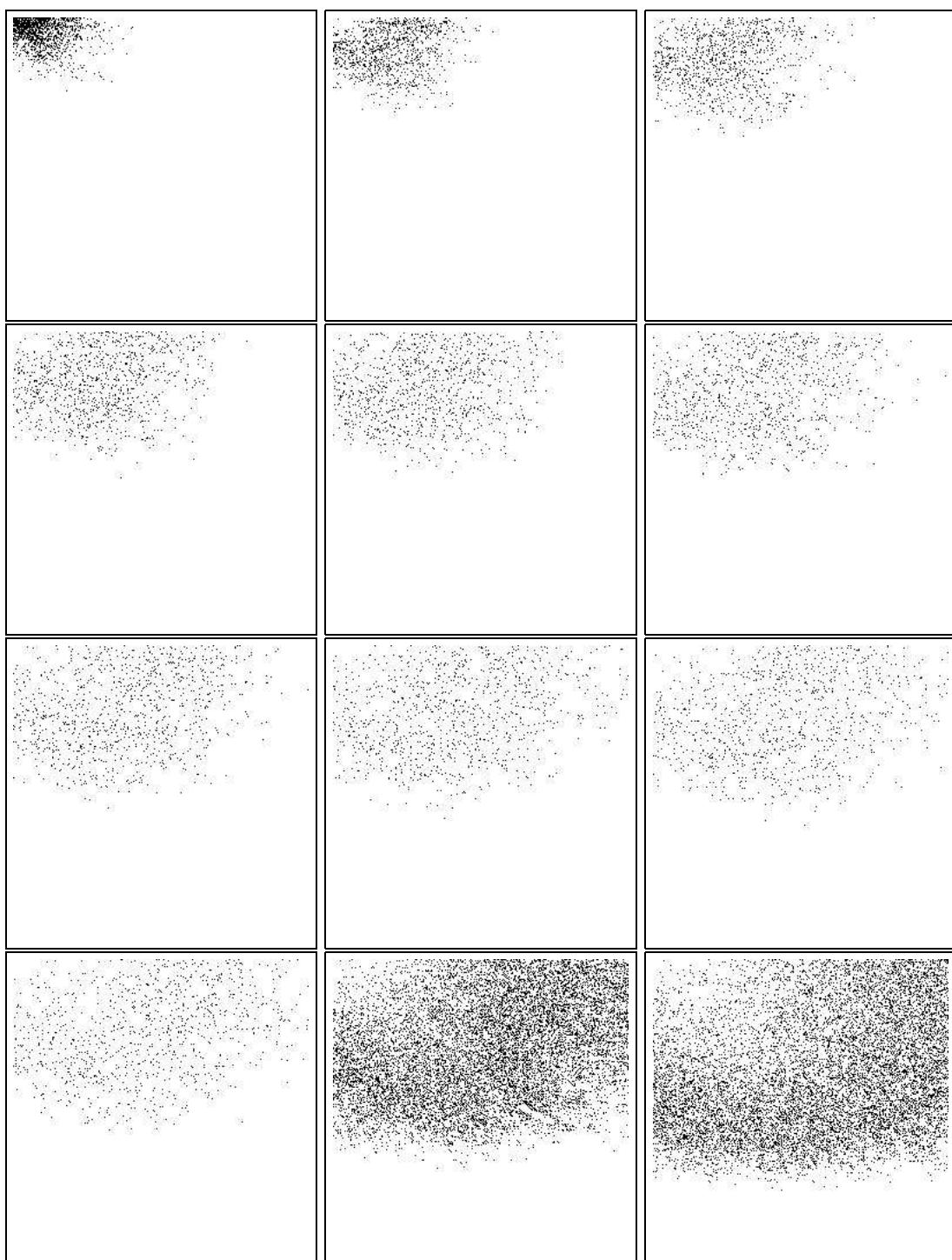


Figura 4.12: Coeficientes enviados con el filtrado progresivo para “lena” usando la DCT, para 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000, 20000 y 30000 coeficientes espectrales transmitidos de un total de 65536, usando el criterio de máxima potencia. En cada imagen sólo aparecen los coeficientes que no han sido filtrados anteriormente.

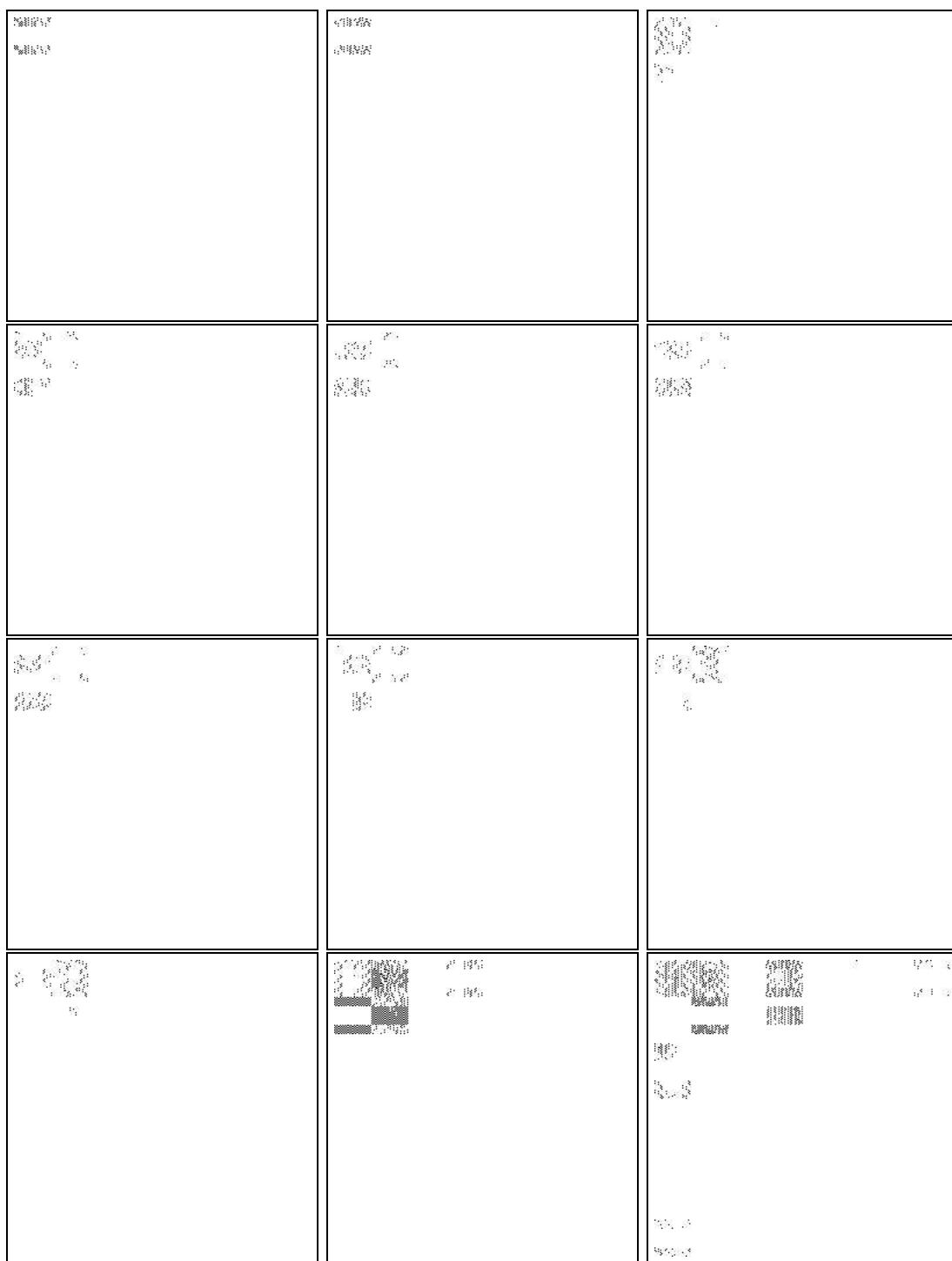


Figura 4.13: Coeficientes enviados con el filtrado progresivo para la región (512,512,768,768) de 256×256 puntos de “B0100” usando la WHT, para 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 2000 y 3000 coeficientes espectrales transmitidos de un total de 65536, usando el criterio de máxima potencia. En cada imagen sólo aparecen los coeficientes que no han sido filtrados anteriormente.

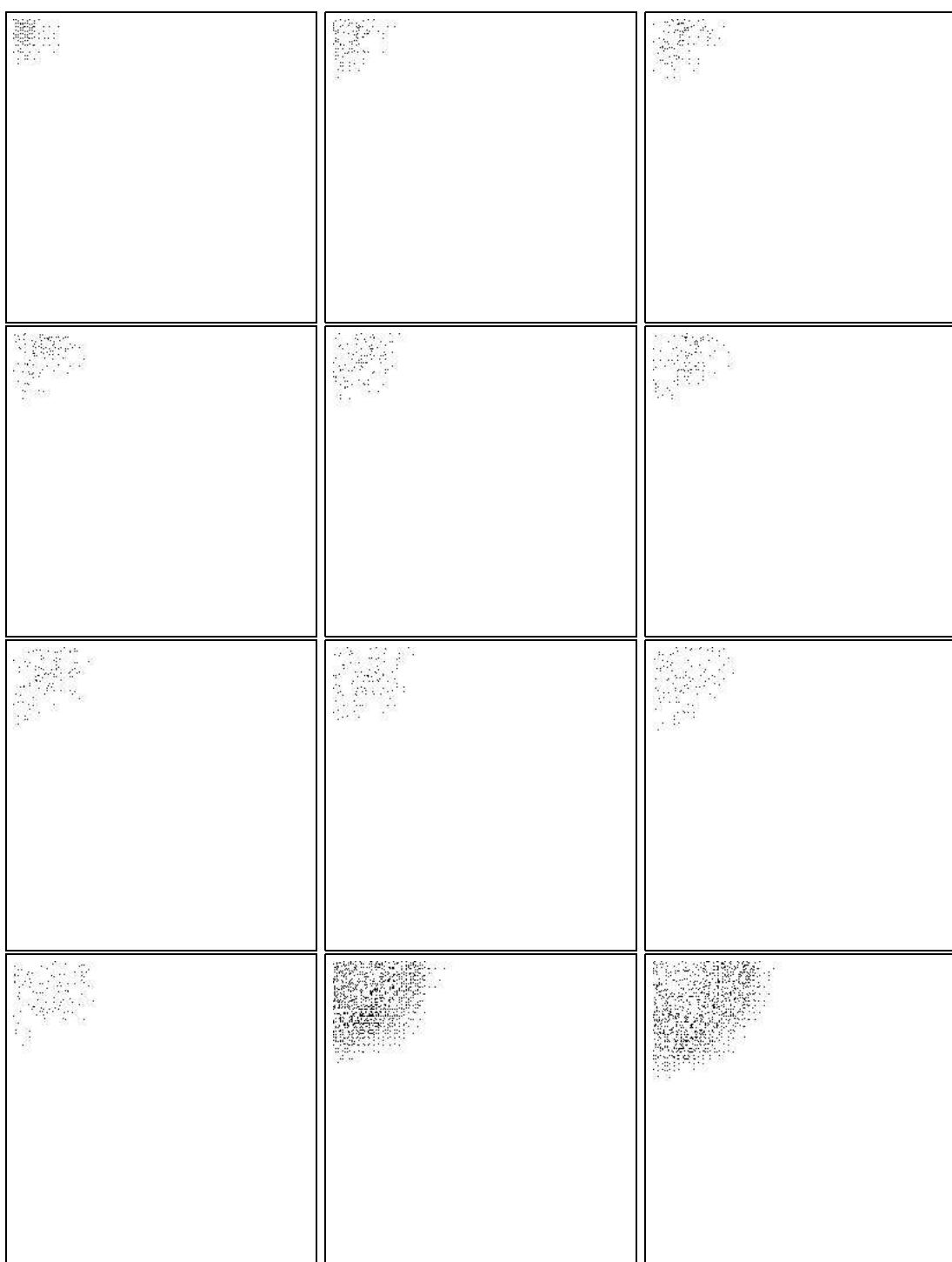


Figura 4.14: Coeficientes enviados con el filtrado progresivo para la región $(512, 512, 768, 768)$ de 256×256 puntos de “B0100” usando la DCT, para 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 2000 y 3000 coeficientes espectrales transmitidos de un total de 65536, usando el criterio de máxima potencia. En cada imagen sólo aparecen los coeficientes que no han sido filtrados anteriormente.

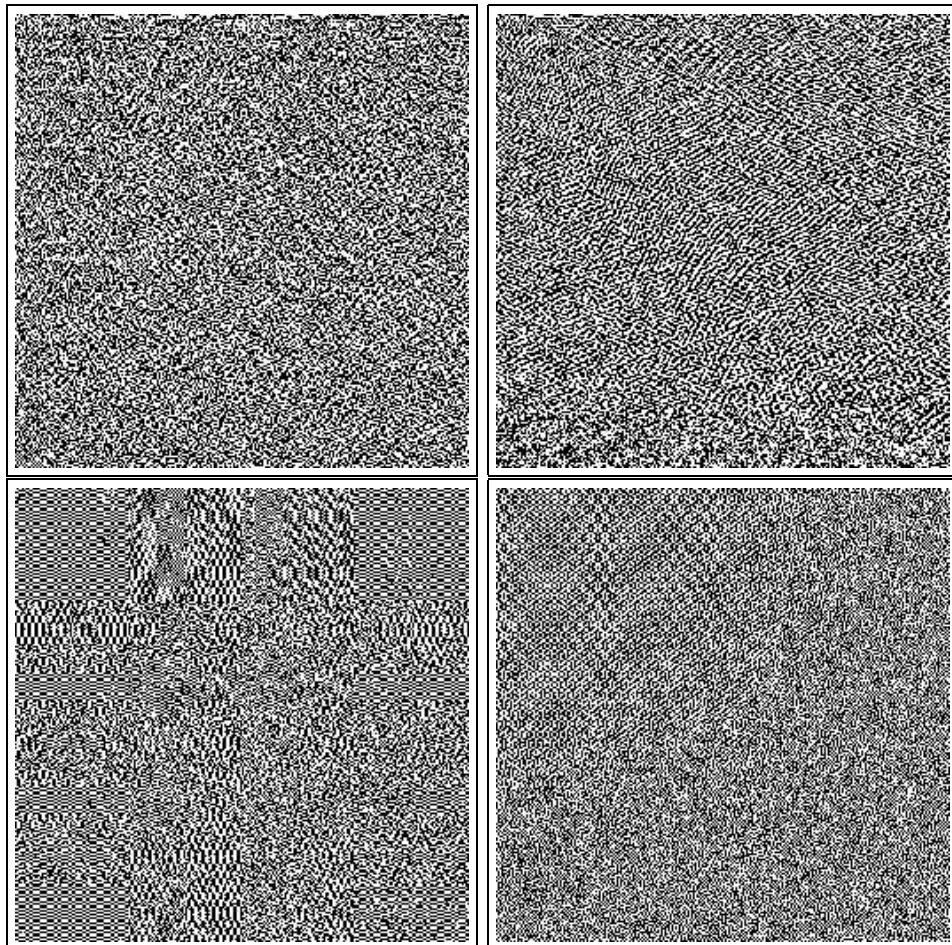


Figura 4.15: Arriba, signos de la WHT (izquierda) y de la DCT (derecha) de “lena”. Abajo, ídem para el detalle de 256×256 puntos de “B0100”.

En segundo lugar, hay que transmitir el valor de los coeficientes (signo y valor absoluto). El signo está muy descorrelacionado y no es posible comprimirlo (ver Figura 4.15). Sin embargo, los coeficientes están ordenados y los valores absolutos forman una secuencia monótona no creciente. Esta puede ser codificada por un código de longitud variable que transmite la diferencia entre los valores absolutos de los coeficientes formando una SDPG.

No se ha realizado una evaluación de qué niveles de compresión son alcanzables emitiendo por máxima potencia. Sin embargo, ya que las coordenadas de cada coeficiente deben ser transmitidas y no existe correlación estadística suficiente entre la magnitud de los coeficientes y su posición dentro del espectro, no son esperables grandes tasas de compresión. Para establecer esta afirmación se ha tenido también en cuenta que hay que enviar el valor de cada coeficiente, aunque posiblemente de forma comprimida.

Otro problema derivado de transmitir por máxima potencia se debe a que primero se envían aquellos coeficientes más grandes y el rango dinámico en el que se mueven es muy superior al rango de los puntos de la imagen. Por lo tanto, la diferencia entre ellos va a ser también importante. Esto significa que justo al comienzo de la transmisión la tasa de compresión va a ser menor, lo que es un grave inconveniente ya que es la información más importante, e interesa que sea recibida lo antes posible.

Filtro establecido: triangular o cuadrado

Cuando transmitimos los coeficientes según un orden preestablecido de antemano, nos ahorraremos la transmisión de las coordenadas. Por desgracia, la resolución de este problema deriva en la aparición de otro nuevo. Ahora los coeficientes no se recorren por orden de amplitud y es complejo encontrar un código eficiente para codificarlos, debido fundamentalmente al rango dinámico tan grande que tienen y a la baja correlación (tanto estadística como espacial) que existe entre ellos.

Las Figuras 4.16, 4.17, 4.18 y 4.19 muestran la magnitud de los coeficientes WHT y DCT para las imágenes “lena” y “B0100” (detalle), cuando usamos el filtro triangular y cuadrado. La ausencia de correlación entre los coeficientes de la secuencia transmitida limita el orden del modelo (usado para su codificación) a 0. En el caso de la WHT, el rango dinámico de los coeficientes es [0,6458608] para “lena” y [0,16576426] para “B0100”.⁹ Para resolver el problema de la codificación de valores tan grandes, lo más lógico parece ser el uso de un código de longitud variable¹⁰ y que el modelo inicialmente asigne una probabilidad muy parecida a todos los símbolos que pueden ocurrir en el intervalo marcado por el rango dinámico. Al ir produciéndose la codificación de la secuencia de coeficientes, la probabilidad de encontrar valores pequeños aumenta considerablemente. Ésta es la única fuente de redundancia que es posible explotar a nuestro juicio.

4.7 Requisitos de una nueva transformación

Por todo lo expuesto en las secciones anteriores, no está muy claro que la WHT sea la mejor opción. En primer lugar, la compactación espectral no es muy alta. Ni siquiera la DCT es tan buena como sería deseable, ya que al comienzo de la transmisión algunos de los elementos más importantes (por ejemplo, las estrellas en las imágenes astronómicas) están muy difuminadas y aparecen problemas de falsos anillos alrededor de las fronteras pronunciadas (*ringing*).

La razón de este fenómeno se debe fundamentalmente a que dentro de una imagen encontramos objetos o estructuras que no casan adecuadamente con las funciones base que usan estas transformadas. Para la WHT, la función base es una función cuadrada y

⁹Recuérdese que la DCT ha sido estudiada porque descorrelaciona más que la WHT, pero finalmente no puede ser usada porque no asegura que la transformación inversa de la imagen sea idéntica punto a punto a la original.

¹⁰Cualquiera de los estudiados en el capítulo 2 es válido.

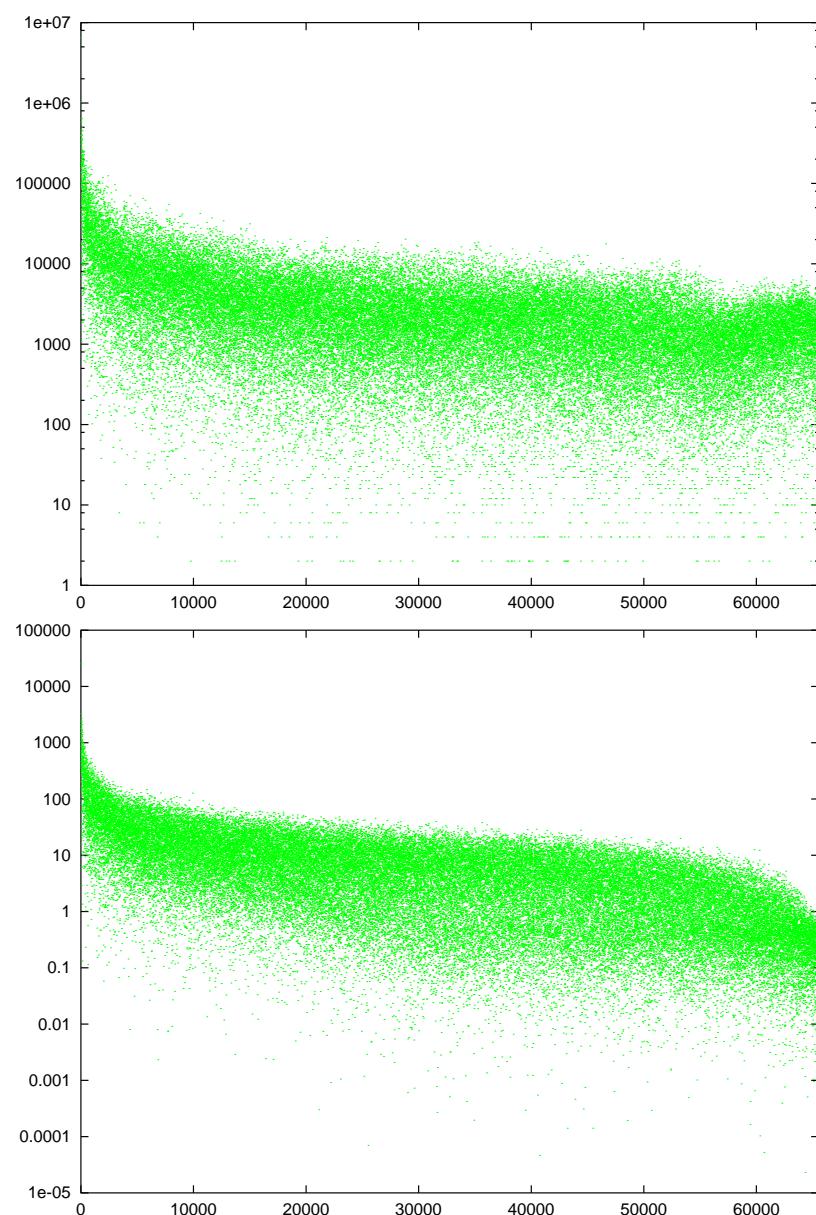


Figura 4.16: Magnitud de los coeficientes espectrales de Walsh-Hadamard y del coseno para “lena” ordenando según el filtro triangular.

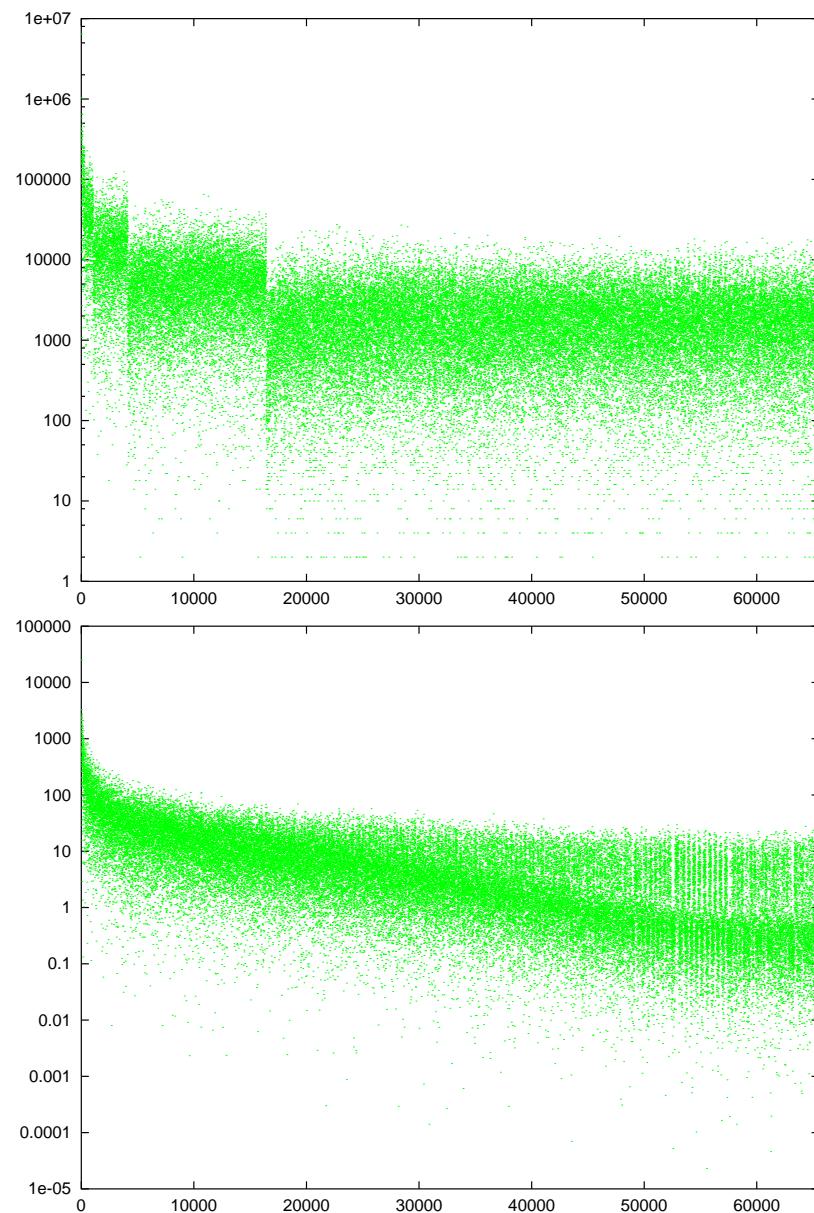


Figura 4.17: Magnitud de los coeficientes espectrales de Walsh-Hadamard y del coseno para “lena” ordenando según el filtro cuadrado.

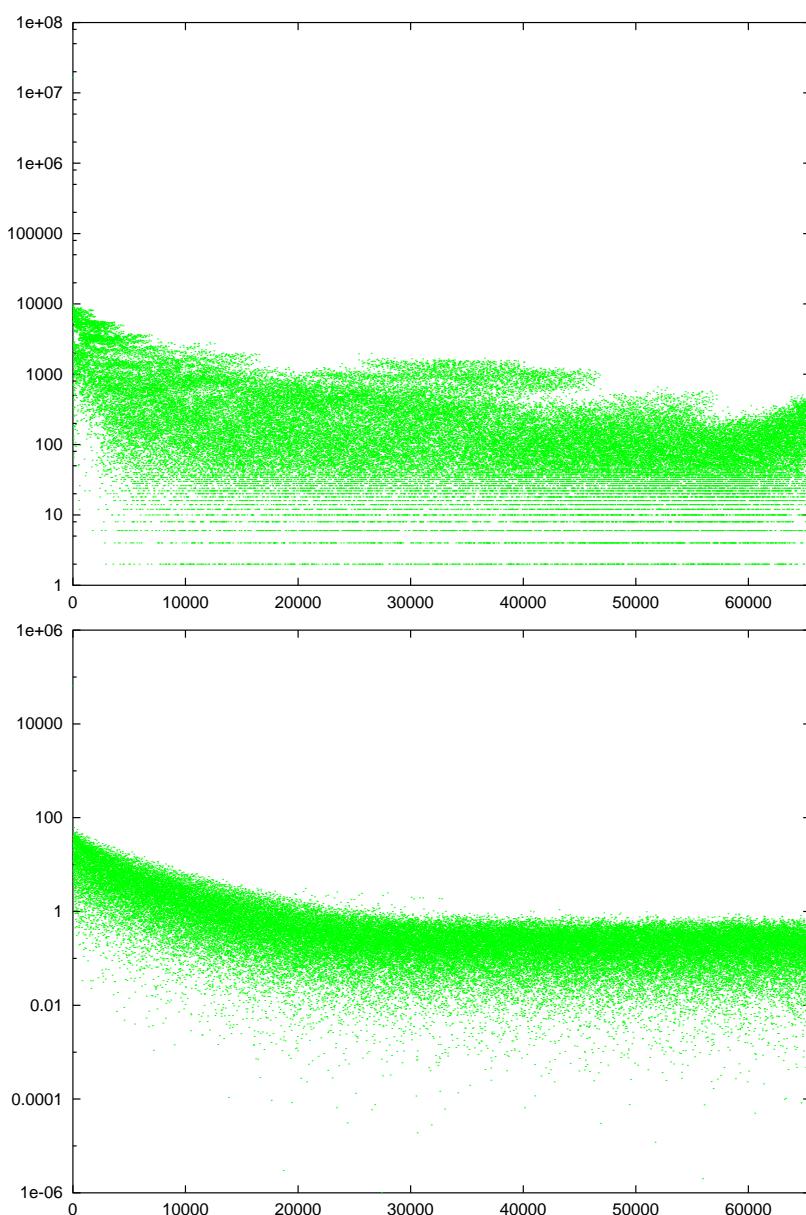


Figura 4.18: Magnitud de los coeficientes espectrales de Walsh-Hadamard y del coseno para una sección de 256×256 puntos situada en la región (512,512,768,768) de “B0100” ordenando según el filtro triangular.

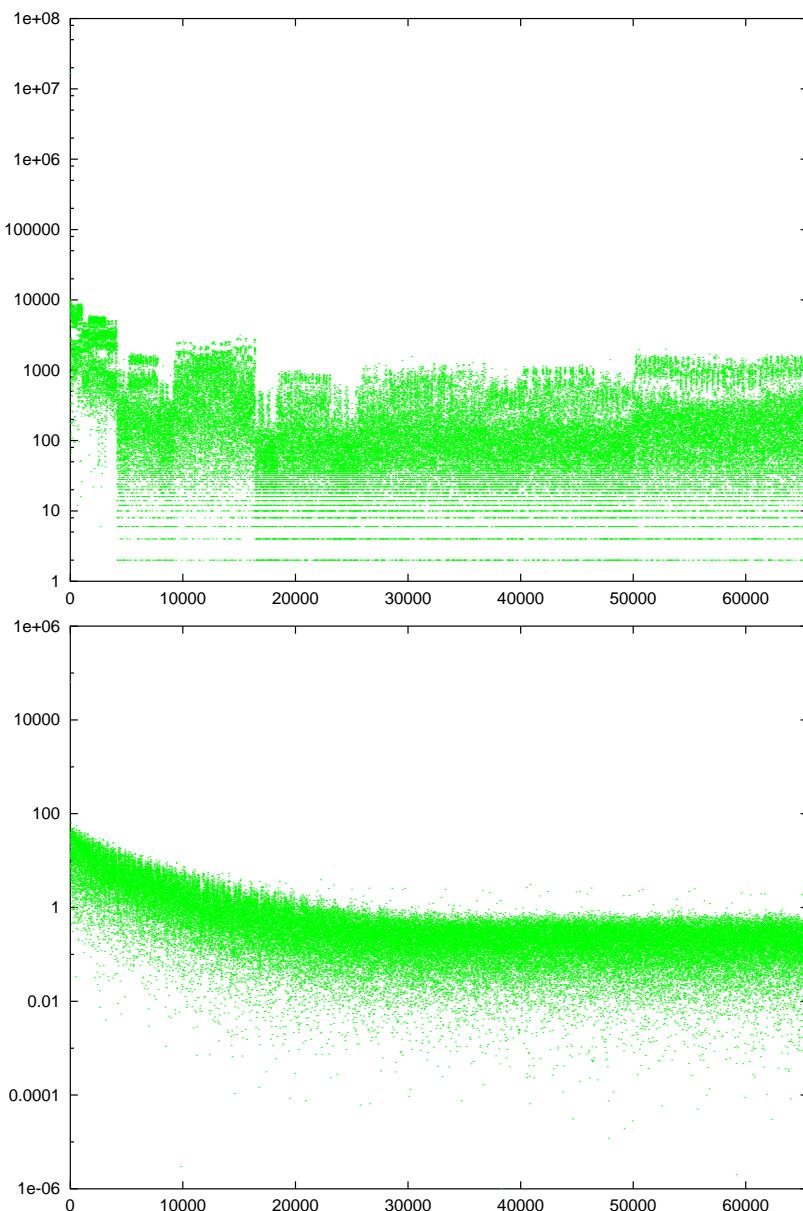


Figura 4.19: Magnitud de los coeficientes espectrales de Walsh-Hadamard y del coseno para una sección de 256×256 puntos situada en la región (512,512,768,768) de “B0100” ordenando según el filtro cuadrado.

para la DCT la función base es una función senoidal, pero en ambos casos se trata de funciones periódicas y no parece muy razonable buscar este tipo de señales estacionarias en la mayoría de las imágenes. Ya es conocido por la teoría de Fourier lo costoso que es representar una Delta de Dirac (lo más parecido a una estrella en el caso bidimensional) como una combinación lineal de funciones senoidales.

Por otra parte, aun contando con la existencia de algoritmos rápidos de cálculo de las transformaciones inversas que arrojan una complejidad de $O(N \times \log_2(N))$, el tiempo que el receptor emplea en la reconstrucción sucesiva puede ser excesivo. Esto puede repercutir (cuando la potencia computacional no es suficiente) en que el número de imágenes o cuadros presentados por unidad de tiempo al usuario sea excesivamente bajo y que por tanto, de lugar a una reconstrucción visualmente desagradable.

Es evidente que existen algunas mejoras. Como sabemos, la WHT es separable. El cálculo de la WHT-2D (inversa) implica calcular primero la WHT-1D (inversa) sobre las filas y a continuación sobre las columnas. Si sólo un coeficiente ha sido recibido, habría que recalcular únicamente una fila y a continuación todas las columnas. Cierta cantidad de tiempo extra puede ser ahorrado porque dentro de cada transformada unidimensional muchas de las mariposas (ver Apéndice B) no tienen que ser calculadas, lo que ya es una gran ventaja, pero nótese que la llegada de un coeficiente, potencialmente modifica a todos los puntos de la imagen. Esto es un gran inconveniente a la hora de disminuir el tiempo de reconstrucción.

La clave para resolver estos los problemas que han sido planteados es un cambio en el modelo de las funciones base. Como ya hemos indicado, la transformadas clásicas (DCT, WHT, DFT (*Discrete Fourier Transform*), etc.) poseen bases formadas por funciones ortogonales periódicas. Por suerte, existe un tipo de transformación diferente en la que las bases no son periódicas: la transformada *wavelet* (WT: *Wavelet Transform*) [22, 23].

Ésta se aprovecha de que en la mayoría de las imágenes de tono continuo, la existencia de señales estacionarias es poco probable. Ejemplos típicos y muy frecuentes en las imágenes son las fronteras y las texturas irregulares. Si somos capaces de usar funciones base localizadas en el espacio, el nivel de compactación espectral va a ser superior.

Pero si las funciones base son locales, ¿cómo es posible representar en función de ellas diferentes frecuencias espaciales?. La respuesta es sencilla: las *wavelets* se pueden desplazar (en nuestro caso a lo largo del dominio del espacio) y también se pueden escalar o dilatar (en otras palabras, estirar o comprimir). Así, cualquier señal, (especialmente si es local) puede ser representada como una combinación lineal de este tipo de señales. Las diferentes frecuencias se representan eficientemente controlando la escala de la *wavelet* y la posición de la señal se codifica adecuadamente mediante la operación de desplazamiento.

Como consecuencia, la WT es capaz de realizar un análisis frecuencial y además, a diferencia de la DFT por ejemplo, un análisis espacio-temporal, de forma simultánea.¹¹ En la WT, la localización de los coeficientes espectrales está determinada (conocemos por

¹¹El análisis temporal o espacial de una señal también es posible realizarlo usando la DFT, pero entonces hay que emplear ventanas temporales.

la posición del coeficiente en el espectro a qué zona de la imagen representa) y es posible establecer una correlación entre ellos, lo que nos va a ser de gran ayuda en la codificación de las coordenadas de los coeficientes cuando usemos la política de transmisión de máxima potencia. Por otra parte, la llegada de un nuevo coeficiente no implica la modificación forzosa de todos los puntos de la imagen reconstruida. Por último, la complejidad de la transformada directa e inversa es $O(N)$.

4.8 La transformada wavelet discreta

La atención que han recibido las *wavelets* en los últimos años es sencillamente abrumadora. Esto ha generado, tanto a nivel teórico como de aplicación (especialmente al procesamiento y compresión de señales) una línea de investigación muy importante.

En este capítulo vamos a analizar la transformada wavelet en su versión discreta (DWT: *Discrete WT*) de una forma muy intuitiva y tratando de que enlace con las principales ideas usadas en el diseño de compresores de señales continuas.

Como muchas de las transformadas usadas en compresión, su función es calcular un conjunto de coeficientes que indican la correlación que existe entre la señal a transformar y una serie de funciones base que forman un espacio ortogonal, tratando así de acumular en el mínimo número de coeficientes la máxima cantidad de información (energía), y si es posible, disminuir además la entropía de los datos.

Sin pérdida de generalidad, trataremos sólo por ahora el caso unidimensional. Sea $f[x], x = 0, 1, \dots, 2^n - 1$ el vector de muestras a transformar. Para encontrar una representación que cumpla el objetivo propuesto, vamos a emplear un modelo espacial como predictor, pero en lugar de usarlo de forma secuencial, procesando el vector desde un extremo a otro vamos usar dicho modelo para calcular primero una versión reducida de tamaño mitad de $f[x]$ a la que llamaremos $l[x], x = 0, 1, \dots, 2^{n-1} - 1$. La idea es usar $l[x]$ para generar un vector $\hat{f}[x]$ que sea tan parecido a $f[x]$ como sea posible. De esta forma, el vector residuo

$$h[x] = f[x] - \hat{f}[x]$$

contendrá la información que el modelo no es capaz de predecir.

Si el modelo de predicción es adecuado, las entropías de $h[x]$ y de $l[x]$ son inferiores a la de $f[x]$.¹² Por ejemplo, sea $f[] = \{1, 3, 2, 1\}$. Una posible forma de encontrar $l[]$ consiste en calcular la media de cada dos muestras: $l[] = \{2, 1.5\}$. A partir de este vector, una predicción que podemos hacer a cerca de $f[]$ es $\hat{f}[] = \{2, 2, 1.5, 1.5\}$, resultando un vector residuo $h[] = \{-1, 1, 0.5, -0.5\}$. Nótese que $h[]$ está formado por coeficientes que de dos en dos sólo se diferencian en el signo.

Como hemos visto, una forma de construir $l[x]$ es calcular la media de cada dos muestras

¹²Los elementos de $h[x]$ tienden a ser cero y en $l[x]$ el número de símbolos usados tiende a disminuir.

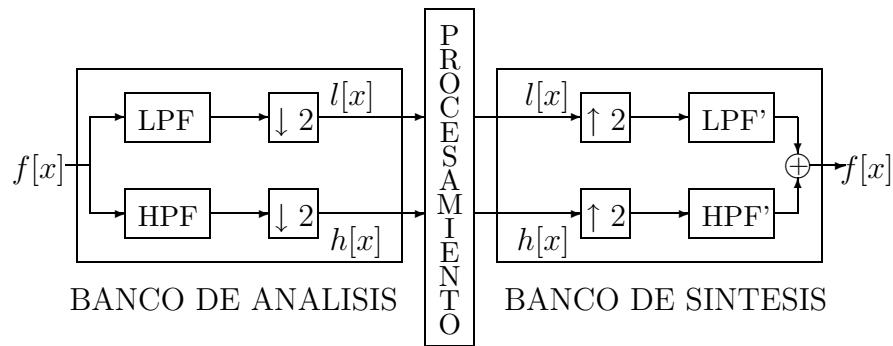


Figura 4.20: Descomposición frecuencial usando QMFs.

de $f[x]$, es decir

$$l[x] = \frac{f[2x] + f[2x+1]}{2}, x = 0, 1, \dots, 2^{n-1} - 1. \quad (4.13)$$

Si para generar $\hat{f}[x]$ usamos cada muestra de $l[x]$ dos veces de forma consecutiva, entonces sólo se necesitan la mitad de las muestras (las pares o las impares) de $h[x]$ para reconstruir $f[x]$. De hecho, $h[x]$ puede calcularse también usando

$$h[x] = \frac{f[2x+1] - f[2x]}{2}, x = 0, 1, \dots, 2^{n-1} - 1. \quad (4.14)$$

Las Ecuaciones 4.13 y 4.14 describen la transformada de Haar (HT: Haar Transform) [35]. En dicha transformada, se obtiene una representación alternativa para $f[x]$ formada por dos vectores de tamaño mitad: $l[x]$ y $h[x]$, que ocupan en conjunto el mismo espacio que $f[x]$. El vector $l[x]$ almacena la media de cada dos muestras de $f[x]$ y $h[x]$ almacena la diferencia entre cada par de muestras dividida entre dos, según el predictor usado en la transformada de Haar que indica que la siguiente muestra debería ser igual a la anterior. En este caso, en el ejemplo anterior obtendríamos $h[] = \{1, -0.5\}$. Nótese que $h[x]$ tiende a describir una distribución de Laplace.

Desde el punto de vista del procesamiento de señales [129], el proceso descrito coincide con la descomposición de una señal en dos bandas de frecuencia [113] (ver Figura 4.20). Una de ellas ($l[x]$) contiene la banda de baja frecuencia y la otra ($h[x]$) la banda de alta frecuencia. Esta descomposición puede ser aplicada de forma recursiva a la banda de baja frecuencia, generándose así lo que se conoce como DWT. Las Ecuaciones 4.13 y 4.14 describen los filtros FIR (*Finite Impulse Response*) que implementan la transformada de Haar [3, 24, 35, 79]. Estos dos filtros tienen una respuesta simétrica (ver Figura 4.21) y se llaman filtros espejo en cuadratura o QMFs (*Quadrature Mirror Filters*) [63]. Juntos forman un banco de dos canales (ver Figura 4.20). En el banco de filtros de análisis se realiza además un submuestreo de la señal (representado en la figura por $\downarrow 2$) tras el filtrado

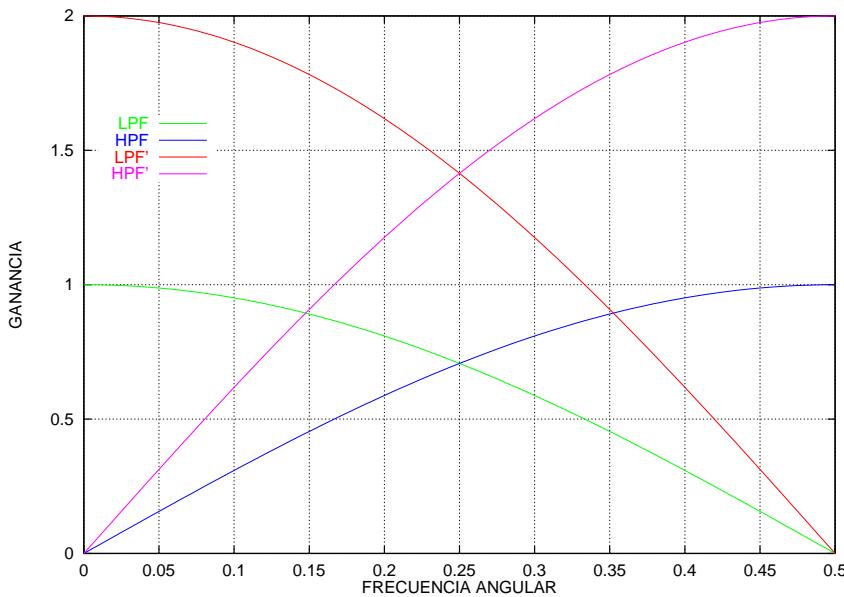


Figura 4.21: Respuesta en frecuencia de los filtros de la transformada de Haar. LPF=Low Pass Filter, HPF=High Pass Filter.

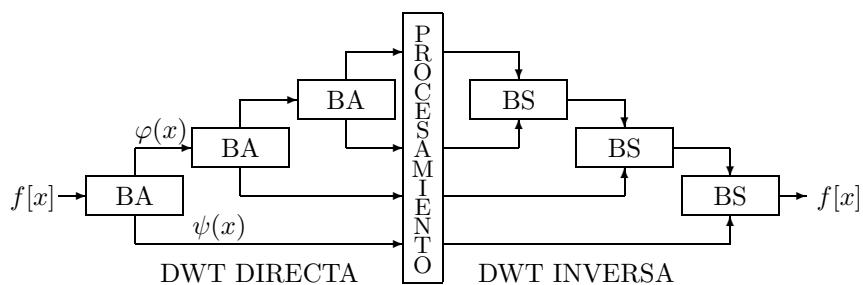


Figura 4.22: Cálculo de la DWT (dyadic) para el caso de una descomposición de 3 niveles. BA=Banco de Análisis, BS=Banco de síntesis.

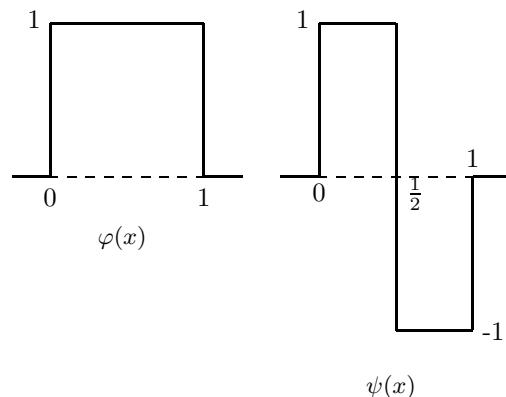


Figura 4.23: Funciones de escala y madre para la transformada de Haar.

y en el banco de filtros de síntesis (también llamados de interpolación) un sobremuestreo (representado por $\uparrow 2$) antes del filtrado.

La transformada directa se calcula aplicando la descomposición (el banco de análisis) n veces de forma recursiva sobre la banda de baja frecuencia. La transformada inversa se calcula aplicando el banco de síntesis también n veces según se describe en la Figura 4.22.

En la bibliografía relacionada [5, 24, 83], al filtro paso bajo de análisis también se le llama función de escala φ y al paso alto se le llama función base, función madre o wavelet ψ . La Figura 4.23 contiene ambas funciones para el caso de la transformada de Haar. En concreto,

$$\varphi(x) = \begin{cases} 1 & 0 \leq x \leq 1 \\ 0 & \text{en otro caso} \end{cases} \quad (4.15)$$

y

$$\psi(x) = \begin{cases} 1 & 0 \leq x < \frac{1}{2} \\ -1 & \frac{1}{2} \leq x < 1 \\ 0 & \text{en otro caso.} \end{cases} \quad (4.16)$$

Si analizamos el resultado de la transformación, el vector de coeficientes representa la correlación del vector de muestras con la función madre ψ a diferentes escalas de resolución y a diferentes desplazamientos, siempre múltiplos de dos. Por ejemplo, llamando $F[x]$ a la DWT de $f[x]$, en $F[0]$ tenemos la media de toda la señal. En $F[1]$ encontramos la diferencia entre la media de la primera mitad de las muestras y la media de la segunda mitad de las muestras. $F[0]$ representa por tanto a la componente de frecuencia 0 y $F[1]$ a la componente de frecuencia 1, cuando por frecuencia queremos decir una oscilación como la que describe la función madre ψ (ver Figura 4.23). Los coeficientes $F[2]$ y $F[3]$ forman la siguiente banda de frecuencia. $F[2]$ contiene la diferencia entre la media del primer cuarto de muestras y la media del segundo, mientras que $F[3]$ es la diferencia entre la media

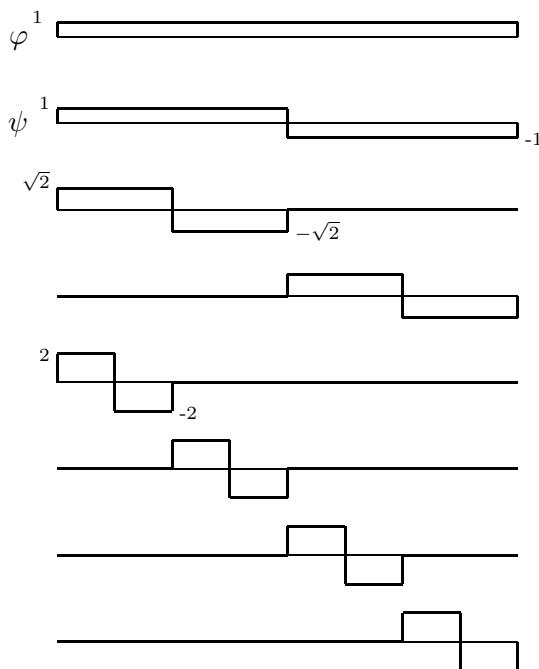


Figura 4.24: Funciones base de Haar, para el caso de 8 puntos (3 niveles). φ calcula la banda de baja frecuencia formada por la media de los 8 puntos. ψ calcula la banda de frecuencia 1. Las siguientes funciones base son aplicaciones recursivas de ψ sobre la banda de frecuencia inferior. Nótese que el número de componentes de frecuencia en cada banda se duplica (de aquí lo de *dyadic*) y que por tanto, el análisis frecuencial se hace por octavas.

del tercer cuarto y la media del último cuarto. Por lo tanto, ambas componentes hablan acerca de la misma frecuencia, pero sobre diferentes desplazamientos dentro de la señal (ver Figura 4.24).

A este modelo de descomposición en bandas de frecuencia se le conoce como *dyadic*. El número de bandas es n (excluyendo a la de frecuencia 0) y cada banda tiene el doble de coeficientes espectrales que la de frecuencia inmediatamente menor. La DWT analiza las señales por octavas (ver Figura 4.25)). Con este tipo de descomposición obtenemos una buena resolución frecuencial en las bajas frecuencias y una buena resolución espacio-temporal en las altas frecuencias.

Para terminar la descripción de la DWT *dyadic*, sólo queda decir que es muy rápida. Como es conocido [26, 82], el filtrado de una señal $f[x]$ puede realizarse convolucionando dicha señal con la respuesta del filtro al impulso unitario y este cálculo es de complejidad lineal $O(N)$ (donde $N = 2^n$) y proporcional a la longitud del filtro. Por lo tanto, la complejidad de la DWT sería $O(N \times \log_2(N))$, porque tenemos n etapas con $O(2^n)$. Sin embargo, ya que a la salida de los filtros de análisis existe un submuestreo, el cálculo de cada banda de frecuencia puede hacerse en la mitad de tiempo, convolucionando el filtro

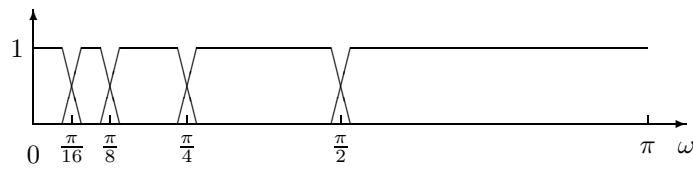


Figura 4.25: Descomposición espectral de una DWT *dyadic* de 4 niveles. ω es la frecuencia angular.

sólo con las muestras impares o pares.¹³ El resultado es que la DWT puede realizarse en $O(N)$ y como la longitud de los filtros suele ser bastante reducida, el algoritmo de la DWT *dyadic*¹⁴ (tanto directa como inversa) es extremadamente rápido.

4.9 La transformada S

La transformada S (ST: *Sequential Transform*) [11, 38, 81, 84, 104, 106] es semejante a la de Haar, pero evita la división en punto flotante (ver Ecuaciones (4.13) y (4.14)) lo cual es fundamental si se desea que la transformada sea totalmente reversible. Sólo de esta forma, la transformación inversa recupera siempre la señal original.

La transformada S directa para 2^n puntos se calcula aplicando los filtros de análisis

$$l[x] = \left\lfloor \frac{f[2x] + f[2x + 1]}{2} \right\rfloor \quad (4.17)$$

donde $\lfloor \cdot \rfloor$ representa al operador “el mayor entero menor que” y

$$h[x] = f[2x] - f[2x + 1]. \quad (4.18)$$

Los filtros de síntesis usados en la transformada inversa se encuentran fácilmente a partir de las Ecuaciones (4.17) y (4.18), resultando

$$f[2x] = l[x] + \left\lfloor \frac{h[x] + 1}{2} \right\rfloor \quad (4.19)$$

y

$$f[2x + 1] = f[2x] - h[x]. \quad (4.20)$$

La ST sustituye la división entre 2 por un desplazamiento de bits a la derecha, lo que aumenta también su velocidad de cálculo. Nótese que al igual que ocurre con la HT, la banda de baja frecuencia almacena la media de la señal y la banda de alta frecuencia

¹³Convolucionar y submuestrear por dos es equivalente a convolucionar sólo con la mitad de las muestras.

¹⁴En adelante, puesto que sólo se usa este tipo de DWT, omitiremos la palabra *dyadic*.

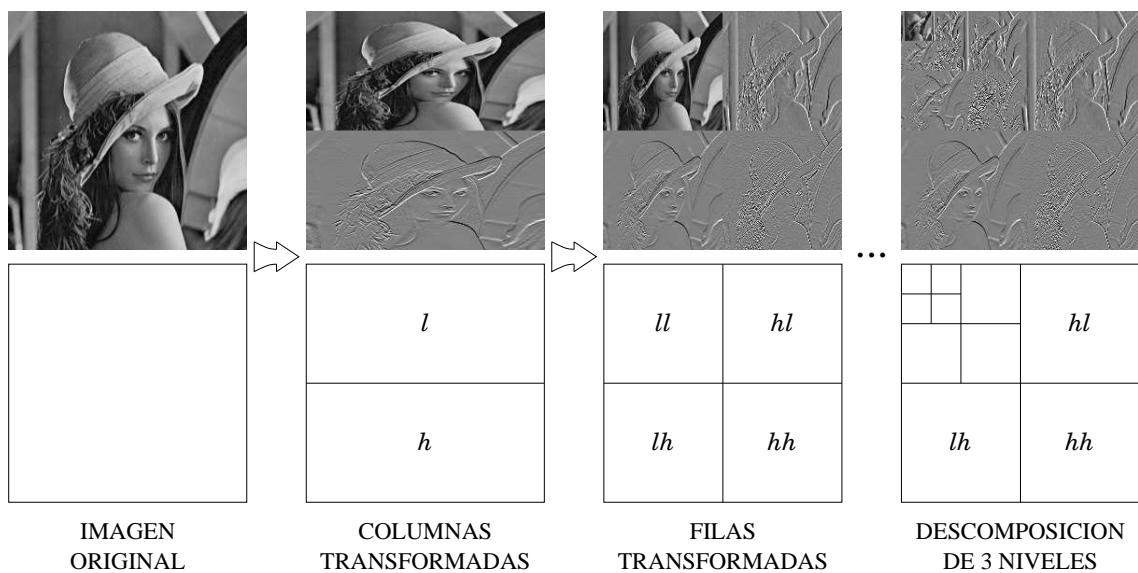


Figura 4.26: Cálculo de la DWT (*dyadic*) bidimensional. Primero se aplican los filtros de análisis sobre las columnas y a continuación sobre las filas. Cuando este proceso se realiza recursivamente tres veces sobre la banda de baja frecuencia, se genera la estructura de datos que se muestra a la derecha. Las bandas paso alto han sido amplificadas en un factor de 2 y desplazadas sumando 128 para que sean visibles.

representa los errores que resultan de predecir que las muestras impares son iguales a las pares, dos a dos. Por ejemplo, $h[0]$ sería igual a 0 si las dos primeras muestras de $f[x]$ fueran idénticas.

La ST sería perfecta ($h[x] = 0, \forall x$) si la señal se describiera totalmente a partir de una serie de escalones horizontales. Esta es una versión muy simplificada de una imagen, pero la ST hace un trabajo fino en aquellas regiones donde la señal varía suavemente. Por ejemplo, la entropía de la imagen “lena” es de 7.45 bits/punto. Cuando aplicamos la ST (usamos sucesivamente el filtro de análisis sobre la banda de baja frecuencia) la entropía se reduce hasta 4.82 bits/punto y el rango dinámico de la señal transformada sólo se ha multiplicado por 2 porque hay que representar el bit de signo de las diferencias (ver Ecuación (4.18)). Dicha propiedad ha sido utilizada para diseñar compresores de imágenes sin pérdida no progresivos [106, 114].

Además, la ST es separable. La transformada bidimensional se calcula aplicando los filtros unidimensionales, primero a las filas y luego a las columnas (o viceversa). La Figura 4.26 muestra el proceso de construcción de la DWT-2D.

La transformada inversa también es separable. Sin embargo, es necesario ver que la Ecuación (4.17) no es lineal debido a la operación de truncado, y esto hace que el orden de las transformaciones sea importante. Por ejemplo, si la transformación fue aplicada primero a las columnas y luego a las filas, la transformada inversa debe ser aplicada primero a las

filas y a continuación a las columnas.

4.9.1 La transformada S unitaria

La política de transmisión de los coeficientes ordenados por potencia minimiza las diferencias entre la imagen original y la transmitida. Este hecho puede comprobarse de forma visual en las Figuras 4.3 y siguientes, y numéricamente sobre las Tablas 4.1 y 4.2.

Desgraciadamente, cuando enviamos primero los coeficientes de máxima potencia, la transformada S tiende a transmitir primero las altas frecuencias frente a las bajas frecuencias. El resultado es que lo primero que el receptor recibe son las fronteras de la imagen. Este comportamiento se ha puesto de manifiesto en la Figura 4.27, que muestra la transmisión de las imágenes “lena” y “B0100”. Numéricamente, lo que está ocurriendo es que la relación señal/ruido entre la imagen transmitida y la original no está siendo minimizada.

La solución pasa por transmitir los coeficientes de baja frecuencia antes que los de alta frecuencia, a igualdad de condiciones, es decir, a igualdad de potencia. Para que la relación señal/ruido sea minimizada con la transmisión de los coeficientes por máxima potencia, los coeficientes transformados usando la ST deben ser ponderados por un factor [106]. Dichos factores dependen de la posición que ocupa el coeficiente dentro del espacio wavelet. Los factores que se usan son los que aparecen en la Figura 4.28.¹⁵ En la práctica se usan los factores que aparecen en la derecha de la Figura 4.28 porque conservan el orden establecido por la factorización ideal y permiten efectuar una factorización implícita de los coeficientes.

De esta forma, se tiende a enviar primero los coeficientes situados en la esquina superior izquierda que son los que representan a las bandas de baja frecuencia. La ponderación provoca que la transformada S sea unitaria y en consecuencia, ahora se cumple que el MSE (entre la imagen original y la imagen transmitida) se minimiza con la transmisión de cada coeficiente. Denotaremos en adelante a la transformada S unitaria por UST (*Unitary ST*). En adelante, trabajaremos siempre con una transformada wavelet unitaria por los motivos que se han expuesto.

4.10 La transformada S+P

La UST funciona muy bien si la imagen contiene regiones con valor constante, pero cuando no es así, causa un efecto llamado “*blocking*” (que podría ser traducido por “*losetizado*”) en ocasiones desagradable (ver Figura 4.27). Esto es consecuencia de que la mayoría de las imágenes de tono continuo no contienen grandes cantidades de rectángulos perfectos y por lo tanto, la correlación de la función madre de Haar¹⁶ bidimensional (ver Figura 4.23) con la imagen va a ser pobre. Por esta razón, para imágenes de tono continuo la UST da lugar a espectros con un gran número de componentes con niveles de energía altos. El resultado

¹⁵Nótese que la ponderación de los coeficientes sólo tiene efecto cuando se usa la política de selección espectral de máxima potencia.

¹⁶Recuérdese que la UST es una adaptación de la HT que usa aritmética entera.



Figura 4.27: Primera fila: Transmisión progresiva de “lena” usando la transformada S (ST). Se han transmitido 1000, 2000 y 3000 coeficientes espectrales. Segunda fila: lo mismo, pero usando la transformada ST unitaria (UST). Tercera fila: Transmisión progresiva de “B0100” (detalle) usando la transformada ST. Se han transmitido 100, 200 y 300 coeficientes espectrales. Cuarta fila: lo mismo, pero usando la transformada UST.

8	4	2		1			
4	2						
2		1					
1				$\frac{1}{2}$			
						2	
							1

16	8	4		2			
8	4						
4							
4		2					
				2			
						1	

Figura 4.28: Factores que multiplican los coeficientes espectrales y que aproximan la transformada S a una transformación unitaria. A la izquierda, la factorización ideal. A la derecha, la factorización usada en la práctica que es idéntica a la ideal salvo en un factor de escala. Los ejemplos están diseñados para una descomposición de 3 niveles.

es que se necesitan un numero elevado de coeficientes espectrales para una aproximación fidedigna de la imagen original.

Existe una forma sencilla de solucionar el problema, que consiste en elegir una función madre para la DWT que sea más suave que la utilizada en la UST. Esto equivale a modificar el esquema de predicción tan simple que usa la UST, por otro más complejo que sea capaz de disminuir la entropía de la banda de alta frecuencia y aumente así la concentración espectral.

En la transformada S+P ((S+P)T: *Sequential+Prediction Transform*) [104, 106] se aplica una fase de predicción sobre la ST. De esta forma, la banda de alta frecuencia se recalcula según la expresión

$$h_d[n] = h[n] - \left\lfloor \hat{h}[n] + \frac{1}{2} \right\rfloor, \quad (4.21)$$

donde $h_d[n]$ es la nueva banda de alta frecuencia, $h[n]$ la generada por la ST y $\hat{h}[n]$ la nueva predicción que responde a la expresión general

$$\hat{h}[n] = \sum_{i=-L_0}^{L_1} \alpha_i \Delta l[n+i] - \sum_{j=1}^H \beta_j h[n+j], \quad (4.22)$$

donde

$$\Delta l[n] = l[n-1] - l[n], \quad (4.23)$$

con el objetivo de que la media de la predicción sea 0 (téngase en cuenta que la banda paso bajo tiene una media igual a la media de la imagen y no tiene porque ser 0). L_0 ,

L_1 y H son valores enteros que determinan la ventana de vecindad usada como contexto de predicción. α_i y β_j son los coeficientes de predicción lineal. En su determinación, los autores [106] utilizaron una batería de imágenes estándar para determinar los valores óptimos de todos estos parámetros de los que depende el filtro.

En la Ecuación (4.22) se observa que la segunda fase de predicción utiliza la banda paso bajo y la paso alto generada tras la primera predicción (ST). Los coeficientes de predicción usados en nuestra implementación de la (S+P)T son $\alpha_0 = 2/8$, $\alpha_1 = 3/8$ y $\beta_1 = 2/8$. El predictor es por tanto

$$\hat{h}[n] = \frac{1}{8} \left\{ 2(\Delta l[n] + \Delta l[n+1] - h[n+1]) + \Delta l[n+1] \right\}. \quad (4.24)$$

En los bordes de la imagen, la predicción es

$$\begin{aligned} \hat{h}[0] &= \frac{\Delta l[1]}{4} \\ \hat{h}\left[\frac{N}{2} - 1\right] &= \frac{\Delta l\left[\frac{N}{2} - 1\right]}{4}. \end{aligned} \quad (4.25)$$

La (S+P)T presenta el mismo problema que la ST a la hora de sobrevalorar las altas frecuencias dentro de la imagen. Por lo tanto, cuando se trata de seleccionar los coeficientes atendiendo a su valor absoluto, el mismo esquema de ponderación usado en la UST puede usarse también para la (S+P)T. Llamaremos a la (S+P)T unitaria U(S+P)T.

Las Figuras 4.29 y 4.30 muestran la transmisión progresiva de las imágenes “lena” y “B0100” cuando usamos la UST y la U(S+P)T. Aunque con “B0100” no se puede apreciar correctamente porque la imagen es demasiado simple, la calidad de las reconstrucciones mejora notablemente con la U(S+P)T porque los efectos de “blocking” han desaparecido totalmente.

4.11 Evaluación de las transformadas en transmisión progresiva

Esta sección analiza el rendimiento de las cuatro transformadas anteriormente descritas: la transformada de Walsh-Hadamard (WHT), la transformada coseno (DCT), la transformada S unitaria (UST) y la transformada S+P unitaria (U(S+P)T).

Para la evaluación del rendimiento se ha usado la relación señal/ruido pico (PSNR), medida en dB, como medida de comparación, en función del número de coeficientes de la transformada que han sido transmitidos. Los resultados se muestran en las Tablas 4.3 y 4.4 para las imágenes “lena” y “B0100”, respectivamente.



Figura 4.29: Transmisión progresiva de “lena” usando la transformada S unitaria (UST) (izquierda) y la transformada S+P unitaria ($U(S+P)T$) (derecha). Se han transmitido 1000, 2000 y 3000 coeficientes espectrales. La política de selecciónpectral es por máxima potencia.

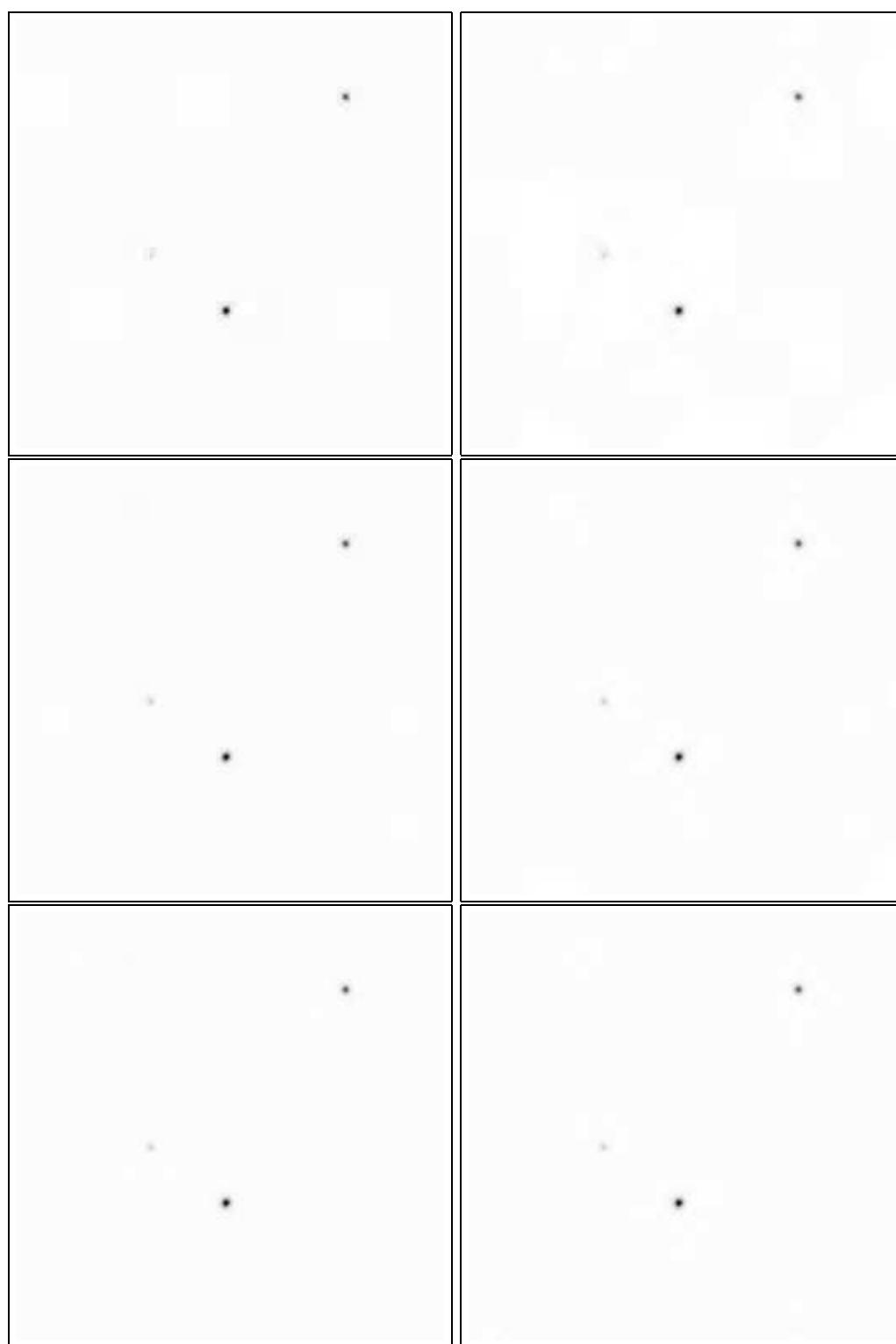


Figura 4.30: Transmisión progresiva de “B0100” usando la transformada S unitaria (UST) (izquierda) y la transformada $S+P$ unitaria ($U(S+P)T$) (derecha). Se han transmitido 100, 200 y 300 coeficientes espectrales. La política de selección espectral es por máxima potencia.

Tabla 4.3: Calidad de la reconstrucción en PSNR(dB) para “lena” cuando usamos la transformada de Walsh-Hadamard (WHT), coseno (DCT), S unitaria (UST) y S+P unitaria (S(S+P)T).

coeficientes transmitidos	transformada			
	WHT	DCT	UST	U(S+P)T
1000	22.20	24.29	23.55	25.71
2000	23.88	26.24	25.90	28.50
3000	24.99	27.58	27.58	30.41
4000	25.87	28.67	28.95	31.94
5000	26.63	29.61	30.11	33.28
6000	27.30	30.45	31.17	34.46
7000	27.91	31.24	32.12	35.49
8000	28.49	31.97	32.99	36.43
9000	29.03	32.67	33.80	37.31
10000	29.54	33.34	34.55	38.08
20000	33.95	39.47	40.58	43.49
30000	38.22	45.49	45.43	47.41

Los resultados son ligeramente diferentes dependiendo de la imagen. Lo más resaltable es la mejora que se produce respecto de la WHT y la DCT cuando usamos la UST o la U(S+P)T. Esta es la mejora típica que puede esperarse cuando en lugar de usar una transformada construida a partir de señales periódicas usamos otra diseñada con *wavelets*.

Las diferencias entre la UST y la U(S+P)T son considerables cuando la imagen contiene elementos suaves y redondeados, como ocurre con “lena”. En este caso, la correlación que existe entre la imagen y la wavelet es superior y por esta razón los PSNRs son mayores. Por el contrario, cuando la imagen contiene fundamentalmente altas frecuencias¹⁷, la diferencia entre la UST y la U(S+P)T es muy pequeña.

4.12 La transformada wavelet para un número de muestras distinto de 2^n

Hasta ahora, todas las transformaciones presentadas han sido definidas sobre un vector con un número de muestras igual a una potencia de dos. Como es lógico, en muchas situaciones vamos a transformar vectores que no cumplen dicha condición. En este apartado se describe la DWT sobre un vector de este tipo.

La transformada wavelet es una aplicación recursiva sobre la banda de baja frecuencia

¹⁷Los objetos que aparecen en “B0100” pueden ser muy bien descritos como Deltas de Dirac sobre un fondo plano.

Tabla 4.4: Calidad de la reconstrucción en PSNR(dB) para “B0100” cuando usamos la transformada de Walsh-Hadamard (WHT), coseno (DCT), S unitaria (UST) y S+P unitaria (U(S+P)T).

coeficientes transmitidos	transformada			
	WHT	DCT	UST	U(S+P)T
100	37.75	38.00	48.99	47.70
200	38.17	38.60	53.86	53.83
300	38.58	39.11	56.98	56.41
400	38.95	39.59	58.36	58.16
500	39.36	40.02	58.82	58.92
600	39.72	40.44	59.19	58.99
700	40.05	40.83	59.31	59.16
800	40.39	41.19	59.42	59.21
900	40.75	41.54	59.48	59.24
1000	41.04	41.87	59.54	59.23
2000	43.09	44.48	61.14	60.91
3000	44.35	46.35	61.93	61.63

de un banco de filtros de dos canales. Por esta razón, comentaremos brevemente cómo transformar un vector de tamaño impar, que es la clave para transformar vectores de cualquier tamaño.

Sea $f[x]$ el vector de muestras a transformar, donde $x = 0, 1, \dots, N$ y N es un número par. Pensemos en que $f[]$ tuviera una muestra más igual a la última, es decir, que existiera un $f[N + 1] = f[N]$ para que el número de muestras fuera un número par. Bajo estas circunstancias, el $\lfloor N/2 \rfloor$ -ésimo elemento de $l[]$ (la banda paso bajo) será igual a $f[N]$ porque la media de 2 números iguales es cualesquiera de los dos números.

Por otra parte, el $\lfloor N/2 \rfloor$ -ésimo elemento de $h[]$ (la banda paso alto) será igual a 0 ya que se calcula como la diferencia entre $f[N]$ y $f[N + 1]$, que son iguales. Como $h[\lfloor N/2 \rfloor]$ es siempre 0, no es necesario representarlo. De esta forma, el número de elementos de $l[]$ es $\lfloor N/2 \rfloor + 1$ y el número elementos de $h[]$ es $\lfloor N/2 \rfloor$, lo que en total suma N y por lo tanto, la transformación directa puede realizarse “in-place”.

El procedimiento de cálculo que acaba de ser descrito se corresponde realmente con el de la transformada S. Para realizar la transformación S+P habría que aplicar a continuación la fase de predicción sobre $h[]$.

La transformación inversa se construye aplicando la transformada inversa estándar a los $\lfloor N/2 \rfloor$ primeros elementos de $l[]$ y a todo $h[]$. A continuación $f[N]$ se hace igual a $l[\lfloor N/2 \rfloor]$. La transformación inversa también es “in-place”.

4.13 Transmisión por planos de bits

La transmisión incremental de imágenes puede ser mejorada si en lugar de enviar los coeficientes completos se transmite por planos de bits. Si p es la posición que ocupa el bit más significativo del coeficiente más grande, la idea es transmitir primero todos los bits que están en la posición p de todos los coeficientes. A continuación se envían todos los bits que están en la posición $p - 1$ y así sucesivamente hasta enviar todos los planos. Éste es un razonamiento lógico si se tiene en cuenta que es más importante indicar al receptor los bits más significativos de los coeficientes menores que los bits menos significativos de los coeficientes mayores.

Para aclarar dicho concepto, supongamos que la transformada wavelet de una imagen ha generado sólo dos coeficientes distintos de cero. El primero es 8192 (10 0000 0000 0000 en binario) y el segundo 4095 (1111 1111 1111). Supongamos que transmitimos sólo los bits que son necesarios para especificar cualquier coeficiente. En este caso, habría que comunicar al receptor que el coeficiente más grande se puede codificar con 14 bits. Esto equivale a decir que el bit más significativo del coeficiente mayor está en el plano 14 del coeficiente.

Si transmitimos los coeficientes completos, la secuencia de bits enviada es: 10 0000 0000 0000 - 00 1111 1111 1111 (empleamos el - para delimitar la transmisión de 14 bits consecutivos que es la única regla que puede seguir el receptor para distinguir un coeficiente de otro). Con la recepción de primer bit, el MSE entre la imagen original y la transmitida se decrementaría en $8192^2/N$ (donde N es el número de puntos que existen en la imagen). Con el siguiente bit el MSE no se decrementaría porque el bit transmitido es 0. El MSE permanecería constante hasta que el bit localizado en el plano 12 del segundo coeficiente fuera recibido y el MSE se reduciría en $4096^2/N$. A continuación, con la llegada de cada bit del segundo coeficiente el MSE se va reduciendo hasta valer 0.

Cuando transmitimos los coeficientes por planos de bit, la secuencia enviada es el barajamiento entre ambos coeficientes: 1000 0101010101 01010101 01010101.¹⁸ Con la recepción del primer bit, el MSE se reduce en $8192^2/N$. Hasta el quinto bit esta situación continua, pero con el sexto bit recibido el MSE se decremente en $4096^2/N$. Si recordamos, para conseguir una imagen reconstruida idéntica, el método anterior necesitaba transmitir 17 bits. Por lo tanto, la transmisión por planos de bits es mucho más eficiente.

Las Figuras 4.31 y 4.32 muestran los planos de bits de la transformada U(S+P) y las reconstrucciones en el receptor para “lena” y (la sección de) “B0100”. Un punto blanco significa un bit igual a 0 y un punto negro un bit igual a 1.

El número total de planos es 16 y por eso se muestran 16 reconstrucciones. La factorización provoca que los primeros coeficientes significativos (aquellos que en valor absoluto son mayores o igual que 2^p donde p es el plano de bits transmitido) estén en la esquina superior izquierda que es la zona que corresponde a las bajas frecuencias.

¹⁸Hemos formado grupos de cuatro bits a partir de dos grupos de dos bits y grupos de ocho bits a partir de dos grupos de cuatro bits.



Figura 4.31: Transmisión progresiva de “lena” por planos de bits usando la U(S+P)T.

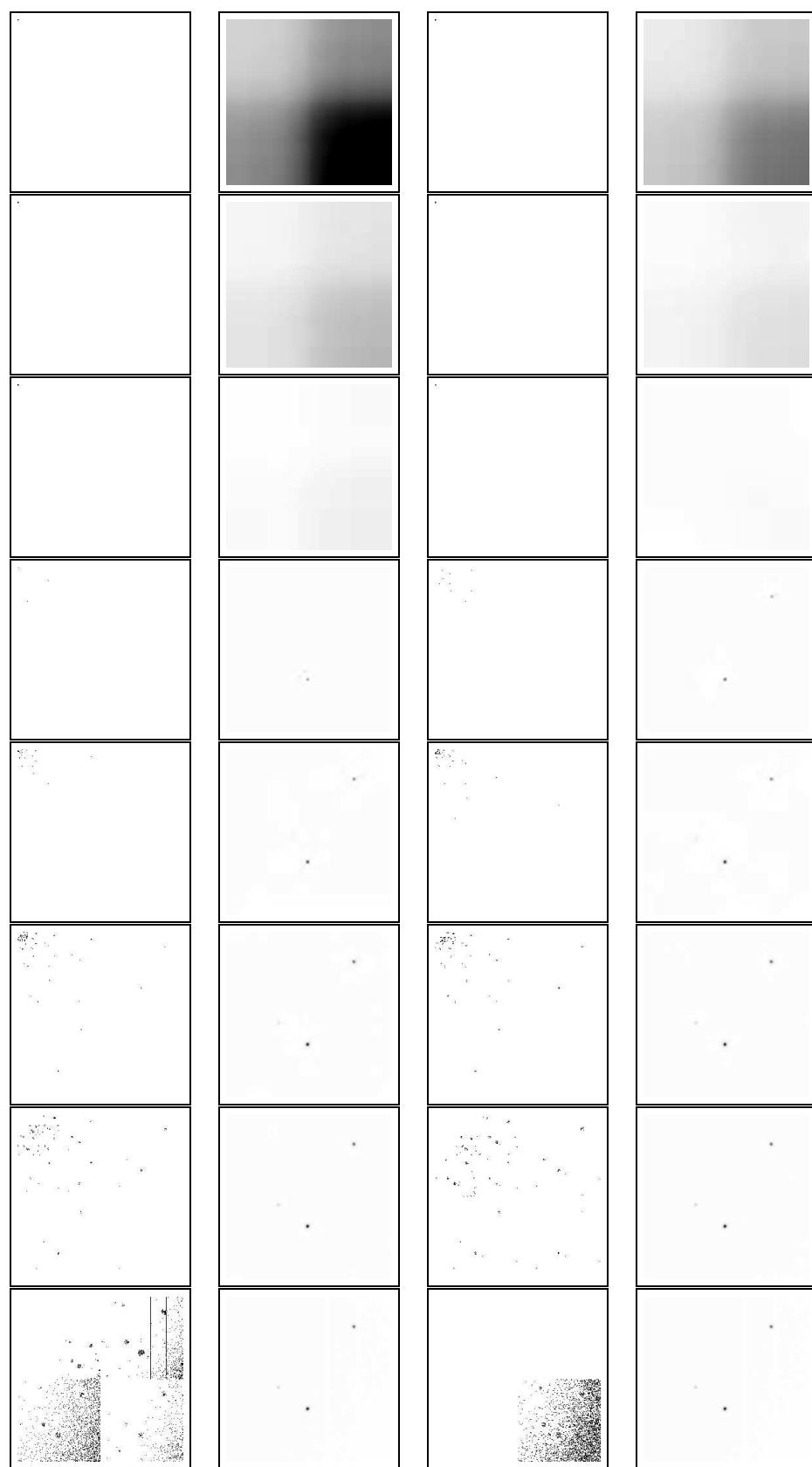


Figura 4.32: Transmisión progresiva de “B0100” por planos de bits usando la U(S+P)T.

4.14 Compresión de los planos de bits

La compresión de los planos de bits es necesaria si se desea optimizar la transmisión. Observando de nuevo las Figuras 4.31 y 4.32 se puede apreciar que, especialmente al comienzo de la transmisión, cuando los primeros planos de bits son enviados, la cantidad de bits iguales a cero es muy superior al de unos. Podemos aprovechar esta circunstancia para comprimir los planos de bits.

Una primera idea podría consistir en enviar explícitamente sólo las coordenadas de aquellos bits que son 1, porque sabemos que el resto es 0. Este procedimiento puede dar buen resultado cuando el número de bits a 1 es muy bajo, pero rápidamente necesitaremos enviar largas listas de coordenadas y debe tenerse en cuenta que son necesarios muchos bits para indicar cada una de ellas.

La clave para determinar un método efectivo de codificación de los planos de bits está en darse cuenta de que existe una cierta dependencia entre los coeficientes dentro del espacio wavelet. Si volvemos a mirar las Figuras 4.31 y 4.32 averiguaremos que existe bastante parecido entre dos octavas de niveles (de frecuencia) consecutivos y entre las tres octavas de un mismo nivel (misma frecuencia). Esta correlación entre coeficientes puede expresarse diciendo que si un coeficiente es significativo en un nivel de frecuencia superior, entonces los 4 coeficientes que ocupan la misma posición relativa dentro del nivel inferior tienden a ser significativos, y viceversa. Decimos por tanto que la DWT-2D posee autosimilaridad entre octavas de una misma frecuencia y entre octavas de frecuencia diferentes.

La relación espacial exacta entre un coeficiente de coordenadas (i, j) ¹⁹ y sus cuatro hijos (si es que existen), es que estos se encuentran en las coordenadas absolutas

$$\mathcal{O}(i, j) = \{(2i, 2j), (2i, 2j + 1), (2i + 1, 2j), (2i + 1, 2j + 1)\} \quad (4.26)$$

y esta relación se propaga de forma recursiva entre todos los coeficientes wavelet. Por lo tanto, un nodo (i, j) del árbol tiene 4 hijos o no tiene ninguno, lo que ocurre en el nivel más bajo de la descomposición. Para aclarar este concepto de dependencia espacial entre coeficientes, la Figura 4.33 muestra algunos ejemplos de árboles cuaternarios. A cada árbol diseñado usando la Expresión 4.26 se le llama “árbol de orientación espacial” (AOE).

A todos los descendientes de (i, j) (hijos, nietos, etc) los denotaremos por $\mathcal{D}(i, j)$. El número total de descendientes es 0 o una potencia de 4 cuyo exponente depende del nivel de la descomposición wavelet. Nótese que $\mathcal{D}(i, j) + (i, j)$ forman un AOE completo.

La relación estadística entre un elemento (i, j) y sus descendientes $\mathcal{D}(i, j)$ es que si (i, j) es 1 entonces al menos uno de sus descendientes es probablemente 1. Por el contrario, si (i, j) es 0, lo más probable es que todos sus descendientes sean 0. Esta es la principal razón por la que podemos diseñar compresores a partir del espacio wavelet.

Como los bits de un AOE están correlacionados, se puede idear un método de codificación que explote dicha correlación. Quizás la forma más sencilla de codificación consista en indicar con un bit si el AOE está formado sólo por ceros (decimos entonces que se trata

¹⁹Las coordenadas se expresan como (fila, columna).

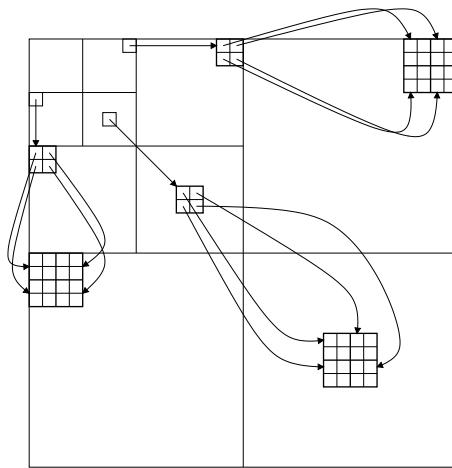


Figura 4.33: Algunos ejemplos de árboles creados en la DWT-2D. Se forman de modo que los hijos del coeficiente (i, j) están en las coordenadas $(2i, 2j)$, $(2i + 1, 2j)$, $(2i, 2j + 1)$ y $(2i + 1, 2j + 1)$.

de un zerotree) o si no es así. Cuando si lo es, un único bit informa al descodificador de que todo el AOE es cero. Si no lo es, podemos aprovecharnos de la definición recursiva que posee un AOE para indicar por separado cuál de los cuatro hijos es un zerotree y descender hasta el nivel que sea necesario para comunicar al descodificador la composición exacta del AOE.

En la Figura 4.34 se muestra un ejemplo con un plano de bits típico. El número de coeficientes wavelet es 8×8 por lo que la descomposición tiene tres niveles. Viendo el ejemplo nos damos cuenta que no todos los coeficientes pueden tener descendientes. Puesto que existen sólo 3 bandas de frecuencia (lh , hl y hh , ver Figura 4.26), el coeficiente $(0,0)$ no tiene hijos. Sólo los coeficientes $(0,1)$, $(1,0)$ y $(1,1)$ poseen descendientes que forman un AOE.

4.14.1 Un método de compresión

Vamos a suponer el siguiente método de codificación sencillo. En primer lugar transmitiremos explícitamente los bits del nivel (wavelet) más alto. Dentro de cada nivel seguiremos el orden establecido en la Figura 4.34 (derecha). De esta forma, emitiremos la secuencia de bits 1001. Seguidamente emitiremos los bits necesarios para indicar el resto de los tres AOE: $\mathcal{D}(0, 1)$, $\mathcal{D}(1, 0)$ y $\mathcal{D}(1, 1)$. Seguiremos también el mismo orden que se ha usado para referenciar a los bits del nivel más alto. En primer lugar, codificamos el AOE situado en las coordenadas $(0,1)$. Como todos los descendientes de este coeficiente son 0, emitimos un único bit igual a 0 indicando esta circunstancia. Luego procesamos el AOE $(1,0)$. Este contiene un bit 1 en la posición $(3,0)$ y por lo tanto transmitimos un 1. El descodificador conoce ahora que el AOE no es un zerotree. Descendemos por el AOE y transmitimos

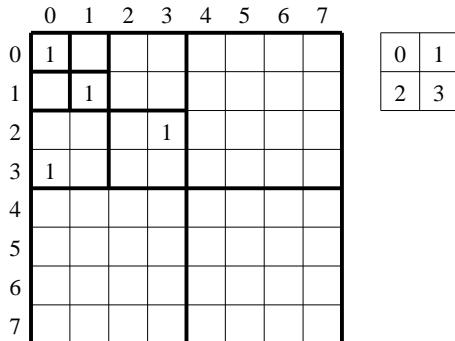


Figura 4.34: Un ejemplo con un plano de bits típico casi al comienzo de la transmisión. Las casillas vacías contienen bits a 0. La imagen transformada es de 8×8 puntos y la descomposición wavelet tiene 3 niveles. A la derecha un orden cualquiera de recorrido de un bloque de 4 coeficientes wavelet.

0010. De esta forma, el descodificador conoce el contenido de los bits (2,0), (2,1), (3,0) y (3,1). Ahora queda transmitir si los AOE situados en dichas posiciones son zerotrees. Como sí lo son, enviamos a continuación 0000. Por último, queda comunicar el AOE (1,1). Como no es un zerotree transmitimos un 1, luego 0100 y después 0000. Si el descodificador conoce el orden que se ha seguido para recorrer los AOE, la descodificación es posible. En este ejemplo, hemos comprimido 64 bits en 23 bits.

4.14.2 Bits de significancia y de refinamiento

Si el plano de bits transmitido en el anterior ejemplo ha sido el p -ésimo, cuando transmitamos el $(p - 1)$ -ésimo, aparecerán posiblemente nuevos coeficientes significativos diferentes de los que aparecen en la Figura 4.34. La misión del compresor va a consistir en indicar la posición de estos nuevos coeficientes que contienen un 1 en el plano $p - 1$. Pero no podemos olvidar que además existirán otros bits a 1 que pertenecen al plano $p - 1$ de los coeficientes que fueron significativos en el plano p . Existen, por tanto, dos tipos distintos de bits dentro de cada plano:

- Los bits de significancia que corresponden a aquellos coeficientes que comienzan a ser significativos en el plano actualmente transmitido.
- Los bits de refinamiento que están formados por los bits de los coeficientes que ya son significativos en un plano superior. Se llaman de esta forma porque lo que hacen es afinar el valor real del coeficiente wavelet.

Como ejemplo, en la Figura 4.35 muestra los bits de significancia y de refinamiento para el cuarto plano de bits de “lena”. La unión de ambos tipos de bits forman el cuarto plano de bits que se presenta en la Figura 4.31.



Figura 4.35: Cuarto plano de significancia (izquierda) y refinamiento (centro) y refinamiento que son 1 (derecha), para “lena”.

El tratamiento que deben recibir ambos tipos de bits es diferente. Los bits de refinamiento (independientemente de si son 0 o 1, ver Figura 4.35-centro) son transmitidos con cada plano de bits. La correlación estadística entre los diferentes planos de bits de un mismo coeficiente es muy baja, lo que deriva en bajas ganancias de compresión si se elimina. Sin embargo, los bits de significancia tienen grandes posibilidades de ser predichos en cada plano porque se sitúan como hijos y hermanos de los bits de refinamiento. Es en su codificación donde realmente podemos efectuar un trabajo de compresión importante.

4.14.3 Acerca del bit de signo

Como recordaremos, como paso previo a la codificación de los planos de bits hemos calculado el valor absoluto de cada uno de los coeficientes wavelet porque si se representaran en complemento a dos, el bit de signo negativo provocaría unos en muchos de los planos superiores, lo que es un inconveniente.

Los bits de signo son muy importantes para la reconstrucción de la imagen en el receptor y deben ser transmitidos en cuanto son necesarios. La técnica más común es enviar el bit de signo asociado a cada bit de significancia que es transmitido. Como es evidente, cada bit de signo se envía una única vez porque los bits de significancia nunca se repiten en planos de bits diferentes.

La Figura 4.36 muestra los bits de signo de los coeficientes wavelet para la imagen “lena” y “B0100”. En el caso de “lena”, la correlación espacial entre ellos parece ser bastante reducida por lo que no son esperables grandes ganancias en la tasa de compresión si este plano es codificado entrópicamente. Tampoco existe una dependencia apreciable entre la magnitud o la significancia del coeficiente y su signo lo que provocaría una cierta adivinación de la forma de los niveles de la descomposición wavelet. En el caso de “B0100” tampoco existe una correlación clara entre los objetos de la escena y los signos de los coeficientes. Sin embargo, en esta imagen ocurre que como casi todo el fondo es prácticamente plano, el bit de signo resulta bastante uniforme también. En este caso tal vez sería útil

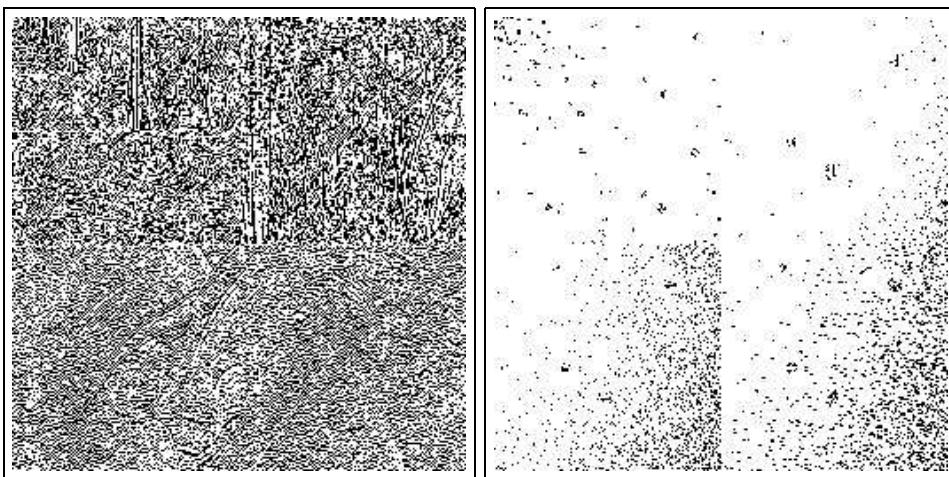


Figura 4.36: Bits de signo de “lena” y “B0100” para $U(S+P)T$

una codificación de orden 0 puesto que el signo positivo es mucho más frecuente que el negativo.

4.15 Un codec progresivo de coeficientes wavelet: SPIHT

Una vez presentada el tipo de redundancia que es posible explotar en el dominio wavelet cuando los coeficientes son transmitidos por planos, vamos a analizar uno de los compresores progresivos más eficientes que se conocen: SPIHT (*Set Partitioning In Hierarchical Trees*) [103, 105].

4.15.1 Un poco de historia

SPIHT fue ideado por Said y Pearlman en 1996 y tiene sus raíces en un algoritmo llamado EZW (*Embedded Zerotree Wavelet*) [110] que fue diseñado por Shapiro en 1993. EZW usa un código aritmético binario y un alfabeto de 4 símbolos para indicar de la forma más compacta posible al descodificador la situación de los zerotrees. EZW es considerado un punto de inflexión en la curva creciente que relaciona la tasa de compresión con el coste computacional de los algoritmos. Desde entonces se ha producido un gran esfuerzo por diseñar algoritmos más eficaces [12, 14, 47, 54, 73, 114, 133, 134] y estandarizaciones como la de JPEG 2000 [50] o CREW (*Compression with Reversible Embedded Wavelets*) de la compañía RICOH [1]. Todos ellos poseen el denominador común de usar un código de longitud variable (casi siempre codificación aritmética) para comprimir entrópicamente aprovechando la redundancia remanente en la DWT-2D. En este sentido, todos estos trabajos son una aplicación de la idea de la codificación universal introducida por Rissanen y Langdon años atrás con la aparición de la codificación aritmética [89, 91, 119].

Sin embargo SPIHT es conceptualmente diferente porque usa un código de longitud fija. La longitud de cada palabra de código es de un bit y esta codificación es eficiente porque la probabilidad de cada símbolo (que coincide con la de cada bit o palabra de código) es 0.5. Su existencia demuestra que es posible diseñar codecs eficientes sin la necesidad de diferenciar entre un modelo probabilístico y un codificador de longitud variable,²⁰ y constituye un buen ejemplo del concepto de codificación universal introducido en la Sección 2.3.1. Además, SPIHT cumple todos los requisitos expuestos en la Sección 4.1.

4.15.2 Funcionamiento de SPIHT

El codificador SPIHT codifica eficientemente los planos de significancia, aprovechando la similaridad o semejanza que existe entre las diferentes escalas, aunque no explota la redundancia que pueda existir entre las tres bandas de una misma escala.

Lo primero que hace es transmitir el plano de bits más significativo (los coeficientes se tratan en forma signo-magnitud). Después entra en un bucle que itera tantas veces como planos de bits tienen que transmitirse. En la emisión de cada plano de bits diferencia entre los bits de significancia (que son comprimidos) y los bits de refinamiento (que no son comprimidos). Los bits de signo son también transmitidos sin comprimir.

SPIHT trabaja particionando los AOEs de forma que tiende a mantener coeficientes no significativos en grandes conjuntos y comunica con un único bit si alguno de los elementos de un conjunto es significativo o no. Esta es la diferencia fundamental entre SPIHT y el método de compresión presentado en la Sección 4.14.1. SPIHT con un único bit es capaz de transmitir la situación de muchos zerotrees mientras que nuestro método usa un bit para cada uno de ellos.

Las decisiones de particionamiento son decisiones binarias que son transmitidas al descodificador e indican los planos de significancia. El particionamiento es tan eficiente que la codificación aritmética binaria de las decisiones sólo provoca una pequeña ganancia. Esto es consecuencia directa de que SPIHT efectúa un número de particionamientos mínimo de forma que la probabilidad de encontrar un coeficiente significativo en cada conjunto es aproximadamente igual a la probabilidad de no encontrarlo.

Por lo tanto, SPIHT en lugar de transmitir las coordenadas de los coeficientes significativos en el plano actual, transmite los resultados de las comparaciones que han provocado la determinación (por parte del codificador) de todos los zerotrees que forman dicho plano. El descodificador no necesita más información para saber que el resto de coeficientes que no pertenecen a ningún zerotree son iguales a 1. De hecho, el descodificador ejecuta exactamente el mismo algoritmo que el codificador y como no dispone de los coeficientes wavelet para saber si son significativos o no, usa los resultados de las comparaciones que le llegan en el *code-stream*. De esta forma la traza de instrucciones es idéntica.

²⁰Al igual que ocurre con la familia de compresores LZ (ver Sección 2.2).

4.15.3 Detalles de la implementación

SPIHT transmite los planos de significancia y de refinamiento en dos fases independientes llamadas de ordenación y de refinamiento, respectivamente. El nombre de la fase de refinamiento es obvio. Sin embargo la fase de ordenación se llama así porque lo que el codificador hace es ordenar los coeficientes atendiendo a su valor absoluto y luego enviarlos según ha resultado dicho orden. Pero nótese que la ordenación que se produce es muy suave ya que no es necesario ordenar todos los coeficientes que son significativos en el plano que se transmite, sólo debemos encontrar qué coeficientes van antes que otros porque son significativos en un plano superior.

La fase de ordenación

Como ya hemos indicado, SPIHT es eficiente porque realiza un número mínimo de comparaciones equiprobables. La clave está en saber cómo formular dichas comparaciones, que se construyen usando un algoritmo de particionamiento de los AOEs.

El codificador y el descodificador manejan conceptualmente dos listas de coeficientes representados por sus coordenadas espaciales. En una se almacenan todos los coeficientes que son significativos en el plano actual de transmisión p . Todos estos coeficientes c verifican que

$$|c| \geq 2^p. \quad (4.27)$$

La otra lista almacena el resto de coeficientes que no son significativos. Inicialmente esta lista contiene todos los coeficientes (tantos como puntos existen en la imagen) y la lista de coeficientes significativos está vacía. Notaremos con LIC (*List of Insignificant Coefficients*) a la lista de coeficientes no significativos y con LSC (*List of Significant Coefficients*) a la de coeficientes significativos.

SPIHT realiza una partición inicial

$$\{(0, 0), (0, 1), (1, 0), (1, 1), \mathcal{D}(0, 1), \mathcal{D}(1, 0), \mathcal{D}(1, 1)\},$$

donde como ya sabemos, $\mathcal{D}(i, j)$ representa a todos los coeficientes descendientes de (i, j) que se determinan aplicando la Ecuación (4.26) recursivamente.

SPIHT averigua qué elementos de esta partición son significativos. Más formalmente, evalúa la función

$$S_p(\mathcal{T}) = \begin{cases} 1 & \text{si algún coeficiente } c \in \mathcal{T} \geq 2^p \\ 0 & \text{en caso contrario,} \end{cases} \quad (4.28)$$

donde \mathcal{T} puede ser un único coeficiente o un conjunto de coeficientes.

En la codificación del primer plano de bits, las 4 raíces

$$\{(0, 0), (0, 1), (1, 0), (1, 1)\}$$

tienen un 50% de posibilidades de ser significativas y a sus descendientes

$$\{\mathcal{D}(0, 1), \mathcal{D}(1, 0), \mathcal{D}(1, 1)\}$$

((0, 0) no tiene descendientes) les ocurre lo mismo, tienen una probabilidad igual a 0.5 (aproximadamente) de ser un zerotree. SPIHT realiza todas estas comparaciones y emite los bits de código correspondientes.

Para gestionar las particiones, SPIHT usa realmente 3 listas: LIC, LSC y LIS (*List of Insignificant Sets*) o lista de conjuntos no significativos, porque es una forma sencilla de distinguir entre coeficientes y conjuntos de coeficientes. Por lo tanto, el contenido inicial de dichas listas es:

$$\begin{aligned} \text{LSC} &\leftarrow \emptyset \\ \text{LIC} &\leftarrow \{(0, 0), (0, 1), (1, 0), (1, 1)\} \\ \text{LIS} &\leftarrow \{(0, 1), (1, 0), (1, 1)\}. \end{aligned}$$

Cuando un coeficiente de LIC no es significativo, no ocurre nada en las listas, pero si lo es, se mueve desde LIC a LSC, para posteriormente ser refinado. De forma similar, si un coeficiente de LIS no es significativo (es un zerotree) no ocurre nada en las listas, pero si es significativo, debe ser particionado en subconjuntos que tengan tantas posibilidades de ser zerotrees como de no serlo.

SPIHT partitiona un $\mathcal{D}(i, j)$ en

$$\{(k, l) \in \mathcal{O}(i, j), \mathcal{L}(i, j)\},$$

donde

$$\mathcal{L}(i, j) = \mathcal{D}(i, j) - \mathcal{O}(i, j).$$

Cada $\mathcal{D}(i, j)$ se descompone en 5 partes: los 4 nodos hijo de (i, j) y el resto de descendientes.

Los $(k, l) \in \mathcal{O}(i, j)$ se insertan en LIC o en LSC dependiendo de si son significativos o no. En el caso de viajar a LIC la inserción debe realizarse al final de la lista para que los coeficientes sean evaluados en la pasada actual. Los $\mathcal{L}(i, j)$ se insertan en LIS para ser más tarde evaluados.

En LIS podemos encontrar, por tanto, 2 tipos de conjuntos: $\mathcal{D}(i, j)$ y $\mathcal{L}(i, j)$, que tienen un número diferente de elementos. SPIHT los diferencia diciendo que los $\mathcal{D}(i, j)$ son de tipo A mientras que los $\mathcal{L}(i, j)$ son de tipo B.²¹

El particionado de un $\mathcal{L}(i, j)$ es distinto, por tanto, al de un $\mathcal{D}(i, j)$. Si un $\mathcal{L}(i, j)$ es significativo entonces se partitiona en 4 conjuntos

$$\{\mathcal{D}(k, l) \in \mathcal{O}(i, j)\},$$

es decir, en los 4 árboles hijos de (i, j) que se vuelven a insertar al final de LIS. Finalmente $\mathcal{L}(i, j)$ desaparece de LIS porque este AOE ha sido partitionado en sus 4 subárboles.

²¹Recuérdese que tanto los AOEs como los coeficientes individuales se representan por un par de coordenadas.

La fase de refinamiento

Entre cada fase de ordenación se realiza otra de refinamiento. Si p es el plano de bits a transmitir, en esta fase se emite el p -ésimo bit más significativo de cada coeficiente almacenado en LSC. Cuando un coeficiente ha sido totalmente enviado (recuérdese que 3/4 de los coeficientes están ponderados por una potencia de 2 y por tanto sus bits menos significativos son cero forzosamente) se elimina de LSC.

En este punto existen dos alternativas de implementación. Si O_p representa los bits emitidos durante la fase de ordenación de la capa p y R_p a los bits de refinamiento, una posibilidad de construcción del *code-stream* es

$$O_p R_p O_{p-1} R_{p-1} O_{p-2} \dots . \quad (4.29)$$

Sin embargo, los autores del algoritmo recomiendan que los bits de ordenación del plano $p - 1$ antecedan a los bits de refinamiento del plano p , es decir

$$O_p O_{p-1} R_p O_{p-2} R_{p-1} \dots . \quad (4.30)$$

Este procedimiento tiene sentido porque de esta forma primero se envían los bits que definen nuevos coeficientes distintos de 0 y a continuación se refinan los coeficientes que ya eran significativos.

Las dos alternativas han sido evaluadas y sus rendimientos son muy similares. En la exposición del pseudo-código que expone el *codec*, por sencillez, se usará la primera opción a la que llamaremos genéricamente “refinar antes” frente a la segunda que será referenciada por “refinar después”.

La fase de cuantificación

La fase de cuantificación se usa para decrementar el umbral de significancia que en SPIHT son siempre potencias de dos. De esta forma se seleccionan los planos de bits de los coeficientes wavelet en el orden correcto.

El codificador

Se presenta sólo el algoritmo de compresor ya que el descompresor es prácticamente idéntico.

1. Fase de inicialización.

- (a) Emitir el índice del plano más significativo. Sea este valor p .
- (b) $LSC \leftarrow \emptyset$.
- (c) $LIC \leftarrow \{(0, 0), (0, 1), (1, 0), (1, 1)\}$.
- (d) $LIS \leftarrow \{(0, 1), (1, 0), (1, 1)\}$.

2. Mientras $p \geq 0$:

(a) **Fase de refinamiento.**

- i. Para cada $(i, j) \in \text{LSC}$:
 - A. Emitir el p -ésimo bit del coeficiente (i, j) .
 - B. Si (i, j) ha sido totalmente enviado borrarlo de LSC.

(b) **Fase de ordenación.**

- i. Para cada $(i, j) \in \text{LIC}$:
 - A. Emitir $S_p(i, j)$.
 - B. Si $S_p(i, j) = 1$ entonces:
 - Mover (i, j) desde LIC a LSC.
 - Emitir el signo de (i, j) .
- ii. Para cada $(i, j) \in \text{LIS}$:
 - A. Si (i, j) es de tipo A entonces:
 - Emitir $S_p(\mathcal{D}(i, j))$.
 - Si $S_p(\mathcal{D}(i, j)) = 1$ entonces:
 - Para cada $(k, l) \in \mathcal{O}(i, j)$:
 - ★ Emitir $S_p(k, l)$.
 - ★ Si $S_p(k, l) = 1$ entonces:
 - Añadir (k, l) a LSC.
 - Emitir el signo de (k, l) .
 - ★ Si no:
 - Añadir (k, l) al final de LIC.
 - Si $\mathcal{L}(i, j) \neq \emptyset$ (tiene al menos nietos) entonces:
 - ★ Mover (i, j) al final de LIS como de tipo B.
 - Si no:
 - ★ Borrar (i, j) de LIS.
 - B. Si no (es de tipo B):
 - Emitir $S_p(\mathcal{L}(i, j))$ (si alguno de los nietos de (i, j) es significativo).
 - Si $S_p(\mathcal{L}(i, j)) = 1$ entonces:
 - Añadir cada hijo $(k, l) \in \mathcal{O}(i, j)$ al final de LIS como de tipo A.
 - Borrar (i, j) de LIS.

(c) **Fase de cuantificación.**

- i. $p \leftarrow p - 1$.

El descodificador

Para encontrar el algoritmo de descodificación debe tenerse presente que todas las instrucciones de salto condicional están controladas por el valor devuelto por $S_p(\cdot)$ (ver Ecuación (4.28)), los cuales forman el *code-stream* generado durante la fase de ordenación y

que por lo tanto son conocidos por el descodificador. Éste puede realizar exactamente la misma traza de instrucciones si el algoritmo que ejecuta es idéntico al del codificador excepto porque donde aparece “emitir” ahora debe poner “recibir”. Ya que se trata del mismo algoritmo, las tres listas LSC, LIC y LIS van a generarse exactamente de la misma forma a como se generaron en el codificador, gracias a lo cual, la reconstrucción del plano es trivial y además, las complejidades del codificador y del descodificador son idénticas.

Sin embargo, si SPIHT va a ser usado como transmisor progresivo,²² el descodificador debe realizar una tarea extra para ajustar los coeficientes重建idos a partir del intervalo de incertidumbre durante la fase de ordenación.

Cuando una coordenada se mueve a LSC, se sabe que el valor del coeficiente c cumple que

$$2^p \leq |c| < 2^{p+1}, \quad (4.31)$$

donde p es el plano de bit transmitido. El descodificador utiliza esta información (más el bit de signo que llega a continuación), para ajustar el valor reconstruido a la mitad del intervalo $[2^p, 2^{p+1}-1]$ ya que de esta forma el error con respecto al valor real del coeficiente será la mitad en promedio. Por ejemplo, si $p = 15$, se sabe que el coeficiente es mayor que 32768 y menor que 65535. Si sólo hacemos que el bit 15 sea 1 estaremos reconstruyendo el coeficiente con el valor mínimo que puede realmente tener. La solución es hacer

$$\hat{c} = \pm 1.5 \times 2^p = \pm 3 \times 2^{p-1}, \quad (4.32)$$

que es el valor que divide al intervalo de incertidumbre en dos mitades iguales. En nuestro ejemplo, el valor de reconstrucción sería $3 \times 2^{15-1} = 49152$.

De forma similar, durante la fase de refinamiento, cuando el descodificador conoce el valor real del bit $(p-1)$ -ésimo, resulta otro intervalo de incertidumbre. Si el bit recibido es un 1, entonces el bit $(p-1)$ -ésimo ya es correcto, pero debemos hacer el bit $(p-2)$ -ésimo igual a 1 para ajustar a la mitad del nuevo intervalo de incertidumbre. Si el bit recibido es un 0, entonces el bit $(p-1)$ -ésimo está equivocado y debe ser puesto a 0. El bit $(p-2)$ -ésimo debe hacerse 1 para ajustar el valor reconstruido a la mitad del intervalo de incertidumbre. Por lo tanto, en cualquier caso, cuando estamos refinando, colocamos el bit recibido a su valor adecuado y el siguiente bit menos significativo se hace 1.

4.15.4 SPIHT + codificación aritmética

A pesar de que SPIHT es muy eficiente, los autores se dieron cuenta que existe cierta dependencia entre $S_p(i, j)$ y $S_p(\mathcal{D}(i, j))$, es decir entre un coeficiente y sus descendientes, y además entre la significancia de coeficientes adyacentes. La implementación proporcionada por los autores y que es usada para comprobar el rendimiento de nuestra implementación explota también estas fuentes de redundancia usando un codificador aritmético

²²Cosa que es lo más normal, aunque podría no ser así ya que también es un compresor de imágenes sin pérdidas muy eficiente.

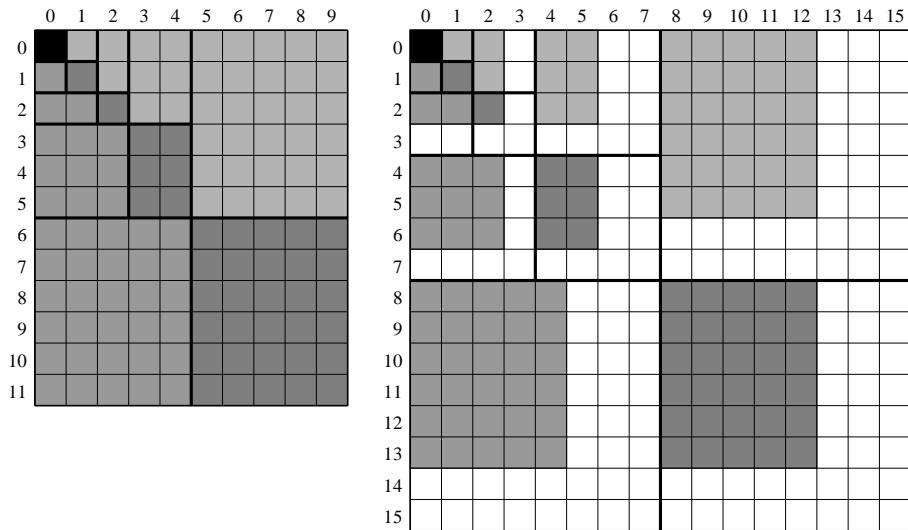


Figura 4.37: Ejemplo de descomposición wavelet de una imagen de dimensiones 12×10 (izq). Mapeo de la transformación sobre una matriz de 16×16 coeficientes (der).

multisímbolo. Sin embargo, también aportan el dato de que las ganancias esperables son inferiores a 1 dB-PSNR para la mayoría de las imágenes.

Por otro lado, el uso de un código aritmético complica la secciónabilidad del *code-stream*, uno de los requisitos expuestos en la Sección 4.1. Es decir, existe una cierta pérdida de capacidad para seccionar el código por cualquier bit. Este inconveniente junto con las mínimas mejoras en las tasas de compresión que son esperables de esta codificación aritmética han provocado que nuestra implementación no la use.

Para distinguir ambas implementaciones, llamaremos SPIHT a la propuesta que nosotros hacemos y SPIHT+arith a la original.

4.15.5 SPIHT sobre imágenes con cualquier tamaño

Tal y como ha sido definido, SPIHT no puede ser usado sobre imágenes que no sean cuadradas o con lado distinto de una potencia de dos. El problema es que los AOE_s no pueden crearse, al menos, tal y como han sido definidos en la Expresión (4.26).

Supongamos por ejemplo una imagen de 12×10 puntos. Nuestra implementación de la transformada S+P genera una descomposición como la mostrada en la Figura 4.37, lado izquierdo. Cada cuadrado simboliza un coeficiente wavelet. A la derecha se ha dibujado el mapeo que habría que realizar para que los AOE_s puedan ser usados sobre esta descomposición. Los que están sombreados contienen información (son presumiblemente diferentes de cero) mientras que los que están sin rellenar son forzosamente 0.

Una forma muy aproximada de crear una estructura de datos como ésta consiste en usar una imagen cuadrada nula con lado igual a una potencia de dos en la que se ha

insertado la imagen a comprimir, ajustándose a la esquina superior izquierda. Tras aplicar la transformada wavelet se obtiene dicha descomposición en la que podemos encontrar, en varios niveles, coeficientes que son siempre cero.

SPIHT puede ser usado directamente sobre dicha descomposición porque los coeficientes que son cero debido a que la imagen no es de lado 2^l (donde l es el número de niveles) forman zerotrees y son identificados como tales en niveles superiores de la descomposición. Sin embargo, SPIHT emplea un bit de código para indicar al descodificador que en esa región existe un zerotree que en realidad es perfectamente predecible porque se conocen las dimensiones reales de la imagen. Por lo tanto, en una implementación más adecuada dicho bit podría dejar de transmitirse.

A pesar de que este problema se repite en cada plano de bit, la cantidad de código redundante introducida no es alta y por eso se ha decidido usar SPIHT sin modificación en nuestras evaluaciones, con el objetivo de no alargar los tiempos de codificación y decodificación de imágenes cuadradas y de lados igual a una potencia de dos (que son sin duda un caso muy frecuente).

4.16 Evaluación

A continuación procedemos a la evaluación del algoritmo SPIHT usado como compresor de imágenes y como transmisor progresivo. En todos los casos el espectro wavelet codificado ha sido el de la transformada S+P unitaria (U(S+P)T).

4.16.1 Evaluación como compresor

Se ha procedido a comprimir las imágenes de prueba del Apéndice A usando dos implementaciones del algoritmo SPIHT: SPIHT (a secas) que responde a nuestra implementación y SPIHT+arith que es la implementación que los autores donaron al *public domain*. Existen dos diferencias fundamentales entre ambas implementaciones, de cara a ser evaluadas sólo como compresores reversibles de imágenes:

- SPIHT no usa codificación aritmética para comprimir los bits de ordenación y refinamiento, mientras que SPIHT+arith sí lo hace.
- SPIHT no codifica (envía como están) los planos de bits menos significativos, mientras que SPIHT+arith usa un compresor aritmético basado en el contexto que se aplica a cada plano de bits.

Las Tablas 4.5, 4.6, 4.7, 4.8 muestran las tasas de compresión y transferencia alcanzadas para las imágenes de *test*.

Los resultados confirman que el uso de un codificador aritmético puede ser evitado, especialmente en el caso de las imágenes de 16 bpp donde ambos compresores alcanzan la misma tasa de compresión promedio mientras SPIHT es aproximadamente un 25% más rápido que SPIHT+arith.

Tabla 4.5: Tasas de compresión (multiplicadas por 1000) de los algoritmos de compresión progresiva de imágenes probados para las imágenes de 8 bpp. En SPIHT, los 3 planos de bits menos significativos no fueron codificados. SPIHT+arith codificó aritméticamente los últimos planos de bits.

método	imágenes				media
	lena	bárbara	boats	zelda	
SPIHT	445	401	428	476	438
SPIHT+arith	476	428	458	509	468

Tabla 4.6: Tasas de transferencia (multiplicadas por 1000) de los algoritmos de compresión progresiva de imágenes probados para las imágenes de 8 bpp. Con SPIHT los 3 últimos planos de bits no fueron codificados y no se realiza corrección del intervalo de incertidumbre (puesto que no es necesario). Con SPIHT+arith los últimos planos de bits fueron comprimidos aritméticamente.

método	imágenes				media
	lena	bárbara	boats	zelda	
SPIHT	81/77	72/70	75/76	81/81	77/76
SPIHT+arith	52/52	50/51	50/49	54/51	52/51

Tabla 4.7: Tasas de compresión (multiplicadas por 1000) de los algoritmos de compresión progresiva de imágenes probados para las imágenes de 16 bpp. Con SPIHT los 3 últimos planos de bits no fueron codificados mientras que con SPIHT+arith los últimos planos de bits fueron comprimidos aritméticamente.

método	imágenes				media
	B0010	29jul24	29jul25	29jul26	
SPIHT	717	585	493	514	577
SPIHT+arith	715	581	501	509	577

Tabla 4.8: Tasas de transferencia (multiplicadas por 1000) de los algoritmos de compresión progresiva de imágenes secuenciales probados para las imágenes de 16 bpp. Con SPIHT los 3 últimos planos de bits no fueron codificados y no se realiza corrección del intervalo de incertidumbre (puesto que no es necesario). Con SPIHT+arith los últimos planos de bits fueron comprimidos aritméticamente.

método	imágenes				media
	B0010	29jul24	29jul25	29jul26	
SPIHT	85/85	80/81	64/68	71/72	75/77
SPIHT+arith	52/51	58/55	50/47	50/48	53/50

Las tasas de compresión son muy buenas. Para las imágenes de 8 bpp, los resultados son muy semejantes a los de CALIC (ver Tablas 3.5 y 3.6). En el caso de las imágenes de 16 bpp, las tasas son las mejores conocidas hasta el momento.

SPIHT sólo plantea un inconveniente. Cuando los planos de bits menos significativos son codificados, la ocurrencia de zerotrees es muy poco probable y SPIHT es más redundante que enviar los planos sin codificar. Por este motivo, en la mayoría de las imágenes, SPIHT se usa sólo hasta el segundo o tercer plano de bits. De esta forma se acelera la compresión y se reduce el tamaño del fichero comprimido. Además, la relación señal/ruido en estos planos de bits es muy baja y su transmisión progresiva no mejora la calidad visual de la reconstrucción.

Las Figuras 4.38 y 4.39 sitúan a SPIHT y SPIHT+arith en el espacio del rendimiento junto con todos los demás compresores evaluados en capítulos anteriores. Por el lugar que ocupan, podemos afirmar que la compresión progresiva es tan eficiente en nivel de compresión como la compresión “secuencial”, tanto en tasas de compresión como en tasas de transferencia. Si tenemos en cuenta que la posibilidad de una visualización progresiva es además accesible, los compresores progresivos de imágenes son la opción a elegir en la mayoría de las situaciones.

Resaltar además la alta simetría del *codec* ya que el compresor y el descompresor ejecutan prácticamente el mismo algoritmo. Esto es fundamental cuando la imagen no está previamente comprimida usando SPIHT en el emisor (ver Sección 4.1).

4.16.2 Evaluación como transmisor

La característica más interesante de un codificador progresivo es evidentemente su uso como transmisor. Se ha dividido la evaluación en dos apartados, uno subjetivo y otro objetivo.

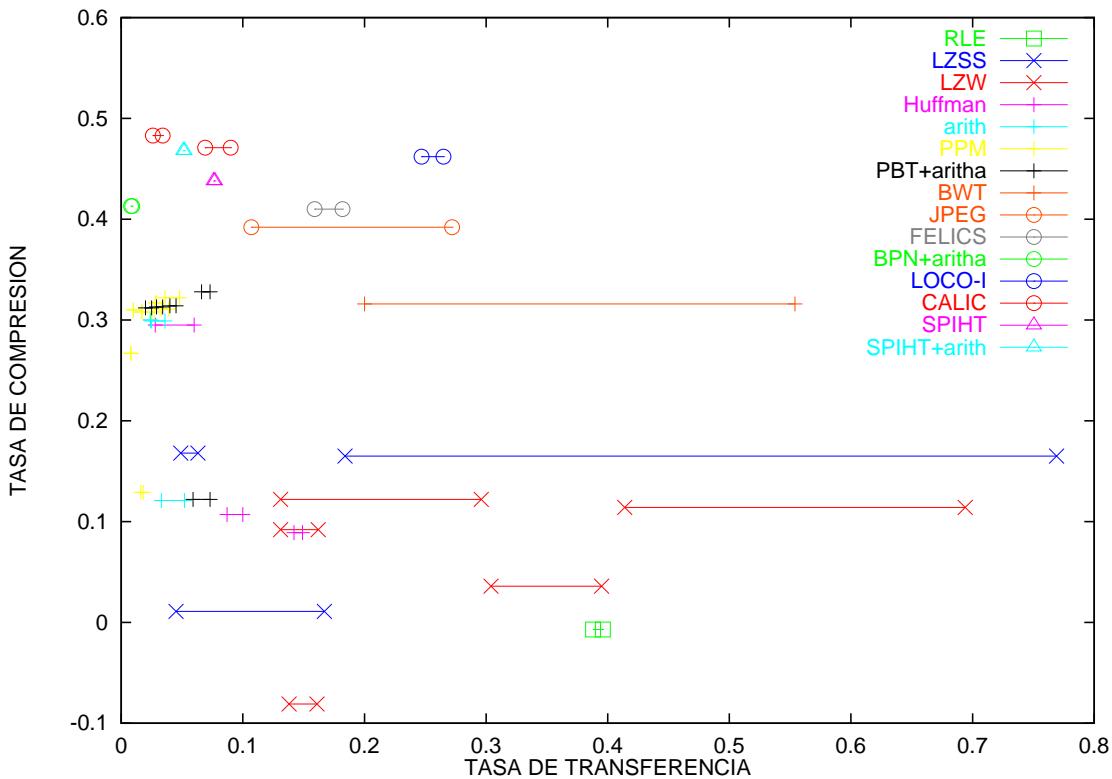


Figura 4.38: Rendimiento de los algoritmos de compresión de imágenes probados para las imágenes de 8 bpp. Los transmisores progresivos tienen asignado un triángulo como símbolo.

Objetivamente

En primer lugar se ha procedido a medir la relación entre la compresión y la relación señal/ruido. Las curvas generadas indican la calidad de la reconstrucción en el receptor en función del número de bits recibidos.

En el caso de las imágenes de 8 bpp (ver Figuras 4.40, 4.41, 4.42 y 4.43) se ha presentado el rango que abarca hasta 1 bit/punto porque en general a este nivel de compresión son visualmente indistinguibles de la original. En el caso de las imágenes de 16 bpp (ver Figuras 4.44, 4.45, 4.46 y 4.47) paramos en 0.1 bits/punto por la misma razón.

En cada gráfica aparecen 5 curvas. La de color verde se corresponde con SPIHT+arith y marca (teóricamente) el límite máximo de relación señal/ruido. La de color azul es para SPIHT, refinando después y realizando la corrección del intervalo de incertidumbre. La de color rojo es para SPIHT, refinando antes y realizando la corrección del intervalo de incertidumbre. La de color rosa es para SPIHT, refinando después pero sin realizar la corrección del intervalo. La de color celeste es para SPIHT, refinando antes y sin corregir.

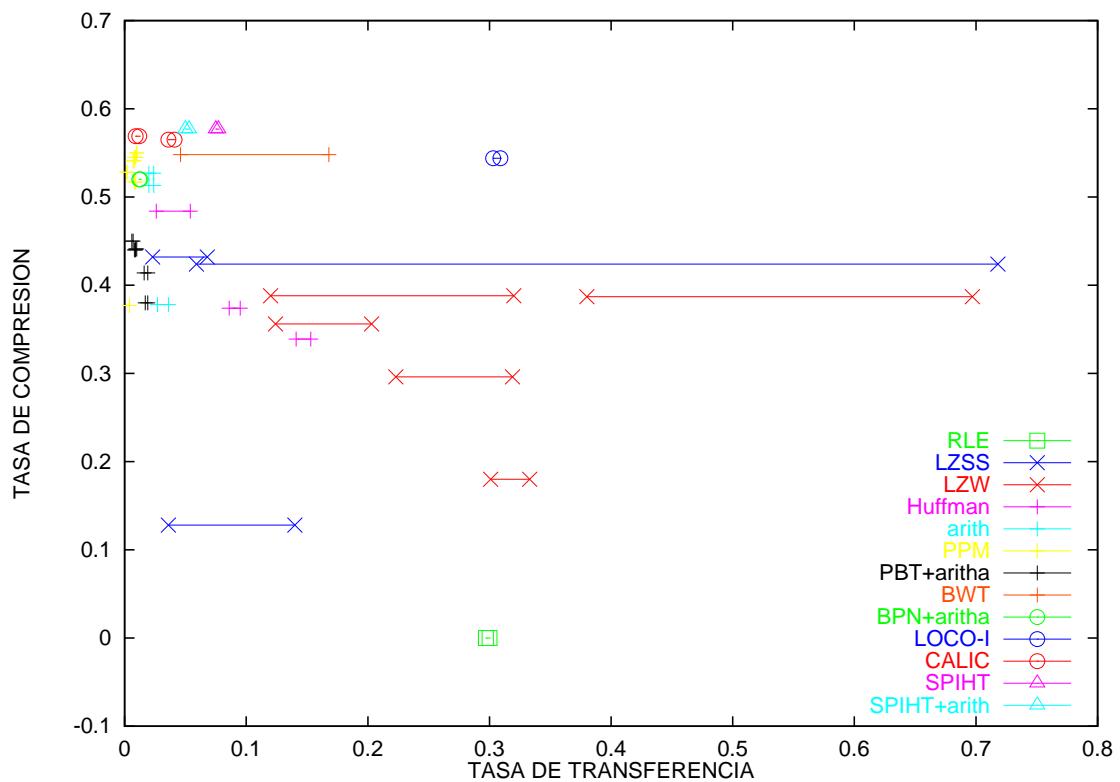


Figura 4.39: Rendimiento de los algoritmos de compresión de imágenes probados para las imágenes de 16 bpp. Los transmisores progresivos tienen asignado un triángulo como símbolo.

Un análisis de las gráficas conlleva los siguientes comentarios:

- El rendimiento de todas las implementaciones de SPIHT es semejante.
- El uso de un codec aritmético (como hace SPIHT+arith) para explotar la correlación entre bits de significancia (los que indican al codificador que un zerotree ha dejado de serlo) aumenta las relaciones señal/ruido, aunque de forma modesta especialmente en el caso de las imágenes de 8 bpp. Realmente las diferencias se hacen insignificantes cuando el nivel de compresión supera 1 bpp, para las imágenes de 8 bpp, y 0.1 bpp para las imágenes de 16 bpp.
- Como a continuación comprobaremos, al principio de la transmisión, la relación señal/ruido no se correlaciona muy bien con la calidad visual de la imagen reconstruida. De hecho, para un observador humano, el refinamiento después es más adecuado mientras las curvas indican lo contrario, si bien es cierto que la diferencia es suave.

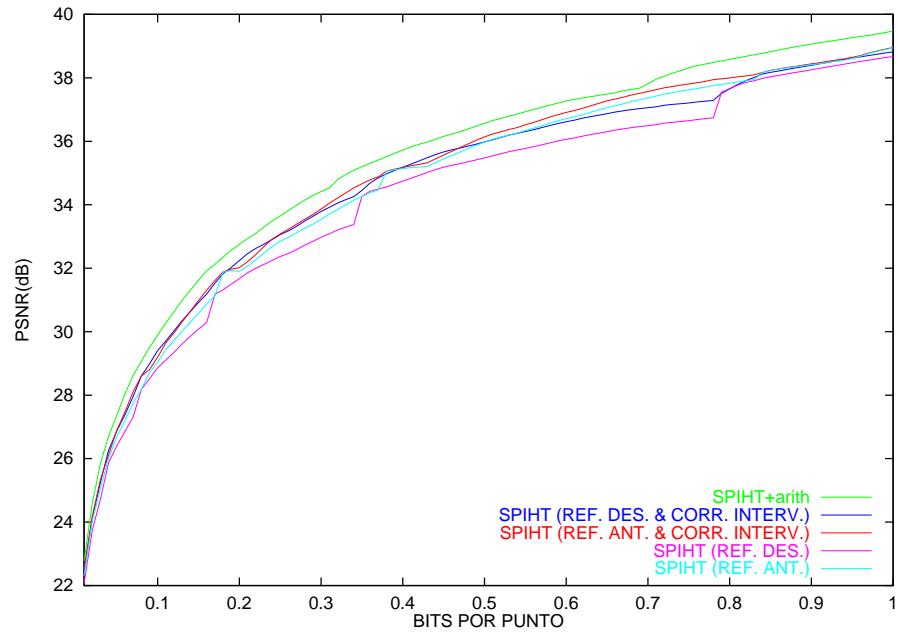


Figura 4.40: Calidad de las reconstrucciones (en PSNR(dB)) de la imagen “lena” en función de las diferentes alternativas de implementación usando SPIHT.

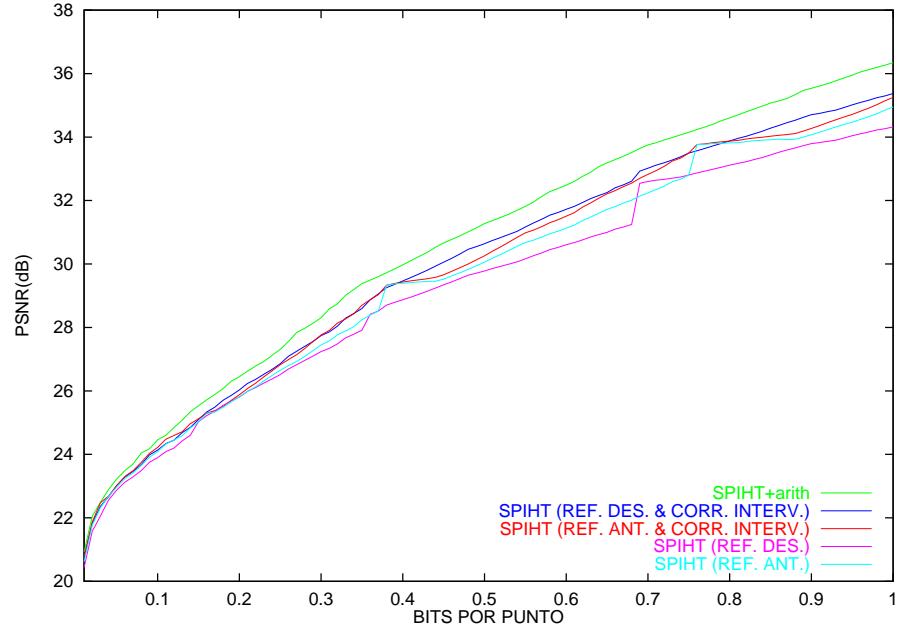


Figura 4.41: Calidad de las reconstrucciones (en PSNR(dB)) de la imagen “bárbara” en función de las diferentes alternativas de implementación usando SPIHT.

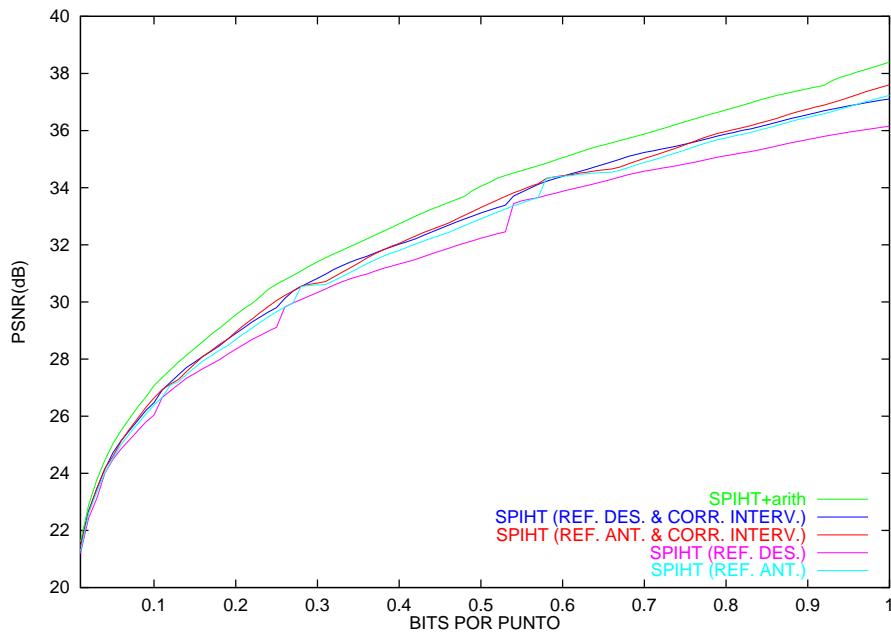


Figura 4.42: Calidad de las reconstrucciones (en PSNR(dB)) de la imagen “boats” en función de las diferentes alternativas de implementación usando SPIHT.

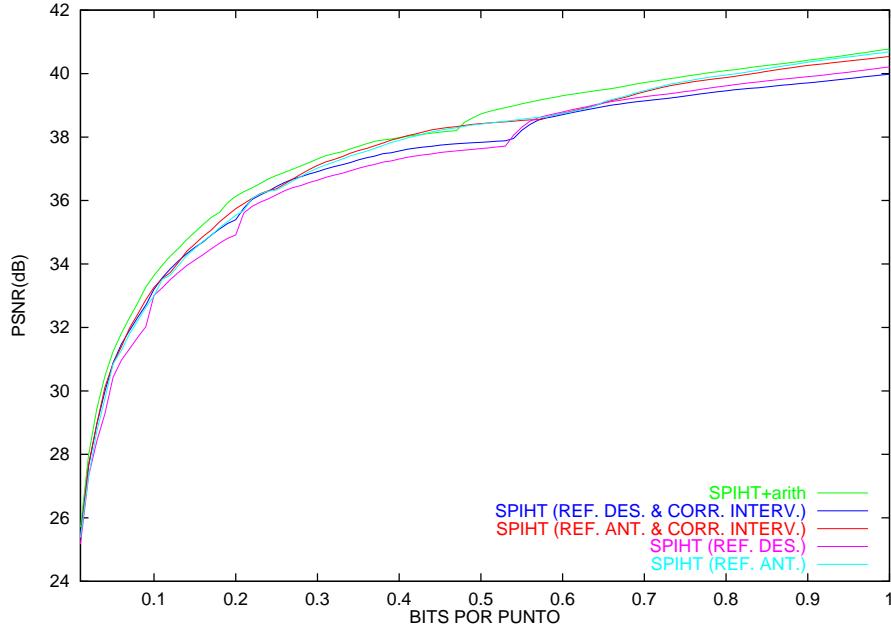


Figura 4.43: Calidad de las reconstrucciones (en PSNR(dB)) de la imagen “zelda” en función de las diferentes alternativas de implementación usando SPIHT.

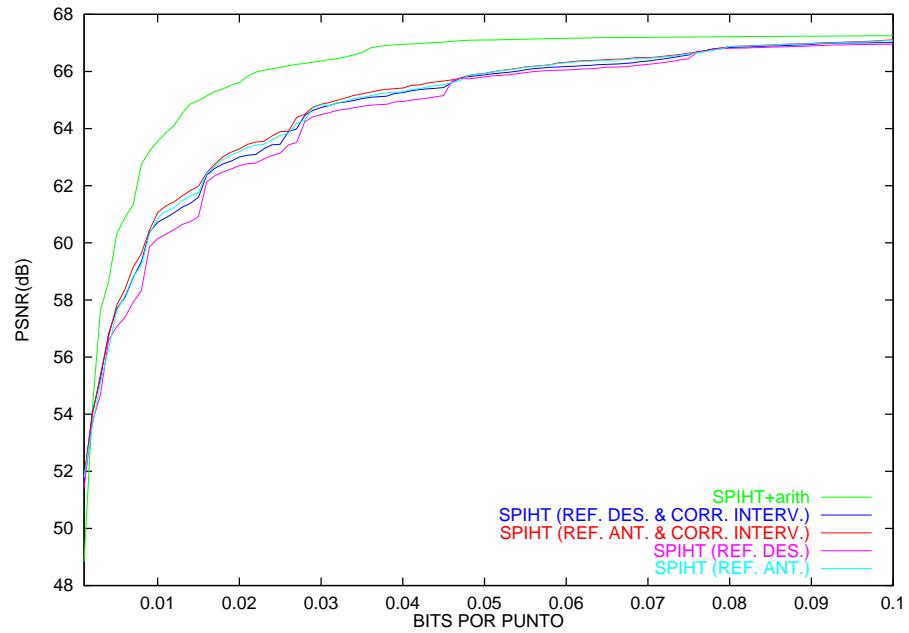


Figura 4.44: Calidad de las reconstrucciones (en PSNR(dB)) de la imagen “B0100” en función de las diferentes alternativas de implementación usando SPIHT.

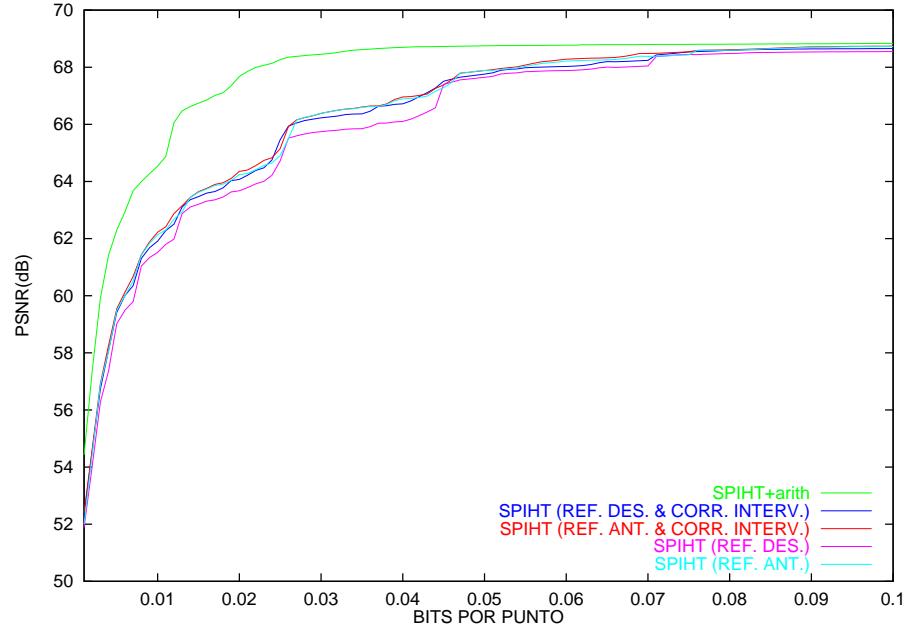


Figura 4.45: Calidad de las reconstrucciones (en PSNR(dB)) de la imagen “29jul24” en función de las diferentes alternativas de implementación usando SPIHT.

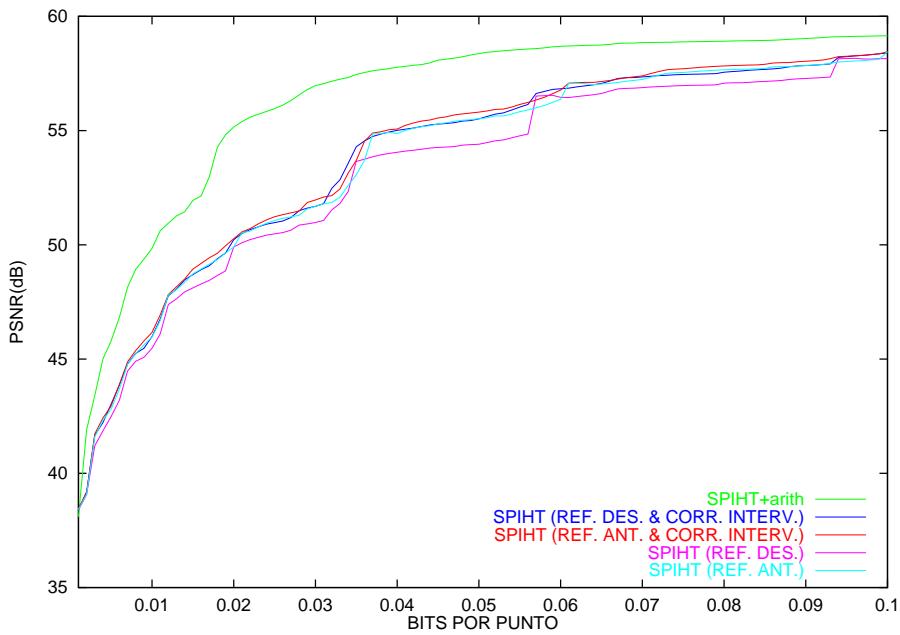


Figura 4.46: Calidad de las reconstrucciones (en PSNR(dB)) de la imagen “29jul25” en función de las diferentes alternativas de implementación usando SPIHT.

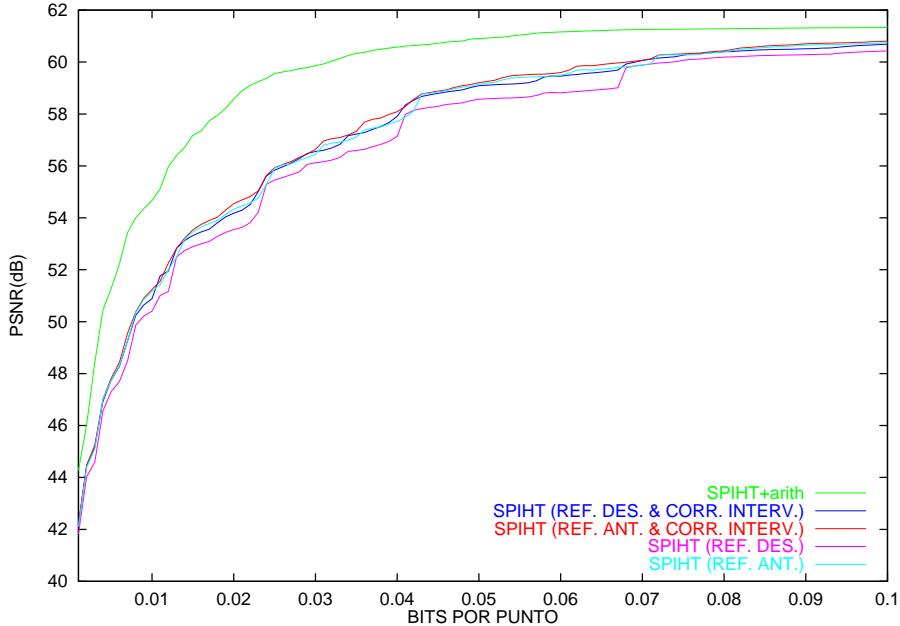


Figura 4.47: Calidad de las reconstrucciones (en PSNR(dB)) de la imagen “29jul26” en función de las diferentes alternativas de implementación usando SPIHT.

Subjetivamente

Primero realizaremos una serie de evaluaciones para determinar si es más conveniente refiniar antes o después. El criterio de decisión será estrictamente visual. En las Figuras 4.48 y 4.49 se muestra la transmisión progresiva de la imagen “lena” y “B0100” muy al comienzo de la transmisión donde existen muchos coeficientes wavelet diferentes de cero que transmitir (bits de significancia) frente a los bits de refinamiento. Se puede apreciar (especialmente en el caso de “lena”) que refinar después es más conveniente porque ayuda a definir dichos coeficientes en menos tiempo. En el caso de “B0100” este efecto es menos aparente porque existen muy pocos objetos dentro de la escena.

A continuación se presenta la transmisión progresiva de las imágenes de prueba usando SPIHT+U(S+P)T. Todas ellas se reconstruyeron corrigiendo el intervalo de incertidumbre y refinando después. Ésta es quizás la evaluación más interesante de un codificador progresivo porque muestra realmente la calidad de las imágenes reconstruidas en el receptor que en ocasiones no se correlaciona adecuadamente con las medidas estándares de error o discrepancia entre imágenes. Junto a las figuras se dan datos acerca de los niveles de compresión, número de bytes transmitidos y relaciones señal/ruido.

Todas las imágenes de 8 bits se han reconstruido progresivamente para 0.01, 0.02, 0.03, 0.04, 0.05 y 0.06 bits/punto. La Figura 4.50 muestra “lena”, la Figura 4.51 a “bárbara”, la Figura 4.52 a “boats” y la Figura 4.53 a “zelda”.

Por el contrario, las imágenes astronómicas se han presentado progresivamente para los niveles de compresión 0.0003, 0.0004, 0.0005, 0.0006, 0.0007, 0.0008, 0.0009, 0.001, 0.002, 0.003, 0.004 y 0.005 bits/punto, pero en dos versiones diferentes. Las Figuras 4.54 y 4.55 muestran sólo una sección de 256×256 puntos con el objetivo de ver qué ocurre localmente con algunos cuerpos celestes. Sin embargo, en las Figuras 4.56, ..., 4.61 y 4.62, ..., 4.67, (que exponen la transmisión de “29jul0025” y “29jul0026” respectivamente), toda la imagen es presentada con el fin de ver qué está ocurriendo globalmente durante la transmisión.

Los resultados obtenidos indican que el algoritmo de transmisión progresivo es muy eficiente. Las imágenes de tono continuo pueden ser reconocibles a 0.01 bits/punto lo que significa que puede ser mostrada por el receptor en un tiempo 800 veces inferior al tiempo que se emplearía si la imagen no estuviera comprimida.

Sin embargo, es en la transmisión de las imágenes astronómicas donde realmente se puede apreciar la utilidad de un compresor progresivo. Las imágenes pueden ser útiles para un observador en un tiempo 50000 veces inferior.²³

²³Factor que además aumenta con el tamaño de la imagen.



Figura 4.48: Comparación entre refinado antes (izquierda) o después (derecha) para “lena”. De arriba a abajo, 0.01 bpp, 0.02 bpp y 0.03 bpp. Se ha usado SPIHT+U(S+P)T.

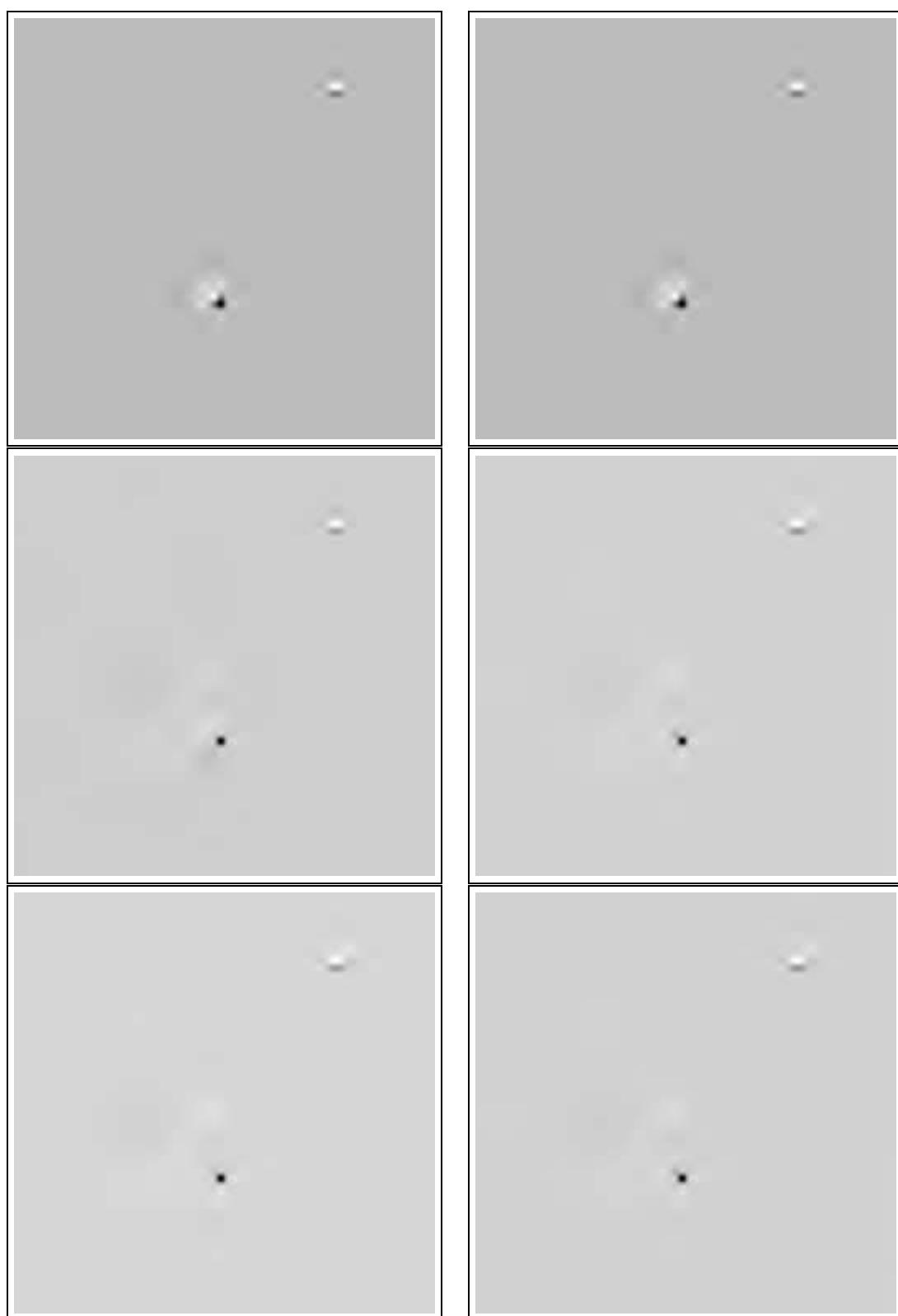


Figura 4.49: Comparación entre refinar antes (izquierda) o después (derecha) para “B0100” (detalle de 256×256 puntos). De arriba a abajo, 0.0003 bpp, 0.0004 bpp y 0.0005 bpp. Se ha usado SPIHT+U(S+P)T.

0.01 bpp (800:1), 328 bytes, 22.48 dB



0.02 bpp (400:1), 656 bytes, 24.14 dB



0.03 bpp (267:1), 984 bytes, 25.28 dB



0.04 bpp (200:1), 1311 bytes, 26.29 dB



0.05 bpp (160:1), 1639 bytes, 26.90 dB



0.06 bpp (133:1), 1967 bytes, 27.42 dB



Figura 4.50: Transmisión progresiva de “lena” usando SPIHT+U(S+P)T. El refinamiento se hace después.



Figura 4.51: Transmisión progresiva de “bárbara” usando SPIHT+U(S+P)T.

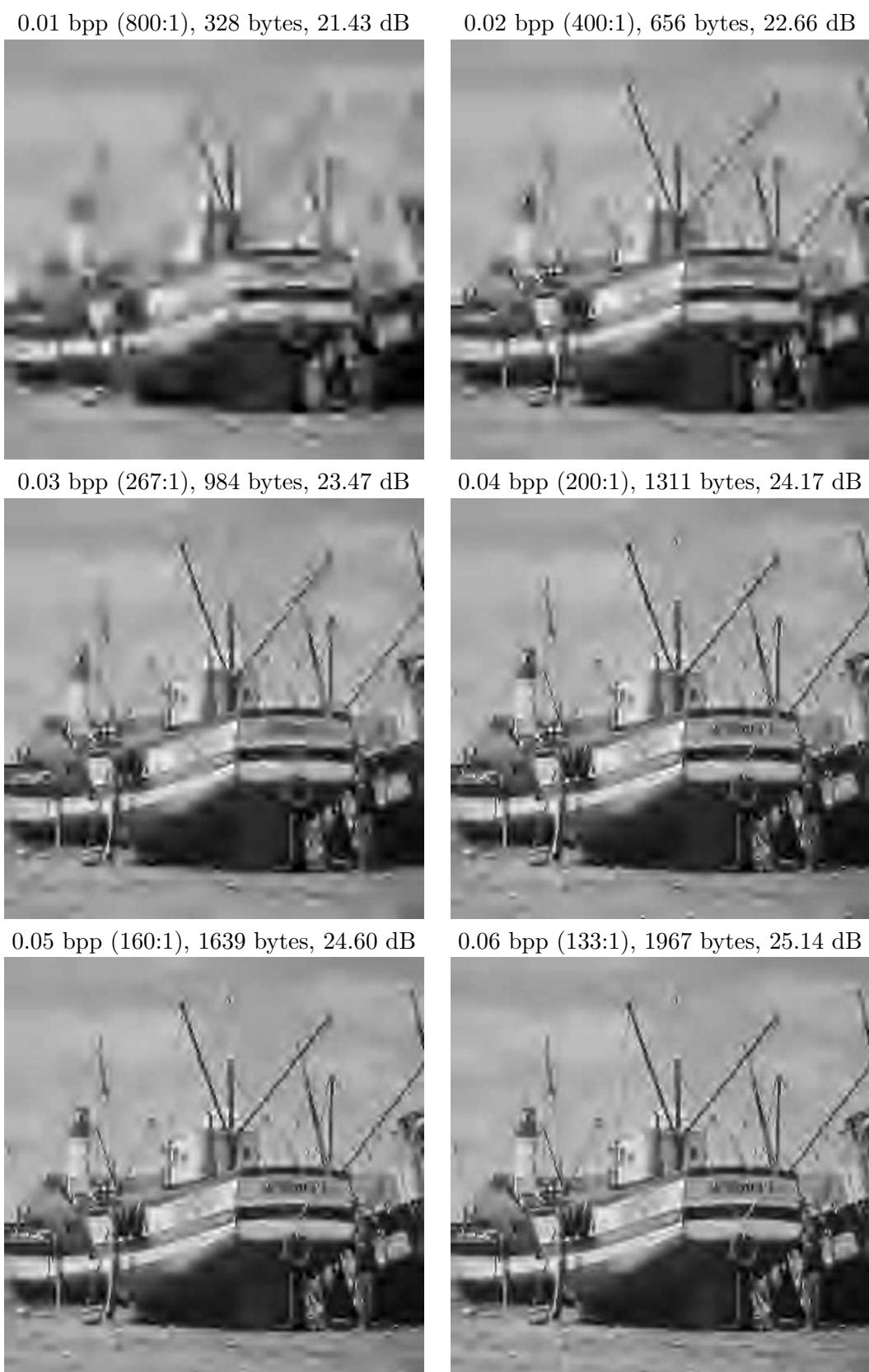


Figura 4.52: Transmisión progresiva de “boats” usando SPIHT+U(S+P)T.



Figura 4.53: Transmisión progresiva de “zelda” usando SPIHT+U(S+P)T.

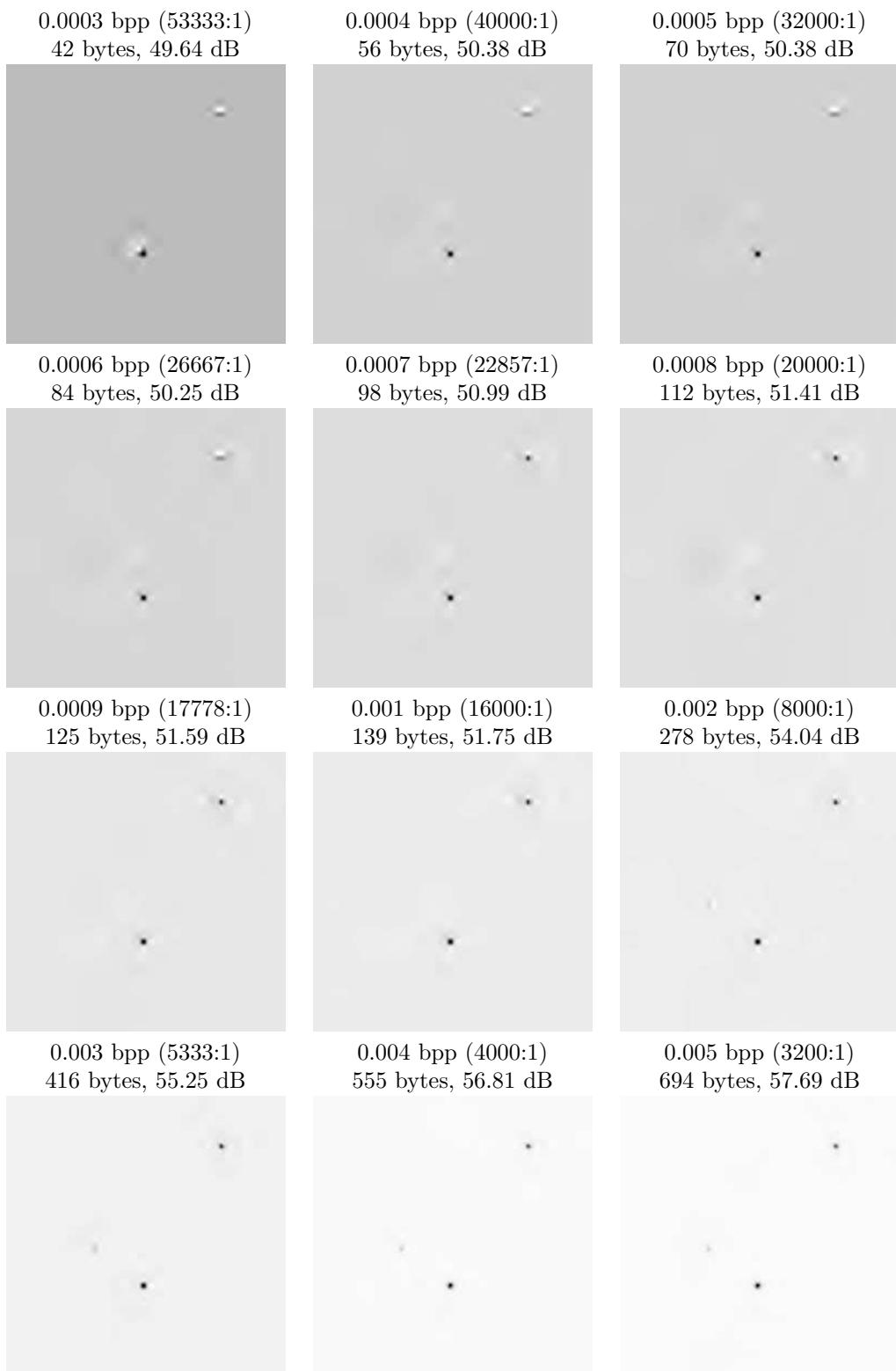


Figura 4.54: Transmisión progresiva de la imagen “B0100” usando SPIHT+U(S+P)T. Aunque toda la imagen ha sido transmitida sólo se muestra el detalle de la sección de 256×256 puntos de coordenadas (512,512,768,768). El valor pico usado en la medición del PSNR es 12460 (máximo-mínimo).

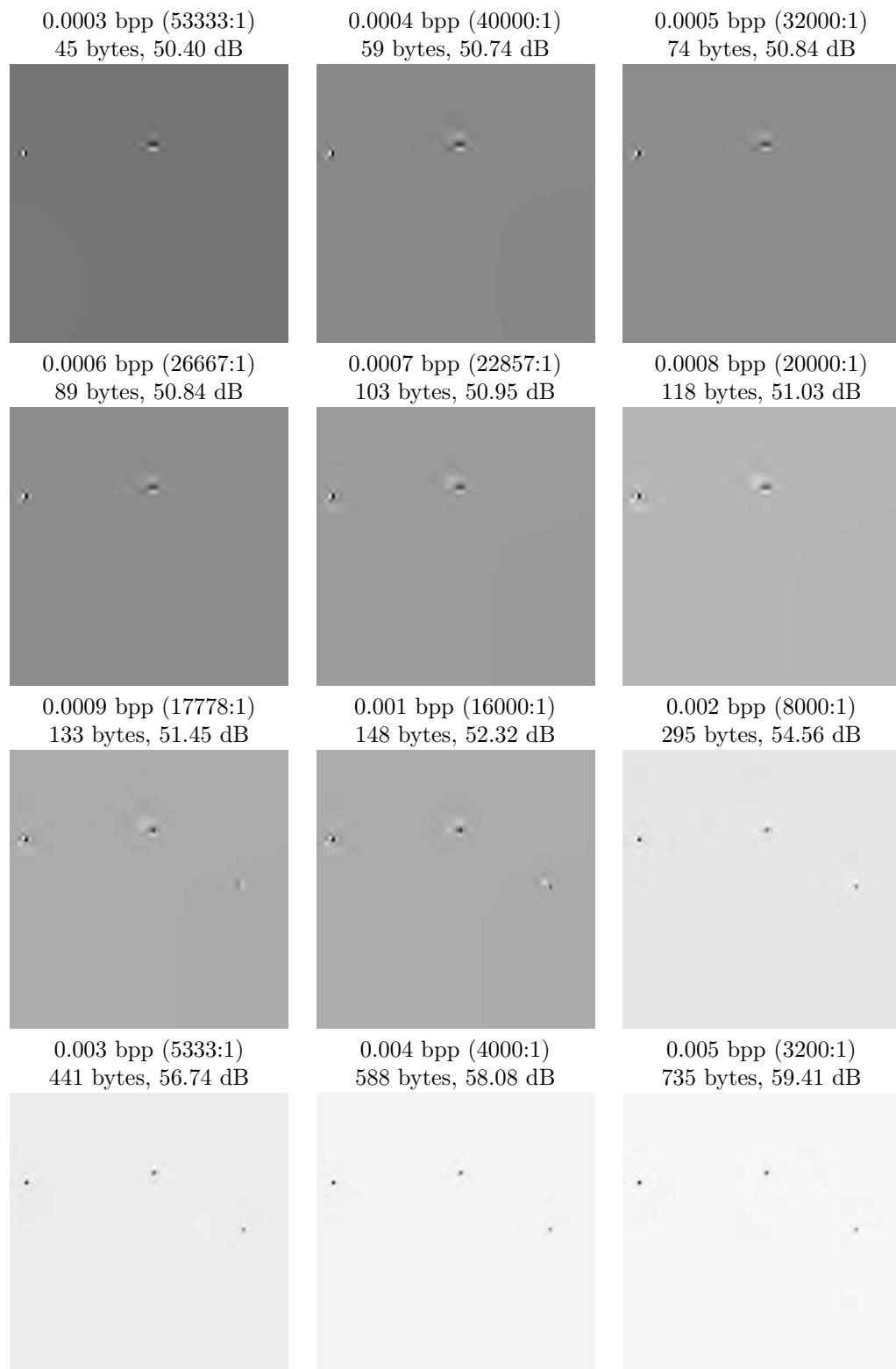


Figura 4.55: Transmisión progresiva de la imagen “29jul24” usando SPIHT+U(S+P)T. Aunque toda la imagen ha sido transmitida sólo se muestra el detalle de la sección de 256×256 puntos de coordenadas (600,400,856,656). El valor pico usado en la medición del PSNR es 62194 (máximo-mínimo).

0.0003 bpp (53333:1), 45 bytes, 36.96 dB

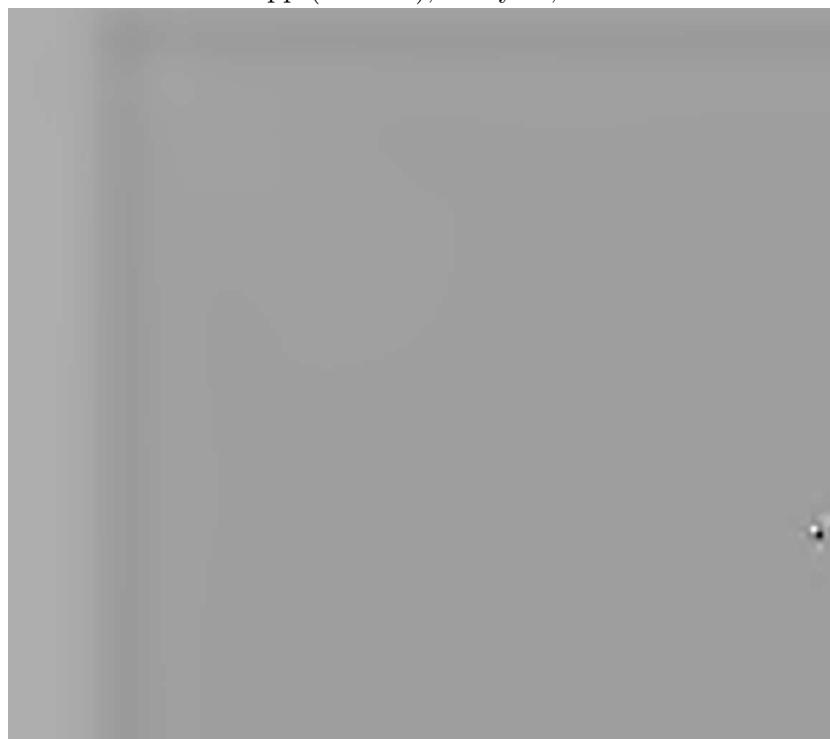


0.0004 bpp (40000:1), 59 bytes, 37.31 dB



Figura 4.56: Transmisión progresiva de la imagen “29jul25” usando SPIHT+U(S+P)T. Toda la imagen es presentada. El valor pico usado en la medición del PSNR es 62231 (máximo-mínimo).

0.0005 bpp (32000:1), 74 bytes, 37.36 dB



0.0006 bpp (26667:1), 89 bytes, 37.42 dB

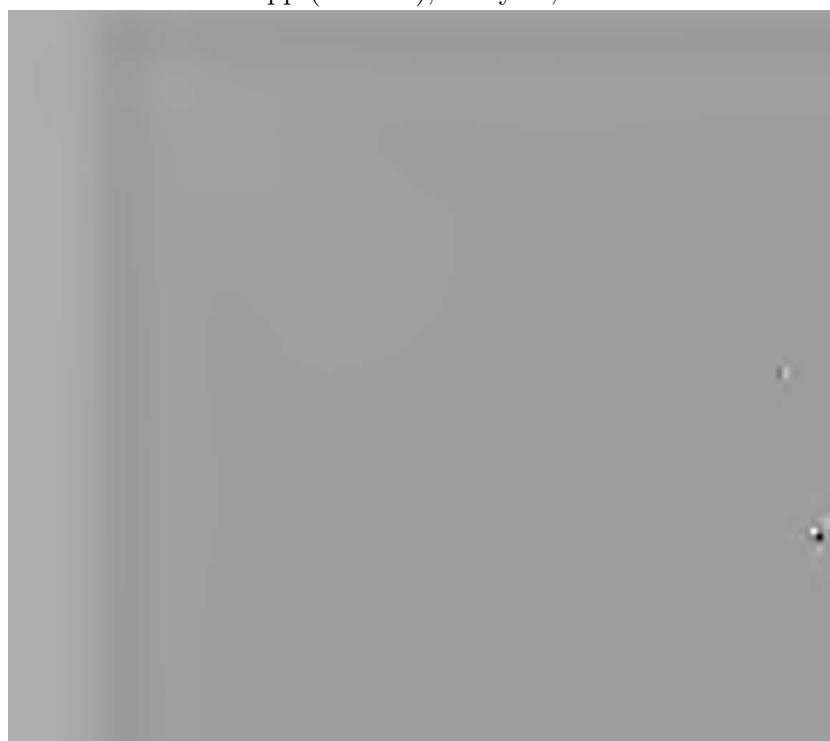


Figura 4.57: Transmisión progresiva de la imagen “29jul25” usando SPIHT+U(S+P)T (cont.). Toda la imagen es presentada.

0.0007 bpp (22857:1), 103 bytes, 38.18 dB



0.0008 bpp (20000:1), 118 bytes, 38.31 dB



Figura 4.58: Transmisión progresiva de la imagen “29jul25” usando SPIHT+U(S+P)T (cont.). Toda la imagen es presentada.

0.0009 bpp (17778:1), 133 bytes, 38.39 dB



0.001 bpp (16000:1), 148 bytes, 38.41 dB



Figura 4.59: Transmisión progresiva de la imagen “29jul25” usando SPIHT+U(S+P)T (cont.). Toda la imagen es presentada.

0.002 bpp (8000:1), 295 bytes, 39.19 dB

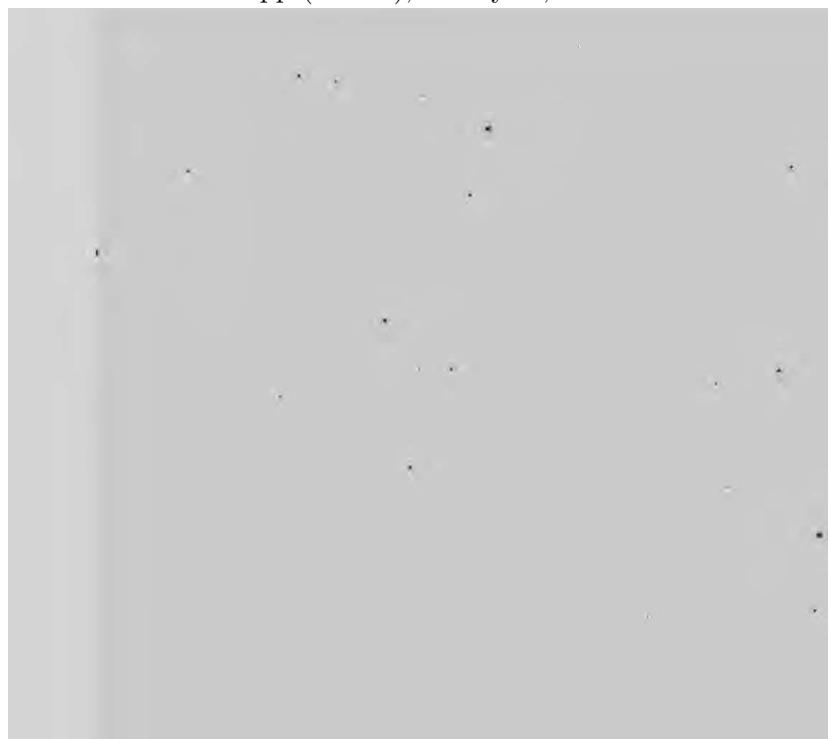


0.003 bpp (5333:1), 441 bytes, 41.67 dB



Figura 4.60: Transmisión progresiva de la imagen “29jul25” usando SPIHT+U(S+P)T (cont.). Toda la imagen es presentada.

0.004 bpp (4000:1), 588 bytes, 42.22 dB



0.005 bpp (3200:1), 735 bytes, 43.03 dB

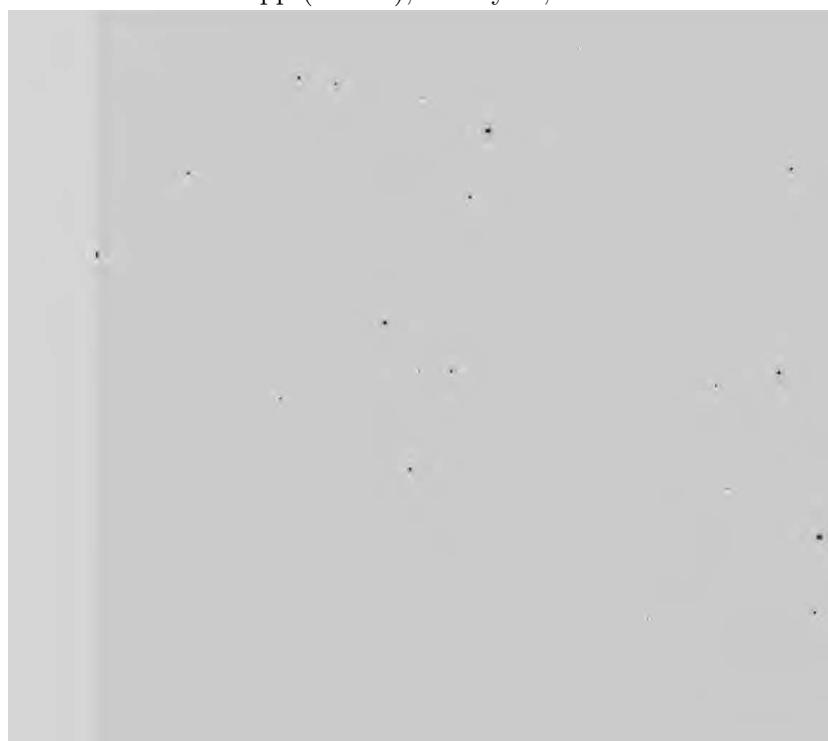


Figura 4.61: Transmisión progresiva de la imagen “29jul25” usando SPIHT+U(S+P)T (cont.). Toda la imagen es presentada.

0.0003 bpp (53333:1), 45 bytes, 39.64 dB



0.0004 bpp (40000:1), 59 bytes, 39.71 dB



Figura 4.62: Transmisión progresiva de la imagen “29jul26” usando SPIHT+U(S+P)T. Toda la imagen es presentada. El valor pico usado en la medición del PSNR es 62210 (máximo-mínimo).

0.0005 bpp (32000:1), 74 bytes, 41.54 dB



0.0006 bpp (26667:1), 89 bytes, 41.52 dB



Figura 4.63: Transmisión progresiva de la imagen “29jul26” usando SPIHT+U(S+P)T (cont.). Toda la imagen es presentada.

0.0007 bpp (22857:1), 103 bytes, 41.75 dB



0.0008 bpp (20000:1), 118 bytes, 41.84 dB



Figura 4.64: Transmisión progresiva de la imagen “29jul26” usando SPIHT+U(S+P)T (cont.). Toda la imagen es presentada.

0.0009 bpp (17778:1), 133 bytes, 42.10 dB



0.001 bpp (16000:1), 148 bytes, 42.18 dB



Figura 4.65: Transmisión progresiva de la imagen “29jul26” usando SPIHT+U(S+P)T (cont.). Toda la imagen es presentada.

0.002 bpp (8000:1), 295 bytes, 44.45 dB



0.003 bpp (5333:1), 441 bytes, 45.16 dB



Figura 4.66: Transmisión progresiva de la imagen “29jul26” usando SPIHT+U(S+P)T (cont.). Toda la imagen es presentada.

0.004 bpp (4000:1), 588 bytes, 46.93 dB



0.005 bpp (3200:1), 735 bytes, 47.74 dB

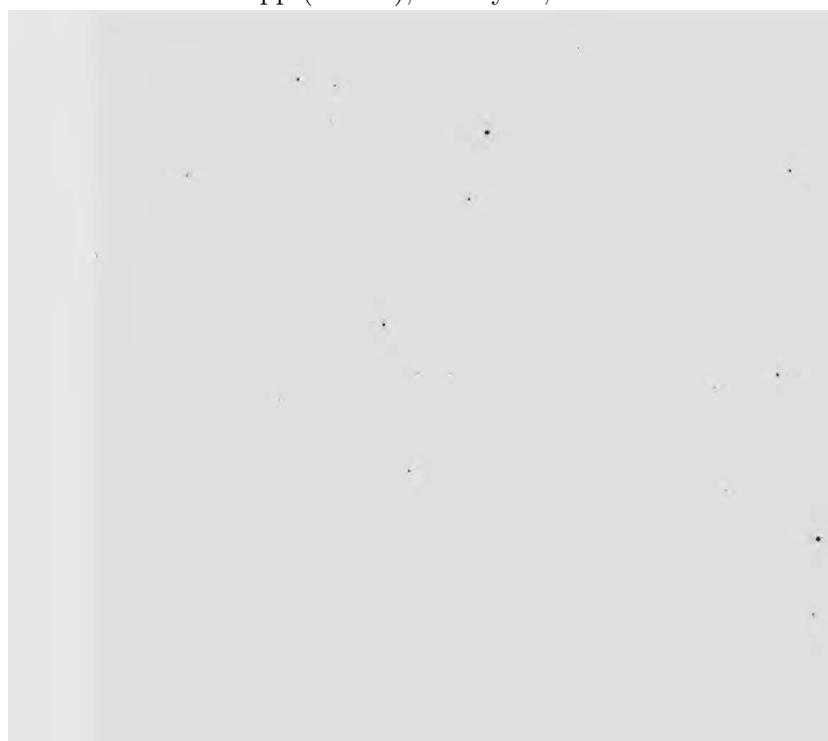


Figura 4.67: Transmisión progresiva de la imagen “29jul26” usando SPIHT+U(S+P)T (cont.). Toda la imagen es presentada.

SPIHT frente a JPEG (*lossy*)

Por último, vamos a comparar el compresor progresivo SPIHT+U(S+P)T con el estándar en compresión (*lossy*) de imágenes JPEG [55, 117]. La evaluación ha consistido en comprimir y descomprimir²⁴ una imagen tanto como fuera posible usando JPEG y luego reconstruir la misma imagen usando SPIHT justamente al mismo nivel de compresión.

Los resultados se muestran en las Figuras 4.68, 4.69, 4.70 y 4.71 para las imágenes de 8 bpp, quedando patente que sobre las imágenes de tono continuo, SPIHT es claramente superior a la versión “*lossy*” de JPEG basada en la transformada coseno. La Figura 4.72 sólo muestra la compresión de la imagen “29jul25” para ambos algoritmos. En este caso, SPIHT también genera una imagen de mayor calidad aunque en esta ocasión las losetas provocadas por JPEG son más pequeñas debido a las dimensiones de la imagen.

²⁴Se ha usado el programa *xv* de John Bradley. La calidad de compresión es el 1% (la mínima seleccionable).



Figura 4.68: Comparativa entre JPEG-lossy (arriba) y SPIHT (abajo) para “lena”. El nivel de compresión es 0.14 bpp (51:1).



Figura 4.69: Comparativa entre JPEG-lossy (arriba) y SPIHT (abajo) para “bárbara”. El nivel de compresión es 0.17 bpp (47:1).



Figura 4.70: Comparativa entre JPEG-lossy (arriba) y SPIHT (abajo) para “boats”. El nivel de compresión es 0.14 bpp (41:1).



Figura 4.71: Comparativa entre JPEG -lossy (arriba) y SPIHT (abajo) para “zelda”. El nivel de compresión es 0.12 bpp (67:1).

“29jul0025” comprimida mediante JPEG, 0.10 bpp (160:1)



“29jul0025” comprimida mediante SPIHT, 0.10 bpp (160:1)



Figura 4.72: Comparativa entre JPEG y SPIHT para “29jul0025”. Toda la imagen es presentada.

4.17 Sobre la implementación software del transmisor

A la hora de implementar mediante *software* un transmisor progresivo de imágenes es necesario cuidar algunos aspectos que son fundamentales de cara al rendimiento real que es posible conseguir.

El transmisor ejecuta fundamentalmente el algoritmo SPIHT tras haber efectuado la U(S+P)T. La transmisión del *code-stream* entre máquinas remotas se realiza usando *sockets*. Aunque el sistema operativo implementa estos sistemas de comunicación entre procesos usando algún *buffer* para desacoplar el emisor del canal, es interesante jugar con otro *buffer* de memoria sobre el que el compresor va escribiendo siempre que sea posible. Cuando este *buffer* se llena, se transmite por completo a través del *socket*.

Por otra parte, hemos encontrado que para minimizar el tiempo que transcurre desde que el usuario en la máquina remota lanza el comando de visualización hasta que la primera imagen aparece, el tamaño de los paquetes transmitidos debe ser relativamente pequeño. Un tamaño de 32 bytes nos ha dado un buen resultado. Para minimizar los casos en que el compresor fuera el cuello de botella, el compresor corre en un hilo (*thread*) y el envío de los datos a través del canal corre sobre otro.

El receptor es la pieza más importante en el sistema de transmisión y visualización, puesto que es el responsable directo de presentar al usuario la máxima cantidad de refinamientos posibles. Su misión es descomprimir el *code-stream* que ha llegado y efectuar la U(S+P)T inversa. Computacionalmente hablando, el receptor puede llegar a ser mucho más costoso que el emisor, dependiendo del número de imágenes por segundo presentadas ya que por cada una de ellas hay que efectuar una transformada inversa.

Para aprovechar al máximo la potencia de la máquina receptora, SPIHT y la transformada inversa corren en dos hilos diferentes. Así, independientemente de la cantidad de bits que han llegado, el número de imágenes presentadas por unidad de tiempo es maximizado. El resultado es que el usuario, independientemente de la velocidad del canal, ve en cada instante, dependiendo de la potencia de la máquina y de su carga, una versión lo más actualizada posible de la imagen recibida.

Finalmente, indicar que el receptor corre en un tercer hilo, al igual que el emisor, las tareas de gestión del *socket* que le entrega los datos.

4.18 Resumen

Este capítulo ha analizado, extensamente, diferentes alternativas a la transmisión progresiva en fidelidad de imágenes. En este sentido se ha estudiado e implementado uno de los compresores y transmisores de imágenes más eficientes: SPIHT+U(S+P)T. Finalmente se ha realizado una evaluación exhaustiva sobre las imágenes de prueba del Apéndice A.

Esencialmente, el diseño de mejores compresores y transmisores progresivos de imágenes de tono continuo pasa por dos puntos fundamentales, los cuales han sido profundamente expuestos: (1) la transformada usada para descorrelacionar la imagen y (2) el codificador

progresivo.

La transformación más eficiente para la compresión de imágenes parece ser la transformada wavelet discreta, al contrario de lo que indican todavía los estándares en compresión de imágenes que usan la transformada coseno. El diseño de filtros mejores, realmente bidimensionales y tal vez especialmente diseñados para un tipo de imagen en concreto (astronómicas, médicas, etc.) es un aspecto muy determinante. Hablamos de filtros reversibles (numéricamente hablando) capaces de proporcionar reconstrucciones exactas para que no existan pérdidas de información.

Pensando en una posible línea de trabajo futuro, un aspecto importante a considerar y que no se ha tenido en cuenta en el diseño de un filtro es su localidad espacial. La tarea fundamental del receptor (a la que más tiempo dedica) es a la reconstrucción progresiva de la señal a partir de una serie de coeficientes que han cambiado respecto de la reconstrucción anterior. La forma más sencilla (aunque no eficiente) de encontrar la imagen es realizar la transformada completa, como si todos los coeficientes fueran nuevos.

Realmente la complejidad del algoritmo de descodificación es muy inferior a esta tarea porque muy pocos coeficientes son descodificados entre reconstrucciones sucesivas (si la velocidad de transmisión es suficientemente lenta en comparación con la potencia de cálculo del receptor). El que el filtro sea local significa que si un coeficiente es modificado, sólo una sección de la imagen reconstruida debe ser recalculada, lo que conlleva una disminución del tiempo de cálculo de la transformada inversa. Además, deberían desarrollarse algoritmos que permitieran efectuar sólo los cálculos provocados por la llegada de un único bit de un coeficiente, que es el paso de mejora más suave que es posible establecer. En las implementaciones realizadas esto no se cumple.

El otro factor de peso es el codificador progresivo de coeficientes espectrales. En este documento sólo se ha estudiado SPIHT porque se considera actualmente como unos de los codecs que mejores *ratios* tasa de compresión/tasa de transferencia proporciona. Pero es evidente que nuestra implementación de SPIHT puede ser mejorada:

- Por una parte, la inclusión de la codificación aritmética cuando se usa como transmisor progresivo es conveniente, especialmente en el caso de las imágenes astronómicas.
- Por otro lado, sería interesante explotar también la correlación entre bandas de frecuencia situadas en el mismo nivel de la descomposición wavelet.

En ambos casos, las tasas de transferencia del codec van a disminuir, pero cuando se transmiten imágenes el canal es casi siempre el cuello de botella, lo que significa que los tiempos de transmisión en realidad van a disminuir.

Otro aspecto fundamental es la minimización del tiempo que transcurre desde que ejecutamos el receptor progresivo solicitando una transferencia de una imagen que en el emisor no se encuentra comprimida usando SPIHT, hasta que la primera imagen es mostrada en el receptor. Este tiempo está dedicado casi en su totalidad al cálculo de los coeficientes wavelet que necesitan una transformación completa y pueden ser importantes si la imagen es muy grande.

Una forma de reducirlo consistiría en usar una transformada que calculara los coeficientes wavelet por planos. Debemos hacer de nuevo hincapié que el tiempo total de cálculo de todos los planos no es relevante si se está transmitiendo simultáneamente, ya que dicho tiempo se solapa con el de transmisión. Si el tiempo de cálculo del primer plano de bits se minimiza, esta alternativa sería la más eficiente.

Conclusiones y Principales Aportaciones

En esta tesis se ha abordado el problema de la compresión de datos bajo la perspectiva de la transmisión de información a través de canales “lentos”, tipo *Internet*. El resultado práctico de este trabajo se concreta en el desarrollo de un modelo que engloba tanto los aspectos de la compresión de imágenes, como los de su transmisión y visualización. En este sentido el producto resultante de la investigación realizada es un compresor eficiente que además permite la transmisión de información de forma simultánea al propio proceso de compresión. De esta forma, compresión, transmisión, descompresión y visualización son procesos que se llevan a cabo en paralelo, siguiendo un esquema segmentado o *pipeline*.

Las principales aportaciones de este trabajo de tesis se concretan en los siguientes apartados:

- Se ha propuesto una metodología de evaluación de compresores sin pérdidas que combina medidas de los niveles y velocidad de compresión/descompresión.
- Se ha realizado una revisión exhaustiva de los métodos de compresión basados en diccionarios y se ha propuesto una implementación del algoritmo LZW usando búsqueda binaria y *hashing*.
- En el contexto de la compresión entrópica se ha desarrollado e implementado la transformada basada en predicción (PBT). Este algoritmo ha demostrado ser muy útil como herramienta descorrelacionadora en las imágenes de tono continuo alcanzando los mejores niveles de compresión conocidos por un compresor de texto. El coste computacional es alto, aunque inferior al de otros compresores similares. Sin embargo, esto no es un inconveniente en transmisión de datos porque permite procesar las secuencias de símbolos como corrientes.
- Se ha llevado a cabo una análisis y evaluación de los compresores de texto estándar junto con las soluciones propuestas en este trabajo.
- En el contexto de la compresión de imágenes se han analizado y evaluado los compresores secuenciales de imágenes *lossless* de última generación.

- Se ha presentado un predictor basado en una red neuronal entrenada mediante el algoritmo de retropropagación del error (BPN). Los resultados muestran que la adaptación del predictor es excelente puesto que parte de un desconocimiento total del contenido de la imagen, superando en algunas de ellas al predictor usado en CALIC que implementa uno de los predictores algorítmicos más sofisticados que existen.
- En el marco de la transmisión progresiva de imágenes, se ha llevado a cabo una evaluación del uso de transformadas ortogonales como codificadores de información que permite expresar el máximo de información usando el mínimo número de coeficientes transformados. Los resultados demuestran que el uso de transformadas basadas en wavelets superan a las tradicionales tales como DCT y WHT.
- Se han evaluado diferentes estrategias de transmisión de los coeficientes transformados. Los resultados muestran claramente que la política de máxima potencia es la más eficiente.
- Se ha implementado y evaluado una estrategia de transmisión de coeficientes wavelet por máxima potencia basada en una partición jerárquica de los árboles de orientación espacial construidos sobre los planos de bits (SPIHT).
- Se propone un algoritmo de compresión progresiva de imágenes basado en una transformada wavelet que junto con el uso de SPIHT ha demostrado ser muy eficiente como comprresor y como transmisor progresivo.

Perspectivas Futuras

Al final de los Capítulos 2, 3 y 4 se han presentado las líneas de trabajo que podrían continuarse. Sin embargo, se ha querido incluir este último apartado las que hemos considerado más interesantes de cara especialmente a mejorar el diseño de transmisores progresivos de imágenes, que es el problema que actualmente nos ocupa:

- Minimización del tiempo inicial de visualización solapando los procesos de transformación y transmisión.
- Determinación de transformadas *wavelet* reversibles más eficientes que permitan aumentar los niveles de compresión.
- Aprovechamiento de la correlación entre bandas del mismo nivel de la descomposición *wavelet* para aumentar las tasas de compresión.
- Determinación de *wavelets* realmente locales, que permitan que la reconstrucción progresiva de la imagen tenga el mismo coste computacional que una única transformada inversa.
- Implementación de una transformada *wavelet* tridimensional que permita la transmisión progresiva de volúmenes.

Apéndice A

Imágenes de Prueba

En este apéndice se presentan las imágenes que se han usado para comparar los diferentes codificadores. Las imágenes de 8 bits/punto y sus histogramas se muestran en las figuras A.1, A.2, A.3 y A.4. Son imágenes típicas en las baterías de imágenes para probar los algoritmos de compresión.

Las imágenes astronómicas han sido amablemente cedidas por el Centro Astronómico Hispano-Alemán de Calar Alto y se muestran en negativo (junto con sus histogramas también) en las figuras A.5, A.6, A.7 y A.8. Son imágenes típicas generadas a partir del espectro visible. Su formato estándar es FITS (*Flexible Image Transport System*) y como puede apreciarse son bastante más voluminosas, todas con 16 bits/punto.

Para todas las imágenes, en la tabla A.1 se muestra la entropía (ver sección 1.1.4), el nivel de redundancia (ver sección 1.1.5), la media de sus puntos, el nivel de gris mínimo y el máximo.

Tabla A.1: Algunos parámetros interesantes de las imágenes de prueba.

imagen	dimensiones	bpp	entropía	redundancia	min	max	media
lena	512 × 512	8	7.45	6.5%	25	246	124.2
bárbara	512 × 512	8	7.47	6.6%	14	238	112.4
boats	512 × 512	8	7.12	11.0%	0	239	136.1
zelda	512 × 512	8	7.27	9.1%	0	187	91.2
media			7.33	8.3%			
B0100	1024 × 1083	16	6.38	60.1%	96	12556	164.7
29jul24	1024 × 1148	16	7.18	55.1%	0	62194	538.0
29jul25	1024 × 1148	16	9.04	43.5%	0	62231	3525.5
29jul26	1024 × 1148	16	8.89	44.4%	0	62210	2220.9
media			7.87	50.8%			

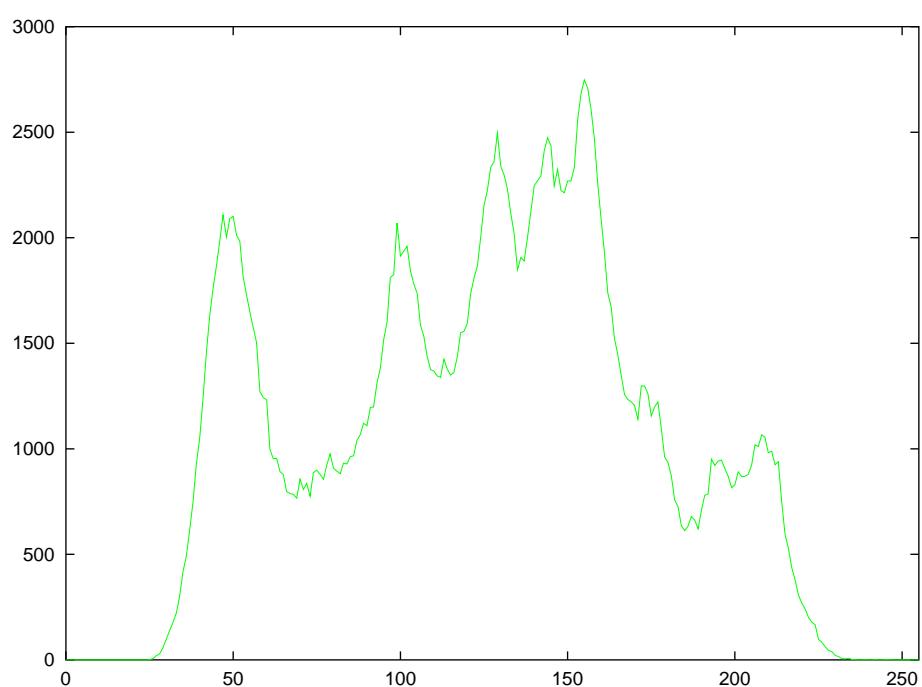


Figura A.1: Imagen "lena", 512×512 puntos, 8 bpp.

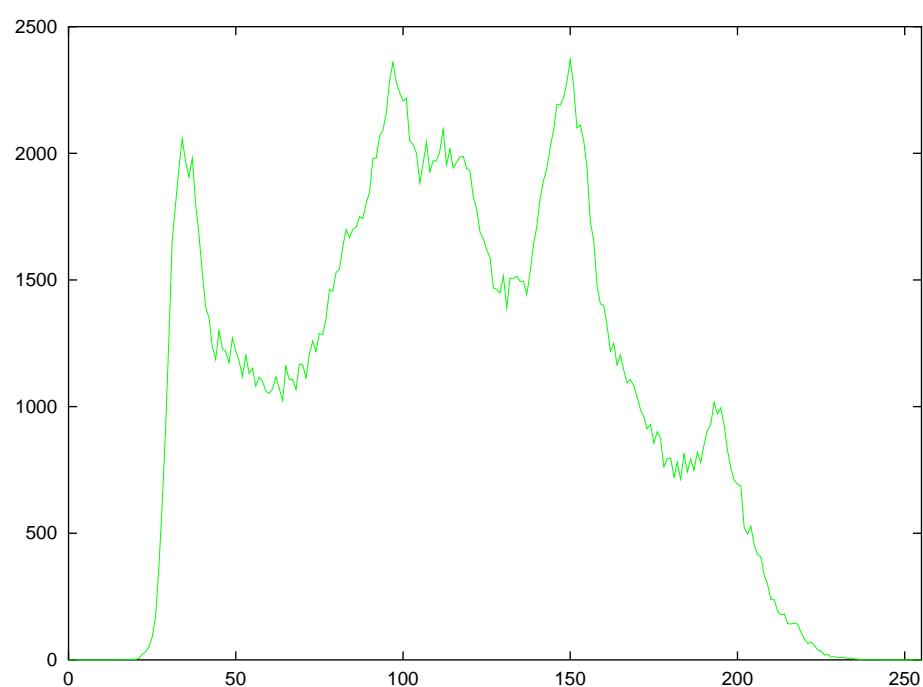


Figura A.2: Imagen “bárbara”, 512×512 puntos, 8 bpp.

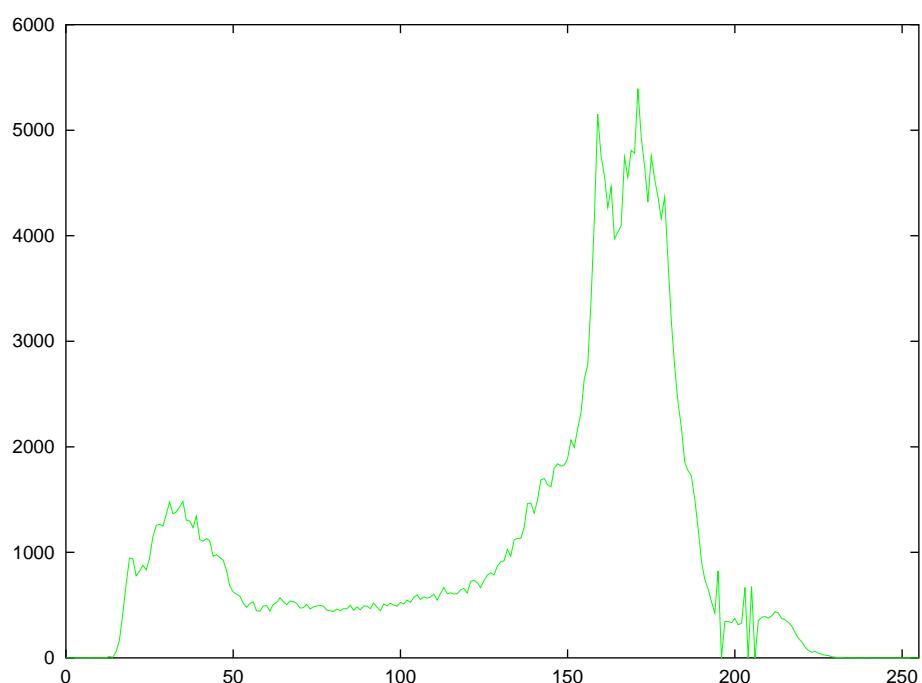


Figura A.3: Imagen “boats”, 512×512 puntos, 8 bpp.

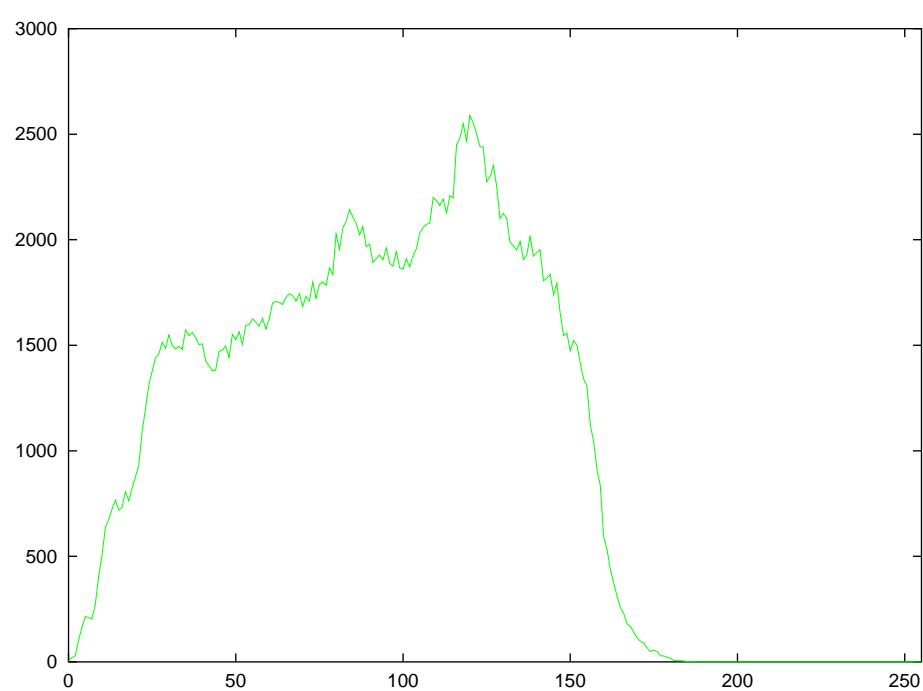


Figura A.4: Imagen “zelda”, 512×512 puntos, 8 bpp.

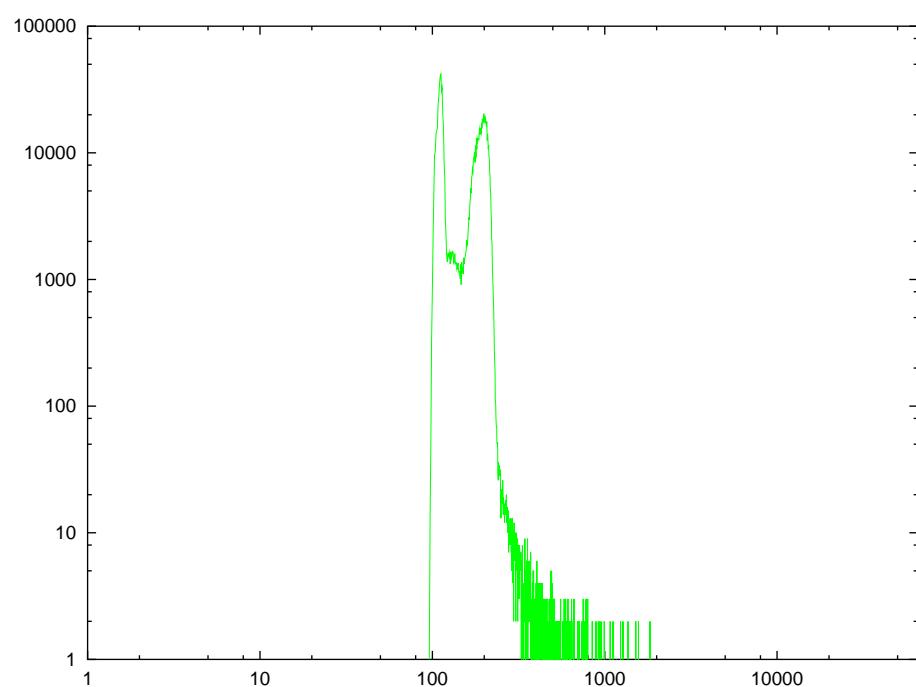


Figura A.5: Imagen “B0100”, 1024×1083 puntos, 16 bpp.

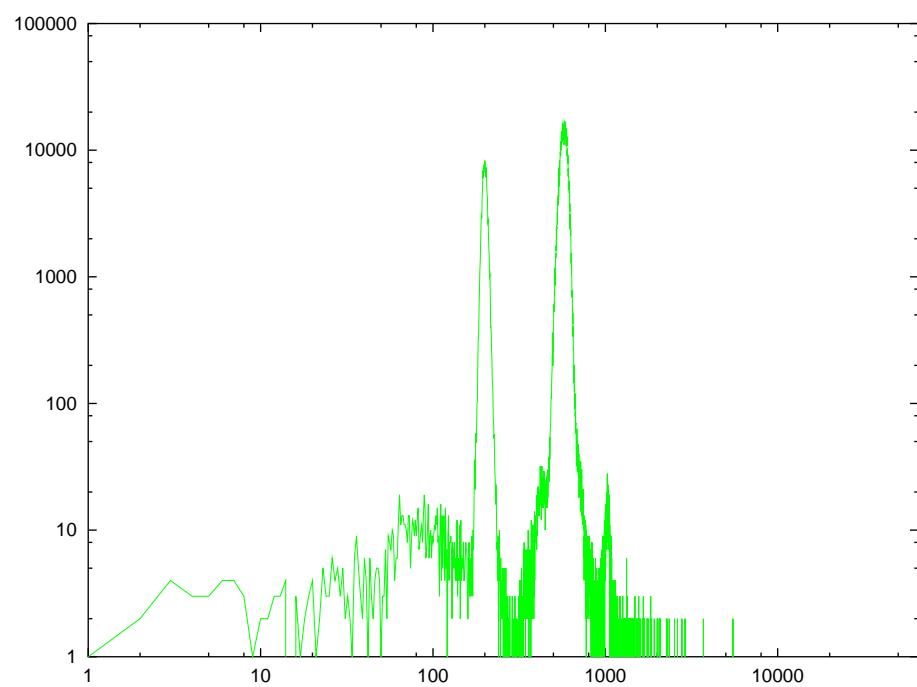
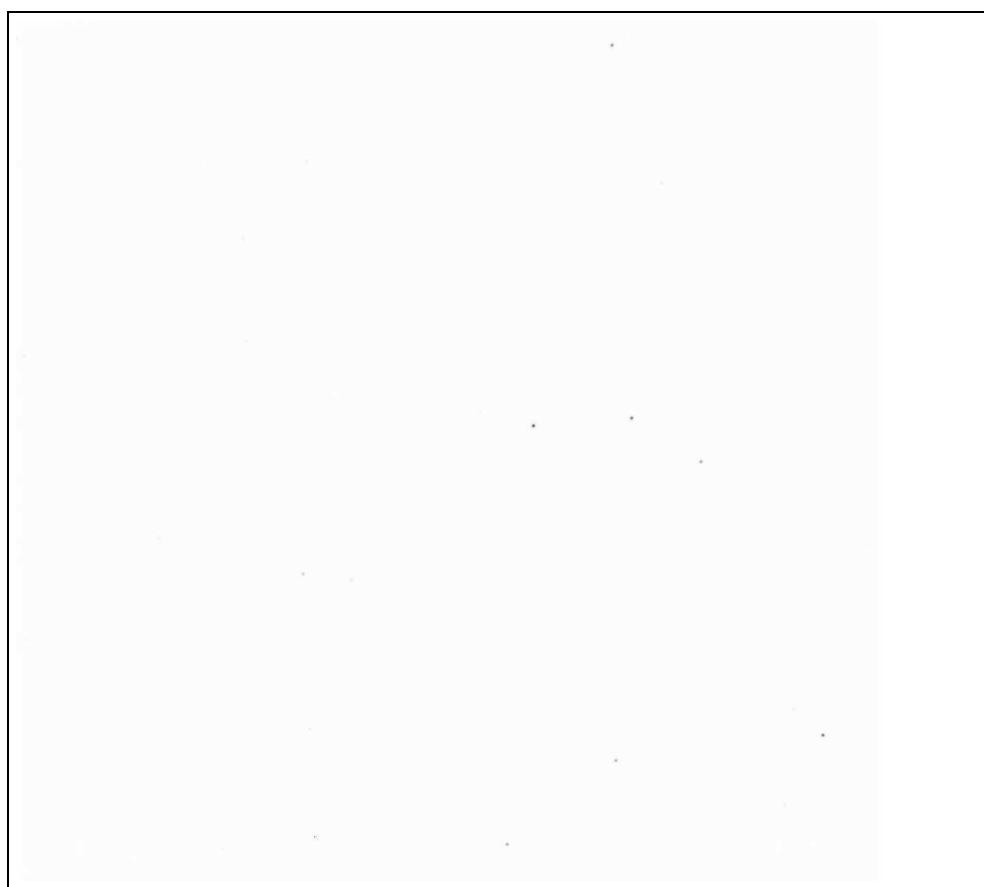


Figura A.6: Imagen “29jul24”, 1024×1148 puntos, 16 bpp.

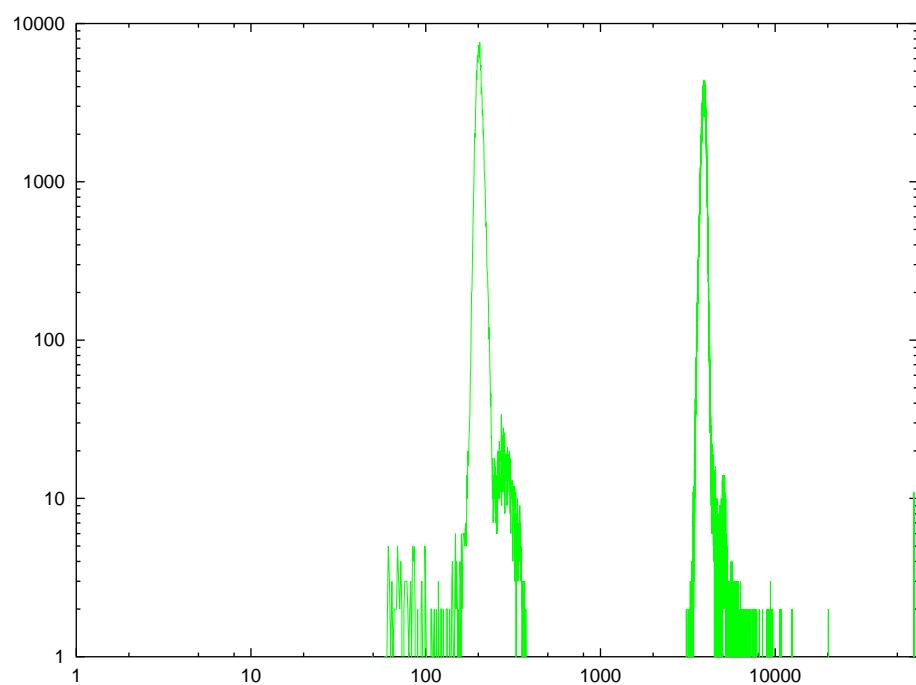


Figura A.7: Imagen “29jul25”, 1024×1148 puntos, 16 bpp.

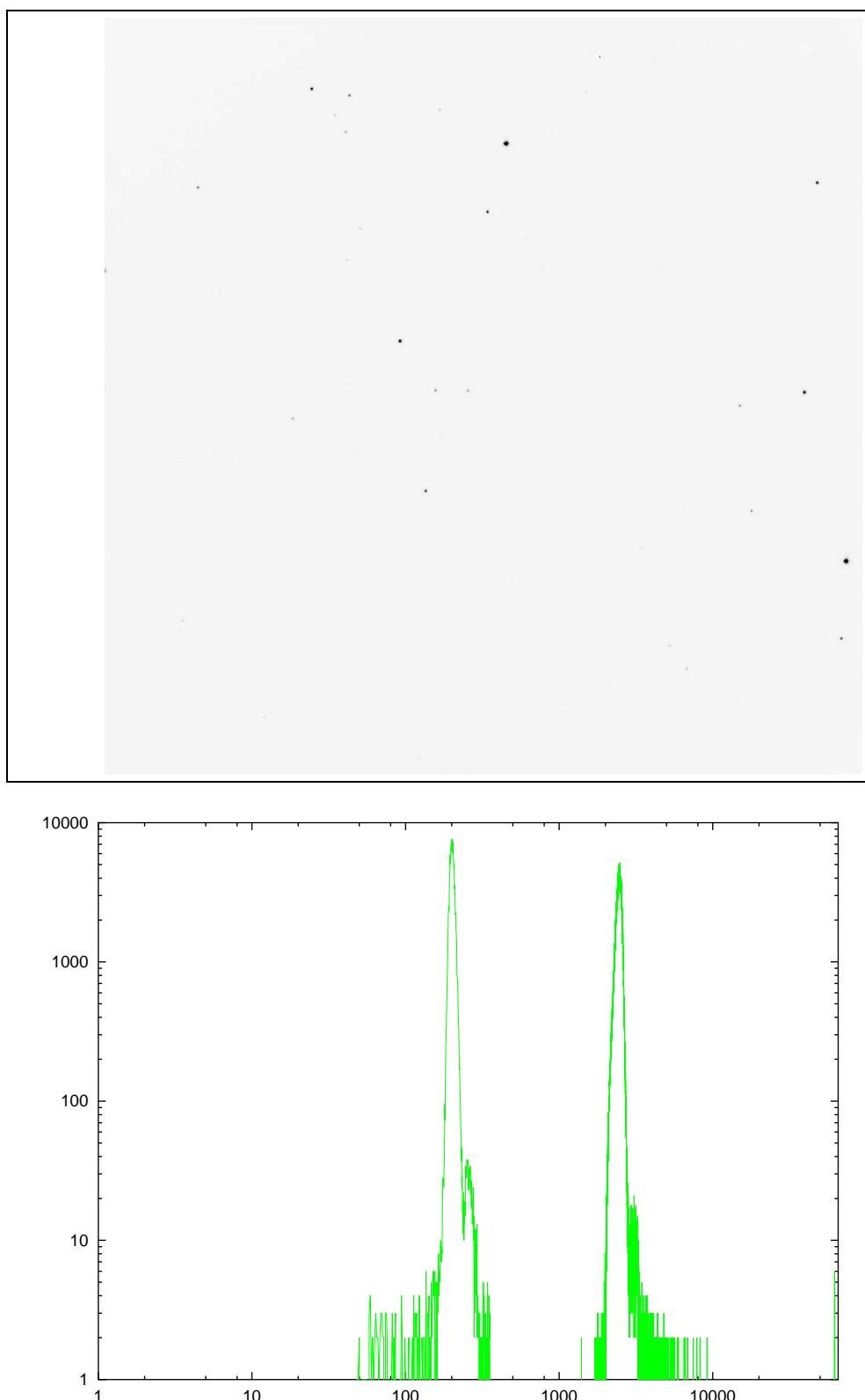


Figura A.8: Imagen “29jul26”, 1024×1148 puntos, 16 bpp.

Apéndice B

Demostraciones

Este apéndice contiene algunas demostraciones matemáticas y algoritmos usados en la tesis pero que no son estrictamente necesarios para su comprensión.

B.1 Algoritmo rápido para el cálculo de la WHT

Cuando la transformación ha sido definida para imágenes que tienen lados potencia de dos, es posible conseguir una versión rápida de la transformación unidimensional (FWHT: *Fast WHT*). A este algoritmo se le conoce como el método de desdoblamiento sucesivo y fue usado por primera vez por Cooley y Tukey en la aceleración de la transformada de Fourier [19].

El método de desdoblamiento sucesivo consiste básicamente en dividir el vector de muestras $f[x]$ en dos vectores iguales de longitud mitad, de forma que la transformada del vector completo puede ser calculada a partir de las transformadas de los dos vectores y en este proceso, algunas operaciones se pueden ahorrar.

La transformada unidimensional puede ser calculada independientemente para los elementos pares e impares de $f[x]$ como

$$F[u] = \sum_{x=0}^{N/2-1} f[2x]k(2x, u) + \sum_{x=0}^{N/2-1} f[2x+1]k(2x+1, u) \quad (\text{B.1})$$

donde los núcleos responden a las expresiones

$$\begin{aligned} k(2x, u) &= (-1)^{\sum_{i=0}^{n-1} b_i(2x)b_i(u)} \\ k(2x+1, u) &= (-1)^{\sum_{i=0}^{n-1} b_i(2x+1)b_i(u)} \end{aligned}$$

y donde $2x$ es de la forma X...X0 y $2x+1$ es de la forma X...X1. Por tanto

$$k(2x+1, u) = (-1)^{\sum_{i=1}^{n-1} b_i(2x+1)b_i(u) + b_0(2x+1)b_0(u)}.$$

Como $b_0(2x + 1) = 0$ y $\sum_{i=1}^{n-1} b_i(2x + 1)b_i(u) = \sum_{i=0}^{n-1} b_i(2x)b_i(u)$, tenemos que

$$k(2x + 1, u) = (-1)^{b_0(u) \sum_{i=0}^{n-1} b_i(2x)b_i(u)}.$$

Descomponiendo el exponente, resulta que

$$k(2x + 1, u) = (-1)^{b_0(u)} \cdot (-1)^{\sum_{i=0}^{n-1} b_i(2x)b_i(u)}$$

de donde llegamos a que

$$k(2x + 1, u) = (-1)^{b_0(u)} \cdot k(2x, u). \quad (\text{B.2})$$

Sustituyendo esta expresión en la ecuación (B.1) se obtiene

$$F[u] = \sum_{x=0}^{N/2-1} f(2x)k(2x, u) + (-1)^{b_0(u)} \sum_{x=0}^{N/2-1} f[2x + 1, u]k(2x, u). \quad (\text{B.3})$$

Nótese que la expresión (B.3) requiere sólo $2 \times (N/2)^2$ sumas frente a las N^2 de la expresión (4.4). El método de desdoblamiento sucesivo puede ser aplicado recursivamente a los vectores de tamaño mitad, obteniéndose el algoritmo para la transformada rápida de Walsh-Hadamard cuya complejidad es proporcional a $N \times \log_2(N)$.

Por último, de la ecuación (B.3) es posible extraer la forma de la mariposa usada en el algoritmo rápido. Cuando $N = 2$, resulta que

$$\begin{aligned} F[0] &= f[0] + f[1] \\ F[1] &= f[0] - f[1]. \end{aligned} \quad (\text{B.4})$$

El resto de mariposas se forman de forma idéntica a como lo hace la FFT (*Fast Fourier Transform*) usando el algoritmo de decimación en el tiempo (DIT) [72]. Finalmente, si se desea situar los coeficientes en su posición espectral (frecuencia) es necesario aplicar el algoritmo de ordenación *bit-reversal*, que consiste en permutar los coeficientes en función de sus índices en un vector unidimensional, complementando sus bits. La figura B.1 muestra un ejemplo sencillo de ésto.

Tras la ordenación, los coeficientes más próximos a la esquina superior izquierda se corresponden con las bajas frecuencias y los coeficientes situados en la esquina inferior derecha identifican las altas frecuencias.

B.2 Algoritmo rápido para el cálculo de la DCT

Se han ideado muchos algoritmos rápidos para la DCT. Todos ellos arrojan una complejidad $N \times \log_2(N)$ [86].

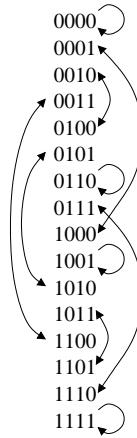


Figura B.1: Un ejemplo de ordenación de 16 números usando el algoritmo *bit-reversal*.

Existen básicamente dos tipos de métodos: los directos y los indirectos. Estos últimos usan otra transformada rápida conocida (por ejemplo la FFT) para efectuar la FCT [80]. La versión presentada es un método directo basado en método de desdoblamiento sucesivo [6].

La transformada directa de la expresión (4.8) puede escribirse como

$$F[u] = \sum_{x=0}^{N-1} f[x] \cos\left(\pi \frac{(2x+1)u}{2N}\right) \quad (\text{B.5})$$

considerando que $\sqrt{\frac{2}{N}}e(u)$ es simplemente un factor de escala que puede ser aplicado al final de la transformación. Vamos a tratar de ver como el método de desdoblamiento sucesivo permite calcular la DCT usando un número de operaciones menor. El desarrollo aquí mostrado pertenece a Hou [41]. En primer lugar se realiza el cambio

$$\begin{aligned} f'[x] &= f[2x] \\ f'[N-x-1] &= f[2x+1] \end{aligned} \quad (\text{B.6})$$

para $x = 0, 1, \dots, N/2 - 1$. Este cambio expresa (B.5) en elementos pares e impares resultando que

$$F[u] = \sum_{x=0}^{N-1} f'[x] \cos\left(\pi \frac{(4x+1)u}{2N}\right). \quad (\text{B.7})$$

Hou usó el esquema de decimación en frecuencia (DIF) [72] para particionar $f'[x]$ en dos subsecuencias $f_1[x]$ y $f_2[x]$ que contienen respectivamente la primera mitad y la segunda mitad de los datos de $f'[x]$, obteniéndose así que

$$\begin{aligned} f_1[x] &= f'[x] \\ f_2[x] &= f[x+N/2] \end{aligned} \quad (\text{B.8})$$

para $x = 0, 1, \dots, N/2 - 1$. Realizando esta sustitución en (B.7) resulta que

$$\begin{aligned} F[u] &= \sum_{x=0}^{N/2-1} f_1[x] \cos\left(\pi \frac{(4x+1)u}{2N}\right) + \sum_{x=0}^{N/2-1} f_2[x] \cos\left(\pi \frac{(4(x+N/2)+1)u}{2N}\right) \\ &= \sum_{x=0}^{N/2-1} f_1[x] \cos\left(\pi \frac{(4x+1)u}{2N}\right) + (-1)^u \sum_{x=0}^{N/2-1} f_2[x] \cos\left(\pi \frac{(4x+1)u}{2N}\right). \end{aligned} \quad (\text{B.9})$$

Como podemos apreciar en (B.9), es posible el cálculo de la DCT mediante un algoritmo de complejidad $2 \times (N/2)^2$, frente a N^2 del algoritmo original. Cuando el método de desdoblamiento sucesivo se aplica recursivamente a ambos subvectores, resulta el algoritmo FCT que arroja un número de operaciones proporcional a $N \times \log_2(N)$.

Bibliografía

- [1] RICOH CREW Image Compression Standard Version 1.0 (Beta). Technical report, RICOH Silicon Valley, Inc., 2882 Sand Hill Road, Suite 115, Menlo Park, CA 94025, April 1999.
- [2] N. Abramson. *Information Theory and Coding*. New York, McGraw-Hill, 1963.
- [3] E.H. Adelson, E. Simoncelli, and R. Hingorani. Orthogonal Pyramid Transforms for Image Coding. *Proceedings of the SPIE*, 845:50–58, 1987.
- [4] N.T. Ahmed, T. Natarajan, and K.R. Rao. Discrete Cosine Transform. *IEEE Transactions on Computers*, 23(1):90–93, 1974.
- [5] M. Antonini, M. Barlaud, P. Mathieu, and I. Daubechies. Image Coding Using Wavelet Transform. *IEEE Transactions on Image Processing*, 1(2):205–220, 1992.
- [6] F. Argüello and E.L. Zapata. Fast Cosine Transform based on the Successive Doubling Method. *J. IEE Electronics Letters*, 26(19):1616–1618, 1990.
- [7] R.B. Arps, T.K. Truong, D.J. Lu, R.C. Pasco, and T.D. Friedman. A Multi-Purpose VLSI Chip for Adaptive Data Compression of Bilevel Images. *IBM J. Res. Develop*, 32(6):775–795, 1988.
- [8] R.B. Ash. *Information Theory*. Dover Publications, 1965.
- [9] T.C. Bell, J.G. Cleary, and I.H. Witten. *Text Compression*. Prentice Hall, 1990.
- [10] C. Bloom. Solving the Problems of Context Modeling. Technical report, California Institute of Technology.
- [11] H. Blume and A. Fand. Medical Imaging III: Image Capture and Display. *Proceedings SPIE of Medical Imaging*, 1091:2–2, 1989.
- [12] R.W. Buccigrossi and E.P. Simoncelli. Image Compression via Joint Statistical Characterization in the Wavelet Domain. *IEEE Transactions on Image Processing*, 8(12):1688–1701, December 1999.

- [13] M. Burrows and D.J. Wheeler. A Block-Sorting Lossless Data Compression Algorithm. Technical Report 124, Digital Systems Research Center, 1994.
- [14] C. Chrysafis and A. Ortega. Line-Based, Reduced Memory, Wavelet Image Compression. *IEEE Transactions on Image Processing*, 9(3):378–389, March 2000.
- [15] J.G. Cleary and W.J. Teahan. Unbounded Length Contexts for PPM. *The Computer Journal*, 36(5), 1993.
- [16] J.G. Cleary and I.H. Witten. A Comparison of Enumerative and Adaptive Codes. *IEEE Transactions on Information Theory*, 30(2):306–315, 1984.
- [17] J.G. Cleary and I.H. Witten. Data Compression using Adaptive Coding and Partial String Matching. *IEEE Transactions on Communications*, 4(32):396–402, 1984.
- [18] Compuserve Incorporated. *Graphics Interchange Format (GIF) Specification*, June 1987.
- [19] J.W. Cooley and J.W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(4):297–301, 1965.
- [20] G.V. Cormack and R.N.S. Horpool. Data Compression Using Dynamic Markov Modeling. *The Computer Journal*, 30(1), 1987.
- [21] T.M. Cover and J.A. Thomas. *Elements of Information Theory*. Wiley Interscience, 1991.
- [22] I. Daubechies. Orthonormal Bases of Compactly Supported Wavelets. *Communications on Pure and Applied Mathematics*, 41:909–996, 1988.
- [23] I. Daubechies. The Wavelet Transform, Time-Frequence Localization and Signal Analisys. *IEEE Transactions on Information Theory*, 36:961–1005, 1990.
- [24] R.A. DeVore, B. Jawert, and B.J. Lucier. Image Compression Through Wavelet Transform Coding. *IEEE Transactions on Information Theory*, 38(2):719–746, March 1992.
- [25] D.L. Duttweiler and C. Chamzas. Probability Estimation in Arithmetic and Adaptive-Huffman Entropy Coders. *IEEE Transactions on Image Processing*, 4(3):237–246, 1995.
- [26] P.M. Embree and B. Kimble. *C Language Algorithms for Digital Signal Processing*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.
- [27] P. Fenwick. Improvements to the Block Sorting Text Compression Algorithm. Technical Report 120, Dept. of Computer Science, The University of Auckland, 1995.

- [28] P. Fenwick. Block Sorting Text Compression – Final Report. Technical Report 130, Dept. of Computer Science, The University of Auckland, 1996.
- [29] P. Fenwick. Symbol Ranking Text Compression with Shannon Recodings. *Journal of Universal Computer Science*, 3(2):70–85, 1997.
- [30] J. Gailly. Compression FAQ. <http://www.cis.ohio-state.edu/hypertext/faq/usenet/compression-faq>.
- [31] J. Gailly. gzip 1.2.4. <http://www.gzip.org>.
- [32] S.W. Golomb. Run-Length Encodings. *IEEE Transactions on Information Theory*, 12:399–401, 1966.
- [33] R.C. Gonzalez and R.E. Woods. *Digital Image Processing*. Addison Wesley, 1992.
- [34] M. Guazzo. A General Minimum-Redundancy Source-Coding Algorithm. *IEEE Transactions on Information Theory*, 26:15–25, 1980.
- [35] A. Haar. Zur Theorie der orthogonalen Funktionen-Systeme. *Mathematische Annalen*, 69:331–371, 1910.
- [36] D. Hamaker. Compress and Compact Discussed Further. *Communications of the ACM*, 31:1139–1140, 1988.
- [37] A. Hartman and H. Rodeh. *Combinatorial Algorithms on Words: Optimal Parsing of Strings*. Springer-Verlag, 1985.
- [38] V.K. Heer and H.E. Reinfelder. A Comparision of Reversible Methods for Data Compression. *Proceedings SPIE of Medical Imaging*, 1233(IV):354–365, 1990.
- [39] G. Held. *Data and Image Compression*. Willey, 1996.
- [40] H. Hirvola. ha 0.999. <ftp://sunsite.unc.edu/pub/Linux/utils/compress/ha0999p-linux.tar.gz>.
- [41] H.S. Hou. A Fast Recursive Algorithm for Computing the Discrete Cosine Transform. *IEEE ASSP*, 35(10):1455–1461, 1987.
- [42] P.G. Howard. *The Design and Analysis of Efficient Lossless Data Compression Systems*. PhD thesis, Department of Computer Science, Brown University, 1993.
- [43] P.G. Howard and Vitter J.S. Fast and Efficient Lossless Image Compression (FE-LICS). In *Data Compression Conference (DCC)*, pages 351–360, 1993.
- [44] P.G. Howard and J.S. Vitter. Analysis of Arithmetic Coding for Data Compression. *Information Processing and Management*, 28(6):749–763, 1992.

- [45] P.G. Howard and J.S. Vitter. Practical Implementations of Arithmetic Coding. In *Image and Text Compression*, pages 765–779. Kluwer Academic Publishers, 1992.
- [46] P.G. Howard and J.S. Vitter. Design and Analysis of Fast Text Compression Based on Quasi-Arithmetic Coding. In *Data Compression Conference (DCC)*, pages 89–107, 1993.
- [47] C. Hsieh and Y. Liu. Fast Search Algorithms for Vector Quantization of Images Using Multiple Triangle Inequalities and Wavelet Transform. *IEEE Transactions on Image Processing*, 9(3):321–328, March 2000.
- [48] K. Huang and B. Smith. *Lossless JPEG Codec version 1.0*. Cornell University, June 1994.
- [49] D.A. Huffman. A Method for the Construction of Minimum Redundancy Codes. *Proceedings of the Institute of Radio Engineers*, 40:1098–1101, 1952.
- [50] ISO/IEC. *JPEG 2000 Committee Draft version 1.0*, cd15444-1 edition, December 1999.
- [51] A.K. Jain. A Fast Karhunen-Loève Transform for a Class of Random Process. *IEEE Transactions on Communications*, COM-24:1023–1029, 1975.
- [52] A.K. Jain. *Fundamentals of Digital Image Processing*. Prentice Hall, 1989.
- [53] The Joint Binary Experts Group (JBIG). *Recommendation T.82: Digital Compression and Coding of Continuous-tone Still Images*. International Telecommunication Union (ITU), March 1993.
- [54] R.L. Joshi, H. Jafarkhani, J.H. Kasner, T. R. Fischer, N. Farvardin, M.W. Marcellin, and R.H. Bamberger. Comparison of Different Methods of Classification in Subband Coding of Images. *IEEE Transactions on Image Processing*, 6(11):1473–1486, November 1999.
- [55] The Joint Photographic Experts Group (JPEG). *Recommendation T.81: Digital Compression and Coding of Continuous-tone Still Images*. International Telecommunication Union (ITU), September 1992.
- [56] The Joint Photographic Experts Group (JPEG). *FCD 14495, Lossless and Near-Lossless Coding of Continuous Tone Still Images (JPEG-LS)*. The International Standards Organization (ISO)/The International Telegraph and Telephone Consultative Committee (CCITT), July 1997.
- [57] D.E. Knuth. Dynamic Huffman Coding. *Journal of Algorithms*, 6(2):163–180, 1985.
- [58] L.G. Kraft. *A Device for Quantizing, Grouping and Coding Amplitude Modulated Pulses*. PhD thesis, MIT, Cambridge, 1949.

- [59] G.G. Langdon. A Note on the Ziv-Lempel Model for Compressing Individual Sequences. *IEEE Transactions on Information Theory*, 29(2):284–287, 1983.
- [60] G.G. Langdon. Probabilistic and Q-Coder Algorithms for Binary Source Adaption. In *Data Compression Conference (DCC)*, pages 13–22, 1991.
- [61] G.G. Langdon and J. Rissanen. Compression of Black-White Images with Arithmetic Coding. *IEEE Transactions on Communications*, 29(6):858–867, 1981.
- [62] R. Lippman. An Introduction to Computing with Neural Nets. *IEEE ASSP Magazine*, 3(4):4–22, April 1987.
- [63] S. Mallat. A Theory for Multiresolution Signal Decomposition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11:674–693, 1989.
- [64] M. Nelson and J. Gailly. *The Data Compression Book*. M&T Books, 1996.
- [65] B. McMillan. The Basic Theorems of Information Theory. *Ann. Math. Stat.*, 24(2):196–219, 1953.
- [66] B. McMillan. Two Inequalities Implied by Unique Decipherability. *IRE Transactions on Information Theory*, IT-2:115–116, 1956.
- [67] J. Miano. *Compressed Image File Formats: JPEG, PNG, GIF, XBM, BPM*. Addison Wesley, 1999.
- [68] J.L. Mitchell and W.B. Pennebaker. Optimal Hardware and Software Arithmetic Coding Procedures for the Q-Coder. *IBM J. Res. Develop*, 32:727–736, 1988.
- [69] A. Moffat. A Note on the PPM Data Compression Algorithm. Technical Report 88/7, Dept. Computer Science, University of Melbourne, 1988.
- [70] A. Moffat. Word-based Text Compression. *Software Practice and Experience*, 19(2):185–198, 1989.
- [71] A. Moffat. Implementing the PPM Data Compression Scheme. *IEEE Transactions on Communications*, 38(11):1917–1921, 1990.
- [72] A.V. Oppenheim and R.W. Schafer. *Discrete-Time Signal Processing*. Prentice Hall, 1989.
- [73] E. Ordentlich, M. Weinberger, and G. Seroussi. A Low-Complexity Modeling Approach for Embedded Coding of Wavelet Coefficients. Technical report, Hewlett-Packard Laboratories, Palo Alto, CA 94304, December 1997.
- [74] R.R. Osorio. *Algoritmos y Arquitecturas para la Codificación Aritmética: Explotación de la Localidad Utilizando Memorias Cache*. PhD thesis, Depto de Electrónica y Computación, Universidad de Santiago de Compostela, 1999.

- [75] R. Pasco. *Source Coding Algorithms for Fast Data Compression*. PhD thesis, Stanford University, 1976.
- [76] W.B. Pennebaker and J.L. Mitchell. Probability Estimation for the Q-Coder. *IBM J. Res. Develop.*, 32:737–752, 1988.
- [77] W.B. Pennebaker and J.L. Mitchell. Software Implementation of the Q-Coder. *IBM J. Res. Develop.*, 32:753–774, 1988.
- [78] W.B. Pennebaker, J.L. Mitchell, G.G. Langdon, and R.B. Arps. An Overview of the Basic Principles of the Q-Coder Adaptive Binary Arithmetic Coder. *IBM J. Res. Develop.*, 32:717–726, 1988.
- [79] W.K. Pratt. *Digital Image Processing*. John Wiley & Sons, Inc., 1991.
- [80] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in C*. Cambridge University Press, 1992.
- [81] M. Rabbani and P.W. Jones. *Digital Image Compression Techniques*. SPIE Opt. Eng. Press, Bellingham, Washington, 1991.
- [82] L. Rabiner and B. Gold. *Theory and Application of Digital Signal Processing*. Prentice-Hall, Englewood Cliffs, New Jersey, 1975.
- [83] M. Ramchandran, K. Vetterli and C. Herley. Wavelets, Subband Coding, and Best Bases. *Proceedings of the IEEE*, 84(4):541–560, April 1996.
- [84] S. Ranganath and H. Blume. Hierarchical Image Decomposition and Filtering Using the S-Transform. *Proceedings SPIE of Medical Imaging*, 941:50–58, 1988.
- [85] K.R. Rao and J.J. Hwang. *Techniques and Standards for Image, Video and Audio Coding*. Prentice Hall, 1996.
- [86] K.R. Rao and P. Yip. *Discrete Cosine Transforms, Algorithms, Advantages, Applications*. Academic Press Inc., 1990.
- [87] R.F. Rice. Some Practical Universal Noiseless Coding Techniques. Technical Report 79/22, Jet Propulsion Laboratory, 1979.
- [88] J. Rissanen and G.G. Langdon. Arithmetic Coding. *IBM J. Res. Develop.*, pages 146–162, 1979.
- [89] J. Rissanen and G.G. Langdon. Universal Modeling and Coding. *IEEE Transactions on Information Theory*, 27:12–23, 1981.
- [90] J.J. Rissanen. Generalized Kraft Inequality and Arithmetic Coding. *IBM J. Res. Develop.*, 20(198-203), 1976.

- [91] J.J. Rissanen. A Universal Data Compression System. *IEEE Transaction on Information Theory*, 29(5):656–664, 1983.
- [92] M.G. Roberts. *Local Order Estimating Markovian Analysis for Noiseless Source Coding and Authorship Identification*. PhD thesis, Stanford University, 1982.
- [93] G. Roelofs. *PNG: The Definitive Guide*. O'Reilly, 1999.
- [94] A. Rosenfeld and A.C. Kak. *Digital Picture Processing*. Academic Press, 1982.
- [95] F. Rubin. Arithmetic Stream Coding Using Fixed Precision Registers. *IEEE Transactions on Information Theory*, 25:672–675, 1979.
- [96] V.G. Ruiz, J.J. Fernández, and I. García. Lossless Compression for Electron Microscopy Images. In *Symposium Nacional de Reconocimiento de Formas y Análisis de Imágenes (SNRFAI)*, volume 2, pages 33–34, Bilbao, Spain, May 1999.
- [97] V.G. Ruiz and I. García. Una Solución Paralela a la Compresión de Imágenes. In *Nuevas Tendencias en la Informática: Arquitecturas Paralelas*. Universidad de Castilla-La Mancha, 1994.
- [98] V.G. Ruiz and I. García. Una Implementación del Algoritmo de Compresión LZW. In *I Jornadas de Informática*, pages 281–292, Puerto de la Cruz, Tenerife, Julio 1995.
- [99] V.G. Ruiz and I. García. A Lossy Data Compressor Based on the LZW Algorithm. In *International Conference on Signal Processing Applications and Technology (ICSPAT)*, pages 1002–1006, Boston, MA, USA, October 7-10 1996.
- [100] V.G. Ruiz and I. García. On the Computation of the Fast Hadamard Transform for Large Amounts of Data. In *Symposium Nacional de Reconocimiento de Formas y Análisis de Imágenes (SNRFAI)*, volume 2, pages 12–13, 1997.
- [101] V.G. Ruiz and I. García. Compresión de Texto basada en un Modelo Probabilístico de Orden 0 y un Codificador de Huffman. Technical Report 1, Depto de Arquitectura de Computadores y Electrónica, Universidad de Almería, 1998.
- [102] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning Representations by Back-Propagating Errors. *Nature*, 323:533–536, October 1986.
- [103] A. Said and W.A. Pearlman. Image Compression Using the Spatial-Orientation Tree. In *IEEE International Symposium on Circuits and Systems*, pages 279–282, Chicago, IL, 1993.
- [104] A. Said and W.A. Pearlman. Reversible Image Compression via Multiresolution Representation and Predictive Coding. In *Proceedings SPIE Conference on Visual Communications and Image Processing '93*, volume SPIE 2094, pages 664–674, Cambridge, MA, November 1993.

- [105] A. Said and W.A. Pearlman. A New Fast and Efficient Image Codec Based on Set Partitioning in Hierarchical Trees. *IEEE Transactions on Circuits and Systems for Video Technology*, pages 243–250, 1996.
- [106] A. Said and W.A. Pearlman. An Image Multiresolution Representation for Lossless and Lossy Compression. *IEEE Transactions on Image Processing*, 5(9):1303–1310, 1996.
- [107] J. Seward. *bzip2* 0.9.0b. <http://www.muraroa.demon.co.uk/>.
- [108] C.E. Shannon. Prediction and Entropy of Printed English. *Bell Systems Technical Journal*, 30:50–64, 1951.
- [109] C.E. Shannon and W. Weaver. *The Mathematical Theory of Communication*. University of Illinois Press, 1963.
- [110] J.M. Shapiro. Embedded Image Coding Using Zerotrees of Wavelet Coefficients. *IEEE Transactions on Signal Processing*, 41(12):3445–3462, 1993.
- [111] J.A. Storer and T.G. Szymanski. Data Compression via Textual Substitution. *Journal of the ACM*, 29(4):928–951, 1982.
- [112] S.W. Thomas, J. McKie, S. Davies, K. Turkowski, Woods J.A., and J. Orost. *compress* 4.1. <ftp://wuarchive.wustl.edu/packages/compression/compress-4.1.tar>.
- [113] T.D. Tran. *Linear Phase Perfect Reconstruction Filter Banks: Theory, Structure, Design and Application in Image Compression*. PhD thesis, University of Wisconsin-Madison, 1999.
- [114] G.A. Triantafyllidis and M.G. Strintzis. A Context Based Adaptive Arithmetic Coding Technique for Lossless Image Compression. *IEEE Signal Processing Letters*, 6(7):168–170, July 1999.
- [115] Kou-Hu Tzou. Progressive Image Transmission: A Review and Comparison of Techniques. *Optical Engineering*, 26(7):581–589, July 1987.
- [116] J.S. Vitter. Design and Analysis of Dynamic Huffman Codes. *ACM*, (4):825–845, 1987.
- [117] G.K. Wallace. The JPEG Still Picture Compression Standard. *Communications of the ACM*, 34(4):30–40, 1991.
- [118] M. Weinberger, G. Seroussi, and G. Sapiro. *JPEG-LS Reference Encoder - V.1.00*. Hewlett-Packard Laboratories, 1999.

- [119] M.J. Weinberger, J. Rissanen, and R.B. Arps. Applications of the Universal Context Modeling to Lossless Compression of Gray-Scale Images. *IEEE Transactions on Image Processing*, 5(4):575–586, 1996.
- [120] M.J. Weinberger, G. Seroussi, and G. Sapiro. LOCO-I: A Low Complexity, Context-Based, Lossless Image Compression Algorithm. In *Data Compression Conference (DCC)*, pages 140–149, 1996.
- [121] T.A. Welch. A Technique for High-Performance Data Compression. *IEEE Computer*, pages 8–19, 1984.
- [122] M.V. Wickerhauser. High-Resolution Still Picture Compression. Technical report, Department of Mathematics, Washington University, 1992.
- [123] R.N. Williams. Dynamic-History Predictive Compression. *Information Systems*, 13(1):129–140, 1988.
- [124] R.N. Williams. *Adaptive Data Compression*. Kluwer Academic Publishers, 1991.
- [125] W.J. Wilson. Chinks in the Armor of Public Key Cryptosystems. Technical Report 93/4, University of Waikato, 1994.
- [126] I.H. Witten and T.C. Bell. The Zero-Frequency Problem: Estimating the Probabilities of Novel Events in Adaptive Text Compression. *IEEE Transactions on Information Theory*, 37(4):1085–1094, 1991.
- [127] I.H. Witten, A. Moffat, and T.C. Bell. *MG (Managing Gigabytes) Information Retrieval System*. <http://www.cs.mu.oz.au/mg/>, 1994.
- [128] I.H. Witten, R.M. Neal, and J.G. Cleary. Arithmetic Coding for Data Compression. *Communications of the ACM*, 30(6):520–540, 1987.
- [129] J.W. Woods and S.D. O’Neil. Subband Coding of Images. *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-34(5):1278–1288, 1991.
- [130] X. Wu and N. Memon. Context-Based, Adaptive, Lossless Image Coding (CALIC). *IEEE Transactions on Communications*, 45(4):437–444, 1997.
- [131] X. Wu, N. Memon, and K. Sayood. A Context-based, Adaptive, Lossless/Nearly-Lossless Coding Scheme for Continuous-tone Images (CALIC). Technical report, University of Western Ontario, London, Ontario, Canada, 1995.
- [132] X. Wu, H. Wang, and J. Liu. *CALICH/CALICA*. Univerty of Western Ontario, ftp://ftp.csd.uwo.ca/pub/from_wu/ edition, 1995.
- [133] Z. Xiong, K. Ramchandran, and M.T. Orchard. Space-Frequency Quantization for Wavelet Image Coding. *IEEE Transactions on Image Processing*, 6(5):677–693, May 1997.

- [134] Y. Yoo, A. Ortega, and B. Yu. Image Subband Coding Using Context-Based Classification and Adaptive Quantization. *IEEE Transactions on Image Processing*, 8(12):1702–1715, December 1999.
- [135] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.
- [136] J. Ziv and A. Lempel. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.