IIC2143 - T01

Leonardo Olivares

14/09/18

1 Implementación del Trie

La tarea consitió en implementar una estructura de datos, conocida como Trie, con el fin de poder autocompletar frases a partir de un prefijo.

Para lograr que el tiempo requerido para procesar una consulta de largo k fuera O(k), se realizó la siguiente implementación del Trie.

Inicialmente, se cuenta con un nodo vacío, que representa la raíz de la estrucutura.

Cada nodo dentro del Trie, cuenta con un valor "**character**" (char), en donde almacena un caracter, un arreglo "**childs**" de largo 27, que contiene punteros a otros nodos (inicialmente vacío), un entero "**max_frequency**", un entero "**next_child**" y un booleano "**is_phrase**".

El nodo principal no almacena ningún caracter. Al insertar una frase, ocurre el siguiente proceso.

- Se comienza en el nodo raíz. Se obtiene el valor *ASCII* del primer caracter de la frase y se traduce a un número entre 0 y 26.
- Por medio del número obtenido, se revisa la posición en el arreglo A del nodo. Si existe un puntero a otro nodo en esa posición, se revisa.
- En caso de que no exista un puntero en esa posición se crea, por medio de memoria dinámica.
- Antes de revisar el próximo nodo, se revisa la frecuencia (max_frequency) de la frase a ingresar. Si la frecuencia es mayor al valor "max_frequency" del nodo, dicho valor cambia al de la frecuencia. Además, si esto ocurre, se almacena el número correspondiente a la posición del arreglo del nodo con el próximo caracter.
- Al ingresar el último caracter de la frase, se almacena un booleano en el nodo, que indica el término de una frase.

El proceso anterior ocurre, para cada frase *database.txt*.

La clave para lograr realizar las consultas en tiempo O(m), se centra en el valor de "next_child" que se almacena en cada nodo.

Esencialmente, "next_child" indica cual es la ruta que el algoritmo debe tomar, para llegar a la frase con mayor frecuencia. Entonces, al ingresar un prefijo, simplemente se busca el nodo que contenga al último caracter, por medio del valor ASCII de cada uno. Una vez en ese nodo, el valor de "next_child" contendrá el índice del arreglo "childs", en donde se encuentra el próximo nodo que forma parte de la ruta para llegar a la frase más frecuente de esa rama.

Por lo tanto, el algoritmo es capaz de seguir las indicaciones de cada nodo, según corresponda, de manera que tarda "k" pasos es conseguir la frase, siendo "k" el largo de la frase.

2 Construcción del Trie

Considerando la implementación descrita anteriormente, la complejidad de construir el Trie, en términos de "n" (cantidad de frases en la base de datos), debe ser O(n).

Esto se debe a que cada frase tiene como máximo 100 caracteres, por lo que, se realizan (100 * n) operaciones en el peor de los casos. Por lo tanto, es de complejidad O(n).

A continuación se muestra un gráfico Tiempo(n), con los tiempos obtenidos a partir de la implementación mencionada. Solo se toma en cuenta el tiempo para la construcción del Trie. Los "m" utilizados fueron todos a partir del *large/database.txt*.

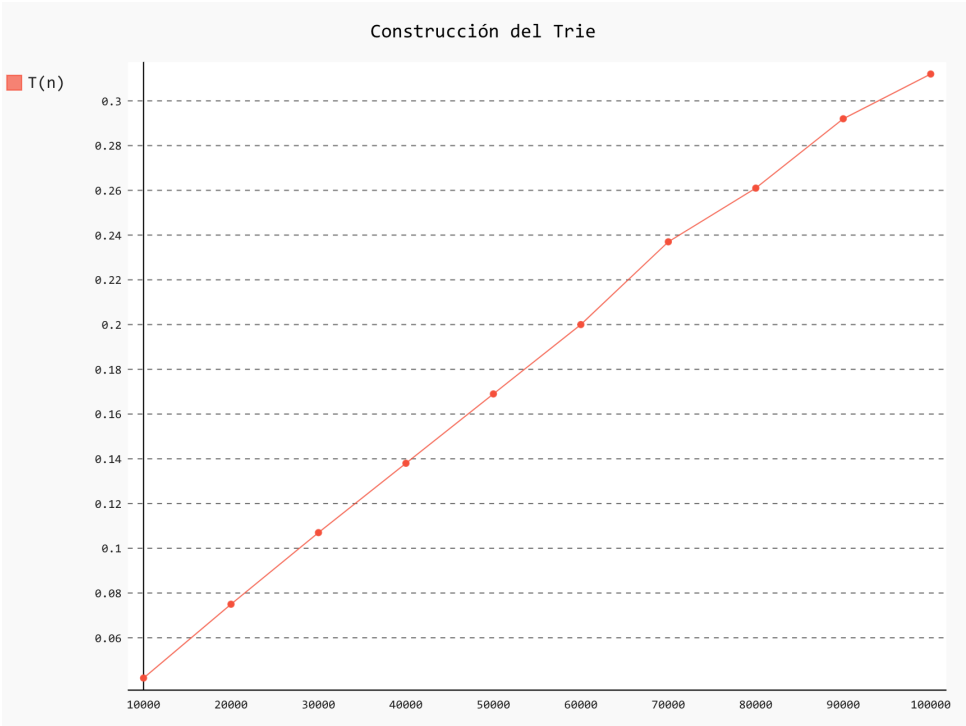

Figure 1: Tiempo (seg) vs N

Se puede observar que el gráfico se comporta de manera esperada. A medida que la cantidad de frases iniciales aumenta, el tiempo para construir el Trie aumenta de manera lineal, con una pendiente constante.

3 Consultas en el Trie

Dada la implementación descrita inicialmente, es posible acceder a la frase mas popular de una rama, en un tiempo O(k), donde k es el largo de la frase.

Por esta razón, si inicialemente tenemos "m" consultas, en el peor de los casos, el largo máximo de cada frase es de 100 caracteres. Por lo tanto, el tiempo esperado para realizar todas la consultas en O(100*m), es decir O(m).

A continuación se muestra un gráfico con Tiempo(m), con los tiempos obtenidos a partir de la implementación mencionada. Solo se muestra el tiempo de las consultas, es decir, al tiempo total, se le restó el tiempo promedio en crear el Trie, para el *large/database.txt*

Se puede observar en la figura 2, que al aumentar la cantidad de consultas a realizar, aumenta el tiempo de ejecución. El aumento de tiempo ocurre de manera lineal con respecto a la cantidad de consultas.

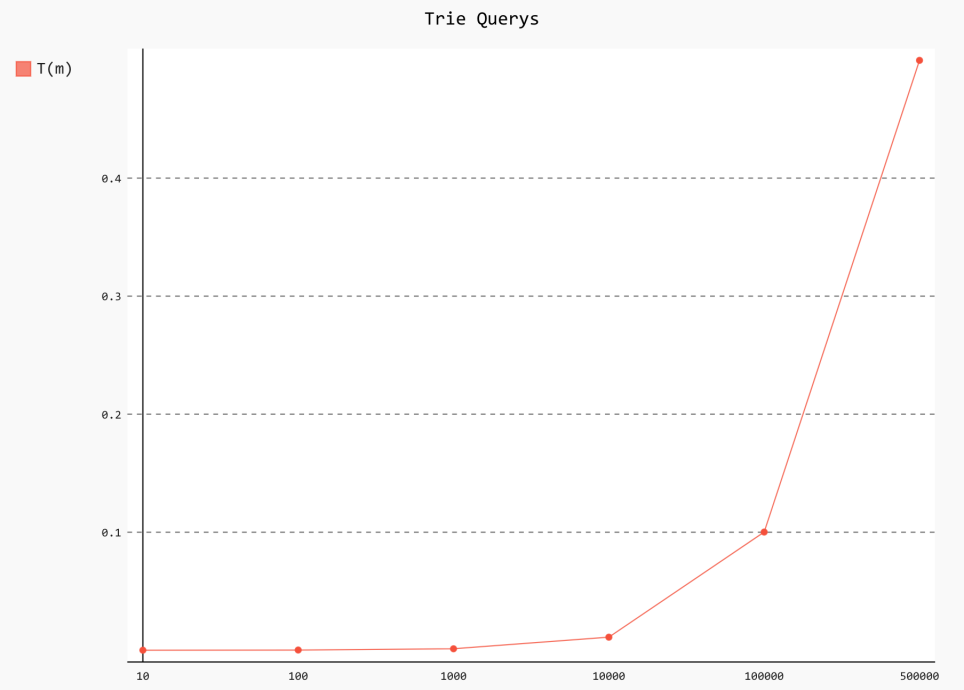

Figure 2: Tiempo (seg) vs M

Por ejemplo, para realizar 100.000 consultas, se toma un tiempo aproximado de 0.1 segundos. Al quintuplicar esa cantidad de consultas, es decir 500.000, el tiempo de ejecución termina siendo 5 veces mayor al mencionado anteriormente, es decir, 0.5 segundos aproximademente.

4 Bonus. Radix Tree

Existe un estructura de datos, conocida como Radix Tree, que representa una versión compacta de un Trie, debido a que es capaz de ahorrar memoria, evitando crear nodos innecesarios en la estructura.

La implementación en general es parecida al Trie. Sin embargo, cuando se inserta una frase o palabra, de ser posible, en lugar de crear un nodo por caracter, almacena nodos con substrings (de largo mayor a 1).

Por ejemplo, al insertar una frase, se busca el nodo de inserción, y se agrega el resto de la frase en un solo nodo. Luego, si se inserta una frase similar, el nodo se separa en dos, uno que almacena lo que tienen ambas frases en común y otro que contiene el resto de la frase mas larga.

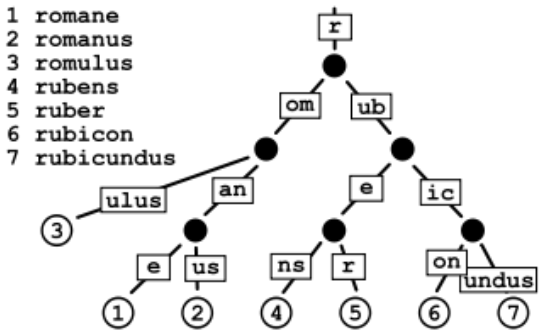

Figure 3: Radix Tree

Se puede observar en la figura 3, que Radix Tree toma en cuenta dos casos, que permiten ahorrar memoria, en comparación al Trie común.

Primero, el Radix Tree evita dejar **nodos redundates**, es decir, aquellos que solo tienen un hijo. Por otro lado, evita **cadenas redundantes**, compuesta por nodos redundantes. Esto ocurre debido a que en un Trie, existen ramas con una sola opción de frase, sin embargo se utiliza memoria inicializando cada nodo.

Para lograr lo anterior, el algoritmo se encarga de no dejar nodos con solo un hijo. En caso de existir los comprime en un solo nodo, evitando inicializar más memoria de la necesaria.