1 Introducción

1.1 ¿Para qué la compresión?

En los últimos años se ha dado un aumento espectacular tanto de la capacidad de almacenamiento de los ordenadores como de la velocidad de proceso de éstos.

A esto lo acompaña una bajada de los precios de memoria principal y secundaria así como también un aumento de velocidad de estos dispositivos. Esto nos hace preguntarnos ¿para qué la compresión?

Sin embargo, el auge que últimamente han tenido las redes de ordenadores hace que cada vez más usuarios pidan más prestaciones a la red sobre la que están conectados. Prestaciones que, como siempre, están por encima de las posibilidades reales. Cuando hablamos de posibilidades nos referimos principalmente a la velocidad de transferencia de datos. Este es el principal *handicap* al que se enfrentan todas las redes.

El cambio a mayores velocidades no es tarea fácil. Básicamente por los siguientes motivos:

- Las grandes compañías de redes WAN son lentas en cuanto a cambios se refiere. Esto se debe a su vez al volumen de cambio requerido, teniendo en cuenta la cantidad de infraestructura que debe ser modificada (cableado, tecnologías, etc.)
- Otro problema es la falta de tecnología que acepte unas velocidades muy elevadas de transmisión. Por ejemplo, podemos pensar que al usar cable de par trenzado para el teléfono podríamos tener una velocidad de transferencia desde nuestro módem de, digamos, 1 Mbps. sin problemas. El problema viene con las largas distancias. Si una compañía de servicios telefónicos quiere mantener simultáneamente 1000 llamadas internacionales (lo cual no es un número muy disparatado), el cable (y la tecnología) debería permitir 1000 Mbps., lo cual es una velocidad lo suficientemente grande como para requerir una

inversión importante, contando además con que se consiga la tecnología apropiada.

En este entorno, para conseguir mayores prestaciones de velocidad, los implementadores de los programas deben recurrir a técnicas que les permitan superar de alguna manera las deficiencias físicas de la red y de los equipos de conexión (que les permita, por ejemplo, ofrecer videoconferencia a través de *módems* de 14.400 bps.)

La técnica más importante en este sentido es la compresión de datos. La compresión de datos es beneficiosa en el sentido de que el proceso de compresión-transmisión-descompresión es más rápido que el proceso de transmisión sin compresión. Podemos expresar la ganancia de velocidad con una fórmula sencilla:

$$\rho = \frac{2*\frac{D}{c} + \frac{r*D}{b}}{\frac{D}{b}} = \frac{\frac{2*b*D + r*c*D}{c*b}}{\frac{D}{b}} = \frac{2*b + r*c}{c}$$

donde ρ es la relación entre tiempo que se tardaría para transmitir comprimiendo y sin comprimir (en tanto por uno), c es la velocidad de compresión en bps (se supone igual a la velocidad de descompresión y depende del algoritmo), D es el número de *bits* que componen el mensaje a transmitir, b es la velocidad de transferencia de la línea en bps y r es el *ratio medio* de compresión del algoritmo utilizado, que se puede escribir como *bits comprimidos* / *bits totales* y se consigue a través de una medida **empírica**.

La fórmula queda sencilla teniendo en cuenta que en el numerador tenemos el tiempo que la transmisión tarda en realizarse si utilizamos compresión: el proceso de comprimir se realiza dos veces (compresión y descompresión) y tarda D/c segundos y la emisión se hace sobre una fracción de los datos originales dada por r, que es $r*D$ bits, a una velocidad de transmisión b: $r*D/b$ segundos.

En el denominador tenemos el tiempo de transmisión.

Vemos que en último término la fórmula no depende de la cantidad de datos a transferir, D, sino de las distintas velocidades de compresión y emisión (como cabría esperar). Siempre que ρ sea menor que 1, se conseguirá ventaja en velocidad al comprimir la información. Si suponemos que ρ es 0.5, el tiempo de transmisión utilizando compresión es la mitad que sin usarla.

Para un valor normal de $r = 1/2$, vemos que será rentable comprimir cuando $\rho < 1$, es decir, si $(2*b + r*c) / c < 1$, es decir, si $\underline{\boldsymbol{b < c/4}}$.

Si suponemos que la información ya está guardada en forma comprimida, la transmisión simplemente reduce su tiempo en un factor *r*.

Pero no sólo es para la transmisión para lo que se usa la compresión. También para el almacenamiento masivo. La necesidad de almacenamiento también crece por encima de las posibilidades del crecimiento de los discos duros o memoria. Nos basta pensar, por ejemplo, en el proyecto del *Genoma Humano* ó en los grandes servidores de *vídeo en demanda* con cientos o miles de películas, ocupando cada una varios Gigabytes.

1.2 Concepto y modelo de Información

La *Teoría de la Información* es la disciplina que se encarga del estudio y cuantificación de los procesos que se realizan sobre la *información*.

Para este estudio, es obvio que necesitamos una manera de *medir* la *información*. Tenemos que pasar de nuestra intuición a una definición matemática que

a. esté de acuerdo con nuestra visión intuitiva
b. nos permita medir y comparar los procesos sobre ella

Para llegar a una medida de la información, repasaremos la idea intuitiva que tenemos de ésta. En primer lugar vemos que la cantidad de información que nos proporciona cierto dato es menor cuanto más esperamos ese dato. Si una persona nos comenta el tiempo que hace en Londres, al decirnos “lluvioso” no obtenemos casi información, ya es lo que estábamos esperando con mayor probabilidad. Si por el contrario nos dicen que “soleado”, recibimos más información, ya que la probabilidad de que esto ocurra es menor.

Podemos considerar las informaciones acerca del tiempo en Londres como datos que recibimos de cierta variable aleatoria que, cada vez que le preguntamos, nos dice el tiempo que hace en Londres. Esta variable aleatoria se convierte en una *fuente de información*. Nuestra idea de la *probabilidad de ocurrencia* de cierto evento que nos da información queda ahora modelada por una variable aleatoria y su conjunto de mensajes y probabilidades asociadas. Para el caso anterior, tenemos, por ejemplo:

Tiempo	***Probabilidad***
Lluvioso	0.70
Soleado	0.30

Con este modelo tendremos algo más concisa nuestra idea de *información*: una fuente de información puede ser modelada como una variable aleatoria; al recibir un mensaje de esa fuente, obtenemos una cantidad de información, que depende ***sólo*** de la probabilidad de emisión de ese mensaje; además, la cantidad de información es una función creciente con la inversa de la probabilidad (cuanta menor probabilidad, mayor cantidad de información recibimos, como vimos con el ejemplo).

La *ganancia de información* que experimentamos, pues, al recibir un mensaje, se puede entender como la *reducción de incertidumbre sobre el estado de la fuente* que experimentamos tras recibir el mensaje.

La última consideración que puede hacerse es una que, por un lado es intuitiva y por otro parece intencionada para llegar a la cuantificación final de la *información*. Se refiere a que la cantidad de información que recibimos con n mensajes de una fuente de m posibles mensajes debe ser la misma que al recibir *un* mensaje de una fuente con m^n mensajes. Esto nos sugiere una función logarítmica para la cantidad de información, ya que si suponemos que tenemos dos fuentes equiprobables de m y m^n mensajes respectivamente, la información al recibir n mensajes de la primera fuente es

$$n * f(m)$$

donde f es la función creciente, por ahora sin determinar (nótese que al ser la fuente equiprobable, la probabilidad de cualquier mensaje es $1/m$. Pero f es función de la *inversa* de la probabilidad, por lo que depende de m). Por otro lado, la recepción de *un* mensaje de la segunda fuente nos da una información

$$f(m^n)$$

Al hacer la igualdad, tenemos que $n * f(m) = f(m^n)$, sugiriendo como se dijo una función logarítmica.

En 1948, **Claude Shannon** propuso una medida para la información que cumplía todas la expectativas anteriores. Si suponemos una fuente **F** de información de n mensajes $\mathbf{F_i}$, cada uno con probabilidad $\mathbf{p_i}$, tenemos que la información al recibir un mensaje queda como:

$$I(F = F_i) = -\log(p_i)$$

Teniendo esta medida, podemos hallar la *medida de información media* de la fuente de información ó *entropía* de **F** haciendo simplemente la esperanza matemática de la información de cada símbolo:

$$H(F) = \sum_{i=1}^{n} p_i * I(F = F_i) = -\sum_{i=1}^{n} p_i * \log(p_i)$$

Si utilizamos logaritmos en base 2, esta medida estará en *bits*. Si utilizamos neperianos, obtendremos *nats*. Pero lo que importa es que esta medida nos da una cota superior de la cota de compresión. No se puede inventar ninguna codificación que consiga una longitud en bits media por símbolo emitido menor que la entropía de la fuente sobre la que se realiza la codificación. Esto es, sin embargo, una cota teórica. Los compresores actuales no llegan a esta cota pero quedan muy cerca.

1.3 Tipos de compresión

Es muy difícil clasificar los distintos tipos de compresión de datos que existen, debido a que, por un lado tenemos muchos algoritmos: LZ77, LZ78, LZW, Huffman, aritméticos, fractales, MPEG, JPEG, etc.

Además, hay muchas aplicaciones que utilizan la compresión con distintas expectativas: compresión de datos, vídeo y voz, compresión en tiempo real, etc. A su vez, cada uno de estos usos requiere unas características de velocidad, reversibilidad (que el algoritmo pueda ser aplicado de forma reversible para obtener los datos originales), pérdida mínima de información (en el caso de los algoritmos con pérdida de información, etc.).

Debido a esto, se ha elegido una clasificación muy general tomando como característica de división la reversibilidad del algoritmo. Así, los algoritmos de compresión se pueden dividir en dos tipos:

- Compresores ***lossless*** o sin pérdidas, en el sentido de que guarda absolutamente toda la información original (es reversible). Se utilizan para la compresión de datos, en los que no se puede dar pérdida de información.
- Compresores ***lossy*** o con pérdidas. La compresión hace que se pierda información de la fuente original. Sin embargo, esta pérdida es insignificante en comparación con la ganancia en compresión. Se utiliza sobre todo en imágenes y sonido, donde se puede "engañar" a los sentidos, donde una pérdida de calidad apenas es percibida (pero ocasiona un *ratio* de compresión mucho mayor).

2 Compresión *lossless*

En esta sección se hará un repaso a los distintos tipos de compresión *lossless* más comunes. Posteriormente se tratará la compresión *lossy*. Se comenzará exponiendo en líneas generales cuales son los tipos de compresión *lossless* más conocidos y utilizados, exponiendo algunos ejemplos de algoritmos reales que apliquen cada tipo de compresión. Finalmente se comentarán posibles adaptaciones para los algoritmos de manera que puedan ser utilizados en entornos de compresión en tiempo real. Como un último punto, se comentará cual puede ser el futuro de la compresión de datos *lossless*.

2.1 Tipos de compresión *lossless*

Hay básicamente dos tipos de compresores / algoritmos de compresión lossless:

- Compresores **estadísticos.**
- Compresores **basados en diccionario** ó **sustitucionales**.

No parece extraño que habiendo usado una medida de información basada en las probabilidades nos encontráramos con compresores que utilicen las propiedades estadísticas de la fuente de información para mejorar la codificación (el conjunto de símbolos de salida asociados a cada mensaje emitido por la fuente) de los mensajes de la fuente. Estos son los compresores **estadísticos**.

Este tipo de compresores parten de:

a. una fuente de información de *n* mensajes
b. las probabilidades de aparición de cada mensaje de la fuente (que pueden ser extraídas a priori de forma experimental o pueden ser dadas y fijas)
c. un *alfabeto* de salida que consta de una serie de símbolos (por ejemplo, el alfabeto binario consta de los símbolos 0 y 1).

y su objetivo es asignar una *codificación* para los mensajes de la fuente. Una *codificación* es una función que a cada mensaje de la fuente asigna una cadena de símbolos del alfabeto de salida. Esta codificación debe ser tal que explote la

redundancia en la información dada por la fuente para producir compresión. Si utilizamos un alfabeto binario, la esperanza matemática de la longitud de las cadenas de símbolos de la codificación, tomando como probabilidades las de los mensajes a los que representan, se debe acercar a la cota teórica de Shannon. Si es igual, la compresión es perfecta: hay una máxima eficiencia.

Por su parte, los compresores **sustitucionales** mantienen un diccionario de las cadenas de mensajes que han sido emitidas anteriormente por la fuente. Cada cadena está representada por un índice en el diccionario. Al procesar los mensajes de la fuente, si una cadena ya ha sido recibida anteriormente, se sustituye en la salida por el índice que ésta ocupa en el diccionario. Como los índices son normalmente más pequeños que las cadenas, se consigue compresión. Otra técnica es utilizar la propia entrada como diccionario, y codificar las repeticiones como un "salto hacia atrás" y una longitud de coincidencia. Más sobre esto después.

2.2 Compresores estadísticos

En esta sección examinaremos los compresores que utilizan la información de las probabilidades de los mensajes de la fuente para construir una codificación.

Para simplificar, supondremos que nuestras fuentes de información son ficheros ASCII de 8 bits y que cada carácter es un mensaje de la fuente (256 mensajes). El conocer las probabilidades de cada mensaje implica que el compresor debe realizar una primera pasada por el fichero para recuperar las frecuencias (o probabilidades, ya que son inversas) de cada mensaje (byte ASCII). Esto puede ser un problema, sobre todo para dispositivos de cinta de una sola pasada. Sin embargo, se puede argumentar que la codificación se puede realizar sobre bloques (aunque dependiendo del tamaño de los bloques se puede degradar la eficiencia o acercamiento a la cota de Shannon) ó utilizarse algoritmos adaptativos, que después se comentarán también.

Entre los compresores estadísticos podemos encontrar varios tipos:

- Compresores del tipo **Huffman** ó **Shannon-Fano**
- Compresores **aritméticos**
- Compresores **predictivos**

2.2.1 Compresores Huffman y Shannon-Fano

Ambos algoritmos terminan construyendo un árbol que representa la codificación que de los mensajes de la fuente se ha realizado, de manera que los nodos hoja contienen cada uno de los mensajes emitidos por la fuente. En el caso más simple, el alfabeto de salida en el que se realiza la codificación es binario. Esto quiere decir que de cada nodo partirán dos ramas, una para el 0 y otra para el 1. El código para cada mensaje se construye siguiendo el camino desde el nodo raíz hasta la hoja que representa el mensaje.

Nótese que este esquema nos permite que si, a la hora de descomprimir, el decodificador Huffman o Shannon-Fano poseen el mismo árbol que se hizo para comprimir, la decodificación es tan sencilla como leer *bits* de la fuente a descomprimir y seguir el camino desde la raíz hacia abajo bifurcando hacia un lado o hacia otro dependiendo del valor del bit. Eventualmente llegaremos a una hoja, que representa al mensaje que se estaba recibiendo.

Pero esto todavía no tiene nada de particular. Lo verdaderamente importante es que la codificación resultante de la aplicación de estos algoritmos asigna longitudes de codificación **inversamente proporcionales** a la probabilidad de aparición de cada mensaje. Es decir, el mensaje que más aparezca tendrá una codificación más corta, con lo que se ahorrará espacio en la transmisión. Recuérdese que los mensajes con más probabilidad son los que menos información daban, por eso, asignamos menos símbolos del alfabeto de salida en su codificación.

Hemos hablado del objetivo de ambos algoritmos: la construcción del árbol de códigos que representa a la codificación obtenida. En donde difieren es en la manera de conseguir el fin. Ambos construyen el árbol de manera que los mensajes con menor probabilidad quedan más abajo en el árbol, sin embargo, Huffman lo hace *botton-up* y Shannon-Fano *top-down*.

En cada paso, Huffman recoge los dos nodos con **menor probabilidad** del árbol. Construye entonces un nodo padre de ambos y se la asigna la probabilidad suma de ambos hijos. Este proceso hace que el árbol crezca y los nodos con menor probabilidad queden más fondo, tal y como queríamos. Además, es óptimo y alcanza la máxima eficiencia cuando las probabilidades de los mensajes son potencias exactas de dos. La implementación de este algoritmo es muy sencilla si utilizamos una estructura de datos conocida como *heap*. Un *heap* es un árbol binario completo de tal manera que cualquier nodo es menor que cualquiera de sus hijos. En particular la raíz contendrá al menor de todos. Las inserciones y extracciones de un *heap* se hacen ambas en $O(\log_2 n)$. Un ejemplo de código C++ que realiza la construcción del árbol podría ser el siguiente:

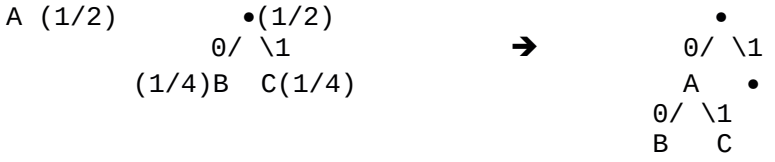
```
      // Crear el heap que contendrá en principio los mensajes a codificar
      heap<nodo> h(nMsg * 2);       // mayor número de nodos del árbol

      // Insertar en el heap los mensajes a codificar
      for (i=0;i<nMsg;i++)
          if (msg[i]->freq != 0)          // Sólo los que hayan salido
            h.insert(msg[i]);

      // Cuando sólo quede uno, será el nodo que debajo contiene a todos
      // los mensajes. El árbol ya está construído. La raíz será h.get().
      while (h.numElem() != 1)
      {
            // Construir un nuevo nodo que será el padre de los dos nodos
            // menores
            nodoTmp = new nodo(0);        // Nuevo nodo, frecuencia 0

            // Recuperar el hijo izquierdo
            tmpSon = h.get();             // Coger el menor
            tmpSon->dad = nodoTmp;        // Actualizar su padre
            tmpSon->branch = 0;           // Y su rama (para el camino)

            nodoTmp->lson = tmpSon;       // Actualizar el hijo izquierdo
            nodoTmp->freq += tmpSon->freq;

            // Recuperar el hijo derecho
            tmpSon = h.get();             // Coger el menor
            tmpSon->dad = nodoTmp;        // Actualizar su padre
            tmpSon->branch = 1;           // Y su rama (para el camino)

            nodoTmp->rson = tmpSon;       // Actualizar el hijo derecho
            nodoTmp->freq += tmpSon->freq;

            // Insertar el nuevo nodo padre
            h.insert(nodoTmp);
      }

      .
      .     // Procesar el árbol, es decir, construir los códigos para
      .     // cada mensaje y codificar la entrada.
      .

      delete h.get();                     // Borra todos los nodos
}
```

Al contrario, Shannon-Fano trabaja desde arriba hacia abajo. Al principio considera todos los mensajes en un solo conjunto. En cada paso, divide el conjunto en dos conjuntos que contengan **casi la misma probabilidad** (es claro que en general la exactitud no es posible). A cada uno de estos dos conjuntos les asigna un valor de bit: 0 ó 1. El proceso se repite recursivamente para cada conjunto generado. Al final se llegará a conjuntos de un solo elemento, los cuales representan cada uno a un mensaje. El árbol se construye así *top-down*. Al ir dividiendo por conjuntos que tengan probabilidad similar, contribuimos a conseguir conjuntos de menos elementos cuanta más probabilidad tengan. Cuantos menos nodos tenga un conjunto, más arriba quedará cada uno

y su codificación será menor. Sigue más o menos el mismo principio, pero este ya no es óptimo.

Un ejemplo de la construcción de un árbol de Huffman lo podemos ver así. Sean los mensajes y probabilidades siguientes

el algoritmo Huffman seguirá los siguientes pasos:



en donde primero se agrupan B y C en un único nodo de probabilidad ½ y posteriormente se construye el árbol final: A = 0, B = 10 y C = 11 (en binario) es decir, 1.5 bits de media por mensaje (no 8 bits / byte como en un fichero con sólo A's, B's y C's).

El principal problema de estos algoritmos es que tienen que el compresor debe realizar primero un recorrido del fichero (o de la porción del mismo, si se realiza por bloques) a comprimir para reunir las frecuencias (o probabilidades, como guste) de los mensajes de la entrada. Como el descompresor no tiene esa posibilidad, ya que sólo recibe los códigos asignados a los mensajes, el árbol ya procesado o las frecuencias de cada mensaje, junto con los datos, deben ser pasados al descompresor. Esto introduce una sobrecarga que hay que evitar de alguna manera. Uno de los programas más antiguos que yo he visto de compresión de datos se llama SQ, de Richard Greenlaw. La solución que aquí se le daba era transmitir el árbol de forma comprimida. Una solución así se encuentra también en uno de los pasos del algoritmo *Imploding* de *PKZIP 1.x*.

Otra solución también es la de hacer estos algoritmos *adaptativos*, en el sentido de que se va construyendo el árbol de manera dinámica tanto por el compresor como por el descompresor. Así el árbol no tiene que pasarse al descompresor. Hay varias adaptaciones de estos algoritmos para convertirlos en adaptativos. Pero quizás la adaptación más intuitiva es la siguiente:

El compresor parte inicialmente de un árbol vacío, que sólo contiene una hoja vacía (*empty leaf*), que siempre estará presente en el árbol, digamos, en la posición más arriba y más a la derecha posible del mismo. Cuando el compresor lee un carácter que no está en el árbol, su salida es el código de la hoja vacía, seguido del carácter que no pertenece al árbol. Acto seguido inserta el nuevo símbolo modificando el árbol para que siga siendo un árbol Huffman. Con esta técnica el receptor ya no tiene que conocer a priori el árbol ni las fre-

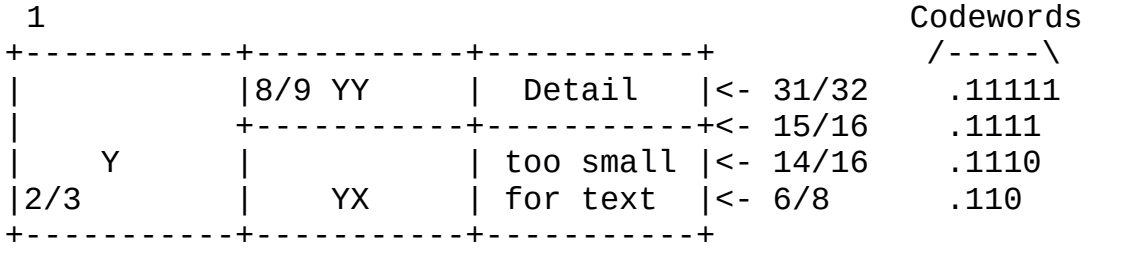
cuencias, ya que los datos comprimidos llevan la información suficiente para reconstruirlo. Sin embargo, este método no da tan buenos resultados como el anterior, ya que hay un periodo de aprendizaje en el que se emiten muchos bits adicionales. Una explicación más detallada se encuentra en [6].

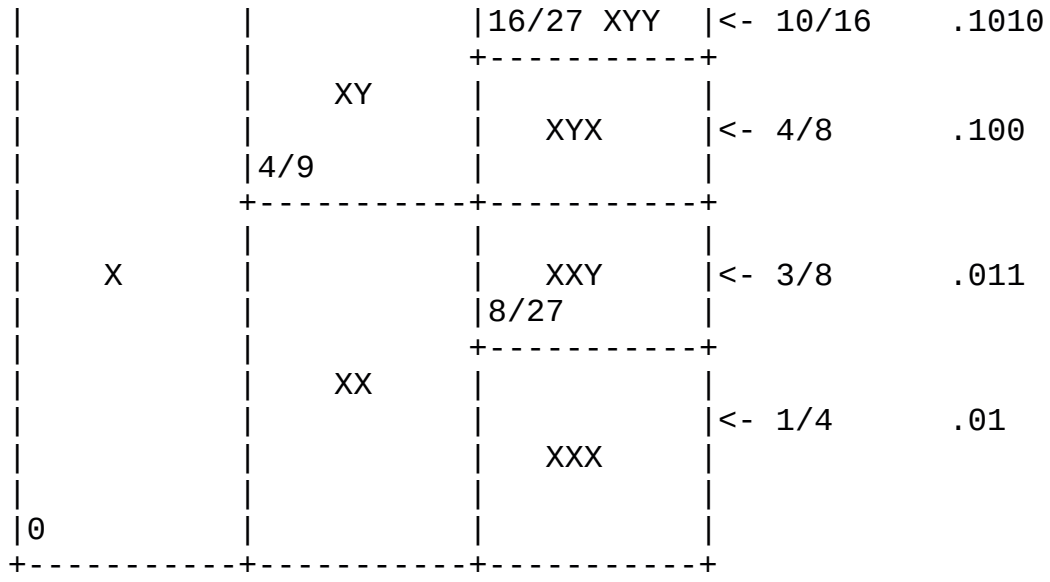
Otra cuestión práctica que resolvía también SQ es la inclusión de los códigos en palabras de 16 bits. Como se ha visto, el algoritmo no pone restricción a la longitud de los códigos generados, pero una posible implementación podría restringir el tamaño máximo de los códigos. SQ trata este problema haciendo el árbol más "achatado", haciendo las comparaciones de los nodos a elegir no sólo sobre la probabilidad de cada uno, sino también sobre la profundidad en el árbol de cada nodo. Esto limita también la eficiencia de los códigos generados.

2.2.2 Compresores aritméticos

Al igual que los anteriores, los compresores aritméticos se basan en las probabilidades de ocurrencia de los mensajes a la entrada. Sin embargo, para producir una codificación toman un esquema totalmente diferente. Se basan en la representación de un valor del intervalo [0,1] con más decimales (más precisión) cuanto más información contengan los datos a comprimir. Supongamos que queremos codificar una entrada que consta de dos símbolos, X e Y, con probabilidades p(X) = 2/3 y p(Y) = 1/3.

Al recibir una X, dividimos el intervalo [0,1] y nos quedamos con los 2/3 inferiores, por ejemplo. Tendremos entonces el intervalo [0, 2/3]. En caso de haber recibido una Y, habríamos cogido el intervalo del tercio inferior, es decir, [2/3, 1]. En cada paso, dividimos por los 2/3 inferiores o el tercio superior el intervalo que tengamos. Al final, el código que emitimos, en binario, es un número que cae en el intervalo. Se elegirá aquel que utilice menos bits en su representación. Este número representa a **toda** la entrada Con este número, representado por los bits que necesite, junto con la información del número de elementos codificados y la probabilidad de cada uno, el descompresor puede reconstruir la entrada. Un ejemplo de la codificación de tres mensajes con la anterior distribución de probabilidad puede ser la siguiente:





comp.compression FAQ (parte 2), Peter Gutmann

En la columna “Codewords” se especifica qué hilera de bits representa a cada grupo de tres mensajes de la fuente. El compresor construirá sólo los trozos de la tabla que le sean necesarios según la entrada. Tras recibir, por ejemplo, XXY, decide que emitirá el número 3/8 (en binario .011) en representación de la entrada.

El proceso que el descompresor realizará al recibir los datos comprimidos, el número de mensajes de que consta y las probabilidades de cada mensaje, procederá como sigue: si suponemos que se recibe .011, el descompresor verá que este número pertenece a los 2/3 inferiores, luego el primer mensaje que apareció fue una X. Sabe que en total son tres, por lo que continúa. También cae dentro de los 2/3 inferiores de [0, 2/3], es decir, el segundo mensaje también fue una X, hasta ahora XX. Sin embargo, 3/8 (.011) está en el tercio superior del intervalo [0, 4/9], por lo que el tercer mensaje es una Y. Eran tres mensajes. Fin. La secuencia original era XXY. ¡Con tres *bits* hemos codificado tres *bytes*!

Este algoritmo es muy eficiente, y no deja de ser curioso e ingenioso. El problema reside, al igual que los algoritmos anteriores, en que las probabilidades deben ser dadas al receptor. La solución también tiende aquí hacia una modificación “adaptativa con la entrada”. Los modelos que van dividiendo los intervalos se convierten ahora en modelos adaptativos (cambiantes dinámicamente con la entrada) y que son capaces de sincronizar con la mínima información a compresor y descompresor. Ejemplos de ellos son el *Dynamic Markov Modeling* o el *PPMC* (*Partial Predictive Matching*), lo suficientemente complejos como para no tratarlos aquí.

2.2.3 Compresores predictivos

Los compresores predictivos son, al contrario que los anteriores, totalmente adaptativos. Procuran **predecir** el siguiente mensaje de la entrada tomando como base de conocimiento la entrada procesada hasta ese momento (en el fondo, también probabilidades). Si el mensaje que se encuentra a la entrada coincide con el predicho, su codificación se podrá hacer con menos bits. Si no, su codificación se hará con más bits, que permitirán entonces sincronizar al descompresor para que mantenga sus tablas internas idénticas a las del compresor sin pasárselas explícitamente.

Poseen ciertas ventajas sobre los anteriores algoritmos, entre ellas su velocidad: al actuar sobre un mensaje cada vez y realizar una predicción que generalmente suele ser de cálculo sencillo, son capaces de dar una alta velocidad de compresión/descompresión. Velocidad y sencillez son dos conceptos que generalmente van unidos, por lo que estos algoritmos, a la vez que rápidos, resultan sencillos de programar, por lo que se pueden convertir en una solución barata para sistemas de compresión transparente en tiempo real, con unas relaciones de compresión aceptables.

Sin embargo, para algunas aplicaciones, no son tan aceptables las relaciones de compresión. Además, su mejora es sustancialmente difícil: la predicción es muy escurridiza en cualquier ámbito. Una idea sería introducir información adicional para cada tipo de fichero que nos diera un patrón con el que predecir en mejores condiciones. En este sentido (aunque no utilizando algoritmos predictivos) se mueve el compresor UC2 de *AIP Development*.

Quizá el compresor más rápido que se haya diseñado nunca pertenece al grupo de los compresores predictivos. Se llama "predictor", y fue inventado en 1987 por Timo Raita y Jukka Teuhola, de la Universidad de Turku, en Finlandia [13]. Fue patentado en 1993 por K. Thomas, y se usa en el *draft* de Internet "PPP Predictor Compression Protocol" (ver ftp://venera.isi.edu/internet-drafts/draft-ietf-pppext-predictor-00.txt). Nos servirá para mostrar un ejemplo de este tipo de compresores.

Su método de predicción es sencillo: predice el siguiente carácter a partir de los dos anteriores de la entrada. Para ello construye una matriz de 256*256 que guarda en cada casilla m[i,j] el byte que anteriormente siguió a dos entradas consecutivas de valores ASCII i y j. Cuando se va procesando la entrada, el algoritmo siempre sabe qué dos mensajes precedieron al actual (salvo para los dos primeros mensajes de la entrada, pero esto no es problema), por ejemplo p1 y p2. Con esta información, su predicción para el carácter actual, pongamos c, será m[p1,p2]. Si acierta, es decir, si c = m[p1,p2], la salida será sólo un bit, que, puesto a 1 informa de que se ha logrado predecir el mensaje actual. Si no

acierta, su salida será un bit puesto a 0, indicando que no se predijo y a continuación el mensaje no predicho. Además, actualiza la tabla para que la vez siguiente sea capaz de predecir: m[p1,p2] = c. Con esta información es capaz de sincronizar al descompresor de manera que la tabla que ambos poseen en memoria es idéntica.

El descompresor parte de la tabla vacía inicial, igual que el compresor. Al recibir un bit a 1, sabe que el mensaje que se transmitió es el que se encuentra en la tabla en la posición indicada por los dos últimos mensajes resultado de la descompresión realizada hasta el momento. Si recibe un bit a 0, sabe que la siguiente información será el mensaje tal cual, y podrá actualizar su tabla al igual que el compresor.

A continuación veremos un ejemplo de cómo quedaría la implementación del compresor y el descompresor. Pero antes consideraremos la matriz inicial. Está claro que ambos, compresor y descompresor deben comenzar con la misma matriz de predicción. Además, los valores presentes en esta matriz *a priori* van a condicionar gran parte del proceso. Normalmente es inicializada a un valor que es muy probable: en ficheros de texto se puede inicializar toda a espacios, y en ficheros binarios al ASCII 0.

El siguiente código muestra un ejemplo de las funciones de compresión y descompresión:

```
// Comprime el stream de entrada en el de salida
int   PREDICTORCompress(stream in,stream out)
{
      char c;                 // Carácter actual a predecir
      char p1 = '\0';         // Último carácter de la entrada
      char p2 = '\0';         // Penúltimo carácter de la entrada
                              // Inicialmente ambos se suponen '\0'
      char matriz[256][256];  // Matriz de predicción. Se supone
                              // inicializada a '\0'

      // Procesar toda la entrada
      while (!in.eof())
      {
            // c es el carácter a predecir
            c = in.getNextChar();

            // ¿Se predice?
            if (c != matriz[p1][p2])
            {
                  // No, la salida es un bit a 0 y el carácter
                  out.putBit(0);
                  out.putChar(c);

                  // La siguiente vez irá mejor
                  matriz[p1][p2] = c;
            }
```

```
            else
            {
                  // Se ha predicho. Sólo se saca un bit
                  out.putBit(1);
            }

            // El carácter que era el último pasa al penúltimo
            // y el último es el c
            p2 = p1;
            p1 = c;
      }

      // Todo bien
      return 0;
}


// Descomprime el stream de entrada en el de salida
int   PREDICTORDecompress(stream in, stream out)
{
      char c;                 // Carácter actual
      char p1 = '\0';         // Último carácter de la salida
      char p2 = '\0';         // Penúltimo carácter de la salida
                              // Inicialmente ambos se suponen '\0'
      char matriz[256][256];  // Matriz de predicción. Se supone
                              // inicializada a '\0'

      // Procesar toda la entrada
      while (!in.eof())
      {
            // Se ha predicho
            if (in.getNextBit())
                  c = matriz[p1][p2];
            else  // No predicho
            {
                  c = in.getNextChar();

                  // Ajustar la matriz para sincronizar
                  matriz[p1][p2] = c;
            }

            // Salida: el carácter actual
            out.putChar(c);

            // Rotar
            p2 = p1
            p1 = c;
      }

      // Todo bien
      return 0;
}
```

Como último punto, puede parecer a simple vista que este esquema es "demasiado sencillo para funcionar". Sin embargo, en realidad funciona bien y bastante rápido. La compresión y descompresión se hacen en tiempo real y, como un ejemplo, se dirá que, por ejemplo, el fichero COMMAND.COM de la

versión MS-DOS 6.20 que ocupa en disco 56.539 bytes queda comprimido a 44.818, es decir, a aproximadamente un 79% de su tamaño. Este resultado es bueno, sobre todo teniendo en cuenta que este es un archivo binario de código, con una distribución que generalmente tiene poca redundancia.

No es este el único ejemplo de compresores predictivos. Hay otras alternativas, como la que se incluye en *PK-ZIP 1.x* con el algoritmo "Reducing". Consiste en considerar un conjunto de seguidores de cada carácter (*follower sets*) que guardan los caracteres que han seguido con más probabilidad a un carácter dado. Como el conjunto de seguidores es pequeño, para identificar un seguidor se pueden utilizar menos bits. El trabajo del compresor es aquí difícil, ya que tiene que calcular el tamaño óptimo de los conjuntos de seguidores de manera que no se pierdan bits inútilmente. El problema aquí también es que los conjuntos deben ser pasados al descompresor. Un ejemplo en pseudocódigo que muestra la descompresión y la codificación que del conjunto de seguidores se hace se puede encontrar en [7].

Por último, estos algoritmos son malos a la hora de controlar grandes espacios de caracteres iguales. Por ello es conveniente hacer la compresión en dos partes, como la realiza el "Reducing", primero comprimiendo los caracteres consecutivos iguales utilizando una variante de RLE (*Run-Length Encoding*, que después veremos) y a la salida el compresor predictivo.

2.3 Compresores basados en diccionario

Un enfoque diferente a los algoritmos anteriores es el que presentan los compresores basados en diccionario. Su idea es construir un diccionario tomando como referencia la entrada procesada hasta ese momento. El diccionario contiene las cadenas de mensajes (en nuestro ejemplo práctico las cadenas de caracteres ASCII o bytes). Estas cadenas están identificadas por un índice, de manera que índice y cadena son de correspondencia biunívoca. El resultado práctico a todo esto es que si en algún momento la entrada que se está procesando actualmente es una cadena que está presente en el diccionario, el compresor puede dar como salida el índice que identifica esa cadena en el diccionario. Teniendo en cuenta que las cadenas pueden ser arbitrariamente largas, la producción del índice en lugar de la cadena representa un ahorro de información, y por lo tanto, compresión.

Incluidos dentro de este grupo consideraremos a los algoritmos *RLE*, que puede ser considerado como poseyendo un diccionario de longitud 1 byte,

el *LZW* y otros derivados de LZ78 y las variantes de LZ77. Veremos todos ellos en orden.

2.3.1 Compresión RLE

Este es, quizá, el compresor más sencillo que existe. También es de los más ineficaces. Al oír por primera vez el término compresión de datos, la mayoría piensa en reducir de alguna manera un número de caracteres repetidos, con la sencilla sustitución de decir "el carácter X se repite N veces". Así es, este es el principal objetivo del algoritmo: reducir las cadenas de caracteres idénticos a una indicación del carácter y la longitud de la repetición.

Se ha incluido al RLE dentro de los compresores basados en diccionario porque se puede considerar que utiliza un diccionario deslizante de tamaño 1 byte para predecir el siguiente carácter a la entrada, aunque así también podría ser clasificado de predictivo; pero la cuestión importante no es su clasificación, sino su estudio.

Aunque el algoritmo parte de una idea sencilla y es fácil de comprender su modo de operación, su implementación no siempre está libre de trampas. En primer lugar, el primer carácter no tiene predecesor. También debemos llevar un carácter de adelanto ó *lookahead* para compararlo con el actual. Si son iguales, es el comienzo de una cadena de repeticiones; si no, la salida debe ser el carácter actual y el *lookahead* pasa ahora a ser el actual. El manejo de un carácter de adelanto requiere entonces que el último carácter se procese de forma especial, al no haber más posibilidad de adelanto.

Existe, además, más de una manera de implementarlo, y todas ellas están patentadas, así que ¡cuidado!

La primera forma, más ineficiente, es utilizar un carácter, llamado comúnmente *DLE*, que sirva para indicar que se ha producido una repetición de un carácter. Para indicar que el carácter es realmente DLE, se puede indicar con la secuencia DLE,0 (cero). He aquí un ejemplo: supongamos que utilizamos la codificación

DLE, número de repeticiones, carácter repetido

para indicar una repetición y

DLE, 0 (cero)

para indicar que se está transmitiendo efectivamente el carácter correspondiente a DLE. Véase cómo "número de repeticiones" debe ser mayor o igual que 3 para que la sustitución tenga el efecto de reducir el número de mensajes a la salida. Con esta codificación podemos representar repeticiones desde 3 hasta 255 caracteres de longitud, dejando incluso libres las codificaciones DLE,0, DLE,1 y DLE,2 para otros cometidos (como representar el mismo DLE). Supongamos además la siguiente entrada, la cual queremos comprimir usando RLE:

'1' '2' '4' '2' (char)**DLE** 'A' 'B' 'B' 'B' 'B' 'B' 'B' 'B' 'B' 'B' 'C' 'C' 'C' 'C'

(se utiliza notación C. Todos los símbolos representan caracteres ASCII, menos DLE, que representa al valor ASCII de un carácter elegido, bien al azar, bien con el propósito, por ejemplo, de sincronizar dos fuentes síncronas). La codificación sería

'1' '2' '4"2' (char)**DLE** (char)0 'A' (char)**DLE** (char)9 'B' (char)**DLE** (char)4 'C'

Este modo de codificación representa el problema de que se debe hacer "transparente" al carácter DLE añadiéndole el carácter ASCII de valor 0. Una entrada que contuviera muchos caracteres DLE se vería levemente afectada por la compresión debido a la sobrecarga introducida por este carácter.

Otra forma de aplicar RLE es utilizando el mismo método que las imágenes PCX estándar. En vez de utilizar un carácter de identificación de repetición utilizan un carácter de "centinela". Este carácter tiene un bit que informa si la siguiente información es acerca de una repetición o son datos sin repetición. Los restantes bits del byte pueden ser considerados como el número de caracteres siguientes a los que afecta el centinela. En caso de ser repetición, este número indicará el número de repeticiones (2 hasta 127), y será seguido del carácter que se repite. En caso de no repetición, indica la posición del siguiente centinela y que se tienen que copiar a la salida los siguientes bytes hasta el centinela. Si suponemos que el centinela utiliza su bit más significativo para indicar con un 0 que no hay repetición y con un 1 que sí la hay, la anterior secuencia queda ahora comprimida como sigue:

(char)6 '1' '2' '4' '2' (char)DLE 'A' (char)(0x80 + 9) 'B' (char)(0x80 + 4) 'C'

donde se han subrayado los centinelas. El primero tiene su bit de repetición a 0. Los demás lo tienen a 1. Aparte de suprimir el enmascaramiento de DLE's, se ha disminuido la longitud mínima de la repetición a codificar a dos, en vez de tres, como antes.

2.3.2 LZ78: LZW

Los compresores basados en diccionario son, con mucho, los más utilizados. Normalmente se usan en combinación con compresores estadísticos en dos fases. Las dos familias más importantes de compresores basados en diccionario nacieron de los trabajos de dos matemáticos, Abraham Lempel y Jakob Ziv en los años 1977 y 1978; estas son LZ77 y LZ78. Actualmente se utilizan más los compresores derivados de LZ77, junto con un compresor estadístico a la salida. También, como veremos, pueden ser aplicadas técnicas estadísticas en los compresores LZ78.

En esta sección veremos el representante más famoso y sencillo de la familia LZ78: LZW. Estudiaremos su funcionamiento, los problemas que entraña su implementación y cómo se han resuelto por implementaciones conocidas como la de *Compress* de *UNIX*. A continuación se verá las posibles mejoras al algoritmo, bien dadas por la aplicación de técnicas estadísticas, bien por mejoras introducidas en variaciones sobre la idea original, como son AP de *Storer* [4] o la codificación MW o la Y [8].

Terry A. Welch, de UNISYS, introdujo y patentó (un terreno cenagoso el de las patentes, ver [8] y [2] por ejemplo) una variante de LZ78 llamada LZW [9, 10]. El propósito del algoritmo es construir un diccionario en el que se guardan todas las cadenas que han aparecido en la entrada. A cada cadena se le asigna un identificador (un número) que la representa. Al ir codificando la entrada, si nos encontramos con una cadena que ya está en el diccionario, la salida del algoritmo será el código de la cadena en el diccionario. El descompresor debe ir construyendo el mismo diccionario que el compresor, pero éste no tiene que pasarse explícitamente, sino que está implícito en la codificación.

La codificación es sencilla (al menos la idea), y se puede resumir como sigue: al principio el compresor parte de un diccionario en el que se han introducido todas las cadenas de longitud 1, es decir, 256 cadenas que constan de un solo carácter (los caracteres ASCII). El algoritmo para comprimir la entrada y mantener actualizado el diccionario es el siguiente:

```
set w = NIL
loop
    read a character K
    if wK exists in the dictionary
        w = wK
    else
        output the code for w
        add wK to the string table
        w = K
endloop
```

Como podemos ver, se va leyendo cada carácter de la entrada secuencialmente y se va construyendo en *w* la cadena que se buscará en el diccionario. Al principio *wK* = *K*, ya que *w* = *NIL*. Por lo tanto, se continúa y *w* = *K*. al llegar el siguiente carácter, *K'* se busca en el diccionario la cadena *wK'* = *KK'*, la cual no está en el diccionario. En este caso la salida es el código para *w*, es decir, *K*, y se añade *wK* al diccionario. Añadir la cadena al diccionario significa asignarle un identificador secuencial consecutivo a partir de la anterior cadena. Como al principio las primeras 256 posiciones están ocupadas el siguiente número a asignar es el 257. Este número ya no cabe en 8 bits, por lo que las salidas (los identificadores de las cadenas reconocidas) del algoritmo son al principio de 9 bits e irán aumentando conforme se vaya quedando pequeño el diccionario.

Al principio, todas las cadenas de longitud 1 tienen asignado un identificador. Este se utiliza como representación de la cadena, y se corresponde con el código ASCII del carácter, pero con 9 bits inicialmente. Durante el proceso, se van añadiendo nuevas cadenas de longitud mayor. Al identificarse cadenas más largas con un número de varios bits, la compresión resulta bastante efectiva. Como un ejemplo de entrada podemos ver el siguiente, que ejemplifica el comportamiento del algoritmo para la cadena "*ESTEOESTE*":

K	wK	existe	diccionario	salida (9 bits)
E	E	Sí	-	-
S	ES	No	ES (256)	069 (E)
T	ST	No	ST (257)	083 (S)
E	TE	No	TE (258)	084 (T)
O	EO	No	EO (259)	069 (E)
E	OE	No	OE (260)	079 (O)
S	ES	**Sí**	-	-
T	EST	No	EST(261)	256 (ES)
E	TE	**Sí**	-	-

En esta entrada, aunque es corta, podemos ver cómo se ha llegado a construir la cadena "EST", representada con el identificador 261. Si posteriormente en la entrada se encuentra alguna cadena de las presentes en el diccionario, ésta se sustituye en la salida por su identificador.

La descompresión es más sencilla, aunque tiene que construir el mismo árbol. Nótese sin embargo que el descompresor recibe inicialmente códigos de 9 bits que usará como índices en el diccionario. Mantendrá el diccionario actualizado a partir de esa información y al recibir, por ejemplo, el código 256, en esa entrada, al igual que en el diccionario del compresor, contendrá la cadena "ES". El algoritmo de descompresión contiene un caso especial. El compresor puede insertar el código de la cadena antes de que el descompresor pueda construirla. Este es el caso de una entrada del tipo *XentreXentreX* en la que *Xentre* está en el diccionario. El compresor reconoce la primera parte: *Xentre* como existente en el diccionario. Al considerar la siguiente *X*, la salida es el

identificador de *Xentre* y añade al diccionario *XentreX*. Como lo siguiente que considera es precisamente *XentreX* y acaba de ser añadida al diccionario, su identificador es expulsado. La salida es entonces (identificador(*Xentre*), identificador(*XentreX*)), en donde este último no puede ser reconstruido por el descompresor. No obstante, el descompresor, al recibir un código que no existe en el diccionario, sabe que es el caso especial, por lo que la cadena de salida es la cadena de salida anterior junto con el primer carácter de esa misma cadena, introduciendo la cadena aumentada al diccionario. En el caso normal, el algoritmo trabaja de forma inversa al anterior. Un ejemplo en pseudo-código podría ser:

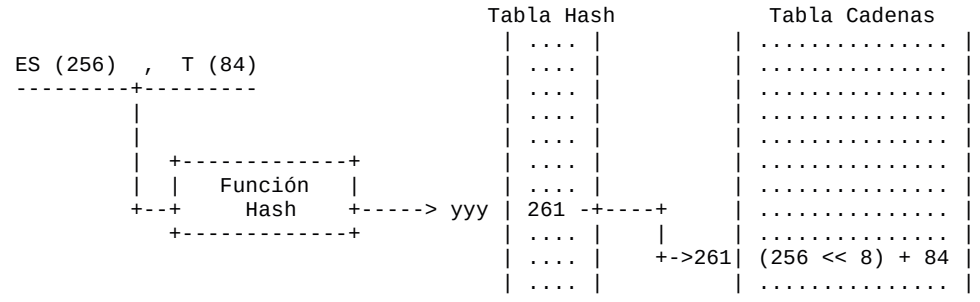
```
Read OLD_CODE
output OLD_CODE

WHILE there are still input characters DO
    Read New_CODE

    // ¿Caso especial?
    IF NEW_CODE is not in the dictionary THEN
        STRING = get string of OLD_CODE
        STRING = STRING+CHARACTER
    ELSE
        STRING = get string of NEW_CODE
    ENDIF

    output STRING
    CHARACTER = first character in STRING

    add OLD_CODE+CHARACTER to dictionary

    OLD_CODE = NEW_CODE
END WHILE
```

Hay varios aspectos que se pueden comentar del algoritmo que estarán relacionados tanto con su eficiencia como con su implementación. En primer lugar, se ha visto que el diccionario iba creciendo a medida que se iban insertando nuevas cadenas en él. Sin embargo, ningún ordenador tiene memoria infinita y debemos poner tope a esta inserción. Además, a medida que el diccionario vaya creciendo, se utilizarán más bits para identificar a cada cadena, con lo que para las cadenas cortas (que son también las más probables) se conseguirá una codificación pobre en el sentido de que ahorra pocos bits. Una vez que hemos puesto un tope (digamos de 12 a 14 bits, es decir, entre 4096 y 16384 cadenas), está claro que el diccionario se llenará y habrá que eliminar las cadenas que guarda. Pero ¿debemos desechar todas las cadenas del diccionario y "vaciarlo por completo" o podemos aprovechar las cadenas que más se han usado y dejarlas en el diccionario?. Esta es una decisión difícil y no hay solución óptima ni mucho menos. Se pueden ver alternativas en las diferentes implementaciones: unas vacían completamente el árbol, lo que resulta menos eficiente en términos de compresión; otras emiten códigos especiales para sincronizar al compresor y descompresor con los que indican que el árbol debe ser

"limpiado", es decir, que se deben eliminar las cadenas menos utilizadas. Esta última opción es la que utiliza el algoritmo "*Shrinking"* de *PK-ZIP 1.x* [7] y *Compress* para UNIX [12].

Otra de las cuestiones a tener en cuenta es la organización del diccionario. En el proceso de cada carácter, bien se hace un acceso al diccionario para leer, bien se hace para escribir, por lo que el diseño de esta estructura, que contiene a las cadenas, es un punto importantísimo. Teniendo en cuenta que el diccionario puede llegar a contener como mínimo 4096 cadenas de caracteres, un enfoque en el que para cada carácter se busque secuencialmente en todo el diccionario es prohibitivo en tiempo. También en espacio, ya que guardar tantas cadenas es impracticable.

Sin embargo, si nos fijamos en la manera en que se construyen las cadenas vemos que se unen, por un lado, una cadena en el diccionario (que puede ser representada por un índice de como máximo 16 bits) junto con un nuevo carácter a la entrada. Esto hace que una nueva cadena puede ser representada, como máximo, por un entero de 32 bits. El diccionario así queda como una tabla de 4096 entradas de enteros de 32 bits. Sin embargo, con esto sólo ahorramos espacio, no tiempo. Para ello, podemos usar un algoritmo de *hashing* [11, sec. 6], el cual asigna un número a cada cadena a través de una *función hash*. A través de este número podemos comprobar rápidamente (mirando en la tabla *hash*) si esta cadena ya ha aparecido (ya está en el diccionario). El siguiente dibujo esquematiza el proceso que el compresor haría (tomando el ejemplo de la anterior codificación) en el momento de introducir en el diccionario la cadena "*EST"*, a la que se le asigna el código 261. Utilizando el prefijo "*ES"* y el carácter "*T*", la *función hash* calcula un valor que utilizará para indizar la *tabla hash*. Esta tabla es un mapa entre identificaciones *hash* y entradas en el diccionario. La posición calculada de la tabla *hash* contendrá un índice del diccionario donde está la cadena. Como no está en el diccionario, en la posición *hash* se pone el identificador secuencial siguiente, igual que se iban asignando códigos a las nuevas cadenas que entraban en la tabla.



(261 es la siguiente entrada consecutiva en el diccionario ^)

Pero hay un problema con los algoritmos de *hashing*: dos cadenas pueden recibir el mismo código *hash*, representado en el dibujo por, *yyy*, lo que se

llama una *colisión*. Este problema no es trivial ni insignificante, ya que una mala (lenta) gestión de las colisiones nos hace tener un algoritmo impracticable en tiempo. Los distintos algoritmos expuestos en [11] tratan de manera diferente las colisiones. El que se ha elegido en *Compress* es el *algoritmo D* ó *direccionamiento abierto con doble hashing (open addressing double hashing)*. Es el que mejores resultados da en situaciones en las que las inserciones y búsquedas en la tabla dominan al borrado de elementos de la misma. A diferencia de otros algoritmos, como el L, trata de dispersar las colisiones en una tabla *hash* (cuyo tamaño es superior al necesario y además número primo) de manera que la posibilidad de colisión se reduzca. Esto se consigue aplicando dos funciones *hash*. Para más información véase [11] y [12].

También a partir del estudio de cómo las cadenas son insertadas en el diccionario se puede deducir que la estructura de árbol resulta apropiada. Un árbol que representa cadenas es denominado un *trie*. Diversas estructuras *trie* pueden aplicarse aquí (como por ejemplo los árboles de Patricia, *Patricia trees*, utilizados también en [3]).

Una posible mejora del algoritmo LZW reside en considerar que los identificadores son representados por un número variable de bits. Teniendo esto en mente, podemos aplicar técnicas estadísticas que nos representen con más bits los identificadores que menos se van usando y con menos bits los que más se usan.

En [8] se incluyen tres algoritmos que resultan de mejoras a LZW: AP (patentado por *Storer*), MW (de Miller y Wegman) e Y.

En el ejemplo de codificación LZW, se vio que las cadenas que se construían eran muy cortas, ya que en cada paso se le agregaba un solo carácter. MW, en vez de sólo añadir el último carácter, añade las dos últimas cadenas para formar una cadena más larga. Con esto se construyen cadenas más largas desde el principio que, en el ejemplo anterior habrían reducido a la mitad la salida. Por su parte, AP representa una mejora sutil del anterior. En que en vez de introducir sólo la cadena nueva al diccionario, se introducen todas las que resultan de unir la cadena que anteriormente coincidió con todos los prefijos de la cadena que ha coincidido ahora. Con ello se construyen más cadenas que posiblemente aparezcan después, ya que también son tomadas a partir de la entrada, es decir, también es adaptativo con la entrada, y que son más largas que las construidas por LZW.

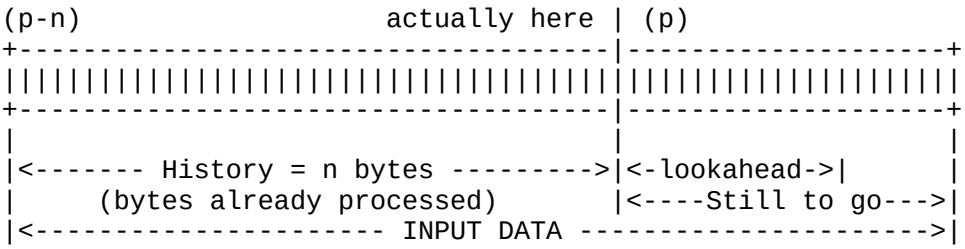
Y es una mezcla de ambos. Para cada carácter de la entrada, construye una lista de "*matches*" o prefijos de coincidencia, a los que añade el carácter leído. Los de la lista que no estén en el diccionario son añadidos a este, y los que sí están continúan para el siguiente carácter. Así se añade más de una cadena al diccionario para cada carácter.

2.3.3 LZ77

En LZ77 es donde realmente se concentra el interés actual. Estos algoritmos tienen tantas decisiones de diseño que cada uno implementado es distinto a los demás. Esto hace que no se puedan patentar o que se puedan saltar de una forma u otra las patentes actuales. De ahí el interés.

Las variantes de LZ77 también mantienen un registro de los últimos caracteres procesados de la entrada, igual que las LZ78. Pero al contrario que estas, no construyen un diccionario explícito. En cada momento, el algoritmo se encuentra procesando en un punto de la entrada, los *n* caracteres anteriores forman la "historia" del algoritmo ó "ventana", lo que equivale al "diccionario". Los caracteres posteriores al punto actual forman lo que se denomina el "*lookahead buffer"* ó buffer de adelantamiento. En cada paso, la cadena que comienza en el punto actual de la entrada se busca hacia atrás en la "historia". Si se encuentra una coincidencia que sea lo suficientemente larga como para tenerla en cuenta (después se comentará esta decisión), a la salida se sustituye la cadena coincidente por un par que indica el "desplazamiento hacia atrás" y la "longitud" de la coincidencia con la cadena hacia atrás. Como los pares (*desplazamiento*, *longitud*) ocupan menos que la cadena que coincidió, se obtiene compresión. Si no se encuentra una coincidencia, la salida es una copia *literal* de la entrada. Posteriormente se avanza la ventana (es decir, se avanza en la entrada) bien *longitud* si hubo coincidencia, bien un carácter si no la hubo.

El hecho de ir desplazando la ventana sobre la entrada hace que estos algoritmos sean llamados de "ventana deslizante". La siguiente figura ilustra el proceso del algoritmo:



Tomado en parte de LZRW3a, Ross N. Williams.

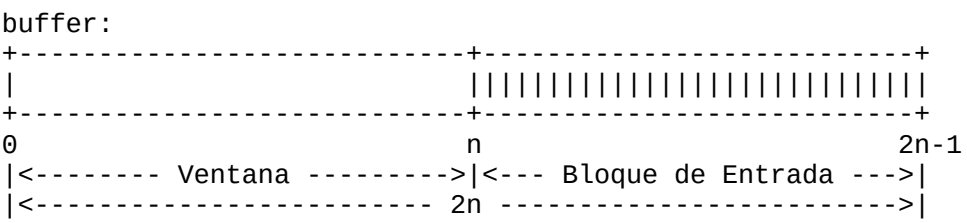
Esta idea tan sencilla, que posteriormente ha ido tomando cuerpo con otras mejoras, es la base de la mayoría de los compresores actuales más conocidos y potentes. El descompresor es sorprendentemente sencillo, ya que su "ventana" está formada por los datos que ha descomprimido anteriormente, y, cuando recibe un par (*desplazamiento, longitud*), sólo tiene que copiar a la salida *lon-*

gitud bytes que están *desplazamiento* bytes antes. En el caso de recibir un *literal*, este es copiado tal cual a la salida.

Existen muchas consideraciones acerca de este algoritmo que hacen que su implementación y estudio sean muy interesantes. Al igual que ocurría con los compresores LZ78, en cada carácter de la entrada se tiene que buscar una coincidencia en la entrada anteriormente procesada. Esto nos hace pensar en un esquema parecido al que introdujimos, utilizando tablas *hash*. Pero aquí hay un problema adicional: la ventana se va desplazando en cada carácter procesado, por lo que las cadenas para las que utilizamos el *hash* cambian de posición en cada avance de la ventana, y una asociación (*valor_hash, posición*) no es posible. Este no es un problema sencillo. En [3] se ofrece una solución con *árboles de Patricia* que está patentada. La solución adoptada en [5] es la que comentaremos aquí.

En primer lugar, se considera la entrada dividida en bloques de longitud *n* (tamaño de la ventana). La entrada se mantiene fija con respecto a los límites de los buffers en la codificación de cada bloque de *n* bytes. Se mantiene además en el buffer al menos *n* caracteres que forman la historia. Para ello se puede utilizar un buffer de *2n* caracteres y comenzar a codificar por la posición *n*. Los caracteres [0, (n-1)] son la ventana o historia (inicialmente vacía) del algoritmo. Los [n,(2n-1)] son la entrada a procesar del bloque actual.

El algoritmo comienza a realizar su función sobre el inicio de la entrada que se encuentra en la posición *n* del buffer. El primer carácter, en la posición *n*, nunca puede ser coincidencia, pues la historia está vacía, luego la ventana avanza hacia [1,n]. La ventana ha avanzado, pero las posiciones relativas de las cadenas con respecto al buffer se mantienen durante la codificación del bloque del tamaño de la ventana. El siguiente dibujo esquematiza la organización:



Como sólo están permitidos valores para *desplazamiento* que caigan dentro de la ventana (*n* a lo sumo), a medida que ésta se va desplazando hacia la derecha, los caracteres que quedan a su izquierda no tienen ya valor para el algoritmo. Con esta organización está claro que cualquier posición de la entrada ([n, (2n-1)]) tiene a su izquierda una historia o ventana de al menos *n* caracteres, como el algoritmo requiere. Cuando se ha codificado el primer bloque de tamaño *n*, este es copiado a las primeras *n* posiciones del buffer, con lo que pasan a ser la historia del siguiente bloque de *n* caracteres.

Hasta aquí hemos definido un marco sobre el que trabajar, ya que ahora las cadenas quedan en posiciones fijas con respecto a los límites del buffer durante la codificación de un bloque. El siguiente paso es idear unas estructuras de datos que nos permitan de forma rápida encontrar la cadena más larga en la ventana que coincida con la entrada pendiente de procesar en el *lookahead*.

La idea es construir listas enlazadas para las cadenas de caracteres de la ventana que posean el mismo valor *hash*. Este valor se calculará a partir de los primeros *k* caracteres de cada cadena, donde *k* corresponde a la longitud mínima de una posible coincidencia (una vez más hay que aclarar que después se comentará esta decisión). Así obtenemos una gran probabilidad de que se consiga al menos una coincidencia de longitud mínima (no es seguro ya que es posible que dos cadenas diferentes tengan el mismo *hash*, es decir, las funciones *hash* no son inyectivas en general). El proceso entonces será sencillo: se calculará el *hash* a la cadena inmediatamente siguiente en la entrada y se seguirá la lista enlazada de cadenas con el mismo valor hacia atrás comparando cada cadena con la que está en el punto actual de la entrada. Si se consigue una coincidencia, la cadena coincidente es sustituida a la salida por el par (*desplazamiento, longitud*) correspondiente. Si no, la salida es el literal a la entrada. En ambos casos se añade como cabeza de la lista de cadenas a la cadena actual y se avanza la ventana.

Para conseguir las listas, se usará un array de "cabeceras de listas", una cabecera para cada valor posible de la función *hash*. Los índices de este array serán los posibles valores de la función *hash*, y el valor asociado a cada índice será la posición en el buffer de la última cadena cuyo valor *hash* coincide con el índice. Las listas pueden ser mantenidas en un **único array de longitud n**. Esta es la parte más difícil de la implementación, ya que todas las listas correspondientes a las cadenas que poseen los mismos valores de la función *hash* se introducen en un único array.

Para ver esto un poco más claro, si estudiamos un poco las características de las listas, vemos que cada cadena comienza en una posición, luego una posición del buffer corresponde a una única lista enlazada. En el array de listas, cada índice, por ejemplo el *i*, guarda la posición de la cadena anterior cuyo valor *hash* coincide con la cadena que comienza en la posición *i* del buffer. Para ilustrar todo esto, supongamos que el array de cabeceras se llama *head*, que la cadena siguiente en la entrada está en el puntero *pInPos*, y que el array de listas se llama *prev*. La primera posición para comparar es la dada por la cabeza de la lista para las cadenas con el mismo *hash* que la que comienza en *pInPos*. Después se sigue la lista para obtener la coincidencia de mayor longitud. El siguiente código esquematiza la búsqueda:

```
// Se calcula el hash de la cadena a la entrada
```

```
nHash = funcionHash(pInPos);

// La primera posición para encontrar una coincidencia es la cabeza
// de la lista de las cadenas con hash 'nHash'
comparePos = head[ nHash ];

while (comparePos != NIL && inWindow( comparePos ) )
{
      obtenerLongitudCoincidencia( pInPos , comparePos );

      // Ir al siguiente elemento de la lista para comparar
      // la siguiente cadena
      comparePos = prev[ comparePos mod n ]; (*)
}
```

En cada comparación se guarda la longitud de la coincidencia y la posición para la que se alcanzó. El resultado del código anterior son los valores de la coincidencia más larga y su posición. La condición de parada es que, bien se acabe la lista, o bien la posición a comparar salga de la ventana, con lo que pierde interés (recuérdese que la ventana se va desplazando hacia la derecha y muchas posiciones quedan obsoletas).

Pero, si hay una correspondencia directa entre posiciones del array de listas y posiciones del buffer, ¿cómo es que el primero tiene longitud *n* y el segundo *2n*? La respuesta es que sólo nos interesan *n* posiciones del array de listas (las que caen dentro de una ventana y, por ende, los valores posibles para el *desplazamiento*), por eso, en la línea señalada con asterisco (*), la indexación sobre *prev* se realiza *módulo tamaño de la ventana*. Esto nos hace que un índice *i* en *prev* corresponda a dos posiciones del buffer: *i* e ((*i*+*n*) *módulo 2n*). ¡Pero estas posiciones tienen una característica especial!: distan exactamente *n* caracteres, con lo que están justo en el límite de la ventana y, por lo que respecta al algoritmo, no se solapan.

Las listas se pueden ir construyendo a medida que avanza el algoritmo. Cada cadena en cada posición de la entrada debe ser registrada en el array de listas dependiendo de su valor *hash*. Para conseguir que las listas estén ordenadas por proximidad, la posición actual será la que se inserte en la cabecera de la lista. Si suponemos que nos encontramos en el punto *x* del buffer, la inclusión de la cadena actual, con valor *hash nHash*, al igual que en el ejemplo anterior, es como sigue:

```
prev[ x mod n ] = head[ nHash ];
head[ nHash ] = x;
```

Queda a la imaginación y la pericia del lector el comprobar que estas actualizaciones mantienen en buen estado las listas de cadenas.

Todos estos pasos nos llevarían a una codificación satisfactoria de un bloque de entrada de *n* caracteres. Sin embargo, al terminar el bloque actual, ya se co-

mentó que éste debe pasar a ser ahora la ventana (la parte izquierda del buffer) y otro bloque nuevo debe ocupar la posición de la derecha. Parte de la información en ambos arrays, es decir, en las listas de cadenas, será ahora inválida. En particular, las cadenas que comienzan en la parte izquierda del buffer. Además, ahora si se han movido realmente las cadenas hacia la izquierda. Hay que actualizar, pues, los arrays *head* y *prev*. A cada elemento debemos restar las posiciones que la ventana se ha deslizado, esto es, *n*. Los valores en ambos arrays que queden negativos serán eliminados (convertidos en NIL). Con esto tendremos otra vez listos ambos arrays para comenzar la codificación de otro buffer.

El problema de esta implementación son los grandes espacios de caracteres idénticos. Todas las cadenas que comienzan en cada posición tendrán el mismo *hash* y provocarán que se comparen con la cadena actual, provocando una comparación de cadenas por cada byte del espacio formado por los caracteres idénticos. Esto lo hace muy lento, y es una situación que hay que evitar. Se suele hacer restringiendo el número de comparaciones por cada byte ó parando la búsqueda en el momento en el que la coincidencia más larga sea mayor que un cierto valor.

Otra mejora que se le añade a esta implementación [5] es lo que se denomina "*lazy evaluation matches*" o evaluación perezosa de las coincidencias. Se refiere a que una coincidencia no se produce si el siguiente carácter de la entrada produce una coincidencia que es más larga (al menos dos caracteres más) que la actual.

Existen multitud de implementaciones de la idea LZ77. La mayoría de los compresores más conocidos la utilizan (*ARJ*, *PK-ZIP*, *RAR*, *DoubleSpace*/*DriveSpace*, etc.). Cabe destacar una serie de algoritmos diseñados por Ross Williams [14]. Estos son *LZRW1a* y *LZRW3a*. El primero es una solución parecida a la anterior que sólo utiliza el array *head*. Por ello, sólo dispone de una cadena cuyo *hash* coincidió con la actual. Dada su simplicidad, es muy rápido; y aún así, su efectividad es muy buena. El segundo utiliza un esquema más elaborado, pero en el fondo, una simplificación del visto anteriormente.

No acaban aquí las consideraciones acerca del algoritmo. Dejamos para después en dos ocasiones el tratamiento de la "coincidencia de longitud mínima", y ello fue porque este valor está muy relacionado con otros, como son el tamaño de la ventana y la longitud máxima admitida en una coincidencia. Estos dos últimos parámetros es claro que delimitan el número de bits que deben ser utilizados para codificar los pares (*desplazamiento, longitud*). En primer lugar, y para aprovechar la codificación binaria, el tamaño de la ventana tiene que ser una potencia de dos. Así todos los posibles valores de *desplazamiento* serán posibles y no se desaprovecha ninguno.

Al considerar el tamaño de la ventana, es claro que cuanto más grande sea ésta, mayor será la compresión al ser mayor también la historia sobre la que se buscan posibles coincidencias, con más probabilidad de encontrar una coincidencia más larga. Sin embargo, un tamaño de ventana grande hace que para codificar los valores de *desplazamiento* utilicemos más bits. Además, las coincidencias más cortas (dos o tres caracteres) son las más probables, y una buena codificación debería ahorrar muchos bits en la codificación de los valores de *desplazamiento* y *longitud* de las coincidencias más probables. Esto restringe el tamaño de la ventana a uno tal que haga que *desplazamiento* quepa en pocos bits. Estos dos problemas enfrentados hacen que el tamaño de la ventana sea una consideración importante. Un tamaño típico es el que asigna 12 bits para el *desplazamiento* (ventana de 4096 bytes) y una *longitud* con un máximo de 4 bits (15 bytes de coincidencia máxima; en realidad 18, ¿por qué?), lo que hace un total de 16 bits para el par y una coincidencia mínima de 3 bytes.

Se pueden aplicar, además, codificaciones de longitud variable dependiendo de la longitud de la coincidencia y de su desplazamiento, pero que a su vez requieren bits adicionales. Estas codificaciones son tan importantes que pueden dar una diferencia notable entre dos compresores del mismo tipo. La mayor parte del esfuerzo de diseño se guía hacia este punto en los compresores en tiempo real que después veremos.

El siguiente paso es llevar la codificación de los desplazamientos, longitudes y literales hacia una eficiencia mayor aplicando técnicas estadísticas a la salida. Estas ideas se estudiarán en el siguiente punto.

2.3.4 Compresores híbridos o de dos fases

Esta sección culminará la codificación LZ77 con la introducción de codificaciones adicionales a la salida de este tipo de algoritmos. Vimos que el modo de codificar los datos a la salida (bien pares (*desplazamiento, longitud*), bien *literales*) era de suma importancia. El último paso es llevar esa codificación usando de una forma modificada los algoritmos estadísticos que vimos anteriormente, tanto Huffman como Shannon-Fano como los compresores aritméticos. La operación consiste en llevar tres estadísticas de frecuencias separadas para cada uno de los datos a la salida del algoritmo LZ77: desplazamientos, longitudes y literales. Con estos datos estadísticos, podemos realizar la codificación independientemente para cada uno de estos tipos de valores.

Esto significa, según el tipo de compresor estadístico utilizado, tener un árbol para cada tipo de dato (Huffman ó Shannon-Fano) o una tabla de frecuencias en la compresión aritmética. En [15] podemos ver un pequeño estudio acerca de la conveniencia de utilizar compresores estadísticos a la salida de un com-

presor LZ77, en el que se llega a la conclusión de que el uso de Huffman es casi tan eficiente que el uso de un compresor aritmético aunque mucho más rápido. También se prueban varias alternativas a la hora de construir los códigos para cada tipo de dato, como son árboles Huffman precalculados o considerar sólo los *k* bits altos del desplazamiento (téngase en cuenta que, por un lado, hay muchos valores posibles para el desplazamiento y el árbol sería muy grande, y, por otro, es obvio que la variabilidad de cada desplazamiento es casi imprevisible. El considerar sólo los *k* bits altos reduce la variabilidad y la magnitud del árbol) y los restantes *verbatim*.

Otra alternativa es utilizar árboles de códigos adaptativos, como los "*splay trees*" [16].

Ejemplos de compresores famosos de este estilo son:

- **ARJ, LHA:** LZ77 + Huffman estático a bloques
- **ZIP 1.x:** LZ77 + Shannon-Fano (Imploding), ≈LZW (Shrinking)
- **ZIP 2.x** y **gzip:** LZ77 + Huffman dinámico ó Huffman estático a bloques (Deflate)
- **HA:** PPMC (Modelo de Markov de 4º orden, es decir, compresión aritmética)
- **UC2:** LZ77 + Huffman estático + lo que ellos llaman "*shared dictionaries among files*". Lo que significa sólo lo saben allí en *AIP Development*.

2.4 Compresores en tiempo real

Con el aumento de velocidad de los ordenadores la compresión se está aplicando cada vez más a entornos en tiempo real o de compresión transparente. En esta sección comentaremos las características y usos de este tipo de compresión.

En cuanto a los usos, tenemos los siguientes:

- compresión de disco en tiempo real: **Stacker**, **SuperStor, DoubleSpace / DriveSpace**, sistema de ficheros de Windows NT, **NTFS,**
- compresión transparente en comunicaciones (no muy aplicada, por cierto, ni en IPv6),
- soporte de .ZIP y .GZ directo en Unix,
- tratamiento, en algunos sistemas operativos, de los archivos comprimidos como directorios,

- sistemas de ayuda comprimidos como los **.HLP** de Windows o los **Portable Document Format (PDF)** de Adobe Systems,
- formatos de vídeo comprimidos (de forma *lossless*) para juegos, presentaciones, etc.

En cuanto a las características, se dirá que la compresión y descompresión debe ser muy rápida y con unas necesidades de memoria reducidas. Por eso aquí también los más utilizados son variantes de LZ77, como *LZRW1a* comentado anteriormente. Una característica importante de este tipo de algoritmos es que la descompresión es tanto más rápida cuanto más se haya comprimido el original.

2.5 El futuro de la compresión de datos

Una vez más: es difícil predecir. No obstante, al ir aumentando la velocidad de los ordenadores, algoritmos que antes no se podían considerar de tiempo real pueden ser utilizados en estos entornos, dando una media superior de compresión. Los compresores basados en diccionario ya están bastante cerca de la cota de Shannon, pero son superados por los aritméticos, más lentos.

Sin embargo, el interés actual se desvía a los compresores *lossy* dado el auge que las técnicas multimedia han experimentado. Con la venida de la televisión digital, por ejemplo, la búsqueda de algoritmos de compresión de imágenes y sonido pasa a primer plano. En el siguiente punto se tratarán los compresores *lossy*. Ejemplos de los avances en este tipo de compresores son, por ejemplo, el estándar MPEG-2, usado actualmente por las compañías de televisión en sus enlaces por satélite, y lo que se denomina *MPEG layer 3*, utilizado en compresión de sonido de calidad CD, con una reducción normal de 8 a 1.

3 Compresión *lossy*

3.1 Introducción

Es obvio que transmitir material multimedia sin comprimir es impensable. La única esperanza es que la compresión masiva sea posible. Afortunadamente,

un gran esfuerzo de investigación durante las pasadas décadas ha llevado a muchas técnicas y algoritmos que hacen la compresión multimedia posible.

Todos los sistemas de compresión requieren dos algoritmos, uno para comprimir los datos en el origen, y otro para descomprimirlos en el destino. En la literatura, estos algoritmos son referidos como algoritmos de codificación y decodificación, respectivamente.

Estos algoritmos tienen ciertas asimetrías que son importantes de comprender. Primeramente, para muchas aplicaciones, un documento multimedia, por ejemplo, una película, será codificada sólo una vez (cuando es guardada en el servidor multimedia) pero será decodificada miles de veces (cuando es vista por los clientes). Esta asimetría significa que es aceptable para el algoritmo de codificación el ser lento y requerir hardware caro siempre que el algoritmo de decodificación sea rápido y no requiera hardware caro. Después de todo, el operador de un servidor multimedia puede alquilar un superordenador paralelo por un par de semanas para codificar su biblioteca de vídeo entera, pero hacer que los consumidores alquilen un superordenador por dos horas para ver un vídeo no es muy buena idea. Muchos sistemas de compresión llegan a hacer la decodificación rápida y simple, incluso al precio de hacer la codificación lenta y complicada.

Por otro lado, para multimedia en tiempo real, como videoconferencia, la codificación lenta es inaceptable. La codificación debe ocurrir al instante, en tiempo real. Consecuentemente, en multimedia en tiempo real se usan diferentes algoritmos o parámetros que al almacenar vídeos en disco, a menudo con apreciablemente menor compresión.

Una segunda asimetría es que el proceso de codificación/decodificación no necesita ser reversible. Esto es, cuando se comprime un fichero, se transmite, y cuando de descomprime, el usuario espera conseguir el original, igual hasta el último bit. En multimedia, este requerimiento no existe. Es usualmente aceptable que la señal de vídeo sea ligeramente diferente de la original. Cuando la salida decodificada no es exactamente igual a la entrada original, el sistema se llama “con pérdidas” (lossy). Si la entrada y la salida son idénticas, el sistema es “sin pérdidas” (lossless). Los sistemas con pérdidas son importantes porque aceptar una pequeña cantidad de pérdida de información puede dar una enorme ganancia en términos de posibilidad de ratio de compresión.

3.1.1 Codificación por entropía

Los esquemas de compresión pueden ser divididos en dos categorías generales: codificación por entropía y codificación de la fuente.

La codificación por entropía sólo manipula las cadenas de bits sin saber lo que significan los bits. Es una técnica totalmente reversible, sin pérdidas y general, aplicable a todos los datos. Se dan tres ejemplos

El primer ejemplo de codificación por entropía es RLE (Run Length Encoding). En muchos tipos de datos, son comunes las cadenas de símbolos repetidos (bits, números, etc.). Éstos pueden ser reemplazados por un marcador especial no permitido de otra forma en los datos, seguido por un símbolo que indica la repetición, y seguido por cuántas veces ha ocurrido. Si el mismo marcador especial sale en los datos, se duplica (como en relleno de carácter (character stuffing)). Por ejemplo, si se considera la siguiente cadena de dígitos decimales:

3150000000000008458711111111111163546740000000000000000000065

Si ahora se introduce A como el marcador y se usan números de dos dígitos para la cuenta de repeticiones, se puede codificar la cadena de dígitos de arriba como:

315A01284587A11316354674A02265

Aquí el RLE ha acortado la cadena de datos a la mitad.

Las repeticiones son comunes en multimedia. En audio, el silencio es a menudo representado por repeticiones de ceros. En vídeo, las repeticiones del mismo color ocurren en enfoques al cielo, muros, y muchas otras superficies planas. Todas estas repeticiones se pueden comprimir mucho.

El segundo ejemplo de codificación por entropía es la codificación estadística. Por esto se entiende usar un código corto para representar símbolos comunes y largo para representar los infrecuentes. El código Morse usa este principio, siendo la E · y la Q --·- y así sucesivamente.

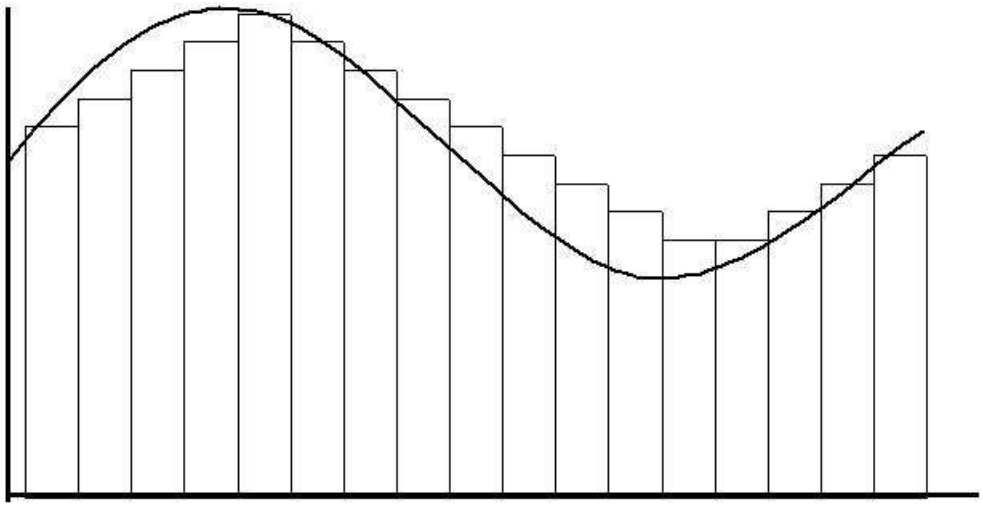
El tercer ejemplo de la codificación por entropía es codificación CLUT (Color Look Up Table). Considerar una imagen usando codificación RGB con 3 bytes por pixel. En teoría, la imagen puede contener hasta 2^{24} diferentes valores de color. En la práctica contendrá normalmente muchos menos valores, especialmente si la imagen es un dibujo animado o generado por ordenador, más que una fotografía. Se supone que sólo 256 valores de color son usados actualmente. Un factor de compresión de casi tres puede ser alcanzado construyendo una tabla de 768 bytes listando los valores RGB de los 256 colores usados actualmente, y después representando cada pixel por el índice de su valor RGB en la tabla. Aquí se ve un ejemplo claro donde la codificación es más lenta que la decodificación porque codificar requiere buscar en la tabla mientras que se puede decodificar con una sola operación de indexación.

3.1.2 Codificación de la fuente

La codificación de la fuente se beneficia de las propiedades de los datos para producir más (usualmente con perdidas) compresión. Aquí también se ilustra la idea con tres ejemplos. El primer ejemplo es la codificación diferencial, en la cual una secuencia de valores (p.e., samples de audio) son codificados representando cada uno de los valores como la diferencia respecto a un valor previo. La modulación diferencial por pulsos (Differential Pulse Code Modulation) es un ejemplo de esta técnica. Es con perdidas porque la señal puede saltar tanto entre dos valores consecutivos que la diferencia no cabe en el campo proporcionado para expresar las diferencias, así que al menos un valor incorrecto será grabado y alguna información se perderá.

La codificación diferencial es un tipo de codificación de la fuente porque se aprovecha la propiedad de que saltos grandes entre datos consecutivos son improbables. No todas las secuencias de números tienen esta propiedad. Un ejemplo de carencia de esta propiedad es una lista generada por ordenador de números de teléfono aleatorios usada por los publicistas para molestar a la gente durante la cena. Las diferencias entre números de teléfono consecutivos en la lista usará tantos bits como para representar los números de teléfono.

Un caso particular de la codificación diferencial es la Modulación Delta, en la que se requiere que cada valor sampleado difiera de su predecesor en +1 ó −1:



El segundo ejemplo de codificación de la fuente consiste en transformaciones. Transformando las señales de un dominio a otro, la compresión puede hacerse mucho más fácil. Considérese, por ejemplo, una transformada de Fourier. Una función del tiempo es representada como una lista de amplitudes. Dados los valores exactos de todas las amplitudes, la función original puede ser

reconstruida perfectamente. De cualquier modo, dando sólo los valores de las primeras, por ejemplo, ocho amplitudes redondeadas a dos decimales, puede todavía ser posible reconstruir la señal de forma que el que escucha no pueda decir si se ha perdido alguna información. La ganancia es que transmitir ocho amplitudes requiere muchos menos bits que transmitir la onda sampleada.

Las transformaciones son también aplicables a datos de imágenes bidimensionales. Suponer que la matriz de 4×4 de la figura representa los valores de escala de grises de una imagen monocroma.. Se puede transformar esos datos restando el valor de la esquina superior izquierda a todos los elementos excepto a él mismo, como en la otra figura. Esta transformación podría ser útil si se usa codificación de longitud variable. Por ejemplo, los valores entre –7 y +7 podrían ser codificados con números de 4 bits y los valores entre 0 y 255 podrían ser codificados con un código especial de 4 bits (–8) seguido por un número de 8 bits.

160	160	161	160
161	165	166	156
160	167	165	161
159	160	160	160

160	0	1	0
1	5	6	-2
0	7	5	1
-1	0	1	0

Aunque esta simple transformación es sin pérdidas, otras, más útiles, no. Una transformación espacial bidimensional especialmente importante es la DCT (Discrete Cosine Transformation). Esta transformación tiene la propiedad de que para imágenes sin discontinuidades muy pronunciadas, casi todo el espectro de potencia está en los primeros términos, permitiendo ignorar los últimos sin mucha pérdida de información. Se volverá al DCT dentro de poco.

El tercer ejemplo de codificación de la fuente es la cuantización de vectores (*vector quantization*), que es también directamente aplicable a datos de imágenes. Aquí la imagen se divide en rectángulos de tamaño fijo. Además de la imagen misma, también se necesita una tabla de rectángulos del mismo tamaño que los rectángulos de la imagen (posiblemente construidos a partir de la imagen). Esta tabla se llama libro de códigos (*code book*). Cada rectángulo es transmitido buscándolo en el libro de códigos y mandando el índice en vez del rectángulo. Si el libro de códigos es creado dinámicamente (a partir de la ima-

gen), debe ser transmitido también. Claramente, si un pequeño número de rectángulos domina la imagen, grandes ahorros en ancho de banda serán posibles aquí.

Un ejemplo de cuantización de vectores se enseña en las siguientes figuras. En la primera se tiene una rejilla de rectángulos de tamaño sin especificar. En la segunda se tiene el libro de códigos. La salida es sólo la lista de enteros 0010220322004 00 de la tercera figura. Cada uno representa una entrada del libro de códigos.

0	0	1	0
2	2	2	0
3	2	2	0
0	4	0	0

Imagen dividida en rectángulos

0	1	2	3	4

*Libro de códigos (**code book**) para la imagen*

0 0 1 0 2 2 0 3 2 2 0 0 4 0 0

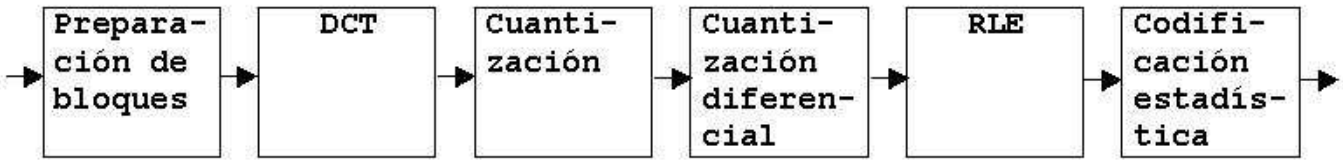
Imagen codificada

Se podría pensar que la cuantización de vectores es sólo una generalización bidimensional de CLUT. Pero la diferencia real es lo que pasa si no se encuentran coincidencias. Tres estrategias son posibles. La primera es sólo usar la mejor coincidencia. La segunda es usar la mejor coincidencia, y añadir alguna información sobre cómo mejorar la coincidencia (p.e. añadir el valor medio). El tercero es usar la mejor coincidencia y añadir todo aquello que sea necesario para permitir al decodificador reconstruir los datos exactamente. Las primeras estrategias son con pérdidas pero se alcanza mucha compresión. La tercera es sin pérdidas pero menos efectiva como algoritmo de compresión. Otra vez, se ve que codificar (coincidencia de patrones) consume mucho más que decodificar (indexar en una tabla).

3.2 El estándar JPEG

El estándar JPEG (Joint Photographic Experts Group) para comprimir imágenes de tonos continuos (p.e. fotografías) fue desarrollado por expertos fotográficos trabajando juntos bajo los auspicios de ITU, ISO e IEC. Es importante para multimedia porque, en una primera aproximación, el estándar multimedia para imágenes en movimiento, MPEG, es sólo la codificación JPEG de cada fotograma separadamente, además de algunas características extra para compresión entre fotogramas y detección de movimiento. JPEG está definido en el estándar internacional 10918.

JPEG tiene cuatro modos y muchas opciones. Es más como una lista de la compra que un sólo algoritmo. Para nuestros propósitos, sin embargo, sólo el modo secuencial con pérdidas es relevante, y es el ilustrado en la figura. Más adelante, se explicará la forma en la que el JPEG es normalmente usado para codificar imágenes de vídeo RGB de 24 bits y se dejarán algunos de los detalles por simplicidad.



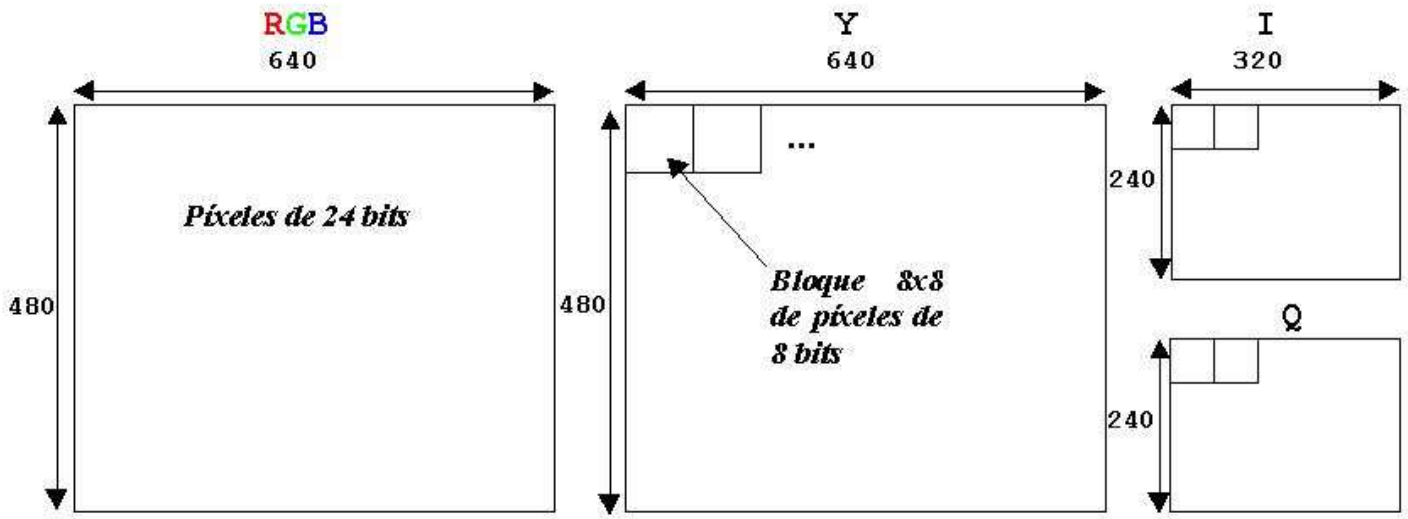
El paso 1 de la codificación de una imagen con JPEG es la preparación de bloques. Por especificidad, se asumirá que la entrada JPEG es una imagen RGB con 24 bit por píxel, como se ve en la figura. Usando la luminancia (brillo) y la crominancia (color) se consigue mejor compresión. Estos valores son otra forma de expresar los valores RGB de una imagen. El ojo es mucho más sensi-

ble a la señal de luminancia que a las señales de crominancia, así que no hace falta tanta precisión en estas últimas. Esto también permite compatibilizar las transmisiones de televisión para receptores en blanco y negro y color. Las dos señales de crominancia se transmiten en estrechas bandas a más altas frecuencias. Volviendo a la compresión, primero se computa la luminancia, Y, y las dos crominancias, I y Q (para NTSC), de acuerdo a las siguientes fórmulas:

$$Y = 0.3R + 0.59G + 0.11B$$
$$I = 0.60R - 0.28G - 0.32B$$
$$Q = 0.21R - 0.52G + 0.31B$$

Para PAL, las crominancias se llaman U y V y los coeficientes son diferentes, pero la idea es la misma. SECAM es diferente a NTSC y PAL.

Se construyen matrices separadas para la Y, I y Q, cada una con elementos en el rango de 0 a 255. Después, se hace la media de bloques cuadrados de cuatro píxeles en las matrices I y Q para reducirlas a 320 × 240. Esta reducción es con pérdidas, pero el ojo casi no lo nota, ya que el ojo responde más a la luminancia que a la crominancia. De cualquier forma, esto comprime los datos por un factor de dos. Ahora se le resta 128 a cada elemento de las tres matrices para poner un 0 en medio del rango. Finalmente, cada matriz se divide en bloques de 8 × 8. La matriz Y tiene 4800 bloques; las otras dos tienen 1200 bloques cada una, como se observa en la figura..



El paso 2 de JPEG es aplicar una transformación discreta de cosenos a cada uno de los 7200 bloques separadamente. La salida de cada DCT es una matriz 8 × 8 de coeficientes DCT. El elemento DCT (0,0) es la media del blo-

que. Los otros elementos dicen cuánta potencia espectral está presente en cada frecuencia espacial. En teoría un DCT es sin pérdidas, pero en la práctica, usando números en coma flotante siempre se introduce algo de error de redondeo que resulta en una pérdida de información. Normalmente, estos elementos decaen rápidamente con la distancia al origen, (0,0), como se ve en la figura.

150	80	40	14	4	2	1	0
92	75	36	10	6	1	0	0
52	38	26	8	7	4	0	0
12	8	6	4	2	1	0	0
4	3	2	0	0	0	0	0
2	2	2	1	1	0	0	0
1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Coeficientes DCT

Una vez completado el DCT, llega el paso 3 del JPEG, llamado cuantización, en el cual los coeficientes DCT menos importantes se quitan. Esta transformación (con pérdidas) se hace dividiendo cada coeficiente en la matriz DCT 8×8 por un peso cogido de una tabla. Si todos los pesos son 1, la transformación no hace nada. Pero si los pesos se incrementan rápidamente desde el origen, las frecuencias espaciales más altas son se desechan enseguida.

Un ejemplo de este paso se da en la figura siguiente. Aquí se ve la tabla de cuantización, y el resultado obtenido dividiendo cada elemento DCT de la figura anterior por el correspondiente elemento de la tabla de cuantización. Los valores de la tabla de cuantización no son parte del estándar JPEG. Cada aplicación debe tener su tabla, permitiendo controlar la compresión con pérdidas.

150	80	20	4	1	0	0	0
92	75	18	3	1	0	0	0
26	19	13	2	1	0	0	0
3	2	2	1	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

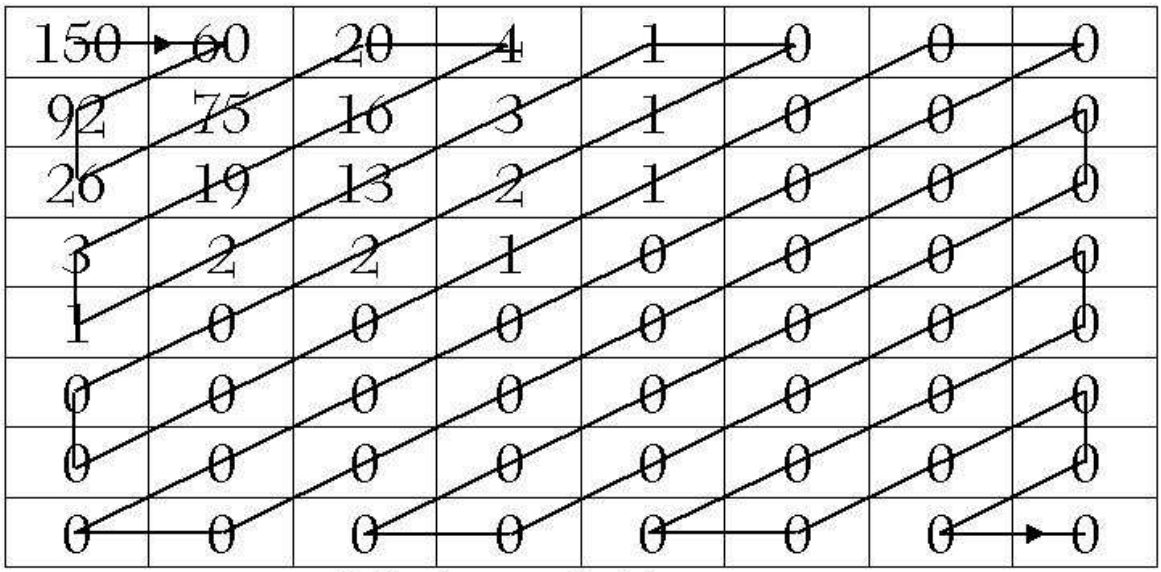
Coeficientes cuantizados

1	1	2	4	8	16	32	64
1	1	2	4	8	16	32	64
2	2	2	4	8	16	32	64
4	4	4	4	8	16	32	64
8	8	8	8	8	16	32	64
16	16	16	16	16	16	32	64
32	32	32	32	32	32	32	64
64	64	64	64	64	64	64	64

Tabla de cuantización

El paso 4 reduce el valor (0,0) de cada bloque (el de la esquina superior izquierda) reemplazándolo por la cantidad en que difiere del elemento correspondiente en el bloque previo. Como estos elementos son la media de sus respectivos bloques, deberían cambiar lentamente, así que tomar los valores diferenciales debería reducir la mayoría de ellos a valores pequeños. No se computan diferenciales de los otros valores. Los valores (0,0) son referidos como las componentes DC; los otros valores son los componentes AC.

El paso 5 lineariza los 64 elementos y aplica RLE a la lista. Recorrer el bloque de izquierda a derecha y de arriba abajo no concentrará los ceros juntos, así que se usa un patrón de recorrido en zig-zag, mostrado en la figura. En este ejemplo, el patrón en zig-zag produce 38 ceros consecutivos al final de la matriz. Esta cadena puede ser reducida a una única cuenta diciendo que hay 38 ceros.



Orden de recorrido de la matriz

150 60 92 26 75 20 4 16 19 3 1 2 13 3 1 0 1 2 2 A005 1 1 A038

Cadena resultante

Ahora se tiene una lista de números que representan la imagen (en el espacio transformado). El paso 6 codifica por Huffman los números para almacenamiento o transmisión.

El JPEG puede parecer complicado, pero es porque es complicado. Aunque como ofrece una compresión de 20:1 o mejor, es ampliamente usado. Decodificar una imagen JPEG requiere ejecutar el algoritmo al revés. A diferencia de otros algoritmos vistos, JPEG es muy simétrico: decodificar lleva tanto como codificar.

Es bastante interesante que debido a las propiedades matemáticas de la DCT, es posible llevar a cabo ciertas transformaciones geométricas (p.e. rotación de imagen) directamente en la matriz transformada, sin regenerar la imagen original. Propiedades similares también se aplican al audio MPEG comprimido.

3.3 El estándar MPEG

Finalmente, llegamos al corazón del asunto; los estándares MPEG (Motion Picture Experts Group). Éstos son los algoritmos más importantes para comprimir vídeos y son un estándar internacional desde 1993. Como las películas contienen imágenes y sonido, MPEG puede comprimir audio y vídeo, pe-

ro como el vídeo necesita más ancho de banda y también contiene más redundancia que al audio, nos centraremos en la compresión de vídeo MPEG.

El primer estándar que se finalizó fue MPEG-1 (estándar internacional 11172). Destacaba por producir una salida con calidad de grabador de vídeo (352 × 240 para NTSC) usando un bit rate de 1.2 Mbps. Como el vídeo sin comprimir puede necesitar hasta 472 Mbps, dejarlo en 1.2 Mbps no es enteramente trivial, incluso a esta baja resolución. MPEG-1 puede ser transmitido por líneas de transmisión de par trenzado para pequeñas distancias. MPEG-1 también se usa para guardar películas en formato CD-ROM y CD-Vídeo.

El siguiente estándar en la familia MPEG fue MPEG-2 (estándar internacional 13818), que fue originalmente diseñado para comprimir vídeo con calidad de difusión en 4 a 6 Mbps, para que pudiera caber en un canal de difusión NTSC o PAL. Más tarde, MPEG fue expandido para soportar más altas resoluciones, incluyendo HDTV.

MPEG-4 es para videoconferencia de media resolución con bajos frame rates (10 frames/s) y en bajos anchos de banda (64 kbps). Esto permite mantener videoconferencias en un solo canal N-RDSI B. Con esta numeración, se podría pensar que el siguiente estándar será MPEG-8. Actualmente ISO los está numerando linealmente, no exponencialmente. Originalmente existió MPEG-3. Estaba destinado a HDTV, pero se canceló el proyecto y se incluyó HDTV a MPEG-2.

Estándar	Calidad	Bit rate	Uso
MPEG-1	VCR	1.2 Mbps	Par trenzado, CD-ROM, CD-Vídeo
MPEG-2	NTSC, PAL, HDTV	4-6 Mbps	Difusión, alta definición
MPEG-3	Proyecto cancelado, incluido en MPEG-2		
MPEG-4	Baja resolución	64 kbps	Videoconferencia en N-RDSI B

Los principios básicos de MPEG-1 y MPEG-2 son similares, pero los detalles son diferentes. En una primera aproximación, MPEG-2 es un superconjunto de MPEG-1, con características adicionales, formatos de fotograma, y opciones de codificación. Parece que MPEG-1 dominará en las películas en CD-

ROM y MPEG-2 en transmisiones de vídeo de larga distancia. Se discutirá primero MPEG-1 y después MPEG-2.

MPEG-1 tiene tres partes: audio, vídeo y sistema, que integra las otras dos. Los codificadores de audio y de vídeo trabajan independientemente, lo que incrementa el problema de cómo las dos fuentes se sincronizan en el receptor. Este problema se soluciona teniendo un reloj de sistema de 90 KHz que suministra el valor actual de tiempo a ambos codificadores. Estos valores son de 33 bits, para permitir a las películas correr durante 24 horas. Las marcas de tiempo se incluyen en la salida codificada y se propagan todo en camino hasta en receptor, que los usa para sincronizar las fuentes de audio y vídeo.

La compresión de audio MPEG se hace sampleando la onda a 32 kHz, 44.1 kHz ó 48 kHz. Puede manejar mono, estéreo disjunto (cada canal comprimido separadamente), o estéreo junto (se explota la redundancia intercanal). Se organiza como tres capas, cada una de ellas aplicando optimizaciones adicionales para conseguir más compresión (y a mayor costo). La capa 1 es el esquema básico. Esta capa se usa, por ejemplo, en el sistema de cinta digital DCC. La capa 2 añade al esquema básico localización de bits adicional. Se usa para audio en CD-ROM y bandas sonoras de películas. La capa 3 añade filtros híbridos, cuantización no uniforme, codificación Huffman y otras técnicas avanzadas.

El audio MPEG puede comprimir un CD de rock 'n roll en 96 kbps sin pérdida de calidad perceptible, incluso para los fans del rock 'n roll sin pérdida de oído. Para un concierto de piano, se necesitan al menos 128 kbps. Esta diferencia es porque la relación señal-ruido del rock 'n roll es mucho más alta que la del concierto de piano (en el sentido de la ingeniería).

La compresión de audio se lleva a cabo haciendo una transformada rápida de Fourier en la señal de audio para transformarla del dominio del tiempo al dominio de la frecuencia. El espectro resultante se divide en 32 bandas de frecuencia, y cada una es procesada separadamente. Cuando hay presentes dos canales estéreo, la redundancia inherente de tener dos fuentes de audio altamente superpuestas también se explota. El audio MPEG-1 resultante es ajustable desde 32 kbps hasta 448 kbps.

Ahora se abordará la compresión de vídeo MPEG-1. Existen dos tipos de redundancia en las películas: espacial y temporal. MPEG-1 usa los dos. La redundancia espacial puede ser utilizada simplemente codificando cada fotograma separadamente con JPEG. Esta aproximación se usa a veces, especialmente cuando se necesita acceso aleatorio a cada fotograma, como en edición de producciones de vídeo. De este modo, es alcanzable un ancho de banda comprimido de 8 a 10 Mbps.

Se puede alcanzar compresión adicional aprovechándose del hecho de que dos fotogramas consecutivos son a menudo casi idénticos. Este efecto es más pequeño de lo que podría parecer al principio, ya que muchos realizadores cortan entre escenas cada 3 ó 4 segundos (cronometrar una película y contar las escenas). De cualquier forma, incluso una secuencia de 75 fotogramas al-

tamente similares ofrece el potencial de una gran reducción superior a simplemente codificar cada fotograma separadamente con JPEG.

Para escenas donde la cámara y el fondo son estacionarios y uno o dos actores se mueven lentamente, casi todos los píxeles serán idénticos de fotograma a fotograma. Aquí, restar cada fotograma al anterior y aplicar JPEG a la diferencia estaría bien. De cualquier forma, para escenas donde la cámara hace zoom o se mueve, esta técnica falla estrepitosamente. Lo que se necesita es alguna forma de compensar este movimiento. Esto es precisamente lo que hace MPEG; esta es la mayor diferencia entre JPEG y MPEG.

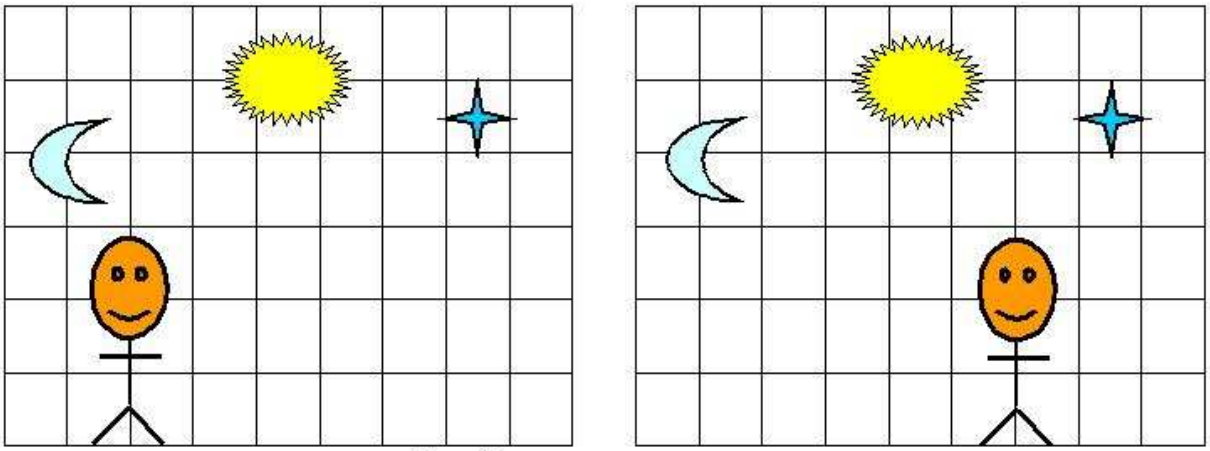
La salida MPEG-1 consiste en cuatro tipos de fotogramas:

1. Fotogramas I (Intracoded): Imágenes estáticas codificadas con JPEG y autocontenidas.
2. Fotogramas P (Predictive): Diferencia bloque a bloque con el último fotograma.
3. Fotogramas B (Bidirectional): Diferencias con el último y con el siguiente fotograma.
4. Fotogramas D (DC-coded): Medias de bloques usadas para avance rápido.

Los fotogramas I son sólo imágenes estáticas codificadas usando JPEG, también usando luminancia de resolución plena y crominancia de resolución media sobre cada eje. Es necesario que aparezcan fotogramas I periódicamente por tres razones. Primero, MPEG-1 puede ser usado para una transmisión multicast, con espectadores conectándose cuando quieran. Si todos los fotogramas dependiesen de sus predecesores hasta el primer fotograma, cualquiera que perdiera el primer fotograma nunca podría decodificar los fotogramas subsecuentes. Segundo, si se recibiera un fotograma con error, no se podría seguir decodificando. Tercero, sin los fotogramas I, mientras se hace un avance o retroceso rápido, el decodificador tendría que calcular cada fotograma sobre el que se ha pasado para saber el valor íntegro del fotograma sobre el cual se ha parado. Por estas razones, los fotogramas I se insertan en la salida una o dos veces por segundo.

Los fotogramas P, en contraste, codifican diferencias interfotograma. Se basan en la idea de los macrobloques, que cubren 16×16 píxeles en el espacio de la luminancia y 8×8 en el espacio de la crominancia. Un macrobloque se codifica buscando su fotograma previo o algo sólo ligeramente diferente de ello.

Un ejemplo donde los fotogramas P serían útiles se da en la figura. Aquí se ven dos fotogramas consecutivos que tienen el mismo fondo, pero que difieren en la posición de un muñeco. Los macrobloques que contienen la escena de fondo concordarán exactamente, pero los macrobloques que contienen al muñeco tendrán una diferencia de posición y habrá que darles un seguimiento.



Dos fotogramas consecutivos

El estándar MPEG-1 no especifica cómo buscar, cómo de lejos buscar, o cómo de buena debe ser una coincidencia para contar. Esto se deja para cada implementación. Por ejemplo, una implementación puede buscar un macrobloque en la posición actual en el fotograma anterior y en todas las demás posiciones con un offset $\pm\Delta x$ en la dirección x y $\pm\Delta y$ en la dirección y. Para cada posición, el número de coincidencias en la matriz de luminancias podría ser computado. La posición con la más alta puntuación sería declarada ganadora, habiendo definido previamente un umbral. De otro modo, se diría que el macrobloque se ha perdido. Por supuesto, también son posibles algoritmos mucho más sofisticados.

Si se encuentra un macrobloque, se codifica cogiendo la diferencia con su valor en el fotograma anterior (para la luminancia y las dos crominancias). Estas matrices de diferencias son entonces objeto de una DCT, cuantización, codificación RLE y codificación Huffman, como en JPEG. El valor del macrobloque en la salida es el vector de movimiento (cuánto se ha movido el macrobloque en cada dirección), seguido por la lista de números codificada con Huffman. Si el macrobloque no se localiza en el fotograma anterior, el valor actual se codifica con JPEG, como en un fotograma I.

Claramente, el algoritmo es altamente asimétrico. Una implementación es libre para probar cada posición plausible en el fotograma anterior si quiere, en un intento desesperado de localizar cada último macrobloque. Esta aproximación minimizará la salida MPEG-1 a expensas de una codificación muy lenta. Podría ser buena para la codificación una única vez de una biblioteca de películas pero sería terrible para videoconferencia en tiempo real.

Similarmente, cada implementación es libre para decidir lo que constituye un macrobloque “encontrado”. Esta libertad permite a los implementadores competir en la calidad y velocidad de sus algoritmos, pero produciendo

siempre MPEG-1 estándar. No importa qué algoritmo de búsqueda se use, la salida final es la codificación JPEG del macrobloque actual o la codificación JPEG de la diferencia entre el macrobloque actual y uno en el fotograma anterior en un offset especificado desde el actual.

Decodificar MPEG-1 es siempre hacia adelante. Decodificar fotogramas I es lo mismo que decodificar imágenes JPEG. Decodificar fotogramas P requiere que el decodificador tenga en un buffer el fotograma anterior y entonces construir el nuevo en segundo buffer basado en macrobloques totalmente codificados y macrobloques que contienen las diferencias con el fotograma anterior. El nuevo fotograma se ensambla macrobloque a macrobloque.

Los fotogramas B son similares a los fotogramas P, excepto que permiten que el macrobloque de referencia esté en un fotograma previo o en uno posterior. Esta libertad adicional permite compensación de movimiento adicional, y también es útil cuando pasan objetos delante o detrás de otros objetos. Para codificar fotogramas B, el codificador necesita tener tres fotogramas decodificados en memoria a la vez: el anterior, el actual y el siguiente. Aunque los fotogramas B dan la mejor compresión, no los soportan todas las implementaciones.

Los fotogramas D se usan sólo para hacer posible mostrar una imagen de baja resolución cuando se rebobina o se avanza rápido. Hacer la decodificación MPEG-1 en tiempo real es suficientemente complicado. Esperar que el decodificador lo haga diez veces más rápido es demasiado. En vez de eso, los fotogramas D se usan para producir imágenes de baja resolución. Cada entrada de fotograma D es el valor medio de un bloque, sin más codificación, haciendo fácil mostrarlo en tiempo real. Esta facilidad es importante para permitir que la gente ojee un vídeo a alta velocidad buscando una escena en particular.

Habiendo ya tratado el MPEG-1, se seguirá con MPEG-2. La codificación es fundamentalmente similar a la codificación MPEG-1, con fotogramas I, P y B. Los fotogramas D no se soportan. También, la transformación discreta de cosenos es de 10×10 en lugar de 8×8, para dar un 50 por ciento más de coeficientes, para mejor calidad. Como el MPEG-2 se pensó para difusión de televisión y para aplicaciones CD-ROM, soporta ambas imágenes, las progresivas y las entrelazadas, mientras que MPEG-1 sólo soporta progresivas. También hay otras diferencias en detalles entre los dos estándares.

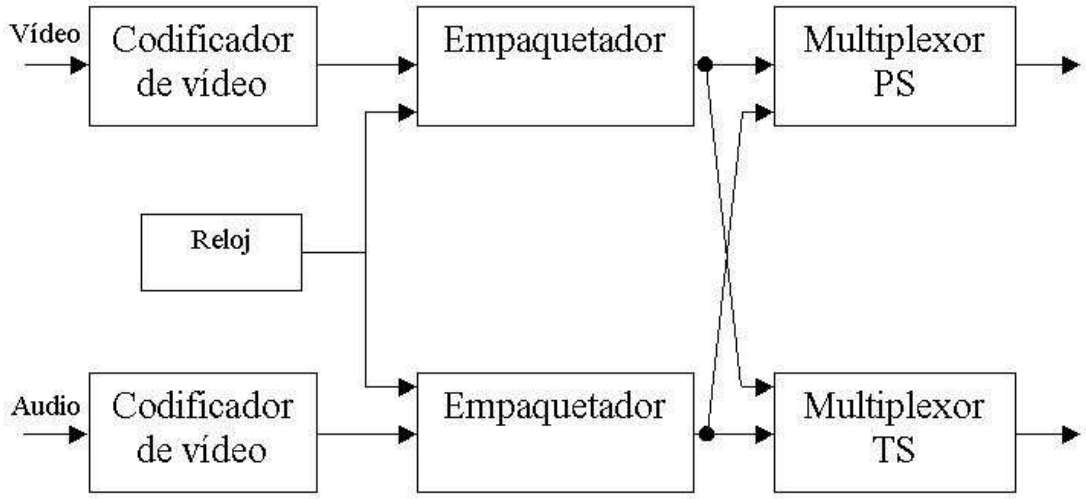
En lugar de soportar un sólo nivel de resolución, MPEG-2 soporta cuatro: baja (352×240), media (main) (720×480), alta-1440 (1440×1152), y alta (1920×1080). La baja resolución es para VCRs y compatibilidad hacia atrás con MPEG-1. La media es la normal para difusión NTSC. Las otras dos son para HDTV.

Además de tener cuatro niveles de resolución, MPEG-2 también soporta cinco perfiles. Cada perfil está dedicado a un área de aplicación. El perfil más importante (main) es para uso de propósito general, y probablemente la mayoría de los chips estarán optimizados para el perfil main y el nivel de resolución main. El perfil simple es similar al main, excepto que excluye el uso de fotogramas B, para hacer el software de codificación y decodificación más fácil. Los

demás perfiles tratan la escalabilidad y HDTV. Los perfiles difieren en términos de la presencia o ausencia de fotogramas B, resolución de crominancia y escalabilidad de la cadena de bits a otros formatos.

El ratio de datos comprimidos para cada combinación de resolución y perfil es diferente. Éstos varían desde más o menos 3 Mbps hasta 100 Mbps para HDTV. El caso normal es más o menos 3 ó 4 Mbps.

MPEG-2 tiene una forma más general de multiplexar audio y vídeo que MPEG-1. Define un número ilimitado de cadenas elementales, incluyendo vídeo y audio, pero también incluyendo cadenas de datos que deben ser sincronizados con el audio y el vídeo, por ejemplo, subtítulos en múltiples idiomas. Cada una de las cadenas se empaqueta con marcas de tiempo. Un simple ejemplo de dos cadenas se enseña en la figura.



La salida de cada empaquetador es un PES (Packetized Elementary Stream). Cada paquete PES tiene unos 30 campos y flags, incluyendo longitudes, identificadores de cadena, control de encriptación, estatus de copyright, marcas de tiempo, y un CRC.

Las cadenas de PES para audio, vídeo y posiblemente datos son entonces multiplexados juntos en una sola cadena de salida para transmisión. Se definen dos tipos de cadenas. La cadena de programa de MPEG-2 es similar a la cadena de sistema de MPEG-1 de hace tres figuras. Se usa para multiplexar juntas cadenas elementales que tienen una base de tiempos común y tienen que ser mostrados sincronizados. La cadena de programa usa largos paquetes de longitud variable.

La otra cadena MPEG-2 es la cadena de transporte. Se usa para multiplexar cadenas juntas (incluyendo cadenas de programa) que no usan una base de tiempos común. Los paquetes de la cadena de transporte son de longitud

fija (188 bytes), para hacer más fácil el limitar el efecto de los paquetes dañados o perdidos durante la transmisión.

Todos los esquemas de codificación que se han discutido están basados en el modelo de codificación con pérdidas seguida de transmisión sin pérdidas. Ni el JPEG ni el MPEG, por ejemplo, pueden recuperarse bien de la pérdida o daño de paquetes. Una aproximación diferente a la transmisión de imágenes es transformar las imágenes de forma que se separe la información importante de la menos importante (como hace DCT, por ejemplo). Entonces añadir una considerable cantidad de redundancia (incluso duplicar paquetes) a la información importante y nada a la menos importante. Si algunos paquetes se pierden o dañan, puede ser todavía posible mostrar imágenes razonables sin retransmisión. Estas ideas son especialmente aplicables a transmisiones multicast, donde el feedback desde cada receptor es imposible de cualquier modo.

4 Bibliografía

1. *A Universal Algorithm for Sequential Data Compression*, Ziv J., Lempel A., IEEE Transactions on Information Theory, 23, no. 3, 1977 (LZ77)

2. *comp.compression FAQ*. Jean-loup Gailly. (Aparición mensual en NEWS).

3. *Data Compression with Finite Windows*. Fiala, E.R., Greene, D.H., Communications of the ACM 32,4 (1989).

4. *Data Compression: Methods and Theory*. Storer, James A., Computer Science Press, 1988.
Este es uno de los mejores libros, aunque yo todavía no he podido leerlo (lo pedí a la biblioteca = nada de nada, no soy profesor).

5. *Fuentes del Infozip Zip 2.x / Unzip 5.x*. Mark Adler, Richard B. Wales, Jean-loup Gailly, Kai Uwe Rommel e Igor Mandrichenko.

6. *Lossless Data Compression*, Apiki, S., BYTE, March 1991.

7. *APPNOTE.TXT de la distribución de PK-ZIP 1.x y 2.x.*

8. *Yabba, paquete con código fuente y documentación sobre "Y coding"*. Bernstein, Daniel J.

9. J. Ziv & A. Lempel, IEEE Transactions on Information Theory, 24, 1978 (LZ78)

10. *A Technique for High Performance Data Compression*. Welch, Terry A., Computer, 17, no. 6, June 1984. (LZW)

11. *Esquemas algorítmicos fundamentales. Vol. III: ordenación y búsqueda*. Knuth, Donald E., sec. 6.4.

12. *Compress para Unix 4.12, fuentes*. Spencer W. Thomas, Jim McKie, Steve Davies, Ken Turkowski, James A. Woods, Joe Orost, Dave Mack, 1991.

13. *Predictive Text Compression by Hashing*. Raita, T. y Teuhola, J., ACM Conference on Information Retrieval, 1987.

14. *Fuentes de LZRW1a y LZRW3a*. Ross N. Williams, 1991.

15. *Data Compression Algorithms of LARC and LHarc*. Haruhiko Okumura, 1989.

16. *Application of Splay Trees to Data Compression*. Douglas W. Jones. Communications of the ACM, August 1988, página 996.

17. *Computer Networks. Third Edition*. Andrew S. Tanenbaum. Prentice-Hall. 1996.