UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

INDEXACIÓN COMPRIMIDA DE IMÁGENES

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

DANIEL ALEJANDRO VALENZUELA SERRA

PROFESOR GUÍA:
GONZALO NAVARRO BADINO

MIEMBROS DE LA COMISIÓN:
BENJAMÍN BUSTOS CÁRDENAS
PATRICIO INOSTROZA FAJARDIN

SANTIAGO DE CHILE
ABRIL 2009

Este trabajo ha recibido el apoyo del proyecto Fondecyt 1-080019

RESUMEN DE LA MEMORIA
PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN
POR: DANIEL VALENZUELA S.
FECHA: 21/04/2009
PROF. GUÍA: GONZALO NAVARRO B.

INDEXACIÓN COMPRIMIDA DE IMÁGENES

El continuo aumento de los volúmenes de información almacenada digitalmente ha fomentado el desarrollo de técnicas para brindar acceso y búsqueda de manera eficiente a los datos. En particular, el manejo de grandes colecciones de imágenes es un problema de gran interés. Un enfoque es tratar las imágenes como secuencias de texto bidimensional. En este contexto, han sido planteadas recientemente dos estructuras de autoindexación para colecciones de imágenes, basadas en extender autoíndices de texto unidimensional. Estas estructuras almacenan la colección en espacio proporcional al requerido para almacenar la colección comprimida, permitiendo a la vez el acceso directo a cualquier parte de la colección y la búsqueda eficiente de patrones en ella. Dos tipos de autoíndices para secuencias de texto son el Arreglo de Sufijos Comprimido y el Índice FM, y en ellos se basan las soluciones para imágenes.

Este trabajo se centra en la implementación de esos dos autoíndices para imágenes. Se implementaron distintas variantes para ambas estructuras buscando la mejor manera de adaptarlas a secuencias bidimensionales, y mejorando significativamente varios de los algoritmos originales. Finalmente se diseñaron y ejecutaron experimentos para comparar las distintas variantes de ambos índices, tanto en términos de espacio requerido por las estructuras, como de tiempo en responder las consultas de acceso y búsqueda de patrones. Las estructuras basadas en el Arreglo de Sufijos Comprimido resultaron mejores en cuanto a tiempo, mientras que aquellas basadas en el Índice FM resultaron mejores en términos de espacio requerido, cuando el rango de colores es pequeño. Por ejemplo, con el Arreglo de Sufijos Comprimido somos capaces de almacenar una colección utilizando un 80 % del espacio que requeriría la representación plana, pudiendo dar acceso a cualquier subimagen a una velocidad aproximada de 1 megapixel por segundo. Con esta estructura somos capaces de contar las ocurrencias de un patrón a una velocidad aproximada de $0,5$ megapixeles por segundo, y podemos localizar la posición de cada ocurrencia en menos de $0,1$ milisegundo. Sobre colecciones con un rango de color más pequeño, utilizando el Índice FM podemos alcanzar niveles de compresión del 50 %, pudiendo llegar al 25 % si aceptamos pérdida de información. Con esta estructura podemos acceder a cualquier subimagen y realizar la búsqueda de patrones a una velocidad de $0,1$ megapixel por segundo.

Agradecimientos

En primer lugar agradezco a mis padres, por todo el cariño y el incondicional apoyo que me han brindado. Sin ustedes no sería lo que soy hoy.

Quiero también agradecer al profesor Nelson Zamorano y a la Escuela de Verano, por ser los responsables de expandir mis horizontes en la adolescencia, acercarme a la U y más tarde permitirme un primer y muy valioso acercamiento a la docencia.

Una mención importante recae también sobre los miembros y colaboradores del Grupo Miércoles de Algoritmos, por generar un grato espacio de trabajo y camaradería. En particular quiero agradecer a Francisco Claude y Antonio Fariña por su ayuda que sin duda contribuyó al desarrollo de este trabajo.

Finalmente quiero agradecer el generoso e incalculable apoyo del profesor Gonzalo Navarro, que no sólo me ha permitido concluir felizmente esta memoria, sino que también me ha permitido ver que este es sólo un primer paso.

Índice General

Capítulo 1

Introducción

Día a día vemos cómo el volumen de información digital crece exponencialmente. Desde el genoma humano hasta repositorios de imágenes del firmamento, pasando por grandes bases de datos de proteínas, bibliotecas completas, registros de transacciones bancarias, y un largo etcétera se encuentran hoy en día almacenadas en computadores. El tener tamaña información disponible en formato digital supone grandes ventajas: La reducción en el espacio físico requerido, la facilidad para generar copias y distribuirlas (y por tanto, la permanencia en el tiempo), y el acceso a la información desde distintos lugares del mundo, gracias al desarrollo de las redes de comunicación.

Sin embargo, el manejo de estos grandes volúmenes de información conlleva también el desafío de dar acceso y búsqueda a la información de manera rápida y eficiente.

Una de las técnicas más importantes para poder manejar de manera eficiente grandes colecciones son los índices. Cuando queremos buscar o acceder a datos en una colección muy grande, una búsqueda secuencial podría tomar tiempos demasiado largos. Por ejemplo, sabemos que el contenido de la Web es hoy de miles de terabytes (de texto, imágenes, audio y video). Si intentáramos realizar una búsqueda de manera secuencial, ésta tardaría días. La solución a este problema está en los índices, estructuras adicionales a los datos que almacenamos, que permiten responder las búsquedas en tiempos mucho menores que una búsqueda secuencial. Por ejemplo, para la Web se utilizan los llamados *índices invertidos*, que almacenan una lista con las ocurrencias de cada palabra. Así, cuando uno busca una frase, el resultado es la intersección de las listas asociadas a cada palabra.

En el área de indexación de texto ha surgido en los últimos años una nueva posibilidad: Dada una secuencia de caracteres, es posible comprimir la secuencia obteniendo una

representación comprimida que funcione a su vez como un índice. Las funcionalidades de indexación de estas estructuras pueden ir desde acceso aleatorio a elementos de la secuencia original hasta capacidad de búsqueda de patrones de manera rápida. Estas estructuras son frecuentemente llamadas autoíndices, pues reemplazan a la secuencia original y pueden por lo tanto ser consideradas como compresión con valor agregado. Las tres familias principales de autoíndices son el Índice FM, el Arreglo de Sufijos Comprimido, y el índice LZ.

La búsqueda sobre grandes colecciones de imágenes es otro gran desafío y existen distintos enfoques para abordarlo. Principalmente, podemos distinguir la búsqueda por concepto, y la búsqueda por contenido (además de estrategias mixtas que combinan ambos enfoques). Para la búsqueda por concepto necesitamos alguna categorización de las imágenes, por ejemplo mediante etiquetas, título, o alguna otra manera de obtener una taxonomía de la colección. Algunos buscadores, por ejemplo, indexan las imágenes asociando a ellas las palabras que están escritas cerca de ellas.

La búsqueda por contenido, en tanto, se centra en propiedades visuales cuantificables de la imagen, como colores, tamaño, etc. Dentro de ésta también existen distintos tipos: búsqueda aproximada, búsqueda por similitud, búsqueda exacta, permitiendo rotaciones, etc. Existen dos enfoques principales para este tipo de búsquedas: el enfoque clásico consistente en tratar la imagen como una señal mediante funciones, como la *Transformada Rápida de Fourier* y el enfoque combinatorial o discreto, en el cual la imagen es tratada como una secuencia bidimensional formada por la intensidad de color de cada píxel.

En este contexto, existe ya un amplio desarrollo en cuanto a búsqueda de patrones en secuencias bidimensionales. Primeramente se desarrollaron algoritmos de búsqueda *on-line*, y al igual que en el caso de secuencias unidimensionales, se ha estudiado también la búsqueda indexada.

La posibilidad de generar autoíndices para secuencias bidimensionales resulta por tanto una interesante y natural dirección de desarrollo de estas técnicas: obtener una representación comprimida de una colección de imágenes que otorgue las funcionalidades de búsqueda exacta de patrones, así como de acceso eficiente a subimágenes. Es importante destacar que la búsqueda exacta no es sólo de interés teórico: búsquedas más realistas (aproximada, permitiendo rotaciones, etc.) se apoyan en la búsqueda exacta.

Recientemente se han planteado dos estructuras para generar autoíndices de colecciones de

imágenes [18]. En esta memoria se implementarán y experimentarán estas técnicas, buscando la mejor manera de llevarlas a la práctica.

1.1. Trabajo realizado

Este trabajo se centró en la implementación de dos autoíndices para imágenes, implementando distintas variaciones para cada uno, y en la comparación de eficiencia en cuanto a espacio requerido por las estructuras y el tiempo en responder las consultas soportadas por cada uno de ellos.

La primera parte del trabajo consistió en implementar el CSA-2D, una estructura recientemente planteada [18]. El principal foco de atención estuvo en implementar y estudiar distintas alternativas de codificación para este índice.

La segunda parte del trabajo consistió en la implementación de un autoíndice para imágenes basado en el Índice FM, indexando las filas de la imagen. Posteriormente se implementó una generalización de este último, en la cual ya no se indexaron las filas de a una, sino por grupos.

Finalmente, se realizaron experimentos con diferentes colecciones de imágenes intentando dar cuenta de posibles escenarios reales en los cuales estas técnicas sean de utilidad: Una colección de ilustraciones góticas, en formato RGB; una colección de mapas antiguos de japón, en formato RGB; una colección de fotos astronómicas, en formato RGB; una colección de imágenes obtenidas por un microscopio electrónico, en escala de grises; y una colección compuesta por algunas de las imágenes de la fuente *Identifont*, en escala de grises.

Los resultados obtenidos fueron bastante promisorios. La técnica basada en CSA tuvo los mejores tiempos tanto para acceso a subimágenes como para búsqueda de patrones, y los niveles de compresión alcanzados son competitivos en los casos en que el rango de colores es grande (RGB). Por otro lado, el índice basado en FM obtuvo excelentes niveles de compresión, sobre todo cuando el rango de colores es pequeño. Respecto a la versión general, se constató que el aumentar el tamaño de las bandas indexadas mejoró la compresión y los tiempos de este índice.

Capítulo 2

Conceptos Básicos

2.1. Entropía de un texto

Consideremos una secuencia de caracteres $T[1,n]$ sobre un alfabeto $\Sigma = \{1,\ldots,\sigma\}$ y sea n_c el número de ocurrencias del carácter c en T. Se define la entropía empírica de orden cero de T como:[1]

$$H_0(T) = \sum_{c\in\Sigma, n_c>0} \frac{n_c}{n} \log \frac{n}{n_c}$$

El valor $nH_0(T)$ representa el tamaño de la máxima compresión alcanzable si se utiliza un compresor que asigna una palabra de código fija a cada símbolo del alfabeto.

Es posible lograr una mejor compresión si el código utilizado para cada carácter depende también del contexto. Así, se define la entropía empírica de orden k de T como:

$$H_k(T) = \sum_{s\in\Sigma^k, T^s\neq\epsilon} \frac{|T^s|}{n} H_0(T^s)$$

donde T^s es la subcadena formada por todos los caracteres que ocurren seguidos por el contexto s en T. La entropía empírica de un texto T representa una cota inferior para el número de bits necesarios para comprimir T utilizando cualquier codificador que codifique cada carácter considerando sólo el contexto de k caracteres que lo siguen en T.

2.2. Rank y Select

Una estructura básica para el desarrollo de los autoíndices es la que dota a un arreglo de bits $B[1,n]$ de las funciones $rank(B,i) = rank_1(B,i)$, la cual entrega el número de unos en

[1]En este documento $\log = \log_2$, salvo que se explicite otra base.

el prefijo $B[1,i]$. Simétricamente, $rank_0(B,i) = i - rank_1(B,i)$. La consulta dual de $rank_1$ es $select(B,j)$, que entrega la posición del j-ésimo 1 en B.

Las primeras estructuras desarrolladas capaces de calcular *rank* y *select* en tiempo constante [21, 4] utilizan $n + o(n)$ bits: n bits del arreglo B y $o(n)$ bits adicionales para responder las consultas de *rank* y *select*. Trabajos posteriores [23] proponen estructuras para *rank* y *select* que permiten comprimir B de modo que ocupe $nH_0(B) + o(n)$ bits. Cotas inferiores obtenidas recientemente [20] muestran que estos resultados son casi óptimos.

Más aún, existen resultados experimentales [10, 22, 5] que muestran que en la práctica se puede obtener un buen desempeño en términos de tiempo y espacio extra.

Los conceptos de *rank* y *select* pueden ser definidos para un caso más general, sobre una secuencia de caracteres T definida sobre un alfabeto Σ. Así, $rank_c(T,i)$ es el número de ocurrencias del carácter c en $T_{1,i}$, mientras que $select_c(T,j)$ corresponde a la posición de la j-ésima ocurrencia de c en T. Los resultados para *rank* y *select* binario pueden generalizados utilizando arreglos de bits como indicadores, obteniendo una representación comprimida que soporte *rank* y *select* en tiempo constante, utilizando $nH_0(T) + O(n) + o(\sigma n)$ bits. Una estructura distinta para soportar la misma funcionalidad es el Wavelet Tree [11], el cual requiere $nH_0(T) + o(n \log \sigma)$ bits, y permite responder *rank* y *select* en tiempo $O(\log \sigma)$.

2.3. Arreglo de Sufijos

Consideremos una secuencia de caracteres $T[1,n]$. El *Arreglo de sufijos* $A[1,n]$ de T es un arreglo de punteros a todos los sufijos de T ordenados lexicográficamente [19]. Asumamos que T termina con un marcador único \$, de tal modo que las comparaciones lexicográficas están bien definidas. $A[i]$ apunta al sufijo $T[A[i],n] = t_{A[i]}t_{A[i]+1}\ldots t_n$, y se tiene que lexicográficamente $T[A[i],n] < T[A[i+1],n]$. El arreglo de sufijos requiere espacio $O(n \log n)$ bits, además del espacio ocupado por el texto mismo.

Dados A y T, las ocurrencias de un patrón $P = p_1p_2\ldots p_m$ en T se pueden contar en tiempo $O(m \log n)$. Para ello, basta notar que toda subcadena de T, $t_kt_{k+1}\ldots t_{k+m-1}$ corresponde a un prefijo de un sufijo de T, $t_kt_{k+1}\ldots t_n$. Como el arreglo de sufijos los ordena lexicográficamente, se tendrá que las ocurrencias del patrón corresponden a un intervalo contiguo en al arreglo de sufijos, $A[sp,ep]$ tal que todos los sufijos $t_{A[i]}t_{A[i]+1}\ldots t_n$ para todo $sp \leq i \leq ep$ contienen al patrón P como prefijo. Para encontrar el intervalo basta con

realizar dos búsquedas binarias. La primera búsqueda binaria determina la posición sp donde comienzan los sufijos lexicográficamente mayores o iguales que P. La segunda búsqueda binaria determina la última posición ep en la cual un sufijo comienza con P. El Algoritmo 1 muestra el seudocódigo.

Una vez que se obtiene el intervalo, las ocurrencias se localizan reportando todos los punteros del intervalo, cada uno de ellos en tiempo constante.

```
sp ← 1;
st ← n + 1;
while sp < st do
    s ← ⌊(sp + st)/2⌋;
    if P > T_{A[s],A[s]+m−1} then
        sp ← s + 1;
    else
        st ← s;
ep ← sp − 1;
et ← n;
while ep < et do
    e ← ⌈(ep + et)/2⌉;
    if P = T_{A[e],A[e]+m−1} then
        ep ← e;
    else
        et ← e − 1;
return [sp, ep]
```

Algoritmo 1 : Búsqueda de P en el arreglo de sufijos A del texto T. Se asume que T termina con \$.

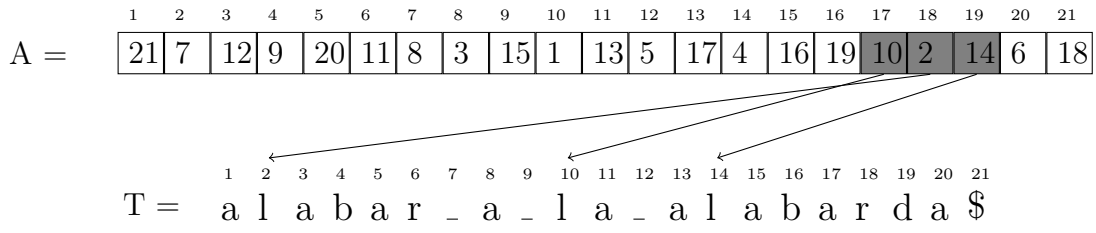


Figura 2.1: Arreglo de sufijos del texto "alabar_a_la_alabarda". El intervalo oscurecido en el arreglo de sufijos corresponde a los sufijos que comienzan con la secuencia "la".

2.4. La Transformada de Burrows-Wheeler (BWT)

La transformada de Burrows-Wheeler [3] de un texto T de tamaño n es una permutación de T que suele ser más compresible que T. Sea $A[1,n]$ el arreglo de sufijos de T. La transformada de Burrows-Wheeler de T, T^{bwt}, se define como $T_i^{bwt} = T_{A[i]-1}$, excepto cuando $A[i] = 1$, caso en que $T_i^{bwt} = T_n$.

Es decir, T^{bwt} se forma recorriendo secuencialmente el arreglo de sufijos A, y concatenando el carácter que precede a cada sufijo.

Otra forma de ver la transformada de Burrows-Wheeler es la siguiente: Considerar una matriz M en donde estén todas las secuencias de la forma $T_{i,n}T_{1,i-1}$ ordenados lexicográficamente. Si se considera una matriz con todos estas secuencias, la última columna corresponde a T^{bwt}.

Mediante este proceso, si llamamos F y $L = T^{bwt}$ a la primera y la última columna de M, respectivamente, resulta de utilidad definir la función LF tal que $LF(i)$ es la posición en F donde aparece el carácter L_i.

Una forma de calcular esta función LF es suponer que tenemos una función C tal que $C(c)$ es el número de ocurrencias de caracteres menores que c en T. Entonces $LF(i) = C(c) + rank_c(L,i)$ con $c = L_i$. Consideremos por ejemplo el carácter $L_8 = l$ en la Figura 2.2. $C(l) = 16$ nos permite saber que, en la columna F, el intervalo correspondiente a las letras l comienza en la posición 17, pues hay exactamente 16 caracteres menores que l en el texto, y los caracteres en F están ordenados lexicográficamente. Ahora bien, para determinar a cuál de las tres ocurrencias de l en F corresponde L_8, basta ver que todas las ocurrencias de l en L se encuentran ordenadas por el carácter que viene a continuación en el texto, y lo mismo ocurre con las l en F. Es decir, los caracteres iguales conservan su ordenamiento relativo en L y F. Luego, $rank_l(L,8) = 2$ nos indica que L_8 es la segunda l en L, por lo tanto, L_8 se encuentra en F en la posición $18 = 16 + 2$.

Utilizando la función LF es posible recuperar el texto original a partir de T^{BWT}. Sabemos que el último carácter del texto es \$, y que $F_1 =$ \$ (pues \$ es lexicográficamente menor que todos los caracteres). Por construcción de M, el carácter que precede a F_i en el texto es L_i. Por tanto, sabemos que el carácter que precede a \$ en el texto es $L_1 = a$. Calculamos $LF(1)$ para conocer la posición de esa a en F, y sabremos que el carácter que la precede en el texto es $L_{LF(1)}$. En el caso general, basta considerar $T[n] =$ \$ y $s = 1$ y luego para cada

$k = n - 1 \ldots 1$, $s \leftarrow LF(s)$, $T[k] \leftarrow T^{bwt}[s]$.

Figura 2.2: Matriz M para la transformada de Burrows Wheeler. La transformada de Burrows Wheeler corresponde a la última columna, $L =$ "araadl_ll$_bbaar_aaaa". Notar que la primera columna de la matriz ordenada, F, corresponde al arreglo de sufijos del texto.

2.5. Códigos de largo variable

Si tenemos una secuencia de n números, entre 1 y n, necesitamos $\log n$ bits para representar cada uno de ellos, si es que utilizamos códigos del mismo largo. Sin embargo, es posible representar la secuencia utilizando una codificación de largo variable, y de esta manera usar menos espacio para la codificación de la secuencia completa. En esta sección presentamos los códigos de largo variable más relevantes para nuestras estructuras.

2.5.1. Codificación Elias-γ y Elias-δ

Tanto la codificación Elias-γ como la Elias-δ [6] permiten codificar secuencias de enteros asignando un largo variable a cada uno de ellos. Si uno simplemente codificara cada entero de la secuencia en binario, no sería posible decodificar la secuencia, pues al ser de largo variable, no sabríamos donde termina cada entero. Para solucionar lo anterior, la codificación Elias-γ de un entero $x > 0$ consiste en la codificación en binario de x, antecedido por el largo en bits de esta codificación escrito en unario. Por ejemplo, si queremos codificar la secuencia 4,13, consideramos primeramente la codificación binaria de 4, 100. Como esta codificación utiliza 3 bits, se tendrá que $\gamma(4) = 110100$. La codificación binaria de 13 es 1101, la cual requiere 4 bits, por lo que $\gamma(13) = 11101101$. Luego, la secuencia entera codificada es $\gamma(4)\gamma(13) = 11010011101101$ Asintóticamente, esta codificación requiere $2\log x + O(1)$ bits.

Un mecanismo más eficiente para x mayores es la codificación Elias-δ, en la cual el entero x es codificado en binario, y es antecedido por el largo en bits de esta representación, esta vez no en unario sino que representado utilizando la codificación Elias-γ. Veamos la codificación para la secuencia $4, 13$. La codificación en binario de 13 es 1101, la cual utiliza 4 bits. Así, $\delta(13) = 1101001101$. La codificación en binario de 4 es 100 la cual requiere 3 bits, por lo que $\delta(4) = 1011100$. Así, la secuencia completa codificada es $\delta(4)\delta(13) = 10111001101001101$ Asintóticamente, esta codificación requiere $\log x + 2\log\log x + O(1)$ bits.

El código resultante al aplicar tanto la codificación Elias-γ como la Elias-δ se considera *código de bloque*, pues cada número original se codifica a un único número. Es *instantáneo* pues un símbolo codificado puede ser decodificado inmediatamente después de ser leído, sin necesidad de hacer referencia a otro símbolo.

2.5.2. Codificación de Huffman

La codificación de Huffman [14] es una técnica clásica para codificar una secuencia de símbolos sobre un alfabeto asignando a cada uno de ellos un código de largo variable, de manera que el tamaño de la secuencia codificada sea óptimo, sujeto a la restricción de generar un *código instantáneo* o *libre de prefijo*, esto es equivalente a decir que ningún símbolo codificado será prefijo de otro. La idea subyacente es asignar a los símbolos más frecuentes códigos más cortos, y aquellos menos frecuentes códigos más largos, garantizando que el código obtenido sea *instantáneo*.

El primer paso para obtener la codificación de Huffman consiste en ordenar los símbolos de acuerdo a sus probabilidades, que en el caso en que conocemos la secuencia completa a codificar es equivalente a ordenarlos por frecuencia. Luego, los dos símbolos con menor probabilidad son reemplazados por un símbolo virtual, cuya probabilidad es la suma de las probabilidades de los símbolos que lo componen. Este proceso se repite sucesivas veces hasta que se tengan sólo dos símbolos. A estos símbolos se asignarán códigos 0 y 1. Este procedimiento se ejemplifica en la Figura 2.3.

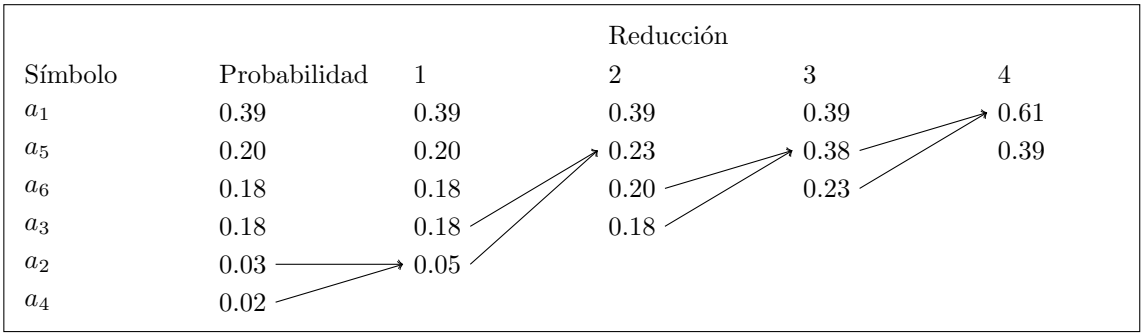


Figura 2.3: Procedimiento de reducción de símbolos para asignar códigos de Huffman. Las flechas muestran el proceso de composición de símbolos en símbolos virtuales, cuya probabilidad asociada es la suma de los símbolos que lo componen.

Luego debemos asignar los códigos a los símbolos originales. Para ello debemos ver los símbolos codificados que tenemos: si alguno de ellos es un código virtual, éste es reemplazado por los dos códigos que lo compusieron, y a ellos se les asigna como código el código del símbolo compuesto que formaron, precedido por 0 y 1 respectivamente. Este proceso se repite sucesivas veces hasta que no queden símbolos virtuales. Este procedimiento se ejemplifica en la Figura 2.4.

Se puede notar que el procedimiento de obtención de los códigos corresponde a la creación de un árbol binario, en el cual las hojas representan a los símbolos que aparecen en la secuencia original mientras que los nodos internos corresponden a los símbolos virtuales. Asimismo, el código de Huffman de un símbolo está determinado por el camino desde la raíz hasta la hoja correspondiente. Así es fácil ver que la codificación de Huffman es *libre de prefijo*: si el código de un símbolo fuese prefijo del código de otro símbolo, el primer símbolo sería un ancestro del segundo en el árbol de Huffman, pero por construcción todos los símbolos son hojas.

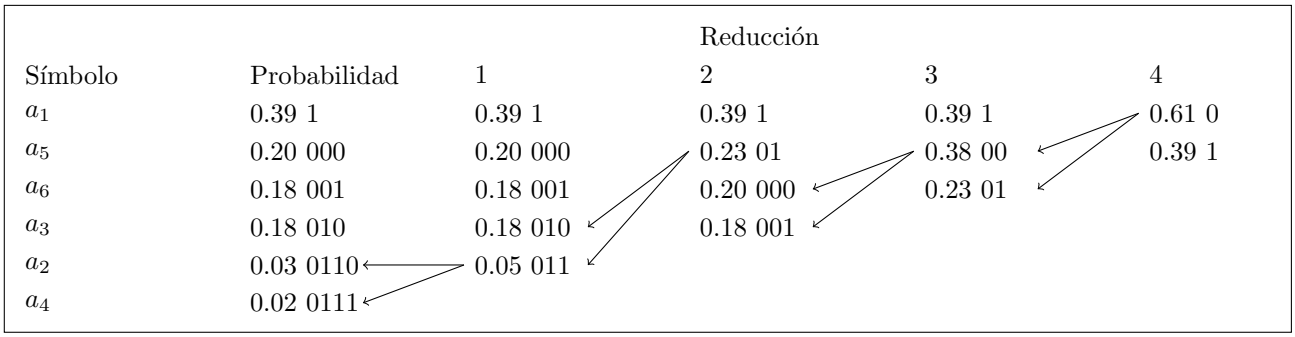


Figura 2.4: Prodecimiento para asignar códigos de Huffman. Una vez que el alfabeto virtual se reduce a dos símbolos, se asignan los códigos 0 y 1. Cada probabilidad tiene debajo el símbolo asociado. Las flechas muestran el proceso de asignación de códigos para los símbolos que dieron lugar a símbolos virtuales.

2.5.3. Códigos Densos (s,c)

Una técnica relativamente nueva para la compresión de texto son los Códigos Densos (s, c) (SCDC) [1]. Debido a que es una técnica relativamente simple, los códigos se pueden generar entre un 45 % a 50 % más rápido que los códigos de Huffman obteniendo una compresión cercana a la óptima.

Consideremos dos enteros positivos s y c. Una codificación (s, c) *stop-cont* asigna a cada símbolo un único código formado por una secuencia de cero o más dígitos en base c (es decir, entre 0 y $c - 1$) terminado con un dígito entre c y $c + s - 1$.

Se puede ver que un código (s, c) es una representación en base c a excepción del último dígito que está entre c y $c + s - 1$. Cada uno de éstos dígitos se almacenará en palabras de b bits, dónde $2^b = s + c$.

Veamos que un código (s, c) *stop-cont* es efectivamente una codificación instantánea. Si consideramos la codificación de dos símbolos distintos y suponemos que uno de los códigos es prefijo del otro, debe ocurrir que la última palabra del código más corto tenga un valor entre c y $c + s - 1$. Entonces, el código más largo tendría una palabra que no es la última con un valor mayor que c, lo cual no es posible.

Si conocemos la distribución de probabilidades de los símbolos a codificar, podemos determinar una codificación (s, c) *stop-cont* entre todas las posibles que minimiza el largo total de la codificación. A esta codificación le llamaremos *Código Denso (s,c)* o *SCDC*.

Así, teniendo los valores de s y c, el proceso de asignación de códigos se resume en:

- A los s símbolos más frecuentes se les asigna un código de largo b, con valores entre c y $c + s - 1$, ambos inclusive.
- A los siguientes sc símbolos más frecuentes se les asigna un código de largo $2b$. La primera palabra está en el rango $[0, c - 1]$, y la segunda en el rango $[c, c + s - 1]$.
- A las siguientes sc^2 palabras se les asigna un código de tres palabras, y así sucesivamente.

Se puede notar que los códigos no dependen de la probabilidad exacta de cada carácter sino que solamente del lugar que ocupen al ordenarlos según su probabilidad.

Para valores pequeños de s, el número de símbolos de alta frecuencia codificados en un byte es pequeño, pero en éste caso el valor de c es grande, por lo que muchas palabras con frecuencia baja serán codificadas usando pocos bytes: sc símbolos serán codificados en dos bytes, sc^2 símbolos serán codificados en tres bytes, sc^3 símbolos serán codificados en cuatro bytes, y así sucesivamente. Visto así, mientras aumentamos el valor de s, ganamos compresión en los símbolos más frecuentes, y perdemos compresión en los símbolos menos frecuentes. En algún momento, la compresión perdida en las palabras menos frecuentes superará la ganancia en las palabras más frecuentes. En éste punto hemos determinado el valor óptimo de s.

Capítulo 3

Trabajo Relacionado

3.1. Estructuras de datos para Rank y Select binario

3.1.1. La solución clásica

En esta sección detallaremos la solución clásica para dar soporte a las funciones *rank* y *select* sobre una secuencia de bits $B_{1,n}$ en tiempo constante [21, 4], utilizando $n + o(n)$ bits.

Consideremos el caso de *rank*. La estructura planteada consiste en un diccionario de dos niveles que almacena las respuestas para algunas posiciones del arreglo, distribuidas de manera uniforme, más una tabla que contiene las respuestas precalculadas para todas las secuencias posibles de un tamaño suficientemente pequeño.

Dividamos la secuencia B en bloques de tamaño $b = \lfloor \log(n)/2 \rfloor$. Luego, agrupemos los bloques consecutivos en superbloques de largo $s = b\lfloor \log(n) \rfloor$.

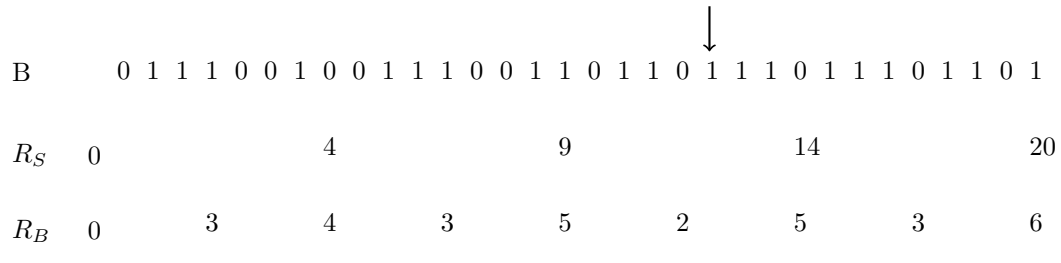
En un arreglo R_s almacenemos la respuesta a $rank$ para las posiciones correspondientes al comienzo de cada superbloque en un arreglo R_s. Es decir, $R_s[j] = rank(B, j \times s),\ j = 0 \ldots \lfloor n/s \rfloor$. El arreglo R_s requiere $O(n/\log(n))$ bits, pues cada elemento requiere $\log(n)$ bits, y hay $n/s = O(n/\log^2 n)$ elementos.

En un arreglo R_b almacenemos para cada bloque la cantidad de unos desde el comienzo del superbloque que lo contiene hasta el comienzo del bloque. Es decir, para cada bloque k contenido en el superbloque $j = k\ div\ b$, $k = 0 \ldots \lfloor n/b \rfloor$ almacenamos $R_b[k] = rank(B, k \times b) - rank(B, j \times s)$. El arreglo R_b requiere $(n/b) \log s = O(n \log\log(n)/\log(n))$ bits pues tiene n/b entradas, y cada una de ella requiere $\log s$ bits.

Finalmente, la tabla R_p almacena las respuestas a todas las posibles consultas de rank para subsecuencias de largo b. Es decir, $R_p[S, i] = rank(S, i)$, para toda secuencia de bits S de largo b, para todo i entre 0 y b. Esta tabla requiere $O(2^b \times b \times \log(b)) = O(\sqrt{n} \log n \log\log n)$

bits.

Todas las estructuras descritas requieren $O(\frac{n}{\log n} + \frac{n \log\log(n)}{\log n} + \sqrt{n} \log n \log\log n) = o(n)$ bits, permitiendo calcular *rank* en tiempo constante como: $rank(B, i) = R_s[i \ div \ s] + R_b[i \ div \ b] + R_p[B[(i \ div \ b) \times b + 1 \ldots (i \ div \ b) \times b + b], i \ mod \ b]$



R_P	0	1	2	3
0000	0	0	0	0
0001	0	0	0	1
⋮	⋮	⋮	⋮	⋮
1110	1	2	3	3
1111	1	2	3	4

$$rank(B, 20) = R_S[2] + R_B[5] + R_P[1110, 0]$$
$$= 9 + 2 + 1 = 12$$

Figura 3.1: Ejemplo de las estructuras adicionales para calcular rank utilizando $n + o(n)$ bits.

En el caso de *select* la idea subyacente es similar, pero en vez de dividir B en intervalos regulares, particionaremos el espacio $[1, n]$ de posibles argumentos para *select* (es decir los valores de j para $select(B, j)$) en bloques y superbloques. Teniendo cuidado de escoger los tamaño de los bloques y superbloques, y con algunas consideraciones especiales (una extensión ingenua tendría el problema de no poder garantizar que una respuesta relativa entre un bloque y un superbloque pueda requerir $\log n$ bits en vez de los $\log(s)$ obtenidos en el caso de rank) es posible soportar *select* en tiempo constante, utilizando $n + o(n)$ bits [21, 4].

3.1.2. La estructura de Raman, Raman y Rao

La estructura propuesta por Raman, Raman y Rao [23] permite responder las consultas rank y select en tiempo constante sobre una secuencia de bits $B_{1,n}$ utilizando una representación comprimida de la secuencia que requiere $nH_0(B) + o(n)$ bits en total.

La idea consiste en separar B en bloques de tamaño $t = \frac{\log n}{2}$. Cada bloque $I = B_{ti+1,ti+t}$ será representado por una tupla(c_i, o_i), en donde c_i representa la clase a la que pertenece, y o_i

indica a cuál de todas las secuencias pertenecientes a la clase c_i pertenece I. Cada clase c agrupa a todas las posibles secuencias de largo t que tienen c unos. Por ejemplo, si $t = 4$, la clase 0 es 0000, la clase 1 es $0001, 0010, 0100, 1000$, la clase 2 es $0011, 0101, 0110, 1001, 1010, 1100$ ...y la clase 4 es 1111. Vemos que hay $t + 1$ clases de bloques de tamaño t, por lo que almacenar c_i requiere $\lceil \log(t+1) \rceil$ bits. Por otro lado, podemos ver que la clase c_i tiene $\binom{t}{c_i}$ elementos. Luego, para representar o_i necesitamos $\lceil \log \binom{t}{c_i} \rceil$ bits. Para representar B utilizamos $\lceil n/t \rceil$ de estas tuplas. El espacio total requerido por la representación de los c_i es simplemente $\Sigma_{i=0}^{\lceil n/t \rceil} \lceil \log(t+1) \rceil = O(n \log(t)/t) = O(n \log \log n / \log n)$ bits. Por otro lado, el espacio requerido por los o_i está dado por:

$$\begin{aligned}
\left\lceil \log \binom{t}{c_1} \right\rceil + \ldots + \left\lceil \log \binom{t}{c_{\lceil n/t \rceil}} \right\rceil &\leq \log \left(\binom{t}{c_1} \times \ldots \times \binom{t}{c_{\lceil n/t \rceil}} \right) + n/t \\
&\leq \log \binom{n}{c_1 + \ldots + c_{\lceil n/t \rceil}} + n/t \\
&= \log \binom{n}{c_{tot}} + n/t \\
&\leq nH_0(B) + O(n/\log n)
\end{aligned}$$

Para poder responder las consultas en tiempo constante necesitamos estructuras adicionales, similares a las mostradas en la Sección 3.1.1. Utilizaremos los mismos arreglos R_s y R_b. Sin embargo, como las tuplas (c_i, o_i) tienen largo variable, necesitamos además dos arreglos $Rpos_s$ y $Rpos_b$ que indiquen para cada superbloque y para cada bloque, respectivamente, la posición en la representación comprimida de B donde comienza la tupla correspondiente al bloque o superbloque correspondiente. Además, para poder calcular *rank* debemos tener una tabla análoga a R_p, pero esta vez no podemos acceder a ella por la secuencia S, pues ella no está disponible. En su lugar, accedemos a R_p por el par (c_i, o_i) que representa la subsecuencia S comprimida.

Las estructuras para *select* son más complicadas, pero la idea es similar a la presentada en la sección anterior.

De esta manera, se obtiene una representación que requiere $nH_0(B) + o(n)$ bits y que permite responder *rank* y *select* en tiempo constante.

3.2. Wavelet Tree

Consideremos una secuencia de caracteres $T = a_1 a_2 \dots a_n$, donde $\forall i = 1 \dots n, a_i \in \Sigma$. El Wavelet Tree es un árbol balanceado, en el cual cada hoja representa un símbolo del alfabeto. La raíz está asociada con la secuencia original $T = a_1 a_2 \dots a_n$. Su hijo izquierdo está asociado con la subsecuencia de T obtenida al concatenar todas las posiciones i que satisfagan $a_i < \sigma/2$ (donde σ es el tamaño del alfabeto Σ) mientras que su hijo derecho se asocia con la subsecuencia complementaria (la concatenación de los $a_i \geq \sigma$). Esta subdivisión se mantiene recursivamente, hasta llegar a las hojas, las cuales estarán asociadas a repeticiones de un símbolo. En cada nodo, la secuencia es representada por un arreglo de bits, que indica qué posiciones (las marcadas con 0) van al hijo izquierdo, y cuáles (las marcadas con 1) van hacia el hijo derecho. Estos arreglos de bits por sí solos bastan para recuperar la secuencia original: Para recuperar el carácter a_i, se comienza en la raíz y se baja a la izquierda o a la derecha según el valor del arreglo de bits asociado a la raíz en la posición i. Al bajar por el hijo izquierdo, se debe reemplazar $i \leftarrow rank_0(B, i)$ y similarmente $i \leftarrow rank_1(B, i)$ cuando se baje por la derecha. Al llegar a una hoja, se tendrá que el símbolo asociado a aquella hoja es el valor de a_i.

De manera similar, podemos también calcular el número de ocurrencias del carácter c, hasta la posición i en la secuencia T, $rank_c(T, i)$, realizando $\log \sigma$ operaciones $rank$. Para ello, se comienza en la raíz y se baja a la izquierda o a la derecha dependiendo de si c pertenece a la primera o a la segunda mitad del alfabeto, respectivamente. Al bajar por el hijo izquierdo, se debe reemplazar $i \leftarrow rank_0(B, i)$ y similarmente $i \leftarrow rank_1(B, i)$ cuando se baje por la derecha. En este segundo nodo ya tenemos representada sólo la mitad del alfabeto. Ahora bajaremos a la izquierda o a la derecha dependiendo de a qué mitad del alfabeto representada en este nodo pertenece el carácter c. Repetimos este proceso recursivamente, hasta llegar a una hoja. Acá se tendrá que el último valor de i obtenido corresponde a $rank_c(T, i)$. Dotando a los arreglos de bits de la estructura para responder $rank$ mostrada anteriormente podremos realizar estas operaciones de manera rápida y ocupando poco espacio.

Similarmente, $select_c(T, i)$, que localiza la i-ésima ocurrencia de c en T, se puede calcular haciendo $\log \sigma$ invocaciones al *select* de bits, en un camino desde la hoja que representa a c hasta la raíz del Wavelet Tree.

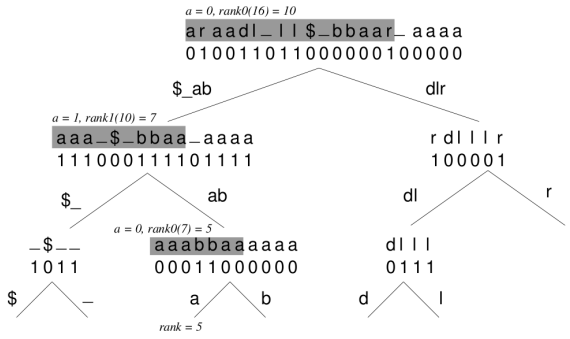


Figura 3.2: Wavelet Tree para la secuencia T = "araadl_ll$bbaar_aaa", que es la transformada de Burrows-Wheeler de "alabar_a_la_alabarda$". Se ilustra el proceso para calcular $rank_a(A, 15)$

3.3. El Índice FM

El Índice FM (FMI) corresponde a una familia de índices de texto completo que realizan búsqueda reversa para determinar las ocurrencias de un patrón en el texto, utilizando la transformada de Burrows-Wheeler. Existen diferentes maneras de representar la transformada de Burrows-Wheeler que dan lugar a diferentes resultados en cuanto a compresión y tiempos de búsqueda.

3.3.1. Búsqueda reversa y la transformada de Burrows-Wheeler

En la Sección 2.3 mostramos que determinar las ocurrencias de un patrón en el texto podía realizarse mediante búsquedas binarias en el arreglo de sufijos. La búsqueda reversa permite también buscar el patrón usando el arreglo de sufijos, pero utilizando otro enfoque.

Consideremos que deseamos buscar un patrón P sobre un texto T, y sea A su arreglo de sufijos. En primer lugar, buscamos determinar el intervalo $[sp_m, ep_m]$ en A donde se ubican todos los sufijos que comienzan con el último carácter de P, p_m. La función C definida en la Sección 2.4 (que puede almacenarse directamente como un arreglo) permite identificar este intervalo mediante $[sp_m, ep_m] = [C(p_m) + 1, C(p_m + 1)]$. Ahora, dado $[sp_m, ep_m]$, nos interesa determinar $[sp_{m-1}, ep_{m-1}]$, el intervalo de los sufijos que comienzan con $p_{m-1}p_m$. Es importante notar que $[sp_{m-1}, ep_{m-1}]$ es un subintervalo de $[C(p_{m-1}) + 1, C(p_{m-1} + 1)]$.

En el caso general, dado $[sp_{i+1}, ep_{i+1}]$, nos interesa determinar $[sp_i, ep_i]$. Para ello, una herramienta fundamental es la función LF definida en la Sección 2.4. Notemos que todas las ocurrencias de p_i en $L[sp_{i+1}, ep_{i+1}]$ aparecen contiguas en F y mantienen su orden relativo.

Para encontrar el nuevo intervalo quisiéramos determinar la primera y la última ocurrencia de p_i en $L[sp_{i+1}, ep_{i+1}]$. Es decir, buscamos b y e tal que $L_b = p_i$ y $L_e = p_i$ son la primera y la última ocurrencia de p_i en $L[sp_{i+1}, ep_{i+1}]$ respectivamente. Luego, $LF(b)$ y $LF(e)$ son la primera y la última fila en F que comienzan con el carácter p_i y que son seguidas con $p_i \dots p_m$. Es decir, $sp_i = LF(b)$ y $ep_i = LF(e)$. Si bien no conocemos ni b ni e, veamos que no son necesarios:

$$\begin{aligned} LF(b) &= C(L_b) + rank_{L_b}(T^{bwt}, b) \\ LF(b) &= C(p_i) + rank_{p_i}(T^{bwt}, b) \\ LF(b) &= C(p_i) + rank_{p_i}(T^{bwt}, sp_{i+1} - 1) + 1 \end{aligned}$$

De manera análoga,

$$\begin{aligned} LF(e) &= C(L_e) + rank_{L_e}(T^{bwt}, e) \\ LF(e) &= C(p_i) + rank_{p_i}(T^{bwt}, e) \\ LF(e) &= C(p_i) + rank_{p_i}(T^{bwt}, ep_{i+1}) \end{aligned}$$

La primera igualdad se tiene por lo visto en la Sección 2.4. La segunda ocurre porque el carácter que está en la posición e de T^{BWT} es justamente el carácter del patrón que estamos buscando, p_i. Finalmente, la tercera se tiene porque b es la primera ocurrencia de p_i en el intervalo. Análogamente para $LF(e)$ sabemos que las ocurrencias de p_i hasta la posición e son las mismas que hasta la posición ep_{i+1} pues e por definición corresponde a la última posición en L en el rango sp_{i+1}, ep_{i+1} de una ocurrencia de p_i. El seudocódigo de la búsqueda reversa se muestra en el Algoritmo 2. En la Figura 3.3 podemos ver un ejemplo de búsqueda reversa.

3.3.2. EL Wavelet Tree FM Index

Consideremos que C se guarda directamente como un arreglo, necesitando $\sigma \log n$ bits. El tiempo de búsqueda del Algoritmo 2 está dominado por el tiempo que demora calcular $2 \times m$ veces la función $rank_{p_i}(T^{BWT}, c)$. Aquí es donde surgen distintas maneras de representar T^{BWT} , dando lugar cada una de ellas a distintos tiempos de búsqueda y requerimientos de espacio [8, 9, 17] .

El Wavelet Tree FM Index (WT-FMI) representa la transformada de Burrows-Wheeler

$sp \leftarrow 1;$
$ep \leftarrow n;$
for $i \leftarrow m$ **to** 1 **do**
    $sp \leftarrow C(p_i) + rank_{P_i}(T^{BWT}, p_i, sp - 1) + 1;$
    $ep \leftarrow C(p_i) + rank_{P_i}(T^{BWT}, p_i, ep);$
    if $sp > ep$ **then**
        return ϕ
return $[sp, ep]$

Algoritmo 2 : Búsqueda reversa de los sufijos que comienzan con $P_{1,m}$.

i	A[i]	L	F
1	21	a	$alabar_a_la_alabarda
2	7	r	_a_la_alabarda$alabar
3	12	a	_alabarda$alabar_a_la
4	9	a	_la_alabarda$alabar_a
5	20	d	a$alabar_a_la_alabard
6	11	l	a_alabarda$alabar_a_l
7	8	_	a_la_alabarda$alabar_
8	3	l	abar_a_la_alabarda$al
9	15	l	abarda$alabar_a_la_al
10	1	$	alabar_a_la_alabarda$
11	13	_	alabarda$alabar_a_la_
12	5	b	ar_a_la_alabarda$alab
13	17	b	arda$alabar_a_la_alab
14	4	a	bar_a_la_alabarda$ala
15	16	a	barda$alabar_a_la_ala
16	19	r	da$alabar_a_la_alabar
17	10	_	la_alabarda$alabar_a_
18	2	a	labar_a_la_alabarda$a
19	14	a	labarda$alabar_a_la_a
20	6	a	r_a_la_alabarda$alaba
21	8	a	rda$alabar_a_la_alaba

i	A[i]	L	F
1	21	a	$alabar_a_la_alabarda
2	7	r	_a_la_alabarda$alabar
3	12	a	_alabarda$alabar_a_la
4	9	a	_la_alabarda$alabar_a
5	20	d	a$alabar_a_la_alabard
6	11	l	a_alabarda$alabar_a_l
7	8	_	a_la_alabarda$alabar_
8	3	l	abar_a_la_alabarda$al
9	15	l	abarda$alabar_a_la_al
10	1	$	alabar_a_la_alabarda$
11	13	_	alabarda$alabar_a_la_
12	5	b	ar_a_la_alabarda$alab
13	17	b	arda$alabar_a_la_alab
14	4	a	bar_a_la_alabarda$ala
15	16	a	barda$alabar_a_la_ala
16	19	r	da$alabar_a_la_alabar
17	10	_	la_alabarda$alabar_a_
18	2	a	labar_a_la_alabarda$a
19	14	a	labarda$alabar_a_la_a
20	6	a	r_a_la_alabarda$alaba
21	8	a	rda$alabar_a_la_alaba

i	A[i]	L	F
1	21	a	$alabar_a_la_alabarda
2	7	r	_a_la_alabarda$alabar
3	12	a	_alabarda$alabar_a_la
4	9	a	_la_alabarda$alabar_a
5	20	d	a$alabar_a_la_alabard
6	11	l	a_alabarda$alabar_a_l
7	8	_	a_la_alabarda$alabar_
8	3	l	abar_a_la_alabarda$al
9	15	l	abarda$alabar_a_la_al
10	1	$	alabar_a_la_alabarda$
11	13	_	alabarda$alabar_a_la_
12	5	b	ar_a_la_alabarda$alab
13	17	b	arda$alabar_a_la_alab
14	4	a	bar_a_la_alabarda$ala
15	16	a	barda$alabar_a_la_ala
16	19	r	da$alabar_a_la_alabar
17	10	_	la_alabarda$alabar_a_
18	2	a	labar_a_la_alabarda$a
19	14	a	labarda$alabar_a_la_a
20	6	a	r_a_la_alabarda$alaba
21	8	a	rda$alabar_a_la_alaba

Figura 3.3: Búsqueda reversa del patrón *ala*. En el primer paso se muestra el intervalo de todos los sufijos que comienzan con *a*, el último carácter de nuestro patrón. Este intervalo se obtiene fácilmente mediante $sp = C(a) + 1 = 5$ y $ep = C(b) = 13$. Para obtener el segundo intervalo, debemos encontrar el subintervalo de los sufijos que comienzan con l que tienen a continuación una a: $1 + C(l) = 17$ nos permite identificar la posición en F donde comienzan los sufijos que comienzan con l. Recordando que los caracteres iguales preservan su orden relativo en L y F, y utilizando $rank_l(L, 4) = 0$ y $rank_l(L, 13) = 3$ sabemos que el nuevo intervalo está dado por $[17, 19]$. Para obtener el tercer intervalo, utilizamos que $1 + C(a) = 5$, $rank_a(L, 16) = 5$ y $rank_a(L, 19) = 7$, por lo que el tercer intervalo es $[10, 11]$.

T^{BWT} del texto mediante un Wavelet Tree, permitiendo calcular $rank_c(T^{BWT}, i)$ en tiempo $O(\log \sigma)$. De esta manera, el conteo de las ocurrencias de P en T puede llevarse a cabo en tiempo $O(m \log \sigma)$.

La idea esencial del WT-FMI fue introducida por Sadakane [24] en términos de una jerarquía de arreglos de bits, cuando el Wavelet Tree aun no existía. La idea de usar un Wavelet Tree como tal fue propuesta posteriormente por Ferragina et al. [7] como un caso particular de otro tipo de Índice FM, el *Alphabet Friendly FM Index* (AF-FMI) orientado a disminuir la dependencia del tamaño del alfabeto, y también por Mäkinen y Navarro como un caso particular del *Run Length FM Index* (RL-FMI) [16]. Ambos índices logran contar las ocurrencias del patrón en tiempo $O(m \log \sigma)$ y requieren espacio proporcional a la entropía de orden k.

Más tarde, Mäkinen y Navarro [17] demostraron que almacenar el Wavelet Tree de la Transformada de Burrows-Wheeler del texto requiere espacio $nH_k(T) + o(n \log \sigma)$, para $k \leq \alpha \log \sigma$, $\alpha \leq 1$, sin necesidad de mayor sofisticación.

3.3.3. Localizando las ocurrencias

El Algoritmo 2 nos entrega como resultado un intervalo en el arreglo de sufijos, cuyo tamaño es el número de ocurrencias.

Para determinar la posición en el texto a la que corresponde cada una de las ocurrencias en el arreglo de sufijos, sin almacenarlo, se muestrea regularmente el texto cada $\log^{1+\epsilon} n / \log \sigma$ posiciones para algún $\epsilon > 0$ y se almacenan los valores de A que apuntan a las posiciones muestreadas del texto en un arreglo A', y se guarda un arreglo de bits $mark[1, n]$ que indica con 1 las posiciones de A que están muestreadas. De esta forma, para determinar $A[i]$ se identifica el menor $r \geq 0$ tal que $LF^r(i)$ (la composición de LF r veces) está marcado. Luego, $A[i] = A[LF^r(i)] + r = A'[rank(mark, LF^r(i))] + r$.

Si representamos el arreglo $mark$ de manera que soporte rank en tiempo constante, el tiempo que demora reportar una ocurrencia está dominado por calcular r veces LF. Así, el tiempo en reportar la posición de cada ocurrencia es $O(\log^{1+\epsilon} n)$, pues r siempre es menor que el período de muestreo.

3.3.4. Acceso a subcadenas

El procedimiento para reconstruir una cadena de largo l a partir de la posición i, $T_{i,i+l-1}$, es similar al expuesto para localizar ocurrencias, pero ahora necesitamos otro muestreo. En un arreglo A^{-1} almacenaremos para cada posición del texto que esté muestreada en A', el valor de la posición en A que lo apunta, es decir, $A^{-1}[j] = p$ con $SA[p] = j \times (\log^{1+\epsilon} n / \log \sigma)$, $j = 1 \ldots n/\frac{\log^{1+\epsilon} n}{\log \sigma}$. De esta manera, para reconstruir la secuencia necesitamos determinar la posición muestreada en el texto más cercana a continuación de $i + l$. Notando que ésta corresponde a la $\lceil (i + l)/s \rceil$-ésima posición muestreada del texto podemos obtener $p = A^{-1}[\lceil (i+l)/s \rceil]$, la posición en el arreglo de sufijos que apunta hacia la siguiente posición muestreada en el texto. Luego podemos calcular $T_{i+l} = LF^r(i)$ (con $r = p - i - l$). Aplicando LF l veces más podemos obtener todos los caracteres de $T_{i,i+l-1}$. Así, mostrar una subcadena de l caracteres del texto requiere tiempo $O((l \log \sigma + \log^{1+\epsilon} n))$

3.3.5. Tamaño del índice

En la Sección 3.2 vimos que el Wavelet Tree está compuesto por $\log \sigma$ niveles B_i, cada uno de los cuales puede ser representado utilizando $nH_0(B_i) + o(n)$ bits [23], ver Sección 3.1.2. Así, el espacio total requerido por el Wavelet Tree es $\sum_{i=1}^{\log \sigma}(nH_0(B_i) + o(n)) = nH_0(T) + o(n \log \sigma)$ bits. Sin embargo, cuando el Wavelet Tree se construye sobre la transformada de Burrows-Wheeler del texto, T^{BWT}, el espacio total resulta ser $nH_k(T) + o(n \log \sigma)$ bits [17], ver Sección 3.3.2. Las estructuras para almacenar el muestreo cada $\log^{1+\epsilon} n / \log \sigma$ posiciones del texto requieren $O((n/\frac{\log^{1+\epsilon} n}{\log \sigma}) \log n) = O(n \log \sigma / \log^{\epsilon} n)$ bits. Utilizando la estructura de Raman, Raman y Rao, el arreglo $mark$ se puede representar utilizando $O((n/\frac{\log^{1+\epsilon} n}{\log \sigma}) \log n) = O(n \log \sigma / \log^{\epsilon} n)$ pues tiene $O(n/\frac{\log^{1+\epsilon} n}{\log \sigma})$ unos. Así, el espacio total requerido por el índice es $nH_k + o(n \log \sigma)$ bits.

3.4. CSA de Sadakane

Consideremos un texto T de tamaño n, su arreglo de sufijos A y un patrón P. El arreglo de sufijos comprimido propuesto por Grossi y Vitter [13] logra reducir el tamaño del arreglo de sufijos A de $n \log n$ bits a $O(n \log \sigma)$ bits. Sin embargo, para poder realizar las búsquedas de un patrón P, este índice requiere que conservemos el texto original. Sadakane [25] propuso una variante del CSA que no requiere del texto original para realizar las búsquedas.

El CSA de Sadakane emula la misma búsqueda binaria que se realiza en el arreglo de sufijos, pero sin necesitar acceso a él, ni al texto. Para ello el CSA cuenta con dos estructuras fundamentales. Se define la función Ψ de tal modo que si $\Psi[i] = i'$ entonces $A[i'] = A[i] + 1$ si $A[i] < n$ y $A[i'] = 0$ si $A[i] = n$. Es decir, la función Ψ nos permite avanzar virtualmente hacia la derecha en el texto, llevándonos desde la posición en la que el arreglo de sufijos apunta a T_i, hacia la posición que apunta a T_{i+1}. Es útil notar que la función Ψ es la inversa de LF, utilizada por el FMI. Como nos moveremos en el arreglo de sufijos, necesitamos saber para toda posición i, qué carácter hay en $T_{A[i]}$. Para ello nos será útil considerar la función C definida en la Sección 2.4, que para cada carácter c nos permite saber mediante $C(c)$ el número de ocurrencias de caracteres lexicográficamente menores que c en T. Notando que el arreglo de sufijos está ordenado primero según el primer carácter de cada sufijo, podemos notar que si $T_{A[i]} = c$, se tendrá que $C(c) < i \leq C(c+1)$. Sin embargo, el CSA de Sadakane no almacena C. En su lugar, almacena un arreglo de bits $newF[1, n]$ tal que $newF[1 + C[i]] = 1$ para todo i entre 1 y σ, y un arreglo $S[1, \sigma]$ que guarde en $S[j]$ el j-ésimo carácter distinto (en orden lexicográfico) que aparece en T. Así, podemos calcular c en tiempo constante como $c = S[rank(newF, i)]$. La propuesta de Sadakane almacena las estructuras en una jerarquía, utilizando arreglos de bits para indicar qué posiciones están almacenados en una jerarquía inferior. Para simplificar la explicación, presentamos el CSA de una manera más práctica, y tal como lo implementó realmente Sadakane. Se realiza un muestreo cada $\log^{1+\epsilon} n$ posiciones del texto de la misma manera que en el FMI, almacenando los valores de A que apuntan a dichas posiciones en el arreglo A', el arreglo de bits $mark$ que indica cuáles son las posiciones en A que están muestreadas, y el arreglo de sufijos invertido A^{-1}, que para cada posición muestreada, almacena el valor de la posición en A que apunta hacia ella.

3.4.1. Acceso a subcadenas

El mecanismo para reconstruir una subcadena de largo l a partir de la posición i, $T_{i,i+l-1}$, consiste en ubicar la última posición muestreada del texto antes de i. Esta posición es $\lfloor i/\log^{1+\epsilon} n \rfloor$. Mediante el arreglo de sufijos invertido, podemos saber que $p = A'[\lfloor i/\log^{1+\epsilon} n \rfloor]$ es la posición en el arreglo de sufijos que apunta a la posición muestreada. A partir de p podemos movernos en el arreglo de sufijos hacia la posición que apunta a T_{i+l} mediante la aplicación sucesiva de Ψ. Así podemos obtener $p' = \Psi^r(p)$, con $r = i - \lfloor i/\log^{1+\epsilon} n \rfloor$. Para

obtener el primer carácter de la subcadena, sabemos que $T_i = S[rank(newF, p')]$. Luego determinamos $p'' = \Psi(p')$ y obtenemos $T_{i+1} = S[rank(newF, p'')]$. Aplicando Ψ l veces y calculando los caracteres respectivos reconstruimos la subcadena $T_{i,i+l-1}$. Podemos ver que el tiempo que toma este procedimiento está dado por la aplicación de $m + \log^{1+\epsilon} n$ veces la función Ψ y la función $rank$. Brindando soporte para ambas en tiempo constante, el tiempo de reconstrucción de una subcadena de largo l es $O(m + \log^{1+\epsilon} n)$

3.4.2. Búsqueda de patrones

Para realizar la búsqueda de un patrón $P_{1,m}$, el CSA esencialmente realiza una búsqueda binaria sobre el arreglo de sufijos de manera similar a lo expuesto en la Sección 2.3. Sin embargo, en cada paso de la búsqueda binaria necesitamos comparar el texto apuntado por una posición s del arreglo de sufijos con el patrón. Como se explicó anteriormente, el carácter del texto correspondiente a $T_{A[s]}$ puede obtenerse en tiempo constante mediante $T_{A[s]} = S[rank(newF, s)]$. Para acceder al siguiente carácter del sufijo que estamos comparando, calculamos $s' = \Psi(s)$, y podemos obtener $T_{A[s]+1} = T_{A[s']} = S[rank(newF, s')]$. Así, podemos extraer tantos caracteres del sufijo como sean necesarios para realizar la comparación lexicográfica con el patrón en la búsqueda binaria. De esta manera obtenemos el intervalo en el arreglo de sufijos en el que se encuentran las ocurrencias de P en tiempo $O(m \log n)$.

Para localizar cada una de estas ocurrencias, el procedimiento es similar al del FMI. Para localizar la posición en el texto de una posición p del arreglo de sufijos, $A[p]$, aplicamos $\Psi(p)$ sucesivas veces, avanzando virtualmente hacia la derecha en el texto, hasta encontrar una posición $p' = \Psi^r(p)$ que esté marcada en $mark$. Luego, $A[p] = A[\Psi^r(p)] - r = A'[rank(mark, \Psi^r(p))] - r$. Así, determinar las posiciones de las occ ocurrencias de P en T utilizando el CSA de Sadakane requiere tiempo $O(m \log n + occ \log^{1+\epsilon} n)$

3.4.3. Espacio del índice

Las estructuras de muestreo A', A^{-1} y $mark$ requieren $O(n/\log^{\epsilon} n)$ bits. Las estructuras $newF$ y S, utilizadas para simular C, requieren $n + o(n)$ y $\sigma \log \sigma$ bits, respectivamente. La estructura que más espacio ocupa es Ψ, que si fuera almacenada como un arreglo, ocuparía $n \log n$ bits. Grossi y Vitter [13] mostraron que es posible representar Ψ de manera más compacta teniendo en cuenta que la función Ψ es monótonamente creciente en las áreas de A

que comienzan con el mismo carácter. Utilizando la codificación Elias-δ se puede representar Ψ en $nH_0(T) + O(n \log \log \sigma)$ bits, soportando $\Psi(i)$ en tiempo constante. Así, el espacio total requerido por el CSA de Sadakane, es $nH_0(T) + O(n \log \log \sigma)$ bits.

Capítulo 4

Autoíndice Basado en Arreglo de Sufijos Bidimensional

En este capítulo presentaremos una nueva estructura, que llamaremos CSA-2D, dada a conocer por primera vez en [18]. El CSA-2D es un autoíndice de una imagen, o una colección de ellas, basado en una conocida técnica para indexar imágenes que utiliza los llamados L-sufijos [15]. La idea consiste en que para cada posición $I[i, j]$ de la imagen se defina un *sufijo bidimensional*, cuyos caracteres son secuencias con forma de L de tamaño creciente, que comienzan en (i, j) y aumentan sus valores de i y de j.

Más precisamente, podemos decir que el l-ésimo carácter del sufijo asociado a la posición (i, j), para $l \geq 0$, es la secuencia (vista como un solo símbolo): $I[i, j + l]I[i + 1, j + l]I[i + 2, j + l] \ldots I[i + l - 1, j + l]I[i + l, j]I[i + l, j + 1]I[i + l, j + 2] \ldots I[i + l, j + l]$. Lo anterior se ejemplifica en la Figura 4.1, a la izquierda. Análogamente al caso unidimensional, suponemos que la imagen tiene una fila y una columna adicional con valores únicos para asegurar que la comparación lexicográfica esté bien definida.

De manera análoga al caso unidimensional (Sección 2.3), se puede generar un arreglo de sufijos bidimensional: El arreglo de sufijos debe apuntar a todas las posiciones (i, j) en orden lexicográfico de acuerdo a los L-sufijos correspondientes a cada (i, j). Incluso es posible indexar una colección de imágenes utilizando un solo arreglo de sufijos, de modo que cada posición del arreglo apunte a alguna celda de alguna imagen. En este caso, si N es el tamaño de la colección (en celdas), y consideramos una subimagen P de $m \times m$, la búsqueda de P en la colección se puede realizar mediante dos búsquedas binarias sobre el arreglo de sufijos, en tiempo $O(m^2 \log N)$. El tiempo de búsqueda puede ser reducido a $O(m^2 + \log N)$ utilizando información extra [19]. Esta estructura puede ser construida en tiempo $O(N \log N)$ y ocupa

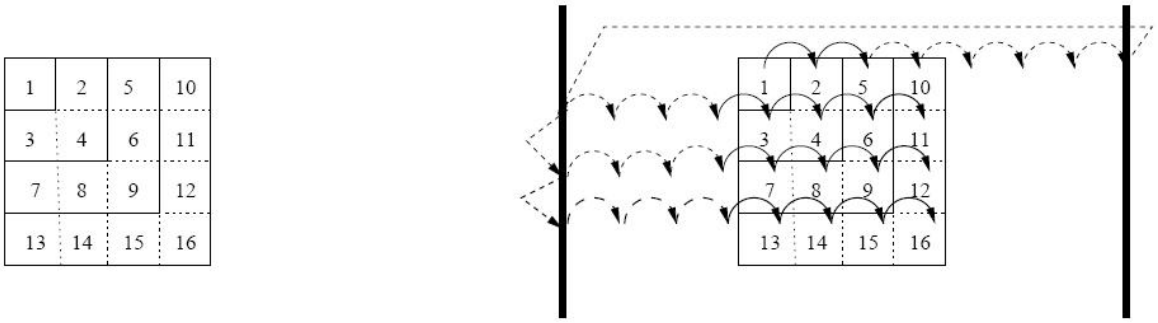


Figura 4.1: A la izquierda podemos ver el orden en el cual se leen las celdas para formar los L-sufijos. A la derecha se muestra el mecanismo para acceder a estas celdas utilizando el muestreo descrito. Las flechas curvas corresponden a la aplicación de Ψ para avanzar hacia la derecha en la imagen virtual. Las flechas punteadas corresponden a aquellos movimientos efectuados para saber en qué parte de la imagen estamos y así poder avanzar a la fila siguiente, mientras que las flechas sólidas corresponden a los movimientos efectuados para determinar el valor de las celdas necesarias para determinar el L-sufijo. Las líneas negras verticales corresponden a las columnas muestreadas. Las líneas punteadas rectas son los movimientos hechos gracias a la información del muestreo.

$O(N \log N)$ bits de espacio. Lo anterior es demasiado, comparado con los $N \log \sigma$ bits que ocupa la imagen, que aún es necesaria para permitir la búsqueda. Para reducir el espacio, en [18] se propone utilizar los conceptos desarrollados en el CSA [25], aplicándolos ahora al caso bidimensional (Sección 3.4).

Consideremos una colección de imágenes I_k de $N_k = R_k \times C_k$ cada una, con $N = \sum_k N_k$. Sea $A[1, N]$ el arreglo de sufijos de la colección. Se define un arreglo $\Psi[1, N]$ de la siguiente manera: Para toda posición en el arreglo de sufijos p, $A[p] = (n_p, i_p, j_p)$ es la celda (i_p, j_p) de la imagen n_p. Entonces $\Psi[p] = p'$ teniendo que $A[p'] = (n_p, i_p, j_p + 1)$ (ó $(n_p, i_p, 1)$ si la imagen tiene sólo j_p columnas). Es decir, Ψ nos permite movernos en el arreglo de sufijos, de manera de recorrer la imagen de izquierda a derecha, como se muestra en la Figura 4.1 a la derecha. Adicionalmente, se considera un arreglo $C[1, \sigma]$ tal que $C[c]$ es el número de celdas en todas las imágenes cuyo valor es menor que c. Este arreglo nos permitirá obtener el valor de una celda a partir de su posición correspondiente en el arreglo de sufijos. Finalmente, se necesita un muestreo de la imagen a intervalos regulares. Sea s el paso según el cual sea realizará el muestreo. Entonces, se almacenarán para cada celda $I[i, j \cdot s]$, para cada imagen, el valor $ISA[n, i, j] = p$ tal que $A[p] = (n, i, j \cdot s)$ (asimismo, también se muestrean la primera y la última columna de cada imagen). Lo anterior es un muestreo del arreglo de sufijos invertido.

Los valores p almacenados se marcan en un arreglo de bits $B[1, N]$. También almacenamos un arreglo de sufijos muestreado $SA[rank(B, p)] = (n, i, j)$. Con las estructuras señaladas, es posible remplazar el arreglo de sufijos A y también la colección de imágenes.

4.1. Acceso a subimágenes

El mecanismo para reconstruir una subimagen de $m_r \times m_c$ definida a partir de la posición (n, i, j) es similar al del CSA de Sadakane. Primero debemos buscar la posición muestreada más cercana a (n, i, j), y ubicarla en el arreglo de sufijos mediante ISA. Esto es, $p_s = ISA[n, i, \lfloor j/s \rfloor]$ es la posición correspondiente a una imagen de tamaño $m_r \times (m_c + j \ mod \ s)$, que contiene a la imagen que buscamos, pues por construcción de ISA tendremos que $A[p_s] = (n, i, \lfloor j/s \rfloor * s)$. Determinamos la posición p en el arreglo de sufijos que apunta a (n, i, j) mediante $p = \Psi^r(p_s)$, con $r = j \ mod \ s$.

Para encontrar el valor c de la celda (n, i, j), debemos notar primero que los sufijos bidimensionales están ordenados primero por el valor de la celda que los define, (n, i, j), al igual que en el CSA de Sadakane. Así, en este caso también se tendrá que el valor c de la celda satisfará que $C[c] < p \leq C[c + 1]$, por lo que podemos determinar el valor de c realizando una búsqueda binaria en C que tomará tiempo $O(\log \sigma)$.

Para reconstruir la celda de la derecha calculamos $p' = \Psi[p]$, que por construcción de Ψ es la posición en el arreglo de sufijos que apunta a la nueva celda. Es decir, $A[p''] = (n, i, j + 1)$. Para determinar el color c' de esta nueva celda basta realizar nuevamente una búsqueda binaria en C. Aplicando Ψ sucesivas veces a partir de p' reconstruimos la primera fila de la imagen. Para reconstruir las siguientes $r - 1$ filas de la imagen realizamos el mismo procedimiento, a partir de $p'_s = ISA[n, i + 1, \lfloor j/s \rfloor] \ldots ISA[n, i + m_r - 1, \lfloor j/s \rfloor]$. De esta manera, podemos reconstruir la imagen buscada en tiempo $O(m_r(m_c + s) \log \sigma)$ más $O(m_r(m_c + s))$ accesos a Ψ.

4.2. Búsqueda de patrones

Para poder buscar ocurrencias de una subimagen en nuestra imagen (o colección de ellas) indexada, simulamos una búsqueda binaria sobre el arreglo de sufijos, a pesar de no tenerlo. Para esto, debemos ser capaces de leer los L-caracteres siguientes a algún $A[p] = (n, i, j)$ dado sólo p, puesto que no contamos ni con A ni con la imagen. El procedimiento es similar

al de la sección anterior, pero esta vez tampoco conocemos la posición (n, i, j) que estamos reconstruyendo. Para determinar el valor c de la celda (n, i, j) debemos encontrar aquel c que satisfaga $C[c] < p \leq C[c+1]$, mediante una búsqueda binaria en C que tomará tiempo $O(\log \sigma)$. Si necesitáramos obtener el color de las celdas $(n, i, j+1)$, $(n, i, j+2)$, etc. simplemente realizamos el mismo procedimiento para $p' = \Psi[p]$, $p'' = \Psi[p']$, y así sucesivamente. Sin embargo, para poder leer los L-sufijos también debemos ser capaces de movernos desde la celda (i, j) hacia la celda $(i+1, j)$. Para esto, aplicamos la función Ψ sucesivamente, a lo más s veces, moviéndonos de esta manera hacia la derecha en la imagen (virtualmente), hasta que encontremos una columna marcada que corresponda a p^*, es decir, $B[p^*] = 1$. Como estamos en una posición muestreada, podemos determinar la posición en la imagen en la que estamos mediante $SA[rank(B, p^*)] = (n, i, j^*)$, lo cual indica que llegamos a la posición $(i, j^* * s)$ de la imagen n. Con esta información podemos consultar al arreglo de sufijos invertido de la imagen, $ISA[n, i+1, j^* - 1]$, lo cual nos dará el q^* tal que $A[q^*] = (n, i+1, (j^* - 1) * s)$. A partir de ahora, para reconstruir las celdas de la imagen el procedimiento es idéntico al de la sección anterior: a partir de q^* aplicamos Ψ sucesivamente (a lo más s veces) hasta llegar al q que corresponda a la celda buscada, $(i+1, j)$. Cabe notar que podemos utilizar ISA para movernos directamente a la fila $i + l$, comenzando en $ISA[n, i + l, j^* - 1]$. Una optimización para reducir el número de veces que aplicamos Ψ consiste en almacenar, durante el proceso de leer las celdas de los L-sufijos para l crecientes un registro $front$ con los valores p_r tales que $A[p_r] = (n, i + r, j + l)$. Así, cuando incrementamos el valor de l, nos basta con calcular Ψ una sola vez para cada valor de $front$, y utilizar el método recién descrito para determinar el nuevo p_{l+1}, primero obteniendo la celda $(i + l + 1, j)$, luego aplicando Ψ para obtener los valores de la fila de abajo del L-sufijo, y finalmente restablecer el invariante, con $A[p_{l+1}] = (i + l + 1, j + l + 1)$. Cabe notar que con el orden que se definió para leer los L-sufijos, determinar la nueva fila inferior resulta más costoso que determinar la nueva fila de la derecha. El proceso descrito se ejemplifica en la Figura 4.1, a la derecha. Así, la búsqueda de un patrón de $m \times m$ patrones se lleva a cabo en tiempo $O(m(m + s) \log N)$ accesos a Ψ [18].

4.3. Localizando las ocurrencias

El resultado de la búsqueda binaria descrita en la sección anterior es un intervalo en el arreglo de sufijos. Como no tenemos acceso al arreglo de sufijos, no podemos simplemente localizar la ocurrencia p como $A[p]$, debemos determinar $A[p]$ para todo $p \in sp, \ldots, ep$ utilizando Ψ.

Para ello, simplemente aplicamos $\Psi(p)$ consecutivamente hasta que el valor de p esté marcado en B. De esta manera, cuando hayamos aplicado Ψ k veces hasta llegar a la posición $(n, i, j * s)$ sabremos que la ocurrencia está en $A[p] = (i, j * s - k)$

4.4. Espacio y compresión

Las estructuras de muestreo requieren $N + 2(N/s) \log N$ bits para B, ISA, y SA, lo cual puede ser $O(N)$ si utilizamos un muestreo con $s = O(\log N)$. El arreglo C en principio requiere utilizar $\sigma \log N$ bits, aunque existen técnicas para reducir este tamaño en caso de que σ sea mucho mayor que N. Otro problema ocurre con la función Ψ, la cual en principio requiere tanto espacio como el arreglo de sufijos mismo, $N \log N$ bits. Para el caso unidimensional se mostró que ésta puede ser representada utilizando $NH_0 + O(N \log \log \sigma)$ bits, gracias a que la función Ψ es creciente en los intervalos que representan sufijos que comienzan con el mismo carácter. Sin embargo, en el caso bidimensional, no es cierto que un L-sufijo sea sufijo de otro L-sufijo, por lo que no existe garantía de que si dos celdas (i, j) y (i', j') tienen el mismo carácter, y el L-sufijo definido por (i, j) es lexicográficamente menor que aquel definido por (i', j') se tenga que el L-sufijo definido por $(i, j + 1)$ sea lexicográficamente menor que aquel definido por $(i', j' + 1)$. No obstante lo anterior, dado el orden que se definió para leer los L-sufijos, si dos celdas (i, j) y (i', j') tienen el mismo carácter, pero las celdas $(i, j + 1)$ y $(i', j' + 1)$ son diferentes, entonces Ψ será creciente. Notar que el orden de lectura de las celdas para formar los L-caracteres está hecho pensando en Ψ, que avanza hacia la derecha. Por esto es que el segundo carácter en ser leído para formar el L-sufijo es justamente el que está inmediatamente a la derecha del carácter que define al L-sufijo.

Además, debido a la homogeneidad espacial, el L-sufijo definido por (i, j) debiese ser similar a aquél definido por $(i, j + 1)$, por lo que ambos L-sufijos debiesen estar cerca en A. Así $\Psi[p]$ podría ser cercano a p, por lo que valores consecutivos de Ψ no debiesen diferir

demasiado. Los dos puntos expuestos anteriormente permiten esperar razonablemente que la diferencia $\Psi[p] - \Psi[p-1]$ sea un número pequeño en muchos casos, por lo que en la práctica podría resultar de utilidad utilizar algún método de codificación diferencial, usando por ejemplo Elias-δ o Elias-γ.

4.5. Implementación

4.5.1. Supuestos

Para la implementación de esta estructura, y las que se presentan en el capítulo siguiente, supondremos que cada píxel de la colección a indexar está o bien en formato RGB de 24 bits, o bien en escala de grises en el rango $0 \ldots 255$, o bien en blanco y negro. De esta manera, siempre podemos almacenar cada píxel en una celda utilizando una variable de 32 bits. En el caso de las imágenes en formato RGB, almacenaremos los colores entrelazados. Esto es, los tres bits más significativos de una celda corresponderán a la concatenación del bit más significativo de cada uno de los componentes de color de la celda (primer plano), los siguientes tres bits de la celda corresponderán a la concatenación del segundo bit más significativo de cada uno de los componentes del color (segundo plano) , y así sucesivamente hasta el octavo plano.

En la práctica, el período de muestreo de la imagen será parametrizable. De este modo se puede permitir un compromiso entre el espacio requerido por el índice y el tiempo en responder las consultas.

4.5.2. Compresión de Ψ

La representación de Ψ es fundamental en esta estructura, pues es el elemento que requiere mayor espacio, y el costo de accesar elementos de Ψ impactará sobre todos los tiempos, pues es la función que nos permite movernos en la colección de imágenes, sin tener acceso a ellas ni al arreglo de sufijos. Para almacenar Ψ se implementaron diferentes métodos de compresión, los cuales fueron comparados en la práctica en base a su requerimiento de espacio y de tiempo.

Para almacenar Ψ se calculan las diferencias entre los elementos $\Psi[p] - \Psi[p-1]$ y se almacena un muestreo a intervalos de tamaño t de los valores de Ψ. De éste modo, para recuperar un valor de Ψ debemos acceder al muestreo anterior al valor buscado y decodificar a lo más t diferencias.

Códigos Delta. La codificación Elias-δ planteada en la Sección 2.5.1 es eficiente para codificar números enteros positivos pequeños. Como en nuestro caso se codificarán tanto valores positivos como negativos, se debe definir una biyección entre el conjunto de números a codificar y un conjunto de números positivos.

Códigos Densos (s,c). Utilizaremos una variante de los SCDC de manera de no necesitar la tabla de símbolos. Al igual que la codificación Elias-δ, los códigos densos (s, c) codifican enteros positivos, por lo que debemos realizar un mapeo entre los números a codificar y un conjunto de enteros positivos análogo al realizado para la codificación Elias-δ. La base de la compresión de imágenes es la homogeneidad local, por lo que resulta esperable que los sufijos definidos por dos posiciones cercanas sean cercanos. De lo anterior, es esperable que las diferencias en la función Ψ sean números con valor absoluto pequeños con frecuencia más alta. Por esto, en vez de ordenar los símbolos por frecuencia, decidimos asignar directamente los códigos más pequeños a los números más pequeños. Es decir, realizamos el mismo procedimiento de la Sección 2.5.3, pero sin ordenar los números por frecuencia, y por ende evitando el sobrecosto de la tabla de símbolos.

Codificación de Huffman. Otra posibilidad es codificar las diferencias consecutivas de los valores de Ψ utilizando códigos de Huffman. Como la diferencia entre dos valores consecutivos de Ψ puede tomar muchos valores distintos, la tabla de asignación de símbolos podría tener un tamaño demasiado grande. Para solucionar este problema, al implementar este método de codificación se define un parámetro *maxHuff*, el cual indica el máximo valor a codificar. Los valores mayores que *maxHuff*, así como los valores negativos, serán codificados con un carácter de escape, y a continuación se almacenará el código binario correspondiente al número que se está codificando.

Codificación de Huffman Run Length. Este método da cuenta de las secuencias de números consecutivos que aparecen en Ψ. Para ello, calculamos el valor de $\Psi[p] - \Psi[p-1]$. Si esta diferencia es distinta de 1, se codificará con Huffman. En el caso en que $\Psi[p]-\Psi[p-1] = 1$ se asume que estamos en una secuencia de números consecutivos en Ψ, por lo que se recuerda la posición p y se sigue avanzando encontrar la posición p' dónde termina la secuencia creciente en Ψ. Es decir, se busca el primer p' posterior a p tal que $\Psi[p'] - \Psi[p'-1] \neq 1$. Luego se almacena el 1 y el largo de la secuencia $p' - p$. Al igual que en la codificación de Huffman, se trabaja con el parámetro *maxHuff*, codificando los valores mayores que *maxHuff* y los valores

negativos con un código de escape, y su código binario.

Codificación de Huffman Run Length por Grupos. Consideramos una variante de la solución anterior, utilizada en [2]. El procedimiento es similar, pero esta vez no se utiliza uno, sino varios caracteres de escape. Se utiliza un parámetro *maxHuff*, número máximo a codificar con Huffman, utilizándose cuatro grupos de códigos distintos:

- Números grandes: Para representar los números mayores que *maxHuff* se reservan 32 códigos de escape. Para codificar alguno de éstos números, se utilizará el código de escape correspondiente al largo en bits de su representación binaria, seguido de la representación binaria del número a codificar.

- Números negativos: Para representar los números negativos se reservan otros 32 códigos de escape. Para codificar un número negativo se utilizará el código de escape correspondiente al largo en bits de la representación binaria del valor absoluto del número, seguido por la representación binaria del mismo.

- Largos de las secuencias consecutivas. Si bien las secuencias consecutivas no tienen un largo acotado, el largo máximo que nos interesa codificar está acotado por el período de muestreo t.

- Incrementos frecuentes: Son todas las diferencias entre 2 y *maxHuff*. Estos valores se codificarán directamente con el código de Huffman correspondiente.

Así, se puede ver que la estructura para codificar requiere un árbol de Huffman para códigos entre 1 y $maxHuff\ +64+t$

4.5.3. Manejo del color

Otro problema que se presenta es que las técnicas de autoindexación sobre las cuales se basa esta estructura no han sido diseñadas para manejar alfabetos extremadamente grandes (por ejemplo, si consideramos imágenes en formato RGB de 24 bits, se tiene que el tamaño del alfabeto es $\sigma = 2^{24} = 16.777.216$).

Una forma de mejorar el desempeño es realizar un mapeo entre los colores que realmente aparecen en la imagen, y el alfabeto completo. Para ello se puede utilizar un arreglo de bits $C_\sigma[1, \sigma]$ en el cual se indiquen con un 1 los colores que están efectivamente presentes en la

imagen. De esta manera, se puede trabajar con un alfabeto más pequeño, donde el color c será representado por $c' = rank(C_\sigma, c)$. Cuando se requieran recuperar los colores originales, simplemente calculamos $c = select(C_\sigma, c')$. Dotando a este arreglo de las estructuras de *rank* y *select*, se requieren $\sigma + o(\sigma)$ bits.

En este escenario, para representar C utilizaremos una secuencia binaria $C_I[1, N]$ donde sólo los bits en las posiciones $C[c]$ están en 1. Es decir, para cada valor de c presente en la imagen, interpretamos $C[c]$ como la posición correspondiente a c en el arreglo C_I. Los colores que no están presentes en la imagen colisionan en el mismo 1 en este arreglo con el color presente más cercano menor a él, pues la cantidad de celdas menores a ellos es la misma. Dotando a este arreglo de las estructuras para *rank* y *select*, éste requiere $N + o(N)$ bits. En este contexto, para determinar el valor c de una celda (n, i, j) tal que $A[p] = (n, i, j)$ ya no podemos realizar la búsqueda binaria en C tal que $C[c] < p \leq C[c + 1]$. En su lugar, simplemente determinamos $c = rank(C_I, p)$, con lo cual además obtenemos una mejora en el tiempo, pues la búsqueda binaria toma tiempo $O(\log \sigma)$ mientras que la consulta de *rank* toma tiempo $O(1)$.

Utilizando la solución práctica de González et al. [10], las estructuras adicionales para *rank* y *select* ocupan un 37.5 % de espacio extra, proporcionando tiempos satisfactorios para *rank* y *select*. En nuestro caso, dependiendo del valor de σ se define en el momento de construcción del índice si resulta más conveniente, en términos de espacio, la representación plana de C utilizando $N \log \sigma$ bits, o la representación mapeada que requiere $1.375N + 1.375\sigma$ bits.

Capítulo 5

Autoíndice Basado en WT-FMI

5.1. Indexando por filas

En esta sección presentamos una manera de indexar imágenes utilizando un autoíndice para texto unidimensional sobre la secuencia producida al concatenar las filas de la imagen, planteada originalmente en [18].

La idea consiste básicamente en almacenar un autoíndice de las filas de la colección de imágenes a indexar, de manera que el espacio sea proporcional al espacio que requiere almacenar la imagen de manera comprimida, y a la vez dar soporte a las consultas de acceso a subimágenes y búsqueda de patrones. Así, para accesar una subimagen, utilizamos el autoíndice de las filas para accesar a cada una de las filas que conforman la subimagen. Para buscar un patrón, buscaremos en el autoíndice cada una de sus filas del patrón. La estructura que se plantea es básicamente la del WT-FMI (Sección 3.3), pero utilizando un mecanismo de muestro más apropiado para secuencias bidimensionales.

Consideremos una colección de imágenes I_k de $N_k = R_k \times C_k$ cada una, con $N = \sum_k N_k$. Sea T_I la concatenación de todas las filas de la imagen I, una secuencia de tamaño $N_I = R_I \times C_I$. Sea T la concatenación de todos los T_I de la colección, concatenando al final un marcador único \$ para que las comparaciones lexicográficas estén bien definidas.

Almacenaremos la transformada de Burrows Wheeler T^{BWT} de T representada por un Wavelet Tree, y la función C que para cada color c del alfabeto nos permite saber el número total de celdas cuyo color es menor a c, mediante $C(c)$.

Con estas dos estructuras ya podemos simular LF, la función que nos permite movernos (virtualmente, en el arreglo de sufijos) de una celda a la celda que está a su izquierda. Para ello, consideremos que tenemos una posición p en el arreglo de sufijos que apunta a la celda

(n, i, j), y queremos determinar la posición p' en A que apunta a $(n, i, j-1)$. De la Sección 2.4, sabemos que podemos obtener $p' = LF(p) = C(c) + rank_c(T^{BWT}, p)$, donde $c = T_p^{BWT}$.

Para poder alcanzar tiempos de búsqueda y de acceso satisfactorios, muestreamos la imagen a intervalos regulares de manera similar a la realizada en la sección anterior. Sea s el paso según el cual sea realizará el muestreo. Entonces, se almacenarán para cada celda $I[i, j \cdot s]$, para cada imagen, el valor $ISA[n, i, j] = p$ tal que $A[p] = (n, i, j \cdot s)$ (asimismo, también se muestrean la primera y la última columna de cada imagen). Lo anterior es un muestreo del arreglo de sufijos invertido. Los valores p almacenados se marcan en un arreglo de bits $B[1, N]$. También almacenamos un arreglo de sufijos muestreado $SA[rank(B, p)] = (n, i, j)$.

Las estructuras de muestreo para ISA y SA requieren $2(N/s) \log N$ bits. Utilizando $s = \log^{1+\epsilon} N / \log \sigma$, estas estructuras requieren $O(N \log \sigma / \log^{\epsilon} N)$ bits. Para almacenar B podemos utilizar la solución de Raman Raman y Rao y utilizar $O(N \log \sigma / \log^{1+\epsilon} N) \log\ N = O(N \log \sigma / \log^{\epsilon} N)$ bits, pues B tiene $(N \log \sigma / \log^{1+\epsilon} N)$ bits en uno. Por otro lado, almacenando T^{BWT} con un Wavelet Tree ocuparemos $NH_k(T)$. De esta manera, el índice requiere $NH_k(T) + o(N \log \sigma)$ bits.

5.1.1. Acceso a subimágenes

El procedimiento de acceso a una subimagen S de tamaño $M = m_r \times m_c$, ubicada en (n, i, j) corresponde simplemente a reconstruir las m_r filas de S de la manera indicada en la Sección 3.3.4.

Como la función LF nos permite movernos hacia la izquierda en la imagen, debemos buscar la posición muestreada siguiente más cercana a $(n, i, j + m_c)$ y ubicarla en el arreglo de sufijos mediante ISA. Esto es, $p' = ISA[n, i, \lceil (j + m_c)/s \rceil]$. A partir de esta posición, calculamos $p = LF^d(p')$ donde $d = s\lceil (j + m_c)/s \rceil - (m_c + j)$ es la distancia entre la muestra más cercana y la celda de más a la derecha que queremos reconstruir. Es decir, sabemos que p satisface $SA[p] = (n, i, j + m_c)$, y sabemos que $I[n, i, j + m_c - 1]$, la celda que queremos reconstruir, es T_p^{BWT}. Aplicamos $LF(p)$ m_c veces para reconstruir la fila i. Repitiendo este proceso m_r veces hemos reconstruido la subimagen buscada. Este proceso requiere tiempo $O(m_r(m_c \log \sigma + \log^{1+\epsilon} N)) = O(M \log \sigma + m_r \log^{1+\epsilon} N))$, pues LF y T^{BWT} requieren tiempo $O(\log \sigma)$.

5.1.2. Búsqueda de patrones

Para buscar las ocurrencias de un patrón P de tamaño $M = m_r \times m_c$, debemos realizar de manera independiente las m_r búsquedas correspondientes a cada una de las filas del patrón utilizando el índice de las filas de la imagen concatenadas, tal como se mostró en la Sección 3.3.1. Cada ocurrencia (n, i, j) de la fila P_k representa que la posición $(n, i - k, j)$ es un candidato a esquina superior izquierda de una ocurrencia de P.

Para determinar las ocurrencias reales de P en I debemos determinar la intersección entre los candidatos aportados al buscar cada fila del patrón. Sin embargo, este proceso requeriría localizar los candidatos a ocurrencia de cada fila, lo cual podría resultar muy costoso en el caso en que una fila del patrón buscado apareciera muchas veces en la colección, mientras otras están pocas (o ninguna) vez.

Una solución efectiva es utilizar el hecho de que el WT-FMI puede contar las ocurrencias de un patrón de tamaño m_c en tiempo $O(m_c \log \sigma)$.

Contamos las ocurrencias de cada fila y las ordenamos de menos a más ocurrencias. Esto toma tiempo $O(M \log \sigma + m_r \log m_r)$. Luego localizamos los candidatos aportados por la fila con menos ocurrencias, occ_0, y los almacenamos en una tabla de hash en la cual mantendremos los $cand_i$ candidatos en curso considerando las i filas que hemos revisado hasta el momento, por lo que $cand_0 = occ_0$. Esto toma tiempo $O(occ_0 \log^{1+\epsilon} N)$. Luego recorremos las filas de menos a más ocurrencias y validamos los candidatos que tenemos en la tabla de hash con la siguiente fila. Para ello hay dos estrategias posibles:

- Si la siguiente fila tiene un número de ocurrencias occ_i relativamente bajo comparado con $cand_{i-1}$, simplemente utilizamos el índice para localizarlas en tiempo $O(occ_i \log^{1+\epsilon} N)$, y las buscamos en la tabla de hash con los candidatos en curso, insertando los $cand_i$ nuevos candidatos en una nueva tabla de hash que reemplazará a la anterior.

- Si en cambio occ_i es relativamente grande comparado con $cand_{i-1}$ resultará más conveniente reconstruir esa fila para cada candidato, y compararla con el patrón buscado. En caso de coincidir, el candidato se agrega a una nueva tabla de hash que reemplazará a la actual. Esto puede realizarse en tiempo $O(cand_{i-1}(m_c \log \sigma + \log^{1+\epsilon} N))$.

Multiplicando por las constantes adecuadas, se puede determinar cuál estrategia usar. Resulta útil notar que $occ_{i+1} \geq occ_i$ pues las filas fueron ordenadas de menos a más ocurrencias y

que $cand_{i+1} \leq cand_i$ pues en cada paso intersectamos los candidatos en curso con los aportados por una nueva línea. Debido a lo anterior, en el momento en que resulte más conveniente la estrategia de reconstruir las filas de los candidatos en vez de localizar las ocurrencias, mantendremos esta estrategia para todas las filas restantes. El tiempo en localizar las ocurrencias de P en I, en el peor caso, corresponde a localizar las ocurrencias de la fila con menos ocurrencias del patrón, y luego reconstruir completamente cada uno de los candidatos. Así, nuestro algoritmo requiere tiempo $O(M \log \sigma + m_r \log m_r + occ_0(M \log \sigma + m_r \log^{1+\epsilon} N)) = O(occ_0(M \log \sigma + m_r \log^{1+\epsilon} N))$, en el caso en que $occ_0 > 0$ (en el caso en que $occ_0 = 0$, en tiempo $O(M \log \sigma)$ se determina que el patrón no aparece en la colección). El Algoritmo 3 muestra el seudocódigo.

```
FM2D-Busqueda(P)
for k ← 1 to m_r do
    intervalo[k] ← FMSearch(P_k);
Sort(intervalo);
candidatos ← FMLocate(intervalo[1]);
cands ← size(intervalo[1]);
for k ← 2 to m_r do
    occs ← size(intervalo[k]);
    if cands(s/2 + m_c log σ) > occs × s/2 then
        for mp ∈ intervalo[k] do
            l ← FMLocate(mp);
            if l ∈ candidatos then
                insert(l, candidatos');
    else
        for l ∈ candidatos do
            if FMAccess(l) = P_{l.fila} then
                insert(l, candidatos');
    candidatos ← candidatos';
    cands ← size(candidatos);
return candidatos
```

Algoritmo 3 : Búsqueda de las ocurrencias de P en la colección indexada por filas. Una vez que el número de candidatos es suficientemente bajo con respecto al número de ocurrencias de la siguiente fila se comienzan a chequear las filas restantes de los candidatos.

5.2. Indexando por bandas

La estrategia presentada anteriormente dará cuenta de la compresibilidad de las filas de la imagen, sin embargo, no se está sacando ventaja de la posible compresibilidad de las columnas del texto. Una idea para que la estructura dé cuenta de esto es no indexar el texto por filas, sino que por pares de filas. En este caso, la secuencia que deberá indexar el autoíndice tendrá un largo menor (la mitad), pero los caracteres a indexar serán más largos (el doble). Esta estrategia permitirá sacar ventaja de la homogeneidad espacial de las imágenes, pero presenta ciertas dificultades: cuando busquemos un patrón de tamaño $m_r \times m_c$, debemos considerar los casos en que el patrón no se encuentre perfectamente alineado con la indexación, por lo que debemos realizar dos búsquedas para determinar todas las ocurrencias del patrón. Para evitar esto, una idea interesante es realizar una doble indexación de la imagen: si en el índice de a pares recién expuesto indexamos los pares de filas $(1,2),(3,4)\ldots(n-1,n)$, también podemos indexar los pares $(2,3),(4,5)\ldots(n-2,n-1)$ de forma que ambas indexaciones se traslapen. De este modo, las ocurrencias del patrón siempre estarán alineadas con alguna de las dos indexaciones (eventualmente tendremos que realizar un chequeo extra cuando m_r sea impar). En este caso se espera que el tamaño del índice sea a lo sumo el doble (y menos cuando haya homogeneidad vertical); mientras que los tiempos de búsqueda disminuyen por lo menos a la mitad (pues ahora el alto de P es $\lfloor m_r/2 \rfloor$ y los pares de filas presentan menos ocurrencias).

Finalmente, la idea anterior puede ser generalizada indexando cada L filas, obteniendo un texto de largo N/L, en el cual cada símbolo tiene un largo de L veces el largo de los símbolos originales (en adelante consideremos unitario el largo de los símbolos originales). En éste caso, para realizar una búsqueda debemos buscar el patrón L veces, para cubrir todos los posibles alineamientos del patrón con la indexación y verificar hasta $L-1$ filas directamente en la imagen.

El caso más general consiste en indexar grupos de L filas, con un desfase de S filas, produciendo un traslape de $L-S$ filas. Esto es, utilizaremos la misma técnica de indexación de la Sección 5.1, pero esta vez indexando una colección I'_k de imágenes procesadas de $\lceil R_k/S \rceil \times C_k$, donde las celdas de la imagen procesada están dadas por:

$$I'[n,i,j] = I[n, i \times S, j] I[n, i \times S + 1, j] \ldots I[n, i \times S + L - 1, j]$$

De esta manera, los caracteres indexados serán de largo L, por lo que estarán definidos sobre un alfabeto $\Sigma_L = \{1, \ldots, \sigma_L\}$, con $\sigma_L \leq \sigma^L$ y el texto indexado tendrá largo $\lceil N/S \rceil$. La Figura 5.1 muestra un ejemplo.

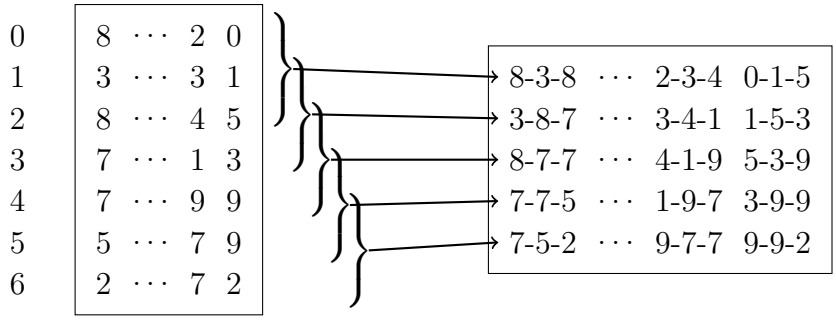


Figura 5.1: Ejemplo de indexación por grupos para $L = 3$, $S = 1$. A la izquierda se ve la imagen original y a la derecha la imagen virtual que será indexada. Se observa que las líneas 1 y 2 de la imagen original aparecen dos veces en la imagen virtual, en las filas 0 y 1, mientras que las líneas 3 y 4 aparecen tres veces, en las filas 1, 2, 3, y 2, 3, 4, respectivamente.

5.2.1. Acceso a subimágenes

Para reconstruir una subimagen de tamaño $m_r \times m_c$ debemos identificar las L-bandas indexadas que cubren la subimagen buscada. Para ello basta con identificar la primera L-banda que cubra parte de la subimagen y luego obtener las $\lceil (m_r - L)/S \rceil$ siguientes para recuperar la subimagen completa.

Sin embargo, en el caso en que $S \leq L/2$, habrán L-bandas totalmente redundantes que podemos omitir. Definiendo $S_{eff} = S \times \lfloor L/S \rfloor$, necesitamos reconstruir sólo $1 + \lceil (m_r - L)/S_{eff} \rceil$ L-bandas. Resulta útil notar que $S_{eff} \geq L/2$, y que $max(S, L - S + 1) \leq S_{eff} \leq L$. Reconstruir cada una de éstas bandas de largo m_c con el WT-FMI requiere tiempo $O((m_c + \log^{1+\epsilon} N/\log\sigma)\log\sigma_L)$, por lo que el tiempo de reconstruir la subimagen es $O(m_r/S_{eff}(m_c + \log^{1+\epsilon} N/\log\sigma)\log\sigma_L)$. La Figura 5.2 ejemplifica el proceso.

5.2.2. Búsqueda de patrones

Para realizar la búsqueda de patrones en este escenario, debemos preprocesar el patrón que buscaremos de la misma manera en que procesamos las imágenes al indexarlas. De ésta manera, las ocurrencias de la imagen en la colección corresponden a ocurrencias de la imagen procesada en la colección procesada.

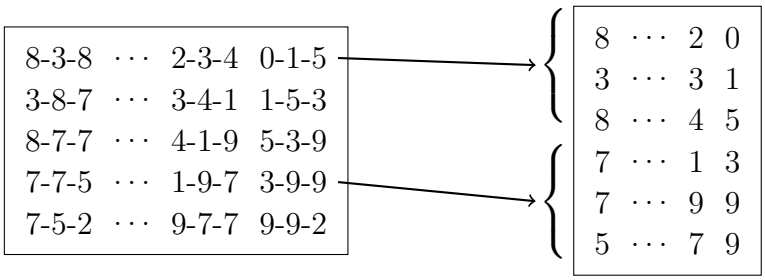


Figura 5.2: Mecanismo para reconstruir una subimagen a partir de una indexación con $L = 3$, $S = 1$, $S_{eff} = 3$. Notar que para reconstruir las primeras 6 filas de la imagen necesitamos sólo la 1^a y la 4^a L-bandas, ya que toda la información contenida en la 2^a y 3^a está también en la 1^a y la 4^a.

Para buscar las ocurrencias en I del patrón P, de tamaño $M = m_r \times m_c$, debemos, en principio, leer el patrón de la misma manera en que procesamos la imagen: primero las filas $1 \dots L$ las consideramos como una primera fila y la buscamos en el índice. Luego las filas $S + 1 \dots S + L$ las consideramos como una segunda fila y la buscamos en el índice, y así sucesivamente hasta haber cubierto todo el patrón.

Sin embargo, podemos aplicar aquí la misma optimización que para accesar las imágenes: cuando hay L-bandas completamente redundantes, no necesitamos buscarlas en el índice, y por tanto debemos realizar la búsqueda de $1 + \lceil (m_r - L)/S_{eff} \rceil$ L-bandas. Además, si $m \mod S_{eff} \neq 0$, debemos chequear manualmente las $m \mod S_{eff}$ filas del patrón que no fueron cubiertas en la búsqueda de L-bandas.

Debemos tener en cuenta que el patrón podría no estar exactamente alineado con las L-bandas que se indexaron, por lo cual debemos leer el patrón comenzando en las filas $1 \dots S_{eff}$ y realizar para cada uno de estos alineamientos una búsqueda tal como en la Sección 5.1.2. Así, para poder buscar un patrón, se necesita que tenga al menos $L + S_{eff}$ filas, para poder leer una L-banda y buscarla en el índice en cada alineamiento.

El Algoritmo 4 muestra el proceso.

De esta manera, la búsqueda en el caso $S = L$ requiere tiempo $O(L\ occ_0^{(L)}((M/L) + (m_r/L)\log^{1+\epsilon} N/\log\sigma)\log\sigma_L) = O(L\ occ_0^{(L)}(M\log\sigma + m_r\log^{1+\epsilon} N))$, donde $occ_0^{(L)}$ es el número de ocurrencias de la L-banda con menos ocurrencias. Esto permite ver que el tiempo de búsqueda será a lo sumo L veces el tiempo de búsqueda del caso en que se indexa por filas. Sin embargo, como en promedio $occ_0^{(L)}/N \approx (occ_0/N)^L$, se espera que en la práctica el aumento en el tiempo no sea demasiado, pudiendo incluso disminuir.

```
FM2DBandas-Busqueda(P)
for fase ← 0 to S_eff − 1 do
    for i ← 0 to ⌈m_r/S_eff⌉ do
        for j ← 0 to m_c do
            P'[i, j] = P[fase+i∗S, j]P[fase+i∗S+1, j] . . . P[fase+i∗S+L−1, j];
    pre_candidatos ← FM2D − Busqueda(P');
    for i ← 1 to size(candidatos) do
        if chequear_filas(candidatos[i]) then
            ocurrencias ← ocurrencias ∪ candidatos[i]
return ocurrencias
```

Algoritmo 4 : Procesamiento del patrón P para realizar la búsqueda por L-bandas. Debemos buscar las S_{eff} maneras distintas de leer el patrón utilizando el índice de las L-bandas. Además, debemos chequear a lo más S_eff filas no cubiertas por las L-bandas.

En el caso en que se indexe con redundancia S, como $L/2 \leq S = S_{eff} < L$, el tiempo de búsqueda es $O(S\ occ_0^{(L)}((M/S_{eff})+(m_r/S_{eff})\log^{1+\epsilon} N/\log\sigma)\log\sigma_L) = O(S\ occ_0^{(L)}(M\log\sigma+ m_r\log^{1+\epsilon} N))$, pues $S_{eff} = \Theta(L)$. Esto ilustra claramente la ganancia de tiempo al indexar con redundancia.

5.3. Espacio y compresión

Consideremos primero los casos sin redundancia, es decir $S = L$. En éste caso, las estructuras que utilizamos corresponden a un WT-FMI para un texto T_L de largo $\lceil N/L\rceil$ con caracteres de largo L. Esto último equivale a decir que el alfabeto de la imagen indexada puede aumentar exponencialmente en el peor caso, llegando a $\sigma_L = \sigma^L$. Sin embargo, dependiendo de la homogeneidad vertical de la imagen este crecimiento podría ser menor. Así, el índice requerirá $\lceil N/L\rceil H_k(T_L) + o(N/L\log\sigma_L) = \lceil N/L\rceil H_k(T_L) + o(N\log\sigma)$ bits, que en el peor caso es $NH_k(T) + o(N\log\sigma)$ bits, para $k \leq \alpha\log_\sigma N$.

De manera similar, el caso con redundancia corresponde a un WT-FMI para un texto $T_{S,L}$ de largo $\lceil N/S\rceil$ con caracteres de largo L. En este caso, el índice requerirá $\lceil N/S\rceil H_k(T_{S,L}) + o((N/S)\log\sigma_L) = \lceil N/S\rceil H_k(T_{S,L})+o((NL/S)\log\sigma)$ bits, que en el peor caso es $(NL/S)H_k(T)+ o((NL/S)\log\sigma)$ bits, para $k \leq \alpha\log_\sigma N$. Esto ilustra el empeoramiento del espacio al usar redundancia.

5.4. Implementación

5.4.1. Manejo del color

De la misma manera que en la implementación del CSA-2D (Sección 4.5.3) , cuando el tamaño del alfabeto sea demasiado grande, el arreglo C será implementado sobre colores mapeados. En este caso, para representar C utilizaremos la secuencia binaria $C_I[1, N]$ donde sólo los bits en las posiciones $C(c)$ están en 1 y el arreglo de bits $C_\sigma[1, \sigma]$ en el cual se indican con 1 los colores que están efectivamente presentes en la imagen, ambos dotados de las estructuras para responder *rank* y *select*. Estas estructuras requieren en total $N + o(N)$ y $\sigma + o(\sigma)$ bits respectivamente. Del mismo modo que en el CSA-2D (Sección 4.5.3) en la práctica representaremos C como un arreglo o con estas dos estructuras evaluando al momento de construir el índice cuál resulta más conveniente en términos de espacio.

En el caso en que reemplacemos C por estas estructuras, además de las representación de los colores señaladas en la Sección 4.5.3, este índice también requiere determinar el número de colores menores que c, $C(c)$. Para ello utilizamos $C(c) = select(C_I, c)$, lo cual si bien no altera los costos teóricos de tiempo, pues ambas consultas cuestan $O(1)$, en la práctica implica tiempos mayores, pues un acceso a memoria para leer $C[c]$ es muchísimo más rápido que calcular la función *select*.

5.4.2. Wavelet Tree

En la práctica existen diversas implementaciones disponibles [5] que podemos utilizar para almacenar el Wavelet Tree de la Transformada de Burrows Wheeler, T^{BWT}:

Wavelet Tree sin punteros. Consiste en almacenar los $\lceil \log \sigma \rceil$ niveles utilizando para cada uno de ellos la solución de Raman, Raman y Rao (Sección 3.1.2).

Wavelet Tree con alfabeto mapeado. Consiste en una mejora de la estructura anterior, consistente en mapear el alfabeto a un rango contiguo para evitar que el Wavelet Tree represente caracteres espurios que no aparecen en la secuencia. En nuestro caso, esta variante conlleva una mejora sólo cuando trabajamos con los colores originales.

Wavelet Tree con forma de Huffman. El Wavelet Tree original es un árbol perfectamente balanceado, sin embargo, hay trabajos prácticos que plantean cambiar su forma por la de un árbol de Huffman [12] para minimizar la redundancia entre los niveles y así alcanzar espacio cercano a la entropía de orden cero. Sin embargo, cuando el alfabeto sobre el que se

define la secuencia es muy grande, esta estructura deja de ser competitiva por el sobrecosto asociado a la tabla de símbolos de Huffman.

Como nuestra estructura necesita indexar de manera eficiente colecciones definidas sobre un alfabeto muy grande (RGB), en nuestra implementación utilizaremos el Wavelet Tree sin punteros cuando representemos C mediante C_I y C_σ, y en el caso en que representemos C mediante un arreglo, utilizaremos el Wavelet Tree con alfabeto mapeado.

5.4.3. Acceso a la Transformada de Burrows Wheeler

En teoría, para determinar el color correspondiente a una posición p en el arreglo de sufijos, se cuenta con T^{BWT}. En la práctica, sin embargo, realizar un acceso a T^{BWT} almacenado como un Wavelet Tree cuesta $\log \sigma$ consultas *rank* sobre bitmaps. Por tanto, preferiremos el método descrito en el capítulo anterior. Si C está almacenado como un arreglo, realizamos una búsqueda binaria que también cuesta $O(\log \sigma)$, sin embargo las operaciones en cada paso son menos costosas que un cálculo de *rank*. Si utilizamos la representación de C_σ y C_I, para determinar el color correspondiente a la posición p en el arreglo de sufijos basta con $c = rank(C_I, p)$ que es aún más rápido.

Capítulo 6

Resultados Experimentales y Discusión

Todos los experimentos que se presentan en este capítulo se ejecutaron en un computador con 4GB de memoria RAM, un core con dos procesadores Intel Pentium IV, cada uno de 3 GHz y 1MB de cache. El código fue compilado utilizando g++ con la opción de optimización -O9

Para probar nuestros prototipos se utilizaron cinco colecciones intentando representar distintos escenarios de aplicación.

- *Arte*[1]: Colección de ilustraciones góticas extraídas de *Les très riches heures du Duc de Berry*.
- *Mapas*[2]: Mapas de Japón hechos en el siglo XIX.
- *Astro*[3]: Colección de imágenes astronómicas de *Sloan Digital Sky Survey* (SDSS), un proyecto financiado por la NASA y la NSF, entre otros, que aspira a generar un mapa de gran parte del universo.
- *Micro*[4]: Colección que muestra distintas patologías a nivel celular, obtenidas por microscopía electrónica.
- *Fuentes*[5]: Algunas de las imágenes de la fuente *Identifont*.

En la tabla 6.1 se muestran las características relevantes de nuestras colecciones.

[1]http://ibiblio.org/wm
[2]http://digitalgallery.nypl.org
[3]http://skyserver.sdss.org/dr3/en/tools/getimg/
[4]http://visualsonline.cancer.gov
[5]http://www.identifont.com/free-fonts.html

Colección	Tipo	Imágenes	Celdas	Representación plana	Tamaño JPG
Arte	RGB	12	6.551.318	24	1,396
Mapas	RGB	10	4.365.440	24	0,842
Astro	RGB	10	9.328.999	24	2,605
Micro	Gris	13	56.583.672	8	2,935
Fuentes	Gris	9	1.119.744	8	5,515

Tabla 6.1: Colecciones de imágenes que se utilizarán en las pruebas. Se muestra el formato de color en que están las celdas, el número de celdas, y los bits por celda (bpc) requeridos por la colección en representación plana y en formato JPG (sin pérdida).

6.1. Autoíndice basado en CSA

En esta sección se muestran los resultados obtenidos para las distintas codificaciones de la función Ψ en el índice CSA-2D.

6.1.1. Compresión de Ψ

En esta sección compararemos el espacio requerido por las distintas codificaciones de la función Ψ. Recordemos que el espacio requerido por las otras estructuras del CSA-2D (las estructuras de muestreo y manejo del color) es parametrizable e independiente de la codificación de Ψ que utilicemos. Para comparar los niveles de compresión de Ψ calculamos el espacio requerido por esta función con las distintas colecciones, utilizando 8, 4 y 2 planos (es decir, los 8, 4 y 2 bits más significativos de cada celda en el caso de las colecciones en escala de grises, y los 24, 12 y 6 en el caso de las colecciones en RGB).

En la tabla 6.2 podemos ver el espacio que ocupa Ψ medido en bits por celda para los distintos métodos de codificación, utilizando el mismo período de muestreo (64) para todas. Se puede apreciar que la versión de Huffman Run Length por Grupos obtiene una compresión superior en todos los escenarios. Resulta interesante notar que la codificación de Ψ utiliza menos bits que la representación plana de la imagen en todas las colecciones a excepción de *Micro*, caso en que nuestro índice requiere más espacio que la colección original.

6.1.2. Acceso a subimágenes

En esta sección se muestran los resultados de tiempo en reconstruir subimágenes definidas a partir de posiciones aleatorias en función del tamaño del índice. El experimento consistió en elegir 1000 posiciones aleatorias y reconstruir la subimagen correspondiente a cada posición

Colección	#	Delta	Códigos S-C	Huffman	Huffman RL	Huffman RL Grupos
Arte	8	19,39	17,71	17,29	18,03	16,24
Arte	4	16,42	14,25	14,00	16,27	13,30
Arte	2	12,47	11,11	10,66	13,29	10,31
Mapas	8	13,35	12,20	11,86	13,79	11,27
Mapas	4	10,86	10,17	9,70	11,45	9,21
Mapas	2	8,91	8,71	7,90	9,52	7,52
Astro	8	20,43	18,73	19,38	19,88	17,22
Astro	4	10,14	10,32	9,69	11,93	9,39
Astro	2	5,07	6,04	4,77	5,41	4,36
Micro	8	20,79	18,42	20,79	20,78	17,06
Micro	4	13,79	12,15	11,94	15,27	11,41
Micro	2	10,17	9,47	9,00	11,77	8,59
Fuentes	8	7,83	8,11	7,17	8,04	6,84
Fuentes	4	7,83	8,11	7,17	8,04	6,84
Fuentes	2	7,54	7,69	6,68	8,02	6,44

Tabla 6.2: Bits por celda utilizados por las distintas codificaciones de Ψ, con un período de muestro de 64.

a partir de las distintas variantes del CSA-2D. Para cada implementación se varió el período de muestreo de las imágenes y de Ψ buscando la configuración que obtuviera menor tiempo de reconstrucción. Para el acceso a subimágenes pequeñas se varió el muestreo de Ψ y de la colección entre 16 y 128. Para las grandes se fijó el muestreo de la colección en 128 y se varió el muestreo de Ψ entre 8 y 128.

En las figuras 6.1 y 6.2, se muestran los resultados obtenidos con las distintas colecciones para reconstruir subimágenes de 10×10 y en las figuras 6.3 y 6.4, se muestran los resultados obtenidos para reconstruir subimágenes de 200×200. Los tiempos están expresados en milisegundos, y el espacio está expresado como fracción del espacio que ocuparía la representación plana de la colección.

En general, la codificación de Huffman Run Length por Grupos resulta ser la más competitiva, a excepción de la colección *Fuentes* (y a veces, *Mapas*), en las cuales la codificación que utiliza Códigos Delta resulta la más competitiva. Esto podría atribuirse a que los Códigos Delta son eficientes para codificar números pequeños, y las colección *Fuentes* y *Mapas* tienen grandes zonas de un mismo color, con lo cual se esperan grandes zonas monótonas en Ψ.

Si bien se esperaba que los códigos densos (s, c) resultaran ser una buena alternativa [18] para codificar, avalados por los buenos tiempos obtenidos en otros escenarios [1] , en nuestros

experimentos el espacio requerido al utilizar esta codificación resulta competitivo, pero los tiempos de acceso no son buenos. Esto se debe a que en nuestro caso, a diferencia de sus usos anteriores, al utilizar códigos (s, c) los tamaños óptimos de símbolos (en la mayoría de los casos menores que 8 bits) implican realizar lecturas de bits no necesariamente alineados con un byte.

6.1.3. Conteo de ocurrencias

Este experimento consistió en medir el tiempo necesario para determinar las ocurrencias de un patrón para distintas variantes del índice CSA-2D en función del tamaño del índice, para cada colección.

Se extrajeron 1000 patrones de manera aleatoria y se realizó la búsqueda sobre cada una de las variantes, construyendo el índice variando los parámetros de muestreo de la imagen y de Ψ buscando la combinación que menor tiempo tomara para la consulta a realizar. Los resultados para patrones de 10×10 se pueden ver en las figuras 6.5 y 6.6, y los resultados para patrones de 200×200 se pueden ver en las figuras 6.7 y 6.8. Los tiempos están expresados en milisegundos, y el espacio está expresado como fracción del espacio que ocuparía la representación plana de la colección.

Al igual que en la sección anterior, la codificación de Huffman Run Length por Grupos resulta la más competitiva, a excepción de la colección *Fuentes* (y a veces *Mapas*), en las cuales utilizar Códigos Delta resulta la mejor opción.

6.1.4. Localización de ocurrencias

La consulta de conteo de ocurrencias entrega como resultado un rango de posiciones en el arreglo de sufijos. En esta sección se exponen los tiempos necesarios para determinar a qué posición en la imagen corresponden dichas ocurrencias. El experimento consistió en seleccionar un conjunto de 1000 posiciones aleatorias del arreglo de sufijos y determinar a qué posición del índice corresponden.

En las figuras 6.9 y 6.10 se muestra el tiempo de localización de ocurrencias en función del tamaño del índice para las distintas variantes del índice. Los tiempos están expresados en milisegundos, y el espacio está expresado como fracción del espacio que ocuparía la representación plana de la colección.

Colección \L		1	2	3
Arte, 8 planos	Sam	5,49		
	Col	4,74		
	Wt	14,37		
	TOT	24,62		
Mapas, 8 planos	Sam	5,41		
	Col	6,46		
	Wt	8,45		
	TOT	20,32		
Micro, 8 planos	Sam	5,40	2,70	1,80
	Col	0,00	0,04	0,84
	Wt	6,77	6,17	5,90
	TOT	12,17	8,90	8,54
Fuentes, 8 planos	Sam	5,51	2,76	1,84
	Col	0,00	0,75	20,43
	Wt	2,18	1,83	1,52
	TOT	7,71	5,34	23,78
Astro, 8 planos	Sam	5,43		
	Col	3,73		
	Wt	11,36		
	TOT	20,52		

Tabla 6.3: Espacio utilizado, expresado en bpc, por cada uno de los componentes de nuestro índice construido sobre las colecciones de prueba que se utilizan en está sección. Sam corresponde al muestreo de la imagen, utilizando un período de muestreo 32, Col es el espacio requerido por la representación de C, y Wt es el espacio que ocupa el Wavelet Tree.

Nuevamente, la codificación de Huffman Run Length por Grupos resulta la más competitiva, a excepción de la colección *Fuentes* (y a veces *Mapas*), en las cuales la codificación que utiliza Códigos Delta resulta la mejor opción.

6.2. Autoíndice basado en Índice FM

En esta sección expondremos los resultados obtenidos por nuestra estructura de indexación por filas y bandas, basada en el Índice FM. $L = 1$ corresponde al caso de indexación por filas. Es importante notar que debido a nuestros supuestos de implementación, no es posible usar $L > 1$ para colecciones RGB, ni $L > 3$ para colecciones en escala de grises.

6.2.1. Acceso a subimágenes

Este experimento consistió en elegir 1000 posiciones aleatorias y reconstruir la subimagen correspondiente a cada posición a partir de nuestro índice, variando el período de muestreo de las imágenes. Las figuras 6.11 y 6.12 muestran los tiempos de acceso para subimágenes pequeñas, y las figuras 6.13 y 6.14 muestran los tiempos de acceso para subimágenes grandes.

En ambos casos se muestra el tiempo de reconstrucción en función del tamaño del índice, éste último expresado como fracción de la representación plana de la colección original. Se observa que si bien esta estructura puede requerir menos espacio que el CSA-2D (sobre todo en las colecciones en escala de grises), los tiempos de acceso son mucho mayores que los del CSA-2D, pudiendo llegar a ser 10 veces más lento. Para un análisis de las variaciones del tamaño del índice al variar los valores de L ver la Sección 6.3.1 y la Sección 6.3.2.

6.2.2. Búsqueda de patrones

Es importante recordar que éste índice, a diferencia del CSA-2D, no soporta la operación de conteo de ocurrencias. El experimento que se presenta en esta sección muestra el tiempo en determinar las posiciones de las ocurrencias de un patrón en la colección. En este experimento se extrajeron 1000 patrones de manera aleatoria y se realizó la búsqueda sobre nuestro índice, considerando distintos valores para el período de muestreo de la imagen. Las figuras 6.15 y 6.16 muestran los tiempos de búsqueda de patrones grandes para las distintas colecciones.

Los tiempos de búsqueda para patrones pequeños no se exponen pues esta estructura no es competitiva. Esto es debido a que este índice requiere localizar todas las ocurrencias de la fila menos frecuente del patrón, y para patrones pequeños escogidos aleatoriamente este número puede alcanzar valores extremadamente altos.

6.3. Indexación por bandas

En esta sección estudiaremos el comportamiento de la estructura de indexación por bandas de mayor tamaño. Para ello trabajaremos sólo con 2 planos en las colecciones RGB, pudiendo así leer bandas formadas hasta de 4 líneas, mientras que en las colecciones en escala de grises trabajaremos con 4 planos, permitiendo así a nuestra estructura indexar bandas compuestas hasta de 7 líneas.

6.3.1. Tamaño del índice

En la tabla 6.4 se muestra el espacio requerido por todas las combinaciones de L y S para la colección *Micro*. Se puede apreciar que en los casos con redundancia ($S \neq L$) el sobrecosto es significativo, por lo que en adelante nos centraremos en el caso $S = L$.

La tabla 6.5 muestra los tamaños del índice desglosado por componentes para los casos

S \L		1	2	3	4	5	6	7
	Sam	3,41	3,41	3,41	3,41	3,41	3,41	3,41
	Col	0,00	0,00	0,56	0,04	0,59	1,73	7,65
1	Wt	2,85	4,57	6,57	8,30	10,09	11,61	13,16
	TOT	6,27	7,97	10,54	11,74	14,09	16,74	22,23
	Sam		1,70	1,70	1,70	1,70	1,70	1,70
	Col		0,00	0,56	0,03	0,59	1,06	6,99
2	Wt		2,28	3,28	4,15	5,04	5,82	6,44
	TOT		3,99	5,54	5,89	7,34	8,59	15,13
	Sam			1,13	1,13	1,13	1,13	1,13
	Col			0,47	0,04	0,47	0,84	6,77
3	Wt			2,19	2,77	3,27	3,78	4,30
	TOT			3,79	3,94	4,87	5,76	12,20
	Sam				0,85	0,85	0,85	0,85
	Col				0,04	0,36	0,73	6,66
4	Wt				2,07	2,45	2,84	3,15
	TOT				2,96	3,66	4,42	10,66
	Sam					0,68	0,68	0,68
	Col					0,29	0,66	6,59
5	Wt					1,96	2,27	2,52
	TOT					2,94	3,62	9,79
	Sam						0,57	0,57
	Col						0,62	6,55
6	Wt						1,90	2,10
	TOT						3,08	9,22
	Sam							0,48
	Col							6,52
7	Wt							1,80
	TOT							8,81

Tabla 6.4: Bits por celda para las distintas combinaciones de S y L para la colección *Micro*, con 4 planos, muestreando la colección cada 64 posiciones.

de prueba de esta sección. Se puede apreciar que al aumentar el tamaño de las bandas el espacio requerido por el Wavelet Tree siempre disminuye. Este resultado es esperado debido a la homogeneidad vertical.

A diferencia del CSA-2D, al aumentar el tamaño de las bandas, las estructuras para el manejo del color (representación de C) se hacen relevantes. Otro factor interesante es que, si bien el muestreo de la imagen sigue sin ser un elemento dominante en cuanto al espacio que ocupa, al aumentar el tamaño de las bandas requiere aún menos espacio, por lo que es posible aumentar el muestreo sin mayor sobrecosto.

6.3.2. Acceso a subimágenes

Este experimento consistió en elegir 1000 posiciones aleatorias y reconstruir la subimagen correspondiente a cada posición a partir de las distintas variantes de nuestro índice. Para ello

Colección \L		1	2	3	4	5	6	7
	Sam	3,53	1,76	1,17	0,88			
Arte,	Col	0,00	0,02	0,49	3,74			
2 planos	Wt	2,97	2,39	2,02	1,83			
	TOT	6,50	4,18	3,69	6,47			
	Sam	3,39	1,70	1,13	0,85			
Mapas,	Col	0,00	0,03	0,52	5,45			
2 planos	Wt	2,45	1,81	1,61	1,35			
	TOT	5,85	3,54	3,27	7,66			
	Sam	3,41	1,70	1,13	0,85	0,68	0,57	0,48
Micro,	Col	0,00	0,00	0,47	0,04	0,29	0,62	6,52
4 planos	Wt	2,85	2,28	2,19	2,07	1,96	1,90	1,80
	TOT	6,27	3,99	3,79	2,96	2,94	3,08	8,81
	Sam	3,44	1,72	1,14	0,86	0,69	0,57	0,50
Fuentes,	Col	0,00	0,11	1,62	0,41	1,51	20,20	319,83
4 planos	Wt	2,18	1,83	1,51	1,28	1,17	1,077	0,96
	TOT	5,63	3,66	4,28	2,55	3,38	21,85	321,30
	Sam	3,45	1,72	1,15	0,86			
Astro,	Col	0,00	0,01	0,48	2,73			
2 planos	Wt	2,13	1,67	1,32	1,07			
	TOT	5,59	3,41	2,95	4,66			

Tabla 6.5: Espacio utilizado, expresado en bpc, por cada uno de los componentes de nuestro índice construido sobre las colecciones de prueba que se utilizan en esta sección, utilizando período de muestreo 64.

construimos nuestro índice con valores de $L = 1 \ldots 4$ para las colecciones RGB, y $L = 1 \ldots 7$ en escala de grises. En todos los casos consideramos $S = L$ (es decir, sin redundancia).

En cada caso, consideramos distintos valores para el período de muestreo de la imagen, entre 8 y 128. Las figuras 6.17 y 6.18 muestran los tiempos de acceso para subimágenes pequeñas sobre las distintas colecciones en función del espacio ocupado por el índice, como fracción del espacio ocupado por la representación plana de la colección. Las figuras 6.19 y 6.20 muestran lo mismo para la reconstrucción de subimágenes grandes.

Resulta importante notar que el aumento en el tamaño de las bandas no repercute de manera tan notoria en los tiempos de reconstrucción de subimágenes pequeñas, pero sí lo hace considerablemente para reconstruir subimágenes grandes. Esto se debe a que en el primer caso el costo de acceder a una o dos L-bandas para cubrir la imagen es comparable al costo de las L-bandas extra a las que se accede para completar los bordes no cubiertos, mientras que en el segundo caso los bordes son despreciables con respecto al número de bandas a leer para cubrir la imagen, número que disminuye de manera proporcional al tamaño de éstas. El hecho de que el tiempo disminuya al leer m/L filas en un Wavelet Tree de altura $\log \sigma_L$ indica que en la práctica $\log \sigma_L << \sigma^L$, lo cual es consecuencia de la homogeneidad vertical.

También hay una repercusión en el tamaño del índice. Se observa en las colecciones en escala de grises que cuando L va aumentando entre 1 y 5 el tamaño del índice disminuye. Cuando L vale 6 se percibe un ligero aumento en el tamaño. Cuando L vale 7 el tamaño del índice se vuelve no competitivo. En las colecciones en formato RGB el comportamiento es similar, alcanzando el óptimo cuando $L = 3$. Esto es atribuible al tamaño requerido por las estructuras asociadas al color, que aumentan al punto de dominar el espacio requerido por el índice, como se mostró en la tabla 6.4.

6.3.3. Búsqueda de patrones

Recordemos que éste índice no soporta la operación de conteo de ocurrencias. El experimento que se presenta en esta sección muestra el tiempo requerido para determinar las posiciones de las ocurrencias de un patrón en la colección.

Se extrajeron 1000 patrones de manera aleatoria y se realizó la búsqueda sobre distintas construcciones de nuestro índice, tomando valores de $L = 1 \ldots 4$ para las colecciones RGB, y $L = 1 \ldots 7$ en escala de grises. En cada caso, consideramos distintos valores para el período de muestreo de la imagen, entre 8 y 128. Las figuras 6.21 y 6.22 muestran los tiempos de búsqueda en función del espacio ocupado por el índice, como fracción de la colección original para patrones grandes. La figura muestra el tiempo de búsqueda para patrones pequeños sobre la colección *Arte*. No se incluye la búsqueda de patrones pequeños para otras colecciones pues los tiempos de búsqueda son demasiado altos. Esto se debe al comportamiento indeseado de la estructura basada en el Índice FM de tener un factor $L\ occ_0^{(L)}$: cuando el tamaño del patrón es suficientemente pequeño, el número de ocurrencias de su fila menos frecuente será demasiado alto, elevando demasiado el tiempo de búsqueda.

En las colecciones RGB se observa que al aumentar de una a dos líneas el tamaño de las bandas el tiempo de búsqueda aumenta, pero no alcanza a ser el doble. Al pasar a tres líneas el tiempo disminuye abruptamente. Esto es atribuible a que el tiempo de búsqueda depende del término $L\ occ_0^{(L)}$, que es L veces el número de ocurrencias de la L-banda menos frecuente del patrón.

En las colecciones en escala de grises el comportamiento es similar. En la colección *Micro* los tiempos de búsqueda aumentan hasta alcanzar el máximo en $L = 4$. Cuando $L \geq 5$ los tiempos disminuyen.

El comportamiento del tamaño del índice fue discutido en la Sección 6.3.2.

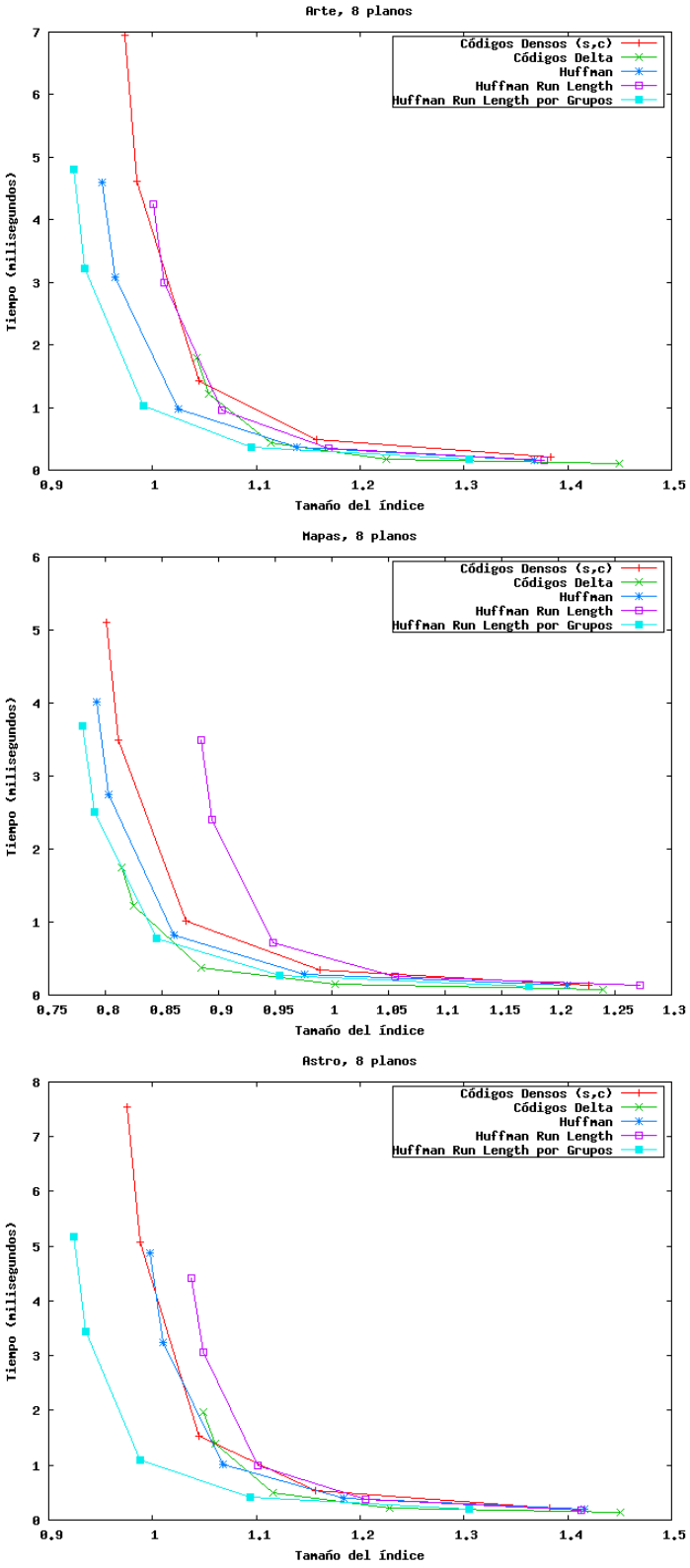


Figura 6.1: Tiempo en reconstruir una subimagen de 10×10 en las colecciones en RGB, utilizando distintas variantes del CSA-2D.

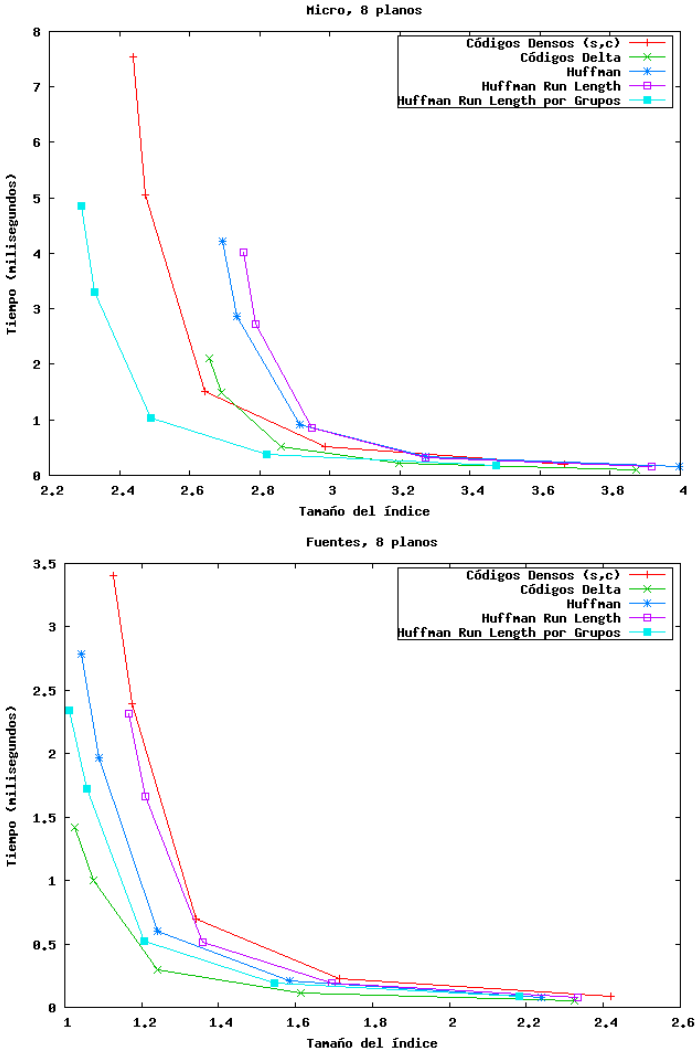


Figura 6.2: Tiempo en reconstruir una subimagen de 10×10 en las colecciones en escala de grises, utilizando distintas variantes del CSA-2D.

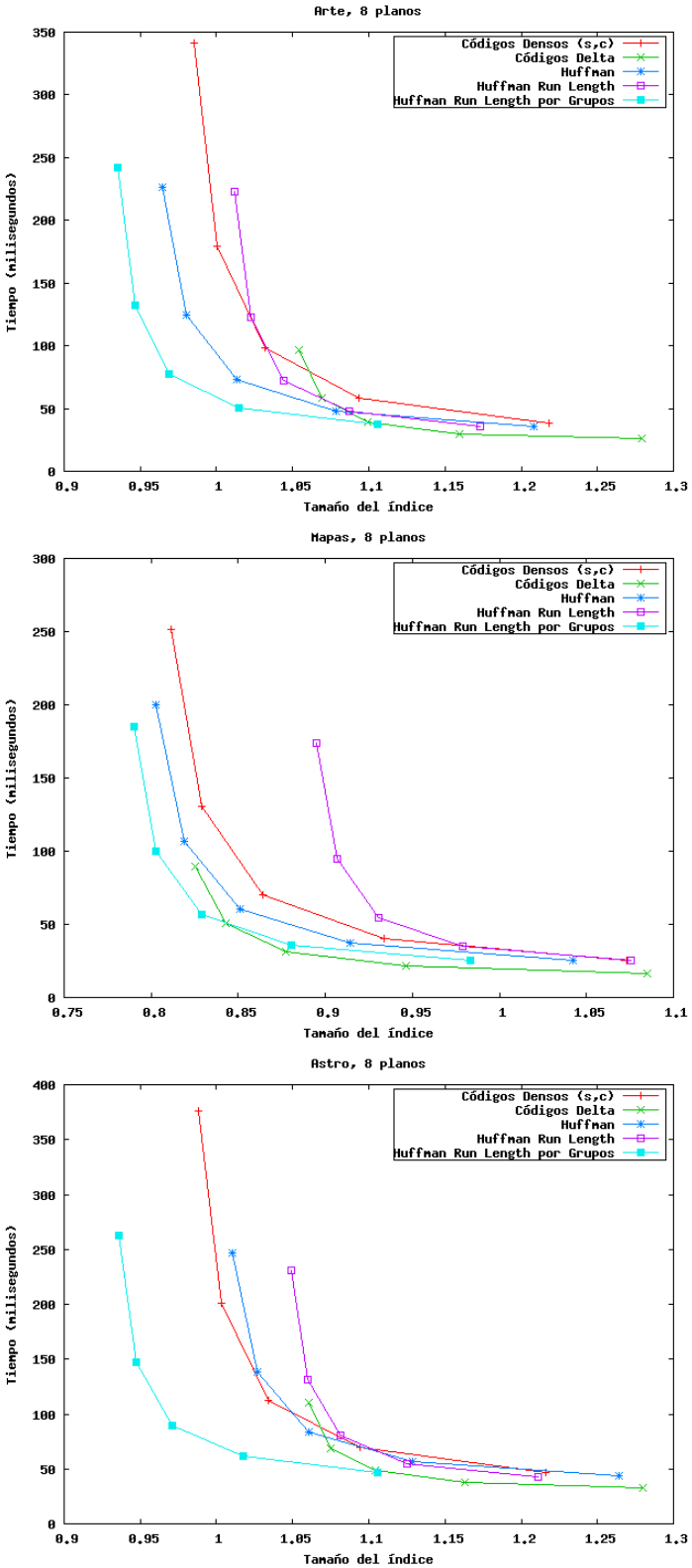


Figura 6.3: Tiempo en reconstruir una subimagen de 200×200 en las colecciones en RGB, utilizando distintas variantes del CSA-2D.

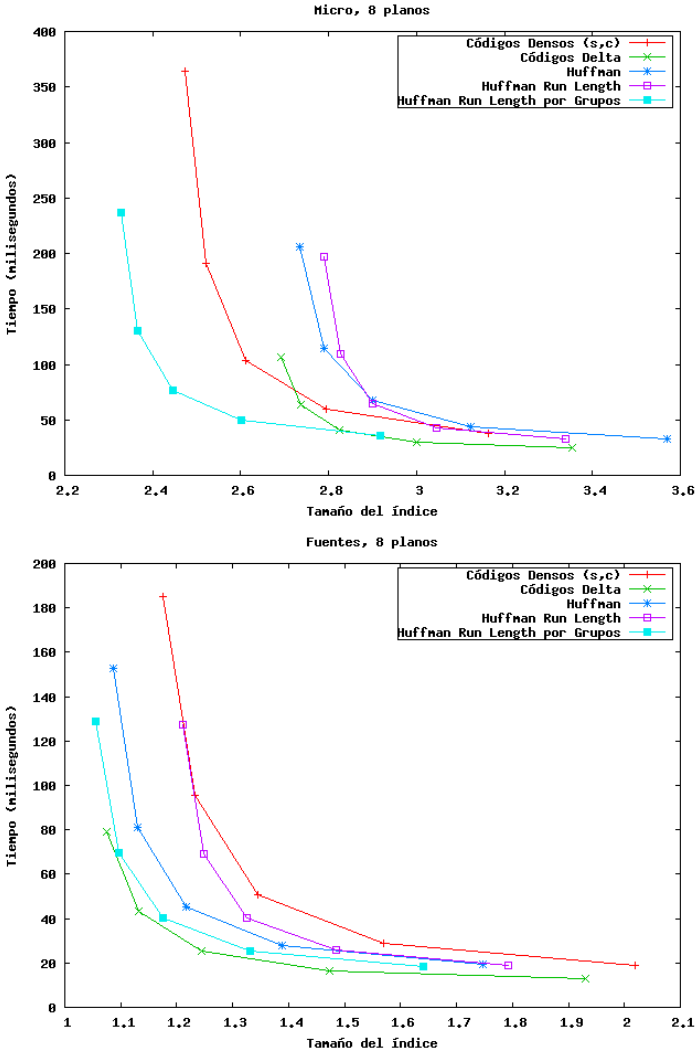


Figura 6.4: Tiempo en reconstruir una subimagen de 200×200 en las colecciones en escala de grises, utilizando distintas variantes del CSA-2D.

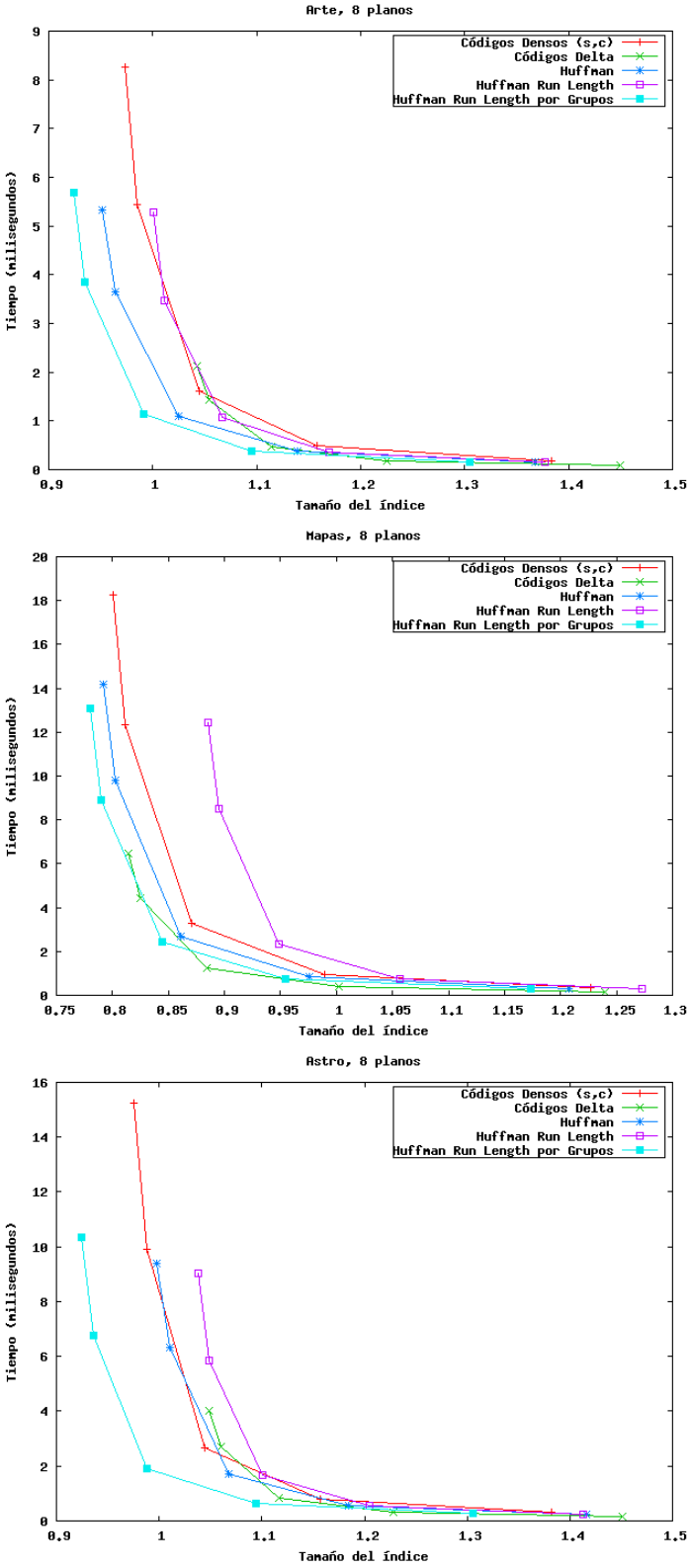


Figura 6.5: Tiempo en realizar el conteo de ocurrencias de un patrón de 10×10 en función del tamaño del índice para distintas codificaciones del CSA-2D, sobre las colecciones en RGB.

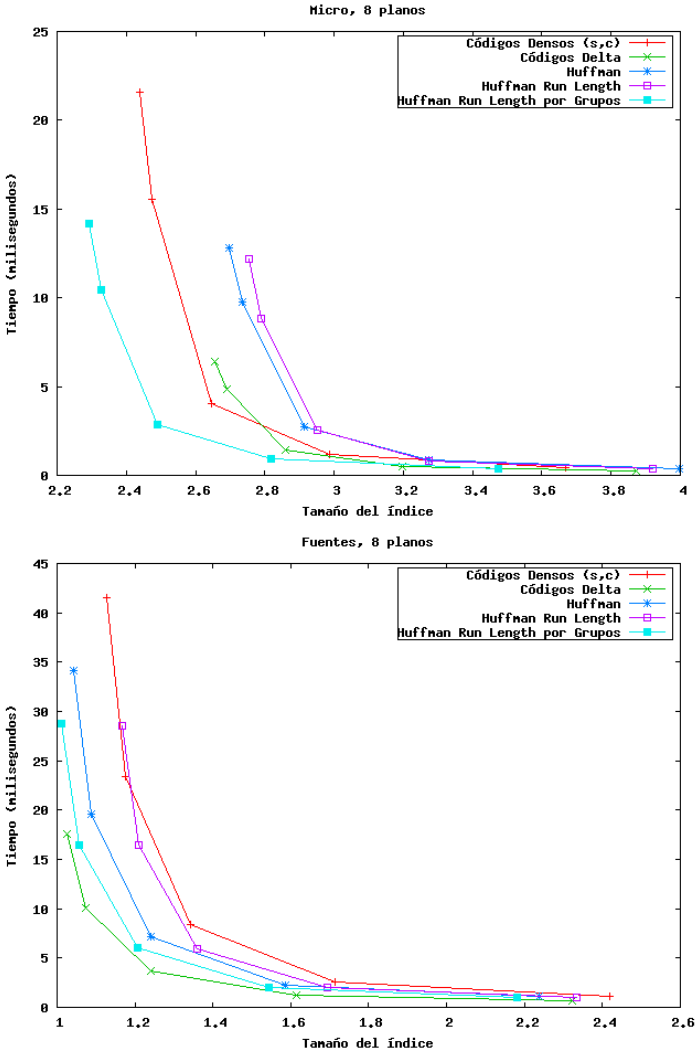


Figura 6.6: Tiempo en realizar el conteo de ocurrencias de un patrón de 10×10 en función del tamaño del índice para distintas codificaciones del CSA-2D, sobre las colecciones en escala de grises.

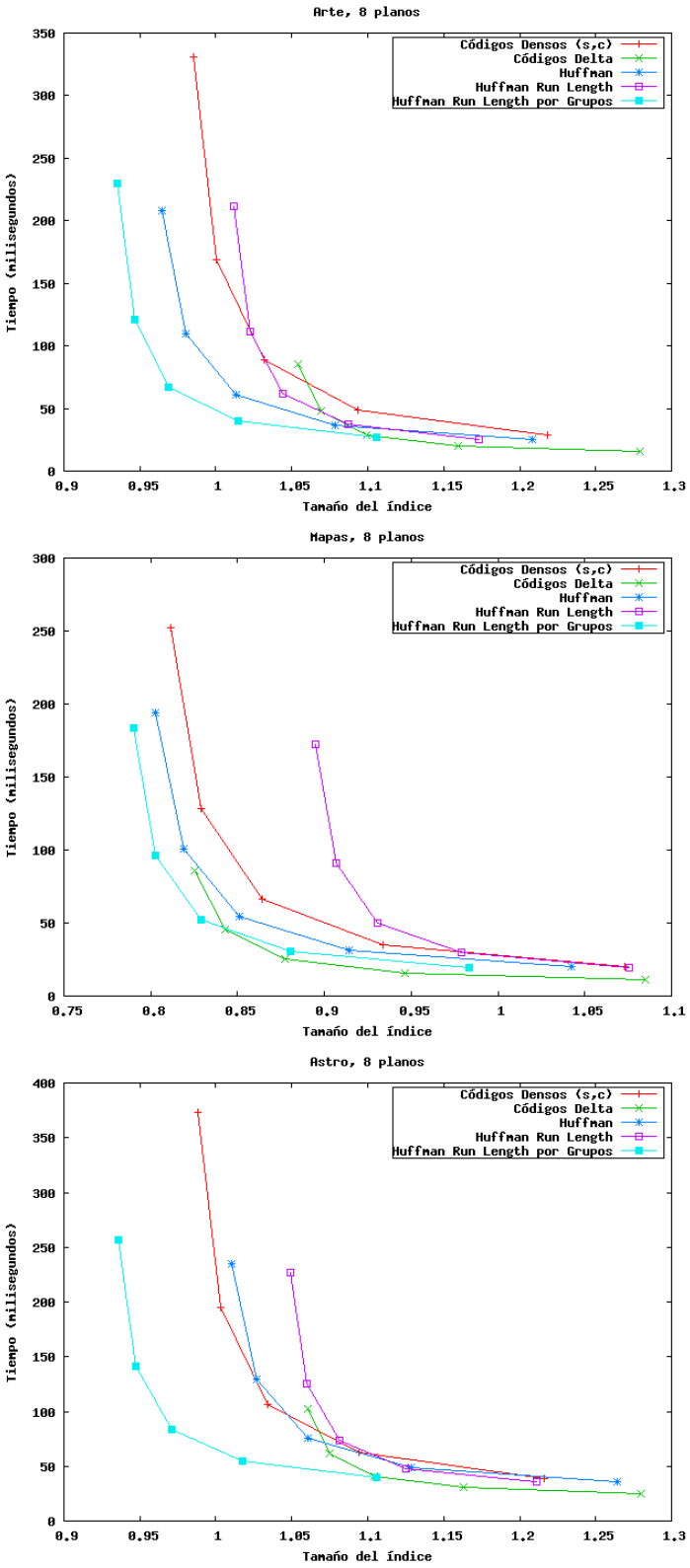


Figura 6.7: Tiempo en realizar el conteo de ocurrencias de un patrón de 200×200 en función del tamaño del índice para distintas codificaciones del CSA-2D, sobre las colecciones en RGB.

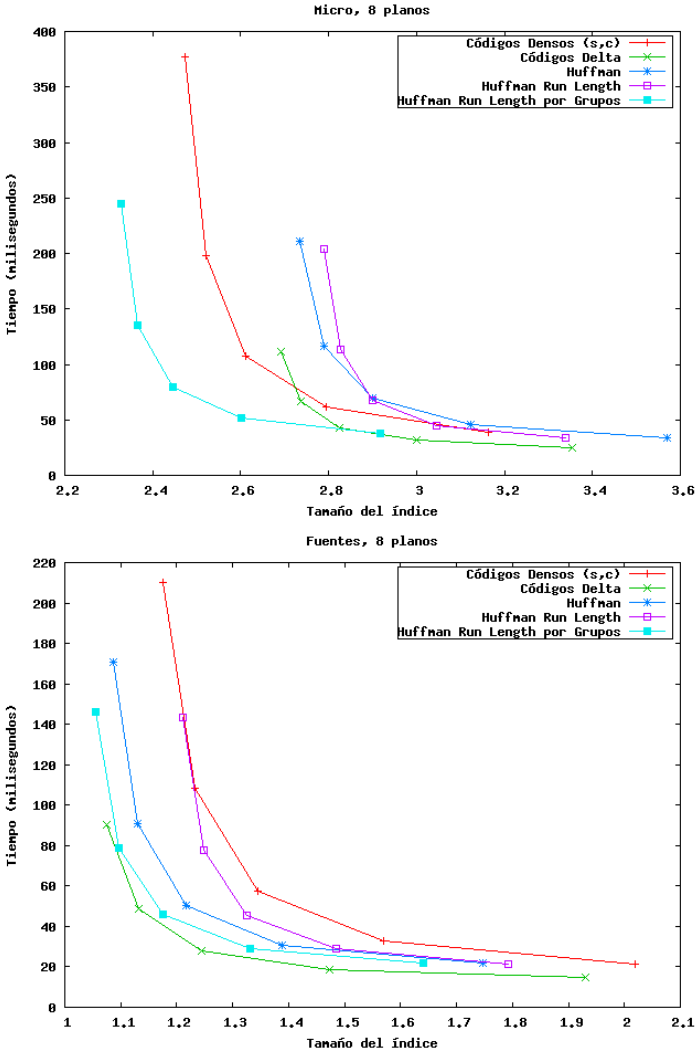


Figura 6.8: Tiempo en realizar el conteo de ocurrencias de un patrón de 200×200 en función del tamaño del índice para distintas codificaciones del CSA-2D, sobre las colecciones en escala de grises.

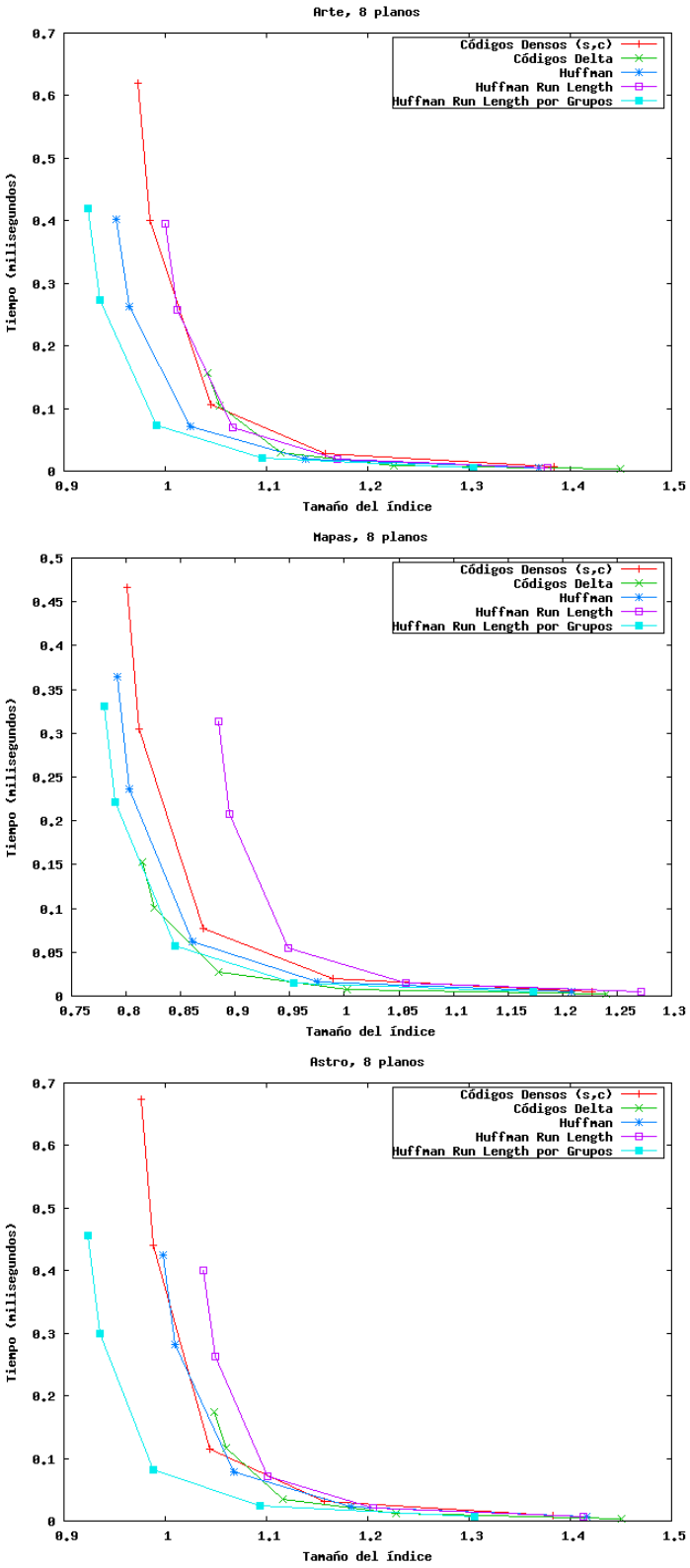


Figura 6.9: Tiempo de localización de una ocurrencia en función del tamaño del índice, para las distintas codificaciones del CSA-2D, sobre colecciones en RGB.

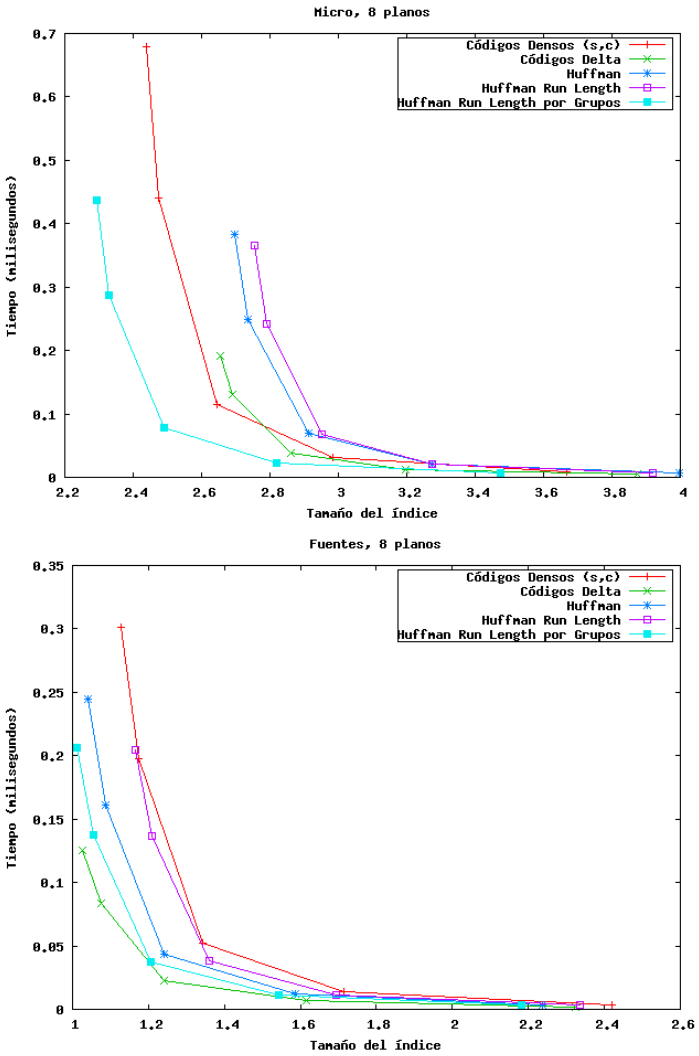


Figura 6.10: Tiempo de localización de una ocurrencia en función del tamaño del índice, para las distintas codificaciones del CSA-2D, sobre colecciones en escala de grises.

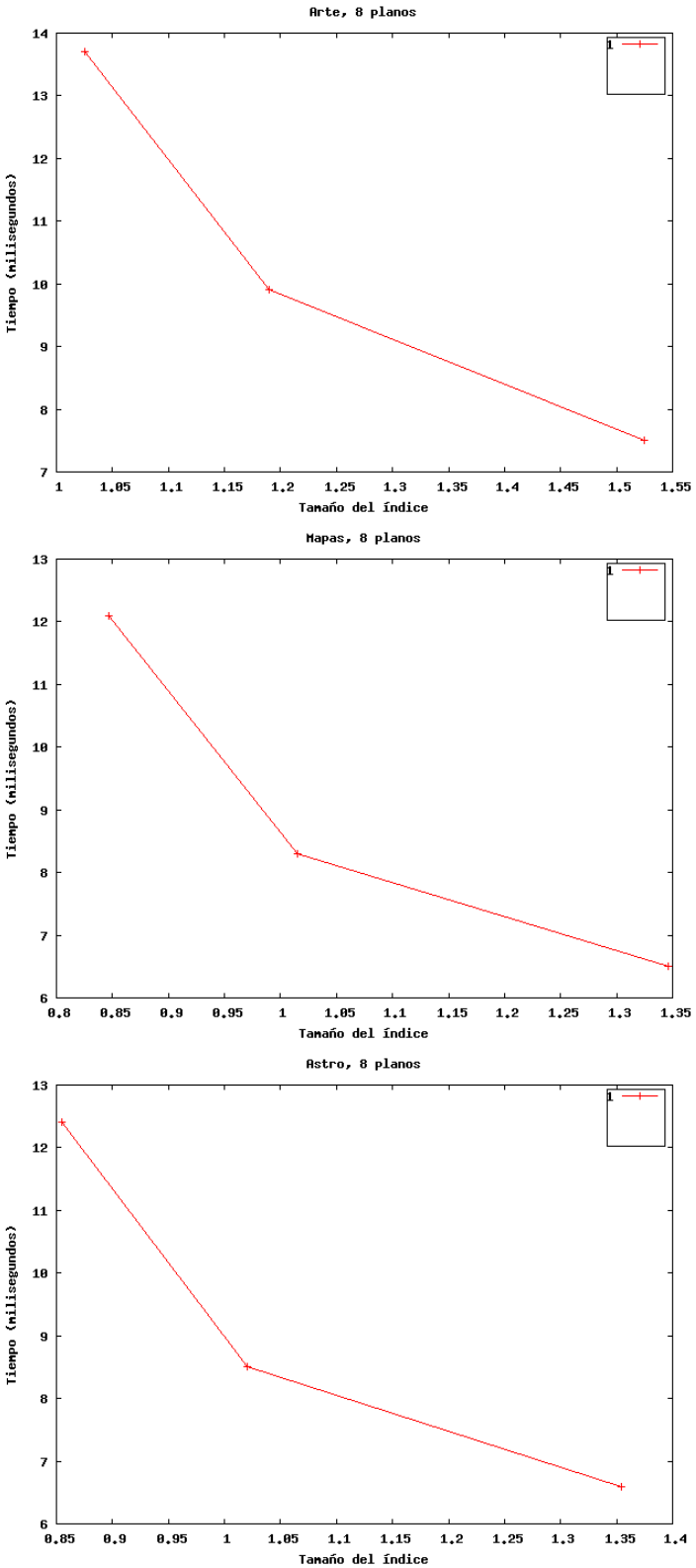


Figura 6.11: Tiempo en reconstruir una imagen de 10×10 en función del tamaño del índice, utilizando la estructura indexación por filas sobre colecciones en RGB.

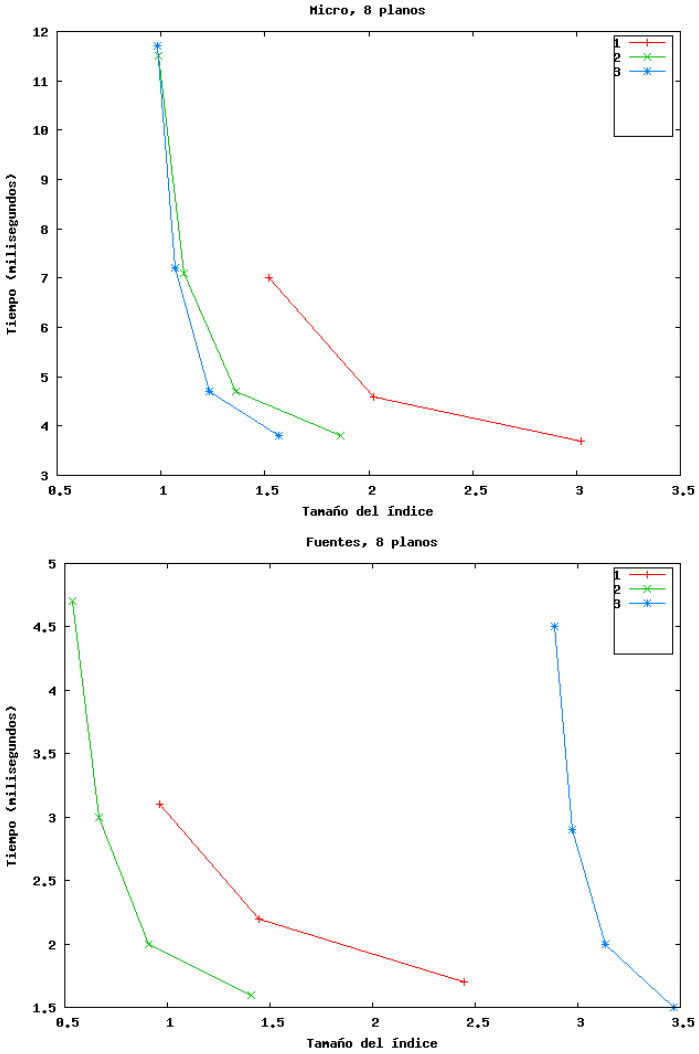


Figura 6.12: Tiempo en reconstruir una imagen de 10×10 en función del tamaño del índice, utilizando la estructura indexación por filas sobre colecciones en escala de grises.

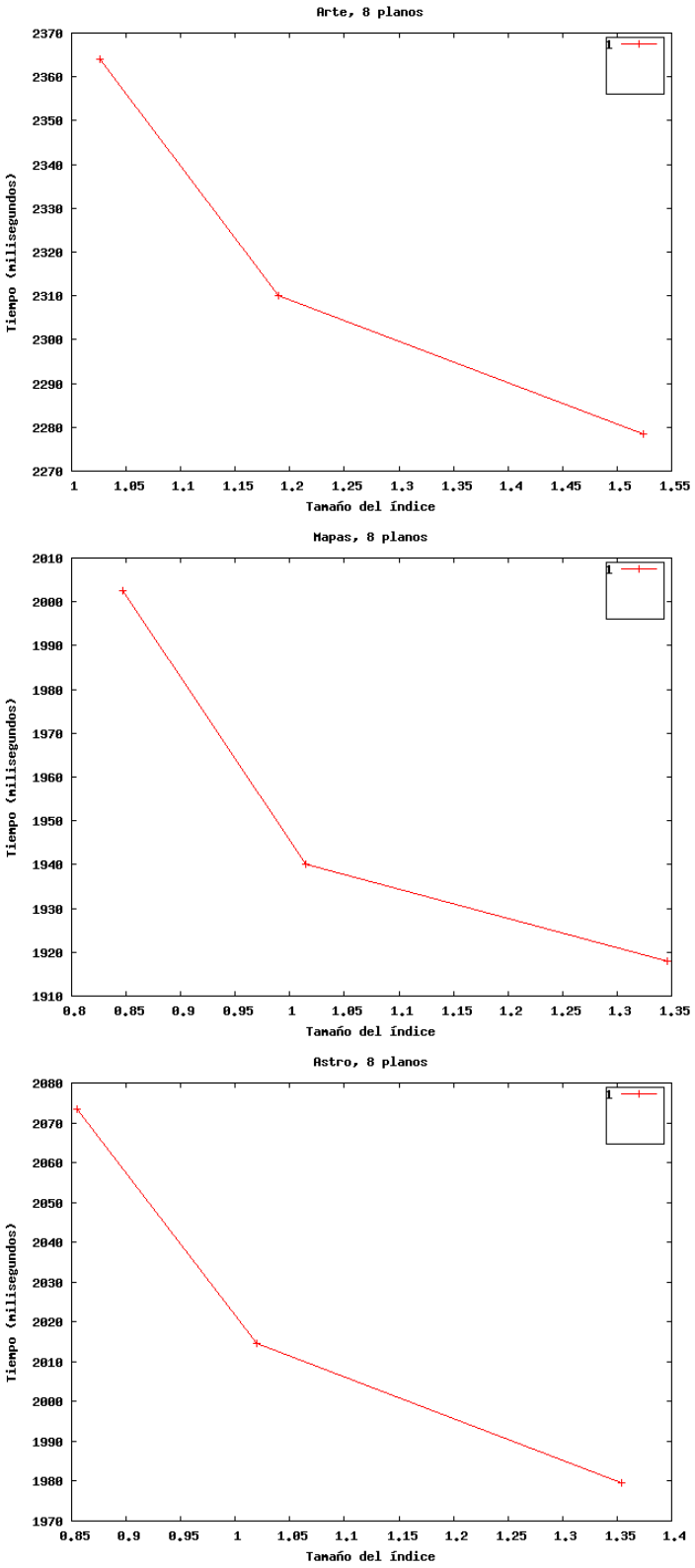


Figura 6.13: Tiempo en reconstruir una imagen de 200×200 en función del tamaño del índice, utilizando la estructura indexación por filas sobre colecciones en RGB.

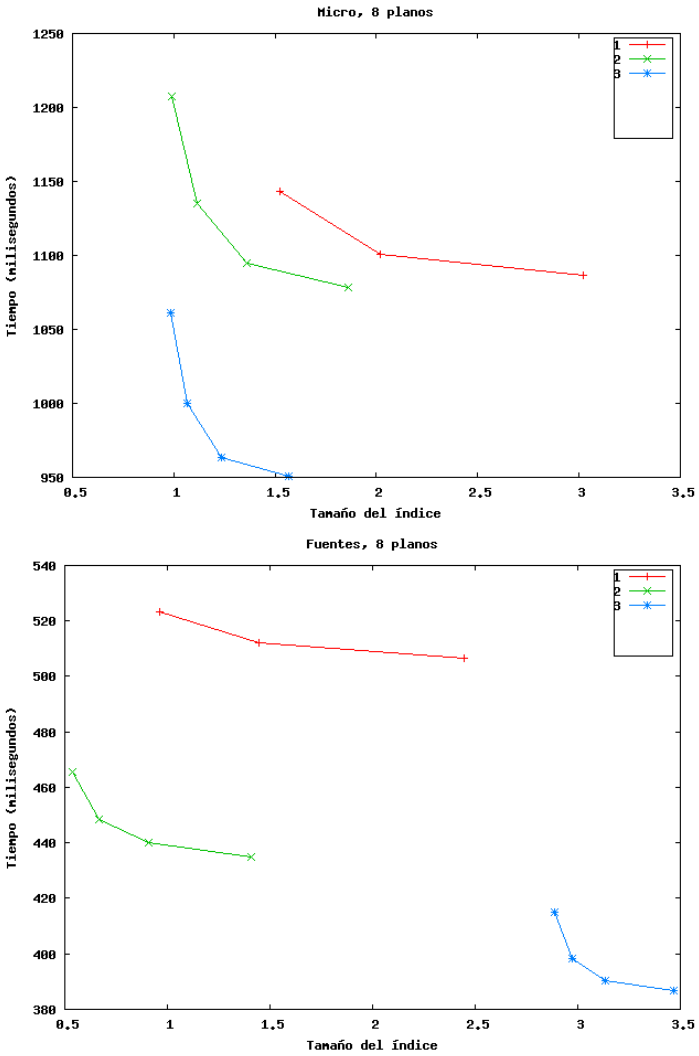


Figura 6.14: Tiempo en reconstruir una imagen de 200×200 en función del tamaño del índice, utilizando la estructura indexación por filas sobre colecciones en escala de grises.

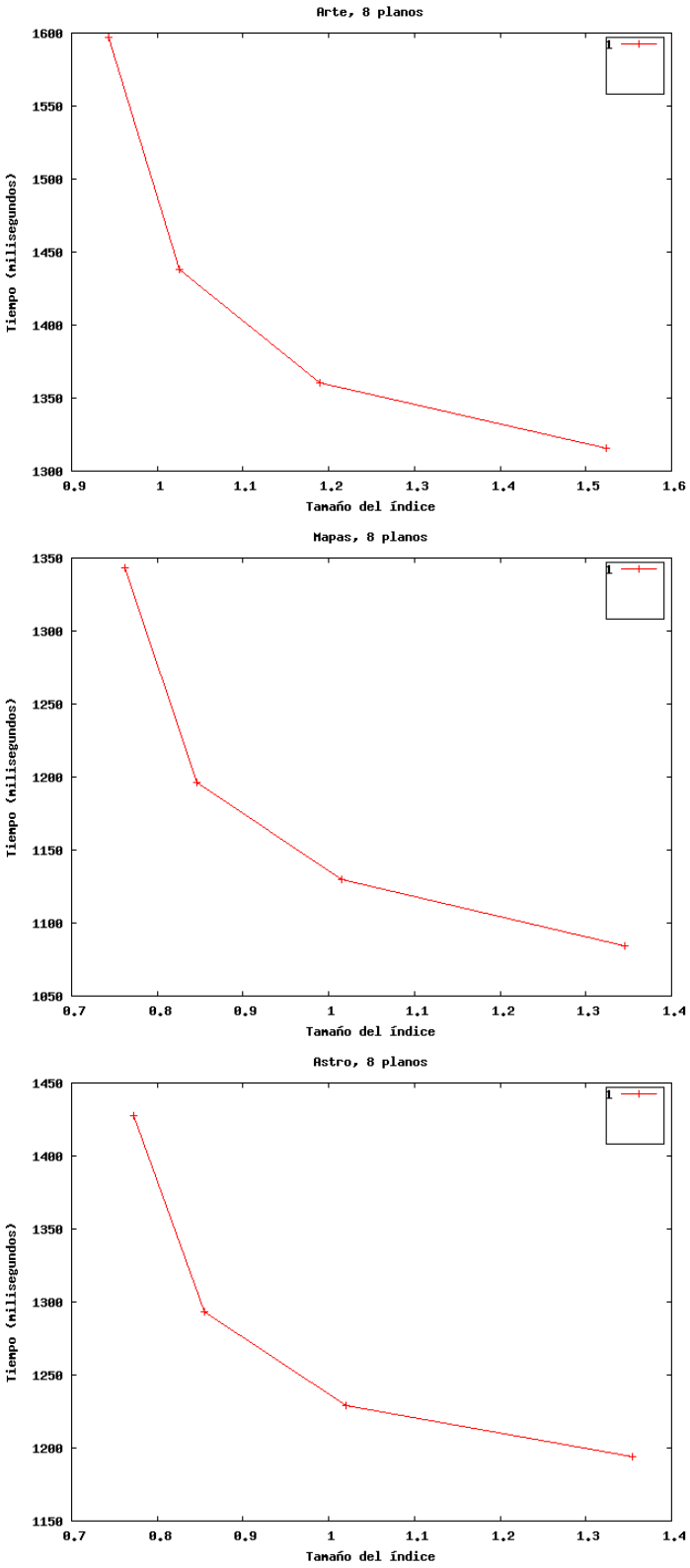


Figura 6.15: Tiempo de búsqueda de un patrón de 200 × 200 en función del tamaño del índice, utilizando la estructura indexación por filas sobre colecciones en RGB.

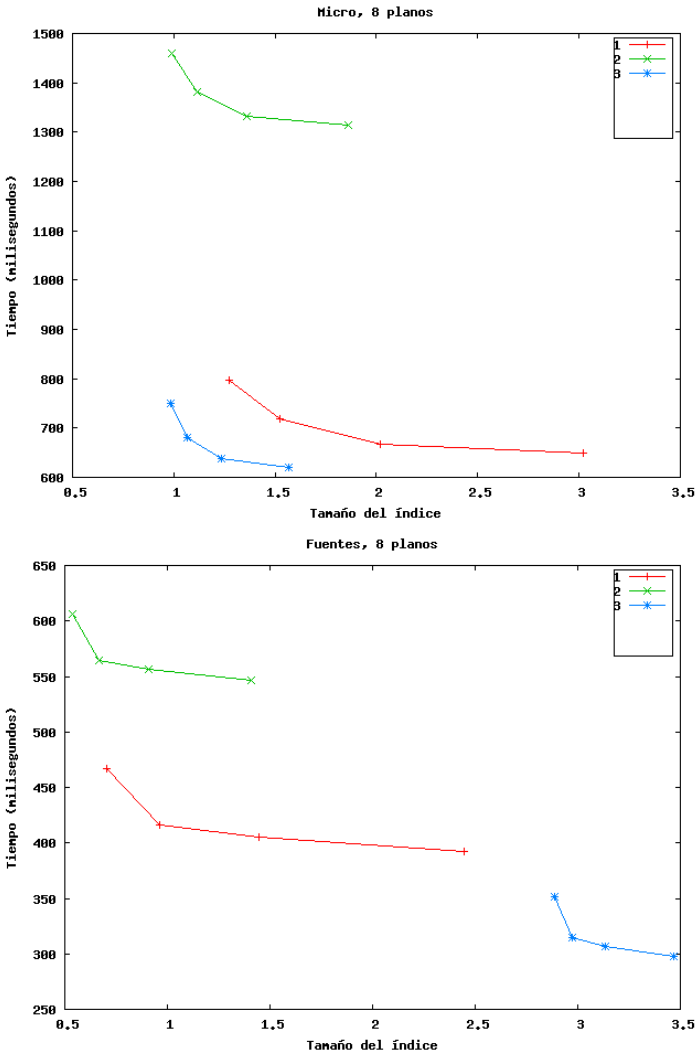


Figura 6.16: Tiempo de búsqueda de un patrón de 200×200 en función del tamaño del índice, utilizando la estructura indexación por filas sobre colecciones en escala de grises.

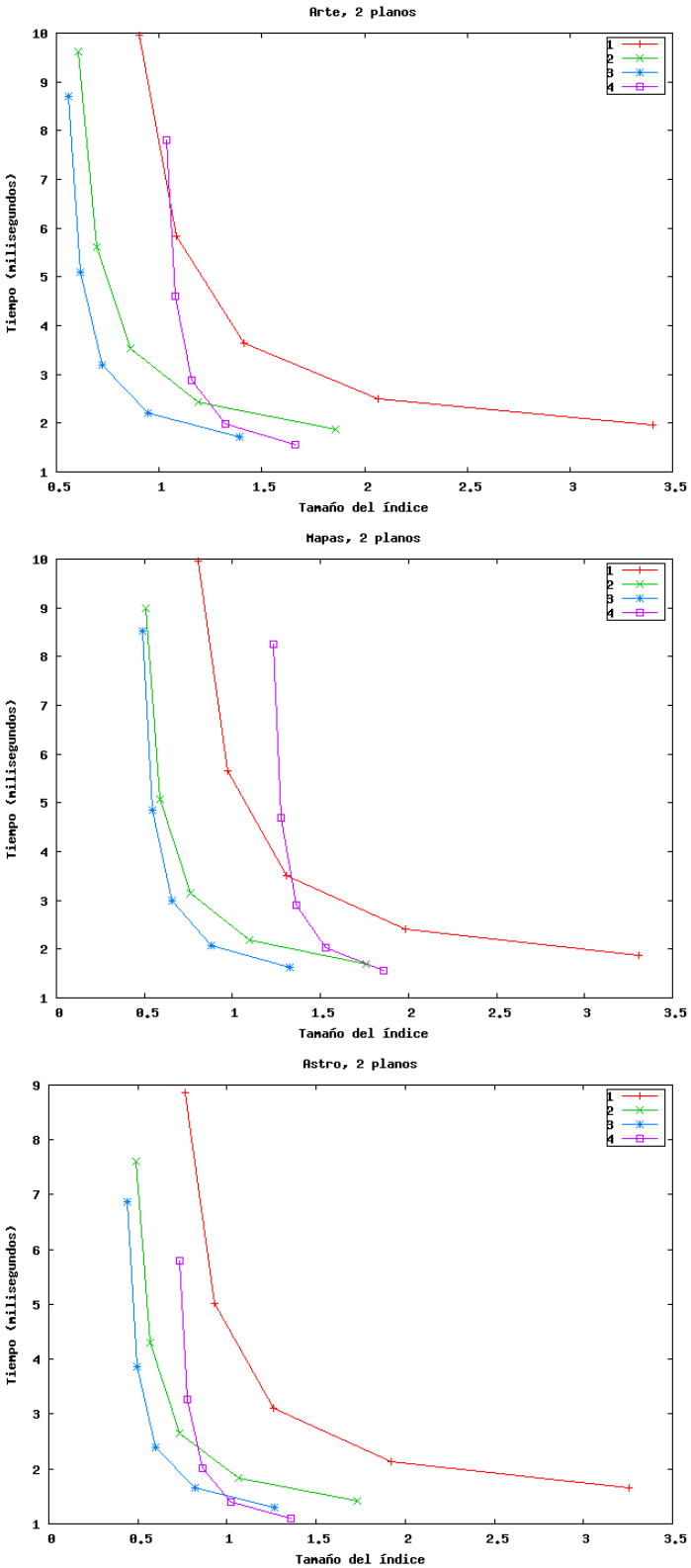


Figura 6.17: Tiempo en reconstruir una imagen de 10×10 en función del tamaño del índice, utilizando la estructura indexación por bandas, para RGB.

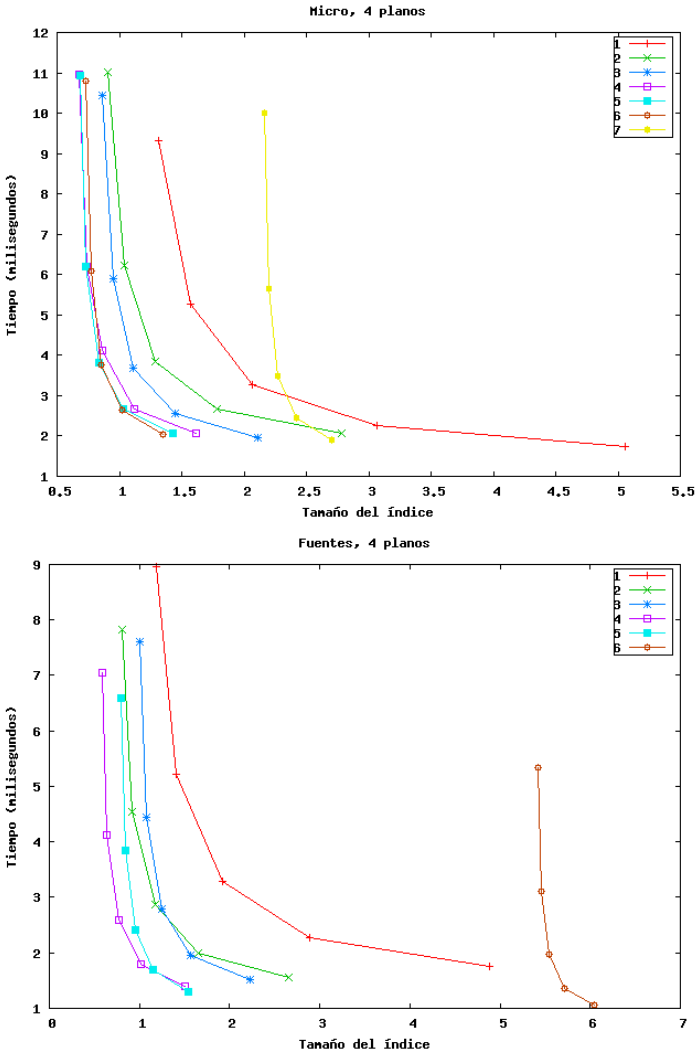


Figura 6.18: Tiempo en reconstruir una imagen de 10×10 en función del tamaño del índice, utilizando la estructura indexación por bandas, para grises. Para la colección *Fuentes* omitimos $S = 7$ pues requiere demasiado espacio (más de 80 veces el tamaño de la colección).

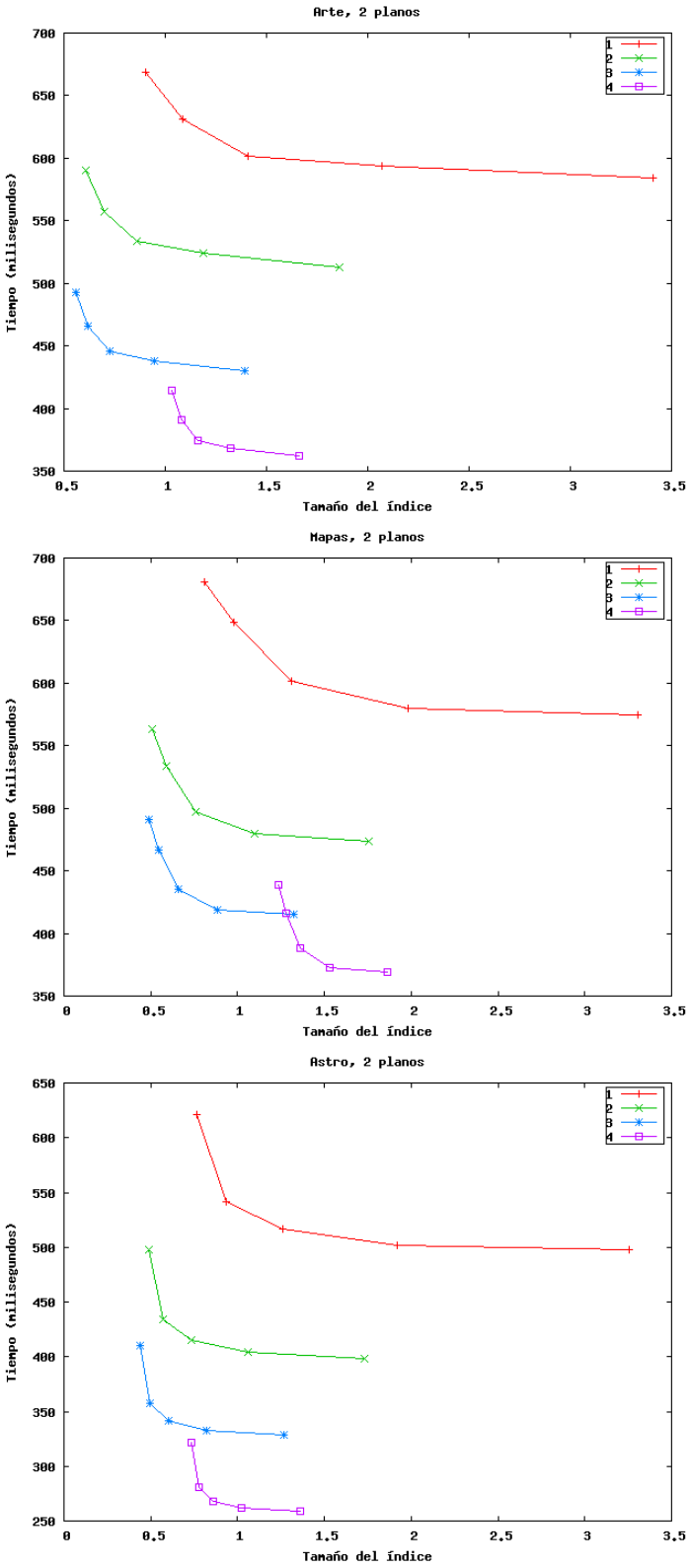


Figura 6.19: Tiempo en reconstruir una imagen de 200×200 en función del tamaño del índice, utilizando la estructura indexación por bandas, para RGB.

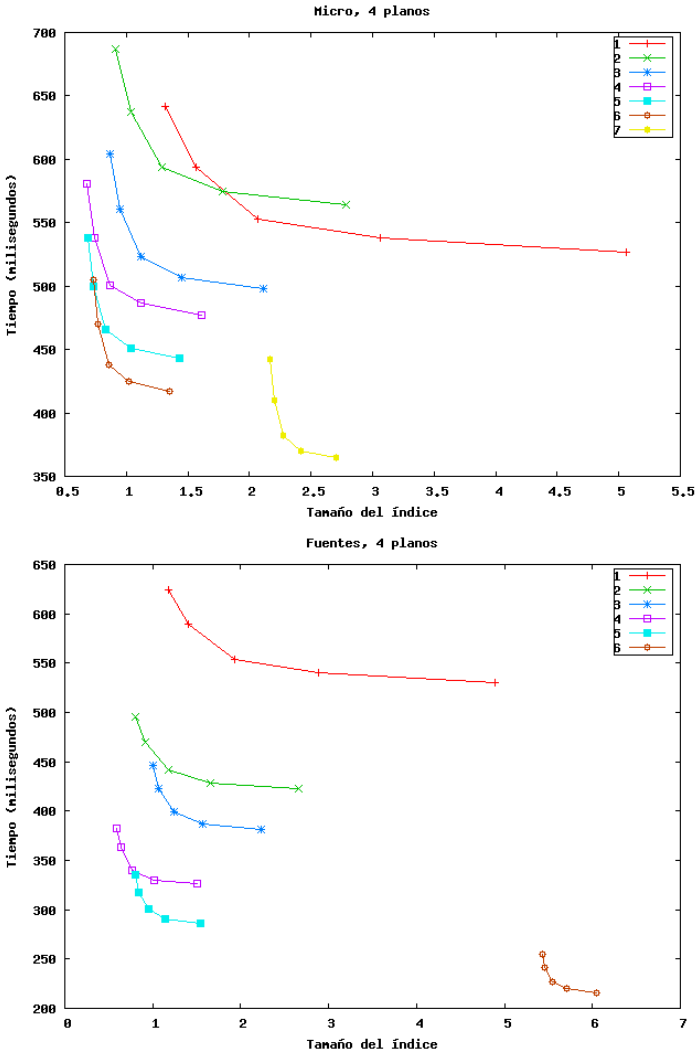


Figura 6.20: Tiempo en reconstruir una imagen de 200×200 en función del tamaño del índice, utilizando la estructura indexación por bandas, para grises. Para la colección *Fuentes* omitimos $S = 7$ pues requiere demasiado espacio (más de 80 veces el tamaño de la colección).

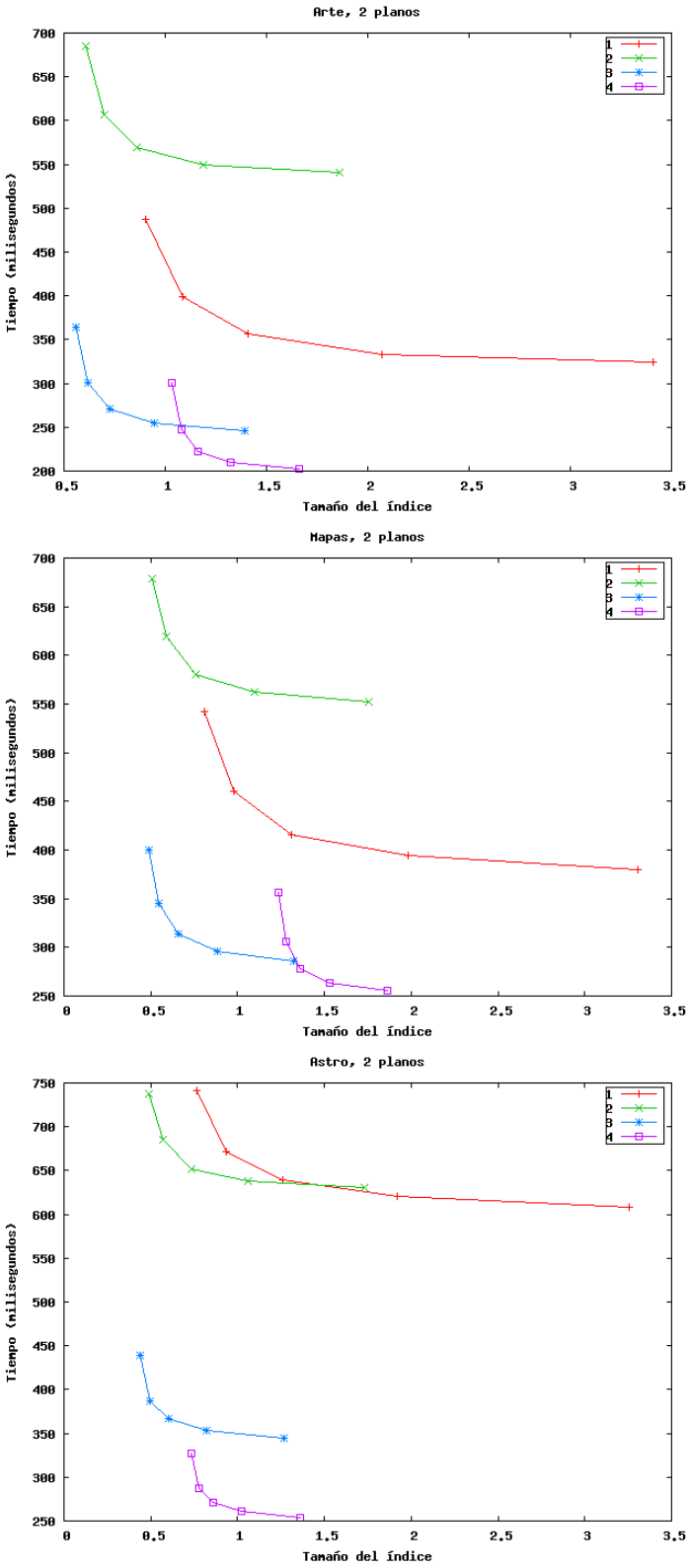


Figura 6.21: Tiempo de búsqueda de un patrón de 200 × 200 en función del tamaño del índice, utilizando la estructura indexación por bandas sobre colecciones en RGB.

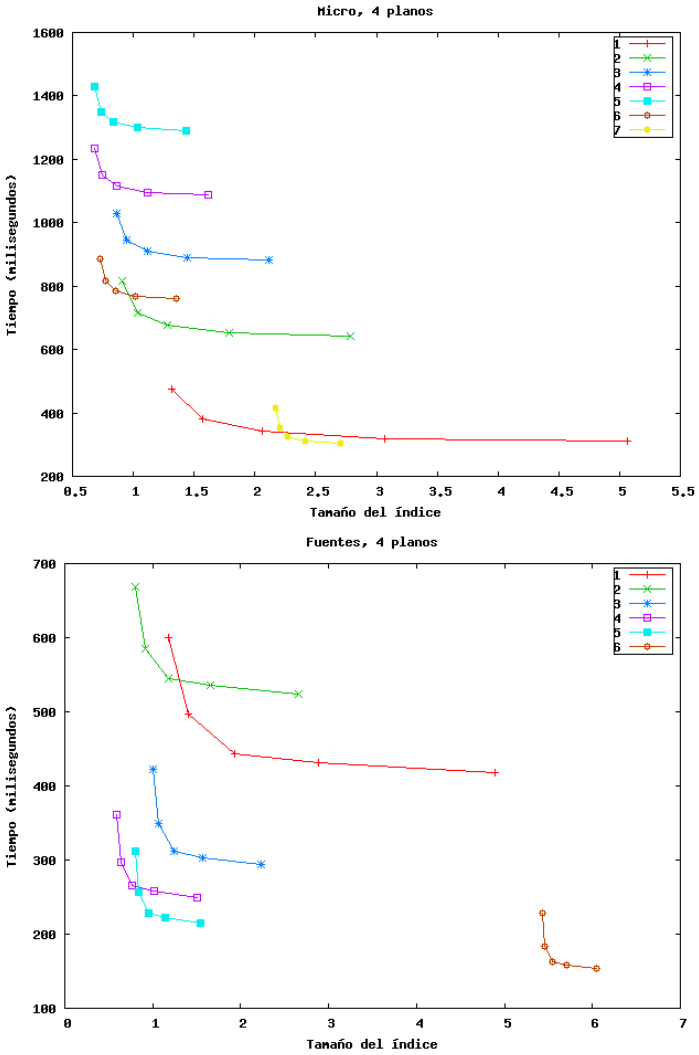


Figura 6.22: Tiempo de búsqueda de un patrón de 200×200, en grises.

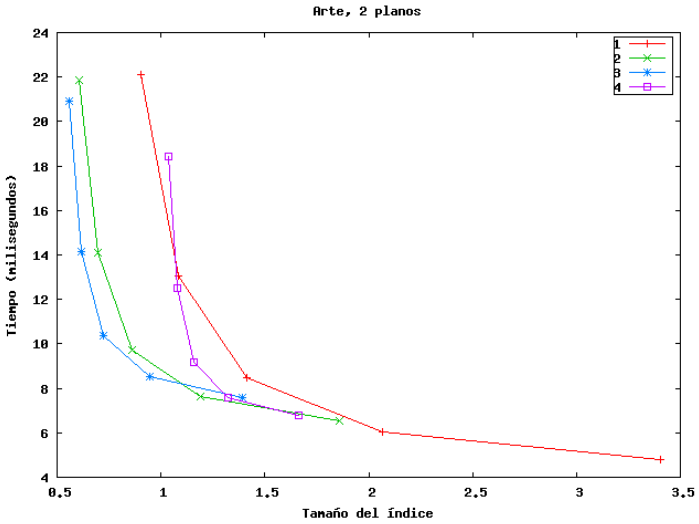


Figura 6.23: Tiempo de búsqueda de un patrón de 20×20 sobre la colección *Arte* (RGB) utilizando la estructura de indexación por bandas.

Capítulo 7

Conclusiones y Trabajo Futuro

En este trabajo se presentó la implementación de dos autoíndices para texto bidimensional y se evaluó su desempeño en términos de espacio requerido por el índice así como de tiempo en responder las distintas consultas que soportan.

Dentro de las distintas variantes del CSA-2D, la que presentó un mejor desempeño fue la codificación Run Length por Grupos, requiriendo en total entre un 79 % y 93 % del tamaño de la colección original para colecciones *RGB* (*Arte*,*Mapas*,*Astro*). Para colecciones en escala de grises (*Fuentes*, *Micro*) no se logró comprimir, pues nuestra estructura requiere entre un 6 % y un 130 % de espacio adicional a lo que utilizaría la colección plana (recordemos que los autoíndices reemplazan a la colección). Al trabajar en colecciones con rango de color más pequeño (lo que se puede lograr mediante la remoción de los planos menos relevantes) el CSA-2D presenta un empeoramiento en sus tasas de compresión. Con esta estructura podemos reconstruir cualquier subimagen a una velocidad de 1 megapixel por segundo. Podemos contar las ocurrencias de un patrón a una velocidad aproximada de $0,5$ megapixeles por segundo. Finalmente, somos capaces de localizar cada ocurrencia en menos de $0,1$ milisegundos.

La segunda estructura está basada en el Índice FM, y consiste básicamente en utilizar el índice para secuencias unidimensionales e indexar las filas de la imagen. Esta técnica resultó ser más efectiva en cuanto a niveles de compresión, lo cual es consistente con los resultados teóricos y experimentales que se conocen del Índice FM. Su mejor desempeño se presenta cuando el rango de colores es pequeño.

Utilizando la estructura sobre las colecciones con todos sus planos alcanzamos requerimientos de espacio cercanos a un 85 % del espacio original para las colecciones *Mapas* y *Astro*, y casi un 50 % para la colección *Fuentes*. En el resto de las colecciones este índice ocupó un

poco (10 % aproximadamente) de espacio adicional al que ocuparía la colección plana. Sin embargo, los tiempos de acceso y búsqueda resultaron entre 10 y 20 veces más lento que el CSA-2D.

Sin embargo, utilizando la estructura de las bandas sobre los planos más relevantes de nuestras colecciones de estudio, esta estructura requirió entre un 49 % y un 62 % del tamaño de la colección original para colecciones RGB. Esto equivale al desempeño de esta estructura sobre colecciones con un rango de color más pequeño. En los casos en que el formato de la colección es escala de grises, requirió entre un 84 % y un 95 % del tamaño de la colección original.

Si comparamos estos resultados con los niveles de compresión que un compresor JPG sin pérdida puede alcanzar (del orden del 50 % en promedio) nuestros resultados son esperanzadores. La gran ventaja de la compresión JPG es que puede perder un bajo porcentaje de calidad y obtener niveles de compresión mucho más altos (10 %).

7.1. Trabajo futuro

Los resultados obtenidos son promisorios y dejan abiertos sendos desafíos. En primer lugar, la búsqueda de una representación de Ψ que nos dé garantías de espacio es un problema que aún permanece abierto. Para sufijos de secuencias unidimensionales es un problema ampliamente estudiado, en el cual se puede garantizar espacio proporcional a la entropía del texto en términos teóricos, y los resultados experimentales han conseguido este desempeño, mientras que en el caso de nuestros sufijos bidimensionales tenemos una representación sin garantías teóricas y que obtiene un buen desempeño en algunos escenarios.

Por otro lado, tampoco se han explorado estructuras de búsqueda reversa basadas en sufijos bidimensionales. Nuestra estructura utiliza búsqueda reversa sobre secuencias unidimensionales como filtro, pero no tenemos garantías de tiempo. Así, aún no es claro si es posible desarrollar autoíndices basados en búsqueda reversa que nos den garantías sobre el tiempo de búsqueda.

Otro punto importante es dar soporte para búsquedas más realistas a partir de nuestras estructuras. Si bien éstas proveen un buen soporte para realizar búsqueda exacta de patrones de manera eficiente, en escenarios de uso reales de búsqueda por contenido resulta de gran interés dar soporte a búsquedas más complejas, como búsqueda aproximada, permitiendo

rotaciones, tolerancia a la iluminación, escalamiento del patrón, etc. En el área de búsqueda *on-line* existen diversas estrategias para dar soporte a éstas búsquedas dividiéndolas en subproblemas que se resuelven utilizando búsqueda exacta.

Finalmente, existen otros tipos de índices que se han construido sobre secuencias bidimensionales, como aquellos que definen una secuencia en espiral a partir de un píxel arbitrario. La posibilidad de encontrar un autoíndice basado en estos índices,y su potencial interés, son algunas preguntas que quedan por responder.

Referencias

[1] N. Brisaboa, A. Fariña, G. Navarro, and J. Paramá. Lightweight natural language text compression. *Information Retrieval*, 10:1–33, 2007.

[2] N. Brisaboa, A. Fariña, G. Navarro, A. Places, and E. Rodríguez. Self-indexing natural language. In *Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5280, pages 121–132. Springer, 2008.

[3] M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, 1994.

[4] D. Clark. *Compact pat trees*. PhD thesis, Waterloo, Ont., Canada, 1998.

[5] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5280, pages 176–187, 2008.

[6] P. Elias. Universal codeword sets and the representation of the integers. *IEEE Transactions on Information Theory*, 21:194–203, 1975.

[7] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. An alphabet-friendly FM-index. In *Proc. 11th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 3246, pages 150–160, 2004.

[8] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proc. 41st IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 390–398, 2000.

[9] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.

[10] R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38, Greece, 2005. CTI Press and Ellinika Grammata.

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

[12] R. Grossi, A. Gupta, and J. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. In *Proc. 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 636–645, 2004.

[13] R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. 32nd Symposium on Theory of Computing (STOC)*, pages 397–406, 2000.

[14] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[15] D. Kim, Y. Kim, and K. Park. Generalizations of suffix arrays to multi-dimensional matrices. *Theoretical Computer Science*, 302(1-3):223–238, 2003.

[16] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.

[17] V. Mäkinen and G. Navarro. Implicit compression boosting with applications to self-indexing. In *Proc. 14th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 4726, pages 214–226, 2007.

[18] V. Mäkinen and G. Navarro. On self-indexing images — image compression with added value. In *Proc. 18th IEEE Data Compression Conference (DCC)*, pages 422–431, 2008.

[19] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[20] P. Miltersen. Lower bounds on the size of selection and rank indexes. In *Proc. 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 11–12, 2005.

[21] J. Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 37–42, 1996.

[22] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. 9th Algorithm Engineering and Experimentation (ALENEX)*, 2007.

[23] R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 233–242, 2002.

[24] K. Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 225–232, 2002.

[25] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.