Pontificia Universidad Católica de Chile
Departamento de Ciencia de la Computación
Estructuras de Datos y Algoritmos — IIC2133

Tarea 1

13 de Agosto, 2018

Diego Iruretagoyena - 14619164

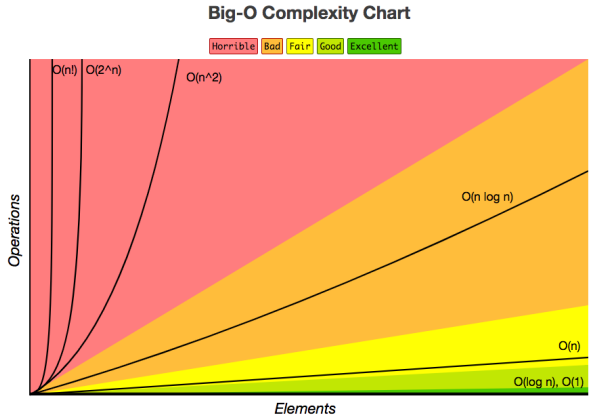
Introducción

A medida que hemos avanzado en el ramo, vamos cada vez aprendiendo mejores estructuras para determinados usos, notando que no existe una estructura que satisfasga todos los problemas, si no que debemos analizar el problema y tomar una decision consecuente con nuestro objetivo.

Cuando estabamos aprendiendo sobre las distintas estructuras de datos, nos fijamos en los pros y contras de cada estructura, con la intencion de hacer mas facil y obvia la decision de que estructura usaremos para resolver un problema. Similarmente, cuando estuvimos aprendiendo sobre algortimos en ordenamiento, nos enfocamos mucho en el tradeoff entre espacio y eficiencia de tiempo para ayudarnos a entender porque un algoritmo sería mejor que otro en determinado caso.

Cada vez vamos escalando en el tipo de estructuras, notando que hay algunas que sirven mucho en casos muy especificos. Para esta tarea hemos implementado un Trie. Un Trie es una estructura parecida a los arboles que hemos estudiados, en donde sus nodos guardan informacion de letras del abecedario. Tiene por objetivo optimizar los tiempos de busqueda en texto y facilitar la sugerencia mas probable de oraciones al recorrer una rama. Su nombre viene de 'information reTRIEval'. Al estructurar estos nodos de una forma particular, palabras e string pueden ser obtenidos ('reTRIEved') de la estructura al recorrerla por sus ramas. Esta estructura es relativamente nueva, siendo sugerida por primera vez en 1959 por el frances René de la Briandais.

En este informe explicaremos las decisiones de implementacion del Trie, su complejidad de ejecucion y construccion, tiempos reales versus teoricos y posibles alteraciones que podriamos realizar para jugar con el trade-off entre memoria/complejidad.



Hay casos en los que un Trie podria asemejarse a un arbol binario, pero la principal diferencia es que los nodos del Trie pueden tener mas de dos hijos.

1. Implementacion del Trie para problema propuesto

Como ya hablamos en la introduccion, para cumplir con el objetivo pedido, utilizamos una estructura llamada Trie.

```
// First, we define a Trie_Node structure with only the necessary information
// For further implementations, each node should have not only the biggest
// priority word, but also the word and priority of the word it was created
// with
// Trie node
struct Trie_Node
{
     struct Trie_Node *children[ALPHABET_SIZE];
     bool is_end;
     char longest_word[MAX_LENGTH];
     int longest_int;
};
```

Comenzamos definiendo un Trie Node, el cual reserva un espacio de memoria del porte del diccionario que queremos potencialmente guardar como hijo (el abecedario, mas la representacion del 'espacio'), un **booleano** que indica si el nodo corresponde al termino de una palabra o no, un **char** que almacena la palabra de mayor preferencia que ha pasado por ese lugar, junto con un **int** que nos indica la prioridad de dicha palabra.

De esta forma, nuestro Trie sera basicamente un arbol de Tries, en donde cada uno tiene espacios para alojar a sus potenciales hijos. Al armar el Trie, iremos creando nodo a nodo, utilizando el espacio del array del nodo principal indicado para almacenar un nuevo nodo, y creando la relacion padre/hijo entre ellos. Luego, se revisa si la prioridad actual es menor a la nueva que esta ingresando y, de ser el caso, se actualiza.
La aplicacion usual suele guardar cada prioridad/palabra en los nodos por los que pasa, pero abusamos de la nocion que podiamos entregar solo la mas prioritaria y guardamos solo esa informacion, borrando el resto. Podriamos cambiar esta cualidad al guardar, ademas, la palabra/prioridad con la que entro esa palabra, en dicha rama.

Pregunta 2

2.1 Cual deberia ser la complejidad en terminos de n de construir el Trie ?

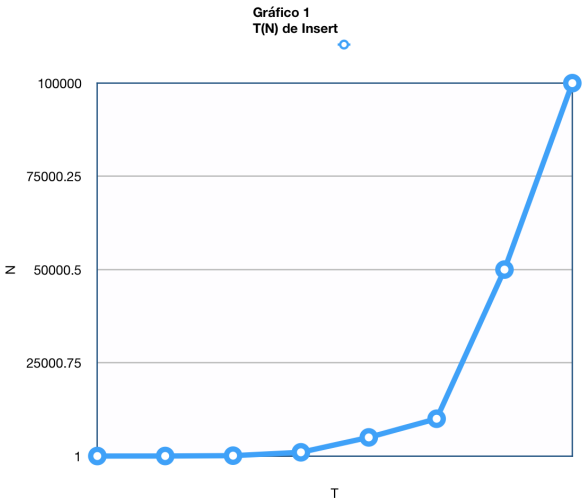
Al crear un Trie, debemos ir letra por letra, palabra por palabra, insertando y creando los nodos. Supongamos que crear un nodo es de O(1) en tiempo, que 'm' es el largo de la palabra mas larga y 'n' el numero de palabras. En dicho caso, el tiempo 'worst case' es de **O(mn)**, o O(n) si fijamos 'n' como el largo de todo lo que queremos insertar. Como mencionamos anteriormente, lo que ganamos en tiempo de ejecucion lo sacrificamos en memoria.

2.2 Tiempo de construccion del Trie

Para poder calcular los tiempos de ejecucion de la creacion de nuestro Trie segun cierto input, utilizamos la base de datos Large y fuimos editando el numero de lineas a leer. Utilizamos la libreria Time.C y colocamos un clock antes de la creacion del primer nodo y al final de la lectura. Luego, restamos los tiempos y calculamos la demora. Los tiempos obtenidos se detallan en el siguiente grafico. Los tiempos estan en segundos.

Grafico T(N)

Input	Tiempo
1	0.000166
10	0.000182
100	0.001147
1,000	0.007415
5,000	0.037036
10,000	0.056545
50,000	0.292419
100,000	0.515334



2.3 Hemos cumplido la complejidad teorica ?

Cuando empezamos el estudio de las posibles complejidades de cada algoritmo, llegamos a notar que la complejidad dependia, en muchos casos, del input que estuviesemos manejando. Este caso no es distinto, ya que el tiempo que demoraremos en armar nuestro Trie completo dependera netamente de cuanta informacion queramos que este aloje. Es por eso que nuestro calculo de complejidad de **O(mn)**, o simplemente O(n), que explicamos en el parrafo anterior, toma sentido al revisar cuanto se demora nuestro algoritmo en manejar distintas cantidades de datos. Notamos que, a medida que vamos aumentando el tamaño del input, el tiempo de ejecucion crece. **Es por esto que concluimos que nuestra Estructura de Datos cumple con el rendimiento esperado.**

Pregunta 3

Como hemos mencionado, un Trie es una estructura de datos eficiente cuando hablamos de obtener informacion de ella. Por ejemplo, si guardamos la informacion en un ABB, uno bien balanceado necesitara un tiempo proporcional a M * log N, en donde M es el string mas largo y N es el numero de keys en el arbol. Usando un Trie, podemos reducir dicha busqueda a O(M). Lo que ganamos en complejidad, nuevamente lo invertimos en memoria.

3.1 Tiempo teórico

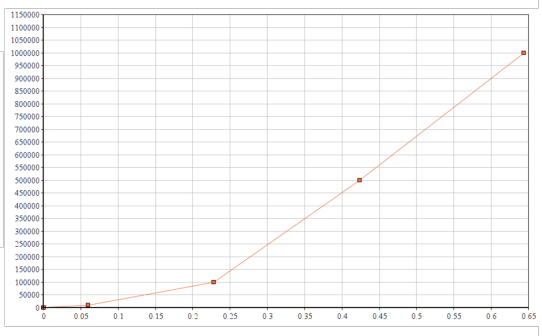
Como nuestra rama mas larga sera igual se larga que la palabra/frase mas larga, el worst case seria, para N palabras de largo M, **O(mn)**, ya que para cada frase tendremos que ir a su ultimo nodo para saber que hemos acertado y poder devolverla.

3.2 Tiempo de demora en consultas

Similarmente como hicimos con el tiempo de insercion, para poder calcular los tiempos de busqueda colocamos dos clocks, uno justo despues de cargar la base de datos y empezar a leer las consultas, y otro justo luego de haberlas terminado todas, sacando la impresion de archivos para no adulterar nuestros resultados con tiempos que no deberiamos contar.

Grafico T(M)

Search	Tiempo
10	0.000008
100	0.000078
1000	0.000680
1,000	0.005762
10000	0.059795
100,000	0.228078
500,000	0.42362
1,000,000	0.644066



3.3. Discusion

Nuevamente vemos como nuestra estructura responde directamente a un aumento en las consultas que le pedimos soportar. Para probar distintos inputs, fuimos cambiando los archivos de queries Large que nos entregaron. Una vez que se nos acabaron, editamos el mas largo y duplicamos su tamaño. Vemos como cada vez que aumentamos las consultas pedidas, notamos un mayor tiempo de ejecucion, sin obtener resultados demasiado mayores a medida que avanzamos. **Es por esto que concluimos que nuestra Estructura de Datos cumple con el rendimiento esperado.**

Pregunta 4: Bonus

Por ultimo, se nos pide proponer una nueva estructura que mantenga la complejidad de insercion/busqueda pero que abuse menos de la memoria propuesta. Existen varias respuestas para esta pregunta.

Primero, podriamos optimizar la estructura que ya hicimos ? Creo que si. Por ejemplo, actualmente estamos reservando todo un espacio de memoria del porte del abecedario a representar para poder guardar los potenciales hijos que vayamos a insertar. Podriamos realizar esto de forma dinamica y solo pedir espacios de memoria que necesitemos, asignandolos en el momento en vez de dejarle todo el espacio desde el principio. Ademas, podriamos guardar la informacion de las letras y palabras en forma binaria, con el objetivo de utilizar menos espacio. Otro punto es el que estamos guardando la informacion de la mejor sugerencia en cada nodo. Esto lo podriamos cambiar al solo guardar esa informacion en las 'stopwords' o finales de palabras/oraciones de sufijos, cosa de no abusar de la memoria que tenemos.//

Luego cabe pensar, existe otra estructura que permita mantener esos tiempos de busqueda / insercion peron abusando menos de memoria ? Por supuesto.

Por ejemplo, existen las Hash Tables. La implementacion HashTable es muy eficiente en espacio comparado con la implementacion basica del Trie. Con strings, un ordenamiento es necesario en la mayoria de las aplicaciones practicas. Lamentablemente el Hashtable complica el orden lexicografico. Ahora, si nuestra aplicacion necesita mucho el orden lexicografico, conviene un Trie, pero si solo lo necesitamos para busqueda y para optimizar el espacio de memoria utilizado, entonces un HashTable es lo indicado. La gracia de estas es, por ejemplo, que pueden compartir espacios para letras iguales, mientras que el Trie tendra el orden de las letras segun posicion y podria repetir muchas veces la misma letra, ocupando espacio innecesario. Estas cumplen ademas la propiedad de ser muy eficientes en busqueda.

Tambien existen los Ternary Search Trees (TSTs), quienes tambien mejoran los tiempos de memoria y mantienen busqueda.

Conclusiones

Como conclusion de esta tarea, pudimos observar como el tener una buena base teorica puede ayudarnos mucho a la hora de tomar decisiones sobre que estructuras deberiamos usar. Al conocer estos usos especificos podemos optimizar de gran manera los procesos que queramos ejecutar. Hemos aprendido y analizado el uso del Trie y ya podemos contarlo como una herramienta en futuos desafios.

Es importante mencionar que hice un uso extensivo de informacion que encontre en internet, siendo mis principales fuentes las que mencionare a continuacion.

Bibliografia:
1. https://medium.com/basecs/trying-to-understand-tries-3ec6bede0014
2. https://www.toptal.com/java/the-trie-a-neglected-data-structure
3. https://www.geeksforgeeks.org/trie-insert-and-search/
4. https://www.topcoder.com/community/data-science/data-science-tutorials/using-tries/
5. http://loup-vaillant.fr/projects/string-interning/benchmark
6. stackoverflow.com/questions/327223/memory-efficient-alternatives-to-python-dictionaries