Árboles de Sufijos Comprimidos en Memoria Secundaria

Norma Edith Herrera[1] and Gonzalo Navarro[2] *

[1] Dpto. de Informática, Universidad Nacional de San Luis, Ejército de los Andes 950, San Luis, Argentina, nherrera@unsl.edu.ar
[2] Dpto. de Ciencias de la Computación, Universidad de Chile, Avenida Blanco Encalada 2120, Santiago, Chile, gnavarro@dcc.uchile.cl

Resumen The text indexes provide fast substring search over large text collections. The size of a text index is from 4 to 20 times the text size, consequently the design of text indexes in external memory is very interesting area of research. The *Compact Pat Tree (CPT)* is one of the most relevant text index for external memory. It represents a suffix tree in compact form in secondary memory. The principal problem of the *CPT* is that it produces many small pages and poor fill ratios. In this paper we present a practical implementation of the *CPT* and we propose modifications in the design which allows to reduce the space wasted.

1. Introducción

Una base de datos de texto es un sistema que provee acceso rápido y seguro a una colección grande de texto. Sin pédida de generalidad, asumiremos que esta colección es un único texto T que posiblemente se encuentra almacenado en varios archivos. Una de las búsquedas más comunes en una base de datos de texto es la *búsqueda de un patrón*: el usuario ingresa una cadena de caracteres P (*patrón de búsqueda*) y el sistema retorna todas las posiciones de T donde P ocurre. Para resolver este tipo de búsqueda podemos o trabajar directamente sobre el texto sin preprocesarlo [1] o podemos preprocesar el texto para construir un índice. 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***, que consiste en contar el número de ocurrencias de P en T, y ***locate*** , que consiste en ubicar todas las posiciones de T donde P ocurre.

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 [6,11]. 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 [5,7,10,13]. Pero en grandes colecciones de texto el índice aún comprimido suele ser demasiado grande como para residir en memoria principal. 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.

* Este trabajo ha sido financiado por el Proyecto Fondecyt 1-080019, Chile, y por el proyecto 22/F614, UNSL, Argentina.

Uno de los índices para texto en memoria secundaria más relevantes es el *Compact Pat Tree* (CPT) [2], que consiste en representar un árbol de sufijos en memoria secundaria y en forma compacta. Si bien no existen desarrollos teóricos que garanticen el espacio ocupado por este índice y el tiempo insumido en la búsqueda, en la práctica tiene un muy buen desempeño requiriendo de 2 a 3 accesos a memoria secundaria tanto para *count* como para *locate*, y ocupando entre 4 a 5 veces el tamaño del texto. Uno de los principales problemas de este índice es el espacio desperdiciado del total de espacio ocupado por el índice. En los experimentos reportados en [2] el desperdicio de espacio varía entre el $40\,\%$ para textos de 900KB hasta el $60\,\%$ para textos de 100MB. Los autores proponen varias técnicas para la reducción de este desperdicio pero no reportan resultados de las mejoras obtenidas con estas técnicas.

En este artículo presentamos una implementación práctica del CPT en la que hemos modificado algunas codificaciones de las originalmente propuestas, teniendo en cuenta consideraciones de índole práctica, y hemos incorporado las técnicas de reducción del desperdicio propuestas por los autores; con esta implementación el índice desperdicia aproximadamente entre un $25\,\%$ y un $39\,\%$ del espacio total ocupado. Presentamos también una modificación en el diseño del CPT que ha permitido bajar el desperdicio de espacio al $20\,\%$ en el peor caso manteniendo la eficiencia del índice. Lo que resta del artículo está organizado de la siguiente manera. En la sección 2 presentamos una breve reseña del trabajo relacionado. En las secciones 3 y 4 explicamos la implementación del CPT realizada y la modificación en el diseño del CPT. Los resultados empíricos que obtuvimos con ambas implementaciones se muestran en la sección 5. Finalmente, en la sección 6 damos las conclusiones y las líneas de trabajo futuro.

2. Trabajo Relacionado

Dado un texto $T = t_1, \ldots, t_n$ 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, un sufijo de T es cualquier string de la forma $T_{i,n} = t_i, \ldots, t_n$ y un prefijo de T es cualquier string de la forma $T_{1,i} = t_1, \ldots, t_i$ con $i = 1..n$. Cada sufijo $T_{i,n}$ se identifica unívocamente por i; llamaremos al valor i ***índice del sufijo*** $T_{i,n}$. Un patrón de búsqueda $P = p_1 \ldots p_m$ es cualquier string sobre el alfabeto Σ.

Entre los índices más populares para búsqueda de patrones encontramos el arreglo de sufijos [11] y el árbol de sufijos [15]. 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.

Un ***arreglo de sufijos*** $A[1, n]$ es una permutación de los números $1, 2, \ldots, n$ 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. Un ***árbol de sufijos*** es un Pat-Tree [6] construido sobre el conjunto de todos los sufijos de T. El Pat-Tree es una variante del Trie[15] 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

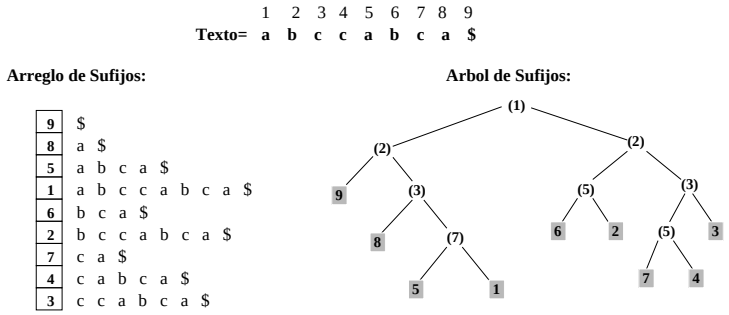

Figura 1. Un ejemplo de un arreglo de sufijos y un árbol de sufijos. El árbol de sufijos se construye sobre la representación binaria del alfabeto evitando así mantener el rótulo de cada rama.

Pat-Tree se construye sobre la representación binaria de los sufijos, evitando así mantener el rótulo de cada rama. La figura 1 muestra un ejemplo de un arreglo de sufijos y un árbol de sufijos para el texto *abccabca$*. En el caso del arreglo de sufijos, se ha indicado junto con cada valor del arreglo el sufijo que ese valor representa. En el caso del árbol de sufijos, para su construcción se ha supuesto que la codificación binaria de los símbolos es $\$ = 00, \mathrm{a} = 01, \mathrm{b} = 10, \mathrm{c} = 11$. Notar que si recorremos de izquierda a derecha las hojas de un árbol de sufijos obtenemos el arreglo de sufijos.

Uno de los principales problemas de estos índices es el espacio ocupado. Un forma de reducir espacio es usar enfoques, como el presentado en [9], que explotan redundancias el índice para ahorrar espacio en su representación, pero no pueden ser considerados índices comprimidos. Otro enfoque consiste en buscar representaciones comprimidas manteniendo las facilidades de navegación en el índice. En este trabajo estamos interesados en estos últimos.

La información almacenada en un árbol de sufijos puede clasificarse 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)** [2] consiste de una representación compacta de cada una de estas componentes del árbol más una estrategia de paginado que permite manejar esta representación en memoria secundaria. En la siguiente subsección veremos cada uno de estos puntos.

2.1. Compact Pat Tree

Para representar la forma del árbol los autores proponen una representación similar a la Jacobson [8] la cual necesita un poco más de espacio que el óptimo requerido ($2n$) pero permite realizar las operaciones de navegación sobre el árbol directamente sobre la representación compacta del mismo.

Para comprimir los valores de salto los autores se basan en que la probabilidad de que un valor de salto sea mayor que j es $2^{-(j+1)(k-1)}$. 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, para comprimir los valores de salto se reserva una cantidad pequeña y fija de bits. Si un valor de salto v ocupa más bits de los que hay reservados, se crea un nuevo nodo interno y se distribuye la representación binaria de v entre el nodo original y el nuevo nodo creado. El nuevo nodo interno creado tiene como

hijo derecho al nodo original y como hijo izquierdo una nueva hoja *dummy*. La hoja *dummy* tiene un valor especial que la distingue del resto y que permite saber durante la búsqueda que el valor real de salto se obtiene concatenando el valor actual con su hijo derecho. Se pueden crean tantas hojas *dummy* como se necesiten.

Para comprimir los índices de los sufijos almacenados en las hojas los autores del CPT proponen la misma técnica que se usa en los PaTries de Shang [14]. Se omiten los l bits de menor orden en cada uno de los índices de sufijos, logrando así ahorrar nl bits en el índice completo. Durante una búsqueda, cada vez que se requiera conocer el índice exacto del sufijo, se deberá buscar entre 2^l sufijos posibles seleccionando aquel cuya búsqueda finalice en la hoja correcta.

Para controlar la cantidad de accesos a memoria secundaria realizados durante una búsqueda en el CPT, se particiona el árbol en componentes conexas llamadas *partes*, cada una de las cuales se almacena en una página de disco. El algoritmo de paginado propuesto por los autores 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. Los valores almacenados en las hojas de cada parte pueden ahora ser o bien índices de sufijos o bien punteros a otra parte (página) del árbol. En consecuencia, se debe agregar un bit por cada hoja para poder distinguir ambos casos, lo que implica un *bitmap* adicional.

La técnica usada para comprimir los valores que representan índices de sufijos tiene un nuevo efecto cuando el índice se almacena en memoria secundaria. 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 de 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.

3. Una Implementación Práctica del CPT

Con el fin de lograr una implementación práctica del CPT, se realizaron modificaciones en la representación del Pat-Tree subyacente, las que explicamos a continuación.

Representación de la forma del árbol. La codificación de la forma del árbol originalmente propuesta, si bien no es óptima en espacio dado que ocupa $B(n)$ bits con $2n < B(n) < 3n$, permite implementar eficientemente todas las operaciones de navegación sobre el árbol. Esto tiene sentido cuando la cantidad de nodos n es relativamente grande. Pero si n es demasiado grande, la codificación del árbol excede la capacidad de memoria principal y por lo tanto se pagina diviendoló en partes y codificando cada parte separadamente. En este caso, la búsqueda en memoria principal se realiza sobre codificaciones de partes del árbol que serán pequeñas, dado que la cantidad n de nodos que hay en cada parte estará limitada por el tamaño de página de disco. Por esta razón, en memoria secundaria tiene sentido reemplazar la codificación del árbol por una que

sea óptima en espacio y eficiente de navegar para n pequeños. En este trabajo se utilizó la representación de paréntesis [12] que consiste en realizar un barrido preorden del árbol colocando un paréntesis que abre cuando visitamos un nodo y un paréntesis que cierra cuando terminamos de barrer el subárbol izquierdo del mismo.

La figura 2 muestra un ejemplo de un CPT construido sobre el mismo texto de la figura 1, en el que se han generado 4 páginas lógicas para el índice. En cada parte, las hojas sombreadas representan índices de sufijos y las restantes representan punteros a otras páginas. En la figura también se muestra la secuencia de bits que codifica la página 0. En esta secuencia, las líneas en negrita representan las divisiones entre las distintas componentes y las líneas de punto representan divisiones entre elementos del mismo tipo. Los primeros 6 bits representan la forma del árbol codificada usando la representación de paréntesis, donde 1 es paréntesis que abre y 0 paréntesis que cierra. Notar que sólo se codifican los nodos internos del árbol. Para cada valor de salto se han utilizado 3 bits por lo tanto ningún nodo interno genera una hoja dummy.

Reducción del desperdicio de espacio. La técnica de paginado del CPT produce desperdicio de espacio dentro de cada página. Los autores proponen técnicas para reducir este desperdicio, sin dar detalles de implementación de las mismas. Una de ellas es el ***merge*** que consiste en analizar, antes de grabar una página, si alguna página hija j tiene espacio suficiente como para contener la página a grabar. En caso de que si, se realiza el merge de las codificaciones de ambas páginas y se graba el subárbol resultante en la página j. La otra técnica es el ***empaquetado*** de páginas que consiste en realizar, cuando finaliza la creación del CPT, una pasada adicional con el fin de tratar de ubicar en una misma página física varias páginas lógicas, tantas como el tamaño de página permita. Lograr el empaquetado óptimo es un problema NP-completo (Bin Packing Problem), por esta razón en nuestra implementación hemos utilizado el algoritmo de aproximación *First Fit* [4]. El proceso de empaquetado implica que cada hoja que sea puntero a página sea modificado de manera tal que indique ahora tanto el número de página como el desplazamiento dentro de la misma. Como no se sabe de antemano cuántas páginas se podrán empaquetar, hay que reservar para cada hoja $\lceil \log_2 B \rceil$ bits adicionales, donde B es el tamaño de página, lo que produce que se pierda el espacio que se había ganado al eliminar bits en la representación de las hojas. Por esta razón, en nuestra implementación se decidió no eliminar bits en la representación de las hojas y fijar la cantidad máxima de páginas lógicas que se empaquetarán en una física. Por ejemplo, para un texto de 10MB se necesitan 24 bits para representar un índice de sufijo; en este caso se reservan esos 24 bits y si una hoja es un puntero a página utiliza una cantidad fija de bits para el desplazamiento y los restantes bits para el número de página.

En el ejemplo de la figura 2 se han generado 4 páginas lógicas dos de las cuales se han empaquetado en la página física 1. Se han utilizado 4 bits para cada hoja. Si la hoja es un puntero a otra página se utiliza 1 bit para empaquetar, lo que indica que como máximo podemos empaquetar dos páginas lógicas en una física.

Mejorando la operación *count.* Las búsquedas de patrones de pequeña longitud finalizan en las primera páginas del CPT. Si la búsqueda es un *locate* y es exitoso, deberemos necesariamente barrer todo el subárbol para poder recuperar los índices de los sufijos que forman la respuesta a la consulta. Si la búsqueda es un *count*, también deberemos

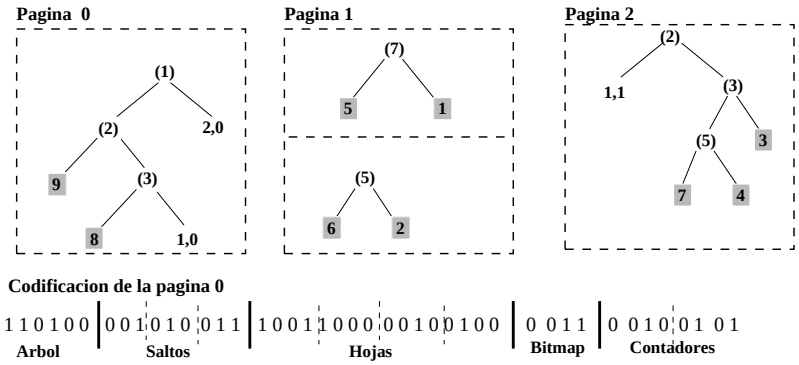

Figura 2. Un ejemplo de un CPT construido sobre el mismo texto de la figura 1.

barrer todo el subárbol ya que, si bien no necesitamos los índices de los sufijos, necesitamos contar la cantidad de sufijos que dependen del subárbol. Para evitar que un *count* realice este barrido, agregamos un contador por cada página que indica la cantidad de sufijos que dependen del subárbol contenido en esa página. Cada uno de estos contadores se mantiene en la página padre a fin de que se puedan usar en el momento de resolver el *count*.

Supongamos que la búsqueda de un patrón P finaliza exitosamente en un nodo x de una de página j, que el subárbol con raíz x tienen n hojas y que hay ant hojas anteriores a la primer hoja del subárbol con raíz x. Sabemos además que cada 0 en el bitmap de la página (que denotaremos con bm) representa un sufijo y cada 1 representa un puntero a otra página cuya cantidad total de sufijos es la que indica el contador correspondiente. Un par de operaciones *rank* [3] sobre el bitmap de la página j permiten resolver el *count*, donde $rank(bm, i)$ es la cantidad de unos en $bm[1..i]$. La figura 3 muestra el pseudocódigo del algoritmo que resuelve la operación *count*. En este algoritmo $SearchPatTree$ es el encargado de realizar la búsqueda en el CPT y $Extraer$ es el encargado de recuperar el contador $pagesant + i$ de la página j. Dado que el bitmap contenido en una página no es demasiado grande, el *rank* se resuelve usando *popcount* [3] sin crear ninguna estructura auxiliar.

4. Modificaciones en el Diseño del CPT

Mientras más pequeña es la codificación de cada parte del CPT, más grande será el subárbol que contendrá cada página y por lo tanto, la altura total, en cantidad de páginas, será menor. Persiguiendo este objetivo, modificamos el diseño del CPT de la siguiente manera: el arreglo de sufijos subyacente que mantiene el CPT (hojas que son índices de sufijo) se almacena en un archivo separado del archivo que contiene la estructura del árbol. Dentro de cada página del CPT sólo mantenemos el valor de aquellas hojas que son punteros a páginas. Para que las operaciones *count* y *locate* puedan realizarse, cada página debe almacenar, en lugar del contador, el desplazamiento de su primer sufijo dentro del arreglo de sufijos. Este desplazamiento se almacena en la página padre, con la misma idea con que usábamos los contadores en la sección anterior.

```
Count(P)
1. SearchPatTree(P, exito, CPTparte, n, ant);
2. If exito
3.    pagesant = rank(CPTparte.bm, ant)
4.    pages = rank(CPTparte.bm, ant + n) − pagesant
5.    c = n − pages
6.    For i = 1 to pages
7.       c = c + Extraer(CPTpart.Contadores, pagesant + i);
8.    return(c) ;
9. end If
```

Figura 3. Resolución de la operación *count* sobre el CPT con el agregado de contadores.

Sabemos que, una vez ubicada la posición del primer sufijo que forma parte de la respuesta a la búsqueda (ya sea *count* o *locate*) todos los demás sufijos se encuentran en posiciones consecutivas del arreglo de sufijos. En consecuencia, la estructura del árbol sólo se utiliza para ubicar la ocurrencia del primer sufijo; la ocurrencia del último sufijo se calcula utilizando los desplazamientos de los sufijos de las páginas hijas. Si la búsqueda es un *count* con una simple resta podemos calcular la cantidad de ocurrencias del patrón; si la búsqueda es un *locate* deberemos además realizar los accesos correspondientes al archivo que contiene el arreglo de sufijos.

5. Evaluación Experimental

Por razones de espacio en esta sección sólo mostramos las gráficas que consideramos más representativas. En esta sección la sigla CPT será usada para la versión explicada en la sección 3 y la sigla CPT-SA será usada para la versión explicada en la sección 4. Para la evaluación experimental se utilizaron los textos *DNA*(contiene secuencias de DNA), *Proteins* (contiene secuencias de proteinas) y *Sources* (contiene código fuente C/JAVA) cada uno de 50MB, obtenidos del sitio http://pizzachili.dcc.uchile.cl. En estos textos 26 bits por hoja son suficientes para mantener un índice de sufijo en las hojas. Si la hoja es un puntero a página utilizará 24 de esos bits para el número de página y 2 para el desplazamiento (limitamos a 4 la cantidad máxima de páginas a empaquetar).

Para establecer la cantidad de bits por salto (bs) a utilizar, se realizaron experimentos con cada texto utilizado.Valores muy pequeños para bs aumentan demasiado la cantidad de hojas dummy generadas, por lo que el espacio ahorrado en la representación de cada valor de salto se pierde con el incremento de la cantidad total de hojas a representar. Valores demasiado grandes para bs tampoco son adecuados dado que, si bien se disminuye el porcentaje de dummy generadas, esta disminución no alcanza para compensar el aumento de espacio usado en la representación de cada valor de salto. Esta observación puede apreciarse claramente en la figura 4 en la que se ha representado el espacio total ocupado y la cantidad total de celdas dummy generadas en función de bs para el lote *DNA*. Los valores más adecuados para bs resultaron 6 y 7 para *DNA*, 12 para *Proteins* y 10 para *Sources*.

El cuadro 1 muestra las estadísticas de construcción para el CPT sobre los 3 textos utilizados. Con respecto a la reducción de espacio lograda con el proceso de empaquetado los porcentajes varían entre el $31\,\%$ para *DNA* y el $52\,\%$ para *Sources*. En este último caso si no se hubieran empaquetado, el índice hubiera ocupado aproximadamente 800MB en lugar de los 424.02MB que se lograron al empaquetar. Sin embargo, si

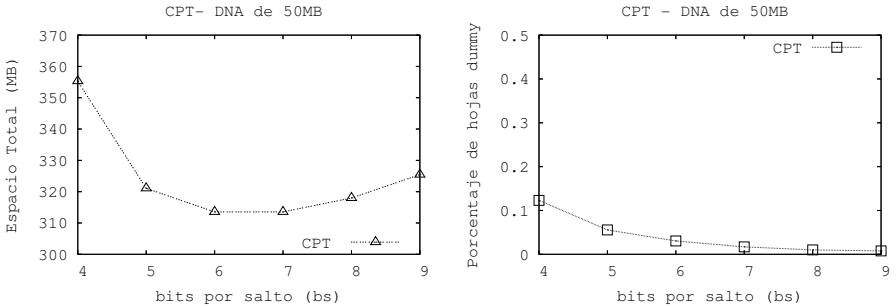

Figura 4. *DNA*: espacio ocupado y porcentaje de hojas dummy generadas por el CPT.

observamos el desperdicio de espacio es justamente en *Sources* donde se da el mayor valor llegando a un un $39\,\%$ de espacio desperdiciado que equivalen a 165.82MB del total ocupado. Esto significa que, si el CPT creado tiene muchas partes pequeñas que generan un bajo porcentaje de llenado de cada página, el empaquetado no es suficiente para solucionar este problema. Esto es lo que está sucediendo con *Sources*, donde se generan partes con subárboles de 236 nodos aproximadamente lo que provoca que en promedio se desperdicien 1.56KB de cada página, aún después de empaquetar. Aumentar el número de bits a usar en el desplazamiento a fin de permitir que se empaqueten más paginas lógicas no soluciona el problema; comprobamos experimentalmente que como máximo se empaquetan 3 páginas lógicas en una física.

El cuadro 2 muestra las estadísticas de construcción sobre el CPT-SA. En este cuadro además del espacio total ocupado, se muestra el espacio ocupado por el árbol y el espacio ocupado por el arreglo de sufijos. En general, la relación entre los 3 textos respecto de espacio total ocupado y espacio desperdiciado se mantiene respecto de lo que sucedía con el CPT.

Si comparamos ambas versiones del CPT podemos observar que el CPT-SA es el que menor espacio total ocupa y el que menor desperdicio produce, bajando aproximadamente a la mitad el porcentaje de desperdicio que se lograba con CPT. En ambos casos estamos usando la misma técnica de paginado lo que provoca que se produzca un desperdicio medio similar por página del árbol. La diferencia radica en que el CPT-SA, al haber separado el arreglo de sufijos subyacente, ocupa mucho menos páginas en la estructura del árbol que lo que ocupa el CPT, logrando así las reducciones antes nombradas. También se puede apreciar que el número medio de nodos por parte en el CPT-SA es más de un $100\,\%$ de lo logrado en el CPT. Todas estas observaciones quedan mejor reflejadas en el cuadro 3, donde se muestran las estadísticas de construcción para *DNA* con ambos índices. En este caso el desperdicio pasó de un $26\,\%$ a un $9\,\%$ y la altura del árbol, medida en cantidad de páginas, disminuyó en 1.

Para poder analizar que sucedía en ambos índices con las búsquedas se realizaron 20.000 operaciones *count* y 20.000 operaciones *locate* con patrones de longitud 5, 10, 15 y 20. El número medio de páginas accedidas y el tiempo medio de resolución de la búsqueda para *DNA* se muestran en la figura 5, en el caso del *locate* los tiempos se han graficado en escala logarítmica. Se puede apreciar en ambos casos que el CPT-SA

Cuadro 1. Estadística de construcción con el CPT.

Texto	***DNA***	***Proteins***	***Sources***
Reducción por Empaquetado	31 %	34 %	52 %
Espacio total Final(MB)	313,54	381,54	424,02
Espacio desperdiciado (MB)	81,63 (26 %)	117,29 (30 %)	165,82 (39 %)
Desperdicio medio por página (KB)	1,04	1,28	1,56
Nro. medio de nodos por parte	460,20	358,30	236,56

Cuadro 2. Estadística de construcción con el CPT-SA.

Texto	***DNA***	***Proteins***	***Sources***
Reducción Esp.por Empaquetado(%)	30 %	34 %	53 %
Espacio ocupado por SA(MB)	162,54	162,54	162,54
Espacio ocupado por el árbol(MB)	89,93	147,21	155,62
Espacio Total(MB)	252,47	310,05	318,16
Esp. desperdiciado (MB)	22,80 (9 %)	47,10 (15 %)	62,85 (20 %)
Desperdicio medio por página (KB)	1,01	1,28	1,62
Nro. medio de nodos por parte	1635,49	922,14	636,33

Cuadro 3. Estadísticas de construcción con $bs = 6$ sobre DNA 50MB.

Indice	***CPT***	***CPT-SA***
Espacio total ocupado(MB)	313,54	252,47 = 89,93 + 162,54
Espacio desperdiciado (MB)	81,63 (26 %)	22,80 (9 %)
Esp. desperdiciado por página (KB)	1,04	1,01
Profundidad del árbol en cant. de páginas	3	2

mantiene sus valores muy cercanos al CPT, logrando en algunos casos superarlo. Si bien el CPT-SA tiene menor altura que el CPT, realiza más accesos a disco dado que debe realizar una acceso más al arreglo de sufijos para resolver una búsqueda. Esta misma situación se repite con *Proteins* y *Sources*.

6. Conclusiones y Trabajo Futuro

En este artículo hemos presentado una implementación práctica del CPT y una modificación del diseño que consiste en eliminar de la representación del árbol el arreglo de sufijos subyacente. Experimentalmente hemos comprobado que el CPT-SA es más competitivo que el CPT tanto en espacio ocupado como en espacio desperidiciado, logrando mantener los costos de búsquedas cercanas al CPT. Como trabajo futuro nos proponemos incorporar al CPT-SA alguna técnica de compresión del arreglo de sufijos con el fin de bajar aun más el espacio ocupado por el índice.

Referencias

1. R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.
2. D. Clark and I. Munro. Efficient suffix tree on secondary storage. In *Proc. 7th ACM-SIAM Symposium on Discrete Algorithms*, pages 383–391, 1996.
3. F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proc. 15th SPIRE*, LNCS 5280, pages 176–187. Springer, 2008.

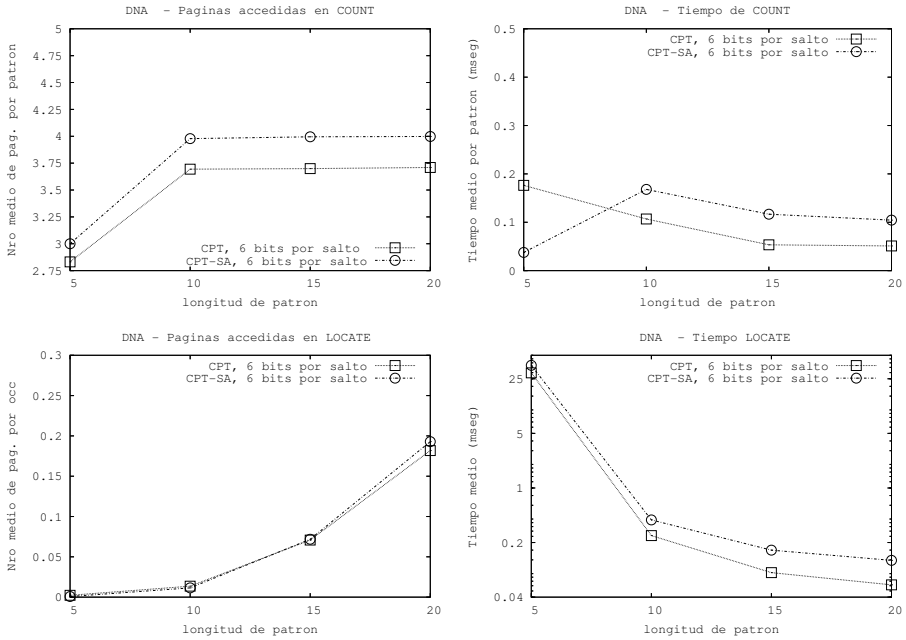

Figura 5. Nro. medio de páginas accedidas y tiempo medio de resolución de *count* y *locate*.

4. E.G. Coffman, G. Galambos, S. Martello, and D. Vigo. *Bin Packing Approximation Algorithms: Combinatorial Analysis*, pages 151–208. Kluwer Academic Publish, 1998.
5. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms*, 3(2):20, 2007.
6. 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.
7. R. González and G. Navarro. A compressed text index on secondary memory. In *Proc. 18th. IWOCA*, pages 80–91. College Publications, UK, 2007.
8. G. J. Jacobson. *Succinct static data structures*. PhD thesis, Carnegie Mellon University, USA, 1988.
9. Stefan Kurtz. Reducing the space requirement of suffix trees. *Softw. Pract. Exper.*, 29(13):1149–1171, 1999.
10. V. Mäkinen and G. Navarro. Compressed text indexing. In M.-Y. Kao, editor, *Encyclopedia of Algorithms*, pages 176–178. Springer, 2008.
11. U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal of Computing*, 22(5):935–948, 1993.
12. J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, 2001.
13. K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Algorithms*, 48(2):294–313, 2003.
14. H. Shang. *Trie Methods for Text and Spatial Data Structure on secondary storage*. PhD thesis, McGill University, 1995.
15. P. Weiner. Linear pattern matching algorithm. In *Proc. 14th IEEE Symposium Switching Theory and Automata Theory*, pages 1–11, 1973.