Laboratorio Nro. 3: Vuelta atrás (Backtrarcking)

Santiago Catrillón Galvis
Universidad Eafit
Medellín, Colombia
scastrillg@eafit.edu.co

Alejandro Arroyave Bedoya
Universidad Eafit
Medellín, Colombia
arroyaveb@eafit.edu.co

Luis Javier Palacio Mesa
Universidad Eafit
Medellín, Colombia
ljpalaciom@eafit.edu.co

1)

8. El algoritmo, contenido en el método auxiliar "dfs()" está basado en un recorrido clásico de dfs que retorna un booleano. Se retorna un booleano para saber si efectivamente existe o no un camino. Así pues, recorriendo los sucesores de cada vértice lo primero que hacemos es añadir a la lista (que nos llevará el registro del camino), cada sucesor y si no uno de ellos no ha sido visitado previamente y si es el objetivo o hay un camino de él hacia el objetivo (recursividad) retornamos verdadero, de lo contrario simplemente borramos este de la lista porque no es parte del camino.

3) Simulacro de preguntas de sustentación de Proyectos

1. –Dijkstra: consiste en ir explorando todos los caminos más cortos que parten del vértice origen y que llevan a todos los demás vértices; cuando se obtiene el camino más corto desde el vértice origen hasta el resto de los vértices que componen el grafo, el algoritmo se detiene.

-Bellman-ford: este algoritmo nos indica el camino mas corto en un grafo dirijido y cuyos pesos pueden ser negativos.

- Búsqueda A*: utiliza la heurística y la convinacion de BFS y DFS para encontrar el camino mas corto de una forma mas rapida - Floyd-Warshall: resuelve el camino minimo entre todos los vertices en una sola ejecucion, utilizando programacion dinamica.

2.

Valor de N	Tiempo de ejecución
4	0
5	0
6	0
7	0
8	1

9	0
10	0
11	0
12	0
13	0
14	4
15	1
16	8
17	4
18	27
19	2
20	164
21	11
22	1657
23	28
24	443
25	58
26	513
27	600
28	4204
29	2238
30	91363
31	22854
32	161033
N	O()

3. Esto depende de la estructura del grafo y de la ubicación de la solución.
Tendriamos dos casos principales. El primero seria para un grafo en el que los vertices tienen muchos sucesores en profundidad, en este caso si se sabe que la solucion esta en lo profundo, nos conviene usar DFS porque llegaremos mucho mas rapido, encambio si la solucion es indeterminada, será mejor usar BFS para no tener que recorrer gran gantidad de sucesores en lo profundo. El Segundo es cuando tenemos muchos sucesores a lo ancho del grafo, en este caso, si la solución no esta lejos de la raiz, es mejor usar BFS, pero si la solucion es incierta y el grafo es muy ancho, lo mejor será usar DFS.
4. -Búsqueda A*: utiliza la heurística y la convinacion de BFS y DFS para encontrar el camino mas corto de una forma mas rapida

-IDA*: es una modificacion del DFS anexandole la informacion heurística para saber cual de los nodos expandir.
- Fringe Search: utiliza el algoritmo heurístico de busqueda por franjas que es una combinacion entre IDA* y A*
- D*: funciona con una seleccion iterative de un nodo de una lista abierta.

5. El algoritmo está basado dfs. Sin embargo en vez de usar un arreglo de booleanos usamos un arreglo de enteros que contendrá el costo mínimo de cada vértice desde el vértice inicial. El algoritmo consiste en ir mejorando cada vez más los costos mínimos hasta que ya no se pueda más. Para esto inicializamos todas las posiciones del arreglo en infinito (o el valor más grande posible) e inicializamos el costo del nodo inicial en un costo de 0, esto es porque el costo para de un nodo a sí mismo debería ser 0 en estos grafos.
Tenemos una lista para llevar el control de la ruta que se está llevando y una pareja global (pair) que lleva el costo más mínimo y la ruta del mismo encontrados hasta el momento. También poseemos una variable acumulativa que nos permite saber en cierta ruta, cuanta distancia acumulada llevamos. Finalmente, cuando llegamos a nuestro objetivo, comparamos este costo con el de la pajera global, y si es un costo menor, le asignamos a esta pareja una nueva ruta y nuevo costo hasta que esta no se pueda mejorar más.

6.
```
public static ArrayList<Integer> costoMinimo(Digraph g, int inicio, int fin) {        int costo[] = new
int[g.size];        for (int i = 0; i < costo.length; i++) {  C1* n            costo[i] = Integer.MAX_VALUE;
 }
 costo[inicio] = 0;
 ArrayList<Integer> lista = new ArrayList<>();
lista.add(inicio);        dfs(g, inicio, fin, costo, lista,
0);        return (ArrayList<Integer>) ruta.first;
  }
```

```
  private static void dfs(Digraph g, int nodo, int objetivo, int[] costo, ArrayList<Integer> list, int acum) {
      ArrayList<Integer> sucesores = g.getSuccessors(nodo);
if (nodo == objetivo) {
        Pair pair = new Pair(list.clone(), acum);  C4n
if (ruta == null) {              ruta = pair;
        } else if ((int) pair.second < (int) ruta.second) {
ruta = pair;
        }
return;
      }
      if (sucesores != null) {
        for (Integer sucesor : sucesores) { C2n
int peso = g.getWeight(nodo, sucesor);C3n*n
if (costo[sucesor] > acum + peso) {
list.add(sucesor);
            costo[sucesor] = acum + peso;
            dfs(g, sucesor, objetivo, costo, list, acum + peso); n*T(n-1)
list.remove(list.size() - 1);
          }
        }
```

```
        }
    }
```

T(n) = C1*n + C2*n + C3n*n + n*T(n-1)
T(n) = C1*n + C2*n + C3*n + C4n*n! //Cada vez que entre al caso base debe hacer una operación que tal vez cuesta O(n) (clonar lista)
T(n) es O(C1*n + C2*n + C3*n + C4n*n!)
T(n) es O(C4n*n!)
T(n) es O(n*n!)

7.

En el peor de los casos se deben visitar todos los nodos antes de encontrar el vértice objetivo, así que estamos tratando con las permutaciones de n vértices.
Es decir, n es el tamaño del grafo.

4) Simulacro de Parcial

1. *a*
```
int res = solucionar(n - a, a, b, c) + 1;
```
b
```
res = Math.max(res, solucionar(n - b, a, b, c) + 1);
```
c
```
res = Math.max(res, solucionar(n - c, a, b, c) + 1);
```

2. a
```
if (pos == path.length) {
```
b
```
if (sePuede(v, graph, path, pos)) {
```
c
```
if (cicloHamilAux(graph, path, pos + 1)) {
```

3.

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

b)
0 -> [0, 3, 4, 7, 2, 1, 6, 5]
1 -> [1, 0, 2, 5, 3, 7, 4, 6]
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 ArrayList<Integer> camino(Digraph g, int inicio, int fin) {
    boolean[] visitados = new boolean[g.size()];
ArrayList<Integer> list = new ArrayList<>();        list.add(inicio);
    if (dfs(g, inicio, fin, visitados, list)) {
```

```
            return list;
        }
        return null;

    }

    private static boolean dfs(Digraph g, int nodo, int objetivo, boolean[] visitados, ArrayList<Integer> list) {
        ArrayList<Integer> sucesores = g.getSuccessors(nodo);
visitados[nodo] = true;         if (sucesores != null) {
            for (Integer sucesor : sucesores) {
list.add(sucesor);
                if (!visitados[sucesor] && (sucesor == objetivo || dfs(g, sucesor, objetivo, visitados, list))) {
return true;
                } else {
                    list.remove(list.size() - 1);
                }
            }
        }
        return false;
    }
```

5. 1. `return 1 + lcs(i - 1, j - 1, s1, s2);`
   2. `return Math.max(ni, nj);`
   3. 2(T(n-1)) + c