Algorithmique & Programmation Avancé

Report Tutorial course 4 (Part 1 & 2)

Algorithmic for traversing a graph and shortest paths

DAFIT, Jiawei

Date: 09/05/2017

Table des matières

I. The Depth Search First algorithm

1. Given a graph, create a function called dfs(Graph G) that performs the deep first search (DFS) algorithm for visiting the graph G. This function must return the list of vertices in the order of their first encounter. In case of choice, the vertex with the smallest identifier will be chosen.

```
public ArrayList<Integer> dfs(Graph G, int startingNode){

    //Un ArrayList contenant les identifiants des noeuds visite dans l'ordre
    ArrayList<Integer> visited = new ArrayList<Integer>();
    //Stack pour la recherche DFS
    Stack<Integer> stack = new Stack<Integer>();
    Node[] adj = G.getAdj();
    //rechercher la position du noeud dans le tableau adj
    int startingPosition = G.getNodePosition(startingNode);
    //ajout de la permiere valeur dans visited et le stack
    visited.add(adj[startingPosition].nodeId);
    stack.push(adj[startingPosition].nodeId);
    //tant que le stack est non vide
    while(!stack.empty()){
        //je regarde les voisins pour le dernier element du stack
        int stackLastElemPosition = G.getNodePosition(stack.peek());
        Stack<Integer> stackVoisin = stackVoisin(adj[stackLastElemPosition]);
        //tant que le stack contenant les voisins est non vide je regarde s'ils sont visited
        while (!stackVoisin.empty()) {
            //si le voisin avec le plus petit id n'est visited
            if (!visited.contains(stackVoisin.peek())) {
                //le voisin est ajouter dans la liste des visited et ajouter dans le stack,
                //puis on arrete la boucle while
                visited.add(stackVoisin.peek());
                stack.push(stackVoisin.peek());
                break;
            }
            else
            {
                //sinon on enleve le dernier element du stack des voisins
                stackVoisin.pop();
            }
        }
        //si le stack des voisins s'est vidé cela signefie que pour le noeud en question
        //tous les voisins ont déjà été visité
        //donc on l'enleve du stack

        if (stackVoisin.empty()) {
            stack.pop();
        }
    }
    return visited;
```

```java
//Creation stack des voisins d'un noeud en ordre decroissant
public Stack<Integer> stackVoisin(Node node){
    //Initialisation d'une ArrayList vide
    ArrayList<Integer> listVoisin = new ArrayList<Integer>();
    //Initialisation d'un Stack vide
    Stack<Integer> stack = new Stack<Integer>();
    //Initialisation de la boucle avec le 1er edge du noeud
    Edge edge = node.firstEdge;
    //Ajouter les voisins du noeud dans la list
    while (edge != null) {
        //ajout dans array list
        listVoisin.add(edge.edgeID);
        //next edge
        edge = edge.nextEdge;
    }
    //Trier la list, ordre croissant
    Collections.sort(listVoisin);
    //Trier en ordre decroissant
    Collections.reverse(listVoisin);
    //Remplissage du stack du plus grand valeur au plus petit
    for (int valeur : listVoisin) {
        stack.push(valeur);
    }
    return stack;
}
```

2. One important application of the depth first search algorithm is to find the connected components of a graph. Write a function cc(Graph G) that takes as input a simple graph and determines the number of connected components. The function cc(Graph G) must use the DFS algorithm. Write a function called isConnected() that returns true if the graph is connected, false otherwise.

```
public int cc(Graph G){
    //List pour stocker tous les noeuds visited
    ArrayList<Integer> visited = new ArrayList<Integer>();
    //List contenant tous les component
    ArrayList<ArrayList<Integer>> componentList = new ArrayList<ArrayList<Integer>>();
    Node[] adj = G.getAdj();
    //Pour chaque noeud
    for (int i = 0; i < adj.length; i++) {
        //je verifie s'il sont visited
        if (!visited.contains(adj[i].nodeId)) {
            //s'ils sont non visited je realise une dfs en partant de ce noeud
            ArrayList<Integer> component = dfs(G, adj[i].nodeId);
            //j'ajoute tous les noeuds visited lors du dfs dans la liste visited
            componentList.add(component);
            //j'ajoute aussi la liste du resultat issue du dfs dans la liste des components
            visited.addAll(component);
        }
    }
    return componentList.size();
}

public boolean isConnected(Graph G){
    //je cherche le nombre de component
    int nombreDeComponent = cc(G);
    //si le graph possede 1 component alors il est connected sinon le graph est no connected
    if (nombreDeComponent == 1) {
        return true;
    }
    else
    {
        return false;
    }
}
```

3. Test the dfs(.) and cc(.) functions with the graph is the graph-DFS-BFS.txt file. Consider as starting node the node 5. What is the order of the first encounter of the nodes? How many components does the graph have? Is it connected?

Dans la méthode main nous avons :

```
Graph graph = new Graph("graph-DFS-BFS.txt");
Dfs rundfs = new Dfs();
ArrayList<Integer> listdfs = rundfs.dfs(graph, 5);
System.out.println("the order of the first encounter of the node 5 by dfs : " + listdfs);
int nombreComponent = rundfs.cc(graph);
System.out.println("The graph have : " + nombreComponent + " components");
boolean isConnected = rundfs.isConnected(graph);
System.out.println("Connected graph : " + isConnected);
```

Output console :

```
the order of the first encounter of the node 5 by dfs : [5, 2, 1, 3, 4, 6, 7]
The graph have : 1 components
Connected graph : true
```

II. Breadth Search First algorithm

1. Given a graph, create a function called bfs(Graph G) that performs the breadth first search (BFS) algorithm for visiting the graph G. This function must return the list of vertices in the order of their first encounter. In case of choice, the vertex with the smallest identifier will be chosen.

```
public ArrayList<Integer> bfs(Graph G, int startingNode){
    //initialisation d'une liste pour stocker les noeuds visited
    ArrayList<Integer> visited = new ArrayList<Integer>();
    //initialisation de queue
    Queue<Integer> queue = new LinkedList<>();
     Node [] adj = G.getAdj();
     //chercher la position du noeud de depart dans le tableau adj
    int startingNodePosition = G.getNodePosition(startingNode);
    //ajouter le noeud de depart dans la liste des visited et dans queue
    visited.add(adj[startingNodePosition].nodeId);
    queue.add(startingNode);
    //tant que queue est non vide
    while (!queue.isEmpty()) {
        //on prend le dernier element de queue et on cherche ses voisins
        int queueLastElemPosition = G.getNodePosition(queue.peek());
        ArrayList<Integer> listVoisin = listVoisin(adj[queueLastElemPosition]);
        //pour chaque voisins trouves
        for(int valeur : listVoisin)
        {
            //on regarde s'ils sont visited
            if (!visited.contains(valeur))
            {
                //s'ils ne sont pas visited, on l'ajout dans la liste visited et
                //on l'ajout dans queue sinon on passe au voisin suivant
                visited.add(valeur);
                queue.add(valeur);
            }
        }
        //on enleve un element de queue
        queue.remove();
    }
    return visited;
}

            //Renvoie les voisins d'un noeud en ordre decroissant
            public ArrayList<Integer> listVoisin(Node node){
                //Initialisation d'une ArrayList vide
                ArrayList<Integer> listVoisin = new ArrayList<Integer>();

                //Initialisation de la boucle avec le 1er edge du noeud
                Edge edge = node.firstEdge;
                //Ajouter les voisins du noeud dans la list
                while (edge != null) {
                    //ajout dans array list
                    listVoisin.add(edge.edgeID);
                    //next edge
                    edge = edge.nextEdge;
                }
                //Trier la list, ordre croissant
                Collections.sort(listVoisin);

                return listVoisin;
            }
```

2. Test the bfs(.)function with the graph is the graph-DFS-BFS.txt file. Consider as starting node the node 5. What is the order of the first encounter of the nodes? How many components does the graph have? Is it connected?

Dans la méthode main nous avons :

```
Bfs runbfs = new Bfs();
ArrayList<Integer> listbfs = runbfs.bfs(graph, 5);
System.out.println("the order of the first encounter of the node 5 by bfs : " + listbfs);
```

Output console :

```
The graph have : 1 components
Connected graph : true
the order of the first encounter of the node 5 by bfs : [5, 2, 6, 1, 3, 7, 4]
```

III. Breadth Search First (BFS) for shortest paths in unweighted (di)graphs

Create a class called BFSShortestPaths. This class will implement the BFS algorithm for shortest paths from a given vertex s (seen in lecture 4). This class will contain:

1. 3 vertex-indexed arrays: boolean[] marked, int[] previous and int[] distance. marked[v] is set to true if v has been visited (false otherwise). previous[v] indicated the vertex preceding v on the shortest path. Distance[v] represents the distance (in number of edges) from the source vertex s to vertex v.

```
public class BFSShortestPaths {
    private int startingNode;
    private int [] nodes;
    private boolean[] marked;
    private int[] previous;
    private int[] distance;

    public int[] getNodes() {
        return nodes;
    }

    public boolean[] getMarked() {
        return marked;
    }


    public int[] getPrevious() {
        return previous;
    }


    public int[] getDistance() {
        return distance;
    }
```

2. Modify the function bfs(graph G) called bfs(Digraph G, int s) which executes the BFS algorithm to calculate all the shortest paths from the root vertex s. This function will update the marked, previous and distance arrays. It will be of void type.

```
public void bfs(Digraph G, int s){
    startingNode = s;
    Node[] adj = G.getAdj();
    int taille = adj.length;
    //initialisation des tableaux
    nodes = new int [taille];
    marked = new boolean [taille];
    previous = new int [taille];
    distance = new int [taille];
    //initialisation des valeurs par defaut dans les tableau
    for (int i = 0; i < taille; i++) {
        nodes[i] = adj[i].nodeId;
        marked[i] = false;
        previous[i] = -2;
        distance[i] = Integer.MAX_VALUE;
    }
    //chercher la position du noeud root
    int startingNodePosition = G.getNodePosition(s);

    //initialisation queue
    Queue<Integer> queue = new LinkedList<>();
    //ajout du noeud root dans queue
    queue.add(s);
    //initialisation du valeur distance pour le noeud root a 0
    //du marked a true et du previous a -1
    distance[startingNodePosition] = 0;
    marked[startingNodePosition] = true;
    previous[startingNodePosition] = -1;

    //tant que queue est non vide
    while (!queue.isEmpty()) {
        //on regarde pour le dernier element de queue en stockant son id entant que
        //tempValue et en cherchant ses voisins
        int tempValue = queue.peek();
        int tempValuePosition = G.getNodePosition(tempValue);
        //on eleve un element de queue
        queue.remove();
        //on genere la liste des voisins du noeud en question
        ArrayList<Integer> listVoisin = listVoisin(adj[tempValuePosition]);
        for(int valeur : listVoisin){
            int valeurPosition = G.getNodePosition(valeur);
            //si le voisin n'est pas visited
            if (!marked[valeurPosition]) {
                //on le marked true,on lui affect tempValue comme previous et on
                //lui affect la distance de tempValue +1 et on ajout ce voisin dans queue
                marked[valeurPosition] = true;
                previous[valeurPosition] = tempValue;
                distance[valeurPosition] = distance[tempValuePosition] + 1;
                queue.add(valeur);
            }

        }
    }
}
```

```java
//Renvoie les voisins d'un noeud en ordre decroissant
    public ArrayList<Integer> listVoisin(Node node){
        //Initialisation d'une ArrayList vide
        ArrayList<Integer> listVoisin = new ArrayList<Integer>();

        //Initialisation de la boucle avec le 1er edge du noeud
        Edge edge = node.firstEdge;
        //Ajouter les voisins du noeud dans la list
        while (edge != null) {
            //ajout dans array list
            listVoisin.add(edge.edgeID);
            //next edge
            edge = edge.nextEdge;
        }
        //Trier la list, ordre croissant
        Collections.sort(listVoisin);

        return listVoisin;
    }
```

3. Add a boolean function hasPathTo(int v) which returns true if there is a path from s to v.

```java
public boolean hadPathTo(int v){
    //on cherche la position du noeud v dans les tableaux
    //et on retourn la valeur du tableau marked à la position
    for (int i = 0; i < nodes.length; i++) {
        if (nodes[i] == v) {
            return marked[i];
        }
    }
    return false;
}
```

4. Add the function distTo(int v) which returns the length of the shortest path from s to v.

```java
public int distTo(int v){
    //on cherche la position du noeud v dans les tableaux
    //et on return la valeur du tableau distance à cette position
    for (int i = 0; i < nodes.length; i++) {
        if (nodes[i] == v) {
            return distance[i];
        }
    }
    return -1;
}
```

5. Add the function printSP(int v) which prints the shortest path from s to v.

```
public void printSp(int v){
    //on cherche la position du noeud v dans les tableaux
    int vPosition = -1;
    for (int i = 0; i < nodes.length; i++) {
        if (nodes[i] == v) {
            vPosition = i;
            break;
        }
    }
    //si le noeud v est visited et la position diff de -1
    // ie. le noeud v existe
    if (marked[vPosition] && vPosition != -1) {
        //initialisation d'une list pour stocker la path
        ArrayList <Integer> sp = new ArrayList<Integer>();
        //on ajout la destination, ie noeud v
        sp.add(v);
        //on stock la position du noeud v a l'aide de tempPosition
        int tempPosition = vPosition;
        //tant que tempPosition n'est pas le noeud root
        //ie. distance > 0
        while (distance[tempPosition] > 0) {
            //on stock la valeur du tableau previous à la position tempPosition
            // avec une autre valeur temporaire : tempPreviousValue
            int tempPreviousValue = previous[tempPosition];
            //on l'ajout dans la liste path
            sp.add(previous[tempPosition]);
            //on cherche la position du noeud tempPreviousValue
            //et cette position deviens la nouvelle tempPosition
            for (int i = 0; i < nodes.length; i++) {
                if (nodes[i] == tempPreviousValue) {
                    tempPosition = i;
                }
            }
        }
        //on reverse la liste pour avoir root vers destination
        Collections.reverse(sp);
        System.out.println("The shortest path from " + startingNode + " to " + v + " is : " + sp);
    }
    else
    {
        System.out.println("Sorry there is no path from " + startingNode + " to " + v);
    }

}
```

6. Test the previous functions with the graph graph-BFS-SP.txt . Find all the shortest paths and deduce the excentricity of each vertex, the diameter and the radius of the graph.

Dans la méthode main nous avons :

Une classe Digraph est créée pour générer les directed graph, il s'agit d'une petit variante de la classe Graph.
(cf. https://github.com/Jiawdft/TP3/blob/master/tp3Graph/src/fr/isep/lab3/Digraph.java)

```
BFSShortestPaths runBfsShortestPaths = new BFSShortestPaths();
Digraph digraph = new Digraph("graph-BFS-SP.txt");
digraph.affichageAdj();
ArrayList<Integer> excentricityList = new ArrayList<Integer>();
//pour chaque noeuds on realise une bfs, on affiche le shortest path vers tous les autres noeuds
//et on cherche son excentricity
for (int i = 0; i < digraph.getAdj().length; i++) {
    System.out.println("Strating Node : " + digraph.getAdj()[i].nodeId);
    runBfsShortestPaths.bfs(digraph, digraph.getAdj()[i].nodeId);
    for (int j = 0; j < digraph.getAdj().length; j++) {
        runBfsShortestPaths.printSp(j);
    }
    int excentricity = runBfsShortestPaths.excentricity();
    if (excentricity != 0) {
        excentricityList.add(excentricity);
        System.out.println("Excentricity for the node " + digraph.getAdj()[i].nodeId + " is " + excentricity);
    }
    else {
        System.out.println("The node " + digraph.getAdj()[i].nodeId + " is isolated");
    }
    System.out.println();
}

int diameter = Collections.max(excentricityList);
System.out.println("The diameter of the graph is " + diameter);
int radius = Collections.min(excentricityList);
System.out.println("The radius of the graph is " + radius);
```

```
public int excentricity(){
    //search the distance between root and the most distant vertex
    int maxDistance = 0;
    //on parcoure le tableau distance et on retourne la plus grande value
    //sauf Integer.MAX_VALUE considere comme infinie
    for (int i = 0; i < distance.length; i++) {
        if (distance[i] != Integer.MAX_VALUE && distance[i] > maxDistance) {
            maxDistance = distance[i];
        }
    }
    return maxDistance;
}
```

Output console :

```
Order : 8
Size : 9
Adjacency list :
0: 1, 3,
1: 2,
2: 3, 4,
3:
4: 5, 7,
5: 6, 7,
6:
7:
Strating Node : 0
The shortest path from 0 to 0 is : [0]
The shortest path from 0 to 1 is : [0, 1]
The shortest path from 0 to 2 is : [0, 1, 2]
The shortest path from 0 to 3 is : [0, 3]
The shortest path from 0 to 4 is : [0, 1, 2, 4]
The shortest path from 0 to 5 is : [0, 1, 2, 4, 5]
The shortest path from 0 to 6 is : [0, 1, 2, 4, 5, 6]
The shortest path from 0 to 7 is : [0, 1, 2, 4, 7]
Excentricity for the node 0 is 5

Strating Node : 1
Sorry there is no path from 1 to 0
The shortest path from 1 to 1 is : [1]
The shortest path from 1 to 2 is : [1, 2]
The shortest path from 1 to 3 is : [1, 2, 3]
The shortest path from 1 to 4 is : [1, 2, 4]
The shortest path from 1 to 5 is : [1, 2, 4, 5]
The shortest path from 1 to 6 is : [1, 2, 4, 5, 6]
The shortest path from 1 to 7 is : [1, 2, 4, 7]
Excentricity for the node 1 is 4

Strating Node : 2
Sorry there is no path from 2 to 0
Sorry there is no path from 2 to 1
The shortest path from 2 to 2 is : [2]
The shortest path from 2 to 3 is : [2, 3]
The shortest path from 2 to 4 is : [2, 4]
The shortest path from 2 to 5 is : [2, 4, 5]
The shortest path from 2 to 6 is : [2, 4, 5, 6]
The shortest path from 2 to 7 is : [2, 4, 7]
Excentricity for the node 2 is 3

Strating Node : 3
Sorry there is no path from 3 to 0
Sorry there is no path from 3 to 1
Sorry there is no path from 3 to 2
The shortest path from 3 to 3 is : [3]
Sorry there is no path from 3 to 4
Sorry there is no path from 3 to 5
Sorry there is no path from 3 to 6
Sorry there is no path from 3 to 7
The node 3 is isolated
```

```
Strating Node : 4
Sorry there is no path from 4 to 0
Sorry there is no path from 4 to 1
Sorry there is no path from 4 to 2
Sorry there is no path from 4 to 3
The shortest path from 4 to 4 is : [4]
The shortest path from 4 to 5 is : [4, 5]
The shortest path from 4 to 6 is : [4, 5, 6]
The shortest path from 4 to 7 is : [4, 7]
Excentricity for the node 4 is 2

Strating Node : 5
Sorry there is no path from 5 to 0
Sorry there is no path from 5 to 1
Sorry there is no path from 5 to 2
Sorry there is no path from 5 to 3
Sorry there is no path from 5 to 4
The shortest path from 5 to 5 is : [5]
The shortest path from 5 to 6 is : [5, 6]
The shortest path from 5 to 7 is : [5, 7]
Excentricity for the node 5 is 1

Strating Node : 6
Sorry there is no path from 6 to 0
Sorry there is no path from 6 to 1
Sorry there is no path from 6 to 2
Sorry there is no path from 6 to 3
Sorry there is no path from 6 to 4
Sorry there is no path from 6 to 5
The shortest path from 6 to 6 is : [6]
Sorry there is no path from 6 to 7
The node 6 is isolated

Strating Node : 7
Sorry there is no path from 7 to 0
Sorry there is no path from 7 to 1
Sorry there is no path from 7 to 2
Sorry there is no path from 7 to 3
Sorry there is no path from 7 to 4
Sorry there is no path from 7 to 5
Sorry there is no path from 7 to 6
The shortest path from 7 to 7 is : [7]
The node 7 is isolated

The diameter of the graph is 5
The radius of the graph is 1
```

IV. Manipulating Weighted digraphs

Create a class called “DirectedEdge”

```java
public class DirectedEdge {
    private final int v;
    private final int w;
    private final double weight;

    public DirectedEdge(){
        v = 0;
        w = 0;
        weight = 0.0;
    }

    public DirectedEdge(int v, int w, double weight) {
        super();
        this.v = v;
        this.w = w;
        this.weight = weight;
    }

    public int from(){
        return v;
    }
    public int to(){
        return w;
    }
    public double weight(){
        return weight;
    }

}
```

Create a class called WDgraph to represent weighted-digraphs.

```java
public class WDgraph {
    // Order N du graph
    private int N;

    //Size M du graph
    private int M;

    //une class Node avec un identifiant et une liste de DirectedEdge
    public class Node{
        int nodeId;
        ArrayList<DirectedEdge> listDirectedEdge;
    }
    //Initialisation adj list
    private Node [] adj;

    public Node[] getAdj() {
        return adj;
    }
```

```java
public WDgraph(String filePath){
    try {
        List<String> listLignes = Files.readAllLines(Paths.get(filePath),
                StandardCharsets.UTF_8);
        N = addNodesFromTxt(listLignes);
        M = addEdgesFromTxt(listLignes);

    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

```java
public int addNodesFromTxt(List<String> listLignes){
    //un node est parfois present dans plusieurs edges,
    //on utilise HashSet car c'est une collection qui
    //n'accepte pas les doublons
    //Dans nodeshs on va stocker les nodes du graph
    HashSet<String> nodeshs = new HashSet<String>();

    //on parcourt tous les edges du "graph.txt"
    for (int i = 0; i < listLignes.size(); i++) {
        String line = listLignes.get(i);
        //pour chaque edge, on recupere les identifiants des nodes
        String directedEdgeData[] = line.split(" ");
        nodeshs.add(directedEdgeData[0]);
        nodeshs.add(directedEdgeData[1]);
    }
    //Un graph d'ordre N (order)
    int N = nodeshs.size();

    //initialisation du graph
    adj = new Node [N];

    //convertion HashSet toArray String []
    String nodesTab[] = nodeshs.toArray(new String[nodeshs.size()]);

    //Initialisation des nodes
    for (int i = 0; i < nodesTab.length; i++) {
        adj[i] = new Node();
        adj[i].nodeId = Integer.parseInt(nodesTab[i]);
        adj[i].listDirectedEdge = new ArrayList<DirectedEdge>();
    }
    return N;
}
```

```java
public int addEdgesFromTxt(List<String> listLignes){
    //on parcourt tous les edges du "graph.txt"
    for (int i = 0; i < listLignes.size(); i++) {
    String line = listLignes.get(i);
    //pour chaque edge, on recupere les identifiants
    //et le poid des nodes
    String directedEdgeData[] = line.split(" ");
    int from = Integer.parseInt(directedEdgeData[0]);
    int to = Integer.parseInt(directedEdgeData[1]);
    double weight = Double.parseDouble(directedEdgeData[2]);

    //on ajout ce directedEdge dans la list du noeud correspondant
    addEdgeToAdj(from, to, weight);

    }
    //Un graph de taille M (size)
    int M = listLignes.size();
    return M;
}
```

```java
public void addEdgeToAdj(int from, int to, double weight){
    //determination de la position du noeud from dans l'adj
    //list (tab)
    int startNodePosition = getNodePosition(from);
    //creation de la directed edge avec les donnees
    DirectedEdge diEdge = new DirectedEdge(from, to, weight);
    //ajout de la directed edge creee dans la list des
    //directed edge du noeud start
    adj[startNodePosition].listDirectedEdge.add(diEdge);
}

public int getNodePosition(int nodeId) {
    for (int i = 0; i < adj.length; i++) {
        if (adj[i].nodeId == nodeId) {
            return i;
        }
    }
    return -1;
}
```

Puis une méthode qui retourne la position d'un nœud dans l'adjacency list (tableau).

```java
public int getNodePosition(int nodeId) {
    for (int i = 0; i < adj.length; i++) {
        if (adj[i].nodeId == nodeId) {
            return i;
        }
    }
    return -1;
}
```

Et enfin une méthode pour afficher l'adjacency list d'un weighted digraph dans le console.

```
public void affichageAdj(){
    System.out.println("Order : " + N);
    System.out.println("Size : " + M);
    System.out.println("Adjacency list of the weighted digraph :");
    for (int i = 0; i < adj.length; i++) {
        System.out.print(adj[i].nodeId + ": ");
        ArrayList<DirectedEdge> listDiEdge = adj[i].listDirectedEdge;
        for (DirectedEdge diEdge : listDiEdge) {
            System.out.print("[" + diEdge.from());
            System.out.print(", " + diEdge.to());
            System.out.print(", " + diEdge.weight() + "] ");
        }

        System.out.println();
    }
}
```

V. Dijkstra Algorithm for weighted digraphs

Create a class called *DijkstraSP*. This class will implement the Dijkstra algorithm for detecting shortest paths in weighted-digraphs. This class will contain the following functions:

1. 3 arrays: boolean[] marked, int[] previous and int[] distance as for the unweighted graphs.

```
public class DijkstraSP {
    private int [] nodes;
    private boolean [] marked;
    private int [] previous;
    private double [] distance;
```

2. A function called verifyNonNegative(WDGraph G) which takes as input a weidhted-directed graph and verifies than all weights in the graph are non negative.

```java
public boolean verifyNonNegative(WDgraph G){
    //initiation d'un boolean
    //return false si all weights are non negative
    //and true if there are at least one negative weight

    Node [] adj = G.getAdj();
    //Parcourir tous les noeuds du graph
    for (int i = 0; i < adj.length; i++) {
        ArrayList<DirectedEdge> listDiEdge = adj[i].listDirectedEdge;
        //Parcourir tous les weighted edge du noeud
        for (DirectedEdge diEdge : listDiEdge) {
            //des qu'un weight est negative, affect true a isNegative
            //et stop le boucle
            if (diEdge.weight() < 0.0) {
                return true;
            }
        }
    }
    return false;

}
```

3. Create a function called DijkstraSP(WDgraph G, int s)which implements the Dijkstra algorithm for shortest paths studied in lecture 4. The input arguments are a weighted-digraph and a root vertex s.

```java
public ArrayList<Integer> dijkstraSP(WDgraph G, int s){

    //le dijkstraSP fonctionne seulement avec des
    //weight non negative
    //on arrete donc le programme lorsqu'on detecte
    //des weight negative et on returne un code retour -2
    boolean isNegative = verifyNonNegative(G);
    if (isNegative) {
        System.exit(-2);
    }

    Node [] adj = G.getAdj();
    //init des tableaux
    nodes = new int [adj.length];
    marked = new boolean [adj.length];
    previous = new int [adj.length];
    distance = new double [adj.length];
```

```java
//Set pour stocker les noeuds non visited
HashSet<Integer> unseattled = new HashSet<Integer>();
//Array pour stocker les noeuds visited
//ainsi que l'ordre de visite
ArrayList<Integer> seattled = new ArrayList<Integer>();
//pour tous les noeuds
//Set distance of node i as D(i) to infinite
for (int i = 0; i < adj.length; i++) {
    nodes[i] =adj[i].nodeId;
    //on considere Integer.MAX_VALUE = infini
    distance[i] = Integer.MAX_VALUE;
    //on considere -1 comme ensemble vide
    previous[i] = -1;
    //tous met tous les noeuds dans unseattled
    unseattled.add(adj[i].nodeId);
}


    int startPosition = G.getNodePosition(s);
    //distance source
    distance[startPosition] = 0;
    //marque source
    marked[startPosition] = true;

    while(!unseattled.isEmpty()){
        double minDistance = Integer.MAX_VALUE;
        int minNode = -1;
        int minNodePosition = -1;
        for(int thisNode : unseattled){
            int thisNodePosition = G.getNodePosition(thisNode);
            if (distance[thisNodePosition] < minDistance) {
                minDistance = distance[thisNodePosition];
                minNode = thisNode;
                minNodePosition = thisNodePosition;
            }
        }
        //minNode == -1 signifie que le graph est disconnected
        //on arrete donc la boucle while
        if (minNode == -1) {
            break;
        }
        unseattled.remove(minNode);
        seattled.add(minNode);
        //mettre a jour les tableaux : distances previous et marked
        for (DirectedEdge diEdge : adj[minNodePosition].listDirectedEdge) {
            int toPosition = G.getNodePosition(diEdge.to());
            double tempDistance = minDistance +diEdge.weight();
            if (tempDistance < distance[toPosition]) {
                distance[toPosition] = tempDistance;
                marked[toPosition] = true;
                previous[toPosition] = minNode;
            }
        }
    }
    return seattled;
}
```

4. As for the previous section, create the functions hasPathTo(int v), distTo(int v) and printSP(int v).

```
public boolean hasPathTo(int v){
    int vPosition = -1;
    for (int i = 0; i < nodes.length; i++) {
        if (nodes[i] == v) {
            vPosition = i;
        }
    }

    if (vPosition == -1) {
        return false;
    }

    return marked[vPosition];
}
```

```
public double distTo(int v){
    int vPosition = -1;
    for (int i = 0; i < nodes.length; i++) {
        if (nodes[i] == v) {
            vPosition = i;
        }
    }

    if (vPosition == -1) {
        return -2;
    }

    return distance[vPosition];
}
```

```
public void printSP(int v){
    ArrayList<Integer> shortestPath = new ArrayList<Integer>();
    int vPosition = -2;
    for (int i = 0; i < nodes.length; i++) {
        if (nodes[i] == v) {
            vPosition = i;
        }
    }
    if (vPosition == -2) {
        System.exit(-3);
    }
    int tempValue = v;
    int tempValuePosition = vPosition;
    while (tempValue > -1) {
        shortestPath.add(tempValue);
        tempValue = previous[tempValuePosition];
        for (int i = 0; i < nodes.length; i++) {
            if (nodes[i] == tempValue) {
                tempValuePosition = i;
            }
        }
    }
    Collections.reverse(shortestPath);
    System.out.println(shortestPath);
}
```

Test the previous functions with the graph graph-WDG.txt.

Main:

```
//Création graph à partir du fichier graph-WDG.txt
WDgraph wdGraph = new WDgraph("graph-WDG.txt");
//Affiche le graph dans le console
wdGraph.affichageAdj();
DijkstraSP runDijkstrapSP = new DijkstraSP();
//Vérifie le signe des weight
boolean isNegative = runDijkstrapSP.verifyNonNegative(wdGraph);
if (isNegative) {
    System.out.println( "there are at least one negative weight in the graph. ");
}
else {
    System.out.println("all weights in the graph are non negative.");
}
//Execute le dijkstra algorithm
runDijkstrapSP.dijkstraSP(wdGraph, 1);
//Vérifie s'il existe un path du noeud 1 vers le noeud 8
boolean hasPathTo = runDijkstrapSP.hasPathTo(8);
System.out.println("has path to v: " + hasPathTo);
//Affiche la plus petite distance entre le noeud 1 et le noeud 8
double distTo = runDijkstrapSP.distTo(8);
System.out.println("Distance to v : " + distTo);
//Affiche le shortest path de 1 vers 8
System.out.print("The shortest path from s to v : ");
runDijkstrapSP.printSP(8);
```

Console output:

```
Order : 8
Size : 15
Adjacency list of the weighted digraph :
1: [1, 2, 9.0] [1, 6, 14.0] [1, 7, 15.0]
2: [2, 3, 24.0]
3: [3, 5, 2.0] [3, 8, 19.0]
4: [4, 3, 6.0] [4, 8, 6.0]
5: [5, 4, 11.0] [5, 8, 16.0]
6: [6, 3, 18.0] [6, 5, 30.0] [6, 7, 5.0]
7: [7, 5, 20.0] [7, 8, 44.0]
8:
all weights in the graph are non negative.
has path to v: true
Distance to v : 50.0
The shortest path from s to v : [1, 6, 3, 5, 8]
```

Code source :

https://github.com/Jiawdft/TP3

Le dossier TP3 sur git contient les tps 3 et 4.