Codificación y compresión

El concepto de entropía

Habitualmente se dice que la teoría de la información tiene su origen en un famoso artículo de C.E. Shannon de 1948 titulado "Una teoría matemática de la comunicación" [Sha48]. Entonces la informática no se había desarrollado (el famoso ENIAC entró en funcionamiento dos años antes y su "programación" se efectuaba conectando cables), sin embargo sí que tenía alguna importancia la transmisión de señales digitalizadas (el telégrafo muestra lo antigua que es la idea). Por ello la teoría introducida por Shannon es fundamentalmente digital y ha permanecido vigente a pesar de que el contexto de las comunicaciones ha cambiado totalmente. Es significativo de la influencia de [Sha48] que sea allí donde aparece por escrito por primera vez la palabra *bit* como abreviatura de *binary digit*, que Shannon atribuye a uno de sus colegas matemáticos pioneros de la informática. El éxito de [Sha48] posiblemente esté ligado en parte a su estilo centrado más en las ideas y en la exposición, que en el rigor matemático. Quizá por ello o por temas de prioridad, el famoso probabilista J. L. Doob incluyó en su reseña de [Sha48] la sorprendente frase *The discussion is suggestive throughout, rather than mathematical, and it is not always clear that the author's mathematical intentions are honorable.*

Partimos de un conjunto finito $S = \{s_1, s_2, \ldots, s_n\}$ que además es un espacio de probabilidad, con p_i la probabilidad de escoger s_i. La idea general es que los elementos de S sean cualesquiera fuentes de información (la notación S tomada de [Rom92] es de *source*) pero más adelante preferiremos considerar S como un conjunto de caracteres con los que componemos unidades mayores de información (que son un modelo para los ficheros digitales).

Shannon se preguntó en la primera parte de su artículo cómo medir la información contenida en S o dicho de otra forma más ilustrativa, la "incertidumbre" al extraer una muestra de S. Por ejemplo, si p_1 es casi 1 y el resto de los p_i son casi 0, apenas tenemos incertidumbre sobre qué elemento de S se va a elegir. El caso opuesto es que $p_1 = \cdots = p_n = 1/n$ en el que no sabemos apostar por un elemento particular (véase en [Rén84] más sobre esta idea). Shannon procedió de una manera teórica y exigió a la función medidora de la información $H = H(p_1, \ldots, p_n)$ que fuera continua, que $H(1/n, \overset{n \text{ veces}}{\ldots}, 1/n)$ fuera creciente en n (más oportunidades, más incertidumbre) y que cumpliera

$$H\big(\frac{1}{n}, \overset{n \text{ veces}}{\ldots}, \frac{1}{n}\big) = H\big(\frac{b_1}{n}, \frac{b_2}{n}, \ldots, \frac{b_k}{n}\big) + \sum_{i=1}^{k} \frac{b_i}{n} H\big(\frac{1}{b_i}, \overset{b_i \text{ veces}}{\ldots}, \frac{1}{b_i}\big), \qquad \forall b_i \in \mathbb{Z}^+ \quad \sum_{i=1}^{k} b_i = n. \tag{1}$$

Esta última propiedad simplemente significa que la cantidad de información no puede variar si subdividimos S en subconjuntos más pequeños de tamaño b_i [Rom92, §1.1] [Mac03, §2.5].

Shannon probó un pequeño resultado que asegura que las únicas funciones con estas propiedades son de la forma $H = K \sum_i p_i \log p_i$ con $K < 0$ y, como estaba interesado en el caso digital, tomó $K = -1/\log 2$. De esta forma, se define la *entropía* como

$$H(p_1, \ldots, p_n) = -\sum_{i=1}^{n} p_i \log_2 p_i. \tag{2}$$

Por convenio, diremos que $p \log_2 p$ vale 0 para $p = 0$, pues es su límite cuando $p \to 0^+$. El nombre y la notación para la entropía vienen heredados de los trabajos de L. Boltzmann en el siglo XIX sobre un concepto análogo en mecánica estadística y termodinámica.

Si $n = 2^N$ y $p_1 = \cdots = p_n = 1/N$, entonces la entropía es N, el número de bits que necesitamos para representar 2^N objetos. En general, si queremos representar un valor de k posibles necesitaremos aproximadamente

$-\log_2(1/k)$ bits y (2) es un promedio de cantidades similares. Esto sugiere, como ya hemos indicado, que la entropía está relacionada con el número de bits necesarios para almacenar la información.

Para concretar esta idea, pensemos en que estamos interesados en comprimir textos y que en S incluimos sólo los caracteres y signos de puntuación del idioma que usamos. En esta situación es fácil tener una idea estadística de los valores de p_i, el propio Shannon aproximó con 4.14 la entropía de las letras en inglés. Para codificar un texto, concatenamos las codificaciones en bits de las letras. Definimos entonces *código* (*binario únicamente descifrable*, si se quiere ser más preciso) como una función inyectiva $C : S \longrightarrow \{\text{cadenas de bits}\}$ que también es inyectiva cuando actúa sobre las cadenas de elementos de S

$$C \ : \ s_{i_1}s_{i_2}\dots s_{i_k} \longmapsto C(s_{i_1})C(s_{i_2})\dots C(s_{i_k}) \tag{3}$$

Por ejemplo, si partimos de $S = \{a, b, c\}$, entonces $C(a) = 0$, $C(b) = 1$, $C(c) = 10$ no es un código válido porque $C(ba) = C(c)$. Sin embargo sí lo es redefiniendo $C(b) = 11$.

En el diseño de un código hay dos objetivos prácticos opuestos:

1. Incorporar al código alguna información redundante de manera que tenga cierta robustez frente a errores debidos a su transmisión por un canal poco fiable.
2. Conseguir que el código no tenga ninguna información redundante para que el almacenamiento de la cadena de bits requiera el menor espacio posible.

Ambas situaciones son muy comunes y constituyen los temas principales de [Sha48]. Por ejemplo, con los parámetros actuales de costes y tecnología, no es posible grabar CDs o DVDs con total fiabilidad. Según parece, contienen del orden de cientos de miles de errores a los que hay que sumar los producidos por el uso (por ejemplo pequeñas ralladuras). Incluso si la reproducción fuera físicamente perfecta, estos dispositivos serían inservibles si no se emplease una codificación redundante con un sofisticado algoritmo de corrección de errores. Números tan prosaicos como el NIF, los de los códigos de barras o los de cuentas bancarias ya contienen redundancia en la información.

Por otro lado, la eliminación de información redundante es la clave para conseguir buenos algoritmos de compresión, que aquí consideramos sin pérdidas. Es decir, pensamos por ejemplo en programas en los que la pérdida de un bit puede ser catastrófica, más que en audio o vídeo donde se aceptan pequeñas pérdidas de calidad en aras de un almacenamiento más eficiente.

Las asignaturas de teoría de códigos de los grados de matemáticas se suelen centrar en la primera situación, mientras que aquí nos centraremos en la segunda. En este contexto, hay un teorema bonito, práctico y sencillo (y posiblemente poco conocido en el ámbito matemático) que limita los milagros que podrían ocurrir al comprimir. Este resultado se suele conocer como *source coding theorem* y en [Sha48] (centrado en la comunicación) se considera en relación con la transmisión por un canal sin ruido, por ello también se denomina *noiseless coding theorem*. Con menos frecuencia se le llama *primer teorema de Shannon*.

Dado un código C y $s_i \in S$, la cadena de bits $C(s_i)$ tendrá cierta longitud $\ell\big(C(s_i)\big)$. Por supuesto, para conseguir comprimir queremos que la longitud media de estas cadenas,

$$\ell(C) = \sum_i p_i \ell\big(C(s_i)\big), \tag{4}$$

sea pequeña. A partir de S podemos construir el conjunto $\mathcal{S}_N$ de cadenas de N elementos. Las probabilidades p_i de los elementos de S inducen una medida de probabilidad en $\mathcal{S}_N$ de manera natural suponiendo que los s_k que integran las cadenas se comportan como variables aleatorias independientes (esto no es cierto en muchas situaciones, por ejemplo en español detrás de una consonante es muy probable que vaya una vocal, volveremos sobre ello más adelante).

$$\mathcal{S}_N = \big\{s_{i_1}s_{i_2}\cdots s_{i_N} \ : \ s_{i_k} \in S\big\}, \qquad \text{Prob}\big(s_{i_1}s_{i_2}\cdots s_{i_N}\big) = p_{i_1}p_{i_2}\cdots p_{i_N}. \tag{5}$$

Para un código definido en $\mathcal{S}_N$, y no necesariamente definido en S, la longitud media es

$$\ell_N(C) = \sum_{i_1,\dots i_N} p_{i_1}\cdots p_{i_N}\ell\big(C(s_{i_1}\cdots s_{i_N})\big). \tag{6}$$

El resultado que hemos anunciado es el siguiente:

Teorema (*source coding theorem*). *Sea ℓ^* el mínimo de $\ell(C)$ sobre todos los posibles códigos definidos sobre S y H la entropía de S, entonces*

$$H \le \ell^* < H+1. \tag{7}$$

Por supuesto, comprimir o codificar carácter a carácter podría ser absurdo si acumulan poca información, por ejemplo si son bits. Con un truco sencillo se pasa al caso general de cadenas.

Corolario. *Sea ℓ_N^* el mínimo de $\ell_N(C)$ sobre todos los posibles códigos definidos en $\mathcal{S}_N$ y H la entropía de S, entonces*

$$H \le \frac{\ell_N^*}{N} < H + \frac{1}{N}, \qquad \textit{en particular } \lim_{N\to\infty}\frac{\ell_N^*}{N} = H. \tag{8}$$

En breve, lo que asegura este resultado es que en media un fichero con N caracteres se puede comprimir como mucho hasta tamaño HN y que, de hecho, hay una igualdad asintótica. Es decir, que la entropía responde a la idea original de una medida de la cantidad de información por cada elemento de S.

Un ejemplo tonto para comprobar el resultado es que si $S=\{0,1\}$ con probabilidades $p_1=p_2=1/2$ estamos considerando ficheros al azar y no debería haber ninguna manera de comprimirlos, lo cual es coherente con que $H=1$.

Pensemos ahora en ficheros binarios que típicamente tengan 75 % de ceros y un 25 % de unos. La entropía de S es $-\frac{3}{4}\log_2\frac{3}{4}-\frac{1}{4}\log_2\frac{1}{4}=0.811$. En estas condiciones, un fichero de 100 Megas con un código adecuado se comprimiría en el límite hasta 81.1 Megas. Si codificamos los bits de uno en uno, obviamente no hay milagros y tendremos como poco los 100 Megas originales, pero si los agrupamos en bloques de 16 bits (dos bytes), el corolario nos asegura que existe un código tal que, en media, el tamaño en Megas estaría entre $100\cdot 0.811$ y $100\cdot(0.811+1/16)\approx 87.4$.

Pensemos ahora en ficheros compuestos por bytes, en los que el byte con valor 0 aparece la mitad de las veces y bytes con valores entre 1 y 20 con probabilidad 0.4 (esto es un modelo cualitativo de lo que sucede en el formato JPEG tras la cuantificación). La entropía es máxima si suponemos equidistribución en ausencia de más datos, es decir, que cada byte entre 1 y 20 tiene probabilidad 0.4/20 y algo similar con los 235 números mayores de 20, entonces

$$-0.5\log_2 0.5 - 20\cdot\frac{0.4}{20}\log_2\frac{0.4}{20} - 235\cdot\frac{0.1}{235}\log_2\frac{0.1}{235} = 3.877. \tag{9}$$

Esto significa que, en esta situación, cada byte lleva menos de 4 bits de información. Un fichero de 100 KB típicamente se podría comprimir con una codificación adecuada a $100\cdot 1024\cdot 3.877 \approx 397000$ bits que son aproximadamente 48.5 KB. Es decir, a menos de la mitad.

Para deducir el Corolario del Teorema, tomamos $\mathcal{S}_N$ en vez de S, con $\mathcal{S}_N$ todas las posibles cadenas de N elementos de S, de esta forma, ℓ^* se transforma en ℓ_N^*. La entropía correspondiente a $\mathcal{S}_N$ es

$$-\sum_{i_1,i_2,\dots,i_N} p_{i_1}p_{i_2}\cdots p_{i_N}\log_2\left(p_{i_1}p_{i_2}\cdots p_{i_N}\right) = -\sum_{i_1,i_2,\dots,i_N} p_{i_1}p_{i_2}\cdots p_{i_N}\Big(\log_2 p_{i_1}+\cdots+\log_2 p_{i_N}\Big). \tag{10}$$

Al desarrollar esta suma, el coeficiente de $-\log_2 p_1$ será

$$\sum_{k=1}^{N} \sum_{\substack{i_1,i_2,\ldots,i_N \\ i_k=1}} p_{i_1} p_{i_2} \cdots p_{i_N} = N p_1 \sum_{j_1,\ldots,j_{N-1}} p_{j_1} \cdots p_{j_{N-1}} = N p_1 \Big(\sum_{i=1}^{n} p_i \Big)^{N-1} = N p_1. \tag{11}$$

En general, el coeficiente de $-\log_2 p_i$ es Np_i y la entropía de $\mathcal{S}_N$ es NH con H la entropía de S. Así pues al aplicar (7) a $\mathcal{S}_N$ se obtiene $NH \le \ell_N^* < NH + 1$, que es (8).

Como ejemplo práctico, un fichero al azar formado por 10^6 bytes, cada uno de los cuales representando (en ASCII) el carácter `A` o el carácter `B` con probabilidades $1/2$, se puede comprimir típicamente a $10^6/8 = 125000$ bytes simplemente cambiando `A` por el bit 0 y `B` por el bit 1, lo cual concuerda con que $H = 1$, ya que la longitud esperada en bits según (8) es 10^6. Los resultados de la compresión de un fichero de este tipo con *software* bien conocido, arrojó los siguientes datos:

`gzip`	$\to$ 159068 bytes	`zip`	$\to$ 159210 bytes
`bzip2`	$\to$ 160213 bytes	`rar`	$\to$ 161495 bytes

Aunque los resultados son parecidos, haciedo pruebas, parece que en ficheros de este tipo, `gzip` es ligeramente superior y `rar` ligeramente inferior. Cabe preguntarse por qué si en este caso es tan fácil obtener la cota inferior de (8), estos algoritmos dan un resultado alrededor de un 30 % peor que lo óptimo. La respuesta es que este tipo de ficheros no es el objetivo principal de los algoritmos asociados. Más bien, están destinados a comprimir textos legibles o, en general, ficheros con muchas repeticiones de bloques de más de un carácter.

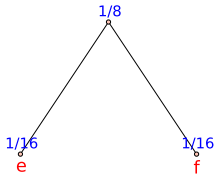
Codificación de Huffman

La prueba del *source coding theorem* es constructiva, es decir, en principio permitiría diseñar un código para el que ℓ_N^* satisfaga la desigualdad (8). En 1952, D.A. Huffman, siendo un estudiante de doctorado, creó un algoritmo que produce un código que minimiza $\ell(C)$. Lo más importante desde el punto de vista actual es que el algoritmo es sencillo y fácil de programar. En él se parte del conjunto $S_0 = S$ y se continúa hasta llegar a S_{n-1}, que sólo tiene un elemento, siendo el objetivo la creación de un árbol binario siguiendo las siguientes reglas:

1. Se toman los dos elementos de menor probabilidad de S_i y se añaden al árbol como hijos de un padre al que se le asigna la suma de las probabilidades.
2. Se construye S_{i+1} sustituyendo en S_i los hijos del paso anterior por su padre.

Dibujaremos el árbol de la manera habitual en teoría de grafos, con la raíz arriba y las hojas abajo, aunque esto es, por supuesto, irrelevante.

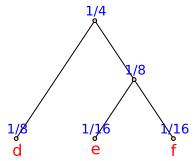
Como ejemplo, tomemos $S = \{a, b, c, d, e, f\}$ con probabilidades respectivas $p_1 = 1/2$, $p_2 = p_3 = p_4 = 1/8$, $p_5 = p_6 = 1/16$. Las probabilidades menores en $S_0 = S$ son las de e y f, por tanto el primer paso es



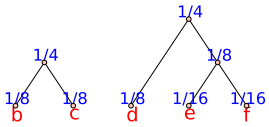
$$\begin{array}{lccccccc} S_1 = \{ & a, & b, & c, & d, & \widehat{ef} & \} \\ & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \\ \text{Prob} & \frac{1}{2} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \end{array}$$

donde $\widehat{ef}$ es el padre de e y f. En general, abreviaremos con $\widehat{xy}$ el elemento del árbol que cumple que x es su primer hijo o desciende de él mientras que y es su segundo hijo o descendiente de él.

Ahora hay varias posibles elecciones de dos elementos de probabilidad $1/8$, la menor. Escojamos por ejemplo d y $\widehat{ef}$. En la siguiente iteración no hay ambigüedad y necesariamente hay que elegir b y c.

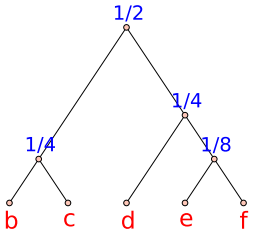


$$S_2 = \{ \quad a, \quad b, \quad c, \quad \widehat{df} \quad \}$$
$$\text{Prob} \quad \downarrow \frac{1}{2} \quad \downarrow \frac{1}{8} \quad \downarrow \frac{1}{8} \quad \downarrow \frac{1}{4}$$

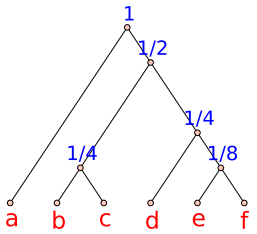


$$S_3 = \{ \quad a, \quad \widehat{bc}, \quad \widehat{df} \quad \}$$
$$\text{Prob} \quad \downarrow \frac{1}{2} \quad \downarrow \frac{1}{4} \quad \downarrow \frac{1}{4}$$

El siguiente paso será agrupar $\widehat{bc}$ y $\widehat{df}$. Finalmente, el padre de ambos se agrupará con a y el padre de ellos completará la raíz del árbol.

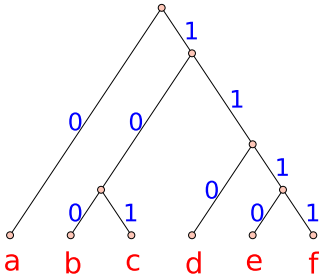


$$S_4 = \{ \quad a, \quad \widehat{bf} \quad \}$$
$$\text{Prob} \quad \downarrow \frac{1}{2} \quad \downarrow \frac{1}{2}$$



$$S_3 = \{ \quad \widehat{af} \quad \}$$
$$\text{Prob} \quad \downarrow 1$$

Una vez construido el árbol, representaremos el camino desde la raíz a un elemento $s \in S$ en una de las hojas como una sucesión de ceros y unos, convencionalmente diremos que cero será el camino de la izquierda y uno el de la derecha. La cadena resultante es la que asignamos como codificación a s.



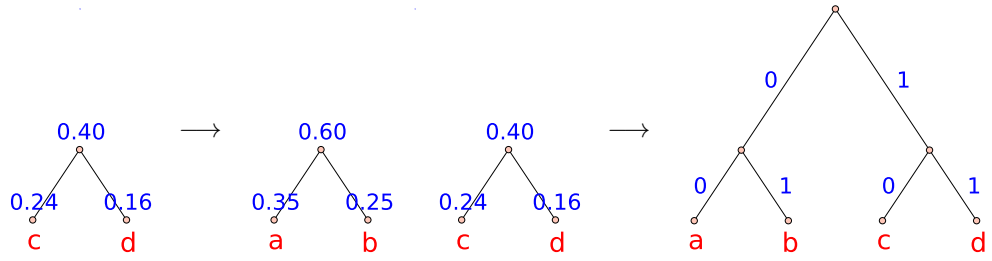
$$a \to 0 \qquad d \to 110$$
$$b \to 100 \qquad e \to 1110$$
$$c \to 101 \qquad f \to 1111$$

$$\ell(C) = 1 \cdot \tfrac{1}{2} + 3 \cdot \tfrac{1}{8} + 3 \cdot \tfrac{1}{8} + 3 \cdot \tfrac{1}{8} + 4 \cdot \tfrac{1}{16} + 4 \cdot \tfrac{1}{16} = \tfrac{17}{8}$$

$$H = -\tfrac{1}{2}\log_2 \tfrac{1}{2} - 3\tfrac{1}{8}\log_2 \tfrac{1}{8} - 2\tfrac{1}{16}\log_2 \tfrac{1}{16} = \tfrac{17}{8}$$

La longitud media es $\ell(C) = 17/8$ y coincide con la entropía; por tanto, según (7), es óptimo en cuanto a la longitud media.

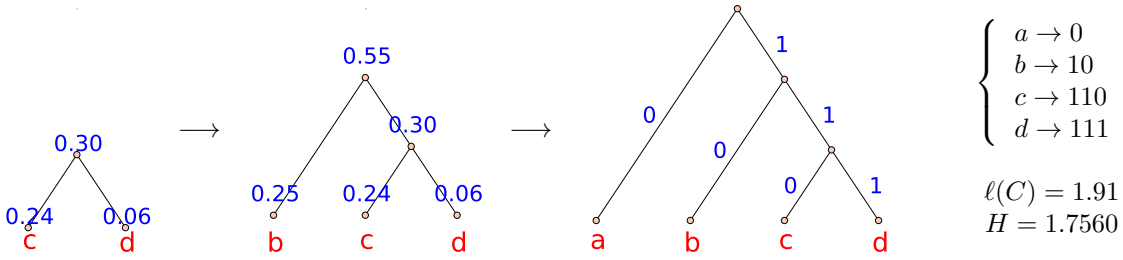
Consideremos $S = \{a, b, c, d\}$. Si las probabilidades p_i no están suficientemente descompensadas, entonces lo mejor es lo obvio: usar las cuatro posibles cadenas de 2 bits. Por ejemplo, con $p_1 = 0.35$, $p_2 = 0.25$, $p_3 = 0.24$, $p_4 = 0.16$



$$\begin{cases} a \to 00 \\ b \to 01 \\ c \to 10 \\ d \to 11 \end{cases}$$

$$\ell(C) = 2$$
$$H = 1.9472$$

Si reducimos p_4 a 0.06 y tomamos $p_1 = 0.45$, entonces ya es más ventajoso reducir la codificación de a a un bit a costa de hacer crecer las codificaciones de c y d a tres bits.



La pregunta natural es cómo estamos seguros de que este algoritmo lleva a un código, es decir, que no hay cadenas distintas de elementos de S que den lugar a codificaciones iguales. La respuesta es más sencilla de lo que parece a primera vista y merece la pena ser generalizada.

Supongamos que tenemos una función $C : S \longrightarrow \{\text{cadenas de bits}\}$ tal que para cualquier $b_1 b_2 \cdots b_N \in \operatorname{Im} C$, $b_i \in \{0, 1\}$ se tiene que la cadena $b_1 b_2 \dots b_k$ no está en $\operatorname{Im} C$ para $1 \leq k < N$. Las funciones de este tipo se llaman *códigos prefijo* porque ningún $C(s_i)$ es el "prefijo" de otro (quizá el ejemplo más empleado en la práctica sea la codificación UTF-8). Es fácil ver que cualquiera de estas funciones es un código, porque una coincidencia de las imágenes de $s_{i_1} s_{i_2} \dots s_{i_k}$ y $s_{j_1} s_{j_2} \dots s_{j_l}$ implica que $C(s_{i_1})$ y $C(s_{j_1})$ violan la condición anterior si $s_{i_1} \neq s_{j_1}$, y si fueran iguales, basta considerar los siguientes.

Por la regla de asignar el 0 a la izquierda y el 1 a la derecha, la imagen de un código prefijo queda representado biyectivamente por un árbol binario (no completo en general, es decir, un padre puede tener un solo hijo) en el que las hojas corresponderán a los elementos que se codifican, porque si alguno estuviera en un nodo interior la condición de prefijo no se cumpliría. En particular, la codificación de Huffman es un código prefijo y por tanto un código válido.

La segunda cuestión es por qué la codificación de Huffman minimiza $\ell(C)$. Vamos a ver que lo hace entre el subconjunto de códigos prefijo. Como mencionaremos más adelante, en realidad siempre un código se puede transformar en código prefijo sin cambiar las longitudes, por tanto también minimiza entre todos los códigos.

Teorema. *El mínimo de $\ell(C)$ sobre todos los códigos prefijos se alcanza con la codificación de Huffman.*

Demostración. Digamos que $S = \{s_1, s_2, \dots, s_n\}$ con $p_1 \leq p_2 \leq \cdots \leq p_n > 0$. Si $n > 1$, el árbol binario que corresponde al código prefijo óptimo lo podemos suponer completo porque si un padre no tuviera dos hijos, ese vértice se podría suprimir del árbol, reduciendo las longitudes de las codificaciones. La longitud de $C(s_1)$ debe ser máxima porque si $C(s_i)$ tuviera longitud mayor, $p_i \ell(C(s_i)) + p_1 \ell(C(s_1)) > p_i \ell(C(s_1)) + p_1 \ell(C(s_i))$ e intercambiar las codificaciones de s_1 y s_i sería ventajoso. De la misma forma, podemos cambiar las codificaciones de s_2 y la del hermano de s_1 en el árbol, porque $\ell(C(s_2))$ es máximo entre los $\ell(C(s_i))$, $i > 1$. Por tanto, podemos suponer que s_1 y s_2 corresponden a hojas hermanas en el árbol.

Ahora procedemos por inducción en n. Si $n = 2$ no hay nada que probar porque evidentemente la longitud media mínima es 1. Supongamos el resultado probado para $n-1$ y sea $S' = \{z, s_3, s_4, \dots, s_n\}$ donde los s_i tienen las probabilidades de antes y z tiene probabilidad $p_1 + p_2$. Un código óptimo como antes en S induce un código en S' (quizá no óptimo) que en el árbol corresponde a sustituir las hojas de s_1 y s_2 por su padre z. Sean H y H' las codificaciones de Huffman en S y S', entonces

$$\ell(C) = p_1 + p_2 + \ell(C') \qquad \text{y} \qquad \ell(H) = p_1 + p_2 + \ell(H'). \tag{12}$$

La segunda igualdad es por la construcción del algoritmo de Huffman y la primera se sigue porque $p_1 \ell(C(s_1)) + p_2 \ell(C(s_2)) = (p_1 + p_2)\ell(C(s_1))$ se ha reemplazado por $(p_1 + p_2)\ell(C(z)) = (p_1 + p_2)\big(\ell(C(s_1)) - 1\big)$.

Por la hipótesis de inducción, $\ell(H') \leq \ell(C')$ y de (12) se deduce que $\ell(H) \leq \ell(C)$. □

La prueba del primer teorema de Shannon

Un punto fundamental en la deducción actual de (7) es la desigualdad de Kraft, introducida por L. Kraft en su tesis de máster de 1949 (un año después de [Sha48], a veces se ha dicho que Shannon no demostró sus

teoremas) y probada independientemente por B. McMillan [McM56]. El siguiente enunciado incluye la propia desigualdad y su recíproco.

Teorema (*Desigualdad de Kraft*). *Dado un código prefijo C sobre $S = \{s_1, \ldots, s_n\}$, las longitudes de las codificaciones $l_i = \ell(C(s_i))$ satisfacen*

$$\sum_{i=1}^{n} 2^{-l_i} \leq 1. \tag{13}$$

Además, dados enteros positivos l_i cualesquiera que verifiquen (13)*, siempre existe un código prefijo de forma que esos enteros son exactamente las longitudes de las codificaciones de los elementos de S.*

La generalización de McMillan, aparte de añadir el recíproco, consiste en suprimir la hipótesis de que sea un código prefijo. De ello se deduce que dado un código C, existe siempre un código prefijo C_p tal que $\ell(C(s_i)) = \ell(C_p(s_i))$ y por ello, en cualquier resultado que hable de estas longitudes podemos suponer que el código es un código prefijo. De todas formas, los códigos prefijos son los más prácticos, porque coinciden con los *instantáneos*, los que pueden descodificarse al tiempo que se reciben [Rom92], sin esperar al final de la transmisión.

Demostración (del primer teorema de Shannon). Utilizando, por ejemplo, multiplicadores de Lagrange (o métodos más elementales), se tiene que la función $f(x_1, \ldots, x_n) = -\sum p_i \log_2 x_i$ alcanza un mínimo en el *simplex* $\sum x_i = 1$, $0 \leq x_i \leq 1$ cuando $x_i = p_i$.

Con la notación de la desigualdad de Kraft, tomemos $x_i = 2^{-l_i} / \sum 2^{-l_j}$, entonces la cota inferior de (7) se sigue de

$$H = f(p_1, p_2, \ldots, p_n) \leq f(x_1, x_2, \ldots, x_n) \leq f\big(2^{-l_1}, 2^{-l_2}, \ldots, 2^{-l_n}\big) = \ell(C), \tag{14}$$

donde se ha aplicado (13) para la segunda desigualdad.

Para obtener la cota superior de (7), elijamos $l_i \in \mathbb{Z}^+$ tales que $-\log_2 p_i \leq l_i < 1 - \log_2 p_i$ (suponemos $p_i \neq 0$ sin pérdida de generalidad). Por el recíproco de la desigualdad de Kraft, debe existir un código prefijo con $l_i = \ell(C(s_i))$, ya que $\sum 2^{-l_i} \leq \sum p_i = 1$. Entonces $\ell(C) < \sum p_i(1 - \log_2 p_i) = 1 + H$ y se obtiene el resultado deseado. □

Demostración (de la desigualdad de Kraft). Digamos que $l_1 \leq l_2 \leq \cdots \leq l_r < l_{r+1} = \cdots = l_n$. Consideremos el conjunto B de todas las cadenas de l_n bits. Si $i \leq r$, el conjunto $B \cap \operatorname{Im} C$ no contiene a ninguna de las $2^{l_n - l_i}$ cadenas que comienzan como $C(s_i)$, mientras que $C(s_i) \in B$ para $i > r$. Por tanto

$$|B| = |B \cap \operatorname{Im} C| + |B - B \cap \operatorname{Im} C| \geq n - r + \sum_{i=1}^{r} 2^{l_n - l_i} = \sum_{i=1}^{n} 2^{l_n - l_i} \tag{15}$$

y basta notar que $|B| = 2^{l_n}$. Otra forma de proceder más simple es asociar a B el árbol binario completo. A las $2^{l_n - l_j}$ cadenas de B que empiezan por $C(s_j)$ les corresponden las hojas de un subárbol binario completo. La condición de prefijo asegura que estos subárboles son disjuntos y se deduce la desigualdad.

Para el recíproco consideramos el árbol binario completo T de altura (*profundidad*) l_n, el cual tiene 2^{l_n} hojas. Establecemos el siguiente algoritmo: Para cada $i \leq r$ consideramos un vértice que diste l_i de la raíz (que tenga profundidad l_i) y eliminamos todos sus descendientes, entre ellos $2^{l_n - l_i}$ hojas de T. Si queremos que este algoritmo pueda llevarse a cabo y que al final queden al menos $n - r$ hojas de T, necesitamos $2^{l_n} - \sum_{i=1}^{r} 2^{l_n - l_i} \geq n - r$, lo cual equivale a la hipótesis (13). Tras el algoritmo, si quedasen más de $n - r$ hojas de T, eliminamos las sobrantes. El resultado será un árbol T' cuyas hojas tienen profundidades exactamente los l_i. Así pues, T' corresponde a un código prefijo con las propiedades esperadas. □

Métodos de diccionario

Comencemos con un ejemplo práctico: La estimación de la entropía de los caracteres empleados en un texto en español es de 4.09. Según (8) debemos esperar que N caracteres en español no se compriman más allá de $4.09N/8$ bytes. Experimentando con un texto de *La regenta* de 1869610 caracteres, los resultados fueron:

`gzip`	$\to$ 730122 bytes	`zip`	$\to$ 730264 bytes
`bzip2`	$\to$ 534165 bytes	`rar`	$\to$ 609319 bytes

Sin embargo la estimación teórica anterior sugiere que alrededor de 955000 bytes es un límite insuperable. Otros ejemplos de textos muestran también razones de compresión muy por debajo del $4.09/8 \approx 1/2$ que predice la teoría.

Imaginemos un tipo de ficheros en que cada byte tiene un 99 % de posibilidades de ser igual al anterior y en el 1 % restante cambia aleatoriamente. Si el fichero es suficientemente grande, la probabilidad de aparición de cada posible byte será $1/256$ y por tanto $H = 8$, que indica que cada byte tiene 8 bits. Si aplicamos (8), se deduce que no hay compresión posible. Sin embargo está claro que hay un método de compresión muy sencillo, llamado RLE (*run-length encoding*): Por cada cadena de bytes iguales usamos un byte para indicar el número de repeticiones y otro para indicar el byte que se repite. En caso de que haya más de 255 repeticiones (lo cual es poco probable) se puede dividir la cadena en dos partes. Por ejemplo, `200A100A100B` significa 300 letras `A` seguidas de 100 letras `B`. Como típicamente cada cadena de bytes iguales se repite 100 veces, la compresión es del orden de 1/50.

La paradoja proviene de que en (8) las probabilidades se asignan a través de (5) suponiendo la independencia de los caracteres, lo cual no es el caso aquí.

La codificación de Huffman tomando como S todas las posibles cadenas de 100 bytes que aparecen en el fichero, en principio es muy ventajosa aquí pero ese dato viene de un conocimiento previo del tipo de fichero. Por otro lado, S puede llegar a ser bastante grande y el árbol correspondiente complicado. Esto muestra dos de las dificultades para aplicar la codificación de Huffman como compresor de propósito general:

1. Depende de la división en símbolos de S que requiere una estadística previa costosa.
2. El árbol debe ser conocido por el descompresor y almacenarlo puede ser poco eficiente.

A partir de los años 70 del siglo XX, se diseñaron algunos métodos, a veces conocidos genéricamente como Lempel-Ziv, por los autores del trabajo original [ZL77], que tenían en mente especialmente textos (los ficheros multimedia no estaban apenas desarrollados entonces) pero con la aspiración de ser algoritmos universales, como dice el título de [ZL77] y justifican sus resultados teóricos. Los tres más conocidos son LZ77, LZ78 y LZW.

Imaginemos que uno tiene un texto en español y carga un "diccionario", una lista con todas las palabras posibles y ahora sustituye cada palabra por una referencia a esa lista. Pensando en que hay unas 80000 palabras pero muchas menos en uso, con dos bytes podremos especificar cada referencia, mientras que la longitud media de una palabra es ligeramente mayor que 4, por tanto la compresión será al menos a la mitad. Es fácil refinar el método y aumentar mucho la compresión, codificando (como en Huffman) las palabras más usadas con menos bytes. El problema de este método de compresión es que es demasiado rígido, no es aplicable si el texto está en otro idioma o es un fichero con datos numéricos.

La solución obvia es intentar crear el diccionario en función del fichero, la desventaja es que hay que incluir el diccionario en el fichero comprimir para que lo conozca el descompresor. Los algoritmos antes mencionados, siguen parcialmente esta idea y por ello se dice que son *métodos de diccionario*. La novedad es que en ellos el diccionario se crea de manera dinámica, al tiempo que se codifica o descodifica. Sólo existe temporalmente mientras se aplican los algoritmos correspondientes. Es decir, el diccionario es más virtual que real, no requiere ningún espacio adicional en la codificación, sino que se deduce de ella.

Los tres algoritmos son parecidos pero quizá a través de LZ78 sea más sencillo entender la idea. Lo veremos con un poco más de detalle e indicaremos en que consisten las variantes para los otros.

El algoritmo LZ78. Para simplificar, suponemos que al final de la cadena que queremos comprimir aparece un carácter de *fin de fichero* (EOF), que representamos con # y que no está en otro lugar.

Lo primero es descomponer la cadena en *frases*, donde una frase es una subcadena que añade un carácter nuevo a otra que haya aparecido antes. Por ejemplo, las divisiones en frases de `salsa␣salada` (donde ␣ indica un espacio) y de `rellena␣la␣encuesta`, serían

`|s|a|l|sa|␣|sal|ad|a#|` y `|r|e|l|le|n|a|␣|la|␣e|nc|u|es|t|a#|`

Por supuesto cada frase aparece sólo una vez. La división en frases se corresponde a un diccionario (una lista), cuya entrada k es la k-ésima frase y por convenio diremos que la entrada 0 es la cadena vacía. En los ejemplos anteriores los diccionarios serían

0	∅	5	␣
1	s	6	sal
2	a	7	ad
3	l	8	a#
4	sa		

y

0	∅	5	n	10	nc
1	r	6	a	11	u
2	e	7	␣	12	es
3	l	8	la	13	t
4	le	9	␣e	14	a#

El propósito del carácter de fin de fichero es asegurarnos de que la última frase no se quede sin almacenar en el diccionario.

El texto original se codifica asignando a cada frase un par, dado por su lugar en el diccionario al quitarle el último carácter y por ese último carácter. En los ejemplos anteriores se tendrían las codificaciones

(0, s), (0, a), (0, l), (1, a), (0, ␣), (4, l), (2, d), (2, #)

y

(0, r), (0, e), (0, l), (3, e), (0, n), (0, a), (0, ␣), (3, a), (7, e), (5, c), (0, u), (2, s), (0, t), (6, #)

Es fácil ver que a partir de estas codificaciones se pueden recuperar los diccionarios y con ellos, descodificar. Por ejemplo, en el primer caso, el descodificador lee (0, s) y sabe que tiene que poner una s en la posición 1, de la misma forma, las posiciones 2 y 3 pasarán a estar ocupados por a y l. A continuación se lee (1, a), y esto significa que hay que poner en la siguiente posición, la 4, lo que hay en la posición 1 y añadirle una a. En general, cuando se lee un par (k, x), se añade al final del diccionario lo que hay en la posición k terminado en x. El texto original es simplemente el diccionario leído de principio a fin.

El primer texto en ASCII ocuparía 13 bytes (13 caracteres) y si en la codificación reservamos un byte para cada número (esto no es posible para diccionarios de más de 256 elementos), entonces la "compresión" ocupa 16 bytes. Esto es más que el original. El segundo ejemplo es todavía peor, originalmente tiene 20 bytes y su codificación tiene 28 bytes. Esta aparente ineficiencia del algoritmo es una falsa impresión que se debe a que el texto es muy breve. Cuando es suficientemente grande, el número de coincidencias con elementos del diccionario aumenta rápidamente. Por ejemplo, los 1869610 caracteres del texto que habíamos manejado antes, dan lugar a un diccionario de tamaño 281192. Es decir, la codificación requiere esta cantidad de pares. El número que aparece en cada par se puede representar con 19 bits, porque $2^{19} > 281192$ y el carácter con 8, en total 27 bits por cada par, lo que hacen 949023 bytes, lo cual rebaja la cota de entropía de los caracteres pero está por encima de lo que consiguen los compresores habituales.

En la práctica, tener que manejar diccionarios grandes en los procesos de codificación y descodificación puede ser ineficiente sin métodos adecuados para gestionarlos [Sal02].

Universalidad del algoritmo LZ78. Por supuesto, una cuestión fundamental es cuánto comprime "típicamente" un algoritmo. La respuesta honesta para cualquier compresor es nada, porque no hay compresión posible de ficheros genéricos cuyos bits toman valores 0 y 1 aleatoriamente con probabilidad 1/2. La clave es que muchos de los ficheros que son habituales para nosotros tienen entropía baja y son bien diferentes de los típicos en un sentido probabilista natural.

La fuerza de la compresión con LZ78 está ligada al número de frases F que se guardan en el diccionario. Cada par de la codificación emplea un carácter y una referencia a un frase del diccionario que requerirá $\lceil \log_2 F \rceil$ bits (donde $\lceil x \rceil$ es el menor entero $n \geq x$). Como hay F pares, la longitud en bits tras aplicar el algoritmo a una cadena s será

$$\ell_{LZ}(s) = F\big(\lceil \log_2 F \rceil + b\big) \tag{16}$$

donde b es el número de bits necesarios para codificar los caracteres que componen s.

En el texto que antes se ha puesto de ejemplo, sustituyendo las 281192 frases que hay en el diccionario y $b = 8$ (caracteres ASCII) se llega al tamaño de 949023 bytes antes mencionados. Si combinamos el procedimiento con Huffman en la parte del par que ocupa el carácter, podremos reemplazar posiblemente el 8 de la fórmula por un 4 (o a lo más un 5), ya que ésta es aproximadamente la entropía de los caracteres en español (a la que se acercaría la codificación de Huffman), lo cual llevaría a 776834 bytes, que se acerca a lo que hacen los compresores reales. Claramente también la parte de los números se presta a más compresión, pues los que correspondan a frases comunes aparecerán muchas veces.

Para comparar LZ78 con el mejor código involucrado en (8), consideramos la longitud media al codificar elementos de $\mathcal{S}_N$, las cadenas de longitud N, con la probabilidad indicada en (5), es decir,

$$\ell^*_{LZ}(N) = \sum_{i_1,\dots i_N} p_{i_1} p_{i_2} \cdots p_{i_N} \ell_{LZ}\big(s_{i_1} s_{i_2} \cdots s_{i_N}\big). \tag{17}$$

Lo que deseamos es que esta longitud media sea pequeña.

Teorema. *Con la notación anterior se tiene*

$$\lim_{N\to\infty} \frac{\ell^*_{LZ}(N)}{N} = H \tag{18}$$

donde H es la entropía de S.

Esto prueba que LZ78 es *universal*, es decir, que asintóticamente comprime tanto como puede comprimir, sea cuales sean las probabilidades en S con una entropía fijada. De nuevo, es importante recordar que la definición de $\ell^*_{LZ}(N)$ implícitamente supone el modelo de variables aleatorias independientes de (5). este modelo puede ser muy poco realista en aplicaciones concretas (por ejemplo en ficheros de texto).

Demostración. Fijamos una cadena $s = s_{i_1} s_{i_2} \cdots s_{i_N}$ y la descomponemos en frases. Si $\mathcal{F}_i$ es el conjunto de frases de s de longitud i y f_i es su cardinal, entonces

$$p_{i_1} p_{i_2} \cdots p_{i_N} = \operatorname{Prob}(s) = \prod_i \prod_{f\in\mathcal{F}_i} \operatorname{Prob}(f) \leq \prod_i \Big(\frac{1}{f_i} \sum_{f\in\mathcal{F}_i} \operatorname{Prob}(f)\Big)^{f_i} \leq \prod_i f_i^{-f_i}, \tag{19}$$

donde se ha usado la desigualdad entre las medias geométrica y aritmética, y que la probabilidad es menor que 1.

Con la ayuda de los multiplicadores de Lagrange, se puede probar que para $x_i \in \mathbb{R}^+$ cumpliendo $\sum x_i = 1$ y $\sum i x_i = \mu$, se tiene

$$\sum_i x_i \log_2 x_i \geq -\log_2(\mu + 1) - 2. \tag{20}$$

Tomemos $x_i = f_i/F$ donde F es, como antes, el número total de frases. Entonces $\mu = N/F$, porque $\sum i f_i$ es la suma de las longitudes de las frases de s. Con ello

$$\sum_i f_i \log_2 f_i = F \log_2 F + F \sum_i x_i \log_2 x_i \geq F \log_2 F - F \log_2(N/F + 1) - 2F. \tag{21}$$

Sustituyendo en (19) después de tomar logaritmos y recordando la definición (16),

$$\log_2\big(\mathrm{Prob}(s)\big) \le -\ell_{LZ}(s) + F\log_2(N/F+1) + (b+3)F. \tag{22}$$

Los dos últimos términos al dividirlos por un N grande serán tan pequeños como deseemos, ya que $F/N = \sum f_i / \sum i f_i \to 0$. Entonces

$$-N^{-1}\mathrm{Prob}(s)\log_2\big(\mathrm{Prob}(s)\big) \ge N^{-1}\mathrm{Prob}(s)\ell_{LZ}(s) - \epsilon\mathrm{Prob}(s) \tag{23}$$

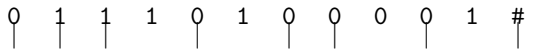
con ϵ arbitrariamente pequeño para N grande. Al sumar sobre $s \in \mathcal{S}_N$, el primer término es la entropía H (recuérdese que la entropía de $\mathcal{S}_N$ es N veces la de S) y el segundo es $N^{-1}\ell^*_{LZ}(N) - \epsilon$. De ello y de la cota inferior en (8), se sigue el resultado. □

El algoritmo LZW. Es una variante de LZ78 que carga un diccionario inicial con todos los posibles caracteres individuales (bytes, elementos de S). Tiene como ventaja que permite eliminar el segundo elemento de cada par de LZ78 a cambio de una codificación un poco más compleja y e alguna excepción en la descodificación. La división en frases es similar a la de LZ78 pero con dos particularidades:

1. Los caracteres individuales están precargados en el diccionario y por tanto no dan lugar a frases.
2. Cada carácter del final de una frase se incluye en el principio de la siguiente.

Todas las frases se almacenan en el diccionario ordenadamente (como en LZ78), tras las entradas iniciales, y la codificación es el lugar que ocupa en el diccionario cada frase sin su último carácter. Suponemos, como antes, que al final de los datos hay un carácter especial para que la última frase se incorpore al diccionario. En este caso, este carácter no aparecerá en la descodificación.

Para simplificar, veamos un ejemplo bit a bit (lo cual no es realista). De esta forma el diccionario inicial contiene simplemente 0 y 1. Tomemos `01110100001#` como la cadena a codificar. La división en frases y el diccionario correspondiente serían:



0	0
1	1
2	01
3	11
4	110
5	010
6	00
7	000
8	01#

Se han colocado las barras divisorias de frases debajo para resaltar que cada carácter sobre una frase está tanto en la frase de la izquierda como en la de la derecha. Para codificar, basta poner la referencia de todas las frases sin su último carácter. Esto es lo mismo que si en la división en frases (sin `#`) los caracteres fronterizos pasasen a la derecha y buscásemos los resultados en el diccionario

$$\big|\ 0\ \big|\ 1\ \big|\ 11\ \big|\ 01\ \big|\ 0\ \big|\ 00\ \big|\ 01\ \big| \qquad \longrightarrow \qquad 0,\ 1,\ 3,\ 2,\ 0,\ 6,\ 2.$$

La descodificación merece unos comentarios adicionales. En principio se aplica el mismo esquema que en LZ78: con el número en curso de la codificación y el anterior, se crea un nuevo elemento del diccionario a partir de los anteriores, y una vez recuperado el diccionario completo basta leer los elementos de la parte no precargada, omitiendo los últimos caracteres de cada uno de ellos para que no se repitan. El problema es que a veces uno de los códigos se refiere a un elemento del diccionario que no ha sido creado. Eso sólo puede ocurrir si la descodificación buscada coincide con la anterior añadiendo su primer carácter al final.

En el ejemplo anterior, el descodificador partiría del diccionario inicial $0 \to$ `0`, $1 \to$ `1` y al leer 0 y 1 en la codificación generaría `01` y ampliaría el diccionario con $2 \to$ `01`. Después leería 3, esta parte del diccionario no

está todavía creada. Como la última descodificación fue `1`, el 3 descodifica como `11` (primer carácter al final) y se amplía el diccionario con $3 \to$ `11`. La descodificación entonces sería por ahora `0111`. Después se leería 2 que corresponde a `01` y se añadiría al diccionario la nueva frase $4 \to$ `110`. Siempre cada frase añadida al diccionario es la última descodificación más el primer carácter de la que está en curso. El esquema completo para el ejemplo es:

Código	0	1	3*	2	0	6*	2
Descodificación	`0`	`1`	`11`	`01`	`0`	`00`	`01`
Diccionario		2→`01`	3→`11`	4→`110`	5→`010`	6→`00`	7→`000`

Los casos que llevan * son los que se refieren a elementos del diccionario que no han sido creados

Quizá es más sencillo entender el proceso con letras y cuando no hay que aplicar la regla "especial" (realmente en los ficheros que se comprimen bien, la situación especial es común). Digamos que partimos de un diccionario inicial $0 \to$ `␣`, $1 \to$ `h`, $2 \to$ `i`, $3 \to$ `o`, $4 \to$ `p`, y queremos descodificar 1, 2, 4, 0, 5, 7, 9, 3. El esquema sería:

Código	1	2	4	0	5	7	9	3
Descodificación	`h`	`i`	`p`	`␣`	`hi`	`p␣`	`hip`	`o`
Diccionario		5→`hi`	6→`ip`	7→`p␣`	8→`␣h`	9→`hip`	10→`p␣h`	11→`hipo`

Es decir, `hip␣hip␣hipo#`.

El algoritmo LZ77. Éste es el algoritmo original que dio lugar al resto. Más que en diccionarios se basa en el concepto de *ventana móvil* (*sliding window*) que esencialmente restringe la búsqueda de coincidencias a una porción de los datos de longitud predeterminada. La ventaja es que el número de bits para indicar referencias se conoce de antemano y no puede crecer. Por otro lado, la codificación emplea ternas, en lugar de los pares de LZ78 o los números sueltos de LZW.

No entraremos en detalles (que pueden consultarse en el original [ZL77], aunque es un poco teórico, o en [Sal02]), simplemente ilustraremos la situación con un ejemplo. Codifiquemos `salsa␣salada` con LZ77 usando una ventana móvil de tamaño 10. Esta ventana tiene dos partes, una corresponde al "pasado" y otra al "futuro". Digamos que sus tamaños son respectivamente 6 y 4 (según [Sal02], en la práctica la primera es de miles de bytes y la segunda de decenas de bytes).

En cada paso se consideran las cadenas que parten del primer carácter futuro y se busca una coincidencia lo más larga posible con cadenas que empiezan en el pasado. La codificación se expresa con una terna $(a, b, \mathtt{c})$ donde a dice la distancia a la que está la coincidencia, b su longitud y `c` es el primer carácter futuro en el que no hay coincidencia, hacia el que se moverá la ventana de futuro en el siguiente paso.

Inicialmente no hay nada con que comparar y entonces las primeras codificaciones son triviales con $a = b = 0$:

						s	a	l	s	$\to (0, 0, \mathtt{s})$
					s	a	l	s	a	$\to (0, 0, \mathtt{a})$
				s	a	l	s	a	␣	$\to (0, 0, \mathtt{l})$

Una vez que la segunda `s` pasa a ser el carácter futuro inicial, se produce una coincidencia de longitud 2, la cadena `sa`, y hay que ir 3 cuadros hacia el pasado para encontrarla:

			s	a	l	s	a	␣	s	$\to (3, 2, \mathtt{␣})$

El resto de la codificación, hasta llegar al fin de los datos sería:

	s	a	l	s	a	␣	s	a	l	$\to (0, 0, \mathtt{s})$
s	a	l	s	a	␣	s	a	l	a	$\to (6, 3, \mathtt{a})$
a	␣	s	a	l	a	d	a			$\to (0, 0, \mathtt{a})$
␣	s	a	l	a	d	a				$\to (4, 1, \#)$

De nuevo, en este ejemplo, la codificación aumenta mucho lo que ocuparían habitualmente los datos iniciales. Esta impresión viene de que es un texto muy breve.

Una situación no ilustrada en el ejemplo anterior, es que no se prohíbe que una coincidencia se extienda hasta los caracteres futuros. Por ejemplo $\mathtt{0\ldots01}\|\mathtt{110\ldots}$ se codificaría con $(1, 3, \mathtt{0})$.

Compresión y archivos de imagen. Ya vimos que en el formato JPEG había una compresión con pérdidas debida al tratamiento de los coeficientes de Fourier. Pero el proceso necesita algún método de compresión sin pérdidas para disminuir el tamaño que ocupan los coeficientes de Fourier cuantificados, que son muchas veces nulos o números pequeños. Aunque el método no está especificado en el formato, lo habitual es usar la codificación Huffman después de llevar a cabo una ordenación previa que tiende a agrupar los ceros y aplicar RLE. Para completar esta información, damos unas indicaciones acerca de otros de los formatos más empleados en el almacenamiento digital de imágenes:

BMP: No lleva compresión o RLE.

PNG: LZ77 y Huffman después de un filtrado de los datos para que la compresión se produzca en mejores condiciones.

GIF: Se fija una paleta de a lo más 256 colores para que cada pixel sólo requiera un byte y después se aplica LZW.

TIFF: No lleva compresión o RLE o RLE combinado con Huffman.

Es curioso que el formato PNG nació no tanto por una evolución natural, sino por una polémica por la posible aplicación en los años 90 de una patente sobre LZW (ya expirada en 2004). Las especificaciones del formato GIF requieren que sea éste su método de compresión y la empresa propietaria de la patente, anunció que el uso comercial de GIF estaría sujeto al pago de licencias. Aunque nunca se puso en práctica esta medida (por las protestas generalizadas), PNG surgió para escapar de su aplicación. En términos generales, PNG es mucho mejor que GIF pero inicialmente no tuvo mucha acogida, sobre todo porque los navegadores principales no lo reconocían, e incluso hoy en día no ha sustituido a GIF.

Referencias

[Mac03] D. J. C. MacKay. *Information theory, inference and learning algorithms*. Cambridge University Press, New York, 2003.

[McM56] B. McMillan. Two inequalities implied by unique decipherability. *IEEE Trans. Information Theory*, 2(4):115–116, 1956.

[Rén84] A. Rényi. *A diary on information theory*. Akadémiai Kiadó (Publishing House of the Hungarian Academy of Sciences), Budapest, 1984. With a foreword by Pál Révész, Translated from the Hungarian by Zsuzsanna Makkai-Bencsáth.

[Rom92] S. Roman. *Coding and information theory*, volume 134 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1992.

[Sal02] D. Salomon. *A guide to data compression methods*. Springer-Verlag, 2002.

[Sha48] C. E. Shannon. A mathematical theory of communication. *Bell System Tech. J.*, 27:379–423, 623–656, 1948.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Information Theory*, IT-23(3):337–343, 1977.