Compresión de textos en Bases de Datos Digitales

Nieves R. Brisaboa[1], Antonio Fariña[1], Gonzalo Navarro[3], Eva L. Iglesias[2], José R. Paramá[1] y María F. Esteller[1]

[1] Database Lab., Univ. da Coruña, Facultade de Informática, Campus de Elviña s/n, 15071 A Coruña. {brisaboa,fari,parama}@udc.es, mfesteller@yahoo.es
[2] Dept. de Informática, Univ. de Vigo , Esc. Sup. de Enxeñería Informática, Campus As Lagoas s/n, 32001, Ourense. eva@uvigo.es
[3] Dept. of Computer Science, Univ. de Chile, Blanco Encalada 2120, Santiago, Chile. gnavarro@dcc.uchile.cl

Resumen Este trabajo presenta una revisión de los métodos de compresión de textos, que permiten la búsqueda directa de palabras y frases dentro del texto sin necesidad de descomprimirlo.
Se presentan las técnicas de compresión basadas en Huffman y dos técnicas más recientes: el método Denso con Post-Etiquetado y el método (s,c)-Denso. Además se muestra como estos nuevos métodos son directamente comparables, en tasa de compresión, con las técnicas basadas en Huffman y cómo proporcionan una compresión más simple y rápida, manteniendo sus características más interesantes. De este modo estas nuevas técnicas son extremadamente adecuadas para la compresión de textos sobre los que haya que realizar operaciones de Text Retrieval, pues facilita la indexación y preprocesado de los mismos sin necesidad de descomprimirlos.

1. Introducción

En los últimos años las colecciones de textos han crecido de forma sustancial debido principalmente a la generalización del uso de las bibliotecas digitales, bases de datos documentales, sistemas de ofimática y la Web.

Aunque la capacidad de los dispositivos actuales para almacenar información crece rápidamente mientras que los costes asociados decrecen, el tamaño de las colecciones de texto crece más rápidamente. Además, la velocidad de la CPU crece mucho más rápidamente que las velocidades de los dispositivos de almacenamiento y de las redes, por lo tanto, almacenar la información comprimida reduce el tiempo de entrada/salida, lo cual es cada vez más interesante a pesar del tiempo extra de CPU necesario.

Sin embargo, si el esquema de compresión no permite buscar palabras directamente sobre el texto comprimido, la recuperación de documentos será menos eficiente debido a la necesidad de descomprimir antes de la búsqueda.

Las técnicas de compresión clásicas, como los conocidos algoritmos de Ziv y Lempel [14,15] o la codificación de Huffman [4], no son adecuadas para las grandes colecciones de textos.

Los esquemas de compresión basados en la técnica de Huffman no se utilizan comúnmente en lenguaje natural debido a sus pobres ratios de compresión. Por otro lado, los algoritmos de Ziv y Lempel obtienen mejores ratios de compresión, pero la búsqueda de una palabra sobre el texto comprimido es ineficiente [6].

En [11] se presentan dos técnicas de compresión, denominadas *Plain Huffman* y *Tagged Huffman*, que permiten la búsqueda de una palabra en el texto comprimido (sin necesidad de descomprimirlo) de modo que la búsqueda puede ser hasta ocho veces más rápida para ciertas consultas, al mismo tiempo que obtienen buenos ratios de compresión.

En [3] presentamos un código denominado *Código Denso con Post-etiquetado*, que mantiene las propiedades del Tagged Huffman mejorándolo en algunos aspectos: (*i*) ratio de compresión, (*ii*) mismas posibilidades de búsqueda, (*iii*) codificación más simple y rápida y (*iv*) una representación más simple y pequeña del vocabulario. En [7] presentamos el *Código* (s, c)*-Denso*, una generalización del Código Denso con Post-etiquetado que mejora el ratio de compresión al adaptar el proceso de codificación al corpus a comprimir.

En este artículo se describen las técnicas de compresión que permiten la realización de búsquedas sobre texto comprimido. En la sección 2 se introducen las técnicas basadas en Huffman. En la sección 3 se muestra la codificación Densa con Post-Etiquetado y en la siguiente sección la codificación (s, c)-Densa. En la sección 5 se muestran resultados empíricos que permiten comparar los ratios de compresión alcanzados por cada técnica. Para finalizar se exponen las conclusiones de este trabajo.

2. Técnicas de compresión basadas en Huffman

Huffman es un método de codificación ampliamente conocido [4]. La idea de la codificación Huffman es comprimir el texto asignando códigos más cortos a los símbolos más frecuentes. Se ha probado que el algoritmo de Huffman obtiene, para un texto dado, una codificación óptima de prefijo libre.

Una codificación se denomina de prefijo libre (o código instantáneo) si no produce ningún código que sea prefijo de otro código. Un código de prefijo libre puede ser decodificado sin necesidad de comprobar códigos futuros, dado que el final de un código es inmediatamente reconocible.

En los últimos años se han desarrollado nuevas técnicas de compresión especialmente indicadas para su aplicación sobre textos en lenguaje natural. Estas técnicas surgen con la premisa de permitir la realización de búsquedas directamente en el texto comprimido, evitando así la necesidad de descomprimir antes de buscar. Todas ellas se basan en la utilización de las palabras como los símbolos a comprimir [5] (en lugar de los caracteres, como sucede en los métodos clásicos). Esta idea encaja perfectamente con los requerimientos de los algoritmos de Recuperación de Textos, dado que las palabras son generalmente los átomos de dichos sistemas.

En [11] Moura et al. presentan dos esquemas de compresión basados en palabras que utilizan Huffman para la codificación. Ambos métodos usan un

modelo semi-estático; esto es, en el proceso de codificación se realiza una primera pasada para obtener las frecuencias de las n palabras distintas del texto original y, en una segunda pasada, se lleva a cabo la compresión del texto. Cada palabra del texto original se reemplaza por un código que es una secuencia de símbolos de un alfabeto de codificación formado por b símbolos.

Los métodos de Moura et al. siguen la idea principal del método Huffman clásico [4]; esto es, pretenden codificar el texto original de forma que se asignen códigos más cortos a aquellos símbolos más frecuentes. Los códigos generados son de *prefijo libre*, lo que significa que ningún código es prefijo de otro.

Para la generación de los códigos Huffman se utiliza un árbol que se construye a partir de las probabilidades de los símbolos (palabras, en este caso) que forman el texto a comprimir. La construcción del árbol Huffman comienza creando un nodo hoja por cada palabra del documento original. A continuación se crea un nodo padre para las $[1 + ((n - b) mod (b - 1))]$ palabras de menor frecuencia de aparición [10] (o para las b palabras de menor frecuencia cuando $((n - b) mod (b - 1)) = 0$). Al nuevo nodo padre se le asigna una frecuencia que es igual a la suma de las frecuencias de sus nodos hijo. Esta operación se repite sucesivamente eligiendo los b nodos con menor probabilidad (ignorando los nodos que ya son hijos) hasta que quede un único nodo sin procesar, que será el *nodo raíz* del árbol (ver [2,9,12] para más detalle).

Una vez construido el árbol ya se puede calcular el código correspondiente a cada palabra del vocabulario. Para ello, en cada nivel del árbol se etiquetan las ramas con los diferentes símbolos del alfabeto de codificación. El código final de cada palabra estará formado por la secuencia de símbolos asignados a las ramas que unen la raíz con la hoja que representa dicha palabra.

Mientras el método básico propuesto por Huffman es mayormente usado como un código binario, esto es, cada palabra del texto original es codificada como una secuencia de bits, en [16,11] se modifica la asignación de códigos de modo que a cada palabra del texto original se le asigna una secuencia de bytes en lugar de una secuencia de bits (por lo tanto, el alfabeto está formado por los símbolos del 0 al 255 en lugar de los símbolos 0 y 1). Se denominan por ello, métodos de *Codificación Huffman orientada a byte basada en palabras*. Estudios experimentales han mostrado que no existe una degradación significativa en el ratio de compresión de textos en lenguaje natural al utilizar bytes en lugar de bits. Sin embargo, la descompresión y la búsqueda sobre textos comprimidos usando bytes en lugar de bits son más eficientes debido a que no es necesaria la manipulación de bits.

El primero de los métodos presentados en [11], la codificación *Plain Huffman*, emplea los 8 bits de cada byte en el proceso de codificación.

El segundo, denominado *Tagged Huffman*, reserva un bit de cada byte como *marca*, lo que permite distinguir claramente el comienzo de cada código dentro del texto comprimido. En concreto, el primer byte de cada código tiene un bit de marca con valor 1, mientras que los bytes restantes del código tienen su bit de marca a 0. Al usar un bit de marca, quedan sólo disponibles 7 bits de cada byte para realizar la construcción del árbol Huffman. Obsérvese que el uso del

código Huffman sobre los 7 bits restantes de cada byte es obligatorio, dado que el bit de marca no es suficiente para garantizar un código de prefijo libre.

Veamos cómo varían los códigos generados por ambas codificaciones en el siguiente ejemplo:

Ejemplo 1 Consideremos un texto con vocabulario $A, B, C, D, E, F, G, H, I, J$ cuyas frecuencias de aparición son 0.2, 0.2, 0.15, 0.15, 0.14, 0.09, 0.04, 0.02, 0.015, 0.015. En la Figura 1 se muestra un posible árbol para la codificación *Plain Huffman* y otro para la *Tagged Huffman*, suponiendo, por simplicidad, que los símbolos del alfabeto de codificación están formados por 3 bits (en lugar de 8). Los códigos resultantes figuran en el Cuadro 1.

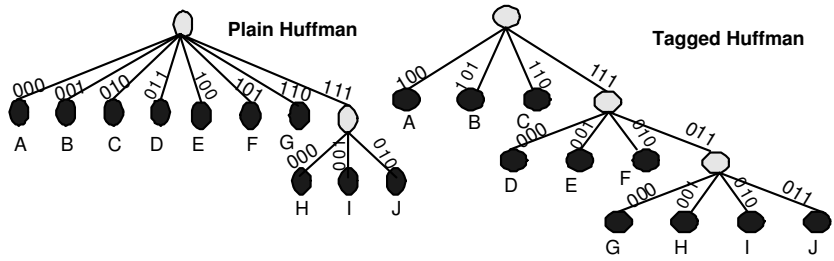

Figura1. Árboles para Plain Huffman y Tagged Huffman

El código Tagged Huffman tiene un precio en términos de compresión: se almacenan bytes completos pero sólo se usan 7 bits de cada byte para codificar información. Por lo tanto, el fichero comprimido crece aproximadamente un 11 % con respecto a la codificación Plain Huffman.

La adición del bit de marca en el código Tagged Huffman permite realizar búsquedas directamente sobre el texto comprimido. Para ello simplemente se comprime el patrón y, posteriormente, se utiliza cualquier algoritmo clásico de emparejamiento de patrones. Si se utiliza Plain Huffman esto no funciona correctamente debido a que la versión codificada del patrón puede aparecer en el texto comprimido a pesar de no aparecer en el texto real. Este problema se debe a que la concatenación de partes de dos códigos puede formar el código correspondiente a otra palabra del vocabulario. Esta situación no puede darse con el código Tagged Huffman gracias al uso del bit de marca que, en cada código determina para cada byte si dicho byte es el primero del código o no. Por ello, los algoritmos de búsqueda usados en Plain Huffman han de tener una fase de postprocesado que permita verificar si una ocurrencia es válida, lo que ocasiona una gran pérdida de rendimiento en las búsquedas.

3. Codificación Densa con Post-Etiquetado

El método Denso con Post-Etiquetado [3] es un nuevo esquema de compresión que conserva las ventajas de la codificación Tagged Huffman pero consigue

mejores ratios de compresión. Esta nueva técnica emplea un algoritmo más sencillo para la generación de códigos que aquel y no está basado en Huffman. Su sencillez a la hora de la codificación permite agilizar también los procesos de compresión y descompresión del texto. Es importante destacar que, a pesar de estas destacables mejoras, se basa igualmente en la utilización de marcas que permiten distinguir los códigos dentro del texto comprimido, por lo que mantiene las mismas posibilidades que la codificación Tagged Huffman a la hora de realizar búsquedas (exactas o aproximadas) directamente sobre el texto comprimido.

La codificación Densa con Post-Etiquetado es una codificación orientada a byte y basada en palabras donde cada código está formado por una secuencia de bytes. Al igual que sucedía con la codificación Tagged Huffman, el primer bit de cada byte se emplea como bit de marca. La única diferencia estriba en que, en lugar de utilizarlo como marca de comienzo de palabra, ahora el bit indica si el byte actual es el último de un código o no. En concreto, el bit de marca es 1 para el último byte de cada código.

Este cambio tiene sorprendentes consecuencias. El que el bit de marca indique el final de un código es suficiente para asegurar que un código no es nunca *prefijo* de otro, sin importar el contenido de los 7 bits restantes, pues únicamente el bit de marca del último símbolo de cada código vale 1. Por lo tanto, ya no es necesario aplicar codificación Huffman sobre los 7 bits restantes y se pueden utilizar todas las combinaciones de 7 bits para cada uno de los símbolos del código.

Proceso de codificación Una vez eliminada la necesidad de utilizar codificación Huffman, el problema reside en encontrar una asignación de códigos óptima; es decir, una codificación que minimice la longitud de la salida. Se sigue conservando, por tanto, la idea de asignar códigos más cortos a las palabras más frecuentes, como en todas las técnicas de compresión.

El proceso de codificación consta de dos fases: en primer lugar se ordenan las palabras por orden decreciente de frecuencias de aparición; a continuación se realiza una asignación secuencial de códigos a cada palabra utilizando 7 bits y añadiendo el bit de marca a cada uno de los bytes que componen un código. Este bit de marca será 1 en el caso del primer bit del último byte, y 0 en el caso del primer bit de cualquier otro byte del código. De este modo, la primera palabra se codificará como **1**0000000, la 128 como **1**1111111, la palabra 129 como **0**0000000 **1**0000000 y la 130 como **0**0000000 **1**0000001. Tras el código **0**1111111 **1**1111111 se pasa a **0**0000000 **0**0000000 **1**0000000, y así sucesivamente.

La codificación, por tanto, es extremadamente simple y resulta ser mucho más rápida que la correspondiente a cualquiera de las técnicas basadas en Huffman, al no ser preciso construir un árbol para realizar la asignación de códigos.

Al contrario de lo que sucedía en la codificación Huffman donde la frecuencia de las palabras influía en la longitud de los códigos asignados, esta nueva codificación no depende de dicha frecuencia sino de la posición que una palabra ocupe dentro del vocabulario (estando éste ordenado decrecientemente por frecuencias). Así, el código asignado a una palabra que ocupe la posición i, con frecuencia 0.15 será el mismo que si esa palabra tuviese frecuencia 0.01.

4. Códigos (s,c)-Densos

Como ya se ha visto en la sección anterior, los Códigos Densos con Post-etiquetado usan 2^{b-1} valores, desde el 0 hasta el $2^{b-1}-1$, para los bytes al comienzo de un código (valores *no terminales*), y utiliza otros 2^{b-1} valores, del 2^{b-1} al 2^b-1, para el último byte de un código (valores *terminales*). Pero la pregunta que se plantea ahora es cual es la distribución óptima de valores *terminales* y *no terminales*, esto es, para un corpus determinado con una distribución de frecuencias de palabras determinada, puede ocurrir que un número diferente de valores no terminales (continuadores) y valores terminales (stoppers) comprima mejor que la alternativa de utilizar 2^{b-1} valores para cada conjunto. Esta idea ya fue apuntada en [8]. Nosotros definimos los *Códigos (s, c)-stop-cont* como sigue.

Definición 1 Dado un conjunto de símbolos fuente con probabilidades $\{p_i\}_{0\leq i<n}$, *una codificación* (s,c)*-stop-cont (en la cual* c *y* s *son enteros mayores que cero) asigna a cada símbolo fuente* i *un código único formado por una secuencia de símbolos de salida en base* c *y un símbolo final que estará entre* c *y* $c+s-1$.

Hemos llamado a los símbolos que están entre 0 y $c-1$ "continuadores" y a aquellos entre c y $c+s-1$ "stoppers". Por lo tanto, cualquier código (s,c) stop-cont es de prefijo libre.

Entre todas las posibles codificaciones (s,c) stop-cont para una distribución de probabilidades dada, el *código denso* es la que minimiza la longitud media de los símbolos.

Definición 2 Dado un conjunto de símbolos fuente con probabilidades decrecientes $\{p_i\}_{0\leq i<n}$, *los códigos correspondientes a su codificación* (s,c)*-Denso ((s,c)-DC) se asignan de la siguiente manera: sea* $k\geq 1$ *el número de bytes del código correspondiente a un símbolo fuente* i, *entonces* k *será tal que:* $s\frac{c^{k-1}-1}{c-1}\leq i<s\frac{c^k-1}{c-1}$

Así, el código correspondiente al símbolo fuente i estará formado por $k-1$ símbolos (de salida) en base-c y un símbolo final. Si $k=1$ entonces el código es simplemente el stopper $c+i$. De otro modo el código estará formado por el valor $\lfloor x/s\rfloor$ escrito en base c, seguido por $c+(x\ mod\ s)$, en el cual $x=i-\frac{sc^{k-1}-s}{c-1}$.

Ejemplo 2 Los códigos asignados a los símbolos $i\,\epsilon\,0\ldots15$ por un (2,3)-DC serán: ⟨3⟩, ⟨4⟩, ⟨0,3⟩, ⟨0,4⟩, ⟨1,3⟩, ⟨1,4⟩, ⟨2,3⟩, ⟨2,4⟩, ⟨0,0,3⟩, ⟨0,0,4⟩, ⟨0,1,3⟩, ⟨0,1,4⟩, ⟨0,2,3⟩, ⟨0,2,4⟩, ⟨1,0,3⟩ y ⟨1,0,4⟩.

Observe que los códigos no dependen en las probabilidades exactas de cada símbolo sino es su ordenación por frecuencia.

Dado que con k dígitos obtenemos sc^{k-1} códigos diferentes, denominamos $W_k^s=\sum_{j=1}^{k}sc^{j-1}=s\frac{c^k-1}{c-1}$, (donde $W_0^s=0$) al número de símbolos fuentes que

pueden ser codificados con hasta k dígitos. Denominamos $f_k^s = \sum_{j=W_{k-1}^s+1}^{W_k^s} p_j$ a la suma de probabilidades de los símbolos fuente codificadas con k dígitos por un (s,c)-DC.

Entonces, la longitud media de los códigos para tal (s,c)-DC, $LD_{(s,c)}$, es

$$LD_{(s,c)} = \sum_{k=1}^{K^s} k f_k^s = \sum_{k=1}^{K^s} k \sum_{j=W_{k-1}^s+1}^{W_k^s} p_j = 1 + \sum_{k=1}^{K^s-1} k \sum_{j=W_k^s+1}^{W_{k+1}^s} p_j = 1 + \sum_{k=1}^{K^s-1} \sum_{j=W_k^s+1}^{W_k^s} p_j$$

$K^x = \lceil \log_{(2^b-x)} \left(1 + \frac{n(2^b-x-1)}{x}\right) \rceil$ y n es el número de símbolos en el vocabulario.

Por la Definición 2 el código denso con Post-Etiquetado [3] es un $(2^{b-1},2^{b-1})$-DC y por ello *(s,c)*-DC puede ser visto como una generalización del código denso con Post-Etiquetado en el que s y c han sido ajustados para optimizar la compresión según la distribución de frecuencias del vocabulario.

En [3] está probado que $(2^{b-1},2^{b-1})$-DC es de mayor eficiencia que el Tagged Huffman. Esto es debido a que el Tagged Huffman es un código $(2^{b-1},2^{b-1})$ *(non dense) stop-cont*, mientras que el método denso con Post-Etiquetado sí es un código $(2^{b-1},2^{b-1})$-Denso.

Ejemplo 3 El Cuadro 1 muestra los códigos asignados a un pequeño conjunto de palabras ordenadas por frecuencia usando el Plain Huffman (P.H.), *(6,2)*-DC, método denso con Post-Etiquetado (ETDC) que es un *(4,4)*-DC, y Tagged Huffman (TH). Por simplicidad se han utilizado símbolos de tres bits (*b=3*). Las últimas cuatro columnas muestran el producto del número de bytes por las frecuencia de cada palabra y su suma (la longitud media de los códigos), se muestra en la última fila.

Los valores (6,2) para s y c no son los óptimos sino que una codificación (7,1)-Densa obtendría el texto comprimido óptimo, siendo en este ejemplo el mismo resultado que en el Plain Huffman. El problema ahora consiste en encontrar los valores s y c (asumiendo un b fijo donde $2^b = s + c$) que minimice el tamaño del texto comprimido.

						Frec × bytes			
Palabra	Frec	P.H.	(6,2)-DC	ETDC	T.H.	P.H.	(6,2)-DC	ETDC	T.H.
A	0.2	[000]	[010]	[100]	[100]	0.2	0.2	0.2	0.2
B	0.2	[001]	[011]	[101]	[101]	0.2	0.2	0.2	0.2
C	0.15	[010]	[100]	[110]	[110]	0.15	0.15	0.15	0.3
D	0.15	[011]	[101]	[111]	[111][000]	0.15	0.15	0.15	0.3
E	0.14	[100]	[110]	[000][100]	[111][001]	0.14	0.14	0.28	0.28
F	0.09	[101]	[111]	[000][101]	[111][010]	0.09	0.09	0.18	0.18
G	0.04	[110]	[000][010]	[000][110]	[111][011][000]	0.04	0.08	0.08	0.12
H	0.02	[111][000]	[000][011]	[000][111]	[111][011][001]	0.04	0.04	0.04	0.05
I	0.005	[111][001]	[000][100]	[001][100]	[111][011][010]	0.01	0.01	0.01	0.015
J	0.005	[111][010]	[000][101]	[001][101]	[111][011][011]	0.01	0.01	0.01	0.015
					Tamaño total comprimido	**1.03**	**1.07**	**1.30**	**1.67**

Cuadro1. Comparativa entre métodos de compresión

4.1. Valores s y c óptimos: Unicidad del mínimo

Antes de dar un algoritmo para obtener los valores óptimos para s y c, $s + c = 2^b$ y $1 \leq s \leq 2^b - 1$, necesitamos mostrar que el tamaño del texto comprimido decrece a medida que se incrementa s hasta alcanzar el valor s óptimo. Tras este punto, al incrementar el valor de s, se produce una pérdida en el ratio de compresión. Este hecho se muestra en la figura 2. Recordemos que el valor de c depende del valor de s pues $c = 2^b - s$.

Aunque disponemos de una demostración formal de la unicidad de un valor óptimo de s, debido a su complejidad, se ha decidido incluir aquí una explicación intuitiva de dicha propiedad que permite entrever el camino seguido en dicha demostración pero sin profundizar en detalles.

Se puede observar que cuando s toma valores muy pequeños (próximos a 1), el número de palabras muy frecuentes que se pueden codificar con pocos bytes (uno o dos bytes) es también muy pequeño (s palabras pueden ser codificadas con un byte y $s \cdot c$ con dos bytes). Pero en este caso c es grande, y por tanto hay un gran número de palabras de baja frecuencia que están siendo codificadas con pocos bytes ($s \cdot c^2$ palabras se codifican con 3 bytes, $s \cdot c^3$ con 4 bytes así sucesivamente), pero cuando c es muy grande, normalmente 3 bytes son suficientes para codificar la última palabra del vocabulario (la menos frecuente).

Es fácil de ver que, a medida que s crece, más palabras de alta frecuencia son codificadas con menos bytes, lo que incrementa su compresión. Pero al mismo tiempo, cuando el valor de s aumenta, las palabras de menor frecuencia necesitan más bytes para codificarse, lo que reduce la compresión de estas palabras.

Como consecuencia, si se prueban todos los valores posibles de s, comenzando en $s = 1$, se observa (Figura 2) que, al principio, la compresión crece enormemente debido a que el aumento de s origina que las palabras de alta frecuencia se codifiquen con códigos más cortos. Poco a poco, para cada incremento de s, la ganancia de compresión obtenida al codificar con pocos bytes las palabras más frecuentes se ve contrarrestada por el incremento del tamaño de los códigos utilizados en la codificación de las palabras menos frecuentes. Existe un punto donde esta ganancia comienza a ser menor que la pérdida de compresión debida a las palabras menos frecuentes. Este punto nos indica el valor óptimo de s.

En la Figura 2 se muestran los ratios de compresión para dos corpus dados, en función del valor de s. Se puede observar cómo los valores de compresión cercanos al óptimo son prácticamente insensibles al valor de s. Este hecho es el que da lugar a que la curva sea muy suave en su parte inferior.

Nuestro algoritmo saca ventaja de esta propiedad. De esta forma no es necesario comprobar el ratio de compresión alcanzado con todos los valores de s posibles, sino que al conocer la forma de la distribución del ratio compresión en función de s, es posible buscar únicamente en la dirección en la que los valores de s originan una mejora en el ratio de compresión.

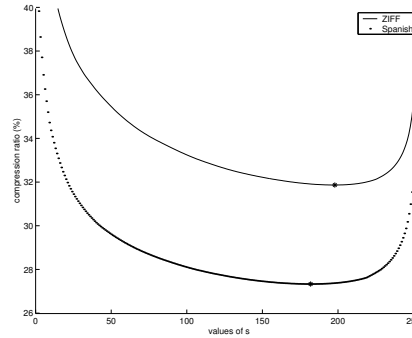

Figura2. Tamaño del texto comprimido en función de s.

4.2. Algoritmo para obtener los valores óptimos de s y c

En esta sección se describe un algoritmo para buscar el mejor s y, consecuentemente, c para un corpus y un b dado. El algoritmo básico presentado es una búsqueda binaria que calcula inicialmente el tamaño del texto para dos valores consecutivos de s: ($\lfloor 2^{b-1} \rfloor - 1$ y $\lfloor 2^{b-1} \rfloor$). Mediante la propiedad de la unicidad del mínimo, sabemos que existe a lo sumo un mínimo local, por lo que el algoritmo dirige la búsqueda hasta obtener el punto que proporcione el mejor ratio de compresión. En cada iteración del algoritmo, el espacio de búsqueda se reduce a la mitad y se calcula la compresión en dos puntos centrales del nuevo intervalo, lo que permite dirigir la búsqueda en la dirección deseada.

El algoritmo **findBestS** calcula el mejor valor para s y c para un b y una lista de frecuencias acumuladas dadas. El *tamaño del texto comprimido* (en bytes) se calcula, para unos valores específicos de s y c, mediante la función **ComputeSizeS**

Finalmente se devuelven los valores de s y c que minimizan el tamaño del texto comprimido.

```
findBestS (b, freqList)
  //Entrada: b valor (2^b = c + s).
  //Entrada: Una Lista de frecuencias acumuladas ordenadas por orden decreciente de frecuencia.
  //Salida: Los mejores valores para s y c
 begin
   Lp := 1;              //Lp y Up puntos menor y mayor
   Up := 2^b − 1;        // del intervalo que se comprueba
   while Lp + 1 < Up do
      M := ⌊(Lp+Up)/2⌋
      sizePp := computeSizeS(M − 1, 2^b − (M − 1), freqList)// tamaño con M-1
      sizeM := computeSizeS(M, 2^b − M, freqList)//          tamaño con M
      if sizePp < sizeM then
         Up := M − 1
      else Lp := M
      end if
   end while
   if Lp < Up then     //Lp = Up − 1 y M = Lp
      sizeNp := computeSizeS(Up, 2^b − Up, freqList); // tamaño con M+1
      if sizeM < sizeNp then
         bestS := M
```

```
        else bestS := Up
        end if
    else bestS := Lp     //Lp = Up = M − 1
    end if
    bestC := 2^b − bestS
    return bestS, bestC
  end
```

Para cualquier s y c, el siguiente algoritmo calcula el tamaño del texto comprimido.

```
computeSizeS (s, c, freqList)
  //Entradas: s, c, y una lista de frecuencias acumuladas
  //Salida: tamaño del texto comprimido s,c
  begin
    k := 1, n := number ofwords in 'freqList'
    Right := min(s, n); Left := Right;
    total := freqList[Right − 1];
    while Right < n do
          Left := Right;
          Right := Right + sc^k;
          k := k + 1
          if Right > n then
               Right := n
          end if
          total := total + k ∗ (freqList[Right − 1] − freqList[Left])
    end while
    return total
  end
```

Calcular el tamaño del texto comprimido para un valor específico de s tiene un coste de $O(\log_c n)$, excepto para $c = 1$, en cuyo caso su coste es de $O(n/s) = O(n/2^b)$. La secuencia más cara de llamadas a **computeSizeS** en una búsqueda binaria es la de valores $c = 2^{b-1}$, $c = 2^{b-2}$, $c = 2^{b-3}$, ..., $c = 1$. El coste total de **computeSizeS** sobre esa secuencia de valores c es $\frac{n}{2^b} + \sum_{i=1}^{b-1} \log_{2^{b-i}} n = \frac{n}{2^b} + \log_2 n \sum_{i=1}^{b-1} \frac{1}{b-i} = O\left(\frac{n}{2^b} + \log n \log b\right)$.

Las demás operaciones de la búsqueda binaria son constantes, aunque tenemos un coste extra de $O(n)$ para calcular las frecuencias acumuladas. Por tanto, el coste total de encontrar s y c es $O(n+\log(n)\log(b))$. Puesto que el mayor b de interés es aquel tal que $b = \lceil \log_2 n \rceil$ (en este punto podemos codificar cada símbolo usando un único stopper), el algoritmo de optimización tiene un coste a lo sumo de $O(n + \log(n) \log\log(n)) = O(n)$, suponiendo que el vocabulario estuviera ya ordenado.

El algoritmo de Huffman es también linear una vez el vocabulario está ordenado, pero resulta más lento en tiempo debido a que las operaciones que se deben realizar para la construcción del árbol son más costosas que las realizadas en la obtención de los valores s y c óptimos.

5. Resultados empíricos

Hemos utilizado algunas colecciones grandes de textos del TREC-2 (AP Newswire 1988 y Ziff Data 1989-1990) y del TREC-4 (Congressional Record 1993, Financial Times 1991, 1992, 1993 y 1994). También hemos usado un corpus de literatura española que creamos previamente. Se han comprimido todos ellos

usando Plain Huffman, Código *(s,c)*-Denso, Denso con PostEtiquetado y Tagged Huffman. Para crear el vocabulario hemos utilizado el modelo *spaceless* [10] que consiste en que si una palabra va seguida de un espacio tan sólo se codifica la palabra.

Hemos excluido el tamaño del vocabulario comprimido en los resultados (por ser despreciable y similar en todos los casos)

Corpus	Tamaño original	Palabras del Voc	θ	P.H.	(s,c)-DC	ETDC	T.H.
AP Newswire 1988	250,994,525	241,315	1.852045	31.18	(189,67) 31.46	32.00	34.57
Ziff Data 1989-1990	185,417,980	221,443	1.744346	31.71	(198,58) 31.90	32.60	35.13
Congress Record	51,085,545	114,174	1.634076	27.28	(195,61) 27.50	28.11	30.29
Financial Times 1991	14,749,355	75,597	1.449878	30.19	(193,63) 30.44	31.06	33.44
Financial Times 1992	175,449,248	284,904	1.630996	30.49	(193,63) 30.71	31.31	33.88
Financial Times 1993	197,586,334	291,322	1.647456	30.60	(195,61) 30.79	31.48	34.10
Financial Times 1994	203,783,923	295,023	1.649428	30.57	(195,61) 30.77	31.46	34.08
Spanish Text	105,125,124	313,977	1.480535	27.00	(182,74) 27.36	27.71	29.93

Cuadro2. Comparación de los ratios de compresión.

El Cuadro 2 muestra los ratios de compresión obtenidos para los corpus mencionados. La segunda columna contiene el tamaño original del corpus procesado y las siguientes columnas indican el número de palabras del vocabulario, el parámetro θ de la Ley de Zipf [13,1] y el ratio de compresión para cada método. La sexta columna, que se refiere al método (s, c)-DC, proporciona los valores óptimos de (s, c) encontrados usando el algoritmo **findBestS**.

Como se puede observar, Plain Huffman obtiene el mejor ratio de compresión (como cabía esperar por ser la codificación óptima de prefijo libre) y el método denso con PostEtiquetado siempre obtiene mejores resultados que el Tagged Huffman, con una mejora de hasta 2.5 puntos. Como esperábamos, (s, c)-DC mejora los resultados obtenidos por $(128, 128)$-DC (ETDC). De hecho, (s, c)-DC es mejor que $(128, 128)$-DC en más de 0.5 puntos y peor que Plain Huffman en menos de 0.5 puntos de media.

6. Conclusiones

Se ha presentado una revisión de los métodos de compresión que permiten búsqueda sobre texto comprimido. Se han mostrado los métodos basados en Huffman, centrándonos especialmente en el Tagged Huffman. También se han comentado las mejoras aportadas por la Codificación Densa con Post-Etiquetado y la generalización de éste: la codificación (s, c)-Densa. Esta codificación mejora el ratio de compresión respecto a la Densa con Post-Etiquetado, adaptando sus parámetros (s, c) al corpus que será comprimido. Se ha proporcionado un algoritmo que calcula los valores óptimos de s, c que maximizan la compresión.

Hemos presentado algunos resultados empíricos comparando nuestros dos métodos con otros códigos de similares características. Se observa que nuestros códigos siempre son mejores que el Tagged Huffman, obteniendo tan sólo un 0.5 % menos de ratio de compresión sobre la codificación óptima Huffman. Además el

método $(s, c) - Denso$ y el Denso con Post-Etiquetado resultan más rápidos y simples de construir.

Referencias

1. R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman, May 1999.
2. T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression.* Prentice Hall, 1990.
3. N. Brisaboa, E. Iglesias, G.Ñavarro, and J. Paramá. An efficient compression code for text databases. In *25th European Conference on IR Research, ECIR 2003*, LNCS 2633, pages 468–481, 2003.
4. D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. Inst. Radio Eng.*, 40(9):1098–1101, 1952.
5. A. Moffat. Word-based text compression. *Software - Practice and Experience*, 19(2):185–198, 1989.
6. G.Ñavarro and J. Tarhio. Boyer-Moore string matching over Ziv-Lempel compressed text. In *Proc. 11th Annual Symposium on Combinatorial Pattern Matching (CPM 2000)*, LNCS 1848, pages 166–180, 2000.
7. GonzaloÑavarro Nieves R. Brisaboa, Antonio Fariña and Maria F. Esteller. (s,c)-dense coding: An optimized compression code for natural language text databases. In *String Processing and Information Retrieval, to appear*, Manaos, Brazil, 2003.
8. J. Rautio, J. Tanninen, and J. Tarhio. String matching with stopper encoding and code splitting. In *Proc. 13th Annual Symposium on Combinatorial Pattern Matching (CPM 2002)*, LNCS 2373, pages 42–52, 2002.
9. David Salomon. *Data Compression: The Complete Reference*. Springer-Verlag, Berlin, Germany /Heidelberg, Germany /London, UK /etc., 2º edition, 2000.
10. E. Silva de Moura, G.Ñavarro, N. Ziviani, and R. Baeza-Yates. Fast searching on compressed text allowing errors. In *Proc. 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR-98)*, pages 298–306, 1998.
11. E. Silva de Moura, G.Ñavarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems*, 18(2):113–139, 2000.
12. Ian H. Witten, A. Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, USA, 1999.
13. G.K. Zipf. *Human Behavior and the Principle of Least Effort*. Addison-Wesley, 1949.
14. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.
15. J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.
16. Nivio Ziviani, Edleno Silva de Moura, G. Navarro, and Ricardo Baeza-Yates. Compression: A key for next-generation text retrieval systems. *IEEE Computer*, 33(11):37–44, 2000.