Laboratorio Nro. 3: Backtracking

Eduard Damiam Londoño
Universidad Eafit
Medellín, Colombia
edlondonog@eafit.edu.co

Gonzalo Garcia
Universidad Eafit
Medellín, Colombia
gggarcia@eafit.edu.co

1) código

8. para encontrar el camino mas corto se uso DFS, lo que se hace es coger el vértice inicial y obtener sus sucesores, si el vértice tiene sucesores entonces se empiezan a recorrer todos unos por 1, a su vez se agrega el vértice en que esta a una lista en esta lista la cual guarda un camino posible, también se marca el vértice como ya recorrido y se suma el peso entre el vértice y el sucesor, luego se hace lo mismo con su sucesor, si en algún momento se alcanza el vértice objetivo entonces se compara la suma total de lo recorrido hasta el momento con la "definitiva" si la definitiva es mayor entonces se intercambien y el camino posible pasa a ser el camino "definitivo", si en vez de eso en alguna iteración la suma es mayor a la "definitiva" entonces no sigue ese camino y se devuelve, cada vez que el algoritmo se devuelve para probar otro camino, se desmarcan los vértices "ya recorridos", y este proceso se repite con todos los caminos posibles.

3) Simulacro de preguntas de sustentación de Proyectos

1. para resolver el problema del camino más cercano, además de fuerza bruta y backtracking existen multitud de algoritmos heurísticos, uno de ellos es el del vecino mas cercano el cual compara todos los vecinos del un vértice y decide irse por el arco que tenga menos peso, otro tipo de algoritmos son los algoritmos genéticos, los cuales se basan en la teoría de la genética, estos algoritmos lo que hacen es coger un conjunto de soluciones del problema he ir cruzándolas o mutándolas con la esperanza de conseguir una solución mejor y otro tipo de algoritmos que se pueden usar para este tipo de problemas son los de enjambre de hormigas, los cuales se basan en el comportamiento de las hormigas, las cuales al recorrer un camino dejan feromonas, si una hormiga encuentra comida (o en nuestro caso el vértice objetivo), regresa dejando otra vez feromonas, mientras mas corto sea el camino menos se evaporaran las feromonas y de esta manera podemos saber cuál podría ser el mas corto.

2.

Valor de N	Tiempos de ejecución
4	**2.187 x10^-7 min**
5	**3.308x10^-7 min**
6	**4.322x10^-7 min**
7	**5.496x10^-7 min**
8	**2.476x10^-6 min**
9	**2.700x10-6 min**
10	**3.75 x10^-6 min**
11	**5.82x10^-6 min**

12	**9.29 x10^-6 min**
13	**1.40x10^-5 min**
14	**2.46x10^-5 min**
15	**5.22 x10^-5 min**
16	**7.18 x10^-5 min**
17	**1x10^-4 min**
18	**0.0002 min**
19	**0.00065 min**
20	**0.0001 min(aquí se pasó de nano a millis)**
21	**0.00047 min**
22	**0.0009 min**
23	**0.001 min**
24	**0.0016 min**
25	**0.002 min**
26	**0.003 min**
27	**0.012 min**
28	**0.165 min**
29	**0.022 min**
30	**0.48 min**
31	**0.611 min**
32	**0.8 min**
N	**O(n^n)**

3. El usar DFS o BFS depende mucho de los datos que se tienen y de lo que se quiere hacer, por ejemplo, DFS puede ser usado para buscar ciclos en un grafo, o determinar si un grafo es conexo o no, si se necesita el camino entre 2 vértices BFS suele ser más rápido que DFS o en Facebook con la sugerencia de amigos donde la idea es no recorrer demasiado, BFS es lo mejor, otra cosa a tener en cuenta es el tamaño del grafo y como esta distribuido ya que esto puede hacer que con uno de los métodos, la pila se llene más rápido que con el otro.
4. Los algoritmos D* consisten en resolver el problema de encontrar la ruta basándose en suposiciones, se hacen suposiciones sobre el camino y a medida que se avanza en la ruta se va agregando la nueva información sobre esta y más suposiciones, otra forma es con una búsqueda voraz los cuales se basan en una heurística y dan un resultado sin preocuparse si hay otro mejor, los algoritmos A* son algoritmos que tienen en cuenta el coste del camino recorrido, el coste de la heurística y escogen el camino con el meno coste estimado, otro método es la búsqueda por franjas que consiste en usar 2 tablas hash para guardar el peso de la ruta desde el origen hasta el nodo actual, y el peso estimado desde el nodo actual hasta el nodo destino, con esto el tiempo para consultar cada nodo se reduce (buscar en una tabla hash es O(1)) , también se usa una lista doblemente enlazada con 3 espacios, uno para el nodo inicial, otro para el actual y otro para el final, luego con una variable de clasificación se va iterando sobre la lista, si la lista se finaliza y no se a encontrado ningún nodo objetivo, se inicia desde el principio.
5. Para el ejercicio 2.1 se crea un grafo hecho con listas de adyacencia con las especificaciones del usuario y ciertas restricciones, si el usuario no cumplía con las restricciones, el dato insertado no se toma en cuenta y se le pide otro, para calcular el camino optimo desde el punto inicial hasta el punto final se uso el mismo algoritmo que para el punto 1.6, con la diferencia de que como el grafo podía no estar completo entonces se agrego un bloque try catch para manejar el caso de que un vértice no tenga mas sucesores.
6.

```
public static void main (String args []){
   DigraphAL g = leer(); O(n)
```

```
    if (g == null){ //C1
        System.out.println("no se puede con los datos proporcionados"); C2
    }

    ArrayList respuesta = CMC(g,1,g.size); //O(a^x+m)
    if (respuesta.isEmpty() || respuesta == null) { //C3
        System.out.print("-1");//C4
    }
    else {
        for(int i=0; i< respuesta.size(); ++i){//O(z)
            System.out.println(respuesta.get(i)+ " "); //C5
        }
    }
}

public static DigraphAL leer(){ O(n)
    Scanner input = new Scanner(System.in); // C1
    String a = input.nextLine(); //C2
    String [] in = a.split(" "); //C3
    if(Integer.parseInt(in[0])<2 || Integer.parseInt(in[0])>105){ //C4
        return null; //C5
    }
    if(Integer.parseInt(in[1])< 0 ||Integer.parseInt(in[1])> 105 ){//C6
        return null; //C7
    }
    DigraphAL g = new DigraphAL(Integer.parseInt(in[0])); //C8
    String []inp; C//9
    for(int i = 0;i< Integer.parseInt(in[1]);++i){ //n
        a=input.nextLine(); //C10*n
        inp = a.split(" "); //C11*n
        if (Integer.parseInt(inp[0])< 1 || Integer.parseInt(inp[0])> g.size ){ //C12 *n
            System.out.println("dato incorrecto"); //C13*n
            i--;   //C14*n
            continue; //C15*n
        }else if(Integer.parseInt(inp[1])< 1 || Integer.parseInt(inp[1])> g.size ){ //C16*n
            System.out.println("dato incorrecto"); //C17*n
            i--;  //C18*n
            continue; //C19*n
        }else if(Integer.parseInt(inp[2])< 1 || Integer.parseInt(inp[2])> 106){//C20*n
            System.out.println("dato incorrecto");//C21*n
            i--; C22*n
            continue; C23*n
        }
        g.addArc(Integer.parseInt(inp[0]),Integer.parseInt(inp[1]),Integer.parseInt(inp[2])); C24
    }
    return g; C25
}

public static ArrayList CMC (DigraphAL g, int inicio, int Final){
    boolean[] recorridos = new boolean[g.size]; //C1
```

```
        if (inicio == Final){ //C2
            return null; //C3
        }
        ArrayList <Integer> camino = new ArrayList <> (); //C4
        ArrayList <Integer> temp = new ArrayList<> (); //C5
        int[] tot = {Integer.MAX_VALUE}; //C6
        camino = CMCAux(g,recorridos,inicio,Final,camino,tot,temp,0);
        camino.add(0,1); //C7
        return camino;//C8
    }

    public static ArrayList CMCAux (DigraphAL g, boolean[] recorridos, int v, int f,ArrayList camino,int[] tot,ArrayList temp,int
            tempo){
        if (v == f){ //C1
            if (tempo < tot[0]){ //C2
                tot[0] = tempo; //C3
                camino = (ArrayList)temp.clone(); //C4
                return camino; //C5
            }
        }
        else if (tempo > tot[0]){ //C6
            return camino; //C7
        }try {
            ArrayList sucesores = g.getSuccessors(v); //O(a)
            if(!sucesores.isEmpty()){ //C8
                recorridos[v-1] = true; //C9
                for(int i =0; i<sucesores.size();++i) {//O(a)
                    if(recorridos[(int)sucesores.get(i)-1]){ // C10*a
                        continue;//C11 *a
                    }
                    temp.add(sucesores.get(i)); //C11 *a
                    camino = CMCAux(g, recorridos, (int)sucesores.get(i), f, camino, tot, temp, tempo + g.getWeight(v, (int) sucesores.get(i))); //O(a^x+m)
                    Integer a = (int)sucesores.get(i); //C12*a
                    temp.remove(a); //C13
                }
                recorridos[v-1] = false; //C14*a
            }
            return camino; //C15 *a

        }catch (Exception e){
          return camino; //C16*a
        }
    }

}
```

En la función CMCAUX la llamada recursiva depende de 2 variables x y m las cuales se reducen en cada iteración, por lo tanto

T(x,m) = aT(x-1,m-1) => a^x+m y como no sabemos cual es mayor la complejidad queda O(a^n+m)

La complejidad total queda

O(C+n+(a^x+m)+z) luego por ley de la suma =>
O(n+(a^x+m)+z)

7. La variable n representa la cantidad de arcos que especifica el usuario ya que de esto depende la cantidad de veces que se toman los datos, a representa la cantidad de sucesores de un vértice ya que de este depende la cantidad que se ejecutan las iteraciones recursivas, x representa la cantidad de vértices que faltan por ser revisados, la m representa la sumatoria de los pesos ya que si esta se pasa de un tope las iteraciones se terminan, y por ultimo z es el tamaño de la respuesta ya que de esta depende un ciclo.

4) Simulacro de Parcial

1. *a.*
   solucionar(n-a,a,b,c);
   b.
   Math.max(res,solucionar(n-b,a,b,c)+1);
   c.
   Math.max(res,solucionar(n-c,a,b,c)+1);
2. a. 0 → 3 → 7
      1 → 0 → 3 → 7
      2 → 1 → 0 → 3 → 7
      3 → 7
      4 → 2 → 1 → 0 → 3 → 7
      5
      6 → 2 → 1 → 0 → 3 → 7
      7
   b. 0 → 3 → 4 → 7 → 2 → 1 → 6 → 5
      1 → 0 → 5 → 3 → 4 → 7 → 2 → 6
      2 → 1 → 4 → 6 → 0 → 5 → 3 → 7
      3 → 7
      4 → 2 → 1 → 6 → 0 → 5 → 3 → 7
      5
      6 → 2 → 1 → 4 → 0 → 5 → 3 → 7
      7
3.

4. unCamino(Graph g, int p, int q){
     boolean visitado[] = new boolean[g.size()];
     LinkedList<Integer> camino = new LinkedList<Integer>();
     LinkedList<Integer> v = new LinkedList<>();
     v.add(p);
     camino.add(p);
     while(v.isEmpty()== false){

```
        int ver = v.poll();
        ArrayList<Integer> sucesores = g,getSuccessor(ver);
        If(sucesores != null){
            For (int i =0; i<sucesores.length; ++i){
                v.add(sucesor.get(i));
                if(!visitados[sucesor.get(i)]){
                    camino.add(sucesor.get(i));
                }
                If (sucesor.get(i)==q){
                Return camino;
                 }
            visitados[sucesor.get(i)] = true;
             }
        }
    }
Return camino;
}
```

5. 5.1 return 1+lcs(i-1,j-1,s1,s2);
5.2 return Math.max(ni,nj);
5.3 T(n)= c * 2T(n-1);

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