Nuevas aproximaciones en indexación de textos utilizando wavelet trees *

Nieves Brisaboa
Laboratorio de Bases de Datos
Universidade da Coruña
brisaboa@udc.es

Yolanda Cillero
Laboratorio de Bases de Datos
Universidade da Coruña
ycillero@udc.es

Antonio Fariña
Laboratorio de Bases de Datos
Universidade da Coruña
fari@udc.es

Susana Ladra
Laboratorio de Bases de Datos
Universidade da Coruña
sladra@udc.es

Oscar Pedreira
Laboratorio de Bases de Datos
Universidade da Coruña
opedreira@udc.es

Resumen

El desarrollo de aplicaciones que trabajan con grandes colecciones de textos ha dado lugar a la necesidad de métodos de indexación que permitan realizar búsquedas eficientes sobre ellos. Durante los últimos años se han propuesto diversas estructuras que tratan de llegar a un buen compromiso entre el espacio ocupado por el texto y el índice, y la eficiencia de la operación de búsqueda.

El wavelet tree es un autoíndice organizado en una estructura de árbol, diseñado originalmente para indexar los carácteres del texto. En este artículo presentamos posibles variantes del método indexando palabras en lugar de carácteres y utilizando distintos esquemas de codificación como base para la construcción del árbol. Estos cambios sobre la estructura original mejoran sus carácterísticas y su eficiencia en la operación de búsqueda.

*Trabajo parcialmente financiado por: Ministerio de Educación e Ciencia (PGE y FEDER) ref. TIN2006-16071-C03-03 y (Programa FPU) ref. AP-2006-03214 (para el último autor); y Xunta de Galicia ref. PGIDIT05SIN10502PR y ref. 2006/4

1. Introducción

El desarrollo de bases de datos documentales y bibliotecas digitales que manejan grandes colecciones de textos ha dado lugar a la necesidad de técnicas de indexación de textos que permitan realizar búsquedas eficientes en estas colecciones. Los índices invertidos son la técnica clásica de indexación de textos. Un índice invertido asocia a cada una de las palabras del texto la lista de punteros a sus ocurrencias en el texto. El problema más importante de los índices invertidos es el espacio que requieren, que puede ser hasta cuatro veces el tamaño del texto, además del espacio necesario para almacenar el propio texto (ya que es muy difícil reproducirlo a partir del índice). Por este motivo, en los últimos años se han propuesto distintas estructuras que tratan de mejorar este aspecto, llegando a un buen compromiso entre el espacio ocupado por el índice y la eficiencia en las búsquedas.

Una de las ideas que se ha seguido es la de utilizar índices comprimidos. Gracias a los desarrollos en técnicas de compresión de textos [4], se han desarrollado técnicas de compresión de los índices invertidos, de modo que el texto y el índice invertido pueden ocupar el

doble la entropía del texto H_0 (siendo la entropía el número mínimo de bytes al que se podría reducir usando la mejor técnica de compresión estadística, la compresión de Huffman [6]). Como se demuestra en [8], la codificación de palabras en lugar de carácteres para la compresión del texto puede suponer que el ratio de compresión pase del 60 % a un 25 % o 30 %. La idea de usar palabras como alfabeto en lugar de carácteres se basa en su distribución estadística. Las palabras tienen una distribución más sesgada, por eso al trabajar con técnicas de compresión estadística se logra esta importante mejora en el ratio de compresión. Además, este enfoque tiene la ventaja de utilizar como pieza base la misma que se usa para indexación orientada a búsqueda: las palabras. Los índices invertidos a bloque sobre texto comprimido son un ejemplo que combina indexación y compresión de textos [11].

Los autoíndices siguen una aproximación distinta para reducir el espacio ocupado. Un índice comprimido saca partido de la compresibilidad del texto. Así, requiere un espacio proporcional al del texto comprimido (por ejemplo, unas dos veces el tamaño de este). Un autoíndice es un índice comprimido que evita la necesidad de almacenar el propio texto además del índice. Contiene información suficiente para reproducir cualquier parte del texto a partir del índice de forma eficiente, reemplazando así el texto. Un wavelet tree [5] es un autoíndice organizado como un árbol binario, diseñado originalmente para indexar los carácteres del texto. En este trabajo presentamos tres variantes del wavelet tree original. Todas ellas están basadas en la idea de construir el árbol a partir de los códigos asociados a cada carácter/palabra del texto con distintos esquemas de codificación para compresión de textos.

El resto del artículo se estructura como sigue: la Sección 2 describe brevemente las distintas técnicas de codificación para la compresión de textos utilizadas en la construcción del wavelet tree. La Sección 3 presenta la estructura original de los wavelet trees. Las Secciones 4, 5 y 6 describen las distintas variantes de la estructura que se obtienen al combinar indexación de palabras o carácteres con los códigos de las distintas codificaciones utilizadas. La Sección 7 finaliza con las conclusiones y trabajo futuro.

2. Técnicas de compresión de textos

Esta sección describe brevemente los esquemas de codificación para compresión de textos que se utilizarán como base para las mejoras del wavelet tree original presentadas en las siguientes secciones, en particular, la compresión Huffman [6, 8, 9] y End-Tagged Dense Code [3, 1, 2].

2.1. Compresión Huffman

La codificación Huffman es quizás la técnica de codificación más conocida, siendo utizada ampliamente en numerosos ámbitos desde su aparición en [6]. Su aplicación inicial a la compresión de textos en lenguaje natural, se basa en codificar los distintos carácteres que contiene el texto, asignándoles a cada uno de ellos un código binario (secuencia de bits) que será tanto más corto, cuanto más frecuente sea el carácter. El proceso de asignación de un código a cada símbolo (carácter) de entrada, se basa en la construcción de un árbol de Huffman. Dicho proceso consta de las siguientes fases:

- Ordenar todos los símbolos de entrada por frecuencia. Dichos símbolos darán lugar a nodos hoja en el árbol Huffman, que irán siendo añadadidos al árbol de menor a mayor frecuencia.

- Elegir los 2 símbolos de menor frecuencia, y colocarlos como nodos hijos de un nuevo nodo interno p; p tendrá como frecuencia la suma de las frecuencias de sus hijos.

- Mientras queden nodos elegibles (nodos sin padre, ya sean hojas asociadas a los símbolos de entrada, o los nodos internos), extraer los 2 nodos de menor frecuencia y repetir el proceso anterior. Al final el último nodo en elegir será la raíz del árbol Huffman.

- Etiquetar recursivamente todas las ramas del árbol desde la raíz hasta las hojas. Las ramas izquierdas recibirán un '0' y las derechas un '1'. De este modo, el código asociado a cada símbolo, vendrá dado por el camino desde la raíz hasta el nodo hoja que lo contiene.

Como hemos comentado, este proceso asegura que se asignan códigos más pequeños a los símbolos más frecuentes, obteniendo una codificación óptima libre de prefijo. Un código no puede ser prefijo de un código más largo. Esta es una propiedad interesante pues da lugar a códigos decodificables instantáneamente (sin tener que mirar los siguientes códigos del texto comprimido). En la Figura 2 se muestran los códigos obtenidos mediante codificación Huffman, a partir del texto: "`bella rosa rosa, ¿bella?¿rosa?.`", así como la secuencia comprimida a que da lugar.

Desafortunadamente la compresión alcanzada al aplicar Huffman a textos en lenguaje natural era pobre (60 %). Las versiones orientadas a la codificación de palabras en lugar de carácteres mejoraron dicha compresión hasta aproximadamente un 25 %. Dicha codificación se hará siguiendo el algoritmo general para la construcción de un árbol Huffman explicado previamente. Aunque en este caso el vocabulario es más grande, el texto se comprime más debido a la distribución estadística de las palabras.

El proceso descompresión es muy sencillo. Suponiendo un texto comprimido como una secuencia binaria de entrada, se partirá de la raíz del árbol, y cada vez que leamos un bit nos moveremos dependiendo de su valor a izquierda o derecha. Cuando hayamos llegado a una hoja, habremos decodificado un símbolo.

El interés de mostrar aquí el funcionamiento de la codificación Huffman, es su utilización como base para la construcción de wavelet trees sobre texto comprimido, como veremos en las Secciones 4 y 5.

2.2. End-Tagged Dense Code

End-Tagged Dense Code (ETDC) es una técnica de compresión estadística basada en palabras y orientada a bytes; esto es, donde cada código está formado por una secuencia de bytes. De este modo, cada símbolo codificado recibirá como código uno o más bytes, de forma que se asignen códigos más cortos a los símbolos más frecuentes. ETDC utiliza el primer bit de cada byte como marca de fin de código. Dicha marca permite distinguir qué bytes dentro de un código son los últimos del código, y cuales no. En particular, ETDC marca a 1 el bit más significativo del último byte de cada código, y a 0 el bit más significativo de los restantes bytes.

El hecho de marcar el fin de un código tiene importantes consecuencias, respecto a otras técnicas basadas en Huffman (p. ej: Tagged Huffman [9]). El bit de marca al final del código incorpora automáticamente la carácterística de prefijo libre a la codificación, independientemente del resto de los bits de cada byte en el código. Dado que únicamente el bit de marca del último byte de cada código vale 1, ningún código podrá ser prefijo de otro. Es decir, dados dos códigos X e Y, si la longitud (el número de bytes) de X ($long_X$) es menor a la de Y, X nunca podrá ser prefijo de Y porque el último byte de X tiene su bit de marca a 1, mientras que el $long_X$-ésimo byte de Y tiene su bit de marca a 0.

Dado que el bit de marca garantiza la propiedad de prefijo libre, ya no es necesario aplicar Huffman para garantizar códigos libres de prefijo. Por lo tanto, para la codificación se pueden utilizar todas las combinaciones de valores posibles para cada byte, sin más que tener en cuenta el bit de marca. El proceso de codificación se realiza como sigue (asumiendo "bytes" de 8 bits):

- Ordenar las palabras distintas del texto (el vocabulario) por orden decreciente de frecuencia.
- Asignar códigos de 1 byte a las 128 primeras palabras. De este modo dichas palabras recibirán respectivamente los códigos 128 a 255, o lo que es lo mismo: códigos de (**1**0000000 hasta **1**1111111).
- Las palabras en las posiciones desde la $2^7 = 128$ a la $2^7 2^7 + 2^7 - 1 = 16511$ reciben

Pos. Palabra	código asignado
0	**1**0000000
1	**1**0000001
2	**1**0000010
...	...
$2^7 - 1 = 127$	**1**1111111
$2^7 = 128$	**0**0000000:**1**0000000
129	**0**0000000:**1**0000001
130	**0**0000000:**1**0000010
...	...
255	**0**0000000:**1**1111111
256	**0**0000001:**1**0000000
257	**0**0000001:**1**0000001
258	**0**0000001:**1**0000010
...	...
$2^7 2^7 + 2^7 - 1$	**0**1111111:**1**1111111
$2^7 2^7 + 2^7 = 16512$	**0**0000000:**0**0000000:**1**0000000
16513	**0**0000000:**0**0000000:**1**0000001
...	...

Cuadro 1: Asignación de códigos en ETDC.

secuencialmente códigos de dos bytes. El primer byte de cada código contendrá un valor entre 0 y $2^7 - 1$, y el segundo byte recibirá un valor entre 128 y 255.

- El proceso continuaría asignando secuencialmente códigos al resto de las palabras del vocabulario.

El Cuadro 1, resume el proceso de codificación secuencial, nótese que la longitud del código asignado a una palabra no depende de su frecuencia directamente (como en Huffman), sino de la posición en el vocabulario.

Como se ha comentado previamente, en la Sección 6 mostraremos la segunda de nuestas propuestas, un autoíndice que se basa en la construcción de un wavelet tree, partiendo de una codifición (ETDC) del texto a indexar.

3. Indexación de textos mediante wavelet trees

Un wavelet tree es originalmente [5] un árbol de búsqueda binario balanceado, donde cada símbolo del alfabeto $\Sigma = \{s_1, s_2, \ldots, s_\sigma\}$ se corresponde con un nodo hoja. En el caso de textos, Σ contiene los carácteres que aparecen en el texto a indexar.

El proceso de construcción de un wavelet tree es el siguiente. La raíz de este árbol contiene un vector de bits B, de la misma longitud que el texto (n), donde se marcan a 1 todas aquellas posiciones que corresponden a los carácteres que aparecen en la segunda mitad del alfabeto: $\{s_{\sigma/2+1}, \ldots, s_n\}$. Los carácteres de la primera mitad del alfabeto $\{s_1, \ldots, s_{\sigma/2}\}$, se corresponderán con posiciones marcadas con 0 en el vector de bits B. De este modo, los 1's corresponderán a carácteres que, concatenados, formarán la secuencia que se tratará en el hijo derecho de la raíz. Mientras tanto, los carácteres que dieron lugar a posiciones marcadas con 0 en B, dan lugar a la secuencia de carácteres que se asociará con su hijo izquierdo. Este proceso de creación de un mapa de bits para cada nodo, se repetirá recursivamente (dividiendo el número de carácteres que se indexaban en cada nodo a la mitad al pasar a sus hijos) hasta llegar a las hojas. Finalmente, el texto original podrá ser descartado, ya que quedará representado por los mapas de bits contenidos en cada nodo interno, junto con el alfabeto utilizado. La figura 1 muestra un wavelet tree construido sobre el texto "`la_cabra_abracadabra`", siendo el alfabeto $\Sigma = \{\_, a, b, c, d, l, r\}$.

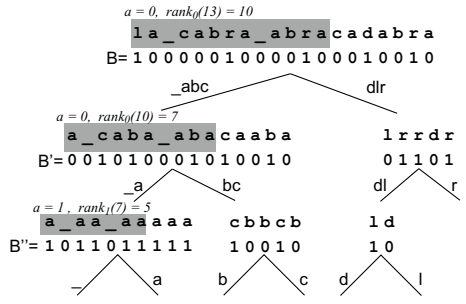

Figura 1: Indexación de texto con wavelet trees.

Una vez construido el wavelet tree, ya puede utilizarse para obtener el símbolo de una posición cualquiera del texto original (necesario para reproducir el texto) y para buscar cualquier ocurrencia de un símbolo en el texto. Para contestar a esas preguntas necesitaremos ser capaces de resolver dos operaciones básicas sobre un vector de bits: *rank* y *select*.

3.1. Funciones básicas para el manejo de vectores de bits

Aunque en principio nos interesa comentar las operaciones *rank* y *select* sobre secuencias de bits, se pueden describir dichas operaciones de un modo más general como sigue. Sea $B = b_1 b_2 \cdots b_n$ una secuencia de símbolos:

- $rank_b(B, i) = y$, si el símbolo b aparece y veces en la secuencia $B_{1,i}$.
- $select_b(B, j) = x$, si la j-ésima aparición del símbolo b en una secuencia B ocurre en la posición x.

Según estas definiciones, Dada una secuencia binaria $B = 1000110$, $rank_1(B, 5)$ devolvería el valor 2, pues en hasta la posición B_5 (incluida) hay dos '1'. Por otro lado, $select_0(B, 4)$, o lo que es lo mismo, buscar la 4^{a} ocurrencia de '0', devolvería la posición 7.

Encontrar una solución eficiente para la operación *rank* es un problema abierto que está siendo punto de interés en la actualidad [7, 10]. Aunque en el caso de secuencias de bits se ha llegado a optimizaciones muy importantes, el caso general sigue siendo complicado.

3.2. Recuperación de texto y búsquedas

Este índice puede utilizarse para obtener el número de ocurrencias de un símbolo en el texto (*count*), para buscar una ocurrencia dada de un símbolo (*locate*) o para obtener el símbolo de una posición cualquiera del texto (*display*).

Display: Supongamos que queremos recuperar el símbolo de la posición i del texto. El bit B_i del mapa de bits B de la raiz determina si este símbolo está indexado en el hijo izquierdo ($B_i = 0$) o derecho ($B_i = 1$) de la raiz. Además, $rank_{B_i}(B, i)$ nos da la posición correspondiente a este carácter en el mapa de bits del nodo hijo. Este proceso se repite hasta llegar a una hoja que nos da el carácter deseado. Por ejemplo, supongamos que queremos recuperar el carácter de la posición 13 del texto indexado en la figura 1 (nótese que en la figura, el texto de cada nodo se muestra sólo por claridad en el ejemplo, ya que realmente no se almacena). Empezando en la raiz del árbol, $B_{13} = 0$ nos indica que este carácter está indexado en la rama izquierda del árbol, y $rank_0(B, 13) = 10$ significa que la posición 10 del mapa de bits B' del nodo hijo corresponde a este carácter. En el siguiente nivel $B'_{10} = 0$ (moverse al hijo izquierdo) y $rank_0(B', 10) = 7$ (la posición del carácter en el bitmap B''). Al descender al siguiente nivel, $B''_7 = 1$ (moverse al hijo derecho) y $rank_1(B'', 7) = 5$. Puesto que el hijo derecho es un nodo hoja (el correspondiente al carácter '`a`'), sabemos que la posición 13 del texto original contiene la ocurrencia 5 del carácter '`a`'. Este proceso puede utilizarse para recuperar el texto completo (un carácter a la vez) a partir de la información contenida en la estructura del índice.

Count y Locate: La utilización del wavelet tree para las búsquedas comienza en el nodo hoja correspondiente al carácter que estamos buscando (este nodo hoja es fácil de localizar a partir de la posición del carácter en el alfabeto Σ) y recorre el árbol desde la hoja hasta la raiz. Supongamos que un nodo hoja está representado por un camino desde la raiz $\overline{b_0 b_1 \dots b_k}$, y que $B, B', \dots, B^k$ son los mapas de bits almacenados en los nodos de dicho camino. La operación *count* se implementa simplemente calculando $rank_{b_k}(B^k, |B^k|)$. Para recuperar la ocurrencia i-ésima de un carácter c, cuyo nodo hoja correspondiente es $\overline{b_0 \dots b_k}$ (*locate*), la búsqueda se lleva a cabo como sigue. Calculando $i_k = select_{b_k}(B^k, i)$ obtenemos que i_k es la posición de la i-ésima ocurrencia de c en B^k. Repetimos este proceso (en el nivel superior) obteniendo $i_{k-1} = select_{b_{k-1}}(B^{k-1}, i_k)$ para movernos al siguiente nivel en el árbol, y así hasta llegar a la raiz. El último $i_0 = select_{b_0}(B^0, i_1)$ nos da finalmente la posición de la i-ésima ocurrencia del carácter c en el texto.

Por ejemplo, si queremos buscar la ocurrencia 4 del carácter a en el ejemplo de la figura 1, la búsqueda comenzará en el nodo hoja $\overline{001}$, que corresponde a este símbolo. En este nodo calculamos $i_2 = select_1(B'', 4) = 6$ y nos

movemos al nodo padre. En el nodo $\overline{0\underline{0}}$ obtenemos $i_1 = select_0(B', 6) = 8$ y continuamos al siguiente nivel. Finalmente, en el nodo $\overline{\underline{0}}$ calculamos $i_0 = select_0(B, 8) = 10$, que nos dice que el carácter que estamos buscando aparece en la posición 10 del texto.

4. Wavelet tree y codificación Huffman orientada a carácter

Aunque el wavelet tree mostrado en la sección anterior permitía indexar los carácteres del texto, dando lugar a un árbol balanceado, también es posible partir de la codificación que Huffman clásico asociaría a los carácteres que componen dicho texto, y realizar la construcción del wavelet tree sobre dichos códigos. La idea es que el código asociado a un carácter indique su posición en el wavelet tree. En principio, el wavelet tree construido ya no será necesariamente un árbol balanceado, pero a cambio se conseguirá reducir el espacio necesario para su almacenamiento (lo cual resulta evidente, ya que el camino hasta llegar a un símbolo situado en una hoja será ahora menor). Además, el camino para llegar a un carácter será más pequeño para los carácteres más frecuentes.

El proceso de construcción asigna a la raíz del wavelet tree un mapa de bits B que tiene el mismo número de bits que el número de carácteres del texto original (n). De forma que $B_i = 0$, si el primer bit del código Huffman asociado al i-ésimo carácter del texto es '0'. Si el dicho bit es '1', entonces $B_i = 1$. A continuación se repite dicho proceso, en los hijos izquierdo y derecho de la raíz (aunque teniendo en cuenta el 2º bit del código asociado a cada carácter). Todos los carácteres del texto cuyo código comenzaba por 0 se indexarán a través del hijo izquierdo, y aquellos cuyo código comenzaba por 1 se tratarán en el hijo derecho. El proceso continuará hasta llegar a las hojas.

En la Figura 2 se puede observar un ejemplo de construcción de un wavelet tree sobre la codificación Huffman clásica de un texto. En la parte superior se muestran los códigos Huffman asociados a cada carácter del texto. La parte inferior muestra el wavelet tree obtenido. Es interesante observar cómo el código de cada carácter determina su posición en el wavelet tree. Por ejemplo, como los códigos correspondientes a 'S', '_', 'L' Y 'A' comienzan por "1", el mapa de bits B' del hijo derecho de la raíz contendrá la secuencia "111001001011101", donde cada uno de los bits se corresponde con el 2º bit del código de dichos carácteres con respecto al texto original.

El proceso de búsqueda es análogo al mostrado en la Sección 1, partiendo de la hoja asociada a un carácter y aplicando *select* reiteradamente hasta llegar al nodo raíz. Además ahora es también necesario un proceso de decodificación (similar también al proceso de recuperar un carácter mostrado en la sección anterior), que recorriendo el árbol de arriba a abajo (y aplicando *rank* en cada nivel) permitirá obtener el carácter que estaría inicialmente en la posición i del texto.

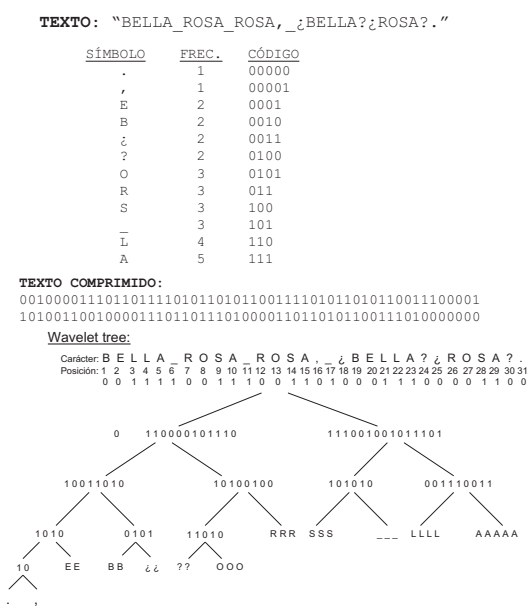

Figura 2: Wavelet tree sobre codificación Huffman orientada a carácteres.

5. Wavelet tree y codificación Huffman orientada a palabra

Siguiendo la misma idea que llevó de compresión orientada a carácteres a compresión orientada a palabras, en esta sección proponemos una primera aproximación que aproveche la ventaja de utilizar palabras en lugar de carácteres, como los símbolos a indexar. Esta primera aproximación consiste en la creación de un wavelet tree utilizando codificación Huffman basada en palabras y orientada a bit como esquema de codificación.

El proceso de construcción del nuevo wavelet tree, así como los métodos para decodificar y buscar, serán análogos a los mostrados en la sección anterior. Sin embargo, ahora los mapas de bits almacenados en cada nivel harán referencia a los códigos asociados a las palabras en lugar de a carácteres.

La Figura 2 muestra un wavelet tree orientado a palabras y bits que indexa el mismo texto mostrado en la Sección 4, aunque en este caso, la construcción tiene en cuenta los códigos Huffman asociados a cada palabra del texto. En la parte superior de dicha figura se muestran también los códigos Huffman asignados a dichas palabras.

La ventaja de esta nueva forma de construcción del wavelet tree está fundada en lo siguiente: la utilización de palabras reduce el tamaño del texto comprimido, y por tanto reducirá también el tamaño de los vectores de bits que habrá que almacenar en cada nivel del wavelet tree. Por ello, y dado que la codificación Huffman basada en palabras y orientada a bits es óptima [8], se espera reducir prácticamente a la mitad el tamaño (en bytes) del wavelet tree mostrado en la Sección 4. El principal inconveniente de la nueva aproximación es que el wavelet tree tendrá generalmente una altura mayor (hay muchos más símbolos en el alfabeto). La longitud media de los códigos asociados a las palabras se sitúa en torno a los 8-10 bits. Sin embargo, para palabras con un número de ocurrencias muy bajo podremos encontrarnos códigos mucho más largos (entre 20-25 bits). En consecuencia, la eficiencia de esta versión podría resentirse (al ser alto el número de *ranks* y/o *selects* que pueden ser necesarios). Por ello, parece mucho más eficiente plantearse implementación de un "wavelet tree" orientado a byte.

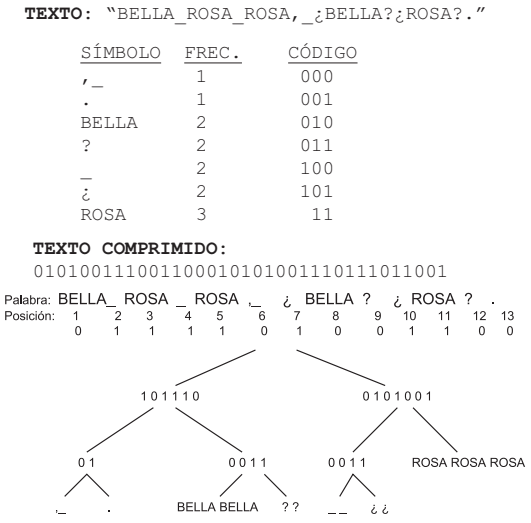

Figura 3: Wavelet tree sobre Huffman orientado a bits y a palabras.

6. Wavelet tree y codificación End-Tagged Dense Code

Establecemos de nuevo un claro paralelismo con las técnicas de compresión, dónde se vio que la utilización de una codificación orientada a byte en lugar de a bit [9, 3, 2] hacía mucho más rápido tanto el proceso de descompresión como el de búsqueda, a costa de perder aproximadamente un 5 % en ratio de compresión. En esta sección se describe una nueva variante del wavelet tree. La idea es indexar las palabras de un texto, y construir el wavelet tree en base a la codificación asignada a las palabras mediante End-Tagged Dense Code (ETDC).

Como ETDC genera códigos formados por secuencias de bytes, el wavelet tree presenta varias novedades. Al estar orientado a byte en lugar de a bit, cada nodo presenta tantas ramas hijas como valores distintos de bytes exis-

tan (256). De estas 256 ramas, las 128 últimas contendrán nodos hoja, mientras que las 128 primeras darán lugar a nodos intermedios. En el primer nivel del wavelet tree habrá por tanto 128 hojas, en el segundo nivel habrá 128^2, en el tercero 128^3, etc. En general, para un texto en lenguaje natural, ETDC nunca necesitará utilizar códigos de longitud ≥ 4 [2], por lo que el wavelet tree tendrá como máximo altura 3.

La construcción del wavelet tree es similar a la mostrada en la sección anterior. Sin embargo, cada nodo contendrá ahora una secuencia de bytes en lugar de un mapa de bits. De este modo, el nodo raíz contendrá un vector de bytes correspondiente al primer byte de los códigos de cada una de las palabras que aparecen en el texto original. Como ya hemos dicho, el nodo raíz contiene 256 hijos. Los 128 últimos son hojas (y están por ello asociados a palabras), mientras que los 128 primeros son nodos intermedios que, a su vez, contienen nuevos vectores de bytes. El primero de ellos (hijo en posición 0) es asociado a todas las palabras cuyo código contenga un '0' en el primer byte, y almacena como "secuencia de bytes" los segundos bytes del código de dichas palabras. Análogamente, el segundo hijo de la raíz (posición 1) almacena una secuencia de bytes formada por el segundo byte de los códigos asociados de las palabras del texto original cuyo código comienza por un byte con valor '1'. Este proceso se aplica a todos los demás nodos hijos de la raíz, y (como en las versiones previas), recursivamente (de arriba a abajo) hasta llegar a los nodos hoja.

La Figura 4 muestra un ejemplo de construcción de un wavelet tree usando ETDC como esquema de codificación de palabras. Además, en la parte superior de dicha figura se pueden observar los códigos asignados a cada una de las palabras que componen el texto de entrada. Supongamos que queremos encontrar la posición de la segunda ocurrencia de la palabra `BELLA`. Comenzamos por la hoja que corresponde a esta palabra, a la que se llega por una rama con código 100 (por ser 100 el último byte del código de `BELLA`). Al estar buscando su segunda ocurrencia calculamos $select_{100}(B, 2)$ en el vector de bytes del nivel anterior. Como resultado obtenemos la posición 3. A este nodo se llega por la rama 000 (primer byte del código de `BELLA`), por lo que tenemos que calcular $select_{000}(B, 3)$ en el vector de bits del nivel anterior (la raíz en este caso). Obtenemos la posición 8, que es la posición en el texto de la segunda ocurrencia de la palabra `BELLA`.

Supongamos que nos interesa buscar la palabra de la posición 13 del texto. Empezamos en la raíz del árbol aplicando $rank_{000}(B, 13)$, y obtenemos 4. Bajamos al siguiente nivel por la rama 000. Como en este nivel a la posición 4 le corresponde 101, calculamos $rank_{101}(B, 4)$ sobre el vector de bits B de este nivel, obteniendo 1. Finalmente, descendemos al siguiente nivel por la rama 101, llegando a una hoja, que corresponde a la palabra " .".

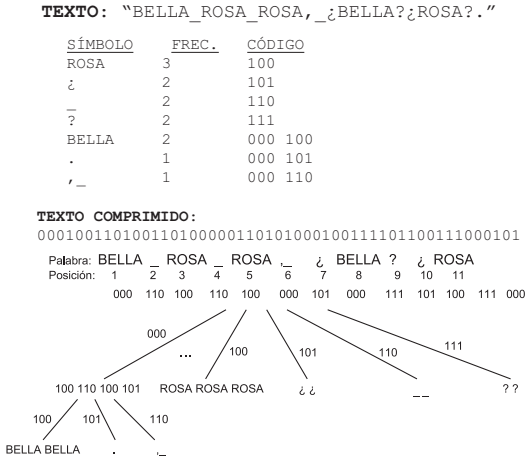

Figura 4: Wavelet tree sobre ETDC.

El proceso de búsqueda y decodificación del texto de nuevo vuelve a basarse en la aplicación de *rank* (para decodificar) y *select* (para buscar). La diferencia respecto a los wavelet trees anteriores es que ahora es necesario disponer de versiones de *rank* y *select* de bytes en lugar de bits. Esto es, dada un secuencia de bytes D y una posición de esa secuencia i, ser capaz de contar el número de ocurrencias de un valor $d \in [0, 255]$ ($rank_d(D, i)$); o

bien, a partir de un valor $d \in [0, 255]$ y una secuencia de bytes D, saber en qué posición j de D aparece la k-ésima ocurrencia de d ($j = select_d(D, k)$). Actualmente existen versiones poco eficientes de estas dos funciones sobre bytes, por lo que es un interesante trabajo de investigación, realizar versiones más eficientes de las mismas.

7. Conclusiones y trabajo futuro

En este trabajo presentamos varias mejoras del wavelet tree basando la construcción del árbol en los códigos asociados a cada símbolo por códigos de compresión. El hecho de crear el árbol sobre estos códigos reduce su tamaño y hace más eficiente las operaciones de búsqueda y reproducción del texto a partir del índice. Además, cuando se codifican palabras en lugar de carácteres, aunque el árbol sea más profundo, sólo es necesario recorrerlo una vez para buscar una palabra. Así, estas modificaciones a la estructura original suponen una mejora en el compromiso espacio-eficiencia en la indexación de textos.

Referencias

[1] Nieves Brisaboa, Antonio Fariña, Gonzalo Navarro, and M. Esteller. (s,c)-dense coding: An optimized compression code for naturallanguage text databases. In *Proceedings of the 10th International Symposium on String Processing and Information Retrieval (SPIRE 2003)*, LNCS 2857, pages 122–136. Springer, 2003.

[2] Nieves Brisaboa, Antonio Fariña, Gonzalo Navarro, and Jose R. Paramá. Lightweight natural language text compression. *Information Retrieval*, 10:1–33, 2007.

[3] Nieves Brisaboa, Eva Iglesias, Gonzalo Navarro, and Jose R. Paramá. An efficient compression code for text databases. In *Proceedings of the 25th European Conference on Information Retrieval Research (ECIR'03)*, LNCS 2633, pages 468–481, 2003.

[4] J. Shane Culpepper and Alistair Moffat. Enhanced byte codes with restricted prefix properties. In *Proceedings of SPIRE 2005: 12th International Conference String Processing and Information Retrieval*, volume 3772 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2005.

[5] R. Grossi, A. Gupta, and J.S. Vitter. High-order entropy-compressed text indexes. In *Proceedings of 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 03)*, pages 841–850, 2003.

[6] D. A. Huffman. A method for the construction of minimum-redundancy codes. In *Proc. Inst. Radio Eng.*, pages 1098–1101, September 1952. Published as Proc. Inst. Radio Eng., volume 40, number 9.

[7] V. Mäkinen and Gonzalo Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 2006. Special issue on "The Burrows-Wheeler Transform and its Applications". To appear.

[8] Alistair Moffat. Word-based text compression. *Software - Practice and Experience*, 19(2):185–198, 1989.

[9] E. Moura, Gonzalo Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems (TOIS)*, 18(2):113–139, 2000.

[10] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):1–66, 2006.

[11] Nivio Ziviani, Edleno Silva de Moura, Gonzalo Navarro, and Ricardo Baeza-Yates. Compression: A key for next-generation text retrieval systems. *IEEE Computer*, 33(11):37–44, 2000.