**Trabajo practico especial**

**Analisis y Diseño de Algoritmos I**

*Parte 2*

*Grupo 31*

*Integrantes:*

* Talú, Bernabé.

[*Berni.talu@hotmail.com*](mailto:Berni.talu@hotmail.com)

* Alvarez, Maximiliano.

[*Maxi25294@gmail.com*](mailto:Maxi25294@gmail.com)

**Introduccion**

Este informe analiza y demuestra teóricamente la implementación de un árbol binario de búsqueda óptimo que, dado un archivo de conjunto de claves(palabras) ordenadas donde, para cada clave se indica su probabilidad, y a partir de allí se procede a construir el árbol binario óptimo.

Para esto, utilizaremos un algoritmo de programación dinámica que, partiendo de un conjunto de claves y su de búsqueda, obtiene la configuración de un árbol binario de búsqueda óptimo.

A grandes rasgos, el diseño de un algoritmo de Programación Dinámica consta de los siguientes pasos:

1. Planteamiento de la solución como una sucesión de decisiones y verificación de que ésta cumple el principio de óptimo.

2. Definición recursiva de la solución.

3. Cálculo del valor de la solución óptima mediante una tabla en donde se almacenan soluciones a problemas parciales para reutilizar los cálculos.

4. Construcción de la solución óptima haciendo uso de la información contenida en la tabla anterior.

En este caso, el algoritmo por programación dinámica utiliza una matriz para ir almacenando los datos de los distintos árboles y la siguiente fórmula para calcular sus costos:

1) Cij = 0 si i>j.

2) Cij = min(Ci,k-1+Ck+1,j) + ∑\_(k=i)^j▒pk si i<=j.

Siendo C la matriz, (i,j) la posición y pk el conjunto de probabilidades obtenidas del archivo claves.

El algoritmo parte de tomar las claves en sub-grupos de distinto tamaño para luego elegir la configuración que minimiza ese sub-árbol. Así va expandiendo el tamaño del conjunto de claves y probando las distintas configuraciones de árboles posibles. Cada resultado intermedio lo almacena en una celda de la matriz, hasta completar la parte derecha superior.

Al finalizar, en la celda [0, n - 1] se encuentra el costo del árbol binario de búsqueda óptimo y, siguiendo la información de las raíces, se puede obtener su configuración.

El tamaño de sub-grupos de claves parte desde 0 hasta la última clave. Por ejemplo, en el archivo” Claves\_1”, las claves van desde 0 hasta 9.

**Arbol binario de búsqueda optimo**

Para la construcción del Arbol binario de búsqueda optimo realizamos los siguientes pasos:

* Utilizamos la función **cargarColeccion** (O(n)) para, abrir los archivos de palabras (Claves\_1 y Claves\_2 según corresponda).
* A medida que se extrae cada palabra, es decir, el dato de tipo string almacenado en cada posición del archivo de palabras, se separa dicho string, mediante la función **separarString** (O(1)).
* La función separarString realiza la separación o división de ese string obteniendo como resultado, por un lado la palabra(clave) y por otro el costo de dicha clave almacenando ambos en un arreglo de dimensión 2(p[0]=calve,p[1]=costo). Luego se utiliza la función **stringToNumber**(O(1)), que lo que hace esta, es convertir el costo, obtenido en la separación, de una variable de tipo string a tipo float.
* Despues utilizando una lista simple, de estructura NodoLista, en la cual definimos una variable de tipo T(clave a guardar), una variable de tipo float(costo de clave) y un puntero al siguiente nodo, se almacena cada clave con su costo ,ordenandolos de menor a mayor por clave, hasta llegar al final de los archivos de Palabras.
* Ya teniendo dicha Lista cargada, procedemos a realizar el rellenado de una matriz de tipo Registro, el cual esta definido como un registro de dos campos, el primero es de tipo float para costo y el segundo es de tipo int para numero de clave.
* Lo que primero realizamos para rellenar la matriz, es utilizar una función llamada **CargarDiagonal**(O(n)), que lo que hace es:
* Ingresa a la lista.
* Extrae el costo de la clave almacenada en la primera posición de la lista, sin modificar la lista, y almacena en la posición[0][0] de la matriz este costo(matriz[0][0].costo=costo).
* Hace lo mismo para almacenar el numero de raíz, ubicado en esa posición de la lista, en la misma posición de la matriz.
* Luego avanza un valor de fila y un valor de columna en la matriz y también accede a la siguiente posición de la lista.
* Asi hasta llegar al final de la lista.
* La diagonal de la matriz queda cargada.
* Luego, creamos una función void llamada Rellenar que esta a su vez invoca a una de tipo float llamada **“CostoTotal”(**O(n)). En esta última función, lo que realizamos es calcular el costo mediante las formulas:

1. [Cij=0 si i>j]
2. [ Cij = min(C[i,k-1]+C[k+1,j]) + k si i<=j]

Para calcular el minimo, creamos una función llamada **“CostoMinimo”**(O(1)), donde aquí comparamos celda a celda según función (1) y (2) , y nos quedamos siempre con el menor costo. De esta manera calculamos el costo total (dependiendo de G, donde este está asociado con el sub-grupo de claves) de respectivas claves permitiendo asi poder llenar las celdas restantes.

* Luego de rellenar todas las celdas de la matriz, procedemos crear el Árbol binario de búsqueda óptimo. El costo de este árbol es el costo ubicado en la matriz en la posición [0,Col-1] , siendo “Col” el numero de columnas de la matriz. La raíz del árbol, coincidirá con el numero de raíz alamcenada en esa posición de la matriz. Luego aplicamos recursivamente C [1, k-1] para el sub-árbol izquierdo y C [k+1,n] para el sub-árbol derecho.
* Finalmente, tomamos un archivo de texto con las palabras a buscar dentro del árbol y así poder generar, como salida, un archivo que visualice el resultado de cada búsqueda junto al costo de búsqueda según el árbol.

**Complejidad temporal**

**Métodos Restantes:**

**InicMatriz(Matriz,Fila,Col) //O(n2)**

Inicializa la matriz con sus respectivos números de filas y columnas.

**arbol.inOrden() //O(n)**

Muesrtra por pantalla el árbol binario optimo, de manera ordenada.

**buscarPalabraArbol(Busqueda1,Resultado1)//O(n)**

Accede a un archivo de palabras(Busqueda\_1) y se fija si cada una de ellas existe o no en el árbol binario óptimo. A medida que realiza la búsqueda, almacena en un archivo (Resultado\_1) los siguientes resultados:

* **Pertenencia al árbol**: Existe: 1/No esxite: 0
* **Costo:** es el número de iteraciones que realiza el método hasta encontrar la palabra a buscar.
* **Costo métrica:** es el mismo resultado almacenado en Costo, pero con la particularidad de que a dicho resultado lo divide por el numero de elementos del árbol.
* **llenarArbol(árbol,matriz,listaelementos,aux,col,posición) //O(N2)**

**Complejidad temporal del algoritmo:**

Sumando las complejidades temporales de todos los llamados en el main, llegamos a la conclusión de que la complejidad total del algoritmo es O(N2).

**Conclusion**

Llegamos a la conclusión, comparándolo con el trabajo anterior, que utilizando la programación dinámica se disminuye notablemente el costo de la búsqueda de palabras en el árbol, gracias a la matriz que almacenó los costos y las raíces, pudimos armar una “ruta” de búsqueda óptima para que tanto el costo métrica, como el costo árbol baje, haciendo que también el rendimiento sea mucho mejor.