Tries, arreglos de sufijos y árboles de sufijos

Mikel Dalmau

8 de Junio de 2015

Resumen

En el siguiente documento se exponen algunas de las estructuras de datos más utilizadas para el tratamiento de strings o cadenas de caracteres, como son los Tries o los árboles de sufijos, los algoritmos para llevar al cabo búsquedas eficientes, mejoras en el uso de espacio, y los ámbitos de uso mas comunes de estas estructuras.

1 Tries

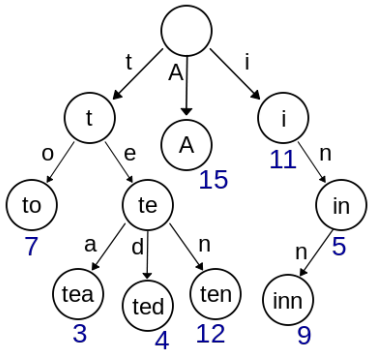


Figura 1: Un trie de las claves "A", "to", "tea", "ted", "ten", "i", "in", y "inn".

Introducidos en 1959 independientemente por Rene de la Briandais1 y Edward Fredking, un trie es una estructura de datos de tipo árbol que permite la recuperación de información (de ahí su nombre del inglés reTRIEval). La información almacenada en un trie es un conjunto de claves, donde una clave es una secuencia de símbolos pertenecientes a un alfabeto. Las claves son almacenadas en las hojas del árbol y los nodos sirven para guiar la búsqueda.

El árbol se estructura de forma que cada letra de la clave se sitúa en un nodo de forma que los hijos de un nodo representan las distintas posibilidades de símbolos diferentes que pueden continuar al símbolo representado por el nodo padre. Por tanto la búsqueda en un trie se hace de forma similar a como se hacen las búsquedas en un diccionario:

Se empieza en la raíz del árbol. Si el símbolo que estamos buscando es A entonces la búsqueda continúa en el subárbol asociado al símbolo A que cuelga de la raíz. Se sigue de forma análoga hasta llegar al nodo hoja. Entonces se compara la cadena asociada a el nodo hoja y si coincide con la cadena de búsqueda entonces la búsqueda ha terminado en éxito, si no entonces el elemento no se encuentra en el árbol.

Por eficiencia se suelen eliminar los nodos intermedios que sólo tienen un hijo, es decir, si un nodo intermedio tiene sólo un hijo con cierto carácter entonces el nodo hijo será el nodo hoja que contiene directamente la clave completa, se analiza este cambio en el apartado 1.2.

Año: 2014 - 2015
Profesor(a): Jesus Bermudez De Andres
Teléfono:
E-mail: mdalmau002@ikasle.ehu.es

GIPUZKOAKO CAMPUSA
CAMPUS DE GUIPUZCOA
Paseo Manuel de Lardizabal, 1
20018 Donostia (Gipuzkoa)

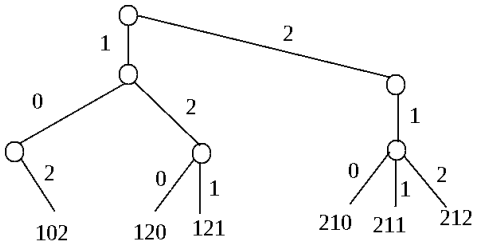
1.1 Definición formal

Un trie es un caso especial de autómata finito determinista (S, Σ, T, s, A), que sirve para almacenar un conjunto de cadenas E en el que:

- Σ es el alfabeto sobre el que están definidas las cadenas.
- S, el conjunto de estados, cada uno de los cuales representa un prefijo de E.
- La función de transición: T : $S \times \Sigma \rightarrow \mathcal{S}$; está definida como sigue: $T(x, \sigma) = x\sigma$ si $x, x\sigma \in S$, e indefinida en otro caso.
- El estado inicial s corresponde a la cadena vacía λ.
- El conjunto de estados de aceptación A ⊆ S es igual a E.

1.2 Análisis y compresión de la estructura

La situación general es la siguiente, el universo U esta compuesto de todos los strings de longitud l sobre un alfabeto de digamos n elementos, U = $\{O, ..., k-1\}^l$. Un conjunto S ⊆ U está representado como un árbol k-ario compuesto por todos los prefijos de los elementos de S.



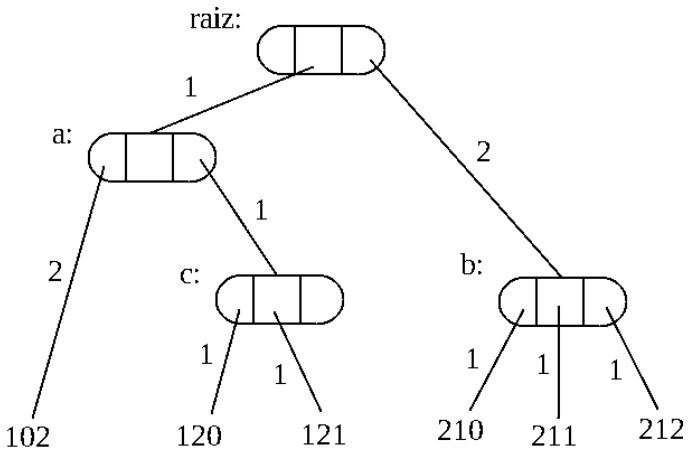
La implementación más sencilla y que antes nos vendría a la mente sería utilizar un array de longitud n para cada nodo interno del árbol. La operaciones Acceder, Insertar y Eliminar serían muy rápidas y sencillas de programar. En particular si guardáramos el inverso de cada elemento de S (en el ejemplo sería {121, 201, 112, 021, 012, 212}), entonces el siguiente programa serviría para realizar la operación:

```
Acceder(x)
    v ← raiz;
    y ← x;
    hacer l veces ( i , y) ← (y mod k, y div k);
        v ← i-ésimo hijo de v
    fin;
    si x = CONTENIDO[v] entonces "Sí" sino "no"
```

Este programa toma tiempo O(l) = O($\log_k N$) donde N = |U|. Desafortunadamente, el espacio requerido por un Trie como el descrito arriba puede ser enorme, del orden O(n·l·k). Esto es, para cada elemento del conjunto S, |S| = n, deberíamos guardar todo el recorrido de

nodos, hasta l nodos, y cada uno de estos tendría grado 1 y utilizaría espacio O(k).

Existe una forma más simple de reducir el espacio requerido a O(n·k), eliminando el término l. Consiste en eliminar los nodos de grado 1 y guardar nodos internos de grado 2 al menos. Puesto que un trie para un conjunto S de tamaño n tiene n hojas habrá como mucho n-1 nodos internos de grado 2 o más. Las cadenas de nodos de grado 1 se sustituyen por un único número, el número de nodos en la cadena. Si transformamos el ejemplo anterior obtenemos:



Aquí los nodos internos están representados como arrays de longitud 3. En las flechas de padre a hijo está indicado el aumento en la profundidad del árbol, esto es 1 + el número de nodos eliminados. En el ejemplo el número 2 en la flecha que sale de la raíz, indica que en el hijo(b) hay que ramificar el tercer dígito (1 + 2). El algoritmo para Acceder, Insertar y Eliminar se complica ligeramente para Tries comprimidos pero sigue manteniendo el mismo orden.

Lo recién expuesto demuestra el siguiente teorema:

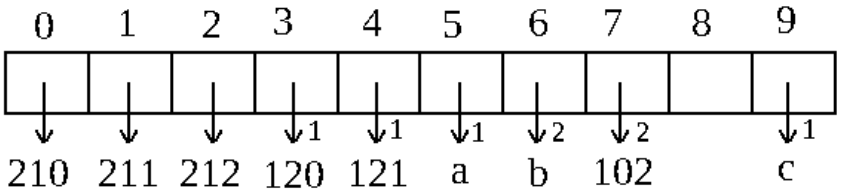
Teorema: *Un Trie comprimido permite las operaciones Acceder, Insertar y Eliminar en tiempo O($\log_k N$), donde N es el tamaño del universo y k es el factor de ramificación del Trie. Un conjunto S de n elementos requiere espacio O(n·k).*

Cabe destacar el interesante equilibrio espacio-tiempo que exhiben los Tries. Elegir un factor de ramificación k alto hará más rápido el Trie pero más costoso en términos de memoria, en cambio elegir un k pequeño hará más lenta la búsqueda pero requerirá menos memoria.

1.3 Tries estáticos

En esta sección se muestra a grandes rasgos como es posible reducir drásticamente el uso de memoria sin aumentar el tiempo de acceso. Los tries estáticos solo permiten la operación Acceder. La reducción se realiza en dos pasos. En el primer paso se utiliza una técnica general de solapamiento que sirve para comprimir tablas dispersas(sparse). Este primer paso reduce el espacio requerido a O(n log log n). En un segundo paso se reduce el espacio requerido hasta O(n) apilando varios números en un único espacio de memoria.

En nuestro ejemplo utilizaríamos un array de longitud 10 para almacenar los cuatro arrays de longitud 3 correspondientes a los nodos, y una tabla adicional para almacenar en que posición comienza de cada nodo.



a	b	c	raíz
7	0	3	4

Supongamos que buscamos el 121. Seguimos el primer apuntador (1) partiendo desde la raíz, el apuntador en la posición 4 + 1 en el array grande. Este nos lleva al nodo a. Seguimos el apuntador (2) desde a, es la posición 7 + 2 del array mayor que nos lleva al nodo c. Finalmente seguimos el apuntador 1 desde c que nos lleva a la posición 3 + 1, es una hoja, comparamos los valores y vemos que hemos hallado el 121.

Supongamos ahora que buscamos el 012, que no se halla en el Trie, seguimos el apuntador 0 desde la raíz, este nos lleva a la posición 4 + 0 que es una hoja, comparamos los contenidos y 121 $\neq$ 012 por lo que 012 no es miembro del conjunto.

Describiré ahora con algo más de detalle la técnica de compresión utilizada. Sea A una matriz de r por s compuesta por ceros y unos con exactamente m entradas iguales a 1. Queremos hallar un pequeño desplazamiento de filas fd(i) tal que si guardamos la i-ésima fila de la matriz A en un array uni-dimensional en la posición fd(i) ningún par de unos colisione. En nuestro ejemplo los 0s corresponden a apuntadores nulos y los 1s a apuntadores no nulos. Un método para computar este desplazamiento de filas es el método decreciente del primero que mejor se ajusta(first-fit decreasing method).

1. Ordena las filas de forma decreciente en función del número de unos. La fila con el número máximo de unos es la primera.

2. Fija fd(0) = 0. Para i $\geq$ 1 elige rd(i) $\geq$ 0 mínimo tal que no ocurran colisiones con las filas colocadas previamente 0 a i - 1.

En nuestro ejemplo el paso 1 podría producir la siguiente matriz.

b:	1	1	1
c:	1	1	0
raíz:	0	1	1
a:	1	0	1

0	1	2	3	4	5	6	7	8	9
1	1	1							
			1	1	0				
				0	1	1			
							1	0	1

En el paso 2 elegiríamos fd(0) = 0, fd(c) = 3, fd(raíz) = 4 y fd(a) = 7. Estas elecciones están ilustradas por la segunda matriz.

Para más detalle sobre Tries estáticos y una demostración completa de las técnicas expuestas y las demostraciones de los teoremas que las respaldan consultar [MEHL] apartado III. 1.2 STATIC TRIES or Compressing Sparse Tables.

1.4 Ventajas

Las ventajas principales de los tries sobre los árboles de búsqueda binaria (BST) son:

- búsqueda de claves más rápida. La búsqueda de una clave de longitud m tendrá en el peor de los casos un coste de O(m). Un BST tiene un coste de O(mlog n), siendo n el número de elementos del árbol, ya que la búsqueda depende de la profundidad del árbol, logarítmica con el número de claves.
- menos espacio requerido para almacenar gran cantidad de cadenas pequeñas, puesto que las claves no se almacenan explícitamente
- mejor funcionamiento para el algoritmo de búsqueda del prefijo más largo.

1.5 Aplicaciones

Un Trie puede usarse para:

- **Reemplazar una Tabla hash**, sobre la que presenta ventajas como que en un trie no se producen colisiones de claves, o no hay que definir una función de hash, o modificarla si añadimos más claves.
- **Como representación de diccionarios** Una aplicación frecuente de los tries es el almacenamiento de diccionarios, como los que se encuentran en los teléfonos móviles, cuando se necesita almacenar información adicional sobre las palabras, sino, un autómata finito determinista acíclico mínimo usa menos espacio que un trie.
- **Algoritmos de correspondencia aproximada**, como los usados en el software de corrección ortográfica.

2 Árbol de sufijos y array de sufijos

2.1 Árbol de sufijos

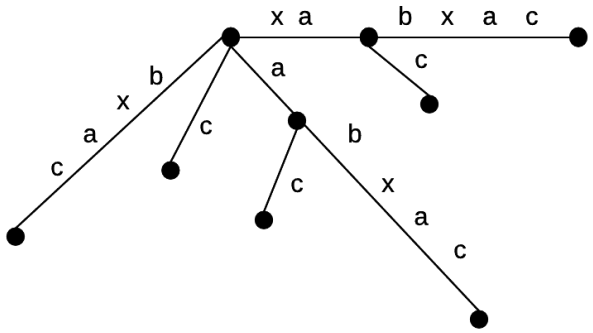
Un árbol de sufijos representa la estructura interna de un string de una forma más profunda que otros algoritmos de preprocesamiento y de búsqueda de patrones. Un árbol de sufijos puede usarse para resolver el problema de búsqueda de la palabra exacta en tiempo linea, pero lo que lo hace interesante es su capacidad de resolver en tiempo lineal problemas de strings más complejos como la búsqueda inexacta o la búsqueda de la subcadena.

Dado un texto T de longitud m, tras O(m) tiempo de preprocesamiento, un árbol de sufijos es capaz de determinar en tiempo lineal O(n) si un string S de tamaño n está presente en T o no. Esto es, el coste de determinar si una palabra aparece en un texto depende de la longitud de la palabra y no del tamaño del texto, una vez realizado el preprocesamiento.

2.1.1 Definición

Un árbol de sufijos T para un string S de m caracteres es un árbol dirigido con exactamente m hojas numeradas de 1 a m. Cada nodo interno, distinto de la raíz, tiene al menos dos hijos y cada arista está etiquetada con una subcadena no vacía de S. Dos aristas salientes de un mismo nodo no pueden tener subcadenas con el mismo caracter de comienzo. La funcionalidad clave del árbol de sufijos es que para cada hoja i, la concatenación de las etiquetas de las aristas desde la raíz a la hoja es el sufijo de S que comienza en la posición i, esto es, $S[i..m]$.

La siguiente imagen muestra un árbol de sufijos para la cadena *xabxac*:



Según la definición de arrriba no está garantizado que exista un árbol de sufijo para cualquier string S. El problema es que si un sufijo de S coincide con el prefijo de otro sufijo de S, entonces no es posible construir un árbol que respeta la definición expuesta ya que el camino del primer sufijo no terminaría en una hoja. Se puede evitar este problemas utilizando un caracter que no pertenezaca al alfabeto a modo de terminación.

2.1.2 Aplicaciones

- Hallar la cadena exacta.
- Subcadena común más larga entre dos strings.
- Comprimir árboles de sufijos en DAGs, principalmente para reducir el uso del espacio.
- Hallar la coincidencia entre sufijo-prefijo más larga para cada par S_i, S_j en una colección de strings S.
- Reconocimiento de ADN contaminado.
- Reconocimiento de estructuras repetitivas en ADN y cadenas bilógicas.

2.2 Array de sufijos

Los arrays de sufijos fueron introducidos por Manber y Myers (1990) como una simple variante eficiente en espacio a los árboles de sufijos.

Un array de sufijos es un array ordenado de todos los sufijos de una cadena dada. Esta estructura de datos es muy simple, pero sin embargo es muy poderosa y es usada en algoritmos de compresión de datos y dentro del campo de la bioinformática , indización de textos completos, entre otros.

2.2.1 Definición

Sea $S = s_1, s_2, ...s_n$ una cadena y sea $S[i, j]$ la subcadena de S que va desde el índice i hasta j. El array de sufijos A de la cadena S va a ser un array de enteros brindando las posiciones iniciales de los sufijos de S en orden léxico.Esto significa que $A[i]$ contiene la posición inicial del i-esimo sufijo más pequeño en S y por tanto se cumple que para todo $1 < i \leq n : S[A[i-1], n] < S[A[i], n]$. Esto es, el sufijo en la posición $Pos(1)$ de S será el de menor valor léxico y en general el sufijo $Pos(i)$ de S será léxicamente menor que el sufijo en $Pos(i+1)$

Al igual que en el caso de los árboles es conveniente añadir un símbolo de fin de string como puede ser \$ al final de S, y ahora interpretaremos que es de menor valor léxico que cualquier otro caracter en el alfabeto.

El siguiente es un ejemplo de un array sufijo de la palabra *mississippi*:

11: i
8: ippi
5: issippi
2: ississippi
1: mississippi
10: pi
9: ppi
7: sippi
4: sissippi
6: ssippi
3: ssissippi

El array sufijo *Pos* es 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3.

Cabe destacar que el array solo contiene enteros y por lo tanto no almacena información sobre el alfabeto usado en la cadena S, por lo tanto el espacio requerido por una cadena de tamaño m es modesto y puede ser almacenado en exactamente m palabras de memoria, asumiendo que el tamaño de una palabra es al menos log m bits.

2.2.2 Transformas árbol de sufijos en array de sufijos en tiempo lineal

Suponiendo que disponemos del espacio suficiente para construir un árbol para S, un array de sufijos puede ser obtenido a partir de un árbol T realizando una búsqueda "léxica" en profundidad de T.

Definimos una arista $(v,\ u)$ como léxicamente menor que otra arista $(v,\ w)$ si y solo si el primer caracter de $(v,\ u)$ es léxicamente menor que el primer caracter de $(v,\ w)$. (El caracter de fin de string \$ es léxicamente menor que cualquier otro).

Puesto que no hay dos aristas salientes del mismo nodo con el mismo caracter de comienzo, siempre existe un orden léxico estricto entre las aristas salientes de un nodo. Este orden implica que el camino desde la raíz de T hasta la primera hoja que se alcance representará el sufijo de menor valor léxico. Por lo tanto el array de sufijos *Pos* es una lista ordenada de los números de sufijos hallados en las hojas de T durante la búsqueda en profundidad.

Dado que el árbol T se construye en tiempo lineal y el recorrido se realiza también en tiempo lineal, podemos concluir que el array *Pos* se construye en tiempo lineal $O(m)$ siendo m el número de caracteres.

2.2.3 Reconocer patrones utilizando un array de sufijos

El array de sufijos para una cadena S permite con un simple algoritmo hallar todas las ocurrencias de cualquier patrón P en S. La clave del algoritmo se basa en una propiedad que dice que si P está en S entonces todas las posiciones en las que ocurre P estarán agrupadas consecutivamente en Pos.

Por ejemplo, $P = issi$ ocurre en *mississippi* comenzando en la posición 2 y 5, que son adyacentes en Pos (mirar ejemplo de anterior). Por lo tanto la búsqueda de ocurrencias de P en S se reduce a una búsqueda binaria a lo largo del array de sufijos. En mayor detalle, supongamos que P es léxicamente menor que el sufijo en la posición del medio de S ($Pos[m/2]$). En ese caso, de hallarse P en S, está necesariamente en la primera mitad de Pos.

Utilizando la búsqueda binaria puede hallarse el menor índice i en Pos(si existe) tal que P coincide exactamente con los n primeros caracteres del sufijo $Pos(i)$. De manera similar puede hallarse el mayor índice i' con dicha propiedad, entonces el patrón P sucederá en S y comenzará en cada posicion dada por $Pos(i)$ hasta $Pos(i')$.

Teorema *Utilizando la búsqueda binaria sobre el array Pos, pueden hallarse todas las ocurrencias de P en S en tiempo O(n log m).*

2.2.4 Mejora del algoritmo de reconocimiento de patrones

Según se lleva a cabo la búsqueda binaria, denotemos por L y R los extremos del intervalo de búsqueda actual. Al inicio L será 1 y R será igual a m. En cada iteración de la búsqueda se consulta la posición $M = \lceil (R + L)/2 \rceil$ del array Pos. El algoritmo de búsqueda guarda un registro de los prefijos más largos de $Pos(L)$ y $Pos(R)$ que coinciden con algún prefijo de P, que denotaremos por l y r respectivamente, y sea $mlr = \min(l, r)$.

Pueden utilizarse los valores mlr para acelerar las comparaciones léxicas de P y $Pos(M)$. Ya que que los sufijos de S están ordenados en el array Pos, si i es un índice entre L y R, los primeros mlr caracteres del sufijo $Pos(i)$ son necesariamente iguales que los primeros mlr caracteres del sufijo $Pos(L)$ y por lo tanto de P. Esto quiere decir que a la hora de comparar P con $Pos(M)$ no es necesario empezar comparando desde la primera posición sino desde la posición $mlr + 1$.

Mantener mlr a lo largo de la búsqueda binaria añade ligeros costes adicionales al algoritmo pero evita realizar muchas comparaciones redundantes. Al comienzo de la búsqueda, cuando $L = 1$ y $R = m$, es neceario compara P con los sufijos $Pos(1)$ y $Pos(m)$ para hallar l, r y mlr. Sin embargo en el caso peor el algoritmo sigue siendo de orden O(n log m).

2.2.5 Acelerando todavía más el algoritmo de reconocimiento de patrones

Decimos que la examinación de un caracter es redundante cuando este ha sido examinado antes. El objetivo de la aceleración es reducir el número de comparaciones redundantes a como máximo una por cada iteración de la búsqueda binaria. El uso de *mlr* por sí solo no es suficiente para alcanzar este objetivo. Puesto que *mlr* es el mínimo entre l y r, cuando $l \neq r$, todos los caracteres de P desde *mlr* +1 hasta el máximo de l y r ya habrán sido examinados, y por lo tanto la comparación de dichos caracteres será redundante. Lo que es necesario es una manera de comenzar las comparaciones en el máximo de l y r.

Definamos $Lcp(i,j)$ (Longes common prefix) como la longitud del prefijo más largo común a los sufijos en las posciones i y j de *Pos*.

Por ejemplo, si $S = mississippi$, el sufijo $Pos(3)$ es *issippi* y el sufijo $Pos(4)$ es *ississippi*, por lo tanto $Lcp(3,4) = 4$.

Para acelerar la búsqueda el algoritmo utiliza $Lcp(L,M)$ y $Lcp(M,R)$ para cada trio (L,M,R) que surge durante la búsqueda binaria.

De esta manera es posible mejorar el órden del algoritmo a O(n + log m).

Para una demostración formal y más detalle sobre el uso de Lcp y como obtenerlo consultar [GUSFIELD] apartados 7.14.4, 7.14.5.

2.2.6 Aplicaciones

Ha sido demostrado todo algoritmo de árbol de sufijos puede ser sistemáticamente reemplazado con un algoritmo que use un array de sufijos unido con información adicional (como un array de prefijos comunes) y resuelve el mismo problema y con la misma complejidad temporal.

Bibliografía

[NAVRAF] GONZALO NAVARRO, MATHIEU RAFFINOT, Flexible pattern matching in strings

[MEHL] KURT MEHLHORN, Data Structures and Algorithms 1: Sorting and Searching

[GUSFIELD] DAN GUSFIELD, Algorithms on strings, trees, and secuences

[Wiki] http://es.wikipedia.org/wiki/Arreglo_de_sufijos

[Wiki] http://es.wikipedia.org/wiki/Trie