Una Mejora en Tiempo del Trie de Sufijos

Jesica Cornejo, Darío Ruano, Norma Herrera

Departamento de Informática, Universidad Nacional de San Luis, Argentina,
jncornej@gmail.com, dmruano@unsl.edu.ar, nherrera@unsl.edu.ar

Abstract. Un ***trie de sufijos*** es un índice para bases de datos de texto que permite resolver eficientemente las operaciones de búsqueda pero que necesita en espacio 10 veces el tamaño del texto indexado. En [14] se propone una nueva representación compacta del trie de sufijos que resulta eficiente en espacio y que permite un posterior paginado del índice. En este artículo presentamos una mejora en tiempo de búsqueda de esta representación compacta del trie de sufijos manteniendo la competitividad en espacio.
Palabras claves: Bases de Datos de Texto, Índices, Trie de sufijos.

1 Introducción

La información disponible en formato digital aumenta día a día su tamaño de manera exponencial. Gran parte de esta información se representa en forma de texto, es decir, secuencias de símbolos que pueden representar no sólo lenguaje natural, sino también música, secuencias de ADN, secuencias de proteínas, etc. Debido a que no es posible organizar una colección de textos en registros y campos, las tecnologías tradicionales de bases de datos (modelo relacional) para almacenamiento y búsqueda de información no son adecuadas en este ámbito.

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. 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.

Una de las búsquedas 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 las ocurrencias del patrón P en el texto T. Para resolver eficientemente estas búsquedas se hace necesario construir un índice que permita acelerar el proceso de búsqueda. Un índice debe dar soporte a dos operaciones básicas de búsqueda de patrón: ***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.

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 ocupa más espacio que el texto en sí mismo, pudiendo necesitar de 4 a 20 veces el tamaño del mismo.

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. Algunos índices orientados a palabras permiten también buscar patrones que sean inicios de palabras. Los índices que son capaces de encontrar

todas las ocurrencias de un patrón dentro del texto se denominan índices orientados a caracteres o ***full text indexes***.

Otra alternativa para reducir el espacio ocupado por el índice es buscar una representación compacta del mismo, manteniendo las facilidades de navegación sobre la estructura. En los últimos años se han realizado grandes avances en este ámbito Las nuevas técnicas de compresión de índices no sólo permiten reducir su tamaño, sino que procesan búsquedas más rápido que la versión sin comprimir [5, 11, 13].

Un ***trie de sufijos*** es un índice que permite resolver eficientemente las operaciones *count* y *locate* pero que necesita en espacio 10 veces el tamaño del texto indexado. En [14] y [15] se aborda el estudio de técnicas de representación del trie de sufijos con el objetivo de proponer una nueva representación compacta del mismo que resulte eficiente en espacio y que permita un posterior paginado del índice. Los autores muestran que con esta nueva representación se obtienen mejoras significativas en espacio manteniendo un buen desempeño en tiempo. Aun así, la representación clásica del trie de sufijos tiene una leve ventaja en tiempo con respeto a lo propuesto. Pero la nueva representación tiene la ventaja de permitir un posterior proceso de paginado, algo que no se puede hacer de manera eficiente con la representación clásica. En este artículo presentamos una mejora en tiempo de búsqueda de esta nueva versión del trie de sufijos.

Lo que resta del artículo está organizado de la siguiente manera. En la sección 2 presentamos el trabajo relacionado, dando los conceptos necesarios para comprender el artículo. En la sección 3 describimos la representación secuencial para el trie de sufijos en la que está basada este trabajo. La sección 4 y 5 presentan nuestra propuesta de mejora y la evaluación experimenttal de la misma. Finalizamos en la sección 6 dando las conclusiones y el trabajo futuro.

2 Trabajo Relacionado

2.1 Indexación en Bases de Datos de Texto

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, definimos:

sufijo de T: cualquier string de la forma $T_{i,n} = t_i, \ldots, t_n$ con $i = 1..n.$.
prefijo de T: 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 Σ.

Entre los índices más populares para búsqueda de patrones encontramos el arreglo de sufijos [10], el árbol de sufijos [17] y el trie de sufijos [4]. Estos índices se construyen basándose en la observación de que un *patrón P ocurre en el texto si es prefijo de algún sufijo del texto*.

Un ***trie de sufijos*** [4] es un trie[3] construido sobre el conjunto de todos los sufijos de T. Cada rama está rotulada por un símbolo de Σ y cada hoja representa un sufijo de T. El conjunto de sufijos 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. En cada nodo hoja se mantiene el índice del sufijo que esa hoja representa. La figura 1 muestra un ejemplo de un texto y su correspondiente trie de sufijos.

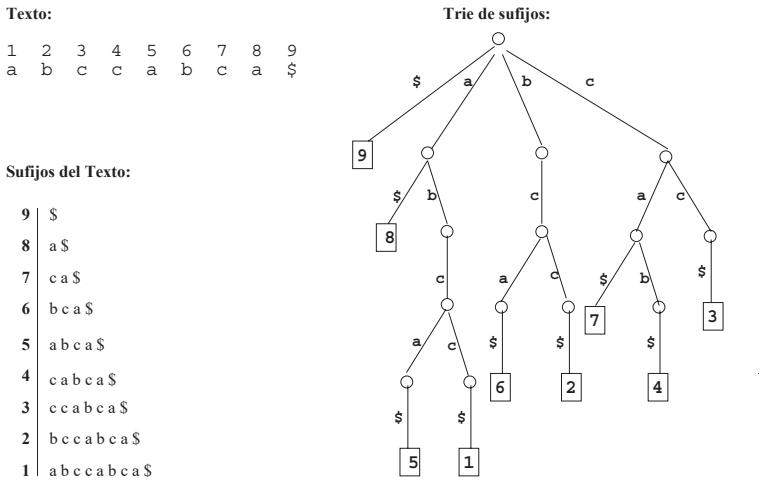

Fig. 1. Un texto, su correspondiente conjunto de sufijos y trie de sufijos.

Una forma de reducir el uso de espacio es reemplazar las ramas del árbol que han degenerado en una lista, por una única rama y agregar a cada nodo interno la longitud de la rama que se ha eliminado. Esta longitud se conoce con el nombre de *valor de salto*. La figura 2 muestra esta modificación para el trie de la figura 1. Esta última versión es la que utilizamos en este trabajo.

Para encontrar todas las ocurrencias de P en T, se busca en el trie utilizando los caracteres de P para direccionar la búsqueda. Se comienza por la raíz y en cada paso, estando en un nodo x con valor de salto j, avanzamos siguiendo la rama rotulada con el j-ésimo caracter de P. Durante este proceso se pueden presentar tres casos:

- que la longitud de P sea menor que j, por lo cual no hay caracter de P para seguir buscando en el árbol. En este caso 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.
- que x sea una hoja del árbol y por lo tanto no tiene un valor de salto asignado sino el índice de un sufijo. En este caso se debe comparar P con el sufijo indicado por la hoja para saber si ese sufijo es o no la respuesta.
- que el nodo x no tenga ningún hijo rotulado con el j-ésimo caracter de P. En este caso la búsqueda fracasa.

En cada caso, si la búsqueda es una operación *locate* y es exitosa, hay que recuperar los índices de sufijos contenidos en las hojas que forman la respuesta a la consulta. Si la búsqueda es una operación *count* y es exitosa, basta con contar la cantidad de hojas que forman parte de la respuesta.

2.2 Representaciones Compactas para Árboles

Los árboles ordenados de búsqueda son una de las estructuras de datos más relevantes. La información asociada a cada nodo suele ser importante, e incluso dominante, en

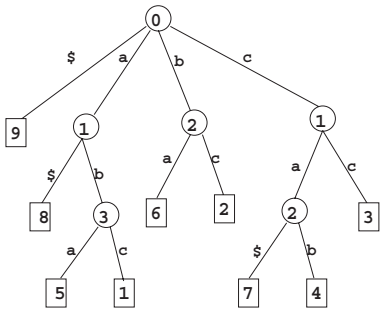

Fig. 2. Representación de Trie de Sufijos usando valores de salto.

el costo de almacenamiento total de un sistema. La cantidad de árboles binarios de n nodos puede ser calculada de la siguiente manera [2]:

$$C_n \equiv \binom{2n}{n} /(n+1)$$

Una codificación compacta de un árbol binario requerirá entonces como mínimo $\log C_n$ bits. Usando la aproximación de *Stirling* se puede demostrar que $2n$ bits son suficientes para representar un árbol de n nodos [2]. En [7] y [9] se presentan varias técnicas de representación compacta de árboles que alcanzan esta cota de $2n$ bits, sin embargo ninguna de ellas provee las funcionalidades requeridas para navegar el árbol.

La representación de paréntesis [12] es una técnica de representación compacta de árboles que consiste en realizar un barrido preorden del árbol y colocar un paréntesis que abre (un bit en 1) cuando visitamos por primera vez un nodo x y un paréntesis que cierra (un bit en 0) cuando terminamos de barrer el subárbol izquierdo de x. Esta técnica de representación alcanza la cota de $2n$ bits para un árbol de n nodos y permite una implementación eficiente de las funciones requeridas para navegar el árbol.

Para poder navegar el árbol necesitamos, dado un nodo x, saber la posición donde se encuentra su hijo izquierdo (*leftchild* (x)), la posición donde se encuentra su hijo derecho (*rightchild*(x)) y la posición del padre (*parent* (x)). En la representación de paréntesis estas operaciones se pueden resolver a partir de las siguientes operaciones básicas:

findclose(i) : retorna la posición del paréntesis que cierra que corresponde al paréntesis que abre en la posición i.

findopen(i) : retorna la posición del paréntesis que abre que corresponde al paréntesis que cierra en la posición i.

excess(i) : retorna la diferencia del número de paréntesis que abren y el número de paréntesis que cierra desde el principio a la posición i.

enclose(i) : retorna la posición del paréntesis más cercano que encierra al nodo de la posición i

En el caso de árboles r-arios no completos, en el cual cada nodo puede tener un número indefinido de hijos, se realiza un recorrido preorden comenzando desde la raíz, se escribe un paréntesis que abre cuando encontramos por primera vez un nodo x y un paréntesis que cierra cuando se termina de barrer todos los subárboles de x.

Para poder navegar el árbol necesitamos, dado un nodo x, la posición del padre, la posición del i-ésimo hijo de x y la posición de su hermano izquierdo y derecho. Estas operaciones se pueden resolver realizado una adaptación para árboles generales de las operaciones básicas para árboles binarios: *findclose*(i), *findopen*(i), *excess*(i) [12].

3 Representación Compacta de un Trie de Sufijos

La representación habitual de un trie consiste en mantener en cada nodo los punteros a sus hijos, junto con el rótulo correspondiente a cada uno de ellos. Existen distintas variantes de representación que consisten en organizar estos punteros a los hijos sobre una lista secuencial, sobre una lista vinculada o sobre una tabla de hashing [8]. Una de las propuestas de representación que mejor desempeño tiene en memoria principal es la de Kurtz, quien basándose en la idea de la representación sobre una lista vinculada, propuso que cada nodo mantenga un apuntador al primer hijo y almacenar los nodos hermanos en posiciones consecutivas de memoria. Esto permite durante una búsqueda, realizar una búsqueda binaria sobre los rótulos para decidir por cual hijo seguir.

La mayoría de las propuestas existentes mantienen explícitamente la forma del árbol con punteros, los que pueden ser punteros físicos (direcciones de memoria principal) o punteros lógicos (posiciones de un arreglo). En [14] se presenta una nueva representación de un trie de sufijos que permite reducir el espacio necesario para almacenar el índice, eliminando la necesidad de mantener los punteros explícitos a los hijos. Esta representación surge como una extensión a árboles r-arios de la técnica presentada en [6] y tiene la ventaja de permitir un posterior proceso de paginado para manejar eficientemente el trie de sufijos en memoria secundaria [16].

Notar que la información contenida en el trie está compuesta por: la topología del árbol, el rótulo de cada rama, el valor de salto de cada nodo, el grado de cada nodo y el índice del sufijo asociado a cada hoja. En [14] se propone representar de manera secuencial cada una de estas componentes, manteniendo la posibilidad de navegar eficientemente sobre el trie:

- Para la topología del trie de sufijos utilizamos un mapa de bits que mantiene la representación de paréntesis que explicamos en la sección 2.2. Llamaremos a esta secuencia *RP*.
- Para la representación de los rótulos de cada rama, de los valores de salto de cada nodo y del grado de cada nodo, se utilizan arreglos colocando los elementos que forman cada arreglo en el orden indicado por un barrido preorden del árbol. Esto permite durante la navegación del árbol moverse coordinadamente sobre todas las secuencias que conforman la representación del mismo. Estas secuencias a su vez se comprimen usando códigos *DAC* (Directly Addressable Variable-Length Code) [1], una técnica de compresión que permite acceso aleatorio y eficiente a cada código en una secuencia de códigos de longitud variable.

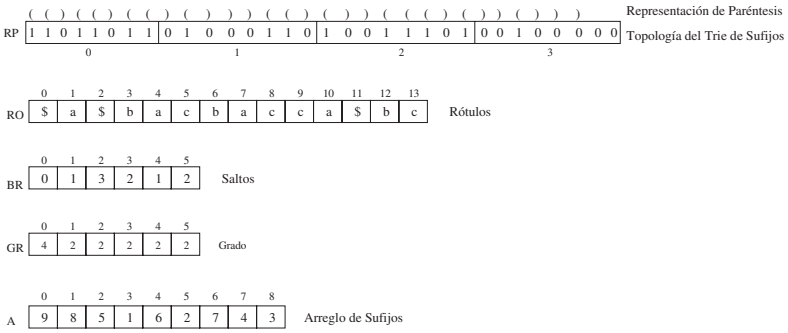

Fig. 3. Representación secuencial del trie de sufijos de la Figura 1.

- Para las hojas también se mantiene un arreglo de índices de sufijos, tomados en orden de izquierda a derecha del árbol.

La figura 3 muestra esta representación para el trie de sufijo de la Figura 1. Las secuencias se muestran sin la compresión DAC a fin de facilitar su entendimiento. Para poder navegar sobre esta representación los autores se basan en el algoritmo que permite navegar sobre una representación de paréntesis [12], adaptándolo para moverse coordinadamente sobre los 5 arreglos. Estos algoritmos necesitan realizar operaciones ***findclose***, ***excess*** y ***enclose*** sobre secuencias binarias o mapa de bits [12].

.

4 Mejorando los Tiempos de Búsqueda

Para mejorar los tiempos de resolución de una búsqueda como primer paso se analizó cuánto tiempo insumían las operaciones involucradas en el proceso de navegación en la representación compacta del trie de sufijos. Para ello se realizaron experimentos que calcularan, del tiempo total del búsqueda, que porcentaje correspondía a cada operación sobre la representación de paréntesis.

Los resultados de estos experimentos permitieron determinar que la operación que mas tiempo consume es *findclose* . La Figura 4 muestra los resultados obtenidos para un texto que contiene secuencias de ADN de tamaño 4MB y 6MB. Sobre el eje x se han representado los distintos tamaños de patrones de búsqueda y sobre el eje y está representado el tiempo insumido. Se ha graficado el tiempo total de búsqueda y el tiempo de *findclose* . Como puede observarse un 99% del tiempo de búsqueda es insumido por la operación *findclose* .

Este comportamiento se debe a que, cuando el algoritmo de búsqueda navega la representación compacta del trie de sufijos y necesita pasar de un nodo y a su hermano derecho, la operación *findclose* debe realizar barridos grandes en *RP*, la representación de paréntesis del trie. Recordemos que entre un nodo y y su hermano derecho se encuentra toda la representación del subtrie que tiene como raíz a y. Mientras mas cerca de la raíz se encuentre y mas grande será el subárbol que depende de él.

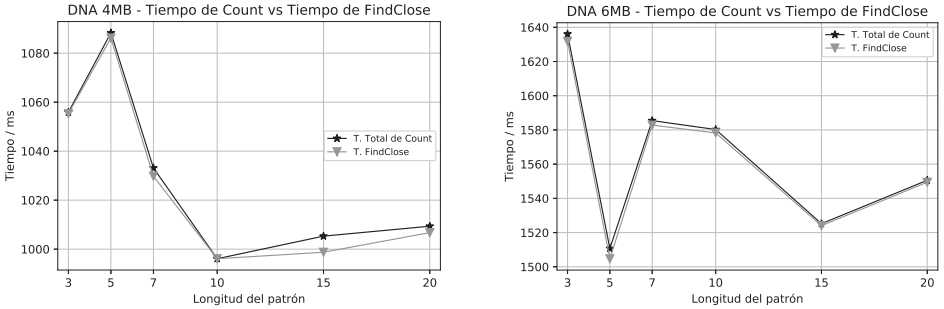

Fig. 4. Tiempo de *findclose* vs tiempo total de búsqueda.

Una forma de evitar estos barridos grandes sobre *RP*, es mantener para cada nodo y la posición donde se encuentra su paréntesis que cierra en *RP*. Esta idea es la que usamos para crear una nueva estructura que llamaremos *ParentClose*. Esta estructura tiene varios niveles y en cada nivel mantiene información que evita los barridos secuenciales grandes dentro de *RP* .

Sea x la raíz, *ParentClose* en el nivel 1 almacena para cada hijo y de x tres valores: la posición del parentesis que cierra de y, la cantidad de nodos del subárbol con raíz y y la cantidad de hojas del subárbol con raíz y. Para poder mantener la navegación dentro de la representación compacta del trie de sufijos, esta información se ordena tomando los hijos de x de izquierda a derecha.

Esta misma información se puede mantener para los restantes niveles del trie. El problema es que si *ParentClose* se mantiene en todos los niveles del trie, el espacio que se necesita haria perder lo ganado con la representación compacta planteada. Además, a medida que bajamos en el trie los subárboles son cada vez mas pequeños por lo que en algún punto será más conveniente (tanto en tiempo como en espacio) trabajar directamente sobre *RP* sin usar *ParentClose*.

En la siguiente sección presentamos los experimentos que permiten analizar el comportamiento de *ParentClose* y determinar empíricamente hasta que nivel conviene crear esta nueva estructura para que resulte competitiva.

5 Evaluación Experimental

Cabe señalar que este trabajo forma parte de un proyecto mayor cuyo objetivo principal es lograr una implementación eficiente en disco del trie de sufijos. Esto significa que como paso posterior a la creación del índice se realizará un paginado del mismo. El proceso de paginación de un índice consiste en dividir el mismo en partes, cada una de las cuales se aloja en una página de disco. Luego el proceso de búsqueda consiste en

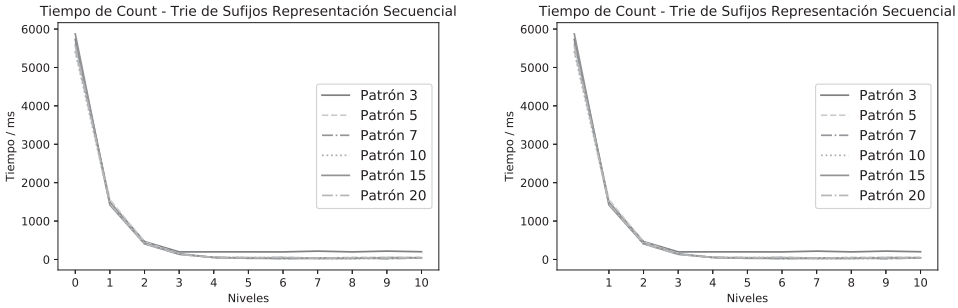

Fig. 5. Tiempo medio de *count* para distintos tamaños de patrones de búsqueda.

ir cargando en memoria principal una parte, realizar la búsqueda en memoria principal sobre esa parte, para luego cargar la siguiente y proseguir la búsqueda. Por esta razón los experimentos se han realizado sobre textos pequeños.

Para realizar esta evaluación hemos utilizado texto que contiene secuencias de ADN. tomados del sitio http://pizzachili.dcc.uchile.cl. Este sitio ofrece una amplia colección de índices comprimidos y de textos, que son los habitualmente usados por la comunidad que trabaja con índices sobre bases de datos textuales. Se han tomado textos de tamaño $4, 6, 8$ y 10 MB. Los tamaños han sido establecidos tomando en cuenta que si paginamos el índice las búsquedas se realizarán sobre parte pequeñas del mismo.

Los resultados aquí mostrados fueron obtenidos realizando búsquedas con patrones de longitud 3, 5, 7, 10, 15 y 20. Utilizamos estas longitudes ya que por prácticas anteriores han demostraron ser representativas. Para cada longitud de patrón se generaron lotes de 500 patrones.

La Figura 5 muestra los resultados obtenidos para texto de tamaño 4MB (izquierda) y 6MB (derecha). Sobre el eje x se han representado los distintos niveles de *ParentClose* y sobre el eje y está representado el tiempo medio de realizar la operación *count* sobre un patrón, expresado en milisegundos. El nivel 0 corresponde a la representación del trie sin el uso de *ParentClose*; el nivel 1 corresponde al trie usando *ParentClose*solo en la raíz; en general el nivel i corresponde al trie mas *ParentClose*para todos los nodos hasta nivel $i-1$. Como puede observase el uso de *ParentClose*mejora drásticamente los tiempos de búsqueda, logrando reducciones de hasta el 26% con nivel 1. Estas mejoras van disminuyendo a medida que aumentamos la cantidad de niveles de *ParentClose*hasta prácticamente estabilizarse en nivel 4. Esto se debe al hecho de que en ese nivel del trie los subárboles son los suficientemente pequeños como para que *ParentClose*no pueda mejorar el barrido secuencial. Con respecto a la longitud del patrón de búsqueda, no tiene efectos significativos sobre el comportamiento de *ParentClose*. Esto puede observarse más claramente en la figura 6 donde se han graficado los resultados por longitud

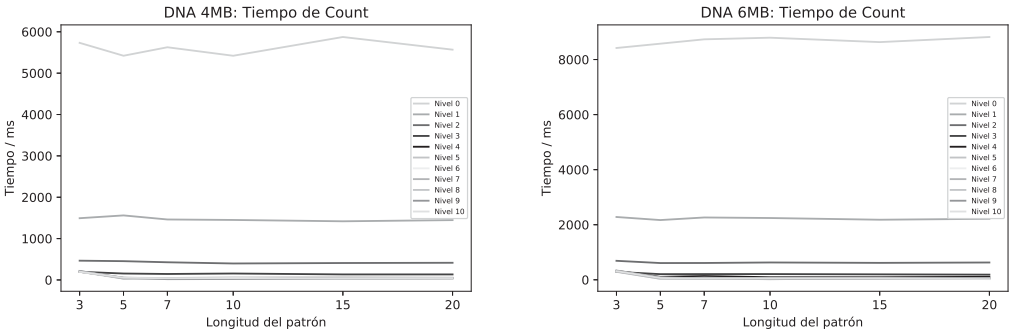

Fig. 6. Tiempo medio de *count* para distintos tamaños de patrones de búsqueda.

de patrón. Allí puede observarse que, para los niveles de *ParentClose*considerados, los tiempos de búsquedas son similares para las distintas longitudes de patrón.

Con respecto al espacio utilizado, la figura 7 muestra el espacio ocupado por *ParentClose*, el espacio ocupado por la representación compacta del trie y el espacio total ocupado (la suma de los dos anteriores). Se puede observar que del total de espacio ocupado solo un 0.03% corresponde a *ParentClose*, notándose un incremento signicativo en el espacio a partir del nivel 7 llegando a ocupar un 9% del espacio total. Esto nos indica que con una pequeña sobrecarga en el espacio ocupado podemos lograr mejoras significativas en los tiempos de búsqueda.

Teniendo en cuenta lo analizado podriamos concluir que el nivel 4 es un punto de compromiso entre espacio y tiempo para *ParentClose*. Las conclusiones obtenidas se mantienen cuando el tamaño del texto crece; no se muestran las gráficas por razones de espacio.

6 Conclusiones y Trabajo Futuro

En este trabajo presentamos una mejora en tiempo del trie de sufijos compacto presentado en [14] [15]. La misma consiste en mejorar los tiempos de la operación *findclose* utilizando una estructura auxiliar con información útil para resolver esta operación. Mostramos que con poco espacio adicional, solo un 0.3%, se pueden conseguir reducciones de hasta un26% en los tiempos de búsqueda. Con respecto al trabajo futuro nos proponemos integrar esta propuesta con técnicas de paginado para el trie de sufijos.

References

1. Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. Directly addressable variable-length codes. In *SPIRE*, pages 122–130, 2009.

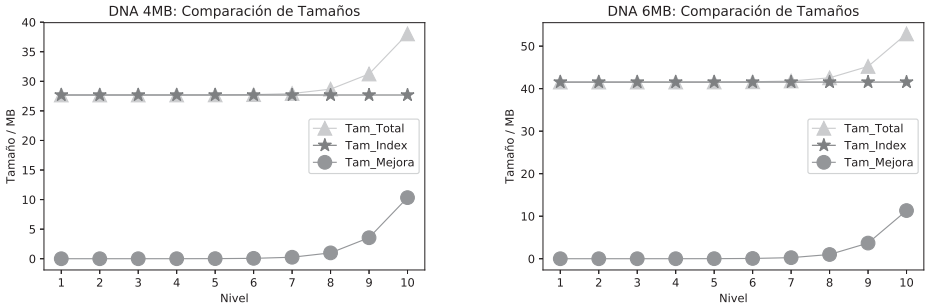

Fig. 7. Espacio total ocupado vs espacio ocupado por *ParentClose*.

2. David Clark. *Thesis: Compact Pat Trees*. PhD thesis, 1996.
3. G. H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures*. Addison-Wesley, 1991.
4. 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.
5. 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.
6. N. Herrera and G. Navarro. Árboles de sufijos comprimidos en memoria secundaria. In *Proc. XXXV Latin American Conference on Informatics (CLEI)*, Pelotas, Brazil, 2009.
7. J. Katajainen and E. Mäkinen. Tree compression and optimization with applications. *Int. Journal of Foundations of Computer Science*, 1(4):425–447, December 1990.
8. A. Thomo M. Barsky *, U. Stege. A survey of practical algorithms for suffix tree construction in external memory. In *Software: Practice and Experience*, 2010.
9. E. Mäkinen. A survey on binary tree codings. *The Computer Journal*, 34(5):438–443, 1991.
10. U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal of Computing*, 22(5):935–948, 1993.
11. E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems (TOIS)*, 18(2):113–139, 2000.
12. J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, 2001.
13. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.
14. D. Ruano and N. Herrera. Representación secuencial de un trie de sufijos. In *XX Congreso Argentino de Ciencias de la Computación*, Buenos Aires, Argentina, 2014.
15. D. Ruano and N. Herrera. Indexando bases de datos textuales: Una representación compacta del trie de sufijos. In *Congreso Nacional de Ingenier´ıa Informática / Sistemas de Información*, Buenos Aires, Argentina, 2015.
16. J. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.
17. P. Weiner. Linear pattern matching algorithm. In *Proc. 14th IEEE Symposium Switching Theory and Automata Theory*, pages 1–11, 1973.