

Lemeunier Gaëtan

Weier Loris

S2D

SAE 2.02 – Exploration Algorithmique

Recherche du plus court chemin dans le graphe

Préambule : Ce projet a été réalisé sur Visual Studio Code avec l'accord du professeur référent. Les tests que nous avons faits sont donc avec libtest.

SAE 2.02 – Exploration Algorithmique

Recherche du plus court chemin dans le graphe

Partie 1 : Présentation de la SAE

Partie 2 : Représentation d'un graphe

Question 1 :

Question 2 :

Question 3 :

Question 4 :

Question 5 :

Question 6 :

Question 7 :

Question 8 :

Question 9 :

Question 10 :

Question 11 :

Question 12 :

Question 13 :

Partie 3 : Calcul du plus court chemin par point fixe.

Question 14 :

Question 15 :

Question 16 :

Question 17 :

Question 18 :

Partie 4 : Calcul du meilleur chemin par Dijkstra.

Question 19 :

Question 20 :

Question 21 :

Partie 5 : Validation et expérimentation

Question 22 :

Question 23 :

Question 24 :

Question 25 :

Question 27 :

Question 28 :

Question 29 :

Question 30 :

Question 31 :

Conclusion de la SAE

Partie 1 : Présentation de la SAE

Dans cette SAE nous souhaitons à partir de graphes donnés d'implémenter des algorithmes vus en mathématiques tel que l'algorithme du point fixe ou celui de Dijkstra. Cela afin de pouvoir déterminer quelle est la solution la plus efficace pour trouver le chemin le plus rapide d'un graphe.

Partie 2 : Représentation d'un graphe

Question 1 :

Pour la construction de la classe Nœud nous avons pris deux attributs :

- Le nom du nœud qui est donc un String
- Une ArrayList contenant des Arcs afin de relier les nœuds

Le constructeur prend en paramètre le nom du nœud que l'on doit créer et l'instancie avec son nom et la liste d'arcs vide.

```
public Noeud(String n) {  
    this.nom = n;  
    this.adj = new ArrayList<Arc>();  
}
```

Question 2 :

Etant donné que l'on identifie un nœud par son nom, il suffit de comparer les deux noms des nœuds et de renvoyer true si les noms sont les mêmes et false sinon.

```
public boolean equals(Noeud n) {  
    return this.nom.equals(n.nom);  
}
```

Question 3 :

On verra plus tard qu'un arc est constitué d'une destination et d'un coût. Pour la fonction ajouter arc on appelle donc le constructeur d'Arc avec une destination et un point d'arrivé en paramètre.

```
public void ajouterArc(String destination, double cout) {  
    this.adj.add(new Arc(destination, cout));  
}
```

Question 4 :

Pour le constructeur d'Arc la seule ambiguïté est de vérifier que le coût est strictement positif et sinon de le mettre à zéro. En plus de cela nous avons ajouter des Getter pour les tests que l'on fera prochainement.

```
public class Arc {  
  
    /**  
     * Attribut dest qui repr  sente le nom du n  ud de destination de l'arc  
     */  
    private String dest;  
  
    /**  
     * Attribut cout qui repr  sente le co  t de l'arc  
     */  
    private double cout;  
  
    /**  
     * Constructeur de la classe Arc  
     * @param d nom du n  ud de destination  
     * @param c co  t de l'arc (doit  tre positif, sinon il est mis   0)  
     */  
  
    public Arc(String d, double c) {  
        this.dest = d;  
  
        if(c < 0)  
            this.cout = 0;  
        else  
            this.cout = c;  
    }  
}
```

Question 5 :

Pour ´crire l'interface Graphe on a ´crit le squelette de plusieurs m  thodes. Une qui retourne tous les n  uds du graphe et une qui retourne les arcs suivant le n  ud donn   en param  tre.

```
public interface Graphe {  
  
    /**  
     * M  thode qui retourne tous les n  uds du graphe  
     * @return ArrayList<String> nom des n  uds  
     */  
    public ArrayList<String> listeNoeuds();  
  
    /**  
     * M  thode qui retourne la liste des arcs partant du n  ud donn    
     * @param n nom du n  ud  
     * @return ArrayList<Arc> liste des arcs du n  ud  
     */  
    public ArrayList<Arc> suivants(String n);  
}
```

Question 6 :

Dans cette question la seule chose qui nous a posé un petit problème est la méthode ajouterArc. Mais ce problème a été vite réglé car il suffisait de vérifier avec une condition si le nœud de départ et celui de destination existait.

```
import java.io.*;
import java.util.ArrayList;

/**
 * Classe GrapheListe qui représente les données associées à un graphe
 */
public class GrapheListe implements Graphe {

    /**
     * Attribut contenant les noms des objets noeuds stockés
     */
    private ArrayList<String> ensNom;

    /**
     * Attribut contenant les objets noeuds stockés
     */
    private ArrayList<Noeud> ensNoeud;

    /**
     * Constructeur de la classe GrapheListe
     * @param noeuds liste des noeuds du graphe
     */
    public GrapheListe(ArrayList<Noeud> noeuds) {
        this.ensNoeud = noeuds;
        this.ensNom = new ArrayList<String>();
        for(Noeud noeud : noeuds){
            this.ensNom.add(noeud.getNom());
        }
    }

    /**
     * Méthode qui permet d'ajouter des noeuds et des arcs à un objet GrapheListe
     * @param depart nom du nœud de départ
     * @param destination nom du nœud de destination
     * @param cout coût de l'arc
     */
    public void ajouterArc(String depart, String destination, double cout){
        // On vérifie que les noeuds existent
        if(this.ensNom.contains(depart) && this.ensNom.contains(destination)){
            // On parcourt la liste des noeuds du graphe
            for(Noeud noeud : this.ensNoeud){
                if(noeud.equals(new Noeud(depart))){
                    noeud.ajouterArc(destination, cout);
                    break; // On sort de la boucle car on a trouvé le noeud
                }
            }
        }
    }
}
```

Question 7 :

Pour avoir le graphe présenté dans la figure 1 il faut créer les nœuds correspondant. Créer GrapheListe avec la liste de nœuds créée juste avant. Puis finalement appeler la méthode ajouterArc sur le GrapheListe afin d'avoir les bonnes liaisons entre les nœuds.

```

public class MainGraphe {

    Run | Debug
    public static void main(String[] args) {

        // ===== Graphe Manuel =====

        // Cr ation des Noeuds
        ArrayList<Noeud> noeuds = new ArrayList<Noeud>();
        collections.addAll(noeuds,
            new Noeud("A"),
            new Noeud("B"),
            new Noeud("C"),
            new Noeud("D"),
            new Noeud("E")
        );

        // Cr ation du graphe
        GrapheListe graphe = new GrapheListe(noeuds);

        // Ajout des arcs
        graphe.ajouterArc("A", "B", 12);
        graphe.ajouterArc("A", "D", 87);
        graphe.ajouterArc("B", "E", 11);
        graphe.ajouterArc("C", "A", 19);❶
        graphe.ajouterArc("D", "B", 23);
        graphe.ajouterArc("D", "C", 10);
        graphe.ajouterArc("E", "D", 43);

        // Affichage des noeuds
        System.out.println(graphe.toString());
        System.out.println(graphe.toGraphviz());
        System.out.println(graphe.listeNoeuds());
    }
}

```

Question 8 :

La m thode `toString` de la classe `GrapheListe` prend le nom de chaque n eud dans la liste de n eud du graphe avec une boucle « `for` » et ´crit derri re une fl che la destination de l'arc et son co t.

```

/**
 * Méthode d'affichage du graphe
 */
public String toString(){
    String s = "";
    // On parcourt la liste des noeuds du graphe
    for(Noeud noeud : this.ensNoeud){
        s += noeud.getNom() + " -> ";
        // On parcourt la liste des arcs adjacents du noeud
        for(Arc arc : noeud.getAdj()){
            s += arc.getDest() + "(" + (int)arc.getCout() + ")";
        }
        s += "\n";
    }
    return s;
}

```

Question 9 :

La méthode toGraphviz ressemble beaucoup au toString de la même classe. En effet les changements sont :

- Encadrer les noeuds et les arcs par « digraph G{...} »
- Ecrire « label » avant le coût de l'arc afin que l'application graphique reconnaisse les différents éléments.

```

/**
 * Méthode qui retourne le graphe au format Graphviz
 */
public String toGraphviz(){
    String s = "digraph G {\n";
    // On parcourt la liste des noeuds du graphe
    for(Noeud noeud : this.ensNoeud){
        // On parcourt la liste des arcs adjacents du noeud
        for(Arc arc : noeud.getAdj()){
            s += noeud.getNom() + " -> " + arc.getDest() + "[label = " + (int)arc.getCout() + "]\n";
        }
    }
    s += "}";
    return s;
}

```

Question 10 :

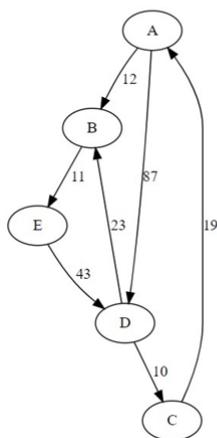
Pour générer des graphes nous utilisons la version web de GraphViz en copiant et collant le résultat de notre code dans la version web. Le graphe se forme parfaitement avec le résultat du terminal.

```

1 digraph G {
2 A -> B [label = 12]
3 A -> D [label = 87]
4 B -> E [label = 11]
5 C -> A [label = 19]
6 D -> B [label = 23]
7 D -> C [label = 10]
8 E -> D [label = 43]
9 }

```

Engine: dot Format: svg Show raw output Download Share



Question 11 :

Pour vérifier que le graphe se construit correctement nous avons écrit des tests :

- On teste la méthode listeNoeuds afin de savoir si les nœuds sont dans le bonne ordre dans le graphe et que ce que renvoi cette méthode n'a pas d'éléments parasites.

```

public void test_02_ajouterArc() {
    // Création des Noeuds
    ArrayList<Noeud> noeuds = new ArrayList<Noeud>();
    noeuds.add(new Noeud("A"));
    noeuds.add(new Noeud("B"));
    noeuds.add(new Noeud("C"));

    // Création du graphe
    GrapheListe graphe = new GrapheListe(noeuds);

```

```

public void test_01_listeNoeuds() {
    // Création des Noeuds
    ArrayList<Noeud> noeuds = new ArrayList<Noeud>();
    noeuds.add(new Noeud("A"));
    noeuds.add(new Noeud("B"));

    // Création du graphe
    GrapheListe graphe = new GrapheListe(noeuds);

    // Appel de la méthode à tester
    ArrayList<String> listeNoeuds = graphe.listeNoeuds();

    // Vérification du résultat
    assertEquals("La taille de la liste doit être 2", graphe.listeNoeuds().size(), listeNoeuds.size());
    assertEquals("Le premier noeud doit être A", "A", listeNoeuds.get(0));
    assertEquals("Le deuxième noeud doit être B", "B", listeNoeuds.get(1));
}

```

- On teste la méthode qui permet d'ajouter des arcs car c'est la base de notre graphe si cette fonction est problématique on risque d'avoir des incohérences dans nos résultats.

On veut donc voir si nos nœuds sont bien placés, que les arcs vont vers les bons nœuds et que les coûts correspondent bien.

Question 12 :

Pour cette question il faut ajouter un nouveau constructeur dans la classe GrapheListe qui prend en paramètre le nom du fichier et créer un graphe à partir de ce qui correspond dans ce fichier.

On utilise le try-catch dans cette fonction car il peut y avoir des problèmes en fonction du nom du fichier mais aussi durant la lecture du fichier. On va lire les lignes du fichier donné après l'avoir ouvert avec un BufferedReader et ensuite grâce à un tableau de String on peut récupérer les informations dans le fichier et créer un graphe.

```

public GrapheListe(String fichier) {
    this.ensNoeud = new ArrayList<Noeud>();
    this.ensNom = new ArrayList<String>();

    try {
        // On lit le fichier
        BufferedReader br = new BufferedReader(new FileReader("../graphes/" + fichier));
        String ligne;

        while ((ligne = br.readLine()) != null) {

            // On découpe la ligne
            String[] tab = ligne.split("\t");
            // On ajoute l'arc
            Noeud noeud = new Noeud(tab[0]);

            // si le noeud n'existe pas
            if(!this.ensNom.contains(tab[0])){
                this.ensNoeud.add(noeud);
                this.ensNom.add(tab[0]);
            }
            if(!this.ensNom.contains(tab[1])){
                this.ensNoeud.add(new Noeud(tab[1]));
                this.ensNom.add(tab[1]);
            }

            this.ajouterArc(tab[0], tab[1], Double.parseDouble(tab[2]));
        }

        // On ferme le fichier
        br.close();
    } catch (FileNotFoundException e) {
        System.out.println("Le fichier n'existe pas");
    } catch (IOException e) {
        System.out.println("Erreur lors de la lecture du fichier");
    }
}

```

Question 13 :

```
public void matriceToArc(String fichier) {
    try {
        BufferedWriter bw = new BufferedWriter(new FileWriter("../graphes/listeArcs.txt"));
        BufferedReader br = new BufferedReader(new FileReader("../graphes/" + fichier));
        String ligne;
        String[] nomNoeuds = null;
        int i = 0;

        while ((ligne = br.readLine()) != null) {
            String[] tab = ligne.split("\t");
            if(i == 0){
                nomNoeuds = tab;
                i++;
            }
            else {
                for(int j = 0; j < tab.length; j++){
                    // On vérifie que la valeur n'est pas 0 ou une lettre
                    if(!tab[j].equals("0.") && !tab[j].matches("^[a-zA-Z]*$")){
                        bw.write(tab[0] + "\t" + nomNoeuds[j] + "\t" + tab[j] + "\n");
                    }
                }
            }
        }

        bw.close();
        br.close();
    } catch (FileNotFoundException e) {
        System.out.println("Le fichier n'existe pas");
    } catch (IOException e) {
        System.out.println("Erreur lors de la lecture du fichier");
    }
}
```

Partie 3 : Calcul du plus court chemin par point fixe.

Question 14 :

```

21         si arc.getCout() + valeurCourante < nouvelleValeur
22             alors
23                 L(arc.getDest()) ← arc.getCout() +
valeurCourante
24                     parent(arc.getDest()) ← noeud
25                     changement ← vrai
26             fsi
27             fpour
28             fpour
29             ftantque
30                 retourner v
31             fin
32
33 ===== Lexique : =====
34
35 - g : Graphe, graphe dont on veut trouver le plus court chemin
36 - depart : Chaine, nom du noeud de départ
37 - i : Entier, indice de parcours
38 - changement : Booleen, indique si il y a eu un changement dans la boucle
39 - noeud : Chaine, nom du noeud courant
40 - valeurCourante : Entier, valeur courante du noeud
41 - arc : Arc, arc courant
42 - nouvelleValeur : Entier, nouvelle valeur du noeud

```

Question 15 :

Pour implémenter la fonction résoudre on a utilisé l'algorithme fait dans la question précédente. En plus de cela la fonction valeur fournit nous aide pour les nœuds parents.

```

public Valeur resoudre(Graphe g, String depart) {
    Valeur v = new Valeur();
    for (String s : g.listeNoeuds()) {
        if (s.equals(depart))
            v.setValeur(s, 0);
        else
            // Double.MAX_DOUBLE ne marche pas
            v.setValeur(s, Double.MAX_VALUE);
        v.setParent(s, null);
    }

    boolean changement = true;

    while (changement) {
        changement = false;
        for (String noeud : g.listeNoeuds()) {
            double valeurCourante = v.getValeur(noeud);
            for (Arc arc : g.suivants(noeud)) {
                double nouvelleValeur = v.getValeur(arc.getDest());
                if (arc.getCout() + valeurCourante < nouvelleValeur) {
                    v.setValeur(arc.getDest(), arc.getCout() + valeurCourante);
                    v.setParent(arc.getDest(), noeud);
                    changement = true;
                    // Affichage de l'évolution de l'algorithme
                    // System.out.println(v.toString());
                }
            }
        }
    }
    return v;
}

```

Question 16 :

```

import java.util.ArrayList;
import java.util.Collections;

/**
 * classe MainBellmanFord : classe principale qui permet de tester l'algorithme de Bellman-Ford
 */
public class MainBellmanFord {

    public static void main(String[] args) {

        BellmanFord bf = new BellmanFord();
        Graphe g = new GrapheListe("graphe.txt");

        Valeur v = bf.resoudre(g, "A");

        System.out.println(v.toString());

        ===== Chemins =====
        for (String s : g.listeNoeuds()) {
            System.out.println("Chemin de A vers " + s + " : " + v.calculerChemin(s));
        }
    }
}

```

Les résultats de ce main sont :

```
A -> V:0.0 p:null
B -> V:12.0 p:A
C -> V:76.0 p:D
D -> V:66.0 p:E
E -> V:23.0 p:B
```

```
Chemin de A vers D : [A, B, E, D]
Chemin de A vers C : [A, B, E, D, C]
Chemin de A vers A : [A]
Chemin de A vers B : [A, B]
Chemin de A vers E : [A, B, E]
```

Dans le module « Graphe » on a calculé les chemins et les valeurs de résolution du graphe et elles correspondent à ce que l'on retrouve grâce à l'algorithme de Bellman Ford.

Question 17 :

Pour les tests qui vérifie que l'algorithme du point fixe fonctionne bien on a préféré essayer avec différents points de départs afin d'être sûr de ne pas avoir de problème.

```
public void test_01_resoudre_A() {

    // Création du graphe
    BellmanFord bf = new BellmanFord();
    Graphe g = new GrapheListe("graphe.txt");

    // Appel de la méthode à tester
    Valeur v = bf.resoudre(g, "A");

    // vérification du résultat
    assertEquals("La valeur de A doit être 0.0", 0.0, v.getValeur("A"));
    assertEquals("La valeur de B doit être 12.0", 12.0, v.getValeur("B"));
    assertEquals("La valeur de C doit être 76.0", 76.0, v.getValeur("C"));
    assertEquals("La valeur de D doit être 66.0", 66.0, v.getValeur("D"));
    assertEquals("La valeur de E doit être 23.0", 23.0, v.getValeur("E"));
}
```

```

public void test_02_resoudre_B() {

    // Création du graphe
    BellmanFord bf = new BellmanFord();
    Graphe g = new GrapheListe("graphe.txt");

    // Appel de la méthode à tester
    Valeur v = bf.resoudre(g, "B");

    // Vérification du résultat
    assertEquals("La valeur de A doit être 83.0", 83.0, v.getValeur("A"));
    assertEquals("La valeur de B doit être 0.0", 0.0, v.getValeur("B"));
    assertEquals("La valeur de C doit être 64.0", 64.0, v.getValeur("C"));
    assertEquals("La valeur de D doit être 54.0", 54.0, v.getValeur("D"));
    assertEquals("La valeur de E doit être 11.0", 11.0, v.getValeur("E"));
}

```

```

public void test_03_resoudre_C() {

    // Création du graphe
    BellmanFord bf = new BellmanFord();
    Graphe g = new GrapheListe("graphe.txt");

    // Appel de la méthode à tester
    Valeur v = bf.resoudre(g, "C");

    // Vérification du résultat
    assertEquals("La valeur de A doit être 19.0", 19.0, v.getValeur("A"));
    assertEquals("La valeur de B doit être 31.0", 31.0, v.getValeur("B"));
    assertEquals("La valeur de C doit être 0.0", 0.0, v.getValeur("C"));
    assertEquals("La valeur de D doit être 85.0", 85.0, v.getValeur("D"));
    assertEquals("La valeur de E doit être 42.0", 42.0, v.getValeur("E"));
}

```

Question 18 :

```

public List<String> calculerChemin(String destination) {
    List<String> chemin = new ArrayList<>();
    String noeud = destination;
    // on remonte le chemin en partant de la destination
    while (noeud != null) {
        chemin.add(0, noeud);
        noeud = this.getParent(noeud);
    }
    return chemin;
}

```

Dans cette question on a modifié la classe valeur en ajoutant cette fonction. On récupère tous les nœuds qui amène du point de départ jusqu'à la destination. On ajoute les nœuds dans une List de String afin de la retourner à la fin du programme.

Partie 4 : Calcul du meilleur chemin par Dijkstra.

Dans le début de cette partie on nous explique le principe de l'algorithme de Dijkstra en nous montrant ses points communs et ses différences avec l'algorithme du point fixe.

On nous explique de A à Z comment celui fonctionne afin d'ensuite l'implémenter dans notre programme.

On nous donne ensuite comment calculer le meilleur chemin avec Dijkstra en algorithmique :

```
1 Entrées :
2     - G un graphe orienté avec une pondération (poids) positive des arcs
3     - A un sommet (départ) de G
4
5     Début
6     Q <- {} // * utilisation d'une liste de noeuds à traiter
7     Pour chaque sommet v de G faire
8         v.distance <- Infini
9         v.parent <- Indéfini
10    Q <- Q ∪ {v} // * ajouter le sommet v à la liste Q
11    Fin Pour
12
13    A.distance <- 0
14    Tant que Q est un ensemble non vide faire
15        u <- un sommet de Q telle que u.distance est minimale
16        Q <- Q \ {u} // * enlever le sommet u de la liste Q
17        Pour chaque sommet v de Q tel que l'arc (u,v) existe faire
18            D <- u.distance + poids(u,v)
19            Si D < v.distance
20                Alors v.distance <- D
21                v.parent <- u
22            Fin Si
23        Fin Pour
24    Fin Tant que
25    Fin
```

Question 19 :

Voici l'algorithme traduit en java :

```

public Valeur resoudre(Graphe g, String depart) {
    ArrayList<String> noeuds = new ArrayList<String>();
    Valeur v = new Valeur();
    for (String s : g.listeNoeuds()) {
        v.setValeur(s, Integer.MAX_VALUE);
        v.setParent(s, null);
        noeuds.add(s);
    }

    v.setValeur(depart, 0);
    while(!noeuds.isEmpty()) {
        String u = noeuds.get(0);
        for (String s : noeuds) {
            if (v.getValeur(s) < v.getValeur(u))
                u = s;
        }
        noeuds.remove(u);
        for (Arc a : g.suivants(u)) {
            double d = v.getValeur(u) + a.getCout();
            if (d < v.getValeur(a.getDest())) {
                v.setValeur(a.getDest(), d);
                v.setParent(a.getDest(), u);
                // Affichage de l'évolution de l'algorithme
                // System.out.println(v.toString());
            }
        }
    }
    return v;
}

```

Les paramètres *g* et *depart* sont respectivement un graphe dont on veut trouver le chemin le plus court et le nom du nœud de départ de ce graphe.

Cette fonction retourne un objet de type *Valeur* contenant les valeurs et les parents des nœuds.

Comme conseillé dans le sujet nous avons créé une interface *Algorithme* car on a une méthode résoudre dans les deux algorithmes.

```

public interface Algorithme {

    /**
     * Méthode qui effectue l'algorithme sur un graphe
     * @param g graphe dont on veut trouver le plus court chemin
     * @param depart nom du noeud de départ
     * @return Valeur, objet contenant les valeurs et les parents des noeuds
     */
    public Valeur resoudre(Graphe g, String depart);

}

```

Question 20 :

De la même manière que pour l'algorithme du point fixe nous avons créé des tests afin de vérifier le bon fonctionnement de Dijkstra.

```
public void test_01_resoudre_A() {  
  
    // Création du graphe  
    BellmanFord bf = new BellmanFord();  
    Graphe g = new GrapheListe("graphe.txt");  
  
    // Appel de la méthode à tester  
    Valeur v = bf.resoudre(g, "A");  
  
    // Vérification du résultat  
    assertEquals("La valeur de A doit être 0.0", 0.0, v.getValeur("A"));  
    assertEquals("La valeur de B doit être 12.0", 12.0, v.getValeur("B"));  
    assertEquals("La valeur de C doit être 76.0", 76.0, v.getValeur("C"));  
    assertEquals("La valeur de D doit être 66.0", 66.0, v.getValeur("D"));  
    assertEquals("La valeur de E doit être 23.0", 23.0, v.getValeur("E"));  
}
```

```
,  
public void test_02_resoudre_B() {  
  
    // Création du graphe  
    BellmanFord bf = new BellmanFord();  
    Graphe g = new GrapheListe("graphe.txt");  
  
    // Appel de la méthode à tester  
    Valeur v = bf.resoudre(g, "B");  
  
    // Vérification du résultat  
    assertEquals("La valeur de A doit être 83.0", 83.0, v.getValeur("A"));  
    assertEquals("La valeur de B doit être 0.0", 0.0, v.getValeur("B"));  
    assertEquals("La valeur de C doit être 64.0", 64.0, v.getValeur("C"));  
    assertEquals("La valeur de D doit être 54.0", 54.0, v.getValeur("D"));  
    assertEquals("La valeur de E doit être 11.0", 11.0, v.getValeur("E"));  
}
```

```
public void test_03_resoudre_C() {  
  
    // Création du graphe  
    BellmanFord bf = new BellmanFord();  
    Graphe g = new GrapheListe("graphe.txt");  
  
    // Appel de la méthode à tester  
    Valeur v = bf.resoudre(g, "C");  
  
    // Vérification du résultat  
    assertEquals("La valeur de A doit être 19.0", 19.0, v.getValeur("A"));  
    assertEquals("La valeur de B doit être 31.0", 31.0, v.getValeur("B"));  
    assertEquals("La valeur de C doit être 0.0", 0.0, v.getValeur("C"));  
    assertEquals("La valeur de D doit être 85.0", 85.0, v.getValeur("D"));  
    assertEquals("La valeur de E doit être 42.0", 42.0, v.getValeur("E"));  
}
```

Comme pour l'algorithme la différence entre les tests est le fait de changer de points de départ dans la méthode résoudre.

Question 21 :

```
public class MainDijkstra {  
  
    public static void main(String[] args) {  
  
        Dijkstra dijkstra = new Dijkstra();  
        Graphe g = new GrapheListe("graphe.txt");  
  
        valeur v = dijkstra.resoudre(g, "A");  
  
        System.out.println(v.toString());  
  
        // ===== Chemins =====  
        for (String s : g.listeNoeuds()) {  
            System.out.println("Chemin de A vers " + s + " : " + v.calculerChemin(s));  
        }  
    }  
}
```

Partie 5 : Validation et expérimentation

Question 22 :

En utilisant l'algorithme de Dijkstra on observe que l'on a beaucoup plus d'itérations qu'en utilisant l'algorithme du point fixe. La principale différence de comportement entre les deux algorithmes réside dans leur approche de mise à jour des valeurs de distance. Dijkstra explore les nœuds du graphe de manière séquentielle en fonction de la distance actuelle, garantissant ainsi la convergence vers la solution optimale. En revanche, l'algorithme du point fixe met à jour toutes les valeurs de distance simultanément à chaque itération, ce qui peut conduire à une convergence plus lente ou à des résultats sous-optimaux.

Question 23 :

On peut en déduire grâce à nos résultats que l'algorithme du point fixe sera peut-être moins précis mais plus rapide que l'algorithme de Dijkstra qui lui va chercher la solution optimale mais en prenant sûrement plus de temps.

Question 24 :

La valeur présente dans les tableaux est calculé en nanosecondes.

Nom du Graphe	Algorithme du point fixe	Algorithme de Dijkstra
Graphe.txt	7872	10851
Graphe1.txt	2094	3763
Graphe11.txt	1614	2755
Graphe51.txt	1359	1419
Graphe101.txt	1471	1540
Graphe501.txt	1822	1930

Nom du Graphe	Algorithme du point fixe	Algorithme de Dijkstra
Graphe1001.txt	1409	1646

D'après nos résultats l'algorithme du point fixe est le plus rapide. Mais comme nous l'avons suggéré cela est peut-être dû au fait que l'algorithme de Dijkstra est plus précis.

Question 25 :

```
// ===== Graphe Aléatoire =====
GrapheListe ga = new GrapheListe(10);

Affichage du graphe
System.out.println(ga.toString());

Affichage du graphe en format Graphviz
System.out.println(ga.toGraphviz());
```

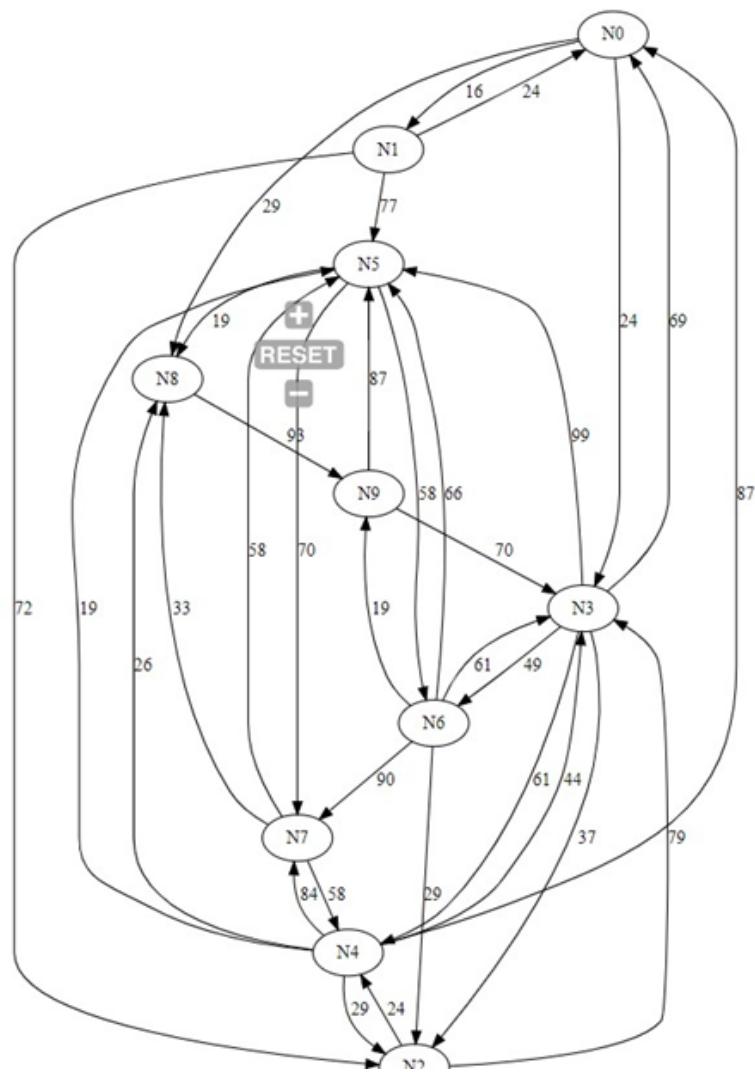
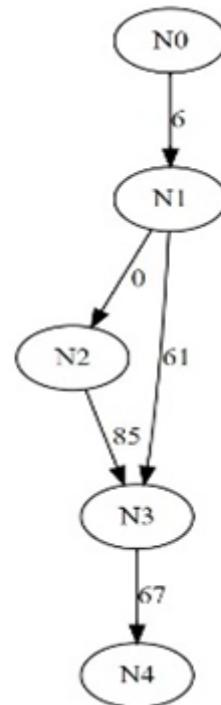
```
* Constructeur de la classe GrapheListe qui prend en paramètre un nombre de noeuds et créer un graphe aléatoire à
* @param nbNoeuds nombre de noeuds du graphe
*/
public GrapheListe(int nbNoeuds) {
    this.ensNoeud = new ArrayList<Noeud>();
    this.ensNom = new ArrayList<String>();

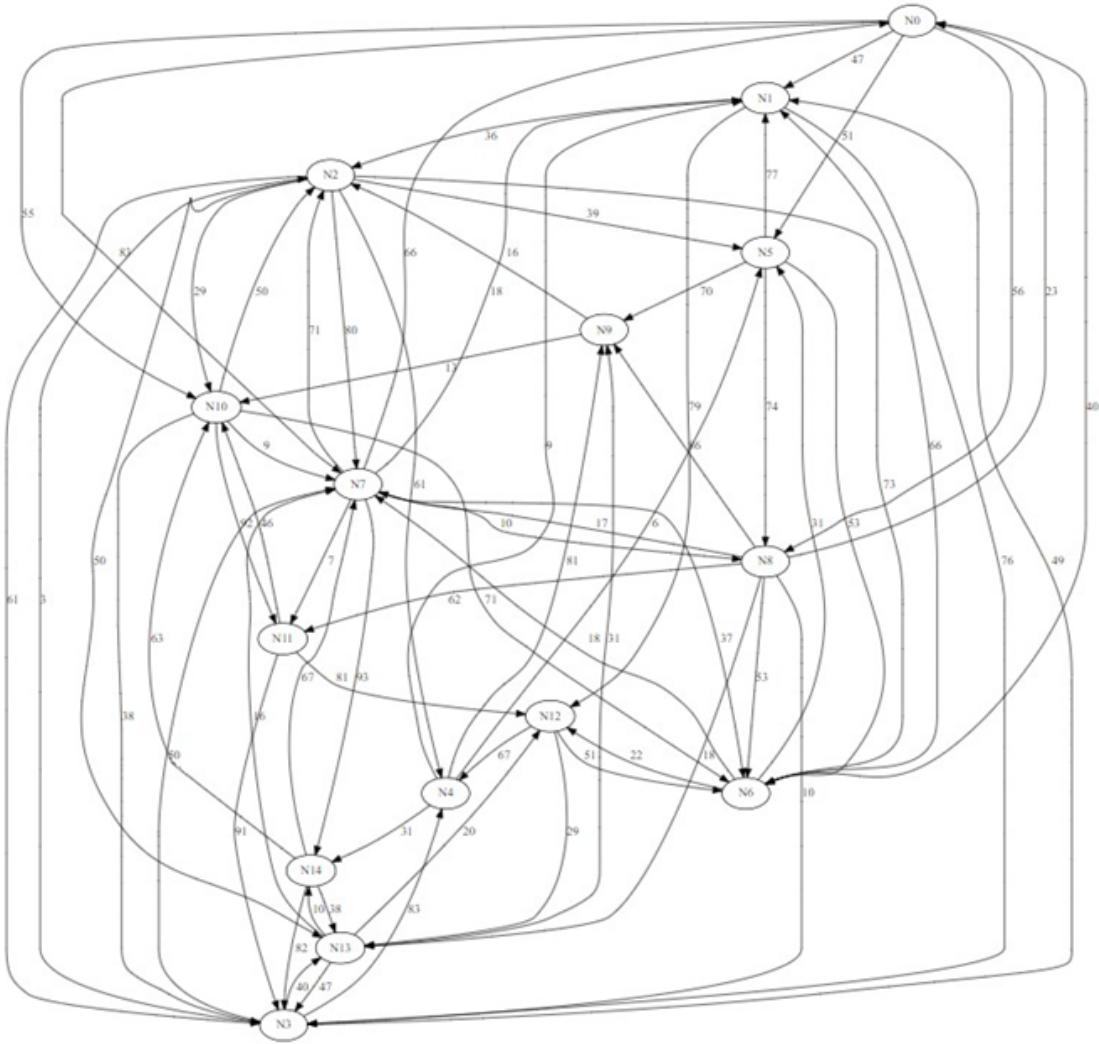
    // On ajoute les noeuds
    for(int i = 0; i < nbNoeuds; i++){
        this.ensNoeud.add(new Noeud("N" + i));
        this.ensNom.add("N" + i);
    }

    // On ajoute au moins un chemin qui va du premier au dernier noeud
    for (int i = 0; i < nbNoeuds - 1; i++)
        this.ajouterArc("N" + i, "N" + (i + 1), Math.random() * 100);

    // On ajoute des chemins aléatoires
    for (int i = 0; i < nbNoeuds; i++) {
        for (int j = 0; j < nbNoeuds; j++) {
            // On ajoute un chemin avec une probabilité de 20%
            // On vérifie que le noeud de départ et d'arrivée ne sont pas les mêmes
            if (Math.random() < 0.2 && i != j){
                // On vérifie que le noeud de départ et d'arrivée ne sont pas déjà reliés
                boolean existe = false;
                for(Arc arc : this.suivants("N" + i)){
                    if(arc.getDest().equals("N" + j)){
                        existe = true;
                        break;
                    }
                }
                if(!existe)
                    this.ajouterArc("N" + i, "N" + j, Math.random() * 100);
            }
        }
    }
}
```

Dans le constructeur de graphe aléatoire on a fait en sorte que les nœuds ne se relient pas en même, qu'un nœud n'a pas plusieurs chemins vers le même nœud. Voici des représentations de graphe aléatoirement générés avec 5, 10 et 15 nœuds.





Question 27 :

```

Long moyenne = 0;
int nbExec = 1000;
Dijkstra dj = new Dijkstra();
Graphe ga = new GrapheListe(100);
for (int i = 0; i < nbExec; i++) {

    Long startTime = System.nanoTime();

    Valeur v1 = dj.resoudre(ga, "N0");

    Long endTime = System.nanoTime();

    // obtenir la différence entre les deux valeurs de temps nano
    Long timeElapsed = endTime - startTime;

    moyenne += timeElapsed;
}

moyenne /= nbExec;
System.out.println("Execution en nanosecondes Dijkstra: " + moyenne);

moyenne = 0;
nbExec = 1000;
BellmanFord bf = new BellmanFord();
for (int i = 0; i < nbExec; i++) {

    Long startTime = System.nanoTime();

    Valeur v2 = bf.resoudre(ga, "N0");

    Long endTime = System.nanoTime();

    // obtenir la différence entre les deux valeurs de temps nano
    Long timeElapsed = endTime - startTime;

    moyenne += timeElapsed;
}

moyenne /= nbExec;
System.out.println("Execution en nanosecondes point fixe : " + moyenne);

```

Selon la moyenne des résultats de ces deux algorithmes même avec des grands graphes l'algorithme du point fixe est le plus efficace.

Question 28 :

Le ratio de performance entre les deux algorithmes change en fonction des nœuds. En effet en dessous de 100 nœuds le ratio de performance entre les deux algorithmes est stable et Bellman Ford est plus efficace. Mais quand on dépasse les 100 nœuds l'algorithme de Dijkstra devient plus performant et le ratio de performance augmente au fur et à mesure.

Question 29 :

On peut donc en conclure que l'algorithme du point fixe est plus efficace sur des cas plus simples mais quand des complications arrivent l'algorithme de Dijkstra devient meilleur.

![Une image contenant texte, capture d'écran, Police Description générée automatiquement]
(file:///C:/Users/loris/AppData/Local/Temp/msohtmlclip1/01/clip_image052.gif)

Question 30 :

```
public GrapheListe genererGraphe(){

    ArrayList<Noeud> noeuds = new ArrayList<Noeud>();
    // parcours les cases
    for (int i = 0; i < getLength(); i++) {
        for (int j = 0; j < getLengthY(); j++) {
            // si pas mur
            if(!getMur(i,j)){
                // creer noeud
                noeuds.add(new Noeud("(" + i + "," + j + ")"));
            }
        }
    }
}
```

```
// creer graphe
GrapheListe graphe = new GrapheListe(noeuds);

// ajoute les arcs
for (int i = 0; i < getLength(); i++) {
    for (int j = 0; j < getLengthY(); j++) {
        // si pas mur
        if(!getMur(i,j)){
            // ajoute les arcs
            // haut
            if(i > 0 && !getMur(i-1,j)){
                graphe.ajouterArc("(" + i + "," + j + ")", "(" + (i-1) + "," + j + ")", 1);
            }
            // bas
            if(i < getLength()-1 && !getMur(i+1,j)){
                graphe.ajouterArc("(" + i + "," + j + ")", "(" + (i+1) + "," + j + ")", 1);
            }
            // gauche
            if(j > 0 && !getMur(i,j-1)){
                graphe.ajouterArc("(" + i + "," + j + ")", "(" + i + "," + (j-1) + ")", 1);
            }
            // droite
            if(j < getLengthY()-1 && !getMur(i,j+1)){
                graphe.ajouterArc("(" + i + "," + j + ")", "(" + i + "," + (j+1) + ")", 1);
            }
        }
    }
}

return graphe;
```

Question 31 :

Dans notre MainLabyrinthe on a coder le fait de trouver tous les chemins les plus courts pour tous les points du labyrinthe étant donné que nous n'avons pas de point de fin défini dans le sujet.

```
public class MainLabyrinthe {  
  
    public static void main(String[] args) throws IOException {  
  
        // création d'un labyrinthe  
        Labyrinthe laby = new Labyrinthe("laby1.txt");  
  
        // création du graphe  
        GrapheListe graphe = laby.genererGraphe();  
  
        // affichage du graphe  
        System.out.println(graphe.toString());  
  
        BellmanFord bf = new BellmanFord();  
  
        Valeur v1 = bf.resoudre(graphe, "(1,1)");  
        System.out.println(v1.toString());  
        for (String s : graphe.listeNoeuds()) {  
            System.out.println("Chemin de (1,1) vers " + s + " : " + v1.calculerChemin(s));  
        }  
  
        Dijkstra dij = new Dijkstra();  
  
        Valeur v2 = dij.resoudre(graphe, "(1,1)");  
        System.out.println(v2.toString());  
        for (String s : graphe.listeNoeuds()) {  
            System.out.println("Chemin de (1,1) vers " + s + " : " + v2.calculerChemin(s));  
        }  
    }  
}
```

Conclusion de la SAE

Cette SAE nous a été bénéfique pour la compréhension de pourquoi il est important de faire des comparaisons algorithmiques dans nos projets. En effet on s'est rendu compte qu'en fonction des algorithmes utilisé cela pouvait faire une grande différence dans nos programmes. Cela peut nous faire gagner un temps précieux et permettre une précision plus grande.

Ce projet était parfois compliqué avec la création de l'algorithme du point fixe et ensuite le traduire en Java. Nous nous sommes rendu compte que pour éviter tout problème dans le code il fallait faire un algorithme qui respectait tous les possibles conditions. Mais cela reste enrichissant car on comprend pourquoi la rigueur dans un projet comme celui-ci est importante.