Algorithmique & Programmation Avancé

Report Tutorial course 3

Graphs, an introduction

DAFIT, Jiawei

Date: 28/03/2017

2.1 Adjacency list representation of an undirected graph

Define a class Graph containing three attributes: the order of the graph (number of nodes), the size of the graph (the number of edges) and an adjacency list representation of the graph.

It is also possible (an even sometimes very useful necessary) to create a class node and a class edge. Then each entry of the adjacency list becomes a list of objects of class node (instead of a list of integer numbers).

```
public class Graph {

//1)--------------------------------------------------------------------------------
    //une class Node avec un identifiant et un element Edge
    private class Node{
        int nodeId;
        Edge firstEdge;
    }
    //une class Edge avec un identifiant du node voisin et l'element Edge suivant
    private class Edge{
        int edgeID;
        Edge nextEdge;
    }
    //un tableau de node : adjacency list representation
    private Node[] adj;

    // Order N du graph
    private int N;

    //Size M du graph
    private int M;
```

1. Initialize an empty graph of order N (input parameter) and 0 edges.

```
//Initialisation : graph vide d'ordre N
public Graph(int N){
    this.N = N ;
    this.M = 0;

    //initialisation tableau de taille N
    adj = new Node[N];
    for (int i = 0; i < adj.length; i++) {
        adj[i] = new Node();
        adj[i].nodeId = i + 1;
        adj[i].firstEdge = null;
    }
}
```

2. Initializes a graph from a specified input stream. You will test this function by reading the graph from the graph.txt file.

3. return the total order and the size of the graph.

```
//2 & 3)-----------------------------------------------------------------------
//Initialisation graph : a partir de graph.txt
public Graph(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();
    }
}
```

```
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 nodesId[] = line.split(" ");
        nodeshs.add(nodesId[0]);
        nodeshs.add(nodesId[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].firstEdge = null;
    }
    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 des nodes
    String nodesId[] = line.split(" ");
    int node1Id = Integer.parseInt(nodesId[0]);
    int node2Id = Integer.parseInt(nodesId[1]);

    //Voir 4)
    addEdgeToAdj(node1Id, node2Id);

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

4. A function called addEdge(int u, int v) that takes as input parameters two integers representing two vertex labels, the endpoints of the new edge. This function will add an edge between two existing nodes to the graph.

```
//4)---------------------------------------------------------------------------------
public void addEdgeToAdj(int u, int v){

    //pour le noeud u
    //Initialisation d'un nouveau element du type Edge avec l'id: u
    Edge edgeU = new Edge();
    edgeU.edgeID = v;
    edgeU.nextEdge = null;
    //dans le cas ou le 1er edge du node est null
    int nodePositionU = getNodePosition(u);
    if (adj[nodePositionU].firstEdge == null) {
        adj[nodePositionU].firstEdge = edgeU;
    }else { //dans le cas ou le 1er edge du node est non null
        //initialisation d'un edge temporaire pour stocker les information du 1er edge et des edges suivantes
        Edge edgeTemp = adj[nodePositionU].firstEdge;

        //Cibler le dernier edge
        while(edgeTemp.nextEdge != null){
            edgeTemp = edgeTemp.nextEdge;
        }
        //la fonction presence permet de verifier si v est deja voisin de u
        boolean presence = presence(u, v);
        if (presence == false) {
            edgeTemp.nextEdge = edgeU;
        }
    }

    //pour le noeud v
    //Initialisation d'un nouveau element du type Edge avec l'id: v
    Edge edgeV = new Edge();
    edgeV.edgeID = u;
    edgeV.nextEdge = null;
    //dans le cas ou le 1er edge du node est null
    int nodePositionV = getNodePosition(v);
    if (adj[nodePositionV].firstEdge == null) {
        adj[nodePositionV].firstEdge = edgeV;
    }else {//dans le cas ou le 1er edge du node est non null
        //initialisation d'un edge temporaire pour stocker les information du 1er edge et des edges suivantes
        Edge edgeTemp = adj[nodePositionV].firstEdge;
        //Cibler le dernier edge
        while(edgeTemp.nextEdge != null){
            edgeTemp = edgeTemp.nextEdge;
        }
        //la fonction presence permet de verifier si u est deja voisin de v
        boolean presence = presence(v, u);
        if (presence == false) {
            edgeTemp.nextEdge = edgeV;
        }
    }
}
```

La fonction getNodePosition permet de trouver la position du nœud dans la liste.

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

La function presence:

```java
public boolean presence(int v, int recherche){
    boolean presence = false;
    int nodePosition = getNodePosition(v);
    Edge edgeTemp = adj[nodePosition].firstEdge;
    while(edgeTemp != null){
        if (edgeTemp.edgeID == recherche) {
            presence = true;
        }
        edgeTemp = edgeTemp.nextEdge;
    }
    return presence;
}
```

5. Create a function Neighbors (int v) that takes a as input a given vertex and prints all the neighbors of that vertex.

```
//5)--------------------------------------------------------------------------
public void neighbors(int v){
    System.out.print("Pour le vertex " + v + " les voisins sont : ");
    int nodePosition = getNodePosition(v);
    Edge edgeTemp = adj[nodePosition].firstEdge;
    while(edgeTemp != null){
        System.out.print(edgeTemp.edgeID + ", ");
        edgeTemp = edgeTemp.nextEdge;
    }
    System.out.println();
}
```

6. Add a function that prints the Adjacency list representation of the graph.

```
//6)--------------------------------------------------------------------------

public void affichageAdj(){
    System.out.println("Order : " + N);
    System.out.println("Size : " + M);
    System.out.println("Adjacency list :");
    for (int i = 0; i < adj.length; i++) {
        System.out.print(adj[i].nodeId + ": ");
        Edge edge = adj[i].firstEdge;
        while (edge != null) {
            System.out.print(edge.edgeID + ", ");
            edge = edge.nextEdge;
        }
        System.out.println();
    }
}
```

Test all these functions with the graph contained the text file.

```
public static void main(String[] args) throws IOException {

    Graph graph = new Graph("graph.txt");
    graph.affichageAdj();



}
```

Console output :

```
Order : 4
Size : 6
Adjacency list :
1: 1, 2, 3,
2: 1, 3,
3: 1, 2, 4,
4: 3,
```

Now, you are going to study some structural properties of the graph. Create a function called Degree (int v) that returns for a given vertex v, its degree.

The degree of a vertex v, denoted d(v), is the number of edges incident to v, and for simple graphs, the degree is equal to the number of neighbors.

```
public int degree(int v){
    System.out.print("Pour le vertex " + v + " le nombre de voisin : ");
    int compteur = 0;
    int nodePosition = getNodePosition(v);
    Edge edgeTemp = adj[nodePosition].firstEdge;
    while(edgeTemp != null){
        compteur++;
        edgeTemp = edgeTemp.nextEdge;
    }
    return compteur;
}
```

Exemple pour le vertex “1” du karate.txt : console output

```
Pour le vertex 1 les voisins sont : 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 18, 20, 22, 32,
Pour le vertex 1 le nombre de voisin : 16
```

Answer the following questions:

1. What is the average, minimal and maximal degree of the graph? What is the edge-density? Do you consider this graph dense or sparse?

The **average degree** of the graph is $d_{av} = \frac{2M}{N}$.

```
public int averageDegree(){
    int averageDegree = 2 * M / N;
    return averageDegree;
}
```

Minimum degree of G, denoted δ:

- $\delta = Min\{d(v)|v \in V\}$

Maximum degree of G, denoted Δ :

- $\Delta = Max\{d(v)|v \in V\}$

```
public int minimumDegree(){
    int minDegree = Integer.MAX_VALUE;
    for (Node node: adj) {
        int degreeTemp = degree(node.nodeId);
        if (degreeTemp < minDegree) {
            minDegree = degreeTemp;
        }
    }
    return minDegree;
}

public int maximumDegree(){
    int maxDegree = 0;
    for (Node node: adj) {
        int degreeTemp = degree(node.nodeId);
        if (degreeTemp > maxDegree) {
            maxDegree = degreeTemp;
        }
    }
    return maxDegree;
}
```

The **Density of the graph (also called edge density)** is
$\gamma = \sum_i d_i = \frac{2M}{N^2}$.

```
public double density(){
    return 2.0 * M / (N * N);
}
```

Console output:

```
Order : 4
Size : 6
Adjacency list :
1: 1, 2, 3,
2: 1, 3,
3: 1, 2, 4,
4: 3,
Average degree : 3
Minimum degree : 1
Maximum degree : 3
Edge density : 0.75
```

La densité du graph est 0.75, on peut donc le considérer comme dense.

2. Are there any isolated nodes? If yes, which ones?

   Tous les nœuds possèdes des voisins, donc il n’y pas de nœud isolé.

3. Are there any loops?

   On peut remarquer que le nœud « 1 » possède comme voisin lui-même, il y a donc une boucle sur le nœud « 1 ».

4. Verify your answers by using the Neighbors (int v) function.

   Method main:

```
public static void main(String[] args) throws IOException {
    Graph graph = new Graph("graph.txt");
    graph.affichageAdj();

    System.out.println("Average degree : " + graph.averageDegree());
    System.out.println("Minimum degree : " + graph.minimumDegree());
    System.out.println("Maximum degree : " + graph.maximumDegree());
    System.out.println("Edge density : " + graph.density());

    graph.neighbors(1);
    graph.neighbors(2);
    graph.neighbors(3);
    graph.neighbors(4);

}
```

Console output: en complément du console output de la question 1)

```
Pour le vertex 1 les voisins sont : 1, 2, 3,
Pour le vertex 2 les voisins sont : 1, 3,
Pour le vertex 3 les voisins sont : 1, 2, 4,
Pour le vertex 4 les voisins sont : 3,
```

Finally, write a function that allows to read data graph from the keyboard input. The program will ask the user the total number of vertices, the total number of edges and user will have to type each edge as in the following example:

```
Enter the number of vertices:
5
Enter the number of edges:
4
Enter the edges in the graph : <to> <from>
1 2
```

```
public Graph(){
    Scanner scan = new Scanner(System.in);
    System.out.println("Enter the number of vertices:");
    N = scan.nextInt();
    adj = new Node[N];
    for (int i = 0; i < adj.length; i++) {
        adj[i] = new Node();
        adj[i].nodeId = i + 1;
        adj[i].firstEdge = null;
    }
    System.out.println("Enter the number of edges:");
    M = scan.nextInt();
    scan.nextLine();

    for (int i = 0; i < M; i++) {
        System.out.println("Enter the edges in the graph : <to> <from>");
        String ligne = scan.nextLine();
        String [] nodes = ligne.split(" ");
        int node1 = Integer.parseInt(nodes[0]);
        int node2 = Integer.parseInt(nodes[1]);
        addEdgeToAdj(node1, node2);

    }
    scan.close();
}


public static void main(String[] args) throws IOException {
    Graph graph = new Graph();
    graph.affichageAdj();
}
```

Console:

```
Enter the number of vertices:
3
Enter the number of edges:
2
Enter the edges in the graph : <to> <from>
1 2
Enter the edges in the graph : <to> <from>
1 3
Order : 3
Size : 2
Adjacency list :
1: 2, 3,
2: 1,
3: 1,
```

2.2 Adjacency matrix representation of an undirected graph

Define a class GraphAdjMatrix containing three attributes: the order of the graph (number of nodes), the size of the graph (the number of edges) and an adjacency matrix.

```
public class GraphAdjMatrix {
    private final int N;
    private int M;
    private boolean [][] adj;
```

1. Initialize an empty graph of order N (input parameter) and 0 edges.

```
//1)--------------------------------------------------------------------------------
public GraphAdjMatrix(int N){
    this.N = N;
    this.M = 0;

    adj = new boolean[N][N];
    for (int i = 0; i < adj.length; i++) {
        for (int j = 0; j < adj[0].length; j++) {
            adj[i][j] = false;
        }
    }
}
```

2. Initializes a graph from a specified input stream. All the entries of the matrix adj must be initialized. You will test this function by reading the graph from the graph.txt file.

```
//2)--------------------------------------------------------------------------------
public GraphAdjMatrix(String filePath) throws IOException{

        List<String> listLignes = Files.readAllLines(Paths.get(filePath),
                StandardCharsets.UTF_8);
        N = addNodesFromTxt(listLignes);
        M = addEdgesFromTxt(listLignes);

}
```

```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 nodesId[] = line.split(" ");
        nodeshs.add(nodesId[0]);
        nodeshs.add(nodesId[1]);
    }
    //Un graph d'ordre N (order)
    int N = nodeshs.size();

    //initialisation d'une matrice carre de taille n
    adj = new boolean[N][N];
    for (int i = 0; i < adj.length; i++) {
        for (int j = 0; j < adj[0].length; j++) {
            adj[i][j] = false;
        }
    }
    return N;
}

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 des nodes
            String nodesId[] = line.split(" ");
            int node1Id = Integer.parseInt(nodesId[0]);
            int node2Id = Integer.parseInt(nodesId[1]);
            adj[node1Id - 1][node2Id- 1] = true;
            adj[node2Id- 1][node1Id- 1] = true;
            }

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

 //Affichage------------------------------------------------------------
 public void affichage(){
     System.out.println("Order : " + N);
     System.out.println("Size : " + M);
     System.out.println("Adjacency Matrix :");
     for (int i = 0; i < adj.length; i++) {
         for (int j = 0; j < adj[0].length; j++) {
             if (adj[i][j] == true) {
                 System.out.print(1 + " ");
             }else {
                 System.out.print(0 + " ");
             }
         }
         System.out.println();
     }
 }
```

Main method:

```java
public static void main(String[] args) throws IOException {
    GraphAdjMatrix graph = new GraphAdjMatrix("graph.txt");
    graph.affichage();
}
```

Console out:

```
Order : 4
Size : 6
Adjacency Matrix :
1 1 1 0
1 0 1 0
1 1 0 1
0 0 1 0
```

Why is it preferably to use an adjacency list representation in practical contexts?

Il est préférable d'utiliser un adjacency list représentation en pratique car, un adjacency matrix utilise une mémoire de $O(n^2)$, il est donc plus rapide pour vérifier la présence ou non d'un arc spécifique, mais plus lent pour itérer tous les arcs. Alors qu'un adjacency list utilise une mémoire qui est proportionnelle au nombre d'arcs, ce qui permettra d'économiser beaucoup de mémoire si nous avons une adjacency list clairsemé (sparse). Il est donc plus rapide pour itérer tous les arcs mais un peu plus lent pour chercher un arc spécifique.

3. Practical application

Why is it possible to say that the nodes 1 and 34 occupy a central position?

The main method:

```java
public class Main {

    public static void main(String[] args) throws IOException {
        Graph graph = new Graph("karate.txt");
        graph.affichageAdj();

    }
}
```

Console output :

```
Order : 34
Size : 78
Adjacency list :
22: 1, 2,
23: 33, 34,
24: 26, 28, 30, 33, 34,
25: 26, 28, 32,
26: 24, 25, 32,
27: 30, 34,
28: 3, 24, 25, 34,
29: 3, 32, 34,
30: 24, 27, 33, 34,
31: 2, 9, 33, 34,
32: 1, 25, 26, 29, 33, 34,
10: 3, 34,
11: 1, 5, 6,
33: 3, 9, 15, 16, 19, 21, 23, 24, 30, 31, 32, 34,
12: 1,
34: 9, 10, 14, 15, 16, 19, 20, 21, 23, 24, 27, 28, 29, 30, 31, 32, 33,
13: 1, 4,
14: 1, 2, 3, 4, 34,
15: 33, 34,
16: 33, 34,
17: 6, 7,
18: 1, 2,
19: 33, 34,
1: 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 18, 20, 22, 32,
2: 1, 3, 4, 8, 14, 18, 20, 22, 31,
3: 1, 2, 4, 8, 9, 10, 14, 28, 29, 33,
4: 1, 2, 3, 8, 13, 14,
5: 1, 7, 11,
6: 1, 7, 11, 17,
7: 1, 5, 6, 17,
8: 1, 2, 3, 4,
9: 1, 3, 31, 33, 34,
20: 1, 2, 34,
21: 33, 34,
```

Nous pouvons remarquer que les nœuds 1 et 34 ont les degrés les plus importants, or nous savons que les nœuds avec le plus haut degré occupent la position centrale, donc nous pouvons conclure que les nœuds 1 et 34 occupe la position centrale.

Code source :

https://github.com/Jiawdft/TP3