Representación Secuencial de un Trie de Sufijos

Darío Martín Ruano, Norma Edith Herrera

Departamento de Informática, Universidad Nacional de San Luis, Argentina,
{ dmruano, nherrera }@unsl.edu.ar

Resumen 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. Por esta razón, es importante contar con una técnica de paginación que permita mantener el índice en memoria secundaria pero resolviendo eficientemente las búsquedas sobre el texto indexado. Para lograr esto, como primer paso debemos contar con una representación que sea adecuada para memoria secundaria, es decir que secuencialice la estructura del árbol. En este trabajo implementamos y evaluamos experimentalmente una representación del trie de sufijos que tiene estas características.
Palabras claves: Bases de Datos de Texto, Índices, Trie de Sufijos.

1. Introducción

La mayoría de los administradores de bases de datos actuales están basados en el modelo relacional, presentado por Edgard F. Codd en 1970. Bajo este modelo cada elemento de la base de datos puede ser almacenado como un registro (tupla) y cada registro a su vez dividido en campos (atributos). La mayoría de las consultas que se realizan a una base de datos relacional se corresponden con *búsquedas exactas*, esto significa obtener todos los registros cuyos campos coinciden exactamente con los campos aportados durante la búsqueda.

En la actualidad la información disponible en formato digital aumenta día a día su tamaño de manera exponencial. Gran parte de esta información se representa como texto, es decir como secuencias de símbolos que pueden representar no sólo lenguaje natural, sino también música, códigos de programas, 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 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 todas las posiciones de T donde P ocurre. Para poder resolver eficientemente esta búsqueda sobre una base de datos de texto se necesita preprocesar T para construir un índice que permita acelerar el proceso de búsqueda.

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.

Un punto importante a tener en cuenta es que, 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 [4][8].

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. Por esta razón es importante contar con una técnica de paginación que permita mantener el índice en memoria secundaria pero resolviendo eficientemente las búsquedas sobre el texto indexado. Para lograr esto, como primer paso debemos contar con una representación que sea adecuada para memoria secundaria, es decir, una representación que secuencialice la estructura del árbol.

En este trabajo implementamos y evaluamos experimentalmente una representación del trie de sufijos con las características antes mencionadas. Esta representación surge como una extensión a árboles r-arios de la técnica presentada en [5] para la paginación de un árbol binario. Cabe mencionar que este trabajo es parte de un proyecto mayor que consiste en lograr una implementación eficiente en disco 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 secciones 3 y 4 presentamos una nueva representación para el trie de sufijos y su evaluación experimental. Finalizamos en la sección 5 dando las conclusiones y el 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 [8], el trie de sufijos [12] y el árbol de sufijos [12] [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 árbol digital o trie [3] 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. Un trie construido sobre un conjunto de strings permite resolver eficientemente no sólo la consulta de pertenencia sino también la búsqueda de strings que comiencen con un prefijo dado. Por lo tanto, si se construye un trie sobre el conjunto de todos los sufijos del texto, basándonos en la observación anterior, podemos resolver la búsqueda de patrones.

Un ***trie de sufijos*** [4] es un trie construido sobre el conjunto de todos los sufijos de T. Cada nodo hoja de este trie mantiene el índice del sufijo que esa hoja representa. La Figura 1 muestra un ejemplo de un texto, su correspondiente conjunto de sufijos y trie de sufijos.

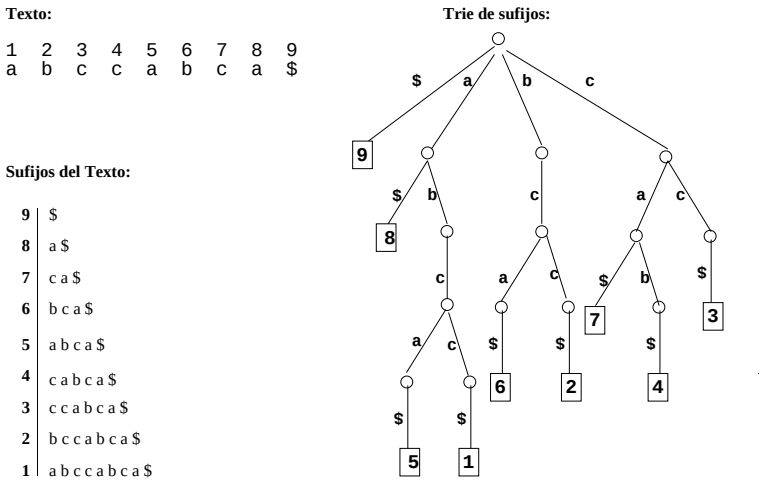

Figura 1. Un texto, su correspondiente conjunto de sufijos y trie de sufijos. A la izquierda de cada sufijo se ha indicado su correspondiente índice.

Una forma de reducir el uso de espacio es reemplazar las ramas del árbol que han degenerado en una lista, por una única rama cuyo rótulo es la concatenación de los rótulos de las ramas reemplazadas. Una variante de esta representación consiste en mantener un único caracter como rótulo de 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 estas modificaciones para el trie de la Figura 1. La versión de valores de salto es la que hemos utilizado 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.

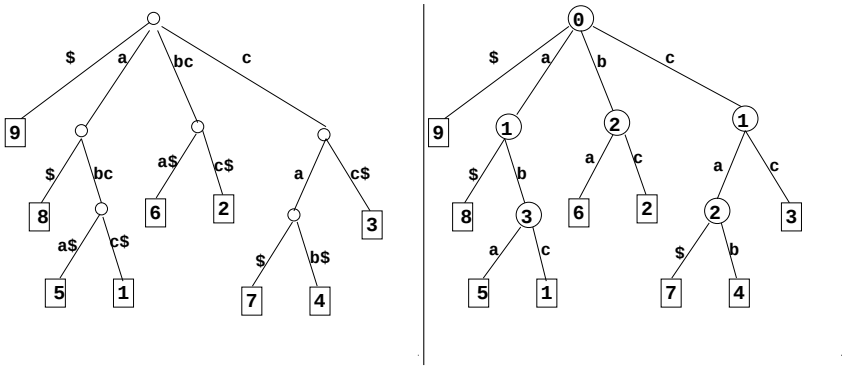

Figura 2. Variantes: concatenación de rótulos (izquierda) y valores de salto (derecha).

3. Representación 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 [6].

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. Para tener una base de comparación, en este trabajo implementamos la versión vinculada de Kurtz, adaptándola a la versión del trie que utiliza valores de salto.

Nuestra propuesta de representación de un trie de sufijos surge como una extensión de la propuesta hecha en [2,5] a árboles r-arios. Dicha representación permitirá por una lado reducir el espacio necesario para almacenar el índice, dado que no existirán los punteros a los hijos, y por otro facilitará un posterior proceso de paginado.

Notar que la información contenida en el trie está compuesta por: la forma 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. Nuestra representación consiste en una representación secuencial de cada una de estas componentes.

La forma del árbol la representamos utilizando la técnica de representación de paréntesis [9]. Esta representación consiste en realizar un barrido preorden sobre el árbol colocando un paréntesis que abre cuando se visita por primera vez un nodo y un paréntesis que cierra cuando se termina de visitar todo el subárbol de ese nodo. Esta representación utiliza un total de $2n$ bits para un árbol de n nodos, manteniendo las facilidades de navegación sobre el mismo [9].

Para la representación de los rótulos de cada rama, de los valores de salto de cada nodo y del grado de cada nodo, utilizamos arreglos colocando los elementos que forman cada arreglo en el orden indicado por un barrido preorden del árbol. Esto permite

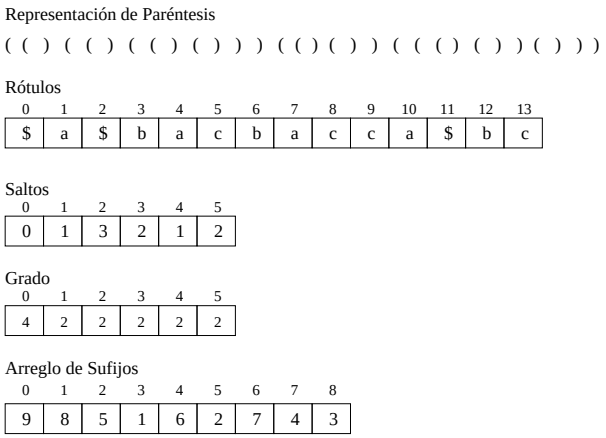

Figura 3. Representación secuencial del trie de sufijos de la Figura 1.

durante la navegación del árbol moverse coordinadamente sobre todas las secuencias que conforman la representación del mismo.

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

Para poder navegar sobre esta representación nos basamos en el algoritmo que permite navegar sobre una representación de paréntesis [9], adaptándolo para movernos de manera coordinada sobre los 5 arreglos. Estos algoritmos necesitan realizar operaciones ***findclose***, ***excess*** y ***enclose*** sobre secuencias binarias [9].

La figura 3 muestra esta representación para el trie de sufijo de la Figura 1.

4. Evaluación Experimental

Recordemos que el objetivo principal de este trabajo fue lograr una representación del trie de sufijos que permita un posterior proceso de paginación en disco. 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 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.

Cuando un índice se maneja en disco, el costo de búsqueda queda determinado por la cantidad de accesos a disco realizadas [11]. Aun así, es importante no descuidar las operaciones que se hacen en memoria principal a fin de lograr un funcionamiento eficiente del índice. Es por esta razón que es necesario evaluar el desempeño en memoria principal de la representación que hemos propuesto.

Hemos implementado el trie de sufijos bajo la representación de Kurtz, que denotaremos con *rk* y la representación secuencial propuesta queda denotaremos con *rs*. La representación *rk* nos da una base que nos permite medir la eficiencia del trie bajo la representación *rs*. Claramente *rk* superará en tiempos de búsqueda a *rs* dado que *rk* ha sido pensada para memoria principal y *rs* ha sido diseñada para memoria secundaria.

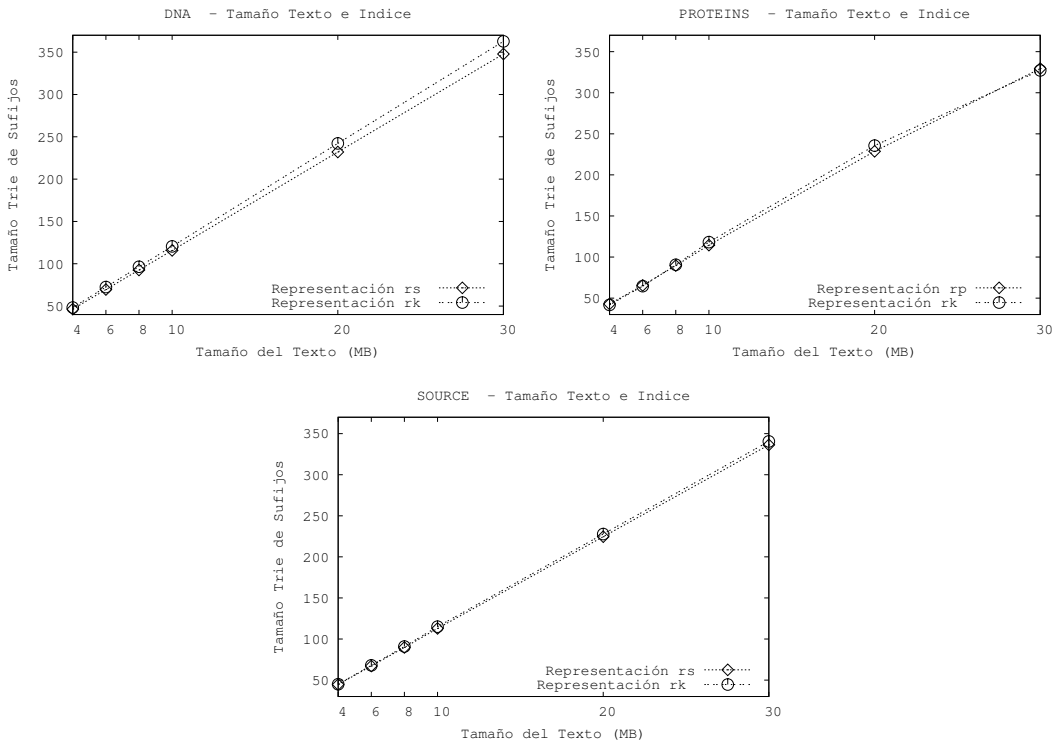

Figura 4. Comparación de tamaño de los índices respecto del tipo y tamaño del texto.

El objetivo de la evaluación aquí presentada es ver cuánto se aleja *rs* de la mejor representación en memoria principal, a fin de poder estimar como se comportará la búsqueda sobre cada una de las partes cuando el trie de sufijos sea paginado en disco.

Para realizar esta evaluación hemos tomado tres tipos de texto: DNA que contiene secuencias de ADN, PROTEINS que contiene secuencias de proteínas y SOURCE que contiene códigos fuente de programas escritos en Java y C. Estos textos han sido 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. Para cada tipo de texto se han tomado partes de tamaño $4, 6, 8$ y 10 MB a fin de poder evaluar la incidencia del tamaño del texto sobre los resultados obtenidos.

Para ambas representaciones hemos evaluado el tamaño del índice y los tiempos de las operaciones *count* y *locate* . Presentamos a continuación los resultados obtenidos. Por cuestiones de espacio mostramos sólo las gráficas que consideramos más relevantes.

4.1. Tamaño del Índice

La figura 4 muestra los resultados con los distintos tipos de texto bajo ambas representaciones. Sobre el eje x se han representado los distintos tamaños de texto y sobre

el eje y está representado el tamaño del índice obtenido. Como puede observarse, la representación *rs* resulta ser la mejor elección para todos los tamaños de texto. En el caso de DNA, se logra reducir el tamaño del índice en un 5 % aproximadamente, en todos los casos. Algo similar sucede con SOURCE logrando en este caso reducciones del $1\,\%$ y $2\,\%$ aproximadamente según el tamaño del texto utilizado.

En cambio para PROTEINS no pasa los mismo. Para texto de 4MB *rk* ocupa menos espacio que *rs* , logrando reducir el espacio en un $2,5\,\%$ aproximadamente; pero a medida aumentamos el tamaño del texto, la diferencia disminuye hasta el punto en que *rs* logra, para texto de 10 MB, ganarle a *rk* reduciendo en un $3\,\%$ aproximadamente el espacio ocupado.

Se observa que a medida que el tamaño del texto aumenta, el comportamiento para SOURCE y DNA se mantine. Sin embargo, esto mismo no sucede con PROTEINS donde se logra reducir el tamaño del índice en un $3\,\%$ aproximadamente, en texto de 20MB, mientras que para texto de 30MB casi no existe diferencias entre los índices.

Es claro que *rs* le gana en espacio a *rk* , en la mayoría de los casos, pero por valores que no son significativos. Sin embargo *rs* tiene una ventaja sobre *rk* : admite algoritmos de compresión sobre algunos de los arreglos involucrados [7]. Recordemos que la representación *rs* utiliza 5 arreglos. Se analizó el porcentaje de incidencia de cada uno de estos arreglos sobre el tamaño total de la representación (no se muestran las gráficas por cuestiones de espacio). Se pudo observar que la mayor cantidad de espacio es utilizada para el arreglo de sufijos subyacente A y para el arreglo que mantiene los valores de salto, luego le siguen los arreglos para mantener el grado de cada nodo, los rótulos y la representación de paréntesis; éste último es el que menos espacio utiliza de los 5 arreglos ocupando sólo un $3{,}60\,\%$ del total del espacio.

El *Directly Addressable Variable-Length Code (DAC*), presentado en [1], es una técnica que permite comprimir una secuencia de códigos de longitud variable permitiendo acceso aleatorio y eficiente a cada uno de ellos. Los autores muestran que esta técnica logra reducciones de alrededor del $30\,\%$ en el espacio requerido para representar la secuencia. Los códigos DAC pueden ser usados para los saltos, los rótulos y los grados. Para el arreglo de sufijos A existen algoritmos específicos de compresión que logran un muy buen desempeño en espacio [10]. Todas estas técnicas permitirían reducir aún más el espacio requerido para representar el trie con *rs*, logrando una diferencia significativa en espacio respecto de *rk*.

4.2. Tiempos de Búsqueda

Presentamos ahora la evaluación experimental de los tiempos de búsquedas. 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 y tipo de texto, se generaron lotes de 500 patrones.

Tiempo medio de *count*

La Figura 5 muestra los resultados obtenidos para ambas representaciones y para los distintos tipos de texto de tamaño 4MB. Sobre el eje x se han representado las distintas

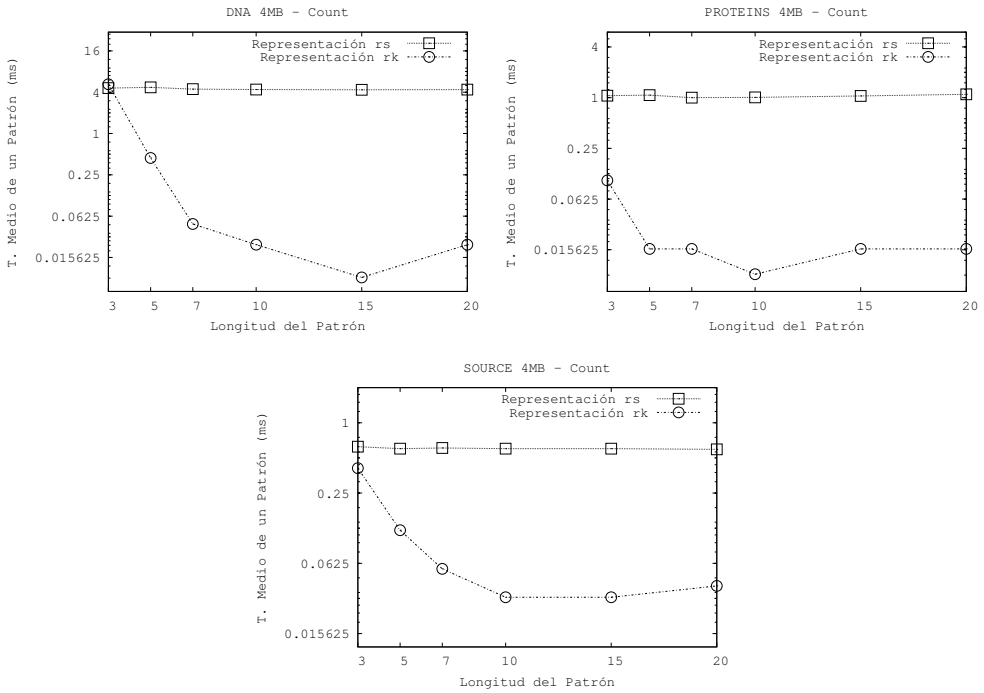

Figura 5. Tiempo medio de *count* para textos de tamaño 4MB

longitudes de patrones y sobre el eje y está representado el tiempo medio, expresado en milisegundos, de realizar la operación *count* sobre un patrón.

Para patrones de longitud 3 las gráficas muestran el mayor acercamiento entre sí: para DNA *rs* mejora el tiempo de *rk* en un $12\,\%$, para SOURCE *rk* mejora a *rs* en un $34\,\%$ y para PROTEINS *rk* mejora el tiempo de *rs* en un $90\,\%$. A medida crece la longitud del patrón los tiempos obtenidos se inclinan significativamente a favor de *rk*.

Estos resultados se mantienen cuando el tamaño del texto crece (no se muestran las gráficas por razones de espacio).

Para entender el por qué de estos resultados debemos tener en cuenta los costos de navegación sobre las representaciones: en *rk* pasar de un nodo a su hermano derecho o a su primer hijo tiene costo $O(1)$, pero en *rs* el costo está dado por las cantidad de operaciones *findclose, excess* y *enclose* realizadas, y esta cantidad aumenta a medida que bajamos en el árbol.

Cuando trabajamos con patrones de longitud 3 las búsquedas se centran en los primeros niveles del trie de sufijos lo que produce que los tiempos obtenidos con ambos índices sean similares; incluso para texto DNA *rs* mejora el tiempo de *rk*. Pero para patrones de mayor longitud, la búsqueda ya no se centran en los primeros niveles y es allí donde *rk* saca ventaja porque mantiene el $O(1)$ para moverse de un nodo a su hermano derecho y a su primer hijo mientras *rs* sólo logra esto en el primer nivel.

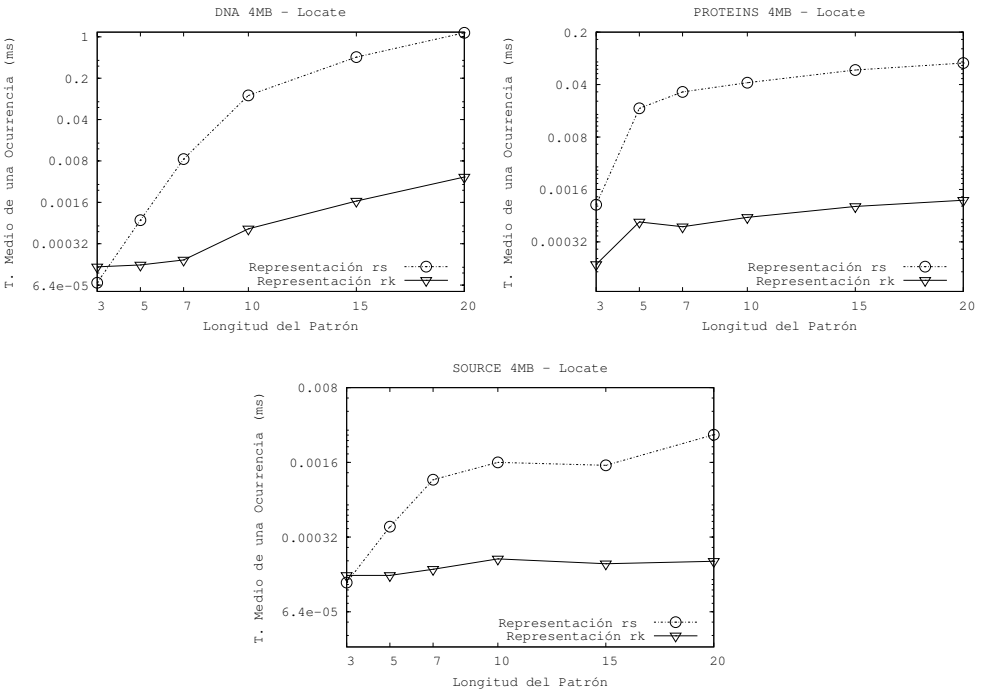

Figura 6. Tiempo medio de *locate* para textos de 4MB.

Tiempo medio de *locate*

La figura 6 muestra los resultados obtenidos para ambas representaciones y para los distintos tipos de texto de tamaño 4MB. Sobre el eje x se han representado las distintas longitudes de patrones y sobre el eje y está representado el tiempo medio de obtener una ocurrencia, expresado en milisegundos.

Para texto DNA y SOURCE donde el patrón es de longitud 3, las gráficas muestran que *rs* mejora el tiempo medio de *rk* en $46\,\%$ y $14,3\,\%$ respectivamente. Pero ya para patrones de longitud 5 los resultados cambian y *rk* mejora el tiempo medio de *rs* en $82,5\,\%$ y $65\,\%$ respectivamente. Y para longitudes mayores esta diferencia va en crecimiento. Para texto PROTEINS *rk* mejora el tiempo medio de *rs* en todos los casos, comenzando de un $84\,\%$ y aumentando a medida crece la longitud del patrón.

Realizar la operación *locate* implica realizar la misma búsqueda que la operación *count* pero debemos retornar las posiciones de todas las ocurrencias en vez de la cantidad de éstas. Por esta razón las conclusiones obtenidas para *count* se mantienen para *locate*, por lo tanto sólo analizaremos los detalles que las diferencian.

Devolver todas las posiciones de ocurrencia de un patrón en *rs* implica indicar el rango correspondiente a las respuestas en el arreglo que mantiene los índices de los sufijos. En cambio, para *rk* , si la búsqueda se detiene en un nodo *n*, debemos recorrer todas las hojas del subárbol que tiene como raíz a *n*. En base a esto, *rs* sería más rápido en retornar todas las posiciones que *rk*, por lo tanto cuando *rs* mejora el tiempo de

búsqueda de *rk* o cuando casi no existe diferencia de tiempo entre las representaciones, *rs* mejora a *rk* con respecto a la operación *locate*, pero cuando *rk* mejora el tiempo en las búsquedas, como pasa en los casos donde el patrón es de longitud mayor a 3, *rk* termina siendo una mejor elección que *rs* .

Ese mismo comportamiento se pudo observar para texto de mayor tamaño (no se muestran las gráficas por razones de espacio).

5. Conclusiones y Trabajo Futuro

En este trabajo hemos abordado el estudio del trie de sufijos. Específicamente nos hemos centrado en el estudio de técnicas de representación de este índice con el objetivo de proponer e implementar una nueva representación del mismo que resulte eficiente en espacio y que permita un posterior paginado del índice.

Las representación propuesta *rs* logra mejorar en espacio a la representación de Kurtz *rk* pero no así en tiempo, pero *rs* tiene la ventaja sobre *rk* de permitir un posterior paginado del índice.

Con respecto al trabajo futuro, nos proponemos implementar la técnica de compresión *Directly Addressable Variable-Length Code (DAC)* [1], para reducir el espacio ocupado por *rs*, analizando la reducción de espacio lograda y el impacto que tiene en los tiempos *count* y *locate* . Posterior a ello, implementaremos una técnica de paginación, rediseñando los algoritmos de creación y búsqueda para esta nueva versión del trie.

Referencias

1. Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. Directly addressable variable-length codes. In *SPIRE*, pages 122–130, 2009.
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. 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. N. Herrera and G. Navarro. Árboles de sufijos comprimidos en memoria secundaria. In *Proc. XXXV Latin American Conference on Informatics (CLEI)*, Pelotas, Brazil, 2009.
6. A. Thomo M. Barsky *, U. Stege. A survey of practical algorithms for suffix tree construction in external memory. In *Software: Practice and Experience*, 2010.
7. V. Mäkinen and G. Navarro. *Compressed Text Indexing*, pages 176–178. Springer, 2008.
8. U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal of Computing*, 22(5):935–948, 1993.
9. J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, 2001.
10. K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Algorithms*, 48(2):294–313, 2003.
11. J. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.
12. P. Weiner. Linear pattern matching algorithm. In *Proc. 14th IEEE Symposium Switching Theory and Automata Theory*, pages 1–11, 1973.