Codificación (s,c)-Densa: optimizando la compresión de texto en lenguaje natural

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 un nuevo método para la compresión de textos, que permite la búsqueda directa de palabras y frases dentro del texto sin necesidad de descomprimirlo.
Este método es directamente comparable, en tasa de compresión, con las técnicas basadas en Huffman orientadas a palabras y proporciona una compresión más simple y rápida, manteniendo sus características más destacables de cara a la realización de búsquedas directas de palabras sobre el texto comprimido, al generar códigos con "marca" y de prefijo libre. De este modo esta técnica es extremadamente adecuada para la compresión de textos sobre los que haya que realizar operaciones de Text Retrieval, pues facilita la indexación y preprocesado sin necesidad de descomprimirlos.
En el presente artículo se describe la Codificación (s,c)-Densa y se muestra el proceso de obtención de los parámetros s y c que maximizan la compresión de un corpus determinado. Este proceso se basa en analizar la distribución de frecuencias de las palabras para, de este modo, generar códigos que minimicen la redundancia del código generado. Además se muestran resultados empíricos que demuestran la efectividad de esta nueva técnica de compresión.

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 [10,11] o la codificación de Huffman [3], 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 [5]. En [8] 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 [2] se presenta un código denominado *Código Denso con Post-etiquetado*, que mantiene las propiedades del Tagged Huffman mejorándolo en algunos aspectos: (*i*) mejores ratios de compresión, (*ii*) mismas posibilidades de búsqueda, (*iii*) codificación más simple y rápida y (*iv*) una representación más simple y pequeña del vocabulario.

En este artículo presentamos el *Código* (s, c)*-Denso*, una generalización del Código Denso con Post-etiquetado que mejora el ratio de compresión. El Código (s, c)-Denso comprime siempre mejor que el Código Denso con Post-Etiquetado y el Tagged Huffman y, sólo un 0,5 % menos que el Plain Huffman. Al mismo tiempo el Código (s, c)-Denso mantiene todas las ventajas del Código Denso con Post-Etiquetado y del Código Tagged Huffman.

2. Trabajo relacionado

Huffman es un método de codificación ampliamente conocido [3]. La idea de la codificación Huffman es comprimir el texto al asignar códigos más cortos a los símbolos más frecuentes. Se ha probado que el algoritmo de Huffman obtiene, para un texto dado, una codificación óptima (esto es, la compresión máxima) 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.

2.1. Compresión Huffman basada en palabras

Las implementaciones tradicionales del código Huffman son basadas en caracteres, esto es, utilizan los caracteres como los símbolos a comprimir. Esta estrategia cambió cuando en [4] se propuso usar palabras en lugar de caracteres. En [8,12], se presenta un esquema de compresión que usa esta estrategia

combinada con un código Huffman. Estos modelos usan un modelo semiestático, esto es, el codificador realiza una primera pasada sobre el texto para obtener la frecuencia de todas las palabras en el texto y posteriormente, en una segunda pasada, se codifica el texto. Durante la segunda fase, los símbolos originales (palabras) son reemplazados por códigos. Para cada palabra en el texto sólo existe un código, cuya longitud varía dependiendo de la frecuencia de la palabra en el texto.

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 [8] 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. 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.

2.2. Códigos Tagged Huffman

En [8] se presentan dos códigos que siguen esta aproximación. En dicho trabajo, se denomina Código *Plain Huffman* al código que ya hemos descrito, esto es, un código Huffman basado en palabras y orientado a bytes.

El segundo código propuesto se denomina Código *Tagged Huffman*. Este código es como el anterior pero con la diferencia de que se reserva un bit de cada byte para indicar si dicho byte es el primer byte de un código o no. Por lo tanto, sólo 7 bits de cada byte son usados para el código Huffman. Obsérvese que el uso del código Huffman sobre los 7 restantes bits de cada byte es obligatorio, dado que el bit de marca no es suficiente para garantizar un código de prefijo libre.

El código Tagged Huffman tiene un precio en términos de compresión: se almacena 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. Esto no puede ocurrir 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 primer byte del código o no. Por esta razón, la búsqueda sobre textos comprimidos con Plain Huffman debe comenzar desde el principio del fichero, mientras que con Tagged Huffman no es necesario.

2.3. Códigos densos con Post-Etiquetado

El Código Denso con Post-Etiquetado [2] comienza con un, aparentemente, simple cambio sobre el Código Tagged Huffman. En lugar de utilizar el bit de marca para indicar cuál es el primer byte de un código, dicho bit indica cuál es el último byte de cada código. Esto es, el bit de marca es 0 en todos los bytes de un código excepto el último, que tiene el bit de marca a 1.

Este cambio tiene consecuencias sorprendentes. Ahora el bit de marca es suficiente para asegurar que el código es de prefijo libre independientemente del contenido de los otros 7 bits. Gracias a esto, no hay necesidad de usar la codificación Huffman sobre los restantes 7 bits. Es posible usar *todas* las posibles combinaciones sobre los 7 bits de todos los bytes

Por otro lado, no estamos restringidos a usar símbolos de 8 bits para formar códigos. Es posible usar símbolos de b bits. Teniendo en cuesta esto, el Código Denso con Post-etiquetado se define como sigue:

Definición 1 Dado un conjunto de símbolos fuente con probabilidades decrecientes $\{p_i\}_{0\leq i<n}$*, el código correspondiente a cada símbolo fuente está formado por una secuencia de símbolos de b bits, todos ellos representados en base* (2^{b-1}) *(esto es, desde 0 hasta* $2^{b-1}-1$*), excepto el último símbolo, que tiene un valor entre* 2^{b-1} *y* 2^b-1*, y donde la asignación se de estos códigos a los símbolos fuente se realiza de un modo completamente secuencial.*

El cálculo de los códigos es extremadamente simple; sólo es necesario ordenar el vocabulario por frecuencia y luego asignar secuencialmente los códigos teniendo en cuenta el bit de marca. Por lo tanto, la fase de codificación será más rápida que usando Huffman.

3. Códigos (s,c)-Densos

Como ya se ha visto, 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 cuál 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 [6]. Nosotros definimos los *Códigos* (s,c)*-stop-cont* como sigue.

Definición 2 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 3 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 1 Los códigos asignados a los símbolos $i\,\epsilon\, 0\ldots 15$ 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

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

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

Por la Definición 3 el código denso con PostEtiquetado [2] 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 PostEtiquetado en el que s y c han sido ajustados para optimizar la compresión según la distribución de frecuencias del vocabulario.

En [2] 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*, mientra que el método denso con PostEtiquetado sí es un código $(2^{b-1},2^{b-1})$-Denso.

Ejemplo 2 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 PostEtiquetado (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 numero 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. Valores *s* y *c* óptimos: Unicidad del mínimo

Antes de presentar el algoritmo que calcula los valores óptimos para s y c, necesitamos probar que cuando tres valores consecutivos de s, digamos $s-1,\ s$, y $s+1$, son usados para comprimir el mismo texto y obtenemos respectivamente $T_{s-1},\ T_s$ y T_{s+1} tales que $T_{s-1} \geq T_s \leq T_{s+1}$, entonces s es el valor óptimo. Esto es, existe tan sólo un mínimo local del tamaño del texto comprimido y está en función de s y c, o hay un conjunto de valores consecutivos de s que obtienen la misma compresión, y en ese caso puede ser elegido cualquiera de ellos. Comenzaremos probando la unicidad del mínimo usando algunos lemas y definiciones.

Lema 1 Dados tres valores consecutivos de $\mathbf{s}$*, sean* $s-1,\ s$ *y* $s+1$*, existen dos números* K_z *y* K_y *tales que:*

(i) $\forall k \geq K_z,\quad W_k^{s-1} > W_k^s, \qquad \forall k < K_z,\quad W_k^{s-1} \leq W_k^s$
(ii) $\forall k \geq K_y,\quad W_k^s > W_k^{s+1}, \qquad \forall k < K_y,\quad W_k^s \leq W_k^{s+1}$
(iii) $K_y \leq K_z$

Lema 2 Para cualquier $k < K_y$ *se cumple la siguiente desigualdad:* $W_k^s - W_k^{s-1} \geq W_k^{s+1} - W_k^s$.

Sea z el primer símbolo, en el ranking de símbolos ordenados por frecuencia, que es codificado con un menor número de bytes por $(s-1, c+1)$-DC que por (s, c)-DC. Sea también y el primer símbolo en el ranking que es codificado con menos bytes por (s, c)-DC que por $(s+1, c-1)$-DC (estamos en todo momento asumiendo que $s+c=2^b$). Note que $y = W^{s+1}_{K_y} + 1$ y $z = W^{s}_{K_z} + 1$. Se puede ver como y estará antes que z en el ranking de palabras, y por ello y tendrá siempre una frecuencia mayor que z.

Sea A la secuencia de símbolos que preceden al símbolo z y B la secuencia de símbolos desde z hasta el final del vocabulario. Del mismo modo, C es la secuencia de símbolos que preceden a y y D la secuencia de símbolos desde y al final del vocabulario. Esto está representado en la Figura 1. Llamamos A_s al tamaño del texto comprimido relacionado con las palabras en A cuando usamos un (s, c)-DC. Del mismo modo, B_{s-1} representa el tamaño del texto comprimido teniendo en cuenta tan solo las palabras en B cuando se usa un $(s-1, c+1)$-DC, y así sucesivamente. De la definición de z y y tenemos: $A_s \leq A_{s-1}$; $B_s \geq B_{s-1}$; $C_s \geq C_{s+1}$; $D_s \leq D_{s+1}$. Ahora llamemos: $d_A = A_{s-1} - A_s$; $d_B = B_s - B_{s-1}$; $d_C = C_s - C_{s+1}$; $d_D = D_{s+1} - D_s$.

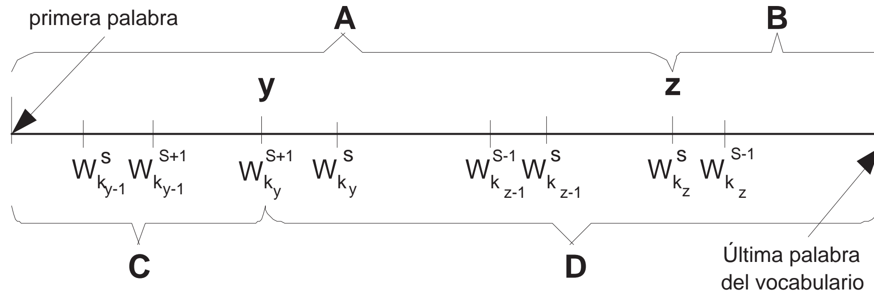

Figura1. Representación de las secuencias de símbolos A, B, C, y D.

Lema 3 Dado un corpus, $\forall s, b \; d_A \geq d_C$

Lema 4 Dado un corpus, $\forall s, b$, $d_C \geq d_D \Longrightarrow d_A \geq d_B$ *y* $d_C > d_D \Longrightarrow d_A > d_B$

Usando los lemas anteriores podemos probar que hay a lo sumo un único mínimo local en cuanto al tamaño del texto comprimido en función de s. Observe que una condición necesaria y suficiente para hacer imposible el mínimo local es que no pueda existir un valor s tal que $T_{s-1} < T_s > T_{s+1}$.

Teorema 1 Sea un corpus textual T codificado con un (s, c)-DC, no existe un valor s tal que $T_{s-1} \leq T_s > T_{s+1}$ *o* $T_{s-1} < T_s \geq T_{s+1}$

Demostración. Por el Lema 1, sabemos que

$$T_s = A_s + B_s; \; T_s = C_s + D_s; \; T_{s-1} = A_{s-1} + B_{s-1}; \; T_{s+1} = C_{s+1} + D_{s+1}$$

Dado que $T_s > T_{s-1}$ deducimos que $d_B > d_A$. Del mismo modo, a partir de $T_s > T_{s+1}$ obtenemos que $d_C > d_D$. Por el Lema 4 tenemos que $d_C \geq d_D$

$\Longrightarrow d_A \geq d_B$, por lo que obtenemos una contradicción. Siguiendo el mismo razonamiento $T_{s-1} \leq T_s \Longrightarrow d_B \geq d_A$ y por $T_{s+1} < T_s$ obtenemos $d_C > d_D$, por lo que tenemos la misma contradicción. Encontramos la misma contradicción cuando consideramos que $T_{s-1} < T_s \geq T_{s+1}$. □

5. Algoritmo para obtener los valores óptimos de *s* y *c*

En esta sección se mostrará 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 $\lfloor 2^{b-1} \rfloor$). Por el Teorema 1, sabemos que existe a lo sumo un mínimo local, por lo que el algoritmo guiará 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.

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, llamando a la función **ComputeSizeS** (que no incluimos por falta de espacio).

```
findBestS (b, freqList)
  //Entrada: b valor (2^b = c + s).
  //Entrada: A 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
```

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 b de máximo interés es aquel 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 $O(n+\log(n)\log\log\log(n))=O(n)$, suponiendo que el vocabulario estuviera ya ordenado.

6. 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 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* [7] 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 [9,1] 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 espera 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 aunque peor que Plain Huffman en tan sólo menos de 0.5 puntos de media.

7. Conclusiones

Hemos presentado el código (s, c)-Denso, un método nuevo para compresión de textos en lenguaje natural. Este método es una generalización del Denso con PostEtiquetado y mejora su ratio de compresión adaptando sus parámetros (s, c) al corpus que será comprimido. Se ha proporcionado un algoritmo que calcula los valores óptimos de s, c para un corpus dado, esto es, el par que maximiza el ratio de compresión.

Hemos presentado algunos resultados empíricos comparando nuestro método con otros códigos de similares características, siendo nuestro código siempre mejor que el método denso con PostEtiquetado y el Tagged Huffman, obteniendo tan sólo un 0.5 % menos de ratio de compresión sobre la codificación óptima Huffman. Nuestro método es más veloz y simple de construir.

Referencias

1. R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman, May 1999.
2. 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.
3. D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. Inst. Radio Eng.*, 40(9):1098–1101, 1952.
4. A. Moffat. Word-based text compression. *Software - Practice and Experience*, 19(2):185–198, 1989.
5. 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.
6. 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.
7. 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.
8. 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.
9. G.K. Zipf. *Human Behavior and the Principle of Least Effort*. Addison-Wesley, 1949.
10. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.
11. J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.
12. N. Ziviani, E. Silva de Moura, G.Ñavarro, and R. Baeza-Yates. Compression: A key for next-generation text retrieval systems. *Computer*, 33(11):37–44, 2000.