Análisis y diseño de algoritmos I

Proyecto final

Árboles binarios de búsqueda óptimos.

Integrantes: Chaju Yamil, Hoses Pedro.

E-mails: yamilchaju@gmail.com, pedrohoses@gmail.com

Índice

Introducción

En el siguiente informe se describe como se desarrolló el problema propuesto por la catedra, el cual consiste en que dada una lista de palabras con su respectiva probabilidad de que se usen, hacer un árbol binario de búsqueda usando programación dinámica para guardar las claves y que minimice el número medio de comparaciones para encontrar una clave o para garantizar que no está.

Nuestra tarea es implementar el problema usando C++ para crear el árbol y así poder recorrerlo en un tiempo medio de búsqueda mínimo, dependiendo de la probabilidad de uso que tengan las palabras.

Solución

Lo primero que se realizó para armar la matriz "C" con la cual armaremos el árbol es:

- C_{ii} = 0 con 0 ≤ i ≤ N (siendo N la cantidad de palabras).
- En la siguiente diagonal a los 0's se colocan las probabilidades de las palabras ordenadas alfabéticamente.

Luego de haber llenado las dos diagonales se procede a tomar las claves de a grupos para llenar las diagonales faltantes, de este modo para la tercera diagonal se toman grupos de a dos claves, para la cuarta de a tres, y así sucesivamente. El grupo de claves que se toman para calcular un casillero son desde la clave que está en la fila del mismo hasta la clave que está en la columna (esto genera una submatriz de la matriz original).
Luego, con el grupo de claves, se hace la siguiente ecuación:

$$C_{ij} = m_{ij} + \min_{i<k<j}\{C_{i,k-1} + C_{kj}\}, si\ 0 \leq i < j \leq n$$

- m_{ij} es el número medio de comparaciones con la raíz.
- $C_{i,k-1}$ es el numero medio de comparaciones con el subárbol izquierdo.
- C_{kj} es el número medio de comparaciones con el subárbol derecho.
- C_{ij} es el numero medio de comparaciones efectuadas en un subárbol óptimo.

Una vez terminada de llenarse la matriz se procede a armar el árbol usando el camino inverso, para esto se toma el casillero C_{1n} (extremo superior derecho) donde estaría guardado el numero medio de comparaciones optimo, se busca en la diagonal de

probabilidades la clave “raíz” con la cual se genero ese casillero, esa clave se inserta en el árbol y se procede a hacer lo mismo con el subárbol derecho e izquierdo a esa clave.
En el siguiente ejemplo mostramos las claves con sus probabilidades, con rojo la probabilidad de la raíz del subárbol, con azul el subárbol izquierdo, con amarillo el derecho, sus divisiones y como quedaría el árbol:

Palabra	Probabilidad
a	0,22
al	0,18
ama	0,20
eso	0,05
si	0,25
sin	0,02
su	0,08

Matriz original:

0	0,22	0,58	1,02	1,17	1,83	1,89	2,15
	0	0,18	0,56	0,66	1,21	1,27	1,53
		0	0,20	0,30	0,80	0,84	1,02
			0	0,05	0,35	0,39	0,57
				0	0,25	0,29	0,47
					0	0,02	0,12
						0	0,08
							0

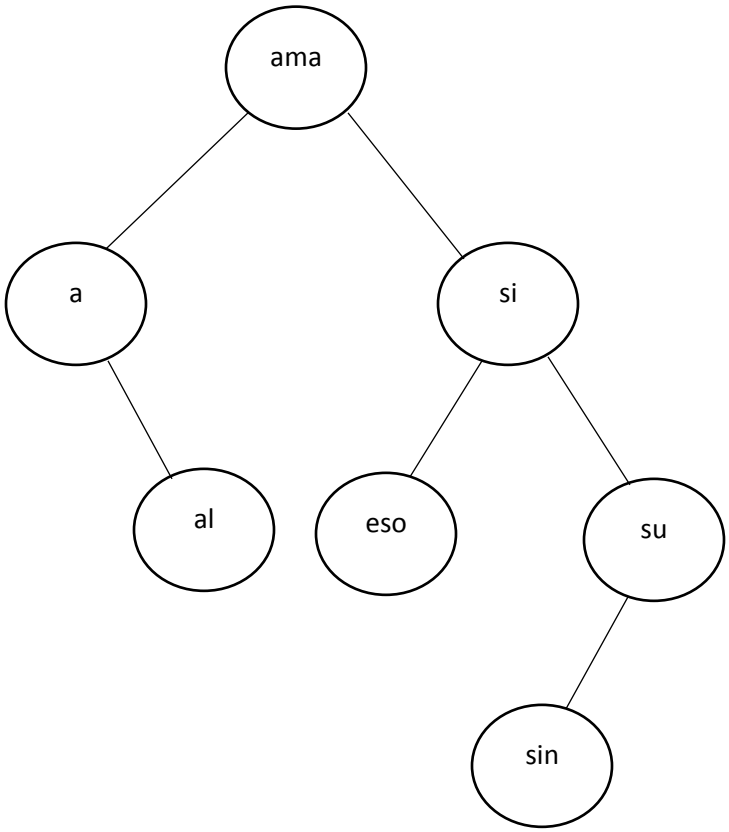
Subárbol izquierdo:

0	0,22	0,58
	0	0,18
		0

Subárbol derecho:

0	0,05	0,35	0,39	0,57
	0	0,25	0,29	0,47
		0	0,02	0,12
			0	0,08
				0

Árbol:



Implementación:

Hicimos una clase árbol con 9 métodos:

- **Árbol():** Inicializamos todas las variables del privado en NULL. Complejidad O(1).
- **Árbol(const map < string , int > & claves):** Llama a los procedimientos para crear la matriz y luego, en base a eso, el árbol. Complejidad O(N^2).
- **String getClave():** Retorna la raíz del árbol. Complejidad O(1).
- **Void getMenores(arbol &actual):** Devuelve un puntero al subárbol izquierdo. Complejidad O(1).
- **Void getMayores(arbol &actual):** Devuelve un puntero al subárbol derecho. Complejidad O(1).
- **Int sumamenores(int fila , int columna):** Devuelve la menor suma entre el numero medio de comparaciones del subárbol derecho con el izquierdo para cada clave del conjunto. Complejidad O(N) siendo N la cantidad de claves del conjunto tomado.

- **Int sumaprobabilidades(int fila , int columna):** Devuelve la suma de todas las probabilidades del conjunto. Complejidad O(N) siendo N la cantidad de claves del conjunto tomado.
- **Void crearArbol(nodoarbol*&arbol, int fila , int columna):** Crea el árbol en base a la matriz creada en "crearmatriz". Complejidad O(N)
- **Void crearmatriz(const map<string,int> & claves):** Carga la matriz de probabilidades ejecutando la función suma menores y suma probabilidades. Del resultado de estas dos funciones, se realiza la suma y se le asigna al casillero correspondiente. Complejidad O(N^2)

Conclusión

A la hora de realizar el trabajo se presentaron dificultades tales como la carga y descarga de la matriz. Esto se pudo solucionar fácilmente una vez que entendimos que el problema respondía al principio de la optimalidad, es decir que si tenemos un arbol optimo , los sub arboles de este también serán óptimos. A partir de ese momento nos dimos cuenta de que a medida que cargábamos las diagonales de la matriz obteníamos los arboles óptimos para esa configuración de claves. Solo bastaba con repetir el proceso abarcando cada vez más cantidad de claves.
Ya cargada la matriz, realizar la descarga fue una tarea mucho mas sencilla porque solo necesitábamos identificar la raíz del arbol y tomar 2 submatrices (menores y mayores) para repetir el proceso.
En lo general el trabajo fue muy interesante y los métodos y las estructuras que utilizamos para resolverlo son las adecuadas: no malgastamos recursos y optimizamos tiempo.