Informe

Proyecto de Programación Dinámica

Árbol Binario de Búsqueda Optimal

Algoritmos y Complejidad - 2016

Ricardo Ferro Moreno (85611)

Ejercicio 1

Se posee un conjunto ordenado de n claves $c_1 < c_2 < \ldots < c_n$ donde cada clave c_i está asociada a una cantidad de accesos k_i de veces. Se desea encontrar un árbol binario de búsqueda optimal, tal que su costo definido como $\sum_{i=1}^{n} k_i * (d_i + 1)$ donde d_i es la profundidad del nodo c_i, siendo 0 la profundidad para el nodo raíz y 1 más la profundidad de padre para el resto.

Para este problema el enunciado sugiere que para cada subárbol que contiene las claves *ci* a *cj* calcular *C(i,j)* el número mínimo de comparaciones para acceder a estos nodos. Podemos pensar en el caso más simple: *C(i,i)* (ó *C(i,j)* cuando *i=j*), lo que significa que nuestro subconjunto tiene un único nodo, y el costo para acceder al árbol entonces es k_i (su peso). Luego nos queda ver qué pasa cuando $i \neq j$. Si *j<i* podríamos suponer que el subárbol no existe (es nulo), y luego su peso es 0. Si *i<j* entonces significa que tenemos un árbol de al menos dos nodos, y luego el peso optimal será el menor de los pesos de los posibles árboles que se pueden formar. Para generalizar, supongamos que para cada posible nodo raíz *r* que está entre *i* y *j*, el costo de C(i,j) es el menor de los pesos. Esto es, el costo optimal del subárbol izquierdo de *r* (lo que sería su hijo izquierdo, o los nodos que van desde *i* hasta *r-1*), más el costo optimal del subárbol derecho de *r* (análogo al hijo izquierdo), más el peso que se obtendría si se usara *r* como el nodo raíz del árbol optimal.

La recurrencia quedaría de la siguiente forma:

$$C(i,j) = \begin{cases} 0 & j < i \\ k_i & i = j \\ min_{i \leq r \leq j}(peso(i,r,j) + C(i,r-1) + C(r+1,j)) & i < j \end{cases}$$

Donde *peso(i,r,j)* sería el peso mencionado recién, si se usara *r* como nodo raíz. Notar que para cualquier nodo *r* entre el rango de i a j, este peso será siempre el mismo, ya que consistiría en sumarle 1 de profundidad a cada elemento de los subárboles izquierdo y derecho $(\sum_{i=1}^{r-1} k_i)+(\sum_{h=r+1}^{j} k_h)+k_r$.

Es decir, que peso *peso(i,r,j)* está definido en realidad como la sumatoria

$$\sum_{i=1}^{j} k_i$$

Notar que *peso(i,r,j)* no depende de quién sea la raíz, sino del peso de los elementos que forman parte de la posible solución optimal, esto es los nodos entre *i* y *j*.
Un posible algoritmo en pseudocódigo para este problema sería:
Sea *costo* una matriz donde se guardarán los costos optimales de cada subconjunto de árboles, por ejemplo para *costo[a][b]* tendrá el valor optimal para un árbol que contenga los nodos $c_a < \ldots < c_b$. La solución parte de los subárboles más pequeños (de un único nodo) y luego se van obteniendo los costos optimales para conjuntos de claves de tamaño 2 hasta N.

```
Inicio
      Inicializar la diagonal de la matriz costo con los valores de k_i para cada c_i
      (Eso abarca los casos bases donde los subárboles tienen un único nodo)

      Desde cantidad = 2 hasta N (voy agarrando conjuntos de "cantidad" nodos)
            Para cada uno de los subconjuntos (i,j) que tienen cantidad de nodos
                  Costo[i,j] = Infinito
                  Obtener peso(i,j) (i.e, la sumatoria de los k desde i hasta j)
                  Para cada posible raiz r (que está entre i y j)
                        Obtener el costo del árbol si la raiz fuera r
                        Si el costo es el mínimo, actualizar Costo[i,j]
      Resultado = Costo[1][N]
Fin
```

Ejercicio 2

Principio de optimalidad:

Sea $C_T(i,j) = \sum_{i=1}^{n} k_i * (d_i + 1)$ el costo de un árbol binario de búsqueda T para las claves desde i hasta j. Supongamos que tenemos un árbol optimal T_{OPT} para las claves de i hasta j, con una raiz k, un subárbol hijo izquierdo T_L formado por las claves desde i hasta k-1,y un subárbol hijo derecho T_R formado por las claves desde k+1 hasta j.
Por la definición de la solución, podemos expresar C(i,j) de T_{OPT} como:

$$C_{TOPT}(i,j) = peso(i,j) + C_{TL}(i, k-1) + C_{TR}(k+1, j)$$

Para demostrar la optimalidad, necesitamos demostrar que los subárboles T_L y T_R también son optimales (con mostrar uno de los dos es suficiente, el otro es análogo).
Supongamos que existe un T_P que tenga menor costo que T_L, esto es:

$$C_{TP}(i, k-1) < C_{TL}(i, k-1)$$

Luego, podemos armar un nuevo árbol F reemplazando a T_L por T_P en T_{OPT}.
El costo de P sería similar al de C_{TOPT} reemplazando $C_{TL}(i, k-1)$ por $C_{TP}(i, k-1)$.

Como C_{TP}(i, k-1) < C_{TL}(i, k-1), entonces C_F(i,j) < C_{TOPT}(i,j). Sin embargo esto contradice la suposición inicial, de que T_{OPT} es un árbol optimal. Por lo tanto, los subárboles contenidos también son optimales. Por lo tanto el problema presenta el principio de optimalidad.

Ejercicio 3

Ejemplo de solapamiento de subinstancias:

Supongamos que se desea calcular el costo optimal de un árbol de 4 nodos, esto es C(1,4). Luego de las posibles soluciones que existirán, podrían ser:

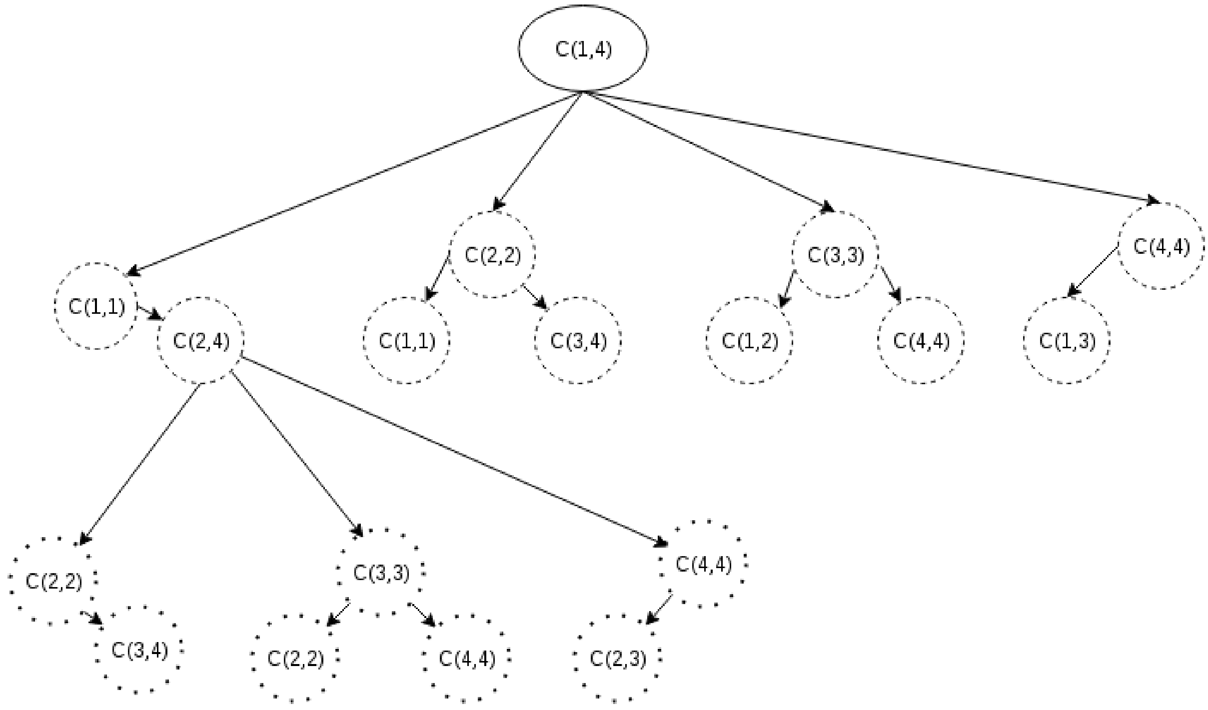
- C(1,1) con hijo derecho C(2,4).
- C(2,2) con hijo izquierdo C(1,1) y derecho C(3,4).
- C(3,3) con hijo izquierdo C(1,2) y derecho C(4,4).
- C(4,4) con hijo izquierdo C(1,3).

Si la opción fuera la primera, por ejemplo, implicaría calcular C(2,4), lo que traería otro conjunto de posibles opciones. Para el subárbol C(2,4), podría ser:

- C(2,2) con hijo derecho C(3,4).
- C(3,3) con hijo izquierdo C(2,2) y derecho C(4,4)
- C(4,4) con hijo izquierdo C(2,3).

Notar que se repite por ejemplo el cálculo de C(2,2), C(3,4) y C(4,4) del paso anterior a éste.

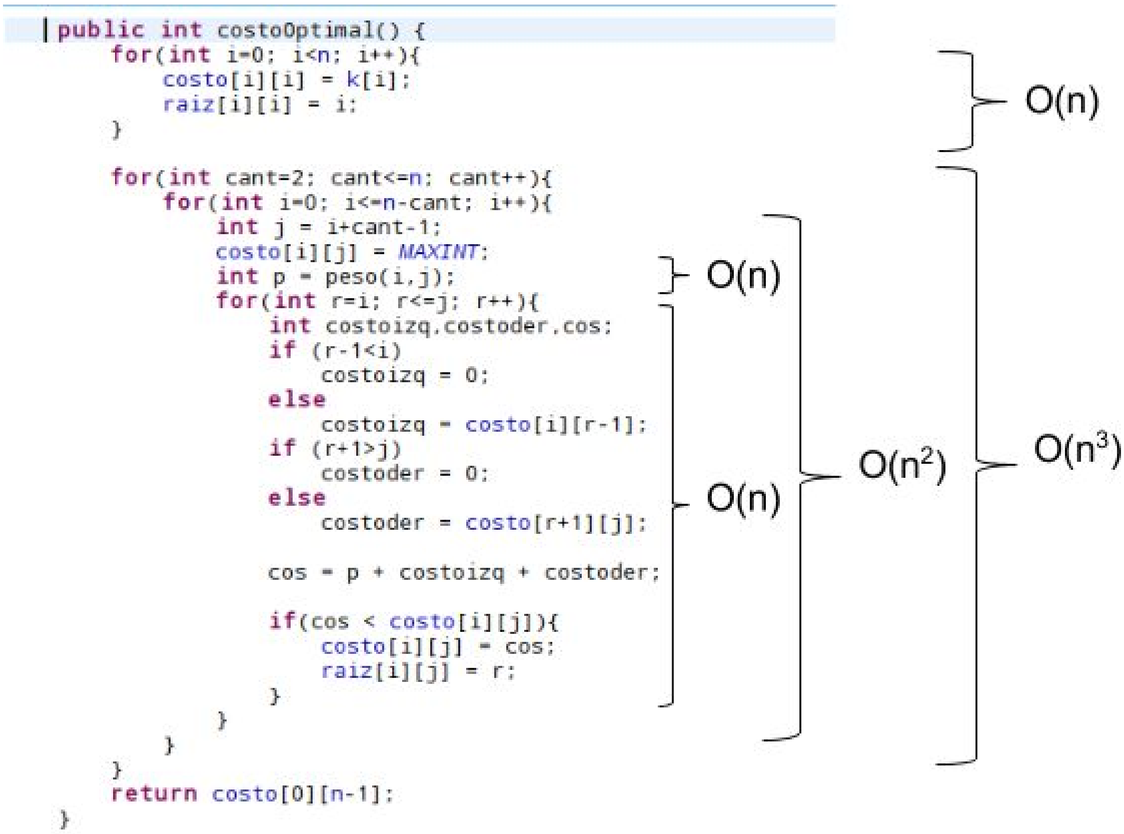
Visto gráficamente:



Notar cómo las sub instancias *C(2,2)*, *C(2,3)*, *C(3,3)* y *C(4,4)* se repiten. El árbol lógicamente no se muestra completamente en la imagen por cuestiones de espacio.

Ejercicio 4

Para la implementación se realizó una pequeña modificación agregando una matriz de raíces para la reconstrucción de la solución. Se puede ver más detalles sobre esta decisión en el Ejercicio 5 donde se muestra la implementación en Java. Como se usaron dos matrices de tamaño NxN, el espacio utilizado es de $O(2n^2)$, esto equivale a que en espacio es de $O(n^2)$. En cuanto al tiempo de ejecución, en el peor de los casos cada uno de los *for* anidados son de O(*n*), por lo tanto el algoritmo para obtener el costo optimal del conjunto de n claves es de $O(n^3)$. La inicialización de la diagonal de la matriz es de O(*n*) por lo que asintóticamente no cambia el orden.
A continuación se muestra un análisis del código:



```
public int costoOptimal() {
    for(int i=0; i<n; i++){
        costo[i][i] = k[i];
        raiz[i][i] = i;
    }

    for(int cant=2; cant<=n; cant++){
        for(int i=0; i<=n-cant; i++){
            int j = i+cant-1;
            costo[i][j] = MAXINT;
            int p = peso(i,j);
            for(int r=i; r<=j; r++){
                int costoizq,costoder,cos;
                if (r-1<i)
                    costoizq = 0;
                else
                    costoizq = costo[i][r-1];
                if (r+1>j)
                    costoder = 0;
                else
                    costoder = costo[r+1][j];

                cos = p + costoizq + costoder;

                if(cos < costo[i][j]){
                    costo[i][j] = cos;
                    raiz[i][j] = r;
                }
            }
        }
    }
    return costo[0][n-1];
}
```

Ejercicio 5

Para la implementación en Java, además de la matriz de *costo* (de tamaño NxN), se hizo uso de una matriz auxiliar *raiz* (también de tamaño NxN) donde se guardan las raíces de los subárboles optimales. La misma es útil por simplicidad a la hora de reconstruir la solución.

Una aclaración importante es el tema de los índices: En el enunciado habla de claves con índices desde 1 hasta N, pero por simplicidad, internamente el manejo de matrices es desde 0 hasta N-1. Lo mismo sucede con la matriz que guarda las raíces de los subárboles optimales, donde las raíces son de la forma índice - 1.

Al momento de reconstruir el árbol binario de búsqueda, notar que el elemento que se guarda es r + 1 para restablecer la notación desde 1 hasta N.

A continuación se muestra el código fuente de la clase Implementación:

```
public class Implementacion {

    private int n;
    private int[] k;

    // Matriz para ir guardando el costo optimal de cada subarbol
    private int[][] costo;
    // Matriz para ir guardando las raices de cada subarbol optimal
    private int[][] raiz;

    private static final int MAXINT = Integer.MAX_VALUE;

    public Implementacion(int[] k) {
        this.n = k.length;
        this.k = k;
       costo = new int[n][n];
       raiz = new int[n][n];
    }

    // Debe devolver un int con el costo para un árbol binario de búsqueda optimal
    public int costoOptimal() {

       // Completo la matriz diagonal (casos bases).
       // Cuando el arbol tiene un solo nodo C(i,j) cuando i=j
       // entonces el costo es Ki*1, es decir el Ki del nodo. C(i,j) = Ki
       // Además la raiz es el nodo mismo.
        for(int i=0; i<n; i++){
            costo[i][i] = k[i];
            raiz[i][i] = i;
        }

        for(int cant=2; cant<=n; cant++){
            for(int i=0; i<=n-cant; i++){
                int j = i+cant-1;
```

```
                costo[i][j] = MAXINT;
                int p = peso(i,j);

                for(int r=i; r<=j; r++){
                    // Calculo, cual seria el costo para cada raiz r posible
                    // Para ello obtengo el costo de los dos subarboles hijos
                    int costoizq,costoder,cos;
                    if (r-1<i)
                      costoizq = 0;
                    else
                      costoizq = costo[i][r-1];
                    if (r+1>j)
                      costoder = 0;
                    else
                      costoder = costo[r+1][j];
                    // peso(i,j) + C[i,r-1] + C[r+1][j]
                    cos = p + costoizq + costoder;

                    // Si el costo es el minimo, lo guardo (y su raiz)
                    if(cos < costo[i][j]){
                          costo[i][j] = cos;
                          raiz[i][j] = r;
                    }
                }
            }
        }
        return costo[0][n-1];
    }

    /*
     * Metodo para obtener la sumatoria de los accesos para
     * un subconjunto de nodos C(i,j)
     */
    private int peso(int i, int j){
        int suma = 0;
        for(int ind=i; ind<=j; ind++)
            suma += k[ind];
        return suma;
    }

    /*
     * Metodo para reconstriur el Arbol Optimal.
     * Utiliza la matriz de raices (raiz)
     */
    private ArbolBinario reconstruir(int i, int j){
       if (j<i || j>n || i<0 || i>=n){
              return null;
       } else{
              int r = raiz[i][j];
              // Si estoy en raiz[i][i], devuelvo un nodo hoja (con hijos null)
              // Sino, significa que estoy en un nodo interno y necesito seguir
              // reconstruyendo el subarbol izquierdo y derecho
              if (i==j)
                return new ArbolBinario(r+1,null,null);
              else
                return new ArbolBinario(r+1,reconstruir(i,r-1),reconstruir(r+1,j));
       }
    }
```

```
    // Se llama luego de haber invocado a costoOptimal
    // Debe devolver un árbol binario optimal representado con la clase ArbolBinario
    public ArbolBinario reconstruirSolucion() {
        return reconstruir(0,n-1);
    }
}
```