



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

ESTRUCTURAS DE DATOS COMPACTAS PARA  
RECUPERACIÓN DE DOCUMENTOS

PROPUESTA DE TESIS PARA OPTAR AL GRADO DE MAGÍSTER  
EN CIENCIAS, MENCIÓN COMPUTACIÓN

DANIEL ALEJANDRO VALENZUELA SERRA

PROFESOR GUÍA:  
GONZALO NAVARRO BADINO

.....  
Daniel Alejandro Valenzuela Serra  
Alumno

.....  
Gonzalo Navarro Badino  
Prof. Guía

---

Este trabajo ha recibido financiamiento del Instituto Milenio de  
Dinámica Celular y Biotecnología (ICDB), proyecto P05-001-F,  
Mideplan

---

SANTIAGO DE CHILE  
DICIEMBRE 2009

# Índice

<b>1. Introducción</b>	<b>1</b>
<b>2. Marco Teórico</b>	<b>3</b>
2.1. Rank y Select . . . . .	3
2.2. RePair . . . . .	3
2.3. Árbol de Sufijos . . . . .	4
2.4. Arreglo de Sufijos . . . . .	4
2.5. Regularidades en el Arreglo de Sufijos y la función $\Psi$ . . . . .	5
2.6. La Transformada de Burrows-Wheeler (BWT) . . . . .	6
2.7. Wavelet Tree . . . . .	6
2.8. Arreglo de Sufijos Comprimido . . . . .	7
<b>3. Trabajo Relacionado</b>	<b>8</b>
3.1. <i>Document Retrieval</i> . . . . .	8
3.2. La solución de Muthukrishnan . . . . .	9
3.3. La solución de Hon et al. . . . .	10
3.4. Soluciones comprimidas . . . . .	10
3.4.1. Solución de Sadakane . . . . .	10
3.4.2. Solución de Hon et al. . . . .	10
3.4.3. Solución de Mäkinen y Välimäki . . . . .	11
3.4.4. Solución de Gagie et al. . . . .	11
3.4.5. Discusión . . . . .	11
<b>4. Objetivos</b>	<b>12</b>
4.1. Objetivo General . . . . .	12
4.2. Objetivos Específicos . . . . .	12
<b>5. Metodología</b>	<b>13</b>
<b>Referencias</b>	<b>15</b>

# 1. Introducción

El volumen de la información almacenada en formato digital crece de manera vertiginosa. Desde el genoma humano hasta detallados mapas de todo el planeta, 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 digitalmente 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.

Pero probablemente uno de los elementos que hace más atractivo el almacenamiento digital de la información, es el hecho de poder utilizar el poder de cálculo de los computadores para buscar la información relevante en esta inmensidad de datos.

Sin embargo, el poder de cálculo por sí solo no basta para dar acceso y buscar la información de manera rápida y eficiente. 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 (incluso utilizando los mejores algoritmos conocidos para esto), ésta tardaría días en entregar los resultados.

La solución a este problema está dada por 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 esencialmente almacenan una lista con las páginas en las que ocurre cada palabra. Así, cuando uno busca una frase, el resultado es la intersección de las listas asociadas a cada palabra.

Una de las limitantes más fuertes de los índices invertidos es que se requiere definir *a priori* las palabras que se indexarán, y por consiguiente no permite la búsqueda de patrones arbitrarios. Lo anterior restringe el uso de los índices invertidos a algunos idiomas occidentales, como el inglés, en que resulta relativamente directo definir las palabras relevantes, no así en idiomas orientales como el chino o el coreano. Menos factible resulta esta estrategia para búsquedas en otras bases de datos, como el genoma humano, en las que lo que se buscan no son palabras ni frases.

El problema de búsqueda clásico, es decir, encontrar las ocurrencias de un patrón arbitrario en una secuencia, fue solucionado de manera óptima (ocupando espacio lineal en el tamaño de la secuencia) por Weiner mediante un índice llamado “Árbol de Sufijos”.

Sin embargo, un problema bastante más realista es el problema de “Recuperación

de Documentos”. En este caso, ya no se trata de indexar una sola secuencia, sino una colección de secuencias (o documentos). La consulta es también un patrón arbitrario, y la respuesta consiste en entregar los documentos más relevantes a la consulta (por ejemplo, aquellos en que el patrón ocurra más veces). Si bien este problema es frecuente en la práctica (cuando buscamos una palabra en internet, el resultado no son las ocurrencias de la palabra, sino que las páginas en que esta palabra aparece), el primer resultado para texto general fue obtenido por Muthukrishnan [Mut02]. Utilizando una variante del Árbol de Sufijos, junto a una estructura adicional llamada “Arreglo de Documentos”, Muthukrishnan soluciona el problema de manera óptima, y resuelve algunas variaciones de interés práctico del problema planteado de manera eficiente.

Una de las variaciones más interesantes en la práctica es el problema llamado *top-k Document Retrieval*, es decir, dado un patrón, encontrar los  $k$  documentos más relevantes (por ejemplo, los  $k$  documentos en que el patrón ocurre con frecuencia más alta). Este problema ha sido recientemente resuelto de manera óptima utilizando espacio lineal.

Sin embargo, los índices clásicos pueden requerir demasiado espacio. Por ejemplo, el Árbol de Sufijos requiere  $O(n \log n)$  bits adicionales a la secuencia original. Lo anterior ha motivado un desarrollo dirigido a reducir el espacio requerido por los índices, obteniéndose interesantes resultados: es posible construir índices que no requieren de la secuencia original, que permiten dar acceso a posiciones arbitrarias en ella, y requieren menos espacio que la secuencia original. Estas estructuras son frecuentemente llamadas autoíndices, pues reemplazan a la secuencia original y pueden por lo tanto ser consideradas como compresión con valor agregado.

Recientemente ha emergido una nueva línea de investigación, que consiste en utilizar los resultados conocidos en el campo de los autoíndices para resolver los problemas de Recuperación de Documentos. Se han estudiado diferentes estrategias para adaptar las estructuras a estos problemas [HSW09, VM07, GPT09]. Un problema común en casi todas ellas es que no se logra comprimir el Arreglo de Documentos, el cual también requiere  $O(n \log n)$  bits.

En este trabajo se estudiarán nuevas maneras de comprimir el Arreglo de Documentos, desde un punto de vista teórico y práctico permitiendo reducir el espacio requerido y los tiempos de acceso de las soluciones comprimidas existentes actualmente para los problemas de *Document Retrieval*. En particular, se estudiará la utilidad de *RePair* [LM00] para este problema. Además, se espera que los resultados puedan ser de utilidad en otros escenarios, dada la relación existente entre el Arreglo de Documentos y el Arreglo de Sufijos.

## 2. Marco Teórico

### 2.1. 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 la función  $rank(B, i) = rank_1(B, i)$ , la cual entrega el número de unos en 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 [Mun96, Cla98] utilizan  $n + o(n)$  bits:  $n$  bits del arreglo  $B$  y  $o(n)$  bits adicionales para responder las consultas de  $rank$  y  $select$ . Posteriormente, Raman, Raman y Rao (RRR) [RRR02] propusieron estructuras para  $rank$  y  $select$  que permiten comprimir  $B$  de modo que ocupe  $nH_0(B) + o(n)$  bits, donde  $H_0$  es la entropía de orden cero de  $B$ :  $H_0(B) = (n_0/n) \log(n/n_0) + (n_1/n) \log(n/n_1)$ , donde  $n_0$  es el número de ceros en la secuencia, y  $n_1$  es el número de unos.

Asimismo, existen resultados experimentales [GGMN05, OS07, CG08] 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  $\Sigma$  de tamaño  $\sigma$ . 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 generalizarse 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 [GGV03], el cual requiere  $nH_0(T) + o(n \log \sigma)$  bits, y permite responder  $rank$  y  $select$  en tiempo  $O(\log \sigma)$ , donde  $H_0(T)$  es la entropía de orden cero de  $T$ :

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

siendo  $n_c$  el número de ocurrencias del carácter  $c$  en  $T$ .

### 2.2. RePair

RePair es un método de compresión *off line* basado en un diccionario, que en la práctica alcanza tasas de compresión competitivas [LM00] y permite descompresión local utilizando exclusivamente el diccionario. Se ha mostrado [NR08] que RePair

requiere espacio proporcional a la entropía de orden  $k$  de la secuencia, con  $k = o(\log_{\sigma} n)$ .

La idea de RePair consiste en buscar el par de caracteres más frecuente en la secuencia a comprimir y reemplazarlo por nuevo símbolo. Este proceso se repite hasta que todos los pares de símbolos ocurran una sola vez.

Si consideramos una secuencia  $T$  sobre un alfabeto de tamaño  $\sigma$ , el algoritmo se puede sintetizar en los siguientes pasos:

1. Identificar los símbolos  $a$  y  $b$  tal que el par  $ab$  es el par más frecuente en la secuencia. Si ningún par aparece más de una vez, detenerse.
2. Crear un nuevo símbolo  $A$  y reemplazar todas las ocurrencias de  $ab$  por  $A$ . Agregar la regla  $R(A) \rightarrow ab$  al diccionario.
3. Repetir desde el paso 1.

El resultado del algoritmo es la secuencia comprimida  $C$  (con símbolos originales y nuevos) y el diccionario  $R$ .

Para descomprimir  $C[j]$  evaluamos: si  $C[j] \leq \sigma$ , se trata de un símbolo original, en cuyo caso retornamos. En caso contrario, expandimos la regla correspondiente  $R(C[j]) \rightarrow ab$  y realizamos el mismo proceso con  $a$  y  $b$  recursivamente. De este modo, reproducimos una subsecuencia de tamaño  $l$  en  $T$  en tiempo  $O(l)$ .

### 2.3. Árbol de Sufijos

El Árbol de sufijos [Wei73] de una secuencia  $T$  es un trie comprimido en el que se insertan todos los sufijos de  $T$ , y cuyas hojas representan la posición en la secuencia del sufijo correspondiente al camino desde la raíz a dicha hoja. Se ha mostrado que esta estructura requiere espacio lineal ( $O(n \log n)$  bits) y que puede ser construida en tiempo  $O(n)$ . Esta estructura permite encontrar las ocurrencias de un patrón  $P$  en  $T$  en tiempo  $O(|P| + occ)$ . La idea consiste en bajar por los nodos del trie, comenzando por la raíz, siguiendo las aristas correspondientes a los caracteres de  $P$ . Si en algún momento no podemos bajar por el trie con un carácter de  $P$ , significa que no hay ocurrencias de  $P$  en  $T$ . Si luego de leer  $P$  finalizamos en un nodo, entonces las ocurrencias de  $P$  corresponden a las hojas del subárbol asociado a dicho nodo. En la figura 1 se muestra el Árbol de Sufijos para la secuencia  $T = \text{"alabar\_a\_la\_alabarda\$"}.$

### 2.4. 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

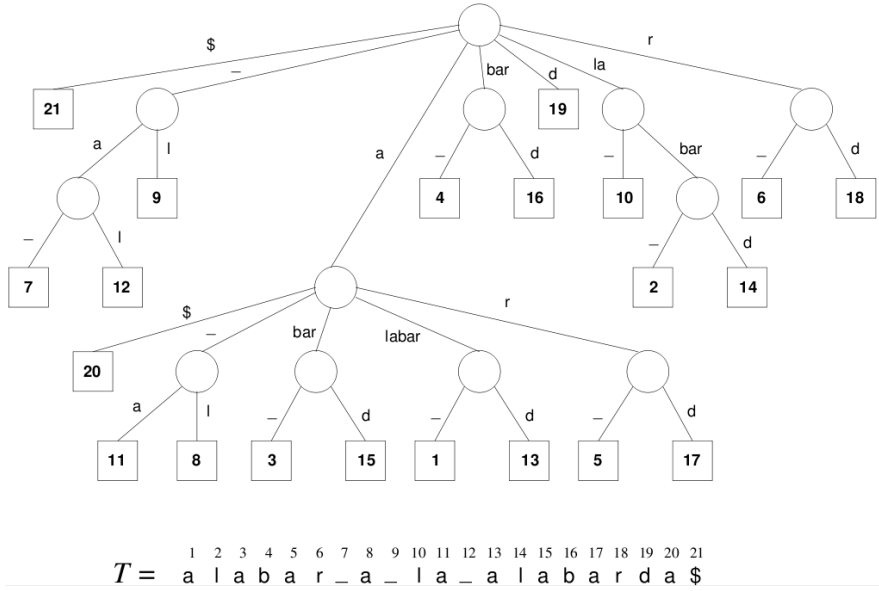


Figura 1: Árbol de Sufijos para la secuencia  $T = \text{“alabar\_a\_la\_alabarda$”}$ .

[MM93]. Supongamos que  $T$  termina con un marcador único  $\$$ , de tal modo que las comparaciones lexicográficas están bien definidas.  $A[i]$  apunta al sufijo  $T[A[i], n] = t_{A[i]}t_{A[i]+1} \dots t_n$ , y se tiene que lexicográficamente  $T[A[i], n] < T[A[i + 1], n]$ . Dados  $A$  y  $T$ , las ocurrencias de un patrón  $P = p_1p_2 \dots p_m$  en  $T$  se pueden contar en tiempo  $O(m \log n)$ . Las ocurrencias del patrón corresponden a un intervalo  $A[sp, ep]$  tal que todos los sufijos  $t_{A[i]}t_{A[i]+1} \dots t_n$  para todo  $sp \leq i \leq ep$  contienen al patrón  $P$  como prefijo. Para encontrar el intervalo basta con realizar dos búsquedas binarias. Una vez que se obtiene el intervalo, las ocurrencias se localizan reportando todos los punteros del intervalo, cada uno de ellos en tiempo constante. En la figura 2 se muestra el Arreglo de Sufijos de la secuencia “alabar\_a\_la\_alabarda” y se ejemplifica el proceso de búsqueda.

## 2.5. Regularidades en el Arreglo de Sufijos y la función $\Psi$

Consideremos la función  $\Psi$ , definida de tal modo que si  $\Psi[i] = i'$  entonces  $A[i'] = A[i] + 1$  si  $A[i] < n$  y  $A[i'] = 1$  si  $A[i] = n$ .

Observemos en el Árbol de Sufijos de la figura 1 que el subárbol asociado al sufijo “abar” contiene las hojas 3 y 15, mientras que el subárbol asociado al sufijo “bar” contiene las hojas 4 y 16, es decir, las mismas posiciones desplazadas en uno. Esto se debe a que cada ocurrencia de “bar” en  $T$  es también una ocurrencia de “abar”. Se ha mostrado que estas repeticiones en el Arreglo de Sufijos corresponden a *runs*

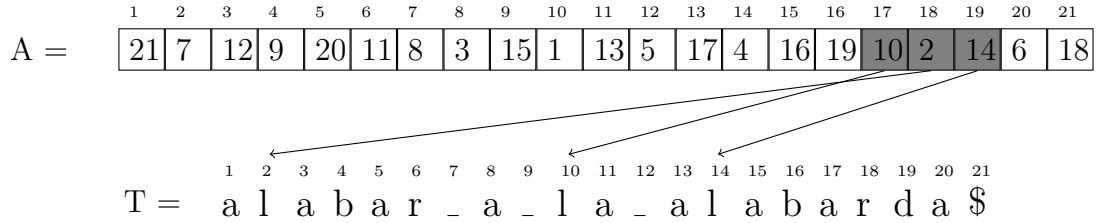


Figura 2: Arreglo de sufijos del texto “alabar\_alabarda”. El intervalo oscurecido en el arreglo de sufijos corresponde a los sufijos que comienzan con la secuencia “la”.

en la función  $\Psi$  (intervalos en los que  $\Psi(i + 1) - \Psi(i) = 1$ ), los cuales pueden ser explotados para comprimir el Arreglo de Sufijos, directamente [GN07] o a través de la función  $\Psi$  [Sad03].

## 2.6. La Transformada de Burrows-Wheeler (BWT)

La transformada de Burrows-Wheeler [BW94] 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  $A$ , 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 todos los strings de la forma  $T_{i,n}T_{1,i-1}$  ordenados lexicográficamente. Si se considera una matriz con todos estos strings, 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  $L_i$ . Sea  $C(c)$  el número de ocurrencias de caracteres menores que  $c$  en  $T$ , entonces  $LF(i) = C(L_i) + rank_c(L, i)$  con  $c = L_i$ . Se puede recuperar  $T$  a partir de  $T^{bwt}$  mediante  $T[n] = \$$  y  $s = 1$  y luego para cada  $k = n - 1 \dots 1$ ,  $s \leftarrow LF(s)$ ,  $T[k] \leftarrow T^{bwt}[s]$ .

## 2.7. Wavelet Tree

Consideremos una secuencia de caracteres  $T = a_1a_2\dots a_n$ , 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

$T = a_1a_2 \dots a_n$ . Su hijo izquierdo está asociado con la subsecuencia de  $T$  obtenida al concatenar todas las posiciones  $i$  que satisfagan  $a_i < \sigma/2$  (donde  $\sigma$  es el tamaño del alfabeto  $\Sigma$ ) 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. En la figura 3 se ejemplifica este proceso.

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.

## 2.8. Arreglo de Sufijos Comprimido

Un Arreglo de Sufijos Comprimido (CSA) es un autoíndice que reemplaza tanto al texto como al arreglo de sufijos basándose en una representación comprimida de éste último. El CSA debe brindar acceso a cualquier carácter del texto, así como responder las consultas  $count(P)$  y  $locate(P)$  para un patrón  $P$  arbitrario, las cuales consisten en determinar el número de ocurrencias de  $P$  en el texto, y encontrar la posición de cada una de ellas, respectivamente. El FM-Index es un autoíndice que consta esencialmente de tres partes: (1) La transformada de Burrows-Wheeler (BWT) del texto  $T$ , representado con un Wavelet Tree que

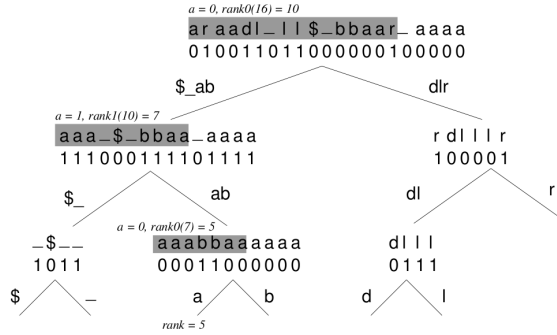


Figura 3: Wavelet Tree para la secuencia  $T = \text{“araadl\_ll\$bbaar\_aaaa”}$ , que es la transformada de Burrows-Wheeler de  $\text{“alabar\_a\_la\_alabarda\$”}$ . Se ilustra el proceso para calcular  $rank_a(A, 15)$

permite calcular  $rank_c(A, i)$ . (2) El arreglo  $C$  descrito en la sección 2.6 (3) Algunas estructuras sublineales para acceder a muestreos regulares de valores.

Se ha mostrado que esta estructura ocupa espacio cercano a la entropía de orden  $k$  de  $T$  [MN07]:  $nH_k(T) + o(n \log \sigma)$  permitiendo calcular el número de ocurrencias  $n_{occ} = count(P)$  en tiempo  $O(m \log \sigma)$ , y  $Locate(P)$  en tiempo  $O(n_{occ} \log^{1+\epsilon} n)$  para una constante  $\epsilon > 0$  que permite un compromiso entre los tiempos y el espacio requerido por la componente sublineal.

### 3. Trabajo Relacionado

#### 3.1. Document Retrieval

Consideremos una colección  $D = \{d_1, d_2, \dots, d_N\}$  de documentos, de largo total  $n$ , y un patrón  $P$ . Podemos definir las siguientes consultas:

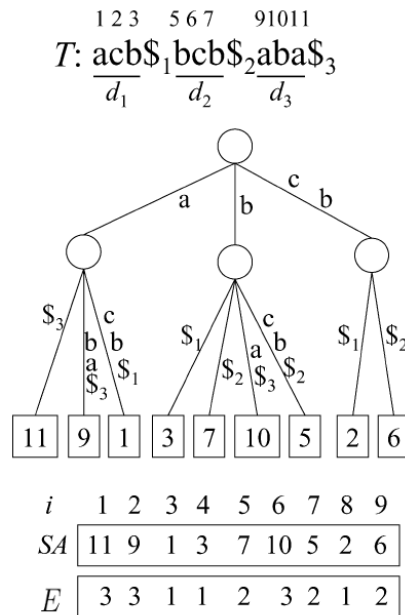
- $Document\ Listing(P)$  : Retornar los  $ndoc$  documentos en que  $P$  aparece.
- $Document\ Mining(P, K)$  : Retornar los  $ndoc$  documentos en que  $P$  aparece al menos  $K$  veces.
- $Top(P, k)$ : Retornar los  $k$  documentos más relevantes para la consulta definida por el patrón  $P$ .

### 3.2. La solución de Muthukrishnan

Las estructuras planteadas por Muthukrishnan [Mut02] permiten resolver de manera óptima (en espacio lineal) el problema de *Document Listing(P)* y permiten mediante pequeñas modificaciones resolver de manera eficiente el problema de *Document Mining* así como otras variaciones.

El Árbol de Sufijos Generalizado (GST) de la colección  $D$ , es un Trie en el que se insertan los sufijos de los  $N$  documentos. Se define además el Arreglo de Documentos  $E$  que permite identificar mediante  $E[i]$  el documento al que pertenece el sufijo  $SA[i]$ . Muthukrishnan muestra que es posible codificar  $E$  utilizando  $O(n \log n)$  bits y al mismo tiempo permitir enumerar los  $ndoc$  documentos presentes en un intervalo arbitrario  $E[sp, ep]$  en tiempo  $O(ndoc)$ . Para ello se define un arreglo  $C$  que almacena en  $C[i]$  la última ocurrencia de  $D[i]$  en  $D[1, i - 1]$ . Finalmente, se dota a  $C$  de estructuras adicionales [BFC00] para poder responder consultas de mínimos en rango (RMQ) en tiempo constante.

De esta forma, para responder la consulta de *Document Listing(P)* identificamos el nodo correspondiente a  $P$  en el Árbol de Sufijos en tiempo  $O(m)$ , obteniendo así el intervalo  $E[sp, ep]$  a partir del cual podemos retornar los documentos en que ocurre  $P$ .



### 3.3. La solución de Hon et al.

Si bien Muthukrishnan planteó [Mut02] una solución eficiente para *Document Mining*( $P, K$ ), desde el punto de vista del usuario esta consulta es de poco interés, puesto que es difícil saber cuál será el número de documentos retornados para un cierto valor de  $K$ . Desde un punto de vista de recuperación de la información, es mucho más interesante el problema *Top*( $P, k$ ). Un criterio sencillo de relevancia puede ser el número de ocurrencias de  $P$ , pero pueden definirse otras funciones de ranking, como  $tf \times idf$  [SWY75].

Recientemente fue planteado por Hon et al. [HSW09] una solución óptima al problema *Top*( $P, k$ ), que requiere  $O(n \log n)$  bits

La idea central de esta solución consiste en utilizar el mismo GST de la colección que utiliza Muthukrishnan, utilizando nodos enriquecidos: La idea central se basa en que cada hoja tendrá una tupla asociada al documento al que corresponde dicho sufijo, con frecuencia 1. Los nodos internos tendrán tuplas asociadas a un documento  $d$  si al menos dos de sus hijos tienen estructuras asociadas a  $d$ . También se almacena un puntero al ancestro más bajo que tenga una estructura asociada al documento  $d$  y algunos valores precalculados para poder responder las consultas.

De esta manera, se tiene una estructura que utiliza  $O(n)$  palabras y permite encontrar los  $k$  documentos con más ocurrencias de  $P$  en tiempo  $O(m + k \log k)$

### 3.4. Soluciones comprimidas

#### 3.4.1. Solución de Sadakane

Sadakane [Sad07] reemplaza el GST de la colección por el CSA correspondiente, y una representación de parentesis del arreglo  $C$  que permite realizar consultas de mínimos en rangos (RMQ) sobre ella. Además, utiliza una secuencia binaria alineada a  $T$  para marcar las posiciones en las que cada documento termina. Así, plantea una estructura que requiere  $|CSA| + 4n + o(n) + N \log(n/N)$  bits, permitiendo responder la consulta *Document Listing*( $P$ ) en tiempo  $O(m + q \log^\epsilon n)$  donde  $q$  es el número de documentos que retornará, y  $|CSA| = \frac{1}{\epsilon} H_0 + O(n)$  bits, para algún  $0 < \epsilon \leq 1$

Además, planteó una extensión, en la cual utilizando  $2|CSA| + 10n + o(n) + N \log(n/N)$  bits, permite además calcular  $tf \times idf$  para todos los documentos.

#### 3.4.2. Solución de Hon et al.

Hon et al. [HSW09] plantean estructuras de datos compactas para resolver *top*( $P, k$ ). Las estructuras utilizadas por Hon et al. son similares a la versión extendida de Sadakane para calcular  $tf \times idf$ :

- El  $CSA$  de la colección completa (concatenada)
- El  $CSA$  de cada documento,  $CSA_d$ ,  $d = 1 \dots N$
- Un bitmap  $B$  indicando con 1 las posiciones de la secuencia concatenada en la cual termina cada documento
- Para  $k = 1, 2, 4, 8, \dots n$ , se almacena un muestro del GST de los documentos, almacenando información de los  $top-k$  documentos más relevantes.

En total, estas estructuras requieren  $2|CSA| + o(n) + N \log(n/N)$  bits.

Para responder una consulta  $top(P, k)$  se aproxima  $k$  a la siguiente potencia de 2. Utilizando el muestro del GST correspondiente, y los CSA de cada documento y de la la colección es posible responder la consulta en tiempo  $O(m + k \log^{4+\epsilon} n)$ .

### 3.4.3. Solución de Mäkinen y Välimäki

La propuesta de Mäkinen y Välimäki consiste esencialmente en construir un CSA de la colección en lugar del GST, y representar el Arreglo de Documentos con un Wavelet Tree. El CSA es utilizado para determinar el intervalo  $[sp, ep]$  de ocurrencias del patrón. Posteriormente, para determinar los distintos documentos presentes en  $D[sp, ep]$ , se utiliza el Wavelet Tree de  $D$ , que junto a ciertas estructuras adicionales permite determinar  $C[i]$ .

De esta manera, su estructura requiere  $|CSA| + n \log N + 2n + o(n \log N)$  bits, y permite responder  $Document Listing(P)$  en tiempo  $O(m + ndoc)$  si  $\sigma, N = polylog(n)$ .

### 3.4.4. Solución de Gagie et al.

Utilizando una propiedad recientemente explorada del Wavelet Tree (*Range Quantile Queries* [GPT09]) Gagie et al. lograron mejorar el requerimiento de espacio de Mäkinen y Välimäki y reducir el número de operaciones. En vez de emular el arreglo  $C$  de Muthukrishnan para determinar los documentos presentes en  $D[sp, ep]$ , Gagie et al. los van descubriendo *in situ* mientras descienden hacia las hojas del Wavelet Tree. Este enfoque permite obtener el documento  $i$ -ésimo en tiempo  $O(1)$ , respondiendo  $Document Listing(P)$  en tiempo  $O(m + ndoc)$ , cuando  $\sigma, N = polylog(n)$ .

### 3.4.5. Discusión

El enfoque seguido por Sadakane y posteriormente por Hon et al. requiere esencialmente construir dos CSA, uno para la colección completa y otro para cada

uno de los documentos. De esta manera pueden calcular  $tf \times idf$  y responder  $top(P, k)$  respectivamente.

Mäkinen y Välimäki, así como Gagie et al. responden las consultas asociadas al Arreglo de Documentos de manera más rápida (pues manejan una representación directa de él), pero tienen el problema de que el Wavelet Tree del arreglo de documentos se representa de manera no comprimida.

El Wavelet Tree clásico, comprimiendo las secuencias de bits utilizando RRR [CG08], ha probado su utilidad al lograr representar la transformada de Burrows-Wheeler de un texto espacio proporcional a la entropía del texto. Sin embargo, el vector de documentos no tiene las mismas propiedades que la transformada de Burrows-Wheeler, y por tanto los enfoques conocidos hasta el momento para comprimir el Wavelet Tree resultan infructuosos aquí. Lo anterior plantea una dirección de investigación interesante, orientada a estudiar nuevas maneras de comprimir un Wavelet Tree, que resulten ser efectivas en este escenario y en otras posibles aplicaciones.

## 4. Objetivos

### 4.1. Objetivo General

El principal objetivo de este trabajo consiste en estudiar, tanto desde un punto de vista teórico, como práctico nuevas estructuras y algoritmos orientados a resolver problemas de “Recuperación de Documentos”. El estudio se centrará principalmente la búsqueda de nuevas estructuras para representar el Arreglo de Documentos de manera comprimida, permitiendo responder las consultas necesarias para los problemas de recuperación de documentos de manera rápida. Asimismo se considerarán varias posibles aplicaciones de los resultados.

### 4.2. Objetivos Específicos

- Compresión de secuencias de bits utilizando *RePair*

Se implementará y estudiará una nueva manera de comprimir secuencias de bits utilizando RePair. Para ello, además de las estructuras originales de *RePair*, deberemos almacenar estructuras adicionales para responder *rank* y *select* de manera eficiente. Estas estructuras son de interés general, y se compararán con las soluciones existentes para responder *rank* y *select*, tanto en secuencias comprimidas [RRR02, CG08, OS07], como descomprimidas [GGMN05].

- Wavelet trees para codificar el Arreglo de Documentos

Nuestra propuesta consiste en utilizar un Wavelet Tree en que las secuencias de bits de cada nivel se compriman con RePair. Resulta importante notar que el Arreglo de Documentos y el Arreglo de Sufijos están fuertemente relacionados: si los documentos tienen la misma longitud, el Arreglo de Documentos corresponde al Arreglo de Sufijos, descartando algunos de los bits menos significativos. Por otro lado, las regularidades en el Arreglo de Sufijos señaladas en la sección 2.5, han sido utilizadas para comprimir el Arreglo de Sufijos utilizando RePair [GN07], lo cual permite suponer que se obtendrán resultados interesantes.

- Otras aplicaciones

Un Wavelet Tree efectivo para comprimir el Arreglo de Documentos puede también ser utilizado para comprimir el Arreglo de Sufijos [CG08]. Lo anterior puede resultar interesante para mejorar resultados en el campo de los autoíndices. En particular, los autoíndices son muy rápidos para responder las consultas de *Count*, pero son bastante más lentos para responder *Locate*. Tal como en [GN07], nuestro Wavelet Tree del Arreglo de Sufijos puede utilizarse como un componente adicional sobre cualquier autoíndice para mejorar los tiempos de *Locate*.

Asimismo, se estudiará la efectividad de este Wavelet Tree para mejorar resultados anteriores en indexación y búsqueda restringida por intervalos [MN06]. En estos casos se utiliza un Wavelet Tree para realizar una búsqueda de puntos en un rango bidimensional.

En ambos casos, se espera que nuestra estructura resulte compresible por las regularidades del Arreglo de Sufijos señaladas en la sección 2.5.

También existen trabajos en que se reemplaza el Índice Invertido por un Wavelet Tree construido sobre las palabras [BFLN08], permitiendo comprimir y emular los algoritmos de intersección de listas. En este caso, es esperable que al comprimir el Wavelet Tree con nuestras técnicas se obtengan resultados similares a los que obtiene RePair sobre lenguaje natural [LM00], logrando capturar la entropía de orden  $k$  del texto.

## 5. Metodología

La primera fase de este trabajo consiste en desarrollar una nueva implementación para comprimir secuencias de bits, utilizando RePair, soportando las consultas *rank* y *select*. Para dar soporte a estas consultas se deben guardar valores precalculados

para ciertas posiciones y punteros que asocien la secuencia comprimida con la secuencia original. Se implementarán distintas maneras de soportar las consultas en búsqueda de las alternativas más eficientes. Se contempla además un análisis teórico de las estructuras que se desarrollen. Estas soluciones tienen interés propio y se compararán las tasas de compresión así como los tiempos en resolver las consultas con las soluciones existentes [CG08, RRR02, OS07].

La segunda fase consiste en implementar un Wavelet Tree que utilice la técnica anterior para comprimir las secuencias de bits de cada nivel. Además, se estudiará el desempeño de un Wavelet Tree mixto, que en algunos niveles utilice la compresión basada en RePair, y en otros niveles comprima utilizando RRR.

Luego se estudiará la efectividad de estos Wavelet Trees para codificar el Arreglo de Documentos. En primera instancia, nuestra solución pretende mejorar las soluciones basadas en almacenar el Arreglo de Documentos directamente [VM07, GPT09], pero también podemos estudiar como combinarlas con soluciones que utilicen distintos enfoques [HSW09]. En ambos casos, resultará interesante comparar nuestro desempeño contra el de los Índices Invertidos.

Como producto secundario, se estudiará la efectividad de nuestro Wavelet Tree como método para codificar el Arreglo de Sufijos, pudiendo compararse con el LCSA [GN07]. También estudiaremos el desempeño de nuestras estructuras en el escenario en que se utiliza un Wavelet Tree sobre las palabras de un texto para reemplazar a los Índices Invertidos, [BFLN08], y en general, en escenarios donde podamos explotar secuencias altamente repetitivas, pero que no necesariamente presenten homogeneidad local.

## Referencias

- [BFC00] Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *Proceedings of Latin American Theoretical INformatics (LATIN)*, pages 88–94, 2000.
- [BFLN08] N. Brisaboa, A. Fariña, S. Ladra, and G. Navarro. Reorganizing compressed text. In *Proc. 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 139–146. ACM Press, 2008.
- [BW94] M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, 1994.
- [CG08] 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.
- [Cla98] D. Clark. *Compact pat trees*. PhD thesis, Waterloo, Ont., Canada, 1998.
- [GGMN05] 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.
- [GGV03] 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.
- [GN07] R. González and G. Navarro. Compressed text indexes with fast locate. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4580, pages 216–227, 2007.
- [GPT09] Travis Gagie, Simon J. Puglisi, and Andrew Turpin. Range quantile queries: Another virtue of wavelet trees. In *SPIRE '09: Proceedings of the 16th International Symposium on String Processing and Information Retrieval*, pages 1–6, Berlin, Heidelberg, 2009. Springer-Verlag.
- [HSW09] Wing-Kai Hon, Rahul Shah, and Shih-Bin Wu. Efficient index for retrieving top-k most frequent documents. In *SPIRE*, pages 182–193, 2009.

- [LM00] N. Jesper Larsson and Alistair Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, November 2000.
- [MM93] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
- [MN06] V. Mäkinen and G. Navarro. Position-restricted substring searching. In *Proc. 7th Latin American Symposium on Theoretical Informatics (LATIN)*, LNCS 3887, pages 703–714, 2006.
- [MN07] 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.
- [Mun96] J. Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 37–42, 1996.
- [Mut02] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *SODA*, pages 657–666, 2002.
- [NR08] G. Navarro and L. Russo. Re-pair achieves high-order entropy. In *Proc. 18th Data Compression Conference (DCC)*, page 537, 2008. Poster.
- [OS07] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. 9th Algorithm Engineering and Experimentation (ALENEX)*, 2007.
- [RRR02] 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.
- [Sad03] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.
- [Sad07] K. Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms*, 5:12–22, 2007.
- [SWY75] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, 1975.

- [VM07] Niko Välimäki and Veli Mäkinen. Space-efficient algorithms for document retrieval. In *18th Annual Symposium on Combinatorial Pattern Matching (CPM 2007)*, LNCS 4525, page 217. Springer-Verlag, 2007.
- [Wei73] Peter Weiner. Linear pattern matching algorithms. In *SWAT '73: Proceedings of the 14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11, Washington, DC, USA, 1973. IEEE Computer Society.