
**Inscripción en el Doctorado en Ciencias
de la Computación**

Universidad Nacional de San Luis

TEMA:

Índices en Memoria Secundaria para Búsquedas en Texto.

TESISTA:

Norma Edith Herrera

ASESOR CIENTÍFICO:

Dr. Gonzalo Navarro, Universidad de Chile (Chile)

PLAN DE TESIS

1. Introducción

Una base de datos de texto es un sistema que mantiene una colección grande de texto y que provee acceso rápido y seguro al mismo. Las tecnologías tradicionales de bases de datos no son adecuadas para manejar este tipo de bases de datos dado que no es posible organizar una colección de texto en registros y campos. Además, las búsquedas exactas no son de interés en este contexto.

Sin pérdida de generalidad, asumiremos que la base de datos de texto es un único texto T que posiblemente se encuentra almacenado en varios archivos. Las búsquedas en una base de texto pueden ser búsquedas sintácticas, en las que el usuario especifica la secuencia de caracteres a buscar en el texto, o pueden ser búsquedas semánticas en la que el usuario especifica la información que desea recuperar y el sistema retorna todos los documentos que son relevantes. En este trabajo estamos interesados en búsquedas sintácticas.

Una de las búsquedas sintácticas más comunes en bases de datos de texto es la **búsqueda de un patrón**: el usuario ingresa un string P (patrón de búsqueda) y el sistema retorna todas las posiciones del texto donde P ocurre. Para resolver este tipo de búsqueda podemos o trabajar directamente sobre el texto sin preprocesarlo o preprocesar el texto para construir un índice que será usado posteriormente para acelerar el proceso de búsqueda. En el primer enfoque encontramos algoritmos como Knuth-Morris-Pratt [15] y Boyer-Moore [3], que básicamente consisten en construir un autómata en base al patrón P que guía el procesamiento secuencial del texto; estas técnicas son adecuadas cuando el texto ocupa varios megabytes. Si el texto es demasiado grande se hará necesario preprocesar el texto para construir un índice o estructura de datos que permita acelerar el proceso de búsqueda.

Construir un índice tiene sentido cuando el texto es grande, cuando las búsquedas son más frecuentes que las modificaciones (de manera tal que los costos de construcción se vean amortizados) y cuando hay suficiente espacio como para contener el índice. Un índice debe dar soporte a dos operaciones básicas:

Count : consiste en contar el número de ocurrencias de un patrón P en un texto T .

Locate : consiste en ubicar todas las posiciones del texto T donde el patrón de búsqueda P ocurre.

Mientras que en bases de datos tradicionales los índices ocupan menos espacio que el conjunto de datos indexados, en bases de datos de texto el índice generalmente ocupa más espacio que el texto pudiendo necesitar de 4 a 20 veces el tamaño del mismo [11, 21]. Algunos índices reducen el espacio ocupado restringiendo el tipo de búsqueda que se pueden resolver. Así, un **índice orientado a palabra**, permite solamente buscar palabras completas en el texto; de esta forma si buscamos el patrón es en el texto *es importante que estén presentes* el índice no retornará la ocurrencia de este patrón dentro de la palabras *estén* y *presentes*. Algunos índices orientados a palabras permiten también buscar patrones que sean inicios de palabras; en el ejemplo anterior estos índices encontrarían las ocurrencias de es en *es* y *estén* pero no las ocurrencias en *presentes*. Los índices que son capaces de encontrar todas las ocurrencias de un patrón dentro del texto se denominan índices orientados a caracteres o **índices de texto completo**; este tipo de índices encontraría las cuatro ocurrencias de la palabra es en el texto dado.

La diferencia entre estos índices reside en la cantidad de posiciones del texto (*puntos de indexación*) que se representan en el mismo: en un índice orientado a palabra sólo se representan aquellas posiciones en las cuales se inicia un palabra mientras que en los índices de texto completo todas las posiciones del texto son consideradas dentro del índice. Dado que el requerimiento de espacio de un índice es proporcional a la cantidad de puntos de indexación en el texto, un índice de texto completo ocupa más espacio que un índice orientado a palabras pero permite resolver un conjunto más amplio de búsquedas. Los índices orientados a palabras son adecuados para búsquedas en texto en lenguaje natural escritos en idiomas tales como español, inglés, francés, etc. Si el texto sobre el que se busca representa secuencias de ADN, secuencias de proteínas o es un texto en lenguaje natural escrito en idiomas tales como japonés o chino, se hace necesaria la construcción de un índice de texto completo.

En general, si un índice requiere k bytes por punto de indexación, entonces su tamaño como índice de texto completo será k veces el tamaño del texto. Una alternativa para reducir el espacio ocupado por el índice es buscar una representación compacta del mismo, manteniendo las facilidades de navegación sobre la estructura [9, 10, 12, 13, 19, 24, 25, 27]. Una línea de investigación reciente se enfoca en construir índices que no necesitan almacenar de manera explícita el texto que indexan. Estos índices, denominados *autoíndices*, mantienen información que permite reconstruir el texto indexado [4, 14, 26, 27].

Pero en grandes colecciones de texto, el índice aún comprimido suele ser demasiado grande como para residir en memoria principal. Como un ejemplo de este caso podemos nombrar las bases de datos conteniendo secuencias de ADN y secuencias de proteínas, que requieren la construcción de un índice de texto completo y cuyo tamaño implicará que el índice resida en memoria secundaria. En estos casos, la cantidad de accesos a memoria secundaria realizados durante el proceso de búsqueda es un factor crítico en la performance del índice [29].

En esta tesis estamos interesados en el diseño de índices de texto completo eficientes, comprimidos, para memoria secundaria.

2. Estado del Arte

Dado un texto $T = t_1, \dots, t_n$ sobre un alfabeto Σ de tamaño σ , donde $t_n = \$ \notin \Sigma$ es un símbolo menor en orden lexicográfico que cualquier otro símbolo de Σ , denotaremos con $T_{i,j}$ a la secuencia t_i, \dots, t_j , con $1 \leq i \leq j \leq n$. Un sufijo de T es cualquier string de la forma $T_{i,n} = t_i, \dots, t_n$ y un prefijo de T es cualquier string de la forma $T_{1,i} = t_1, \dots, t_i$ con $i = 1..n$. Cada sufijo del texto se identifica unívocamente por la posición del texto donde ese sufijo comienza, es decir, el valor i identifica al sufijo $T_{i,n}$; llamaremos al valor i *índice del sufijo* $T_{i,n}$. Un patrón de búsqueda $P = p_1 \dots p_m$ es cualquier string sobre el alfabeto Σ .

Entre los índices más populares para búsqueda de patrones encontramos el *arreglo de sufijos* [21], el *trie de sufijos* [30] y el *árbol de sufijos* [30]. Estos índices son la base para el diseño de índices eficientes en memoria secundaria y se construyen basándose en la observación de que un patrón P ocurre en el texto si es prefijo de algún sufijo del texto.

Arreglo de Sufijos: un arreglo de sufijos $A[1, n]$ es una permutación de los números $1, 2, \dots, n$ tal que $T_{A[i],n} \prec T_{A[i+1],n}$, donde \prec es la relación de orden lexicográfico. El arreglo de sufijos representa el conjunto de los n sufijos del texto ordenados lexicográficamente guardando en cada posición de A el índice del sufijo. Buscar un patrón P en T equivale a buscar todos los sufijos de los cuales P es prefijo, los cuales estarán en posiciones consecutivas de A . El proceso de búsqueda consiste entonces en dos búsquedas binarias que identifiquen el segmento del arreglo A que contiene todas las posiciones de T donde P ocurre. La figura 1 muestra un ejemplo de un arreglo de sufijos; en el ejemplo se ha indicado junto con cada valor del arreglo el sufijo que ese valor representa.

Trie de Sufijos: un árbol digital o trie es un árbol que permite almacenar un conjunto finito de strings. En este árbol, cada rama está rotulada por un símbolo del alfabeto y cada hoja representa un string del conjunto almacenado en el árbol. El conjunto total de strings se obtiene recorriendo todos los caminos posibles desde la raíz hasta una hoja y concatenando los rótulos de las ramas que forman cada uno de esos caminos. Un trie de sufijos es un trie construido sobre el conjunto de sufijos del árbol, en el cual cada hoja mantiene el índice del sufijo que esa hoja representa. Para encontrar todas las ocurrencias de P en T , se busca en el trie utilizando los caracteres de P para direccionar la búsqueda. Si la misma finaliza en un nodo interno x , entonces todas las hojas del subárbol con raíz x forman la respuesta. Si la búsqueda finaliza en una hoja, es necesario realizar una comparación entre P y el sufijo que empieza en la posición del texto indicada por la hoja para determinar si esa hoja es la respuesta. La figura 1 muestra el trie de

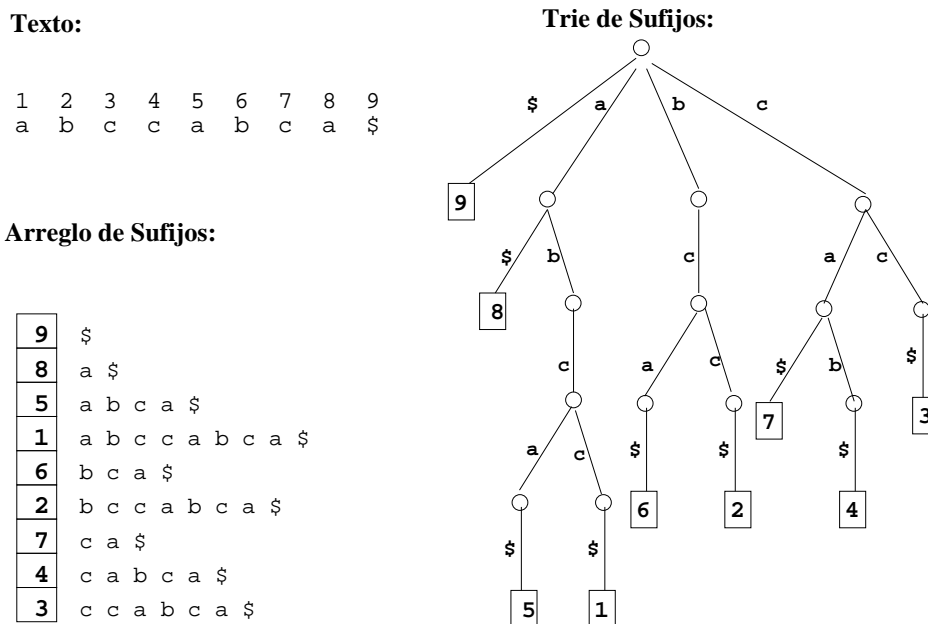


Figura 1: Un ejemplo de un arreglo de sufijos y de un trie de sufijos.

sufijos para el texto dado. Notar que si recorremos de izquierda a derecha las hojas de un árbol de sufijos obtenemos el arreglo de sufijos.

Árbol de Sufijos: Un árbol de sufijos es un Pat-Tree [11] construido sobre el conjunto de todos los sufijos de T . El Pat-Tree es una variante del trie que consiste en eliminar todas aquellas porciones del árbol que han degenerado en una lista. Para poder realizar esta modificación, cada nodo del árbol almacena un número, llamado *valor de salto* que indica la longitud de la lista que ha sido eliminada. En el árbol de sufijos el Pat-Tree se construye sobre la representación binaria de los sufijos, evitando así mantener el rótulo de cada rama: la rama izquierda se utiliza para el 0 y la derecha para el símbolo 1. Para buscar un patrón P comenzamos por la raíz y en cada paso, estando en un nodo con valor de salto j , si el j -ésimo bit de P es 0 se continúa por el subárbol izquierdo y si es 1 se continúa por el subárbol derecho. La búsqueda finaliza o bien cuando llegamos a un hoja o cuando se acaba el patrón P . En el primer caso se compara el sufijo que empieza en la posición del texto indicada por la hoja con P para saber si es o no la respuesta. En el segundo caso, la búsqueda habrá finalizado en un nodo interno x ; entonces se compara P con una de las hojas del subárbol con raíz x , si esa hoja es parte de la respuesta todas las hojas de ese subárbol lo son, caso contrario ninguna lo es. La figura 2 muestra el árbol de sufijos para el mismo texto dado en el ejemplo de la figura 1. Para cada sufijo del texto se muestra cuál es su representación binaria suponiendo que la codificación binaria de los símbolos es $\$ = 00$, $a = 01$, $b = 10$, $c = 11$.

Los arreglos y árboles de sufijos son efectivos para manejar cadenas de longitud no limitada, pero esta eficiencia se degrada considerablemente si el texto es lo suficientemente grande como para que el índice resida en memoria secundaria. Los índices clásicos como las Listas Invertidas [6] y Prefix B-Tree [2] tienen un muy buen desempeño en memoria secundaria pero su eficiencia degrada considerablemente cuando las claves de búsqueda tienen una longitud arbitrariamente grande, como ocurrirá si intentamos indexar todos los sufijos del texto. En la actualidad, el diseño de índices para textos que tengan una buena performance en memoria secundaria es un tema de creciente interés.

Entre los índices para texto en memoria secundaria más relevantes encontramos:

String B-Tree [8]: consiste básicamente en un B-Tree en el que cada nodo es representado como un Pat-Tree [11]. Este índice requiere tanto para *count* como para *locate* $O(\frac{m+occ}{b} + \log_b n)$ accesos

Sufijos:

```
$ =00
a $ =0100
a b c a $ =0110110100
a b c c a b c a $ =011011110110110100
b c a $ =10110100
b c c a b c a $ =1011110110110100
c a $ =110100
c a b c a $ =110110110100
c c a b c a $ =11110110110100
```

Árbol de Sufijos:

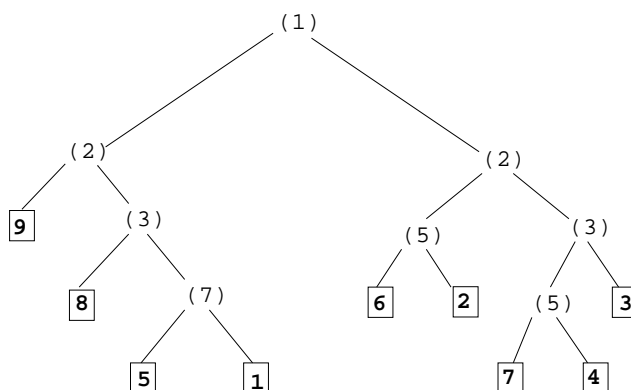


Figura 2: Codificación binaria de los sufijos (ordenadas lexicográficamente) y su correspondiente árbol de sufijos.

a memoria secundaria en el peor caso, donde occ es la cantidad de ocurrencias de P en T y b es el tamaño de páginas de disco medido en enteros. No es un índice comprimido y su versión estática requiere en espacio de 5-6 veces el tamaño del texto más el texto.

Compact Pat Tree [5]: representa un árbol de sufijos en memoria secundaria y en forma compacta. Si bien no existen desarrollos teóricos que garanticen el espacio ocupado por el índice y el tiempo insumido en resolver la búsqueda, en la práctica el índice tiene un muy buen desempeño requiriendo de 2 a 3 accesos a memoria secundaria tanto para *count* como para *locate*, y ocupando entre 4 y 5 veces el tamaño del texto más el texto.

disk-based Compressed Suffix Array [20]: adapta el autoíndice comprimido para memoria principal presentado en [27] a memoria secundaria. Requiere $n(H_0 + O(\log \log \sigma))$ bits de espacio (donde $H_k \leq \log \sigma$ es la entropía de orden k de T [22]). Para la operación *count* realiza $O(m \log_b n)$ accesos. Para la operación *locate* realiza $O(\log n)$ accesos lo cual es demasiado costoso.

disk-based LZ-Index [1]: adapta a memoria secundaria el autoíndice comprimido para memoria principal presentado en [24]. Utiliza $8 n H_k(T) + o(n \log \sigma)$ bits; los autores no proveen límites teóricos para la complejidad temporal, pero en la práctica es muy competitivo.

Dado que se utilizará como base de desarrollo de esta tesis el *Compact Pat Tree* y el *Locally Compressed Suffix Array (LCSA)* los mismos se explican en detalle en las siguientes secciones.

3. Compact Pat Tree

La información almacenada en un árbol de sufijos puede dividirse en tres categorías: la forma del árbol, los valores de salto en los nodos internos y los índices de los sufijos en los nodos hojas. Un *Compact Pat Tree (CPT)* [5] consiste de una representación compacta de cada una de estas componentes del árbol más una estrategia para manejar esta representación en memoria secundaria. Comenzaremos explicando la estructura en memoria principal para luego pasar al método de paginación.

3.1. Compact Pat Tree en Memoria Principal

Representación compacta de la forma del árbol

En el CPT se representa la forma de cada árbol como un string binario formado por un *header*, seguido de la codificación del subárbol izquierdo y de la codificación del subárbol derecho (ver figura

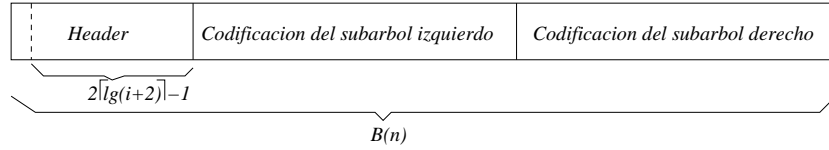


Figura 3: Representación compacta de la forma del árbol

3). El campo *header* se forma con dos valores: un único bit que indica cuál de los dos subárboles es el más pequeño y un código que indica el tamaño, en cantidad de nodos, del subárbol más pequeño. Para representar el número i , este código ocupa $2^{\lceil \lg(i+2) \rceil} - 1$ bits y se forma concatenando la codificación unaria de $\lfloor \lg(i+1) \rfloor$ con la representación binaria de $i+1$. Esta técnica permite construir un código tal que ningún valor codificado es prefijo de otro. Por ejemplo, si el subárbol más pequeño es el izquierdo y este subárbol tiene 2 nodos, el campo *header* sería 1011 donde el primer bit 1 significa *izquierdo* y los últimos tres bits 011 son el resultado de codificar el número 2 como la concatenación de $(\lfloor \log 3 \rfloor)_1 = 01$ con $(3)_2 = 11$, reutilizando el último bit de $(\lfloor \log 3 \rfloor)_1$ como primer bit de $(3)_2$. Las codificaciones de los subárboles izquierdo y derecho se construyen recursivamente usando este mismo procedimiento.

Para asegurar que las operaciones de navegación sobre el árbol puedan implementarse eficientemente, la codificación de un árbol de n nodos se aumenta de manera tal que ocupe el mismo espacio que el máximo requerido para codificar un árbol de n nodos. Sabiendo que el campo *header* ocupa $2^{\lceil \lg(i+2) \rceil}$ bits, se puede demostrar que el tamaño de la codificación de un árbol de n nodos ocupa $B(n) = 3n - 2^{\lceil \lg(n+1) \rceil} - 2v_2(n+1) + 2$ bits, donde v_2 es la cantidad de unos que hay en la representación binaria de su argumento. Esto significa que la codificación de la forma del árbol ocupa menos de 3 bits por nodo [5].

Representación compacta del valor de salto

La compresión del valor de salto que se almacena en cada nodo se basa en la distribución de estos valores. Vamos a asumir que los sufijos son strings independientes, donde el 0 y el 1 tienen igual probabilidad de ocurrir. Consideremos un nodo interno con k hojas en su subárbol; la probabilidad de que su valor de salto sea mayor que j es igual a la probabilidad de que k strings binarios aleatorios coincidan en sus primeros $(j+1)$ bits, esto es $prob(s > j) = 2^{-(j+1)(k-1)}$. Esta fórmula indica que la mayoría de los valores de salto son cero y que la probabilidad de valores grandes decrece geométricamente.

En base a esto, se codifican los valores de salto de la siguiente manera: se reserva una pequeña cantidad fija de bits para almacenar los valores de salto en los nodos internos. En caso de que un valor de salto ocupe más bits de los que hay reservados (overflow), se crea un nuevo nodo interno y se distribuye el valor entre el nodo original y el nuevo nodo creado. Además, también se crea una nueva hoja *dummy* a fin de completar el nuevo nodo interno creado. La figura 4 muestra un ejemplo de un caso de overflow, en el que se han reservado 5 bits para el valor de salto. El valor de salto 73 se ha dividido en los valores 2 y 9 teniendo en cuenta la representación binaria del 73. La nueva hoja *dummy* debe tener algún valor especial que permita reconocerla para evitar comparaciones con ella. Si un valor de salto es demasiado grande, se crearán tantos nodos internos y y nodos dummy como sean necesarios.

Hay dos supuestos en los que se basa esta representación: que los sufijos son independientes y que

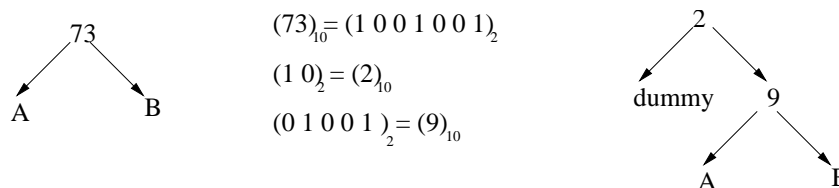


Figura 4: Representación compacta de los valores de salto.

estos sufijos son cadenas binarias generadas por un proceso aleatorio uniforme. Sin embargo estos supuestos no son reales. Para superar estos inconvenientes, se convierten los strings de entrada en strings binarios uniformes ejecutando un algoritmo de compresión de datos.

Representación compacta de los índices de los sufijos

Los índices de los sufijos almacenado en las hojas admiten compresión si estamos dispuestos a sacrificar la performance. La técnica usada en el CPT es la misma que se usa en los PaTries de Shang [28]. Se omiten los l bits de menor orden en cada uno de los índices, logrando así ahorrar nl bits en el índice completo. Luego, durante una búsqueda, cada vez que se requiera conocer el índice exacto, se deberá buscar entre 2^l sufijos posibles seleccionando aquel cuya búsqueda finalice en la hoja correcta. Esto implica un costo adicional de 2^l durante la búsqueda y un factor multiplicativo de 2^l para convertir un nodo en su lista de índices de sufijos posibles.

3.2. Compact Pat Tree en Memoria Secundaria

Para controlar la cantidad de accesos a memoria secundaria realizados durante una búsqueda en el CPT, se particiona el árbol en componentes conexas, a las que llamaremos *partes*. Cada una de las partes se almacena en una página de disco usando la representación vista en la sección 3.1, con la única diferencia que los valores almacenados en las hojas de cada parte pueden ahora ser o bien índices de sufijos o bien punteros a otra parte del árbol. En consecuencia, se debe agregar un bit en las hojas para poder distinguir ambos casos.

El algoritmo propuesto por los autores para particionar el árbol en partes es un algoritmo greedy que procede en forma bottom-up tratando de condensar en una única parte un nodo con uno o los dos subárboles que dependen de él. En este proceso de particionado las decisiones se toman en base a la profundidad de cada nodo involucrado, donde la profundidad de un nodo a es la cantidad máxima de páginas que se deben acceder en un camino que comience en a y termine en una hoja del subárbol con raíz a .

Para particionar un árbol se comienza asignando cada hoja a una parte con profundidad 1 y luego, en forma bottom-up, se van procesando cada uno de los nodos del árbol según las siguientes reglas:

- si ambos hijos tienen la misma profundidad h y las partes que contienen a los hijos y el nodo corriente entran en una página de disco, se unen ambas partes junto con el nodo corriente y se establece la profundidad del nodo corriente y de esta nueva parte en h .
- si ambos hijos tienen la misma profundidad h y las partes que contienen a los hijos y el nodo corriente no entran en una página de disco, se cierran las partes de los hijos y se crea una nueva parte para el nodo corriente con profundidad $h + 1$.
- si los hijos tienen profundidades h y k con $h < k$ y el nodo corriente y el hijo de mayor profundidad (k) entran en una página de disco, se cierra la parte del hijo con menor profundidad (h), se une el nodo corriente con la parte del hijo de mayor profundidad y se establece la profundidad de esta nueva parte en k .
- si los hijos tienen profundidades h y k con $h < k$ y el nodo corriente y el hijo de mayor profundidad (k) no entran en una página de disco, se cierran las partes de ambos hijos y se crea una nueva parte para el nodo corriente con profundidad $k + 1$.

Este método de particionamiento del árbol puede producir un número considerable de páginas que contienen partes pequeñas del árbol, lo que implica una pérdida importante del espacio total ocupado en disco por el índice. Para mejorar esta situación los autores proponen dos técnicas. Una consiste en, antes de grabar una nueva parte en disco, analizar si es posible realizar un merge de la misma con algunas

de las páginas hijas. La otra técnica consiste en realizar una pasada sobre el CPT creado con el fin de agrupar varias páginas lógicas en una física.

Bajo este método de paginación, la codificación de cada página consiste de:

- Un campo conteniendo la cantidad de nodos en la página.
- Una codificación compacta de la estructura del árbol y de los valores de salto, según lo explicado anteriormente.
- Un arreglo conteniendo las hojas del subárbol representado en esa página, las que ahora pueden ser o índices de sufijos o direcciones de otra página.

La técnica usada para comprimir los valores en las hojas que representan índices de los sufijos tiene un nuevo efecto cuando el índice se almacena en memoria secundaria. Por un lado, reconstruir el valor real del índice implica, en el peor caso, 2^l accesos a posiciones consecutivas del texto que puede resolverse con un único acceso a disco, es decir, el costo de esta reconstrucción es relativamente bajo en función de los accesos a disco necesitados. Además, reducir el espacio que se necesita para cada elemento del árbol, implica que el grado de salida de cada parte del CPT almacenada en un página sea mayor y, en consecuencia, la altura total en cantidad de página sea menor.

Pero ahora, las hojas de una parte del CPT pueden ser tanto índices de sufijos como punteros a otras páginas, y en ambos casos se debe aplicar el mismo método de omitir los l bits de menor orden. Es imposible, por el costo que implica, pensar en reconstruir el valor de una dirección de una página usando el mismo método que se usa para los índices a los sufijos. En el caso de las páginas, se puede seguir usando este método transformando cada dirección de una página en un múltiplo de 2^l , donde l es la cantidad de bits a omitir. Realizar esta transformación tiene efecto sobre el espacio necesitado para almacenar una hoja dado que la cantidad de bits a omitir debe fijarse de manera tal de asegurar que el espacio restante sea suficiente para realizar la transformación de las direcciones de página.

4. Locally Compressed Suffix Array

Un arreglo de sufijos A construido sobre un texto T de longitud n es compresible si T lo es. La entropía de orden k de T (H_k) se refleja en A formando secuencias largas $A[i, i + l]$, denominadas *pseudo-repeticiones* que aparecen en otro lugar $A[j, j + l]$ con todos los valores incrementados en uno, es decir:

$$A[j + s] = A[i + s] + 1 \text{ con } 0 \leq s \leq l$$

Por ejemplo, en el arreglo de sufijos de la figura 1 $A[7, 8] = 7, 4$ es una *pseudo-repetición* dado que aparece con los valores incrementados en 1 en $A[2, 3] = 8, 5$.

Si particionamos A en *pseudo-repeticiones* de tamaño maximal, el número de partes que obtendríamos sería a lo más $nH_k + \sigma^k$, para algún k [25]. Esta propiedad ha sido usada por varios autores para comprimir un arreglo de sufijos A [17, 18]. La técnica más exitosa se basa en usar estas regularidades para definir la función $\Psi(i) = A^{-1}[A[i] + 1]$ (o $A^{-1}[1]$ si $A[i] = n$). Dentro de una *pseudo-repetición* de A , se cumple que $\Psi(i) = \Psi(i - 1) + 1$, por lo tanto una codificación diferencial de Ψ es altamente comprimible [27].

El Locally Compressed Suffix Array (LCSA) [13] es un técnica para compresión de arreglos de sufijos que consiste en convertir las *pseudo-repeticiones* en repeticiones reales, que luego son factorizadas usando Re-Pair [16]. Para lograr esto, primero se construye A' de la siguiente manera:

$$A'[1] = A[1] \text{ y } A'[i] = A[i] - A[i - 1]$$

Si tomamos una *pseudo-repetición* de A de la forma $A[j + s] = A[i + s] + 1$ con $0 \leq s \leq l$, se cumple que $A'[j + s] = A'[i + s]$ para $1 \leq s \leq l$, lo que implica que las *pseudo-repeticiones* de A se han convertido en repeticiones reales de secuencias de números en A' .

Luego de esto se recurre a Re-Pair [16], un método de compresión basado en diccionario, para factorizar las repeticiones de A' . El método utilizado es el siguiente:

- (1) Se identifica el par más frecuente $A'[i]A'[i + 1]$; sea ab ese par.
- (2) Se crea un nuevo símbolo entero $s \geq n$ mayor que cualquier símbolo existente en A' y se agrega la regla $s \rightarrow ab$ al diccionario.
- (3) Se reemplaza toda ocurrencia de ab en A' por s .
- (4) Se repiten los pasos anteriores hasta que todo par tenga frecuencia 1.

El resultado de este proceso es una tabla de reglas R y la secuencia de símbolos en las A' fue comprimido; llamaremos a esta secuencia C . La tabla de reglas R puede mantenerse como un vector de pares donde la regla $s \rightarrow ab$ se representa en $R[s - n + 1] = a : b$

El proceso para descomprimir C es sencillo. Para descomprimir $C[i]$ primero verificamos si $C[i] < n$. Si lo es, entonces es un símbolo original de A' y el resultado de la descompresión es el mismo $C[i]$; si no lo es, obtenemos los símbolos que $C[i]$ representa accediendo a $R[C[i] - n + 1]$, y expandimos recursivamente los símbolos obtenidos. Este proceso permite reproducir u celdas de A' en tiempo $O(u)$.

Para poder recuperar los valores originales del arreglo de sufijos A , se necesitan algunas estructuras adicionales:

- Una muestra de valores de A a intervalos regulares l .
- Un mapa de bits L para marcar la posición de A' donde comienza cada símbolo de C (que puede representar varios símbolos de A').
- $o(n)$ bits adicionales para poder responder las consultas *rank* sobre L en tiempo constante, donde $rank(L, i)$ es la cantidad de 1 en $L[1, i]$.

La recuperación de una parte del arreglo de sufijos $A[i, j]$ se realiza de la siguiente manera:

- (1) Si $[i, j]$ no incluye un múltiplo de l , extendemos i a izquierda y j a derecha hasta incluir tal múltiplo.
- (2) Para asegurarnos de expandir un número completo de símbolos de C extendemos i a izquierda y j a derecha hasta que $L[i] = 1$ y $L[j + 1] = 1$.
- (3) Obtenemos $A'[i, j]$ descomprimiendo $C[rank(L, i), rank(L, j)]$ usando el proceso explicado anteriormente.
- (3) Obtenemos $A[i, j]$ usando las diferencias en $A'[i, j]$ y una muestra de A incluida en $[i, j]$.

El tamaño del LCSA es $O(H_k \log(1/H_k) n \log n)$ para algún $k \leq \alpha \log_\sigma n$ y $0 < \alpha < 1$ y el tiempo de descompresión depende del tamaño del intervalo del arreglo de sufijos a descomprimir más la extensión de ese intervalo para incluir un múltiplo de l [13].

5. Objetivos del Trabajo de Tesis

5.1. Objetivo General

En este trabajo de tesis se encara el estudio de índices de texto completo comprimidos para memoria secundaria con el fin de diseñar un nuevo índice, con estas características, que resulte competitivo tanto en el espacio utilizado como en la cantidad de accesos a discos realizados para resolver las operaciones *locate* y *count*. Se pretende proveer una implementación pública y funcional de este nuevo índice disponible para la comunidad científica relacionada al área de estudio.

5.2. Objetivos Específicos

A continuación se enumeran los objetivos específicos que se esperan lograr a través del desarrollo de esta tesis:

- a) Realizar un estudio detallado de los índices de texto completo para memoria secundaria, con el fin de comprender las técnicas existentes en el área de estudio.
- b) Realizar un estudio detallado del índice CPT con el fin de lograr una implementación eficiente del mismo.
- c) Diseñar e implementar un algoritmo que permita construir el índice CPT en memoria secundaria (algoritmo no provisto por los autores).
- d) Diseñar e implementar modificaciones del índice CPT que reduzcan el espacio utilizado por el índice. Estas modificaciones consistirán básicamente en lograr representaciones para la forma del árbol, para los valores de salto y/o para las hojas, que logren una mayor compresión
- e) Utilizando como base el método de paginación propuesto en el CPT, diseñar e implementar un nuevo índice de texto completo comprimido para memoria secundaria que sea eficiente tanto en el espacio utilizado como en la cantidad de accesos a disco realizados para resolver una búsqueda. Este nuevo índice consistirá en modificar el diseño del CPT para que el arreglo de sufijos subyacente que el CPT mantiene se comprima mediante LCSA
- f) Estudiar la forma en que la redundancia que existe en los textos compresibles se refleja en la topología de un árbol de sufijos. Esto permitirá convertir el CPT en un grafo dirigido acíclico reutilizando subárboles repetidos, con el consecuente ahorro de espacio. La técnica LCSA es un buen punto de comienzo para encarar este estudio.
- g) Comparar el índice obtenido con los existentes y proveer una implementación pública y funcional disponible para la comunidad científica.

6. Metodología

Para lograr los objetivos planteados, el trabajo a desarrollar se organizará en cuatro grandes etapas:

- Recopilación y estudio de la bibliografía existente sobre la temática de estudio.
- Diseño e implementación de los nuevos algoritmos necesarios para lograr cada uno de los objetivos planteados en la sección anterior.
- Análisis y evaluación de los nuevos algoritmos propuestos con el fin de perfeccionarlos y generar nuevas propuestas.
- Obtención, análisis y difusión de los resultados obtenidos durante el desarrollo del trabajo.
- Elaboración de la implementación que se liberará como código libre para la comunidad científica relacionada al área de estudio.

Cabe señalar que estas etapas no son secuenciales en el tiempo; los resultados parciales que se vayan obteniendo servirán como retroalimentación para el diseño de nuevos algoritmos y para el perfeccionamiento de los algoritmos ya existentes.

En primer lugar el trabajo se concentrará en proponer cambios en el diseño del índice CPT para mejorar el desempeño del mismo. La experiencia lograda durante esta primera etapa del trabajo servirá de

base para el diseño de un nuevo índice de texto completo comprimido en memoria secundaria. La evaluación de las propuestas que vayan surgiendo durante el desarrollo del trabajo serán tanto teóricas como empíricas. Se medirá la calidad de los cambios y nuevos diseños propuestos a partir de dos factores: cantidad de espacio utilizado por el índice y cantidad de accesos a memoria secundaria realizados para resolver las consultas. Para la evaluación empírica se cuenta con un conjunto de textos de prueba ampliamente usados y aceptados por la comunidad científica del área de estudio; los mismos se encuentran disponibles en el sitio <http://pizzachili.dcc.uchile.cl>.

7. Actividades Realizadas

Como ya se ha empezado a trabajar en el presente plan de tesis, se detalla a continuación las actividades que ya han sido realizadas:

- Se ha realizado la recopilación bibliográfica de la temática relacionada a este trabajo de tesis, esto incluye: índices comprimidos en memoria principal, índices comprimidos en memoria secundaria, algoritmos para la creación de árboles y arreglos de sufijos en memoria secundaria,
- Se ha diseñado e implementado un algoritmo que permite la creación del CPT en memoria secundaria. Este algoritmo combina las ideas de paginación del CPT con el algoritmo de creación de un árbol de sufijos propuesto en [7].
- Se han diseñado e implementado los algoritmos de creación y búsqueda en un CPT en el cual se reemplaza la codificación de la forma del árbol originalmente propuesta por los autores, por la representación de paréntesis propuesta en [23]. Esta representación utiliza $2n$ bits para codificar un árbol de n nodos en lugar de los $B(n)$ bits (con $2 < B(n) < 3n$) del CPT original. Si bien esta representación es menos eficiente en búsquedas, como sólo se buscará en memoria principal sobre subárboles pequeños (limitados por el tamaño de página de disco) el desempeño del índice no se verá afectado. Los tiempos inclusive podrían mejorar dado que, reducir el espacio usado para codificar la forma del árbol, permite que cada página mantenga subárboles más grandes y la altura total del CPT (en cantidad de páginas) disminuya.
- Se ha diseñado y se está implementado una segunda modificación al CPT que consiste en almacenar la hojas del árbol que son índices de sufijos en un archivo separado. Para realizar esto, cada parte del árbol, en lugar de mantener el conjunto total de hojas sólo mantiene las hojas que son punteros a páginas y un bitmap para indicar el tipo de cada hoja. De esta forma se espera lograr que las partes contengan subárboles más grandes y, en consecuencia, la altura total en cantidad de páginas sea menor. Además, esta variación permitirá que se usen codificaciones distintas para las hojas que sean punteros a páginas y para las hojas que sean índices de sufijos.

8. Programa de Cursos de Doctorado

Los cursos de doctorado a realizar deberán cubrir las siguientes temáticas:

- Complejidad de algoritmos
- Compresión de textos
- Bases de Datos Avanzadas

Referencias

- [1] D. Arroyuelo and G. Navarro. A lempel-ziv text index on secondary storage. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4580, pages 83–94, 2007.
- [2] R. Bayer and K. Unteraurer. Prefix B-trees. *ACM Trans. Database System*, pages 11–26, 1977.
- [3] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.
- [4] N. Brisaboa, A. Fariña, G. Navarro, A. Places, and E. Rodríguez. Self-indexing natural language. In *Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS. Springer, 2008.
- [5] D. Clark and I. Munro. Efficient suffix tree on secondary storage. In *Proc. 7th ACM-SIAM Symposium on Discrete Algorithms*, pages 383–391, 1996.
- [6] C. Faloutsos. Access methods for text. *Computing Surveys*, 17(1):49–74, 1985.
- [7] Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000.
- [8] P. Ferragina and R. Grossi. The string B-tree: a new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, 1999.
- [9] P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.
- [10] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms*, 3(2):20, 2007.
- [11] G. H. Gonnet, R. Baeza-Yates, and T. Snider. *New indices for text: PAT trees and PAT arrays*, pages 66–82. Prentice Hall, New Jersey, 1992.
- [12] R. González and G. Navarro. A compressed text index on secondary memory. In *Proc. 18th International Workshop on Combinatorial Algorithms (IWOCA)*, pages 80–91. College Publications, UK, 2007.
- [13] 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.
- [14] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symp. on Discrete Algorithms (SODA'03)*, pages 841–850, 2003.
- [15] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):323–350, 1977.
- [16] N. Jesper Larsson and Alistair Moffat. Offline dictionary-based compression. In *DCC '99: Proceedings of the Conference on Data Compression*, page 296, Washington, DC, USA, 1999. IEEE Computer Society.
- [17] V. Mäkinen. Compact suffix array: a space-efficient full-text index. *Fundam. Inf.*, 56(1,2):191–210, 2002.
- [18] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic J. of Computing*, 12(1):40–66, 2005.
- [19] V. Mäkinen and G. Navarro. *Compressed Text Indexing*, pages 176–178. Springer, 2008.

- [20] V. Mäkinen, G. Navarro, and K. Sadakane. Advantages of backward searching — efficient secondary memory and distributed implementation of compressed suffix arrays. In *Proc. 15th Annual International Symposium on Algorithms and Computation (ISAAC)*, LNCS 3341, pages 681–692. Springer, 2004.
- [21] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal of Computing*, 22(5):935–948, 1993.
- [22] G. Manzini. An analysis of the burrows—wheeler transform. *J. ACM*, 48(3):407–430, 2001.
- [23] J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, 2001.
- [24] G. Navarro. Indexing text using the ziv-lempel trie. *Journal of Discrete Algorithms (JDA)*, 2(1):87–114, 2004.
- [25] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):2, 2007.
- [26] K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *ISAAC '00: Proceedings of the 11th International Conference on Algorithms and Computation*, pages 410–421, London, UK, 2000. Springer-Verlag.
- [27] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Algorithms*, 48(2):294–313, 2003.
- [28] H. Shang. Trie methods for text and spatial data structure on secondary storage. *PhD Thesis*, 1995.
- [29] J. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.
- [30] P. Weiner. Linear pattern matching algorithm. In *Proc. 14th IEEE Symposium Switching Theory and Automata Theory*, pages 1–11, 1973.