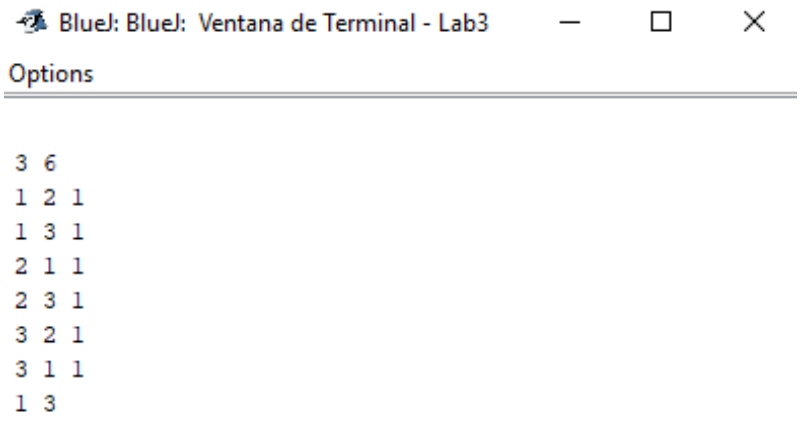
Laboratorio Nro. 3: Vuelta Atrás (BackTracking)

Luisa María Vásquez Gómez
Universidad Eafit
Medellín, Colombia
lmvasquezg@eafit.edu.co

Juan José Parra Díaz
Universidad Eafit
Medellín, Colombia
jjparrad@eafit.edu.co

1) Simulacro de preguntas de sustentación de Proyectos

7. Prueba del 1.6 (ejercicio en línea) con grafo completo.



8. Este recorre todos los sucesores de un nodo, sin repetir, pero antes de ello, revisa que el peso del camino acumulado al nodo que va a visitar sea inferior al peso del camino con menor peso encontrado. Si decide que el peso es menor, guarda el camino en un entero y pasa al siguiente nodo, pero si es mayor, se devuelve hasta donde fue menor por última vez y retira el camino agregado del entero. Al hallar un camino, se sabe que fue uno con menor peso que el anterior, entonces lo guarda y junto a este, el peso que consumió. Al final, se imprime el camino más corto encontrado.

3) Simulacro de preguntas de sustentación de Proyectos

1. Aparte de la fuerza bruta, la cual revisa absolutamente todas las soluciones posibles a un problema, y el backtracking, que descarta los caminos erróneos antes de entrar en ellos, ahorrándose así explorar dentro de los caminos sin salida, existen los algoritmos voraces, los cuales no siempre llegan al camino más corto, pero sí a uno suficientemente eficiente, eligiendo una de las opciones y yéndose por ella, sin arrepentimiento, teniendo así un tiempo de ejecución muy bajo debido a su poca complejidad, pero con resultados menos precisos.
2.

Valor de N	Tiempo de ejecución
4	0ms
5	0ms
6	0ms
7	0ms
8	0ms
9	0ms
10	16ms
11	47ms
12	219ms
13	1358ms
14	9155ms
…	
32	+500seg
N	**O(n!)**

3. BFS consiste en revisar primero todos los hijos de un nodo, antes de pasar a sus nietos, mientras que DFS revisa primero toda la descendencia de un hijo para luego pasar al siguiente, hasta que se acaben los nodos. En el recorrido de grafos, no se puede decir cuál es mejor sin antes revisar qué tipo de grafo es. Si es un grafo que tiene pocas ramas pero muy largas o si su solución no se encuentra en los extremos del mismo, es mejor usar BFS, puesto que recorre de manera más amplia, mientras que si el grafo tiene muchas ramas por nodo pero poco profundas, DFS es la mejor elección.

4. El algoritmo de bloques es una variación de DFS, en el cual los nodos se agrupan en bloques, lo que hace más eficiente la búsqueda en estos. El algoritmo de dijkstra es otro algoritmo de búsqueda en grafos, el cual se asimila a BFS en que usa colas de prioridad. Este algoritmo el más eficiente a la hora de hallar el camino más corto ente dos nodos. Funciona explorando todos los nodos y calculando las distancias, y luego recorriendo y escogiendo

cuál de ellos lo puede llevar a la solución más óptima, es decir, mira las distancias y las guarda mientras va recorriendo el grafo.

5. El ejercicio en línea 2.1 funciona leyendo por medio de un BufferedReader, la información de la entrada, con la cual va creando un grafo, línea por línea, hasta que no haya más arcos que agregar. Con el grafo creado, se comienza a explorar por medio de un algoritmo DFS recursivo. Este recorre todos los sucesores de un nodo, sin repetir, pero antes de ello, revisa que el peso del camino acumulado al nodo que va a visitar sea inferior al peso del camino con menor peso encontrado. Si decide que el peso es menor, guarda el camino en un entero y pasa al siguiente nodo, pero si es mayor, se devuelve hasta donde fue menor por última vez y retira el camino agregado del entero. Al hallar un camino, se sabe que fue uno con menor peso que el anterior, entonces lo guarda y junto a este, el peso que consumió. Al final, se imprime el camino más corto encontrado.

6.

```
public DigraphAM inputManager() throws IOException{                    O(a)
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));   C
        String entrada = br.readLine();                                 C
        String[] uno = entrada.split(" ");                              C
        nodos = Integer.parseInt(uno[0]);                               C
        int arcos = Integer.parseInt(uno[1]);                           C
        visitados = new boolean[arcos];                                 C
        DigraphAM g = new DigraphAM(arcos);                             C
        for (int i = 0; i < arcos; i++){                                C*a
            entrada = br.readLine();                                    C*a
            String[] dos = entrada.split(" ");                          C*a
            int origen = Integer.parseInt(dos[0]);                      C*a
            int destino = Integer.parseInt(dos[1]);                     C*a
            int peso = Integer.parseInt(dos[2]);                        C*a


            g.addArc(origen, destino, peso);                            C*a
        }
        return g;                                                       C
```

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

```
    }

    public void recorrer(Digraph g, int origen, int destino, int peso, int camino){          O(n!)
        if(visitados[origen]){                                                              C
            return;                                                                         C
        }
        if(origen == destino){                                                              C
            pesoMin = peso;                                                                 C
            caminoFinal = camino;                                                           C
        } else {                                                                            C
            ArrayList sucesores = g.getSuccessors(origen);                                  O(n)
            if(sucesores == null){                                                          C
                return;                                                                     C
            }
            for(int i = 0; i < sucesores.size(); i++){                                      C*n
                visitados[origen] = true;                                                   C*n
                peso += g.getWeight(origen, (int)sucesores.get(i));                         C*n
                if(peso <= pesoMin){                                                        C*n
                    camino = camino * 10 + (int)sucesores.get(i);                           C*n
                    recorrer(g, (int)sucesores.get(i), destino, peso, camino);              T(n-1)
                }
                peso -= g.getWeight(origen, (int)sucesores.get(i));                         C*n
                camino /= 10;                                                               C*n
                visitados[origen] = false;                                                  C*n
            }
        }
    }

    public static void main(String [] args){                                               O(n!)
        EjercicioEnLinea gr = new EjercicioEnLinea();                                       C
        System.out.println();                                                               C
        try{                                                                                C
```

```
        DigraphAM g = gr.inputManager();                                    O(a)
        gr.recorrer(g, 1, gr.nodos, 0, 1);                                  O(n!)
    } catch(IOException e){}                                                C
    ArrayList<Integer> solucion = new ArrayList<Integer>();                 C
    do{                                                                     C*n
        solucion.add(gr.caminoFinal % 10);                                  C*n
        gr.caminoFinal /= 10;                                               C*n
    } while  (gr.caminoFinal > 0);                                          C*n
    for(int i = solucion.size()-1; i >= 0; i--){                            C*n
        System.out.print(solucion.get(i) + " ");                            C*n
    }
}
```

Complejidad = O(n!)

7. En el cálculo de la complejidad, la variable 'n' representa el número de nodos, de tal manera que ésta se puede leer como el factorial del número total de nodos de un grafo.

4) Simulacro de parcial

1.

a) n-a,a,b,c
b) res,solucionar(n-b,a,b,c)+1
c) res,solucionar(n-c,a,b,c)+1

2.

a) graph.length-1
b) v,graph,pat,pos
c) graph,path,pos+1

3.

a) DFS
0 = 0 3 7 4 2 1 5 6
1 = 1 0 3 7 2 4 6 5
2 = 2 1 0 3 7 5 4 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

b) BFS
0 = 0 3 4 7 2 1 6 5
1 = 1 0 2 5 3 4 6 7
2 = 2 1 4 6 0 5 3 7
3 = 3 7
4 = 4 2 1 6 0 5 3 7
5 = 5
6 = 6 2 1 4 0 5 3 7
7 = 7

4.

```
public static boolean hayCaminoDFS(Digraph g, int a, int b){
boolean visitados[] = new boolean[g.size()];
return hayCaminoDFSAux(g,a,b,visitados);
}
private static boolean hayCaminoDFSAux(Digraph g, int v, int w, boolean[]
visitados){
visitados[v] = true;
ArrayList<Integer> sucesores = g.getSuccessors(v);
if(sucesores != null){
for (int i=0;i< sucesores.size();i++){
if(!visitados[sucesores.get(i)]&&(sucesores.get(i)==w||
hayCaminoDFSAux(g,sucesores.get(i),w,visitados))){
return true ;
}
}
return false;
}else{
if(v==w){
return true;
}else{
return false;
```

```
}
}
```

5.

a) 1
b) ni,nj
c) Tn= 2*T(n-1)+ C

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