Representación en Memoria Secundaria del Trie de Sufijos

Darío Ruano, Norma Herrera, Carina Ruano, Ana Villegas

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

Resumen 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. Esto implica que un índice construido sobre una base de datos de texto residirá en memoria secundaria y en consecuencia la cantidad de accesos a disco realizados durante el proceso de búsqueda será un factor crítico en la performance del mismo. Un *trie de sufijos* es un índice para este tipo de bases de datos que necesita en espacio 10 veces el tamaño del texto indexado. Si bien existen algoritmos de construcción de un trie de sufijos en memoria secundaria, no se conocen algoritmos para paginar dicho índice. En este artículo presentamos una propuesta de representación de un trie de sufijos y una técnica de paginado del mismo.
Palabras claves: Bases de Datos de Texto, Índices, Memoria Secundaria, Trie.

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, 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.

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.

Resolver la búsqueda de un patrón con algoritmos clásicos tales como Knuth-Morris-Pratt [5] o Boyer-Moore [1] sobre colecciones masivas de texto es ineficiente dado que

estos algoritmos procesan secuencialmente el texto guiándose con un autómata construido en base al patrón P. Estas técnicas logran en algunos casos costos sublineales y sólo son apropiadas cuando se trabaja sobre textos que ocupan varios megabytes. Si el texto es demasiado grande, como es el caso de una base de datos de texto, se hará necesario preprocesar el texto para construir un índice o estructura de datos que permita acelerar el proceso de búsqueda.

Los índices para bases de datos de texto ocupan demasiado espacio, por lo tanto, 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.

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][7]. Esto implica que un índice construido sobre una base de datos de texto residirá en memoria secundaria y en consecuencia la cantidad de accesos a disco realizados durante el proceso de búsqueda será un factor crítico en la performance del mismo. Como es bien conocido, una de las claves para el diseño de índices eficientes en memoria secundaria es la localidad de referencia. Un algoritmo que no explota la localidad de referencia puede ser eficiente en memoria principal, pero degradará notablemente si los datos residen en memoria secundaria y se deja el manejo de las páginas de nuestro índice al sistema operativo (memoria virtual) [9]. Por esta razón, el diseño de técnicas de paginado eficientes es de vital importancia para índices en memoria secundaria.

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 ejemplo, si construimos un trie de sufijos sobre un texto de 10GB, el espacio requerido para almacenarlo será de 100GB. Por esta razón, el diseño de técnicas de representación y de paginado de un trie es de interés en el ámbito de bases de datos de texto. Cabe señalar que si bien existen algoritmos de construcción de un trie en memoria secundaria [6], no existen algoritmos para paginar un trie de manera tal que la búsqueda resulte eficiente en cantidad de accesos a disco.

En este artículo presentamos una propuesta de representación secuencial de un trie de sufijo y una técnica de paginación del mismo, la que surge como una extensión a árboles r-arios de la técnica presentada en [2] para la paginación de un árbol binario.

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 nuestro aporte, proponiendo una nueva representación para el trie de sufijos y una técnica de paginado para el mismo. Finalmente en la sección 5 damos las conclusiones y el trabajo futuro.

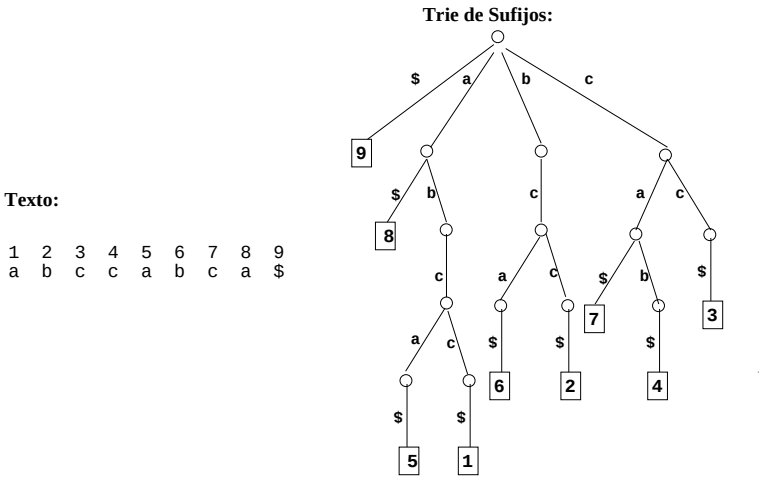

Figura 1.

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 [7], el trie de sufijos [10] y el árbol de sufijos [10] [4]. 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***.

En las siguiente subsecciones explicamos el trie de sufijos y la técnica de paginado, que formaron la base para el desarrollo de la propuesta presentada en este trabajo.

2.1. Trie de Sufijos

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. 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 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.

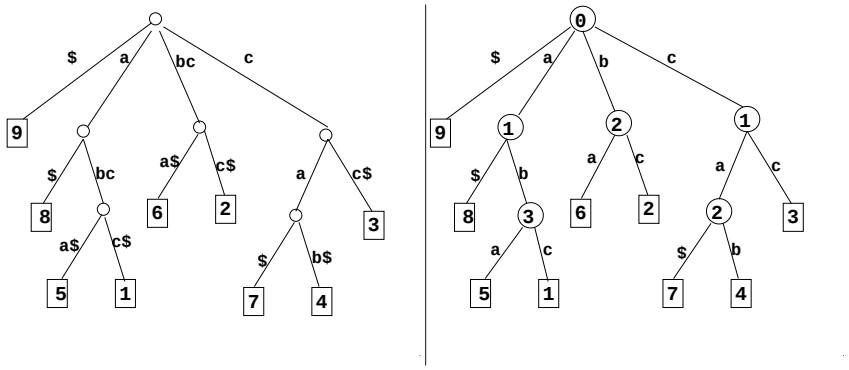

Figura 2. Variantes del trie de sufijos: concatenación de rótulos (izquierda) y valores de salto (derecha).

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 y su correspondiente trie de sufijos. Notar que si recorremos los sufijos del texto según el orden dado por las hojas (de izquierda a derecha) obtenemos todos los sufijos ordenados lexicográficamente. 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. 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. La búsqueda 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 situaciones:

- 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. Árboles de Sufijos en Memoria Secundaria

Un ***árbol de sufijos*** es un Pat-Tree [4] construido sobre el conjunto de todos los sufijos de T. 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.

La técnica de paginación propuesta por los autores del CPT consiste en particionar el árbol en componentes conexas, a las que denomina *partes*, cada una de las cuales se almacena en una página de disco.

El algoritmo utilizado para particionar el árbol de sufijos 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, el algoritmo comienza asignando cada hoja a una parte con profundidad 1 y luego, en forma bottom-up, procesa cada uno de los nodos de este árbol binario según las siguientes reglas:

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

Cerrar una parte implica grabarla en una página de disco y reemplazar, en el árbol original, ese subárbol por el número de página donde fue grabado. Las figuras 3, 4, 5 y 6 muestran la representación gráfica de cada uno de los casos. 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 durante el proceso de búsqueda.

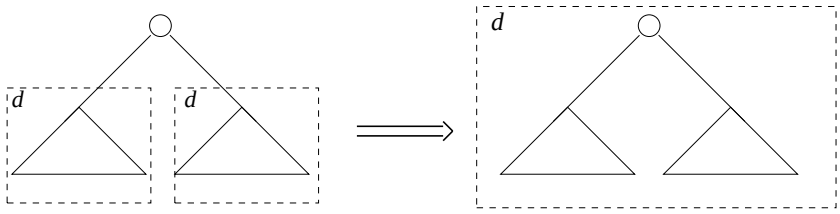

Figura 3. Paginación caso a).

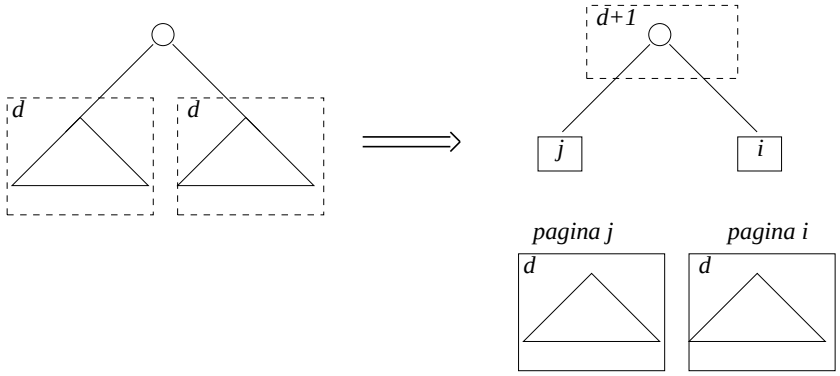

Figura 4. Paginación caso b).

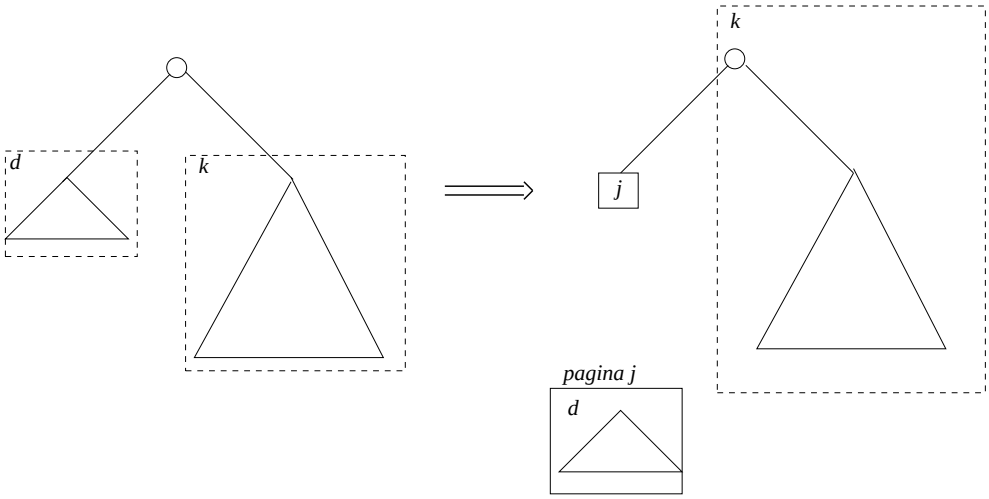

Figura 5. Paginación caso c).

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

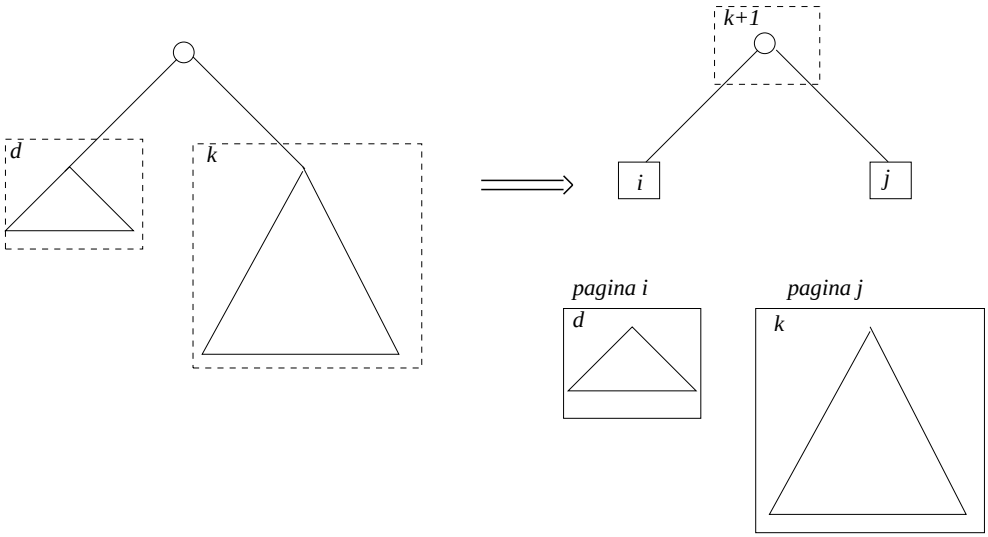

Figura 6. Paginación caso d).

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].

Todas 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).

Nuestra propuesta de representación de un trie de sufijos permite por una lado reducir el espacio necesario para almacenar el índice, dado que no existirán los punteros a los hijos, y por otro facilita un posterior proceso de paginado, que permitirá manejar eficientemente el trie en memoria secundaria (técnica que explicaremos en la próxima sección).

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 [8]. 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 [8]. La figura 7 muestra esta representación para el mismo árbol de la figura 2 (derecha).

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

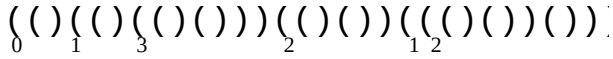

Figura 7. Representación de paréntesis para la forma del árbol. Debajo de algunos paréntesis se ha indicado el valor de salto del nodo que ese paréntesis representa. Los nodos se toman de acuerdo a un barrido preorden.

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.

4. Trie de Sufijos en Memoria Secundaria

Tal como lo mencionáramos en la secciones anteriores, la técnica de paginación que proponemos consiste en una extensión para árboles r-arios del paginado utilizado en el CPT.

Al igual que el algoritmo explicado en la sección 2.2, nuestra técnica de paginado también consiste en particionar el árbol en componentes conexas, denominadas *partes*, cada una de las cuales se almacena en una página de disco.

El algoritmo procede en forma bottom-up tratando de condensar en una única parte un nodo con uno o más subárboles que dependen de él. En este proceso de particionado las decisiones se toman en base a la profundidad de cada nodo involucrado (recordar la definición de profundidad dada en la sección 2.2), dado que la profundidad indica la cantidad de accesos a disco que deberá realizar el proceso de búsqueda para llegar desde esa parte a una hoja del árbol.

Para particionar un árbol, el algoritmo comienza asignando cada hoja a una parte con profundidad 1 y luego, en forma bottom-up, procesa cada uno de los nodos de este árbol r-ario según las siguientes reglas:

a) se ordenan los hijos del nodo corriente de mayor a menor según su profundidad.
b) si el nodo corriente y el hijo de mayor profundidad d entran en una página de disco, colocamos en una misma página el nodo corriente y tantos hijos como entren en una página. Estos hijos se toman de acuerdo al orden establecido en el paso a). Se establece la profundidad de la nueva parte creada en d, donde d es el máximo de las profundidades de los hijos, y se cierran las partes de aquellos hijos que no conforman la nueva parte creada.
c) si el padre no puede unirse con el hijo de mayor profundidad, se cierran todas las partes hijas y se crea una nueva parte para el nodo corriente con profundidad $d + 1$, donde d es el máximo de las profundidades de los hijos.

5. Conclusiones y Trabajo Futuro

Basándonos en los resultados que se muestran en [2] hemos diseñado una técnica de paginado del trie de sufijos, índice para el cual no se conoce ninguna versión eficiente en memoria secundaria. Si bien la propuesta aquí presentada se describe para un trie de sufijos, puede ser fácilmente extendida para la representación en memoria secundaria de cualquier árbol r-ario.

Como trabajo futuro nos proponemos estudiar experimentalmente las técnicas aquí propuestas.

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. 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. D. E. Knuth, J. H. Morris, and V. R.Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):323–350, 1977.
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. U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal of Computing*, 22(5):935–948, 1993.
8. J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, 2001.
9. J. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.
10. P. Weiner. Linear pattern matching algorithm. In *Proc. 14th IEEE Symposium Switching Theory and Automata Theory*, pages 1–11, 1973.