Laboratorio Nro. 3: Backtracking

Isabela Muriel Roldán
Universidad Eafit
Medellín, Colombia
imurielr@eafit.edu.co

Mateo Flórez Restrepo
Universidad Eafit
Medellín, Colombia
mflorezr@eafit.edu.co

1) Explicación ejercicio 1.6

El algoritmo del ejercicio 1.6 recibe un grafo, un nodo inicial, y otro al que se quiere llegar. Lo que este hace es revisar cada nodo sucesor que no haya sido visitado hasta encontrar un camino hacía el nodo objetivo, esto lo hace con ayuda de una pila auxiliar que permite ir almacenando los nodos por los que se pasa, sin embargo este método no asegura que el camino encontrado sea el más corto, simplemente revisa si es posible llegar de un punto a otro.

3) Simulacro de preguntas de sustentación de Proyectos

1. Para resolver el problema del camino más corto existen varios métodos además de DFS y BFS, entre ellos está el algoritmo de Dijkstra y otros algoritmos voraces.
2.

Valor de N	Tiempo de Ejecución
4	0
5	0
6	0
7	0
8	0
9	0
10	0
11	0
12	0
13	0
14	3
15	2
16	5
17	3
18	14
19	1
20	80
21	3
22	603
23	11
24	152
25	21

26	**175**
27	**207**
28	**1424**
29	**777**
30	**30374**
31	**7243**
32	**51151**
N	**O( )**

3. Existen casos en los que es preferible utilizar DFS y otros BFS, por ejemplos, cuando se busca el camino más corto, teniendo en cuenta las distancias que hay de un nodo a otro es recomendable utilizar DFS, por otro lado, si simplemente se busca llegar de un nodo a otro recorriendo la menor cantidad de nodos posibles, es mejor utilizar BFS pues este analiza los nodos que están al mismo nivel por lo que no es necesario recorrer una rama completa innecesariamente.
4. Existen muchas formas de búsqueda en un grafo, entre ellas DFS, BFS, Dijkstra, A*, IDA*, Fringe Search o D*.
5. Para implementar la solución al ejercicio en línea planteado usamos recorrido por DFS y algoritmos voraces. Lo primero que hace el programa es solicitar al usuario mediante entrada de teclado (Scanner) los vértices del grafo y la cantidad de arcos entre vértices a agregar, luego solicitara los datos de los arcos (inicio, fin y el peso) por cada uno de ellos. Teniendo en cuenta todos esos datos el programa creara un grafo y por medio del algoritmo DFS verificara que haya un camino entre el primer vértice y el ultimo. En caso de que sea verdadero el grafo será enviado al algoritmo voraz para encontrar el camino más corto que hay entre el vértice inicial 1 y el final n, teniendo en cuenta el peso de la arista, en cuanto lo encuentre añadirá cada vértice por el que pasa a una lista en ese mismo orden y luego esta será retornada como el camino más corto. Si, por el contrario, no hay camino el programa inmediatamente retornara -1.
6. **Cálculo de complejidad del algoritmo 2.1:**

```
public static boolean hayCaminoDFS(Digraph g, int inicio, int fin){
      boolean[] visited = new boolean[g.size()];
      return hayCaminoDFS(g,inicio,fin,visited);
   }
     private static boolean hayCaminoDFS(Digraph g, int nodo, int objetivo, boolean[] visited){
      visited[nodo] = true;                                          //C1
      ArrayList<Integer> sucesores = g.getSuccessors(nodo);          //C2
      if(sucesores != null){                                         //C3
         for(Integer sucesor : sucesores){                           //C4*n
            if(!visited[sucesor] && (sucesor == objetivo ||          //C5*n*T(n-1)
                        hayCaminoDFS(g, sucesor, objetivo, visited))){
               return true;                                          //C6
            }
         }
      }
      return false;                                                  //C7
   }
```

T(n) = C*n*T(n-1) → en el peor de los casos
Resultado de ecuación de recurrencia:

DOCENTE MAURICIO TORO BERMÚDEZ
Teléfono: (+57) (4) 261 95 00 Ext. 9473. Oficina: 19 - 627
Correo: mtorobe@eafit.edu.co

T(n) = C*C^n* n!
T(n) = O(C^n)

```
public static ArrayList<Integer> recorrido(Digraph g, int inicio)
    {
       boolean[] recorrido = new boolean[g.size()];                    //C1
       ArrayList <Integer> res = new ArrayList<Integer>();             //C2
       int index = 0;                                                  //C3
       int actual = inicio;                                            //C4
       do {                                                            //C5*n
          int cerca = 0;                                               //C6*n
          int pesoMin = 1000000;                                       //C7*n
          recorrido[actual] = true;                                    //C8*n
          ArrayList<Integer> sucesores = g.getSuccessors(actual);      //C9*n
             for (int sucesor: sucesores){                             //C10*n^2
              if (g.getWeight(actual,sucesor) < pesoMin                //C11*n^2
                && sucesor != actual
                && !recorrido[sucesor] )
              {   cerca = sucesor;                                     //C12*n^2
                 pesoMin = g.getWeight(actual,sucesor);                //C13*n^2

              }
           }
          res.add(actual);                                             //C14*n
          if(actual==g.size()-1){break;}                               //C15*n
          actual = cerca;                                              //C16*n
       } while (index < g.size());                                     //C17*n

       return res;                                                     //C18
    }
```

T(n)= C(n)+C(n^2)+C
T(n)= O(C(n+n^2+1))
T(n) =O(C*n^2)
T(n)= O(n^2)

7. **n y m en la complejidad:** En los cálculos de la complejidad anterior, estas variables se refieren a la cantidad de veces que el algoritmo tiene que ejecutar algún paso o acción al momento de compilarse para llegar al resultado esperado. En los ejercicios anteriores usamos en la gran mayoría solo la variable n ya que en los códigos solo se trabajaba con un solo tipo de entrada igual para cada caso. La variable m la usaríamos si hubiera más de una entrada para poder diferenciar el alcance de la complejidad de los códigos.

4) Simulacro de Parcial

1. n-a, a, b, c
   res, solucionar(n, b, c, a)
   res, solucionar(n, c, a, b)
2. graph.length
   v, graph, path, pos

graph, path, pos+1

3.

0 -> [0,3,7,4,2,1,5,6]
1 -> [1,03,7,4,2,6,5]
2 -> [2,1,0,3,7,4,5,6]
3 -> [3,7]
4 -> [4,2,1,0,3,7,5,6]
5 -> [5]
6 -> [6,2,1,0,3,7,4,5]
7 -> [7]

0 -> [0,3,47,2,1,6,5]
1 -> [1,0,2,5,3,4,6,7]
2 -> [2,1,6,0,5,3,4,7]
3 -> [3,7]
4 -> [4,2,1,6,0,5,3,7]
5 -> [5]
6 -> [6,2,1,0,5,3,4,7]
7 -> [7]

4.

```
public static ArrayList<Integer> hayCaminoDFS(Digraph g, int inicio, int fin){
        boolean[] visited = new boolean[g.size()];
        ArrayList<Integer> camino = new ArrayList<Integer>();
        Stack<Integer> pila = new Stack<Integer>();
        hayCaminoDFS(g,inicio,fin,visited, pila);
        camino.add(inicio);
        while(!pila.isEmpty()){
           camino.add(pila.pop());
        }
        return camino;
    }

private static boolean hayCaminoDFS(Digraph g, int nodo, int objetivo, boolean[] visited, Stack<Integer> c){
        visited[nodo] = true;
        ArrayList<Integer> sucesores = g.getSuccessors(nodo);
        if(sucesores != null){
           for(Integer sucesor : sucesores){
              if(!visited[sucesor] && (sucesor == objetivo || hayCaminoDFS(g, sucesor, objetivo, visited, c))){
                 c.push(sucesor);
                 return true;
              }
           }
        }
        return false;
    }
```

5. 1
ni, nj
2*T(n-1)

DOCENTE MAURICIO TORO BERMÚDEZ
Teléfono: (+57) (4) 261 95 00 Ext. 9473. Oficina: 19 - 627
Correo: mtorobe@eafit.edu.co