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

LISTADO EFICIENTE Y EN ESPACIO REDUCIDO DE DOCUMENTOS CON SUS FRECUENCIAS

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

EDUARDO IGNACIO ESCOBAR SILVA

PROFESOR GUÍA:
GONZALO NAVARRO BADINO

MIEMBROS DE LA COMISIÓN:
JEREMY BARBAY
EDGARD PINEDA LEONE

SANTIAGO DE CHILE
2014

Resumen

En este trabajo se propone un nuevo método para la recuperación de documentos eficiente en espacio reducido. En términos generales, en recuperación de documentos se busca responder eficientemente a consultas sobre una colección de documentos con aquellos documentos cuyo contenido satisface algún criterio especificado en las consultas. Para acelerar las consultas los documentos son indexados con alguna estructura de datos. Las soluciones tradicionales para estos problemas basadas en índices invertidos no son adecuadas para dominios en los cuales los patrones de consulta son arbitrarios. Por ello, para colecciones cuyo contenido son, por ejemplo, secuencias de ADN, secuencias de proteínas, datos multimedia o algunos lenguajes naturales estas soluciones no son aplicables. Los índices de texto completo ofrecen una alternativa. Estos permiten indexar patrones generales pero incurren en un excesivo costo en espacio. Muthukrishnan diseñó una solución que utiliza este tipo de índices junto con otras estructuras para resolver listado de documentos. Su algoritmo es óptimo en tiempo pero consume más de veinte veces el espacio que ocupa la colección de documentos de entrada. Sadakane desarrolló una variante del algoritmo de Muthukrishnan. Para reducir el espacio introduce algunas modificaciones y diseña estructuras compactas que reemplazan las utilizadas por Muthukrishnan. Además extiende el algoritmo para resolver consultas de listado de documentos jerarquizadas. El espacio ocupado por el algoritmo de Sadakane para consultas jerarquizadas resulta excesivo para muchas aplicaciones prácticas. Aquí se proponen nuevas estructuras compactas para abordar este problema. Los resultados experimentales muestran que la nueva estrategia resuelve el problema de listado de documentos con sus frecuencias en un espacio menor y con la misma eficiencia que la solución original de Sadakane.

Tabla de Contenido

Introducción **1**

1. Conceptos básicos **3**
1.1. Preliminares . . . 3
1.2. Representaciones sucintas de árboles . . . 4
1.2.1. Codificación implícita (Heap) . . . 5
1.2.2. Codificaciones sucintas . . . 5
1.3. Diccionarios rank/select . . . 7
1.3.1. Bitmaps RRR . . . 7
1.3.2. Árboles Wavelet . . . 8
1.3.3. rank, select y access . . . 9
1.4. Auto-índice . . . 10
1.4.1. Funciones $SA(\cdot), SA^{-1}(\cdot)$ y $LF(\cdot)$ . . . 11

2. Trabajo Relacionado **13**
2.1. Listado de Documentos . . . 13
2.2. Listado de documentos con sus frecuencias . . . 14

3. Nueva estrategia para reducir el espacio **17**
3.1. Diagnóstico . . . 17
3.2. Extensión del árbol wavelet . . . 19
3.3. Solución propuesta . . . 20

4. Experimentación **21**
4.1. Textos de Prueba . . . 21
4.2. Implementación . . . 22
4.2.1. Estrategias . . . 22
4.2.2. Implementación . . . 23
4.3. Resultados Experimentales . . . 24

Conclusión **31**

Bibliografía **32**

A. Anexo **34**
A.1. SAIS (Suffix Array Induced Sorting Algorithm) . . . 34

Introducción

En términos generales, en recuperación de documentos se busca responder eficientemente a consultas sobre una colección de documentos con aquellos documentos cuyo contenido satisface algún criterio especificado en las consultas. Tradicionalmente las soluciones para estos problemas se han basado en índices invertidos, limitando el rango de consultas a patrones predeterminados. Navarro et al.[13] formulan algunos de los problemas fundamentales del área, de la siguiente manera. Dada una colección $\mathcal{D}$ de documentos tales que $\sum_{d \in \mathcal{D}} |d| = n$ y un patrón p se definen los problemas:

(1) *Listado de Documentos*: Encontrar todos los *ndocs* documentos distintos en $\mathcal{D}$ que contienen p como una subcadena.

(2) *Cálculo de frecuencia*: Es resolver (1) y además calcular el número de ocurrencias de p en cada documento obtenido en (1).

(3) *Recuperación Top-k*: Encontrar los k documentos donde p aparece con mayor frecuencia.

Muthukrishnan [12] propuso una solución para el problema de listado de documentos con patrones arbitrarios. Él diseña una estructura que le permite encontrar los documentos rápidamente a expensas de un excesivo consumo de espacio. La solución ocupa en total $\mathcal{O}(n \log n)$ bits. Sadakane [16] propuso una solución alternativa donde reduce la información redundante de Muthukrishnan. También la extiende para resolver el problema de cálculo de frecuencias. En la práctica ninguna de estas alternativas es aplicable sobre colecciones de grandes volúmenes debido al excesivo espacio que demandan. En este trabajo se analiza el efecto que tiene el largo de los documentos sobre el espacio que ocupa la solución de Sadakane y se propone una alternativa para reducirlo. Para ambas soluciones se llevaron a cabo una serie de pruebas sobre distintas colecciones para evaluar cómo se comportan en diferentes escenarios. En estas se midió la reducción en espacio que ofrece la estrategia propuesta y cómo esto afecta en los tiempos de respuesta.

Este documento se organiza de la siguiente manera. En el capítulo 1 se precisan formalmente los conceptos que dan forma a las ideas que se introducen más adelante. Aquí también se describen las principales estructuras que componen las soluciones que se implementaron. En el capítulo 2 se expone la solución de Muthukrishnan para el problema (1) y se describe cómo Sadakane la mejora y extiende para resolver (2). En el capítulo 3 se presenta la propuesta: se explican las causas por las cuales el espacio para la solución de Sadakane crece desmedidamente para ciertas colecciones y cómo se puede mejorar dicha situación. En el capítulo 4 se comienza por describir las características de las colecciones para las pruebas. Luego, se discuten aspectos de la implementación de las estructuras y cada una de las estrategias que

se evaluaron para resolver (2). Al final del capítulo se analizan los resultados obtenidos en las pruebas.

Capítulo 1

Conceptos básicos

1.1. Preliminares

Sea $A[1..n]$ una secuencia de n símbolos de un alfabeto Σ. El i-ésimo *sufijo* de A, A_i, es la subsecuencia $A[i..n]$ de A. El *arreglo de sufijos* $SA[1..n]$ de A es una permutación de $(1 \dots n)$ tal que $A_{SA[i]} \leq_\Sigma A_{SA[j]}$, $\forall j > i$, donde $\leq_\Sigma$ es el orden lexicográfico en Σ^*.

Para una colección de arreglos $\mathcal{A} = \{A^1, \dots, A^k\}$, considere la concatenación $C = A^1\$_1 \dots A^k\$_k$. Los $\$_1, \dots, \$_k$ son símbolos que delimitan los arreglos y que satisfacen $\$_i < \$_j$ si $j > i$ y $\$_i < \sigma \in \Sigma$. El i-ésimo *sufijo generalizado* de C, C_i, es la subsecuencia $C[i..j]$ donde $j = \text{mín}\{j \mid C[j] = \$_\ell \text{ para } \ell > i\}$.

Un *árbol de sufijos* de una secuencia A es un trie con $|A|$ hojas. Cada arco está etiquetado con una subsecuencia de A tal que cada sufijo de A corresponde a la concatenación de las etiquetas de un camino desde la raíz del árbol a una hoja. En cada nodo del trie, los hijos se ordenan según el orden lexicográfico entre las secuencias que etiquetan sus arcos. Si $l_1, \dots, l_n$ son las hojas del árbol de sufijos de acuerdo al orden en que aparecen al recorrer el árbol de sufijos de izquierda a derecha, entonces l_i corresponde al sufijo $A_{SA[i]}$. Dado un nodo v del árbol de sufijos, se denota por $\sigma(v)$ la secuencia obtenida por la concatenación de las secuencias que etiquetan los arcos en el camino desde la raíz hasta v en el orden que aparecen. De los nodos v del árbol de sufijos tales que $\sigma(v)$ contiene el prefijo p, se define el *locus* de p como el nodo con el menor $|\sigma(v)|$.

Un *arreglo de documentos* es una estructura que indica el documento al cual pertenece cada sufijo de la concatenación de una colección de documentos. Sean $\mathcal{D} = \{d_1, \dots, d_k\}$ una colección de documentos donde cada $d_i \in \mathcal{D}$ representa una cadena, $G = d_1\$_1 \dots d_k\$_k$ un arreglo formado por una concatenación de los documentos de $\mathcal{D}$ y SA_G el arreglo de sufijos generalizado de G. Se define el arreglo de documentos D_G como el arreglo $D_G[1..|G|]$ tal que $D_G[i] = j$ si el sufijo $G_{SA_G[i]}$ pertenece al documento d_j.

Dado un arreglo $A[1..n]$ de objetos tomados de un conjunto bien ordenado, el $rmq(\cdot)$ de un intervalo $[l..r]$ de A se define como $rmq_A(l, r) = \text{argmin}_{l \leq i \leq r}\ A[i]$. Análogamente, se define el

$RMQ(\cdot)$ del intervalo $[l..r]$ de A como $RMQ_A(l,r) = \text{argmax}_{l \leq i \leq r}\ A[i]$.

La *transformada de Burrows-Wheeler* o *BWT* [3] es una permutación de una secuencia. Esta es una transformación reversible. Si una secuencia contiene varias subsecuencias que se repiten frecuentemente, entonces su BWT tiene la siguiente propiedad: la probabilidad de ocurrencia de un σ fijo en una posición dada es muy alta si cerca de dicha posición existe otro σ. Una secuencia con esta propiedad se puede compimir eficientemente. La BWT del arreglo A se puede definir en términos de su arreglo de sufijos como $BWT_A[i] = A[(SA[i] - 2 \text{ mód } |A|) + 1]$.

Dada una variable aleatoria discreta X cuyos valores posibles pertenecen al conjunto $\mathcal{A}_X$ y una función de probabilidad $P_X(\cdot)$ Shannon [17] define la *entropía* de $(X, \mathcal{A}_X, P_X)$ como $H(X) = -\sum_{a \in \mathcal{A}_X} P(a) \lg P(a)$ (se asume que $0 \lg 0 = 0$). Manzini [10] define la *entropía empírica* de orden cero de una cadena s, $H_0(s)$, como la entropía de $(X_s, \mathcal{A}_s, P_s)$, donde $\mathcal{A}_s$ es el alfabeto de s y la probabilidad $P_s(\sigma)$ para $\sigma \in \mathcal{A}_s$ es igual a la frecuencia con que ocurre σ en s.

1.2. Representaciones sucintas de árboles

Usualmente para representar los enlaces entre los nodos de un árbol se utilizan punteros a direcciones de memoria. Estructuras basadas en punteros permiten navegar rápidamente a través de los nodos pero ocupan un espacio muy por sobre la cota teórica de información de los objetos que representan. Existen diversas codificaciones compactas para árboles que ofrecen mecanismos para recorrer eficientemente la estructura y consumen mucho menos espacio que las convencionales. Además, en algunos casos, pueden resolver ágilmente una variedad de consultas que en una representación basada en punteros no es posible sin estructuras auxiliares. Considere árboles ordinales generales, esto es, sin restricciones en el grado de sus nodos. El número de estos árboles con un total de n nodos es $\frac{1}{n}\binom{2n-2}{n-1} \approx \frac{4^{n-1}}{\sqrt{\pi n^3}}$. El logaritmo de esta cantidad se aproxima a $2n - 2 - \lg\sqrt{\pi n^3}$ (para n grande) y corresponde a la longitud mínima necesaria de una secuencia de bits para codificar unívocamente cada uno de estos árboles con n nodos. Por lo tanto, un poco menos de $2n$ bits son suficientes para representar la topología de estos árboles. Estos son significativamente menos (para n grande) que los bits que se requieren en una representación basada en punteros ($n\lceil \lg n \rceil$). Más aún si se tiene en cuenta que, en una implementación tradicional, cada dirección de memoria se almacena en una palabra de máquina de w bits; suma que puede ser mayor que los $\lceil \lg n \rceil$ bits necesarios para distinguirlas entre sí. A modo de ejemplo considere que se quiere representar un árbol ordinal con un total de 2^{23} nodos. En una codificación compacta serían necesarios alrededor de 2^{24} bits mientras que en la basada en punteros con direcciones de 32 bits ocuparía 2^{28} bits. Es decir, la segunda ocupa 16 veces el espacio de la primera. Existen distintos métodos para codificar árboles que son más espacio eficiente que las convencionales. A continuación se describen algunos de éstos.

1.2.1. Codificación implícita (Heap)

Un árbol t-ario es un árbol donde cada nodo tiene grado 0 o t. Un árbol t-ario se puede representar secuencialmente, ubicando sus nodos dentro de un arreglo. Para ello se dispone la raíz en la posición 0 del arreglo. Los hijos del nodo en la posición i (del arreglo) se ubican entre las posiciones $ti+1$ y $(t+1)i$; de este modo el padre de i se encuentra en la posición $\lfloor (i-1)/t \rfloor$ (en una representación tradicional para ir desde un nodo a su padre se requieren enlaces dobles). El arreglo tiene un largo de $\sum_{k=0}^{h-1} t^k$, donde h es la altura del árbol. Para un árbol etiquetado, en cada posición del arreglo, se almacena la etiqueta del nodo correspondiente. En caso de que no exista un nodo asociado a esa posición se asigna un valor especial. Esta representación no requiere de enlaces explícitos entre los nodos. Si sólo se quiere codificar la topología del árbol basta con un bitmap del largo señalado donde la existencia de un nodo se indica con un 1 en la posición correspondiente; en caso contrario con un 0. Se debe notar que esta representación de un árbol es espacio eficiente en la medida que éste tenga una estructura suficientemente balanceada; de otro modo se incurre en una pérdida de espacio sustancial. Además, esta representación implícita resulta muy eficiente para la navegación principalmente por dos razones: (1) para visitar un hijo (padre) en lugar acceder a memoria para obtener la dirección del hijo (padre) la calcula con unas pocas operaciones aritméticas sencillas sobre la dirección del hijo (que previamente se puede tener cargada en un registro) y (2) garantiza que el árbol se almacena en un bloque contiguo de memoria lo que ofrece una mejor localidad.

1.2.2. Codificaciones sucintas

BP, LOUDS y DFUDS son representaciones sucintas para árboles no etiquetados que soportan las operaciones básicas de navegación en tiempo constante.

Estas utilizan un bitmap de largo $2n$ para representar la topología del árbol y un diccionario rank/select sobre el bitmap que ocupa $o(n)$ bits adicionales para navegar eficientemente a través de la estructura. LOUDS y DFUDS se basan en que un árbol se puede especificar completamente por la secuencia de los grados de sus nodos en algún orden dado.

LOUDS

El método Level Order Unary Degree Sequence o LOUDS, propuesto por Jacobson [7], consiste en representar un árbol por la secuencia de enteros compuesta por los grados de los nodos ordenados por nivel, de izquierda a derecha. Los enteros se representan con el siguiente código de prefijo binario: dado un entero $i \geq 0$, i se codifica con la cadena 1^i0. Una vez formada la secuencia se le antepone el prefijo 10 (este representa la super-raíz [7]). Cada nodo se identifica con el índice del 1 que le corresponde en el bitmap. A este último se le llamará B. Si las posiciones de B se enumeran desde 1 en adelante, el i-ésimo ($i \geq 1$) hijo del nodo v (*ith-child(v)*) se calcula mediante $select_0(B, rank_1(B, v)) + i$ y el padre de v ($parent(v)$) mediante $select_1(B, rank_0(B, i))$.

BP

Balanced Parenthesis o BP [7, 11] es un método que utiliza una secuencia de paréntesis balanceados para representar un árbol. Esta secuencia se construye haciendo un recorrido depth-first por el árbol. Al visitar un nodo por primera vez, se agrega un paréntesis de apertura a la secuencia y al regresar desde un nodo hacia su padre se añade un paréntesis de cierre. Al concluir el recorrido se agrega un paréntesis de cierre adicional. De esta manera la codificación que se obtiene es una cadena de paréntesis balanceados. En este contexto resulta conveniente expresar las operaciones sobre secuencias de paréntesis. Se conviene entonces que un paréntesis de apertura corresponde a un 1 y uno de cierre a un 0. De este modo $rank_{(}()$ y $rank_{)}(\cdot)$ equivalen a $rank_1(\cdot)$ y $rank_0(\cdot)$. Las equivalencias para *select* son análogas. Las siguientes operaciones, definidas sobre cadenas de paréntesis balanceados, se pueden realizar con un número constante de aplicaciones de rank y select [2]:

- $findopen(i)$:
  encuentra la posición del paréntesis de apertura que le corresponde al paréntesis de cierre en la posición i.
- $findclose(i)$:
  encuentra la posición del paréntesis de cierre que le corresponde al paréntesis de apertura en la posición i.
- $enclose(i)$:
  dado un paréntesis de apertura en la posición i, calcula la posición del paréntesis más cercano al paréntesis de apertura que encierra (junto con su pareja) a el paréntesis en i.

Cada nodo se corresponde con un paréntesis de apertura y se identifica con el índice de éste en la cadena de paréntesis balanceados. El primer hijo del nodo v (*first-child(v)*) es el índice $v+1$. El siguiente hermano de v (*next-sibling(v)*) se obtiene con $findclose(v)+1$ y el padre de v (*parent(v)*) con $enclose(v)$.

DFUDS

Depth First Unary Degree Sequence o DFUDS [2] es una representación que combina características de LOUDS y BP. Al igual que BP, DFUDS codifica un árbol en una secuencia de paréntesis balanceados. Esta secuencia se obtiene de la concatenación de los grados de los nodos al recorrer el árbol en orden depth-first. Un entero $i \geq 0$ se codifica con la cadena de paréntesis $(^i)$. Para que la cadena sea balanceada es necesario agregarle un paréntesis de apertura al comienzo. Un nodo se identifica con la posición del primer paréntesis con el que se codifica su grado en la secuencia. Sea B la cadena de paréntesis balanceados que representa un árbol en DFUDS. Las operaciones básicas de navegación se computan de la siguiente manera: el i-ésimo hijo del nodo v (*ith-child(v)*) se calcula con $findclose(select_{)}(B, rank_{)}(B, v)+1)-i)+1$. El padre de v (*parent(v)*) se resuelve con $select_{)}(rank_{)}(findopen(v-1)))+1$.

1.3. Diccionarios rank/select

Una gran variedad de estructuras compactas dependen de una representación sucinta de secuencias que ofrezca soporte para resolver eficientemente dos operaciones fundamentales: *rank* y *select*. En su forma más general estas operaciones se definen de la siguiente manera. Dada una secuencia $A[1..n]$ sobre un alfabeto Σ, un símbolo $\sigma \in \Sigma$ y un índice $1 \leq i \leq n$, $rank_\sigma(A, i)$ es el número de ocurrencias de σ en el prefijo $A[1..i]$; y $select_\sigma(A, i)$ es la posición en A donde σ ocurre por i-ésima vez. Formalmente,

$$rank_\sigma(A, i) = |\{j \mid A[j] = \sigma,\ 1 \leq j \leq i\}|$$

$$select_\sigma(A, i) = \text{mín}\{j \mid rank_\sigma(A, j) = i\}.$$

Los diccionarios rank/select se pueden dividir en dos categorías: diccionarios binarios y diccionarios sobre secuencias generales. Además de la distinción evidente entre los tamaños de los alfabetos de uno y otro, difieren en cómo se representan las secuencias para soportar eficientemente las operaciones rank y select. Existe una gran variedad de artículos sobre diccionarios rank/select sobre secuencias binarias. Estos presentan distintas propiedades que resultan más o menos adecuados según la aplicación.

1.3.1. Bitmaps RRR

Sea B un bitmap de largo u con t bits encendidos, entonces B se puede representar implícitamente por una cadena de $\lceil \log \binom{u}{t} \rceil$ bits almacenando un índice v respecto de una tabla que contiene los vectores de bits característicos de todos los posibles bitmaps de largo u con t bits encendidos.

Dado un bitmap B de longitud n, sea $u = f(n)$. Se divide B en $p = \lceil n/u \rceil$ bloques de u bits cada uno. Sea U_i el bitmap de largo u correspondiente a la subcadena de B desde $(i-1)u$ hasta $iu-1$, para $1 \leq i \leq p-1$ y U_p el bitmap correspondiente a la subcadena de los últimos $n \bmod u$ bits de B (i.e. desde $(p-1)u$ hasta $n-1$). Cada bitmap U_i con u_i bits encendidos queda completamente especificado por la tupla (u_i, v_i), donde v_i es un índice como el que se describió previamente. Entonces B se representa por la concatenación de las tuplas (u_i, v_i). El primer componente es representado con un campo de longitud fija de $\lceil \lg(u+1) \rceil$ bits. El segundo es representado con un campo de longitud variable de $\lceil \lg \binom{u}{v_i} \rceil$ bits.

La representación de B como una secuencia de (u_i, v_i) tiene los siguientes requerimientos de espacio: para todos los valores u_i son necesarios $\mathcal{O}(n \lg(f(n)+1)/f(n))$ bits y para todos los valores v_i es $nH_0(B) + \mathcal{O}(n/f(n))$ bits.

Al elegir $u = \lceil \frac{\lg n}{2} \rceil$ se obtiene que el espacio ocupado por los u_i's suma $\mathcal{O}(n \lg \lg n / \lg n)$ bits y por los v_i's es $nH_0(B) + \mathcal{O}(n/\lg n)$ bits. De este modo se consigue un total de $nH_0(B) + o(n)$ bits.

Para soportar operaciones *rank* en tiempo constante sobre el bitmap B se utilizan tres tablas: R, S y T. R almacena respuestas de $rank_1(\cdot)$ para algunas posiciones (de B) en intervalos regulares, S la suma relativa de 1's dentro de bloques de B de un largo fijo y

T, para cada una de las posibles secuencias de bits (de una longitud corta dada), todos los valores de $rank_1(\cdot)$ sobre estas.

Sean r, s y u unos enteros positivos. Los primeros dos corresponderán a los tamaños de los intervalos considerados en las tablas R y S, respectivamente, y el último al largo de los bitmaps de T. Para $1 \leq i < n/r$, la entrada i de la tabla R contiene el valor de $rank_1(B, ir-1)$. La segunda tabla en la entrada i, $1 \leq i < n/s$, almacena la diferencia entre $rank_1(B, i\,s-1)$ y $R[\lfloor i\,s/r \rfloor]$. Ambas tablas en la posición 0 contienen el valor 0 (i.e. $R[0] = S[0] = 0$). En T se reúnen todas las respuestas a $rank_1(\cdot)$ para cada secuencia de bits de largo u. Más precisamente, se define B_i como i-ésimo bitmap (en orden numérico) de largo u, entonces en la entrada (i, j), para $0 \leq i < 2^u$, $0 \leq j < u$, T contiene el valor de $rank_1(B_i, j)$.

Las consultas de *rank* sobre B se pueden responder en tiempo constante con estas tablas. Sea $\phi_u(B, j)$ una función que relaciona la cadena de bits de largo u de B que comienza en la posición j, donde $0 \leq j \leq n-u$ y $u > 0$, con el índice que le corresponde dentro de la tabla T. Dada una posición i de B, esta se puede descomponer como $i = q\,r - 1 + p = q'\,s - 1 + p'$, con $0 \leq p < r$, $0 \leq p' < s$ y $p' = q''\,u + p''$ tal que $0 \leq p'' < u$, entonces $rank_1(B, i) = R[q] + S[q'] + \sum_{j=0}^{q''-1} T[\phi_u(B, q'\,s + j\,u), u-1] + T[\phi_{p''}(B, q'\,s + q''\,u), p''-1]$. $rank_0(\cdot)$ se puede calcular mediante $rank_1(\cdot)$ con $rank_0(B, i) = i - rank_1(B, i)$ [1].

Si se eligen para los parámetros r, s y u los valores $\lg^2 n$, $\lg n$ y $\frac{\lg n}{2}$, respectivamente, el espacio requerido para R es $\mathcal{O}(n \lg n / \lg^2 n)$, para S es $\mathcal{O}(n \lg \lg n / \lg n)$ y para T es $\mathcal{O}(\sqrt{n} \lg n \lg \lg n)$. De este modo RRR consigue $rank_b(\cdot)$ en tiempo constante con espacio $o(n)$.

1.3.2. Árboles Wavelet

Para diccionarios rank/select sobre secuencias generales, el árbol wavelet es una alternativa. Éste es una estructura de datos introducida por Grossi et al. [6] que descompone una secuencia general en un conjunto de secuencias binarias. Dada una secuencia definida sobre un alfabeto Σ, permite responder consultas rank/select en tiempo $\mathcal{O}(log|\Sigma|)$. La estructura es posible organizarla de diversas maneras para ajustar el espacio y obtener una representación más compacta.

Un árbol wavelet sobre una secuencia $A[1..n]$ definida sobre un alfabeto Σ es un árbol binario balanceado etiquetado. Si $|\Sigma| = 1$ entonces el árbol es una hoja cuya etiqueta contiene el largo de la secuencia, n. De otro modo la raíz está etiquetada con el bitmap $B_{root}[1..n]$, que se define como: $B_{root}[i] = 0$ si $A[i] = \sigma_j$ y $j \leq \lfloor |\Sigma|/2 \rfloor$, en caso contrario $B_{root}[i] = 1$. Los hijos izquierdo y derecho son árboles wavelet sobre las subsecuencias $A_0[1..m]$ y $A_1[1..n-m]$ de A, donde $0 \leq m \leq n$. A_0 es la subsecuencia de A formada por la secuencia de los símbolos σ_j en A tales $j \leq \lfloor |\Sigma|/2 \rfloor$. Análogamente se define la subsecuencia A_1 de A compuesta por los símbolos σ_j en A tales $j > \lfloor |\Sigma|/2 \rfloor$.

El árbol wavelet tiene una profundidad de $\lceil \lg |\Sigma| \rceil$. La suma de las longitudes de los bitmaps

[1] Con la finalidad de facilitar la transición de la teoría a la práctica, sólo en esta sección (Bitmaps RRR) donde predominan las sutilezas, las posiciones sobre las cuales opera $rank(\cdot)$ se numeran desde 0.

de cada nivel es n; con la salvedad del último nivel que puede ser menor. El conjunto de todos los bitmaps ocupa a lo más $n \lg |\Sigma|$ bits. Este árbol tiene exactamente $|\Sigma|$ hojas y $|\Sigma| - 1$ nodos internos. El valor que cada hoja almacena es el número de ocurrencias, dentro de la secuencia A, del símbolo que representan. El almacenamiento del contenido de todas las hojas ocupa $|\Sigma| \lg n$ bits.

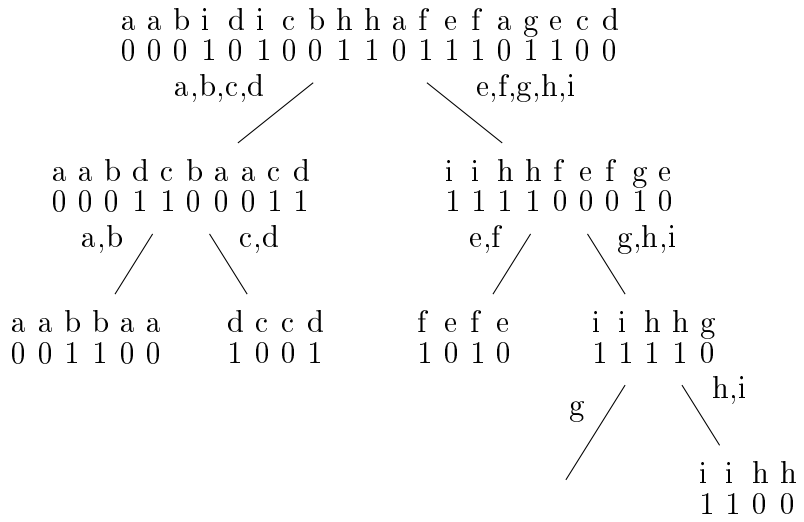

Figura 1.1: Árbol wavelet para la secuencia $A = aabidicbhhafefagecd$. Las hojas no se dibujan.

El espacio ocupado por un árbol wavelet es posible reducirlo mediante distintas técnicas; éstas, en general, se caracterizan por comprimir la estructura. El costo de esta reducción de espacio se manifiesta en un aumento en el tiempo que tarda una consulta sobre la estructura. Para disminuir la redundancia del árbol wavelet se distinguen dos enfoques: comprimir los bitmaps y comprimir la topología. Un árbol wavelet compacto es un árbol wavelet que utiliza algún método de compresión. Es importante señalar que la estructura debe retener la capacidad de operar eficientemente una vez aplicado el esquema de compresión.

El espacio total de un árbol wavelet sin compresión está acotado por $n\lceil \lg |\Sigma| \rceil + |\Sigma| \lg n + 2w(|\Sigma| - 1)$, donde w es el número de bits de las palabras de la máquina (nótese que para que el árbol wavelet ofrezca un soporte eficiente de las operaciones rank y select es necesario poder resolver ágilmente consultas rank/select sobre los bitmaps del árbol. Para ello cada bitmap B requiere de $o(|B|)$ bits adicionales, los que suman un total de $o(n \lg |\Sigma|)$ bits). Para comprimir los bitmaps se puede a recurrir a RRR (sección 1.3.1) y para la topología a cualquiera de los métodos descritos en la sección 1.2.

1.3.3. rank, select y access

Las operaciones rank, select y access se resuelven en tiempo $O(\lg |\Sigma|)$. Antes de describirlas es necesario introducir algunas definiciones. Sean $lch(\cdot)$, $rch(\cdot)$ y $par(\cdot)$ funciones entre los nodos de un árbol binario. Las primeras dos asocian un nodo con sus hijos izquierdo y derecho,

respectivamente; la tercera con su padre. Dado un nodo v de un árbol wavelet, se define B_v como el bitmap que forma parte de la etiqueta dicho nodo.

$rank_\sigma(v, i)$:
Se recorre el árbol desde la raíz hasta llegar a la hoja correspondiente a σ en el árbol wavelet. En el recorrido se va actualizando el valor de i al que le corresponde al nodo que está visitando. Una vez alcanzada la hoja el valor de i es la respuesta de la consulta. Más precisamente, rank se resuelve recursivamente de la siguiente manera: si v es una hoja retornar i. De otro modo, si $\sigma < \lfloor |\Sigma_v|/2 \rfloor$ se aplica $rank_\sigma(lch(v), rank_0(B_v, i))$; en caso contrario $rank_\sigma(rch(v), rank_1(B_v, i))$.

$select_\sigma(v, i)$:
En este caso se procede en sentido inverso, desde una hoja hasta la raíz. Si v es un hijo izquierdo entonces se actualiza $i = select_0(B_{par(v)}, i)$, si es un hijo derecho $i = select_1(B_{par(v)}, i)$. Luego se asigna $v = par(v)$ y se repite el proceso hasta llegar a la raíz.

$access(v, i)$:
Esta operación accede al símbolo en la posición de la secuencia A. Para encontrar $A[i]$ a partir del árbol wavelet sobre A se recorre el árbol desde la raíz hasta alcanzar una hoja, una vez ahí se deduce el símbolo $A[i]$. En cada nivel de la recursión se accede al bit $B_v[i]$. Entonces si su valor es 0 se baja por la rama izquierda del árbol y el sucesor $s = lch(v)$, si no por la derecha y $s = rch(v)$. Luego se aplica $access(s, rank_{B_v[i]}(B_v, i))$.

1.4. Auto-índice

Un índice es una estructura de datos que permite resolver consultas sobre textos (aquí se utiliza texto en un sentido amplio; como una secuencia arbitraria de símbolos) sin la necesidad de recorrer secuencialmente el texto en su totalidad. Los índices proveen funcionalidades para resolver rápidamente consultas sobre grandes colecciones de textos. Entre sus aplicaciones más comunes está la de, dado un patrón de consulta, encontrar el número de ocurrencias (*count*) y las ubicaciones (*locate*) de éste dentro de un texto. Los árboles de sufijos [18] y arreglos de sufijos [9] son índices de texto completo (i.e. permiten indexar patrones arbitrarios, a diferencia de otros, como un índice invertido, que sólo puede indexar patrones del texto que pueden ser sintácticamente separados) que pueden resolver count y locate rápidamente. Sin embargo, estos índices tradicionales consumen entre 4 y 20 veces el espacio ocupado por el texto. Además, requieren acceder al texto para evaluar count y locate. Esta excesiva demanda de espacio es prohibitiva para muchas aplicaciones prácticas. Un auto-índice es un índice que junto con proveer funcionalidades para responder eficientemente consultas sobre un texto contiene al texto en una representación compacta y a la vez puede extraer subsecuencias de éste rápidamente (sin tener que descomprimir todo el texto).

A continuación se describen las operaciones fundamentales ($SA(\cdot)$ y $LF(\cdot)$) sobre las cuales descansa el funcionamiento los arreglos de sufijos compactos del tipo FM [4]. Además, se incluye la función $SA^{-1}(\cdot)$, que es necesaria para aplicar los algoritmos de listado de documentos

con frecuencias de términos que se explican más adelante.

1.4.1. Funciones $SA(\cdot), SA^{-1}(\cdot)$ y $LF(\cdot)$

La función $SA(\cdot)$, dados una secuencia A y un entero i entre 1 y $|A|$, entrega la posición (dentro de A) que ocupa el i-ésimo sufijo (en orden lexicográfico) de A. $SA^{-1}(\cdot)$ es su inversa.

Ambas funciones se pueden implementar recorriendo hacia atrás el arreglo A a partir de su BWT. Para ello se utiliza la función llamada $LF(\cdot)$ [4].

Definición 1. La función $LF : [1..n] \rightarrow [1..n]$, donde $n = |A|$, es tal que $LF(i)$ es el índice de SA tal que el elemento $BWT_A[i]$ ocurre en la posición $SA[LF(i)]$ de A. Asimismo, $LF(\cdot)$ se puede definir implícitamente por la ecuación

$$SA[LF[i]] = SA[i] - 1. \tag{1.1}$$

Lema 1 Para $k \geq 1$, $SA[i] = SA[LF^k(i)] + k$.

Esta igualdad se deriva de aplicar k veces la ecuación 1.1 sobre sí misma.

A continuación se describe cómo soportar $SA(\cdot)$ y $SA^{-1}(\cdot)$ en términos de $LF(\cdot)$ y una submuestra de sus valores. Luego se explica cómo calcular $LF(\cdot)$.

Función: $SA(\cdot)$
En términos generales la forma en la cual se soporta $SA(\cdot)$ consiste en construir el arreglo de sufijos y conservar sólo un subconjunto de sus valores, a partir de los cuales es posible obtener cualquier otro valor mediante aplicaciones de $LF(\cdot)$.

Se construye un bitmap $B[1..n]$, donde $n = |A|$, con $B[i] = 1$ si $SA(i)$ apunta a una posición muestreada de A y se almacena en el arreglo, $SA'[1..rank_1(B, n)]$, una muestra de los valores del arreglo de sufijos tal que $SA'[i] = SA(select_1(B, i))$.

Una vez descartado el arreglo de sufijos, los valores de $SA(\cdot)$ se calculan de la siguiente manera: si $SA(i)$ está muestreado se obtiene directamente; de otro modo, se busca el menor k tal que $B[LF^k(i)] = 1$. Luego aplicando el lema 1 se tiene que $SA(i) = SA'[rank_1(B, LF^k(i))]+k$.

Función Inversa: $SA^{-1}(\cdot)$
La inversa del arreglo de sufijos se implementa de similar modo. Se crea un arreglo, $I[0..\lceil n/b \rceil + 1]$, en el que se muestrea cada b valores de SA^{-1} de manera tal que $I[i] = SA^{-1}[b \cdot i]$ para $i \geq 0$. Luego, dada una posición i de A, para obtener $SA^{-1}(i)$ se procede de la siguiente manera: si su inversa está muestreada en I (esto es, si $i \mod b = 0$) es trivial; de otro modo, se busca

la primera posición, j, muestreada a la derecha de i. Esta corresponde a $j = \lfloor i/b \rfloor + 1$. Como i está $k = \text{mín}(j \cdot b, n-1) - i$ puestos atrás de j, $SA^{-1}(i) = LF^k(I[j])$.

Función: $LF(\cdot)$
Considere el arreglo $E_A[1..|\Sigma|]$ [2] tal que $E_A[\sigma]$ es el número de ocurrencias de símbolos menores que σ en A. En términos de E_A y $rank(\cdot)$ se puede formular $LF(\cdot)$ como $LF(i) = E_A[\sigma] + rank_\sigma(BWT_A, i)$ donde $\sigma = BWT_A[i]$.

[2] En [4] lo llaman C. Aquí se denota por E para evitar confusiones con el arreglo C de Muthukrishnan.

Capítulo 2

Trabajo Relacionado

2.1. Listado de Documentos

Solución de Muthukrishnan

A continuación se describe la solución de Muthukrishnan [12] para el problema de listado de documentos. Dado un patrón p, primero busca el locus de p, u_p, en el GST de la colección. Luego busca las hojas l_{sp} y l_{ep} que corresponden a las hojas que se encuentran más hacia la izquierda y más hacia la derecha del subárbol con raíz u_p. Dado que $D[i]$ indica el documento al que pertenece el sufijo apuntado por i, el problema del listado de documentos se reduce a listar los distintos valores en $D[sp..ep]$. Para poder resolver esta consulta en tiempo óptimo $\mathcal{O}(|p| + ndocs)$ Muthukrishnan utiliza un arreglo C que encadena las ocurrencias de los sufijos del mismo documento. C representa una lista de listas enlazadas donde cada elemento de C, $C[i]$, apunta a la hoja predecesora del GST que contiene un sufijo que ocurre en el documento $D[i]$. Formalmente, se define C como el arreglo $C[1..|D|]$ tal que $C[i] = \text{máx}\{m \mid m < i, D[i] = D[m]\}$; si no existe dicho máximo $C[i] = -1$. Muthukrishnan muestra que los documentos $D[i]$ tales que $i \in [sp..ep]$ y $C[i] < sp$ forman una lista de los documentos (sin repetición) que contienen p. Esta lista se puede obtener en tiempo $\mathcal{O}(|p| + ndocs)$ y tiene una complejidad de espacio $\mathcal{O}(n \log n)$ bits (ver figura 2.1).

Solución de Sadakane

El algoritmo de Sadakane [16] es esencialmente el mismo que el de Muthukrishnan. La diferencia está en que las estructuras que utiliza para reducir el espacio introducen unas pequeñas variaciones. Sadakane no representa los arreglos C y D explícitamente, en lugar de ello, construye un rmq sucinto sobre C, y calcula D a partir de un conjunto que llama D' y el arreglo de sufijos SA. D' es el conjunto (ordenado) de las posiciones donde comienza cada documento dentro de la concatenación A. Si j es el número de posiciones en D' menores o iguales a $SA[i]$ entonces $D[i] = j$. Para ello D' se representa con un bitmap que marca con 1 posiciones de inicio de cada documento en A; de manera que los valores $D[i]$ se pueden reproducir con $rank_1(D', SA[i])$. Además, sustituye el árbol de sufijos por un arreglo de sufijos compacto (CSA). Pero la principal diferencia con la solución de Muthukrishnan se encuentra en que al no poder acceder a los valores C, para evitar repeticiones de documentos, Sadakane

utiliza un vector de bits $V[1..|D|]$ para marcar los documentos que ya han sido mostrados. Los algoritmos de Muthukrishnan y Sadakane se ilutran en la figura 2.1.

Si se conoce el valor de $SA[i]$, $D[i]$ se puede calcular en tiempo $\mathcal{O}(1)$ con una consulta $rank(\cdot)$ sobre una estructura de datos que representa D' de tamaño $\mathcal{O}(k \log \frac{n}{k}) + o(n)$ bits (sección 1.3.1). Para la consulta $rmq(\cdot)$ sobre C, Sadakane diseña un rmq sucinto que ocupa $4n + o(n)$ bits y tiene complejidad de tiempo $\mathcal{O}(1)$. En total, la solución de Sadakane, ocupa $|CSA| + 4n + o(n) + \mathcal{O}(k \log \frac{n}{k})$ bits.

```
1: procedure MUTHU_DLP(s, sp, ep)
2:     if sp > ep then return
3:     end if
4:     j ← rmq_C(sp, ep)
5:     if C[j] < s then
6:         output D[j]
7:         Muthu(s, sp, j − 1)
8:         Muthu(s, j + 1, ep)
9:     end if
10: end procedure
```

```
1: procedure SADA_DLP(sp, ep)
2:     if sp > ep then return
3:     end if
4:     j ← rmq_C(sp, ep)
5:     ℓ ← rank_1(D', SA[j])
6:     if V[ℓ] = 0 then
7:         output ℓ
8:         V[ℓ] ← 1
9:         Sada_DLP(sp, j − 1)
10:        Sada_DLP(j + 1, ep)
11:    end if
12: end procedure
```

Figura 2.1: Algoritmos de Muthukrishnan (izquierda) y de Sadakane (derecha) para el listado de documentos.

2.2. Listado de documentos con sus frecuencias

Sadakane propone construir para cada documento d_ℓ un arreglo de sufijos compacto CSA_ℓ. Para encontrar el número de ocurrencias o la frecuencia de un patrón p en el documento d_ℓ, $freq(p, d_\ell)$, primero determina el intervalo $[sp..ep]$ correspondiente a p en CSA_ℓ. El intervalo $[sp..ep]$ delimita todos los sufijos que tienen p como prefijo, por lo tanto, $freq(p, d) = ep - sp + 1$. Para resolver el problema de cálculo de frecuencias, esto es, encontrar las frecuencias con que ocurre p en cada documento $d \in \mathcal{D}$, se utiliza la solución del listado de documentos para obtener dichos documentos. A los k arreglos de sufijos compactos se le suman todas las estructuras para resolver el problema del listado de documentos. El algoritmo que propone Sadakane para el cálculo de frecuencia se describe a continuación.

Paso 1. Se dertermina el intervalo $[sp..ep]$ que delimita p en el CSA global.

Paso 2. Para cada $\ell \in D[sp..ep]$ se calculan los índices $i, j \in [sp..ep]$ de más hacia la izquierda y más hacia la derecha tales que $D[i] = D[j] = \ell$. Esto es equivalente a calcular $i = rmq_C(sp, ep)$ y $j = RMQ_{C'}(sp, ep)$, donde C es el arreglo del mismo nombre en la solución del listado de documentos y C' es un arreglo que se describe a continuación. C' representa una lista de listas enlazadas al igual que C pero difiere en que cada elemento apunta a los

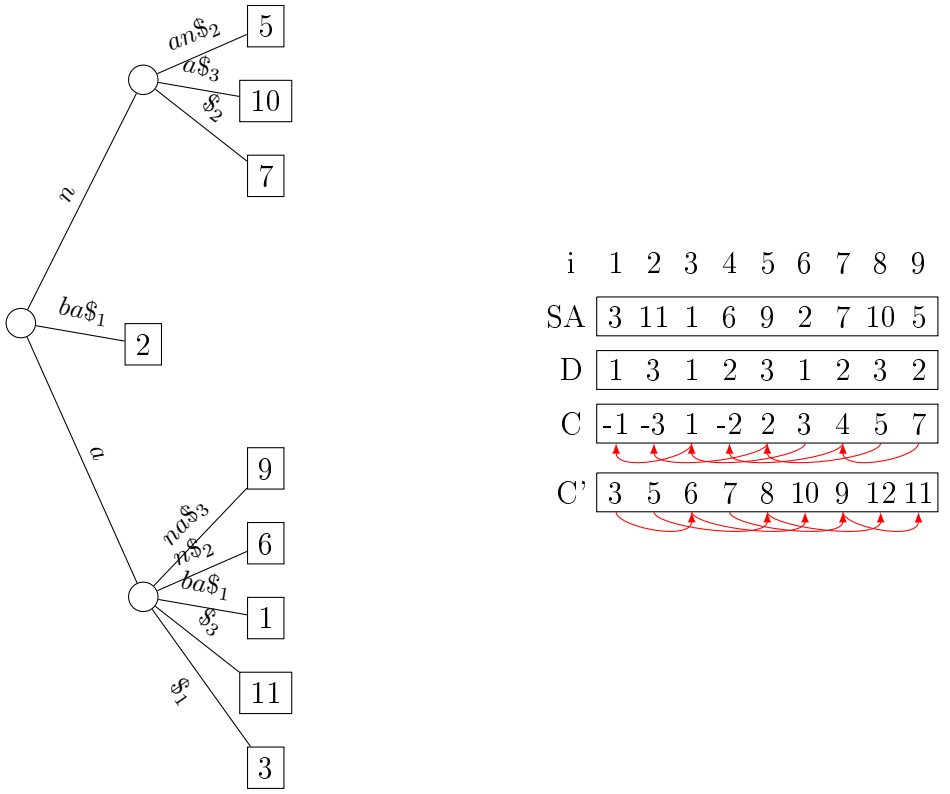

Figura 2.2: (izquierda) Árbol de sufijos de la concatenación $A = aba\$_1nan\$_2ana\$_3$. (derecha) Arreglo de sufijos, arreglo de documentos y arreglos C y C' sobre la secuencia C.

sufijos que le suceden dentro del mismo documento al que pertenecen. Formalmente, C' se define como el arreglo $C'[1..|D|]$ tal que $C'[i] = \text{mín}\{m \mid m > i, D[i] = D[m]\}$; si no existe dicho mínimo $C'[i] = |D| + D[i]$. Estas estructuras se ilustran en la figura 2.2. El algoritmo *Sada_DLP* se puede generalizar adecuadamente para encontrar los índices i, j. Primero, se extienden los parámetros que recibe *Sada_DLP* con la variable *type* que indicará cuáles índices listar. Se conviene que para especificar los índices de inicio (i) se utiliza el valor 0 en *type* y para los de término (j) el valor 0. En la cuarta línea se aplica $j \leftarrow rmq_C(sp, ep)$ si el valor de *type* es 0; si no se aplica $j \leftarrow RMQ_{C'}(sp, ep)$. Para listar los índices de interés en lugar de los documentos es necesario sustituir la séptima línea por **output** j. Además, el orden de recursión depende del valor de *type*. Si éste es 1 se debe invertir el orden, esto es, primero se evalúa la recursión sobre el intervalo $[j + 1..ep]$ y luego sobre $[sp..j - 1]$. Se llamará a esta versión $Sada\_DILP_G$. Entonces, para obtener los i, j para cada $\ell \in D[sp..ep]$, basta con aplicar $Sada\_DILP_G(sp, ep, 0)$ para los índices i y $Sada\_DILP_G(sp, ep, 1)$ para los índices j. En la figura 2.3 se muestra este algoritmo. Nótese que los índices i, j obtenidos con $Sada\_DILP_G$, en general, no están ordenados por lo cual es necesario ordenar cada una de las listas antes de proceder con el siguiente paso.

Paso 3. A partir los pares de índices i_ℓ, j_ℓ para $\ell \in D[sp, ep]$ obtenidos en el paso anterior, se calculan los índices i'_ℓ, j'_ℓ de CSA_ℓ tales que el par $CSA[i_\ell]$ y $CSA_\ell[i'_\ell]$ referencian a un mismo sufijo y a su vez el par $CSA[j_\ell]$ y $CSA_\ell[j'_\ell]$ también referencian a un mismo sufijo.

Para calcular los i'_ℓ, j'_ℓ se parte por observar la siguiente propiedad: todas las posiciones de los sufijos prefijados por p se encuentran en un intervalo contiguo en CSA, y el orden relativo

del subconjunto de estas posiciones que referencian sufijos de un documento d_ℓ es el mismo en el que aparecen en CSA_ℓ.

Se procede primero por encontrar las posiciones globales $spg = CSA[i_\ell]$ y $epg = CSA[j_\ell]$. Luego, para convertir spg y epg en las posiciones locales correspondientes, spl y epl, se les debe restar la posición inicial que tiene el documento ℓ dentro de la concatenación. Del bitmap D' se puede extraer la posición inicial t_ℓ del documento d_ℓ dentro de la concatenación mediante $select_1(D', rank_1(D', spg))$. Entonces, $spl = spg - t_\ell + 1$ y $epl = epg - t_\ell + 1$. Finalmente, los índices i'_ℓ, j'_ℓ se obtienen aplicando $SA_\ell^{-1}(\cdot)$, esto es, $i'_\ell = SA_\ell^{-1}(spl)$ y $j'_\ell = SA_\ell^{-1}(epl)$.

Sadakane [16] resuelve el problema de cálculo de frecuencias en tiempo $\mathcal{O}(Search(p) + ndocs(Lookup(n) + \log\log ndocs))$, donde $Search(p)$ es el tiempo que tarda encontrar el patrón p y $Lookup(n)$ es el tiempo para calcular una entrada del arreglo de sufijos ($SA(i)$) o de su inversa ($SA^{-1}(i)$) a partir del arreglo de sufijos compacto. La complejidad de espacio de esta solución es $2|CSA| + 8n + o(n) + \mathcal{O}(k \log \frac{n}{k})$ bits.[1]

```
1:  procedure SADA_DILP_G(sp, ep, type)
2:      if sp > ep then return
3:      end if
4:      if type = 0 then
5:          j ← rmq_C(sp, ep)
6:      else
7:          j ← RMQ_C'(sp, ep)
8:      end if
9:      ℓ ← rank_1(D', SA[j])
10:     if V[ℓ] = 0 then
11:         output j
12:         V[ℓ] ← 1
13:         if type = 0 then
14:             Sada_DILP_G(sp, j − 1, type)
15:             Sada_DILP_G(j + 1, ep, type)
16:         else
17:             Sada_DILP_G(j + 1, ep, type)
18:             Sada_DILP_G(sp, j − 1, type)
19:         end if
20:     end if
21: end procedure
```

Figura 2.3: Algoritmo para listar, sin repetición, los índices de inicio y término.

[1] $8n + o(n)$bits se deben a la implementación de los rmq de C y C' que utiliza Sadakane, esta componente se puede reducir a $4n + o(n)$ con implementaciones actuales.

Capítulo 3

Nueva estrategia para reducir el espacio

3.1. Diagnóstico

En la práctica, los recursos espaciales que exige la solución de Sadakane son tan elevados que resulta una alternativa inviable para aplicaciones sobre colecciones de volúmenes moderados o grandes [14]. Uno de los problemas presentes en esta estrategia, que influye directamente sobre la demanda desmesurada de espacio, es que el arreglo de los CSA_ℓ's sobre los documentos no comprime bien. Cada CSA_ℓ se compone, principalmente, de tres elementos: una tabla de acumulación (para cada símbolo σ contiene una entrada que indica el número de símbolos presentes en la secuencia A que son menores que σ), un árbol wavelet compacto sobre la BWT_A y una muestra de los valores del arreglo de sufijos invertido de A. Los árboles wavelet, además de los bitmaps, se componen, a su vez, de otras tablas. Estas últimas junto con las tablas de acumulación de los CSA_ℓ's no se encuentran en una representación compacta. Más aún, para una colección de documentos, en la medida que mayor es el número de símbolos que los alfabetos de sus documentos comparten entre sí mayor es la información redundante presente en las tablas de la colección de los CSA_ℓ's correspondientes a dichos documentos. A modo de ejemplo, considere las colecciones d_1, d_2 y d_2, d_3, donde $d_1 = abc$, $d_2 = abd$ y $d_3 = ace$. La información redundante asociada a las tablas de la primera colección es mayor que la de la segunda. Luego, la tendencia del tamaño del arreglo de CSA_ℓ's es a moverse en la misma dirección del número de documentos que comprende la colección, independientemente del volumen de ésta. Es decir, a mayor número de documentos mayor es la información redundante asociada a dicho arreglo.

Considérese dos colecciones, una con $ndocs_1$ documentos y la otra con $ndocs_2$, ambas distribuidas sobre un mismo contenido, donde $ndocs_2$ es varias veces mayor que $ndocs_1$. Incluso si el factor que relaciona esas cantidades no es muy grande, la diferencia entre los tamaños de las estructuras de los CSA_ℓ's es importante. En la tabla 3.1 se puede apreciar este fenómeno. Al comparar las colecciones de las primera y segunda filas, se advierte que el espacio ocupado por los árboles wavelet y las tablas asociadas al arreglo de los CSA_ℓ's se extiende entre una y tres cuartas partes su valor. Las colecciones de English200 experimentan un crecimiento más pronunciado debido a que tienen un alfabeto más extenso que las otras.

	English200		Proteins200		DNA200	
factor	bpc	%	bpc	%	bpc	%
1	5,13	83	4,88	82	2,60	72
4	8,63	89	6,29	86	3,24	76
16	20,13	94	11,88	91	5,93	84
32	55,90	97	34,10	96	16,66	92

Tabla 3.1: El espacio se mide en bpc, esto es, bits por símbolo. Se muestra el espacio que requieren los árboles wavelet y tablas asociados al arreglo de los CSA_ℓ's construidos sobre distintas colecciones. English200, DNA200 y Proteins200 aluden a prefijos, cada uno de 200MB, de los corpus de Pizza&Chili de textos de inglés, secuencias de ADN y secuencias de proteinas, respectivamente (ver sección 4.1). A partir de cada texto se elaboraron cuatro colecciones diferentes. Estas comparten el mismo contenido, pero se distribuyen en un número de documentos distinto. *factor* indica la cantidad de documentos que tiene una colección respecto de aquella que le corresponde en la primera fila. Las colecciones iniciales (i.e. de la primera fila) de los tres textos cuentan con 3,200 documentos cada una. Además, se indica la fracción que representan los árboles wavelet y tablas respecto del espacio total que ocupa el arreglo de los CSA_ℓ's (i.e. incluyendo los valores de las muestras de los arreglos de sufijos invertidos).

Los documentos cortos tienen un efecto extremadamente negativo sobre la compresión de un CSA_ℓ. Una colección de documentos pequeños puede tener una influencia dramática sobre el tamaño del arreglo de CSA_ℓ's debido a que los bitmaps de los árboles wavelet pueden llegar a ocupar un espacio varias veces mayor al de las secuencias que representan. Esto se debe a que la información auxiliar asociada a una representación compacta de un bitmap corto puede llegar a influir más que la información del contenido en el tamaño de la estructura.

En las últimas dos filas de la tabla 3.1 se ve cómo, la combinación de los efectos de una colección numerosa y de documentos pequeños afectan drásticamente el espacio. Para las colecciones de la tercera fila el espacio requerido por los árboles wavelet y tablas es bastante elevado. En la cuarta fila se puede observar que éste es casi triplicado en cada caso, alcanzando hasta más de 55 bpc para English200 y son responsables de más del 90 % del espacio.

En la tabla 3.1 también se aprecia una tendencia al predominio de los árboles wavelet y tablas por sobre el conjunto de los valores muestreados de los arreglos de sufijos invertidos en el volumen de la estructura. Todas estas colecciones fueron construidas con una tasa de muestreo ligeramente superior a un 3 %. Sin embargo, lo mismo sucede incluso para tasas altas de hasta un 10 %, con la salvedad de las colecciones de ADN con 3,200 y 12,800 documentos, donde ocupan alrededor de unas dos quintas partes del espacio; esta particularidad se produce porque el alfabeto de estas colecciones es muy pequeño.

Se debe notar que una representación completa de arreglos de sufijos de 32 bits requiere un poco más de 32 bpc (el espacio adicional se debe a valores para los bordes de los documentos). Por lo tanto, cualquier representación compacta que se mueva por sobre o alrededor de este valor es ineficaz tanto en espacio como en tiempo.

Las desventajas asociadas a la cantidad y extensiones de los documentos de una colección podrían ser evitadas si se reemplazaran las tablas y árboles wavelet locales para cada CSA_ℓ

por una sola tabla y un solo árbol wavelet vinculado a todas las secuencias. Así se conseguiría reducir la información redundante que tiene el primer enfoque.

3.2. Extensión del árbol wavelet

Además de las operaciones tradicionales rank/select/access para llevar a cabo la nueva estrategia se propone la operación $E_\sigma(A, i)$, que corresponde al número de símbolos en el prefijo $A[1..i]$ menores que σ. Formalmente

$$E_\sigma(A, i) = |\{j \mid A[j] < \sigma,\ 1 \leq j \leq i\}|.$$

$E_\sigma(v, i)$:
Es similar al cálculo de $rank_\sigma(\cdot)$. Se recorre el árbol desde la raíz hasta encontrar la hoja que corresponde a σ . El valor de i en cada nivel es actualizado con $rank_0(B_v, i)$ si se siguió el enlace izquierdo para llegar a v o $rank_1(B_v, i) - \lfloor |\Sigma_v|/2 \rfloor$ (donde Σ_v es el alfabeto correspondiente al nodo v) en caso contrario. Asimismo, se mantiene un contador cuyo valor inicial es cero y cada vez que se toma la ruta derecha, se le suma la cantidad de 0's que ocurren en el prefijo $B_v[1..j]$. Una vez alcanzada una hoja, el contador contiene la respuesta. En la figura 3.1 se muestra una implementación de esta función.

```
1:  procedure E(v, i, z, j)
2:      if z = 1 then return 0
3:      end if
4:      f ← ⌊z/2⌋
5:      if i ≤ f then
6:          j' ← rank_0(B_v, j)
7:          return E(lch(v), i, f, j')
8:      else
9:          j' ← rank_1(B_v, j)
10:         return rank_0(B_v, j) + E(rch(v), i − f, z − f, j')
11:     end if
12: end procedure
```

Figura 3.1: Este algoritmo calcula el número de símbolos menores que σ_i en el prefijo $A[1..j]$ dado el árbol wavelet de la secuencia A. v es un nodo del árbol wavelet, i es la posición que ocupa σ_i (en orden ascendente) dentro del alfabeto del nivel actual del árbol wavelet, z es el tamaño del alfabeto del nivel actual, y B_v representa el bitmap del nodo v. El valor de $E_{\sigma_i}(A, j)$ se obtiene con $E(root, i, |\Sigma|, j)$.

nivel de recursión	0	1	2	3
i	4	4	2	1
z	9	4	2	1
j	12	7	2	1
f	4	2	1	–
j'	7	2	1	–
$rank_0(B_v, j)$	7	5	1	–

Tabla 3.2: En esta tabla se muestra un ejemplo de la aplicación del algoritmo $E(wroot, 4, 9, 12)$ para calcular $E_d(A, 12)$ donde $A = aabidicbhhafefagecd$, $\Sigma = \{a, b, c, d, e, f, g, h, i\}$, $|\Sigma| = 9$ y $wroot$ es la raíz del árbol wavelet de la secuencia A.

3.3. Solución propuesta

La forma en la cual se propone abordar los problemas descritos para conseguir una reducción importante en el espacio del índice consiste en reemplazar los CSA_ℓ por una nueva estructura. Por cada documento se construye un arreglo de sufijos invertido compacto, que se denomina $CISA\_E$. Dada un documento d, éste contiene una muestra de los valores de la inversa del arreglo de sufijos sobre d junto con las posiciones de inicio y término del documento d dentro la secuencia donde se concatenan los documentos. En lugar de mantener un árbol wavelet y una tabla de acumulación (secciones 1.3.2 y 3.1) por cada documento, emplea un árbol wavelet global que es compartido por todos los $CISA\_E$'s. Este árbol wavelet que se llamará WT_CBWT es construido sobre la concatenación de las BWT's de los documentos, esto es, sobre la cadena $CBWT = BWT(d_1)..BWT(d_k)$. A partir de esta nueva estructura y la operación $E(\cdot)$ es posible recuperar los valores de $SA_\ell^{-1}(\cdot)$.

El reemplazo de los múltiples árboles wavelet pequeños y tablas de acumulación por un único árbol wavelet reduce significativamente la redundancia asociada a la solución de Sadakane. Sin embargo, este nuevo enfoque supone un aumento en los tiempos de cómputo.

En este contexto la expresión que se presentó en la sección 1.4.1 para obtener $LF(\cdot)$ no es posible aplicarla directamente, no obstante con la ayuda de la operación $E(\cdot)$ introducida en la sección 1.3.2 es simple de ajustar. A partir de la nueva estructura, los valores de las tablas de acumulación de cada documento d se pueden determinar mediante $E_d[\sigma] = E_\sigma(CBWT, ep_d) - E_\sigma(CBWT, sp_d - 1)$, donde sp_d y ep_d son las posiciones que delimitan la BWT del documento d en CBWT. Similarmente, dado que no se cuenta con los árboles wavelet sobre las BWTs para cada documento d, para proporcionar $rank_\sigma(\cdot)$ sobre dichas secuencias, ahora se procede con dos aplicaciones de $rank_\sigma(\cdot)$ sobre WT_CBWT, a saber, $rank_\sigma(BWT_d, i) = rank_\sigma(CBWT, spos + i - 1) + rank_\sigma(CBWT, spos - 1)$.

Nótese que ahora una aplicación de $LF(\cdot)$ para uno de los CSA_ℓ requiere dos bajadas por el árbol WT_CBWT para $E_d[\sigma]$ y otras dos para $rank_\sigma(\cdot)$. En consecuencia cuesta $4 \lg |\Sigma|$ en lugar de $\lg |\Sigma_d| + \mathcal{O}(1)$.

Capítulo 4

Experimentación

4.1. Textos de Prueba

En esta sección se especifican las colecciones sobre las cuales se realizaron las pruebas de las distintas estrategias. Estas colecciones se constituyen de muestras de distintos tipos de datos que representan diferentes escenarios de aplicación.

ClueWiki	Páginas web en inglés extraídas de Wikipedia. Esta colección de 141 MB está formada por 3,334 páginas web.
KGS	Registros de partidas del juego Go del año 2009 en formato sgf (http://www.u-go.net/gamerecords). 18,838 registros componen esta colección de 75 MB.
DNA200	Secuencias de ADN. Esta colección se extrae del prefijo de 200 MB del corpus de secuencias de ADN de Pizza&Chili. Éste es una concatenación secuencias de genes (sin descripciones) que se codifican con las letras A, G, C, T para cada una de las cuatro bases. Además contiene otros pocos símbolos especiales.
English200	Secuencias de textos en inglés del proyecto Gutenberg. Esta colección se extrae del prefijo de 200 MB del corpus de textos de inglés de Pizza&Chili. Éste es una concatenación de archivos de textos seleccionados del proyecto Gutenberg sin incluir los encabezados (http://pizzachili.dcc.uchile.cl/).
Proteins60	Secuencias de proteínas. Esta colección de 60 MB se obtuvo de un archivo que contiene una concatenación de 143,244 secuencias de proteínas de humanos y ratones. Cada uno de los 20 aminoácidos son codificados con una letra mayúscula (http://www.ebi.ac.uk/swissprot).

	ClueWiki	KGS	DNA200	English200	Proteins60
H_0	5,250	4,412	1,947	4,4069	4,231
	(65,63 %)	(55,16 %)	(24,34 %)	(55,08 %)	(52,89 %)
$\|\Sigma\|$	91,377	64,296	4,010	43,965	28,265
largo	41,278	1,398	1,024	1,024	1,024

Tabla 4.1: $H_0, |\Sigma|$ y largo corresponden a las medias de la entropía de orden cero, del número de símbolos del alfabeto y del largo de los documentos de las colecciones. La entropía de orden cero se indica en bits por símbolo. Entre paréntesis aparece la razón de compresión con respecto a almacenar los símbolos en bytes.

Proteins60, English200 y DNA200 se obtuvieron al dividir los respectivos textos en segmentos de 1024 caracteres.

Las colecciones ClueWiki, KGS y Proteins60 son las mismas que se emplearon en el trabajo de Navarro et al. [14], con la excepción de que Proteins60 distribuye su contenido de forma diferente. En la tabla 4.1 se indican las propiedades de los documentos de cada colección.

4.2. Implementación

4.2.1. Estrategias

En este trabajo se implementaron tres estrategias para abordar el problema de listado de documentos con frecuencias de términos. Estas comparten el mismo algoritmo general pero emplean distintas estructuras para su aplicación. Los mejores tiempos de respuesta, en general, coinciden con las soluciones con una mayor complejidad de espacio (más adelante se discuten las condiciones bajo las cuales para algunas estrategias esto no se cumple). Las estrategias implementadas se llamarán: SADA, SGS y FS.

- SADA es la solución de Sadakane. Esta se compone de un *CSA* sobre la concatenación de los contenidos de todos los documentos, que incluye un árbol wavelet sobre la BWT de dicha concatenación, una muestra de los valores del arreglo de sufijos de la concatenación y el bitmap B (RRR); de un arreglo de CSA_ℓ's, del bitmap D' (RRR) y de los RMQs compactos sobre los arreglos C y C'.
- SGS es la nueva estrategia propuesta. Ésta comparte con SADA el *CSA*, el bitmap D' y los RMQs. En lugar de utilizar múltiples árboles wavelet y tablas de acumulación para secuencias locales (las BWT's de los documentos) en un arreglo de CSA_ℓ's, SGS mantiene un árbol wavelet sobre una única secuencia global (CBWT). Junto con ello incluye un muestra de los valores de los arreglos de sufijos invertidos de todos los documentos.
- FS es la estrategia que ofrece los mejores tiempos, pero también es la que más espacio ocupa. Tiene en común con las otras dos las estructuras para el bitmap D', RMQs de C y C' y el árbol wavelet asociado al *CSA*. A diferencia de ellas, no mantiene secuencias locales ni una global, simplemente utiliza una representación completa (no

compacta) para los arreglos de sufijos invertidos sobre los documentos y para el arreglo de sufijos sobre la concatenación. Nótese que, como las BWT's asociadas a los CSA_ℓ's sólo son necesarias para calcular los valores no muestreados de cada CSA_ℓ, estas resultan superfluas si se conocen todos los valores.

4.2.2. Implementación

En esta sección se especifican las estructuras que se utilizaron en este proyecto junto con explicar, en términos generales, cómo se implementaron o de dónde se obtuvieron. Para los diccionarios rank/select compactos RRR se recurre a la implementación de libcds (http://libcds.recoded.cl).

Para los RMQs se recurre a la implementación de Giuseppe Ottaviano (https://github.com/ot/succinct) de la estructura de Fischer y Heun [5].

Para los árboles wavelet con rank/access/E se implementó una estructura ligera que se integra fácilmente con bitmaps externos.

Para la construcción de los arreglos de sufijos se desarrolló una implementación del algoritmo de Ge Non et. al. [15] con unas modificaciones que permiten aplicarlo sobre secuencias con alfabetos medianos o grandes (i.e. sin demandar cantidades excesivas de memoria). En el anexo se entrega una reseña de los pasos de este algoritmo.

Para el CSA global se implementó un arreglo de sufijos compacto como se describe en la sección 1.4.1. La BWT se representa con un árbol wavelet y E con una tabla plana. Nótese que un árbol wavelet binario tiene $2|\Sigma| - 1$ nodos. Para representar su topología explícitamente basta con $w(2|\Sigma|-1)$ bits (i.e. incluyendo la raíz pero sin considerar los punteros nulos de sus hojas), donde w es el tamaño de una palabra de máquina. Si el alfabeto es demasiado extenso respecto a la longitud de la secuencia que representa, el espacio ocupado por la topología puede ser muy relevante. Las representaciones sucintas de la topología de un árbol BP, LOUDS y DFUDS reducen el espacio requerido por esta a cambio de una navegación (a través del árbol) más lenta. Para aplicaciones con un alfabeto moderado (respecto del tamaño de la secuencia) el ahorro no es significativo y no justifica el deterioro en el rendimiento. La mayor parte de los recursos de un árbol wavelet se destina a sus bitmaps, para su representación se emplearon secuencias de RRR. Aun con una implementación cuidadosa, donde la tabla T descrita en la sección 1.3.1 se representa mediante una tabla universal (static) que es compartida por cada instancia de un bitmap RRR y que proporciona soporte para $select_{1/0}(\cdot)$ sin incurrir en un gasto adiconal de espacio, los bitmaps del wavelet consumen la mayor parte de recursos.

Para el bitmaps B también utilizaron bitmaps RRR. Nótese que si BC es la cadena que representa a la concatenación de los contenidos de los documentos, en el CSA global además de los valores muestreados uniformemente, también se deben mantener todas aquellas posiciones que corresponden al comienzo de un documento en BC. Formalmente, sea a_j la posición donde comienza el documento j dentro la cadena BC. Dada una frecuencia de muestreo b, los caracteres muestreados son todos los $BC[a_j + i \cdot b]$ tales que $a_j + i \cdot b < a_{j+1}$ para $i \geq 0$

(se asume que $a_i < a_j$ si $j > i$).

SADA y SGS emplean dos arreglos de sufijos invertidos compactos distintos. El de la primera estrategia se implementa de forma similar al CSA, con la diferencia de que en lugar de muestrear los valores de $SA(\cdot)$ guarda los valores de la inversa. Este se construye sobre secuencias locales individuales. El de SGS no cuenta con una tabla para $E(\cdot)$ ni con un árbol wavelet para la BWT; para ambos utiliza estructuras externas (globales y compartidas entre varios componentes) para calcular los valores de $SA^{-1}(\cdot)$.

Para FS, se implementaron dos variantes de los *CSA* y CSA^{-1}. En el primero se reemplazan el bitmap B y los valores muestreados por un arreglo de sufijos completo (la BWT todavía es necesaria para el $bwd\_search(\cdot)$). El CSA^{-1} es sustituido completamente por un arreglo de sufijos invertido (sin compresión).

Sadakane, al final del segundo paso en su solución, utiliza el algoritmo de Andersson et. al. [1] para ordenar los dos conjuntos de índices (en el rango $[sp..ep]$). Éste, para una entrada de n elementos, tiene una complejidad de tiempo de $\mathcal{O}(nr + n \log\log n)$ y de espacio de $\mathcal{O}(n + 2^{w/r})$, donde w es el tamaño de las palabras de máquina y $r \geq 1$ es un parámetro ajustable. Estimativamente, para esta aplicación, el único valor de r que podría ser útil es 2. Con 1 el espacio resulta excesivo y con un valor mayor que 2 el tiempo comienza a ser dominado por la componente nr. Esto porque los conjuntos de índices que son necesarios ordenar al final del paso 2 contienen a lo más *ndocs* elementos, por lo tanto, no son muy grandes. Aquí, simplemente, se optó en todas las soluciones por utilizar el algoritmo de ordenamiento de la librería estándar de C++.

4.3. Resultados Experimentales

En las figuras 4.1 a 4.5 se comparan las estrategias SADA, SGS y FS. Éstas fueron aplicadas a cada una de las colecciones descritas en 4.1 sobre rangos $[sp..ep]$ aleatorios del arreglo de sufijos global. Los tiempos miden el tiempo que tarda en listar cada uno de los documentos y sus respectivas frecuencias de los patrones que ocurren en el rango especificado (esto no incluye el tiempo de buscar el patrón). El espacio se mide en bpc (bits por símbolo). Se verifica que, para las colecciones estudiadas, aplicando la misma tasa de muestreo en las estructuras CSA y CSA^{-1} de ambos métodos: SGS y SADA, los tiempos de respuesta de este último corresponden a alrededor de un 62 % los del primero. Esto es, el tiempo que SGS tarda en resolver una consulta es cercano a 1,6 veces el que le toma a SADA (sin incluir la búsqueda del patrón). Ambas estrategias realizan varias aplicaciones de $LF(\cdot)$ para responder una consulta. La diferencia entre los rendimientos de ellas se debe que la lógica de $LF(\cdot)$ es mucho más compleja en SGS; requiere de dos llamadas a $E(\cdot)$ y una a $rank(\cdot)$ adicionales. Además, éstas son sobre árboles wavelet significativamente más grandes.

En cuanto al espacio ocupado por ambas técnicas, el de SGS es, para todos los casos estudiados, menor que el de SADA. Para ClueWiki el ahorro en espacio que se obtiene es de un 14 %, lo cual considerando la degradación en su rendimiento no supone una mejoría. Diferente es la situación con las otras colecciones. Para DNA200 (figura 4.4) SGS consume

un 45 % del espacio que ocupa SADA; para las otras tres (figuras 4.2, 4.3 y 4.5) varía entre un 20 % y 28 %.

La razón de la mejoría del espacio de SGS sobre SADA se debe a que al aplicar SADA sobre colecciones abundantes en documentos cortos no resulta conveniente, incluso si se dispone de suficiente espacio para ello.

Como se explicó en la sección 3.1, por cada documento SADA mantiene un CSA_ℓ que contiene una muestra de los valores del arreglo de sufijos invertido, un árbol wavelet compacto y una tabla de acumulación (además de otros datos marginales). Las últimas dos componentes le proporcionan la información necesaria para soportar la operación $LF(\cdot)$.

En cuanto a los árboles wavelet, la memoria se distribuye en sus bitmaps, su topología y los mapas de los caracteres de la secuencia que representa.

Para secuencias cortas los árboles wavelet no consiguen mantener un tamaño competitivo. En la tabla 4.2 se observa que, con la excepción de DNA200, que tiene un alfabeto muy pequeño, los árboles wavelet construidos sobre secuencias cortas consumen varias veces el espacio de la secuencia original. Para english200 y KGS es, incluso, suficiente para almacenar la secuencia y su arreglo de sufijos sin compresión (con palabras de 32 bits). Nótese que a pesar de la entropía media de los documentos de ClueWiki es la mayor de todas colecciones, los árboles wavelet construidos sobre estos comprimen mejor debido que la longitud media de sus documentos es mucho mayor respecto de las otras.

	ClueWiki	KGS	DNA200	English200	Proteins60
wt. bpc	5,73	48,79	8,68	47,95	33,06
doc. len.	41,278	1,398	1,024	1,024	1,024

Tabla 4.2: En esta tabla se muestran el espacio promedio en bpc que ocupan los árboles wavelet sobre la BWT de los documentos de cada colección y el largo medio de éstos.

En las colecciones estudiadas si se emplea en SGS una tasa de muestreo alrededor un 2 % sobre la de SADA es posible alcanzar el mismo rendimiento pero en un espacio significativamente menor. En particular, se puede obtener el mismo desempeño con sólo un tercio del espacio que ocupa SADA.

En algunos casos el espacio que SADA llega a demandar es tan elevado que si se dispone de una cantidad marginal de memoria por sobre éste, es posible y conveniente sustituir SADA por FS. Esto es justamente lo que ocurre para KGS y English200 (figuras 4.2 y 4.3), donde FS requiere sólo un 3 % y 1 % de memoria adicional, respectivamente, en relación con de la alternativa SADA con un muestreo de 10 %, y su rendimiento es de 17 veces superior, para el primer caso, y hasta 22 veces para el segundo.

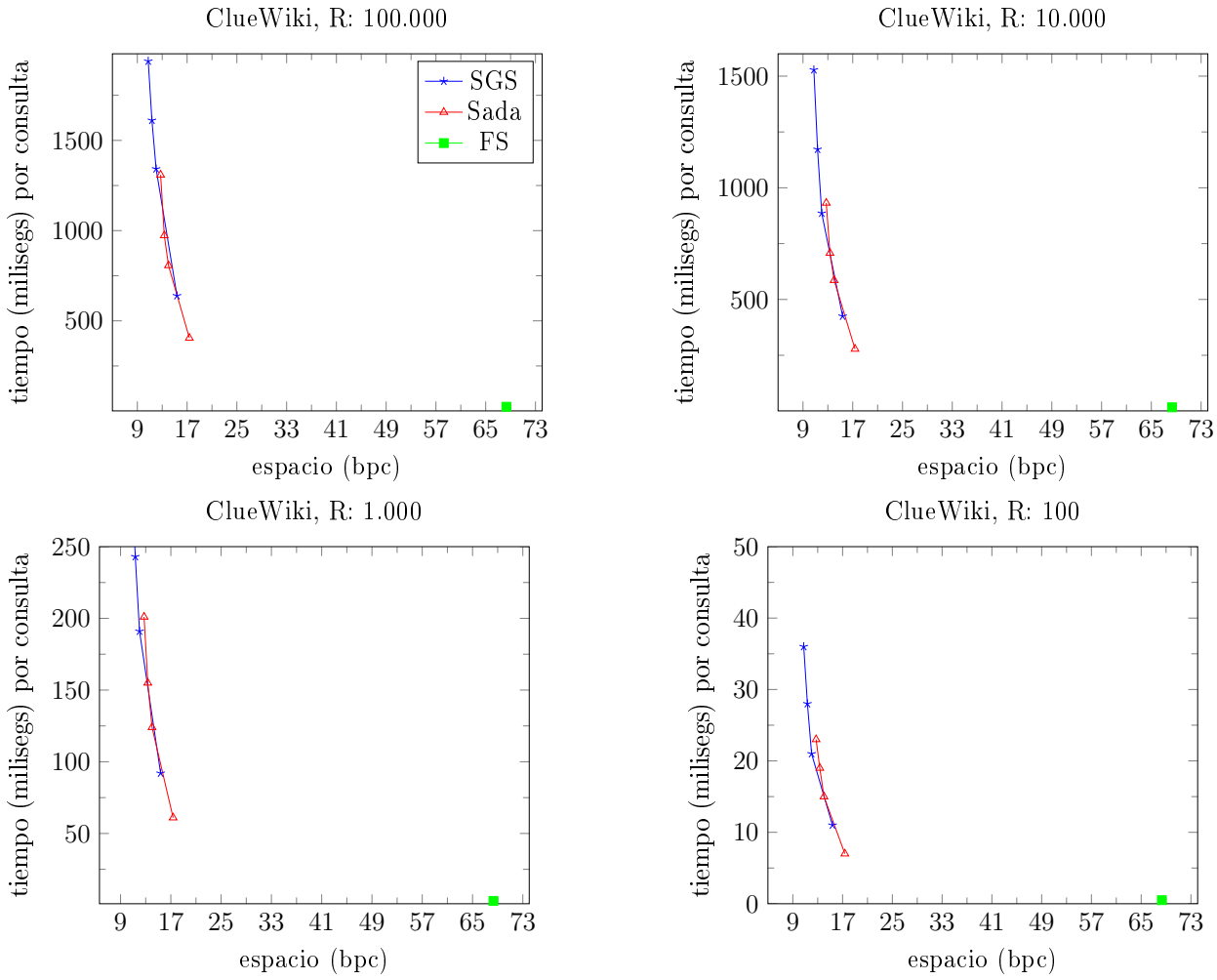

Figura 4.1: Aplicación de las estrategias sobre la colección ClueWiki para rangos $[sp..ep]$ de $100, 1{,}000, 10{,}000$ y $100{,}000$.

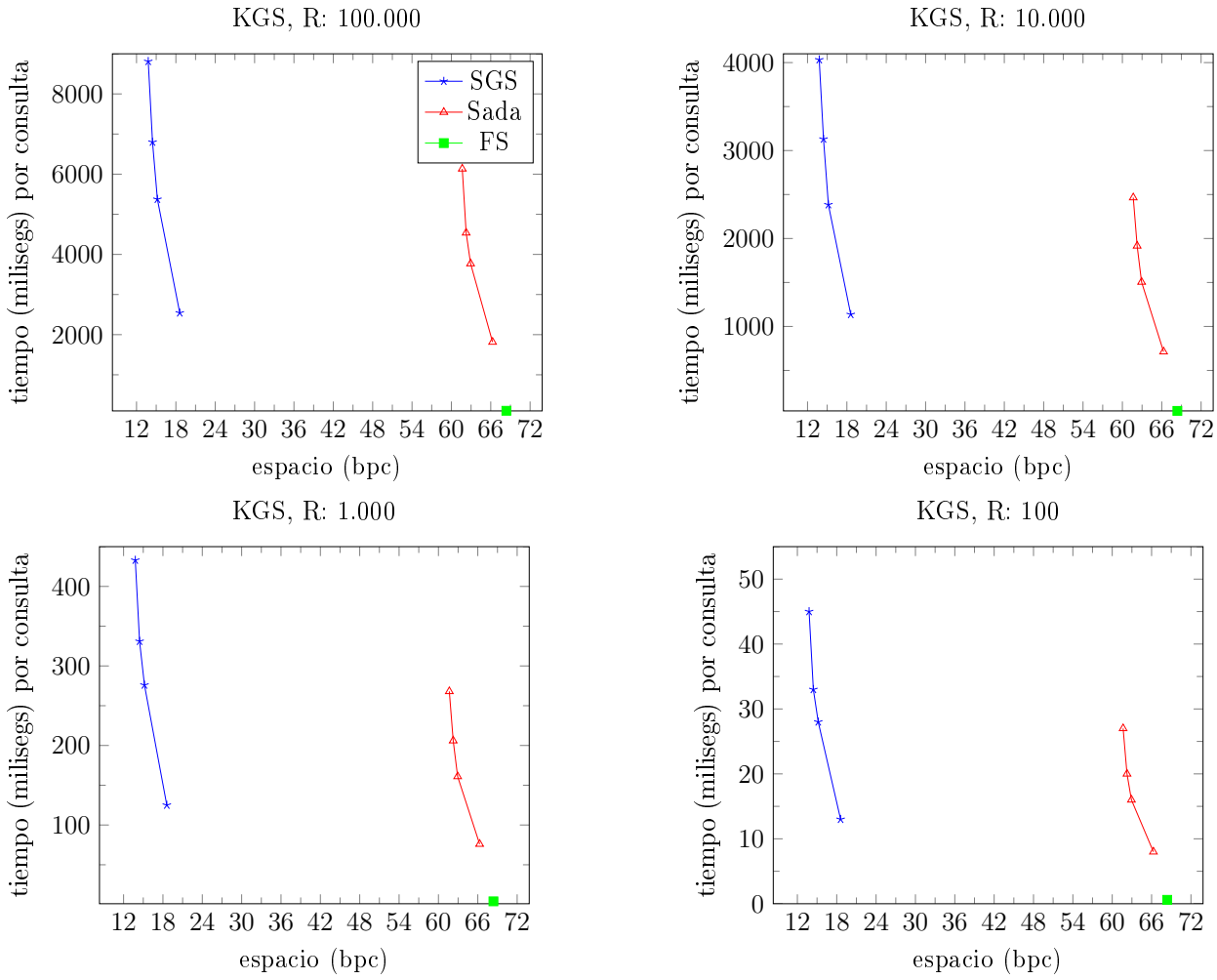

Figura 4.2: Aplicación de las estrategias sobre la colección KGS para rangos $[sp..ep]$ de $100, 1{,}000, 10{,}000$ y $100{,}000$.

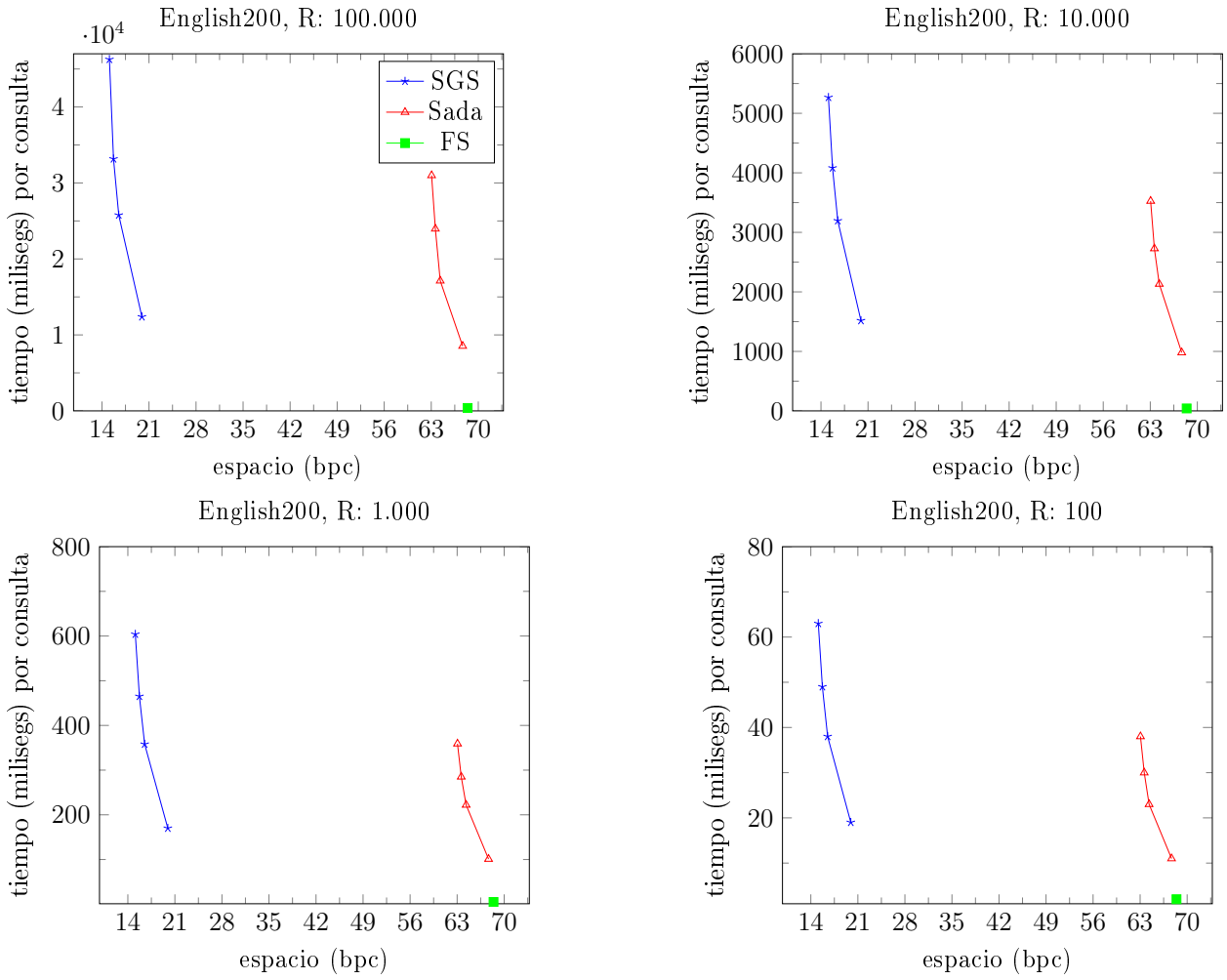

Figura 4.3: Aplicación de las estrategias sobre la colección English200 para rangos [*sp*..*ep*] de 100, 1,000, 10,000 y 100,000.

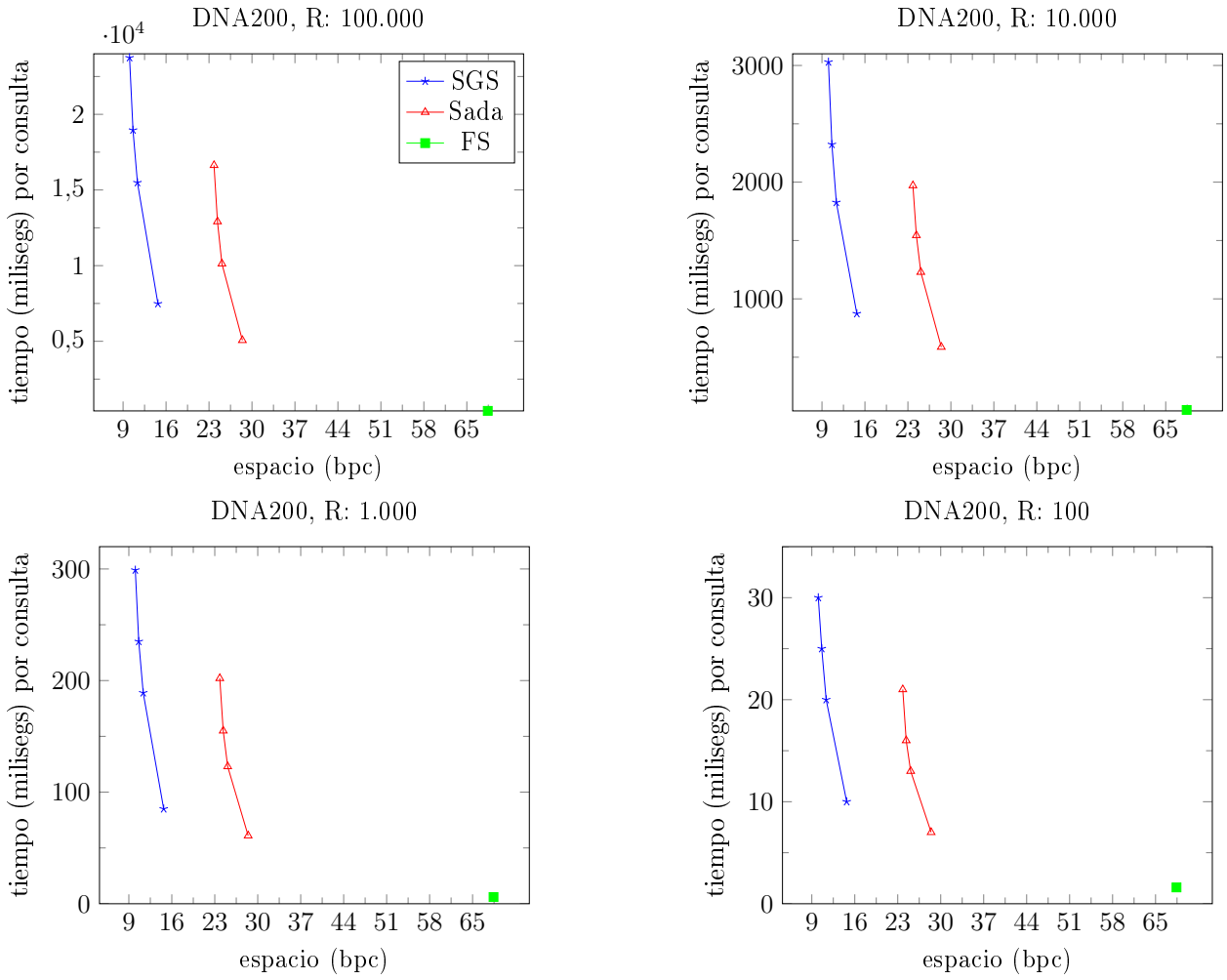

Figura 4.4: Aplicación de las estrategias sobre la colección DNA200 para rangos [*sp*..*ep*] de 100, 1,000, 10,000 y 100,000

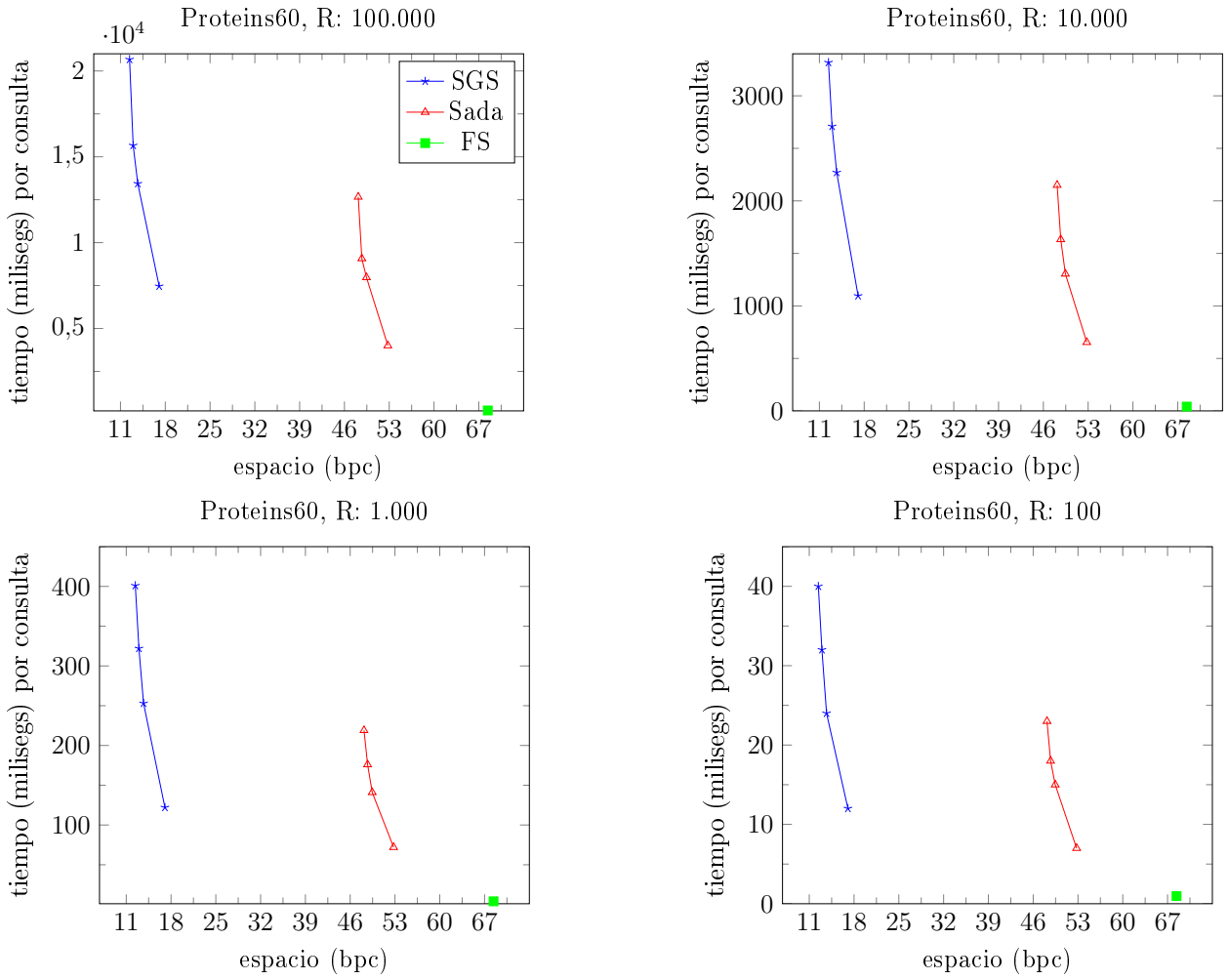

Figura 4.5: Aplicación de las estrategias sobre la colección Proteins60 para rangos [*sp*..*ep*] de 100, 1,000, 10,000 y 100,000.

Conclusión

En este trabajo se abordó el problema de listado de documentos con sus frecuencias. Este consiste en, dada una colección de documentos y un patrón de consulta, encontrar todos los documentos donde el patrón ocurre y las frecuencias con la que aparece en cada uno de estos documentos. Se comprobó que la nueva estrategia para colecciones de documentos cortos es competitiva. En este escenario consume un espacio significativamente menor que la solución original de Sadakane. Aunque resulta algo más lenta, cuando sus respectivos *CSA* y CSA^{-1} se construyen con la misma tasa de muestreo, con un muestreo adicional puede alcanzar el mismo rendimiento que Sadakane y gastando en total sólo un tercio del espacio de ésta. Las pruebas sobre ClueWiki sugieren que SGS no ofrece una mejoría para colecciones con documentos medianos o largos. En general, se espera que para estos casos no se produzca una reducción suficientemente importante en el espacio como para justificar la pérdida de eficiencia.

Uno de los aspectos que es posible mejorar en el algoritmo de Sadakane es la última etapa del paso 2, donde dos conjuntos de índices con la misma cardinalidad, a lo más *ndocs*, deben ser ordenados. A partir de cierto tamaño para dichos conjuntos, paralelizar los ordenamientos podría introducir una reducción en los tiempos de respuesta.

Otro avance sería encontrar un algoritmo de ordenamiento que explote las características de estos conjuntos: ambos se componen de números enteros positivos sin repetición. Una alternativa es evaluar el algoritmo de Andersson et.al. [1] que sugiere Sadakane.

La solución SGS mantiene dos árboles wavelet: WT_CBWT para el arreglo de CISA's y otro para el CSA global. Ambos se construyen sobre secuencias que son permutaciones del texto de entrada (una cadena larga que representa la concatenación de los contenidos de los documentos de una colección). Un enfoque que podría ayudar a reducir el espacio sería encontrar una estructura que permita eliminar la información redundante en estos árboles pero que al mismo tiempo conserve la funcionalidad que éstos ofrecen. Para el CSA esta consiste en brindar el soporte para encontrar patrones en el texto y calcular los valores de su arreglo de sufijos, mientras que para el segundo consiste sólo en determinar los valores de su arreglo de sufijos invertido. El espacio del primer wavelet se mueve entre $2,7$ y 4 bpc y el del segundo es al menos una vez el de su compañero. Se estima que una estructura como la descrita podría conceder un ahorro del orden de unos 3 bpc.

Bibliografía

[1] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? In *Prococeedings 27th Annual ACM Symposium on Theory of Computing (STOC)*, pages 427–436, 1995.

[2] D. Benoit, E. Demaine, I. Munro, R. Raman, V. Raman, and S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

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

[4] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings 41st IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 390–, 2000.

[5] J. Fischer and V. Heun. A new succinct representation of rmq-information and improvements in the enhanced suffix array. In *Prococeedings 1st International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies (ESCAPE)*, pages 459–470, 2007.

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

[7] G. Jacobson. Space-efficient static trees and graphs. In *Proceedings 30th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

[8] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2-4):143–156, 2005.

[9] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. In *First Annual ACM-SIAM Symposium on Discrete Algorithms (DA)*, pages 319–327, 1990.

[10] Giovanni Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.

[11] I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graph. In *38th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 118–126, 1997.

[12] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proceedings 13th Annua ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 657–666, 2002.

[13] G. Navarro, S. Puglisi, and D. Valenzuela. Practical compressed document retrieval. In *Prococeedings 10th International Symposium on Experimental Algorithms (SEA)*, LNCS 6630, pages 193–205, 2011.

[14] G. Navarro, S. J. Puglisi, and D. Valenzuela. General document retrieval in compact space. *ACM Journal of Experimental Algorithmics*, 2013. Por publicarse.

[15] G. Nong, S. Zhang, and W. H. Chan. Linear suffix array construction by almost pure induced-sorting. In *Proceedings 19th Data Compression Conference (DCC)*, pages 193–202, 2009.

[16] K. Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms*, 5(1):12–22, 2007.

[17] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, pages 379–423, 623–656, 1948.

[18] P. Weiner. Linear pattern matching algorithms. In *Proceedings 14th Annual IEEE Symposium on Switching and Automata Theory (SWAT)*, pages 1–11, 1973.

Apéndice A

Anexo

A.1. SAIS (Suffix Array Induced Sorting Algorithm)

SAIS es un algoritmo propuesto por Ge Nong et al. [15] para la construcción de arreglos de sufijos en tiempo lineal y eficiente en la práctica, con una complejidad de espacio de $5{,}13n$ bytes ($4n$ son necesarios para un arreglo de sufijos de 32 bits). Su implementación es relativamente sencilla en cuanto ésta no requiere de estructuras sofisticadas (a diferencia del algoritmo de Ko y Aluru [8] del cual SAIS deriva).
Este algoritmo permite resolver el problema de construcción sin imponer estrictas restricciones sobre el volumen de la entrada. Nótese que el espacio que requiere es el consumido por los datos de entrada (A) y salida (SA) más $|A|$ bits.

Definiciones

Antes de comenzar a describir el algoritmo *SAIS* es necesario dar algunas definiciones. Un sufijo A_i se dice que es *tipo* S o *tipo* L si $A_i < A_{i+1}$ o $A_i > A_{i+1}$, respectivamente. El último sufijo A_n que consiste únicamente del centinela, se define de tipo S. Un carácter $A[i]$ es del mismo tipo que el sufijo A_i. Un carácter $A[i]$ se dice que es LMS si $A[i]$ es tipo S y $A[i-1]$ es tipo L. Una *subcadena LMS* es o una subcadena $A[i..j]$ tal que $A[i]$ y $A[j]$ son caracteres LMS y no tiene más caracteres LMS o la subcadena compuesta sólo por el centinela. Todos los sufijos que comienzan con un mismo carácter ocupan un *bloque* contiguo dentro del arreglo de sufijos y estos bloques se subdividen en dos partes: en un *bloque tipo* L y un *bloque tipo* S, que contienen sufijos tipo L y tipo S, respectivamente.

Descripción de *SAIS*

En base a las definiciones previas, a continuación se entrega una descripción de alto nivel del algoritmo *SAIS*. Este recibe una secuencia $A = a_0..a_{n-1}\$$ y calcula su arreglo de sufijos SA.

(1) Se almacena en un bitmap el tipo (S o L) de todos los caracteres de A.

(2) Se buscan todas las subcadenas LMS y se guarda una referencia a sus posiciones. Esto

es posible hacerlo sin consumir espacio adicional. El mismo espacio destinado a contener el arreglo de sufijos se puede emplear para este propósito en esta etapa.

(3) Se induce el orden entre todas las subcadenas LMS a partir de las referencias y el bitmap de tipos de la siguiente manera:
   (i) se buscan los finales (extremos derechos) de cada bloque tipo S en SA y se colocan todos los sufijos LMS en sus correspondientes bloques tipo S.
   (ii) Se buscan las cabezas (extremos izquierdos) de cada bloque tipo L en SA. Se recorre SA en orden creciente y para cada $A[SA[i]-1]$ de tipo L que se encuentre, se coloca $SA[i]-1$ en la cabeza actual del bloque tipo L y se mueve dicha cabeza una posición hacia adelante.
   (iii) Se recorre SA en orden decreciente y para cada $A[SA[i]-1]$ de tipo S que se encuentre, se coloca $SA[i]-1$ en el final actual del bloque tipo S y se mueve dicho final en una posición hacia la izquierda.

(4) Se les asigna un nombre a las subcadenas LMS en A y se construye la cadena $S^{(1)}$, donde el i-ésimo elemento corresponde al nombre de la i-ésima subcadena LMS en A. La longitud de esta cadena ($S^{(1)}$) es a lo más la mitad de $|A|$.

(5) Si en $S^{(1)}$ no hay repeticiones entonces se calcula el arreglo de sufijos $SA^{(1)}$ de $S^{(1)}$ directamente; en caso contrario se aplica recursivamente el algoritmo sobre la cadena de nombres $S^{(1)}$.

(6) Se induce el arreglo de sufijos SA a partir de $SA^{(1)}$ de la siguiente forma: se buscan los finales de cada bloque tipo S en SA y se colocan todos los elementos de $SA^{(1)}$ en sus correspondientes bloques tipo S en SA, conservando el orden relativo dentro de $SA^{(1)}$, luego se aplica (ii) y (iii).

Como se dijo al principio, esta presentación de *SAIS* no tiene como objetivo dar una explicación detallada sobre cómo el algoritmo alcanza las complejidades de tiempo y espacio mencionadas, ni enunciar los teoremas sobre los cuales descansa y tampoco desarrollar una demostración sobre la correctitud de la solución. Por lo mismo, a continuación, en términos muy generales, se decribe la idea que hay detrás para el manejo de memoria. Además de reservar espacio para la entrada A, el bitmap de los tipos y el arreglo de sufijos SA, *SAIS* también requiere espacio para almacenar: las referencias a las posiciones de las subcadenas LMS en A, los nombres asignados a cada subcadena LMS y, en cada recursión, la instancia reducida del problema original, la cadena $S^{(1)}$, que es una representación compacta de una secuencia de las subcadenas LMS que aparecen en A. Para no consumir espacio adicional, la estrategia consiste en utilizar el espacio destinado al arreglo de sufijos SA. Esto resulta posible, porque el algoritmo en algunas etapas una vez que lee ciertos datos puede luego descartarlos y además porque dentro de la recursión resuelve parcialmente el problema. Las implicancias de esto es que la memoria reservada para SA durante la aplicación de *SAIS* cuenta con suficientes espacios disponibles para guardar estos datos temporales.