26/05/2025

Rapport SAE S2-02

TAHRI MARYAM
WISPELAERE MATHIS

Sommaire

Représentation d’un graphe

Question 1 :Écrire la classe Arc et un constructeur prenant en paramètres le nœud de destination dest (chaîne de caractères) et le coût cout (valeur réelle positive) de l’arc créé.

```java
public class Arc {  25 usages
    private String dest;  4 usages
    private double cout;  4 usages
    private String ligne;  4 usages

    /**
     * Constructeur qui prend en paramètre la destination et le cout d'un arc
     * @param dest
     * @param cout
     */
    public Arc(String dest, double cout) {  2 usages
        this.dest = dest;
        if (cout > 0) {
            this.cout = cout;
        }
        this.ligne = "Aucune"; // pas de ligne
    }

    /**
     * Constructeur qui prend en paramètre la destination et le cout d'un arc et le numéro de ligne
     * @param dest
     * @param cout
     * @param ligne
     */
    public Arc(String dest, double cout, String ligne) {  3 usages
        this.dest = dest;
        this.cout = cout;
        this.ligne = ligne;
    }

    /**
     * Méthode qui renvoie le numéro de ligne de l'arc
     * @return String
     */
    public String getLigne(){ return ligne;}  4 usages

    /**
     * Méthode qui renvoie la destination de l'arc
     * @return String
     */
    public String getDest() { return dest; }

    public double getCout() { return cout; }

    /**
     * Méthode qui renvoie l'affichage de l'arc
     * @return
     */
    public String toString() { return "(" + dest + ", " + cout + ", ligne " + ligne + ")"; }
}
```

La classe Arc modélise une liaison entre deux sommets dans un graphe. Chaque arc est défini par :

- une destination (dest);
- un coût (cout) ;
- et éventuellement une ligne (ligne);

Elle possède deux constructeurs :

- L’un permettant de créer un arc avec une destination et un coût, en initialisant la ligne à "Aucune" si elle n’est pas précisée.
- L’autre permettant de spécifier aussi le nom ou numéro de la ligne.

Des guetteur (getDest, getCout, getLigne) permettent de récupérer les attributs de l'arc.
La méthode toString fournit une représentation des arc.

Représentation d'un graphe

Question 2 :Écrire la classe Arcs.

```java
import java.util.ArrayList;
import java.util.List;

public class Arcs {  4 usages
    List<Arc> arcs;  3 usages

    /**
     * Constructeur par défaut de l'objet
     */
    public Arcs() { arcs = new ArrayList<Arc>(); }

    /**
     * Méthode d'ajout d'un arc
     * @param a
     */
    public void ajouterArc(Arc a) { arcs.add(a); }

    /**
     * Méthode de renvoie de la liste des arc
     * @return
     */
    public List<Arc> getArcs() { return arcs; }
}
```

La classe Arcs représente une collection d'arcs sortants à partir d'un même sommet dans un graphe. Elle possède une liste d'objets Arc.
Elle a :

- Un constructeur par défaut qui initialise une liste vide d'arcs.
- Une méthode ajouterArc(Arc a) permettant d'ajouter un nouvel arc à la collection.
- Une méthode getArcs() qui retourne la liste complète des arcs stockés.

Cette classe est utile pour modéliser les successeurs d'un sommet dans une structure de graphe, notamment lorsqu'on utilise une représentation en liste d'adjacence, où chaque sommet est associé à un objet Arcs.

Représentation d'un graphe

Question 3 :Écrire l'interface Graphe.

```java
import java.util.List;

public interface Graphe {  1 usage  1 implementation
    public List<String> ListeNoeuds();  1 usage  1 implementation
    public List<Arc> suivants(String n);  1 usage  1 implementation
}

```

L'interface Graphe définit le contrat minimal que doit respecter certaines classes représentant un graphe. Elle implémente deux méthodes essentielles :

- ListeNoeuds() : retourne une liste de tous les nœuds (ou sommets) du graphe. Chaque nœud est représenté par une chaîne de caractères (String).

- suivants(String n) : retourne une liste des arcs sortants à partir du nœud n, c'est-à-dire tous les arcs dont le sommet de départ est n.

Représentation d'un graphe

Question 4 :Écrire la classe GrapheListe

```java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class GrapheListe implements Graphe { 26 usages
    private ArrayList<String> noeuds; 11 usages
    private ArrayList<Arcs> adjacence; 10 usages

    /**
     * Contructeur par défaut d'un GrapheListe
     */
    public GrapheListe() { 10 usages
        noeuds = new ArrayList<>( initialCapacity: 10);
        adjacence = new ArrayList<>( initialCapacity: 10);
    }

    /**
     * Méthode qui permet d'ajouté un arc pondéré entre deux noeuds
     * Si les noeuds de départ ou de destination n'existe pas, ils sont créer et ajouté au graphe
     * @param depart
     * @param destination
     * @param count
     */
    public void ajouterArc(String depart, String destination, double count) { 49 usages
        int ind1 = getIndice(depart);

        if (!noeuds.contains(depart)) {
            ajouterNoeud(depart);
            ind1 = getIndice(depart);
            adjacence.add(new Arcs());
        }
        if (!noeuds.contains(destination)) {
            ajouterNoeud(destination);
            adjacence.add(new Arcs());
        }
        Arc arc = new Arc(destination, count);
        adjacence.get(ind1).ajouterArc(arc);
    }

    /**
     * Méthode qui permet d'ajouter un arc au graphe a partir d'un arc déjà existant
     * @param depart
     * @param a
     */
    public void ajouterArc(String depart, Arc a) { adjacence.get(getIndice(depart)).ajouterArc(a); }

    /**
     * Méthode qui permet de récupré l'indice du noeud dans la liste de noeud en attribut
     * @param n
     * @return
     */
    public int getIndice(String n) { 5 usages
        for (int i = 0; i < this.noeuds.size(); i++) {
            if (this.noeuds.get(i).equals(n)) {
                return i;
            }
        }
        return -1;
    }

    /**
     * méthode qui renvoie l'attribut qui contient la liste de noeuds
     * @return
     */
    public List<String> ListeNoeuds() { return this.noeuds; }
```

```java
    /**
     * Méthode qui renvoie les arcs qui parte du noeuds en paramètre
     * @param s
     * @return
     */
    public List<Arc> suivants(String s) { 7 usages
        int ind = getIndice(s);
        return adjacence.get(ind).getArcs();
    }

    /**
     * Méthode pour rajouté un noeud au graphe
     * @param n
     */
    public void ajouterNoeud(String n) { 3 usages
        if (!noeuds.contains(n)) {
            noeuds.add(n);
            adjacence.add(new Arcs()); // Initialize an Arcs object for the new node
        }
    }

    /**
     * Méthode qui renvoie l'affichage de l'objet
     * @return
     */
    public String toString(){
        String res = "";
        for (int i = 0; i < this.noeuds.size(); i++) {
            res += this.noeuds.get(i) + " -> ";
            if (!(adjacence.isEmpty())){
                List<Arc> as = adjacence.get(i).getArcs();
                for (int j = 0; j < as.size(); j++) {
                    res += as.get(j).toString() + " ";
                }
                res += "\n";
            }
        }
        return res;
    }

    /**
     * Constructeur qui prend en paramètre un fichier et construit le graphe dans le fichier
     * @param nomFichier
     */
    public GrapheListe(String nomFichier){ 5 usages
        noeuds = new ArrayList<>( initialCapacity: 10);
        adjacence = new ArrayList<>( initialCapacity: 10);
        try{
            BufferedReader br = new BufferedReader(new FileReader(nomFichier));
            ArrayList<String> txt = new ArrayList<~>();
            String line =br.readLine();
            while (line != null){
                txt.add(line);
                line = br.readLine();
            }
            br.close();
            for (String s : txt) {
                String[] split = s.split( regex: "\t");
                this.ajouterArc(split[0],split[1],Double.valueOf(split[2]));
            }
        }catch( IOException e){
            System.out.println("problème du fichier");
        }
    }
}
```

Représentation d'un graphe

Question 4 :Écrire la classe GrapheListe

La classe GrapheListe est une implémente de l'interface Graphe. Elle représente un graphe à l'aide d'une liste. Elle utilise deux structures principales :

noeuds : une liste contenant les noms des sommets
adjacence : une liste parallèle à noeuds, où chaque élément est un objet Arcs représentant les arcs sortants du sommet correspondant.

Elle possède plusieurs méthode:
La méthode ajouterNoeud(String n) ajoute un nouveau sommet au graphe s'il n'existe pas encore.

ajouterArc(String depart, String destination, double cout) permet d'ajouter un arc pondéré entre deux sommets. Si l'un des sommets n'existe pas encore, il est automatiquement ajouté.

Une autre version de ajouterArc , ajouterArc(String depart, Arc a), permet d'ajouter un arc déjà construit.

getIndice(String n) retourne l'indice d'un sommet dans la liste noeuds.

ListeNoeuds() retourne tous les sommets du graphe.

suivants(String s) retourne la liste des arcs sortants du sommet s.

La méthode toString() fournit une représentation du graphe : pour chaque sommet, elle liste tous les arcs sortants.

Le constructeur GrapheListe(String nomFichier) permet de créer un graphe en lisant les données depuis un fichier texte.

Représentation d'un graphe

Question 5 :Écrire une méthode main qui crée le même graphe que celui représenté en Figure 1

```
55
56      public static void main(String[] args) {
57          GrapheListe graphe = new GrapheListe();
58          graphe.ajouterArc( depart: "A", destination: "B", count: 12.0);
59          graphe.ajouterArc( depart: "A", destination: "D", count: 87.0);
60          graphe.ajouterArc( depart: "B", destination: "E", count: 11.0);
61          graphe.ajouterArc( depart: "C", destination: "A", count: 19);
62          graphe.ajouterArc( depart: "D", destination: "B", count: 23.0);
63          graphe.ajouterArc( depart: "D", destination: "C", count: 10.0);
64          graphe.ajouterArc( depart: "E", destination: "D", count: 43.0);
65          System.out.println(graphe);
66      }
67  }
68
```

Ce main permet uniquement de créer un graphe en ajoutant des arc ce qui ajoute aussi des nœuds. a la fin on utilise un system.out pour vérifier que notre affichage se fait correctement.

Représentation d'un graphe

Question 6 :Dans la classe GrapheListe, écrire une méthode toString permettant d'afficher le graphe sous cette forme.

```java
/**
 * Méthode qui renvoie l'affichage de l'objet
 * @return
 */
public String toString(){
    String res = "";
    for (int i = 0; i < this.noeuds.size(); i++) {
        res += this.noeuds.get(i) + " -> ";
        if (!(adjacence.isEmpty())){
            List<Arc> as = adjacence.get(i).getArcs();
            for (int j = 0; j < as.size(); j++) {
                res += as.get(j).toString() + " ";
            }
            res += "\n";
        }
    }
    return res;
}
```

La méthode toString() permet de générer une représentation textuelle lisible du graphe. Elle affiche, pour chaque sommet du graphe, la liste des arcs sortants (avec destination, coût, et éventuellement ligne).

Représentation d'un graphe

Question 7 :Écrire des tests unitaires vous permettant de vérifier que le graphe est bien construit

```java
@Test
public void testGrapheContientNoeud() {

    GrapheListe graphe = new GrapheListe();
    graphe.ajouterArc( depart: "A",  destination: "B",  count: 12);
    graphe.ajouterArc( depart: "A",  destination: "D",  count: 87);

    List<String> noeuds = graphe.ListeNoeuds();
    assertTrue(noeuds.contains("A"));
    assertTrue(noeuds.contains("B"));
    assertTrue(noeuds.contains("D"));

}

@Test
public void testGrapheSuivantOk() {
    GrapheListe graphe = new GrapheListe();
    graphe.ajouterArc( depart: "A",  destination: "B",  count: 12);
    graphe.ajouterArc( depart: "A",  destination: "D",  count: 87);

    List<Arc> arcsA = graphe.suivants( s: "A");
    assertEquals( expected: 2, arcsA.size());
    assertEquals( expected: "B", arcsA.get(0).getDest());
    assertEquals( expected: 12, arcsA.get(0).getCout());
    assertEquals( expected: "D", arcsA.get(1).getDest());
    assertEquals( expected: 87, arcsA.get(1).getCout());
}
@Test
public void testGrapheArcInconnu(){
    GrapheListe graphe = new GrapheListe();
    graphe.ajouterArc( depart: "A",  destination: "B",  count: 12);
    graphe.ajouterArc( depart: "A",  destination: "D",  count: 87);

    assertEquals( expected: -1 , graphe.getIndice( n: "C"));
}
}
```

TEST 1 vérifie que :
Lorsqu'on ajoute des arcs, les sommets source et destination sont bien enregistrés dans la liste des nœuds du graphe.
Cela garantit que la méthode ajouterArc crée automatiquement les sommets s'ils n'existent pas encore.

TEST 2 vérifie que :
La méthode suivants("A") retourne bien la liste des arcs sortants depuis le sommet "A".
Il verifie les destination et leur cout et qu'il y a bien 2 arcs.

TEST 3 verifie que :
La méthode getIndice("C") retourne bien -1 si le sommet "C" n'existe pas dans le graphe.

Calcul du plus court chemin par point fixe

Question 8 : En utilisant le TAD Graphe, écrire l'algorithme (en pseudo-code, sans oublier le lexique) de la fonction pointFixe(Graphe g, Noeud depart) qui modifie les valeurs (parent et distance) associées aux nœuds du graphe pour trouver le chemin le plus court partant du nœud de départ passé en paramètre. On supposera qu'on peut directement changer les valeurs de L(X) et parent(X) par une simple affectation comme L(X) ← +∞.

```
Lexique:
    Graphe g: le graphe à traiter
    Noeud depart: le nœud de départ
    Valeurs L: la fonction de valeur
    Noeud X, N: des nœuds du graphe
    double cout: le coût d'un arc
    boolean modifié: indique si une valeur a été modifiée

Algorithme pointFixe(Graphe g, Noeud depart):
    Valeurs L = new Valeurs()
    Pour chaque Noeud X dans g.listeNoeuds() faire
        L(X) <- infini
        Parent(X)<- null
    Fin Pour
    L.setValeur(depart, 0)

    modifié = true
    Tant que modifié faire
        modifié = false
        Pour chaque Noeud X dans g.listeNoeuds() faire
            Pour chaque Arc (N, cout) dans g.suivants(X) faire
                Si L.getValeur(X) + cout < L.getValeur(N) alors
                    L.setValeur(N, L.getValeur(X) + cout)
                    L.setParent(N, X)
                    modifié = true
                Fin Si
            Fin Pour
        Fin Pour
    Fin Tant que
    Retourner L
Fin
```

Calcul du plus court chemin par point fixe

Question 9 :En utilisant la classe Valeurs fournie et votre algorithme, implémenter l'algorithme du point fixe dans une classe nommée BellmanFord en programmant la méthode de signature Valeurs resoudre(Graphe g, String depart) Cette méthode prend en paramètres un graphe g et une chaîne représentant le nœud de départ et retourne un objet de type Valeurs correctement construit contenant les distances et les parents de chaque nœud (après convergence de l'algorithme).

```java
import java.util.List;

public class BellmanFord implements Resoudre {  10 usages

    /**
     * Constructeur par défaut
     */
    public BellmanFord() {}  5 usages

    /**
     * Méthode qui permet d'appliquer l'algorithme du point fixe de BellmanFord
     * @param g
     * @param depart
     * @return
     */
    public Valeurs resoudre(Graphe g, String depart){  18 usages
        Valeurs l = new Valeurs();
        for (int i = 0 ; i < g.ListeNoeuds().size(); i++){
            l.setValeur(g.ListeNoeuds().get(i), Double.MAX_VALUE);
            l.setParent(g.ListeNoeuds().get(i),  parent: null);
        }
        l.setValeur(depart,  valeur: 0);
        boolean modif = true;
        while (modif){
            modif = false;
            for (String noeud : g.ListeNoeuds()) {
                List<Arc> arcs = g.suivants(noeud);
                for (Arc arc : arcs) {
                    double nouvelleValeur = l.getValeur(noeud) + arc.getCout();
                    if (nouvelleValeur < l.getValeur(arc.getDest())) {
                        l.setValeur(arc.getDest(), nouvelleValeur);
                        l.setParent(arc.getDest(), noeud);
                        modif = true;
                    }
                }
            }
        }
        return l;
    }
```

La classe BellmanFord implémente l'algorithme du point fixe de Bellman-Ford, utilisé pour calculer les plus courts chemins dans un graphe pondéré.
Elle contient une méthode principale resoudre qui, à partir d'un sommet donné, retourne un objet Valeurs contenant la distance minimale depuis ce sommet vers tous les autres. Elle contient aussi le parent de chaque nœud dans le chemin optimal.

Calcul du plus court chemin par point fixe

Question 10 :Écrire une méthode main dans la classe Main qui applique l'algorithme du point fixe sur le graphe fourni pour construire le chemin le plus court. Afficher les valeurs de distance pour chaque nœud et vérifier qu'elles correspondent aux valeurs que vous avez calculées dans le module ff Graphe ff.

```java
public class MainBellmanFord {

    public static void main(String[] args) {
        GrapheListe graph = new GrapheListe();

        graph.ajouterArc( depart: "A",  destination: "B",  count: 12);
        graph.ajouterArc( depart: "A",  destination: "D",  count: 87);
        graph.ajouterArc( depart: "B",  destination: "E",  count: 11);
        graph.ajouterArc( depart: "C",  destination: "A",  count: 10);
        graph.ajouterArc( depart: "D",  destination: "C",  count: 10);
        graph.ajouterArc( depart: "D",  destination: "B",  count: 23);
        graph.ajouterArc( depart: "E",  destination: "D",  count: 43);

        Valeurs v =  new Valeurs();
        BellmanFord bf = new BellmanFord();
        v = bf.resoudre(graph, depart: "A");
        //System.out.println(v.toString());
    }
}
```

Ce main permet de représenter un graphe qui applique la methode de Bellman Ford en nous renvoyant le chemin le plus court .

Calcul du plus court chemin par point fixe

Question 11 :Écrire un test unitaire qui vérifie que l'algorithme du point fixe est correct et que les parents des nœuds sont bien calculés.

```java
    @Test
    public void testResoudreParentChemin() {
        GrapheListe graphe = new GrapheListe();
        graphe.ajouterArc( depart: "A",  destination: "B",  count: 12);
        graphe.ajouterArc( depart: "A",  destination: "D",  count: 87);
        graphe.ajouterArc( depart: "B",  destination: "E",  count: 11);
        graphe.ajouterArc( depart: "D",  destination: "B",  count: 23);
        graphe.ajouterArc( depart: "D",  destination: "C",  count: 10);
        graphe.ajouterArc( depart: "E",  destination: "D",  count: 43);
        graphe.ajouterArc( depart: "C",  destination: "A",  count: 19);

        BellmanFord bellmanFord = new BellmanFord();
        Valeurs valeurs = bellmanFord.resoudre(graphe,  depart: "A");

        assertEquals( expected: null, valeurs.getParent( nom: "A"));
        assertEquals( expected: "A", valeurs.getParent( nom: "B"));
        assertEquals( expected: "D", valeurs.getParent( nom: "C"));
        assertEquals( expected: "E", valeurs.getParent( nom: "D"));
        assertEquals( expected: "B", valeurs.getParent( nom: "E"));
    }
    @Test
    public void testResoudreValeurChemin() {
        GrapheListe graphe = new GrapheListe();
        graphe.ajouterArc( depart: "A",  destination: "B",  count: 12);
        graphe.ajouterArc( depart: "A",  destination: "D",  count: 87);
        graphe.ajouterArc( depart: "B",  destination: "E",  count: 11);
        graphe.ajouterArc( depart: "D",  destination: "B",  count: 23);
        graphe.ajouterArc( depart: "D",  destination: "C",  count: 10);
        graphe.ajouterArc( depart: "E",  destination: "D",  count: 43);
        graphe.ajouterArc( depart: "C",  destination: "A",  count: 19);

        BellmanFord bellmanFord = new BellmanFord();
        Valeurs valeurs = bellmanFord.resoudre(graphe,  depart: "A");

        assertEquals( expected: 0, valeurs.getValeur( nom: "A"));
        assertEquals( expected: 12, valeurs.getValeur( nom: "B"));
        assertEquals( expected: 76, valeurs.getValeur( nom: "C"));
        assertEquals( expected: 66, valeurs.getValeur( nom: "D"));
        assertEquals( expected: 23, valeurs.getValeur( nom: "E"));

    }
}
```

Le premier test Sert a vérifier les parent de chaque noeud dans le graphe.

Le second test fait la meme chose avec les valeurs.

Calcul du plus court chemin par point fixe

Question 12 :Écrire la méthode List calculerChemin(String destination) dans la classe Valeurs. Cette méthode retourne une liste de nœuds correspondant au chemin menant au nœud passé en paramètre depuis le point de départ donné lors de la construction de l'objet Valeurs.

```
95      public List<String> calculerChemin(String destination){  1 usage
96          List<String> res = new ArrayList<>();
97          String noeudParent = destination;
98          while (noeudParent != null) {
99              res.add(noeudParent);
100             noeudParent = parent.get(noeudParent);
101         }
102         return res;
103     }
104 }
```

Cette methode sert a afficher le chemin sous forme de liste.

Calcul du meilleur chemin par Dijkstra

Question 13 :Dans une classe Dijkstra, recopier cet algorithme en commentaire puis traduisez ces lignes en java pour écrire la méthode Valeurs resoudre(Graphe g, String depart) qui prend en paramètres le graphe et le nom du nœud de départ pour calculer les plus courts chemins vers les autres nœuds du graphe avec l'algorithme de Dijkstra

```java
/**
 * Méthode qui permet d'appliquer l'algorithme de Dijkstra
 * @param g
 * @param depart
 * @return
 */
public Valeurs resoudre(Graphe g, String depart) {  18 usages
    Valeurs valeurs = new Valeurs();
    List<String> noeuds = g.ListeNoeuds();
    List<String> Q = new ArrayList<>(noeuds);

    for (String v : noeuds) {
        valeurs.setValeur(v, Double.MAX_VALUE);
        valeurs.setParent(v,  parent: null);
    }

    valeurs.setValeur(depart,  valeur: 0.0);

    while (!Q.isEmpty()) {
        String u = trouverMin(Q, valeurs);
        Q.remove(u);
        for (Arc arc : g.suivants(u)) {
            String v = arc.getDest();
            if (Q.contains(v)) {
                double d = valeurs.getValeur(u) + arc.getCout();
                if (d < valeurs.getValeur(v)) {
                    valeurs.setValeur(v, d);
                    valeurs.setParent(v, u);
                }
            }
        }
    }
    return valeurs;
}
```

Cet algorithme represente celui de dijkstra.

Calcul du meilleur chemin par Dijkstra

Question 14 :De la même manière que pour l'algorithme du point fixe, écrire des tests unitaires pour vérifier le bon fonctionnement de votre algorithme

```java
public class TestDijkstra {
    @Test
    public void testResoudreParentChemin() {
        GrapheListe graphe = new GrapheListe();
        graphe.ajouterArc( depart: "A",  destination: "B",  count: 12);
        graphe.ajouterArc( depart: "A",  destination: "D",  count: 87);
        graphe.ajouterArc( depart: "B",  destination: "E",  count: 11);
        graphe.ajouterArc( depart: "D",  destination: "B",  count: 23);
        graphe.ajouterArc( depart: "D",  destination: "C",  count: 10);
        graphe.ajouterArc( depart: "E",  destination: "D",  count: 43);
        graphe.ajouterArc( depart: "C",  destination: "A",  count: 19);

        Dijkstra dijkstra= new Dijkstra();
        Valeurs valeurs = dijkstra.resoudre(graphe,  depart: "A");

        assertEquals( expected: null, valeurs.getParent( nom: "A"));
        assertEquals( expected: "A", valeurs.getParent( nom: "B"));
        assertEquals( expected: "D", valeurs.getParent( nom: "C"));
        assertEquals( expected: "E", valeurs.getParent( nom: "D"));
        assertEquals( expected: "B", valeurs.getParent( nom: "E"));
    }

    public void testResoudreValeurChemin() {
        GrapheListe graphe = new GrapheListe();
        graphe.ajouterArc( depart: "A",  destination: "B",  count: 12);
        graphe.ajouterArc( depart: "A",  destination: "D",  count: 87);
        graphe.ajouterArc( depart: "B",  destination: "E",  count: 11);
        graphe.ajouterArc( depart: "D",  destination: "B",  count: 23);
        graphe.ajouterArc( depart: "D",  destination: "C",  count: 10);
        graphe.ajouterArc( depart: "E",  destination: "D",  count: 43);
        graphe.ajouterArc( depart: "C",  destination: "A",  count: 19);

        Dijkstra dijkstra= new Dijkstra();
        Valeurs valeurs = dijkstra.resoudre(graphe,  depart: "A");

        assertEquals( expected: 0, valeurs.getValeur( nom: "A"));
        assertEquals( expected: 12, valeurs.getValeur( nom: "B"));
        assertEquals( expected: 76, valeurs.getValeur( nom: "C"));
        assertEquals( expected: 66, valeurs.getValeur( nom: "D"));
        assertEquals( expected: 23, valeurs.getValeur( nom: "E"));

    }
}
```

Le premier test Sert a vérifier les parent de chaque noeud dans le graphe.

Le second test fait la meme chose avec les valeurs.

Calcul du meilleur chemin par Dijkstra

Question 15 :Écrire un programme principal MainDijkstra qui : • utilise un graphe par défaut ; • calcule les chemins les plus courts pour des nœuds donnés ; • affiche des chemins pour des nœuds donnés.

```java
public class MainDijkstra {
    public static void main(String[] args) {
        GrapheListe graph = new GrapheListe();

        graph.ajouterArc( depart: "A", destination: "B", count: 12);
        graph.ajouterArc( depart: "A", destination: "D", count: 87);
        graph.ajouterArc( depart: "B", destination: "E", count: 11);
        graph.ajouterArc( depart: "C", destination: "A", count: 10);
        graph.ajouterArc( depart: "D", destination: "C", count: 10);
        graph.ajouterArc( depart: "D", destination: "B", count: 23);
        graph.ajouterArc( depart: "E", destination: "D", count: 43);

        Valeurs v2 =  new Valeurs();
        Dijkstra dj = new Dijkstra();
        v2 = dj.resoudre(graph, depart: "A");
        //System.out.println(v2.toString());
    }
}
```

Ce main permet de représenter un graphe qui applique la methode de Dijkstra en nous renvoyant le chemin le plus court .

Validation et expérimentation

Question 16 :Dans la classe GrapheListe, écrire un constructeur prenant en paramètre le nom du fichier contenant le descriptif (sous forme de liste d'arcs) du graphe et construisant l'objet graphe correspondant

```java
/**
 * Constructeur qui prend en paramètre un fichier et construit le graphe dans le fichier
 * @param nomFichier
 */
public GrapheListe(String nomFichier){  5 usages
    noeuds = new ArrayList<>( initialCapacity: 10);
    adjacence = new ArrayList<>( initialCapacity: 10);
    try{
        BufferedReader br = new BufferedReader(new FileReader(nomFichier));
        ArrayList<String> txt = new ArrayList<~>();
        String line =br.readLine();
        while (line != null){
            txt.add(line);
            line = br.readLine();
        }
        br.close();
        for (String s : txt) {
            String[] split = s.split( regex: "\t");
            this.ajouterArc(split[0],split[1],Double.valueOf(split[2]));
        }
    }catch( IOException e){
        System.out.println("problème du fichier");
    }
}
```

Le constructeur GrapheListe(String nomFichier) permet de créer un graphe en lisant les données depuis un fichier texte.

Validation et expérimentation

Question 17 :Présenter les résultats obtenus. Quel algorithme fonctionne le mieux experimentalement ? Vous attendiez vous à ces résultats ? Pourquoi ?

Graphe 1

```
temps d'éxecution de Dijkstra ; 456600ms
```

```
Temps d'éxecution de bellmanFord
404600ms
```

Graphe 2

```
temps d'éxecution de Dijkstra :186800ms
```

```
Temps d'éxecution de bellmanFord
191100ms
```

Graphe 3

```
temps d'éxecution de Dijkstra ; 1451300ms
```

```
Temps d'éxecution de bellmanFord : 1276500ms
```

Graphe 4

```
temps d'éxecution de Dijkstra : 791700ms
```

```
Temps d'éxecution de bellmanFord : 821500ms
```

Graphe 5

```
temps d'éxecution de Dijkstra : 1056800ms
```

```
Temps d'éxecution de bellmanFord : 1302400ms
```

En général le temps d'exécution avec la méthode Dijkstra est plus rapide que la méthode de Bellman Ford. Nous nous attendions à ces resultat car il y a beaucoup plus de boucle dans la methode de Belleman Ford que dans celle de Dijkstra.

Application : recherche de plus courts chemins dans le métro parisien

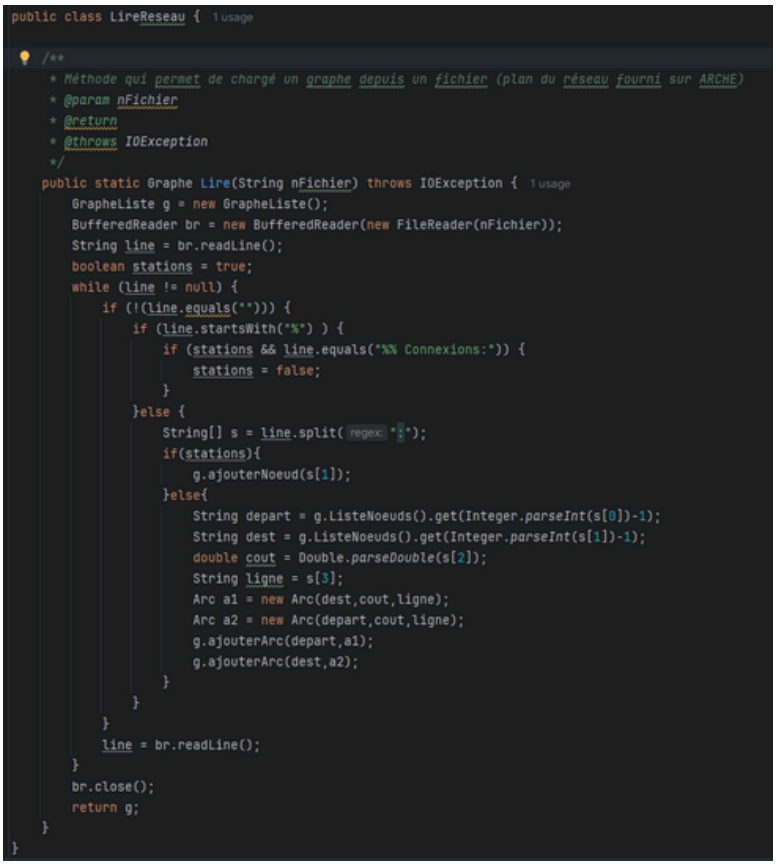
Question 18 :Modifier la classe Arc afin d'ajouter le numéro de ligne de l'arc en question.

```java
/**
 * Constructeur qui prend en paramètre la destination et le cout d'un arc et le numéro de ligne
 * @param dest
 * @param cout
 * @param ligne
 */
public Arc(String dest, double cout, String ligne) {  3 usages
    this.dest = dest;
    this.cout = cout;
    this.ligne = ligne;
}
```

Il suffit d'ajouter un attribut ligne et de l'initialiser dans le constructeur.

Application : recherche de plus courts chemins dans le métro parisien

Question 19 :Écrire une classe LireReseau qui contient une méthode statique Graphe lire(String fichier) qui construit et retourne un graphe à partir d'un plan de réseau dans le format ci-dessus.



```java
public class LireReseau {  1 usage

    /**
     * Méthode qui permet de chargé un graphe depuis un fichier (plan du réseau fourni sur ARCHE)
     * @param nFichier
     * @return
     * @throws IOException
     */
    public static Graphe Lire(String nFichier) throws IOException {  1 usage
        GrapheListe g = new GrapheListe();
        BufferedReader br = new BufferedReader(new FileReader(nFichier));
        String line = br.readLine();
        boolean stations = true;
        while (line != null) {
            if (!(line.equals(""))) {
                if (line.startsWith("%") ) {
                    if (stations && line.equals("%% Connexions:")) {
                        stations = false;
                    }
                }else {
                    String[] s = line.split( regex: ":");
                    if(stations){
                        g.ajouterNoeud(s[1]);
                    }else{
                        String depart = g.ListeNoeuds().get(Integer.parseInt(s[0])-1);
                        String dest = g.ListeNoeuds().get(Integer.parseInt(s[1])-1);
                        double cout = Double.parseDouble(s[2]);
                        String ligne = s[3];
                        Arc a1 = new Arc(dest,cout,ligne);
                        Arc a2 = new Arc(depart,cout,ligne);
                        g.ajouterArc(depart,a1);
                        g.ajouterArc(dest,a2);
                    }
                }
            }
            line = br.readLine();
        }
        br.close();
        return g;
    }
}
```

La classe LireReseau permet de construire un graphe à partir d'un fichier texte représentant un réseau (comme un plan de métro).

Lire(String nFichier):
Charge un fichier contenant une liste de stations suivie d'une liste de connexions.
Les stations sont ajoutées comme nœuds du graphe.
Les connexions (avec coûts et lignes) sont ajoutées comme arcs entre les stations concernées.

Application : recherche de plus courts chemins dans le métro parisien

Question 20 :Dans une classe MainMetro, définir une méthode main permettant d'appliquer vos deux algorithmes de recherche de plus courts chemin à 5 trajets de votre choix. Dans votre rapport, vous consignerez sous forme de tableau, la station de départ, celle d'arrivée, le plus court chemin trouvé (sous forme de liste d'identifiants de station) et le temps de calcul de celui-ci par chacun de vos algorithmes :
départ arrivée chemin temps calcul Bellman-Ford temps calcul Dijkstra

Depart	Arrivée	chemin	Temps dijkstra	Temps Belleman Ford
Saint-Augustin	Maraîchers	[Maraîchers, Buzenval, Nation, Rue des Boulets, Charonne, Voltaire, Saint-Ambroise, Oberkampf, République, Strasbourg Saint-Denis, Bonne Nouvelle, Rue Montmartre, Grands Boulevards, Richelieu Drouot, Chaussée d'Antin, La Fayette, Havre Caumartin, Saint-Augustin]	5687600ns	15123700ns
Solférino	Tuileries	[Tuileries, Concorde, Assemblée Nationale, Solférino]	4312800ns	7236300ns
Maisons-Alfort les Juilliottes	Raspail	[Raspail, Denfert Rochereau, Saint-Jacques, Glacière, Corvisart, Place d'Italie, Nationale, Chevaleret, Quai de la Gare, Bercy, Dugommier, Daumesnil, Michel Bizot, Porte Dorée, Porte de Charenton, Liberté, Charenton-Écoles, École Vétérinaire de Maisons-Alfort, Maisons-Alfort, Stade, Maisons-Alfort les Juilliottes]	2723300ns	11701400ns
Malesherbes	Alésia	[Alésia, Mouton-Duvernet, Denfert Rochereau, Raspail, Edgar Quinet, Montparnasse Bienvenue, Notre-Dame-des-Champs, Rennes, Sèvres Babylone, Rue du Bac, Solférino, Assemblée Nationale, Concorde, Madeleine, Saint-Lazare, Europe, Villiers, Malesherbes]	2865000ns	4727600ns
Pyrénées	Bel Air	[Bel Air, Picpus, Nation, Avron, Alexandre Dumas, Philippe-Auguste, Père Lachaise, Ménilmontant, Couronnes, Belleville, Pyrénées]	2738300ns	4921000ns

Application : recherche de plus courts chemins dans le métro parisien

Question 21 :Ecrire une version 2 de vos algorithmes de plus courts chemins (nommée resoudre2) qui ajoute une pénalité de 10 au coût de l'arc en cas de changement de ligne (il vous faudra donc comparer la ligne empruntée pour arriver à une station et celle utilisée par le nouvel arc).

Dijkstra

```java
/**
 * Méthode qui permet d'appliquer l'algorithme de Dijkstra en rajoutant une pénalité de temps sur les changement de ligne
 * @param g
 * @param depart
 * @return
 */
public Valeurs resoudre2(Graphe g, String depart) {  no usages
    Valeurs valeurs = new Valeurs();
    List<String> noeuds = g.ListeNoeuds();
    List<String> Q = new ArrayList<>(noeuds);
    String ligne="";

    for (String v : noeuds) {
        valeurs.setValeur(v, Double.MAX_VALUE);
        valeurs.setParent(v,  parent null);
    }

    valeurs.setValeur(depart,  valeur 0.0);
    String u = trouverMin(Q, valeurs);
    Q.remove(u);
    //initialisation de la ligne
    for (Arc arc : g.suivants(u)) {
        String v = arc.getDest();
        ligne = arc.getLigne();
        if (Q.contains(v)) {
            double d = valeurs.getValeur(u) + arc.getCout();
            if (d < valeurs.getValeur(v)) {
                valeurs.setValeur(v, d);
                valeurs.setParent(v, u);
            }
        }
    }
    while (!Q.isEmpty()) {
        u = trouverMin(Q, valeurs);
        Q.remove(u);
        for (Arc arc : g.suivants(u)) {
            String v = arc.getDest();
            if (Q.contains(v)) {
                double d = valeurs.getValeur(u) + arc.getCout();
                if (!arc.getLigne().equals(ligne)) {
                    d+=10;
                }
                if (d < valeurs.getValeur(v)) {
                    valeurs.setValeur(v, d);
                    valeurs.setParent(v, u);
                    ligne = arc.getLigne();
                }
            }
        }
    }
    return valeurs;
}
```

Cette méthode calcule les plus courts chemins depuis un sommet depart, mais avec une pénalité de +10 à chaque changement de ligne (ex. : changement de métro).

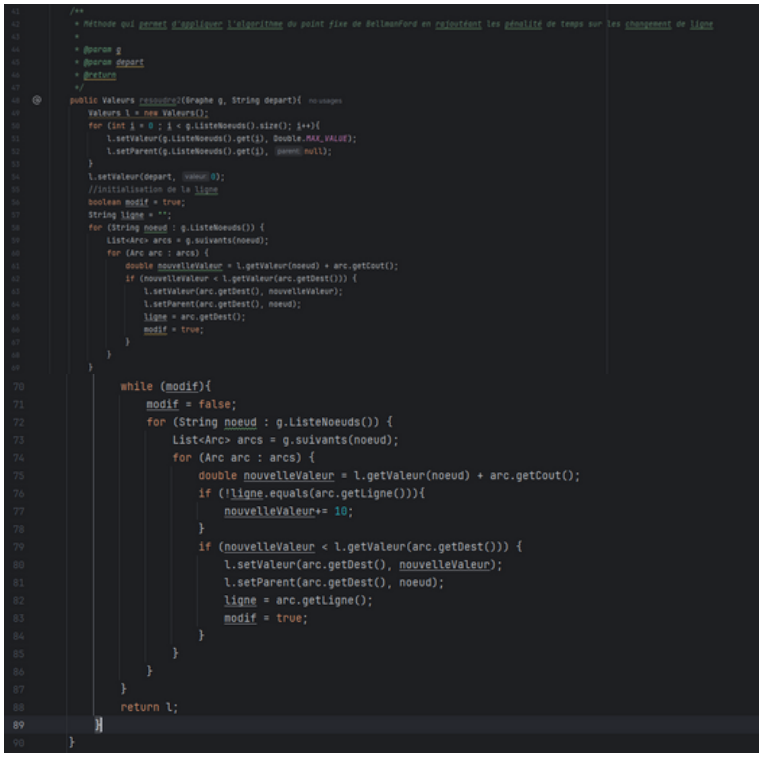
Initialise tous les sommets avec une distance infinie (Double.MAX_VALUE), sauf le point de départ .
Utilise une liste Q des sommets non encore traités.

À chaque itération :
Sélectionne le sommet u de Q ayant la valeur minimale et met a jour les voisin.
Si le voisin est sur une autre ligne que celle utilisée pour venir à u, ajoute 10 au coût du trajet.

Application : recherche de plus courts chemins dans le métro parisien

Question 21 :Ecrire une version 2 de vos algorithmes de plus courts chemins (nommée resoudre2) qui ajoute une pénalité de 10 au coût de l'arc en cas de changement de ligne (il vous faudra donc comparer la ligne empruntée pour arriver à une station et celle utilisée par le nouvel arc).

Bellman Ford



```
    /**
     * Méthode qui permet d'appliquer l'algorithme du point fixe de BellmanFord en rajoutant les pénalité de temps sur les changement de ligne
     *
     * @param g
     * @param depart
     * @return
     */
    public Valeurs resoudre2(Graphe g, String depart){
        Valeurs l = new Valeurs();
        for (int i = 0 ; i < g.ListeNoeuds().size(); i++){
            l.setValeur(g.ListeNoeuds().get(i), Double.MAX_VALUE);
            l.setParent(g.ListeNoeuds().get(i), parent: null);
        }
        l.setValeur(depart, valeur: 0);
        //initialisation de la ligne
        boolean modif = true;
        String ligne = "";
        for (String noeud : g.ListeNoeuds()) {
            List<Arc> arcs = g.suivants(noeud);
            for (Arc arc : arcs) {
                double nouvelleValeur = l.getValeur(noeud) + arc.getCout();
                if (nouvelleValeur < l.getValeur(arc.getDest())) {
                    l.setValeur(arc.getDest(), nouvelleValeur);
                    l.setParent(arc.getDest(), noeud);
                    ligne = arc.getDest();
                    modif = true;
                }
            }
        }
        while (modif){
            modif = false;
            for (String noeud : g.ListeNoeuds()) {
                List<Arc> arcs = g.suivants(noeud);
                for (Arc arc : arcs) {
                    double nouvelleValeur = l.getValeur(noeud) + arc.getCout();
                    if (!ligne.equals(arc.getLigne())){
                        nouvelleValeur+= 10;
                    }
                    if (nouvelleValeur < l.getValeur(arc.getDest())) {
                        l.setValeur(arc.getDest(), nouvelleValeur);
                        l.setParent(arc.getDest(), noeud);
                        ligne = arc.getLigne();
                        modif = true;
                    }
                }
            }
        }
        return l;
    }
```

Cette méthode initialise tous les sommets en leur attribuant une valeur infinie , sauf le point de départ .
On parcourt tous les sommets pour faire une première mise à jour des valeurs et parents sans pénalité de ligne (ligne initialisée juste après ce passage).:
Tant qu'il y a des modifications :
Pour chaque arc du graphe , si le trajet change de ligne (comparé à la précédente), +10 est ajouté au coût.
Si la nouvelle distance est meilleure, on met à jour la distance et le parent, et on continue la boucle.

Application : recherche de plus courts chemins dans le métro parisien

Question 22 :Dans la méthode main de la classe MainMetro, ajouter le code permettant de recalculer le plus court chemin des 5 trajets de la question 20, en pénalisant cette fois-ci les changements de ligne.

```
public static void main(String[] args) {
    try{
        Graphe g2 = LireReseau.Lire( nFichier: "Graphe/plan.txt");
        for (int i = 0;i <5;i++){
            String depart = g2.ListeNoeuds().get((int)(Math.floor(Math.random()*285)));
            String arrive = g2.ListeNoeuds().get((int)(Math.floor(Math.random()*285)));
            Valeurs v =  new Valeurs();
            Valeurs v2 =  new Valeurs();
            Valeurs v3 =  new Valeurs();
            BellmanFord bf = new BellmanFord();
            Dijkstra dj = new Dijkstra();


            long datedeb1 = System.nanoTime();
            v = bf.resoudre2(g2,depart);//Utilisation de resoudre2 pour les pénalité de temps
            long datefin1 = System.nanoTime();
            long execBellmanFord = datefin1-datedeb1;
            long datedeb2 = System.nanoTime();
            v2 = dj.resoudre2(g2,depart);//Utilisation de resoudre2 pour les pénalité de temps
            long datefin2 = System.nanoTime();
            long execDijkstra1 = datefin2-datedeb2;
            List<String> c1 = v.calculerChemin(arrive);
            System.out.println("Départ : "+depart+"\nArrivé : "+arrive);
            System.out.println(c1);
            System.out.println("Temps d'execution Dijkstra : "+execDijkstra1+"ns\n" +"Temps d'execution de BerllmanFord : "+execBellmanFord+"ns");

        }
    }catch(IOException e){
        System.out.println("LireReseau Erreur : "+e.getMessage());
    }
}
}
```

Ce main permet d'afficher les chemin le plus court en prenant compte des pénalités.

Application : recherche de plus courts chemins dans le métro parisien

Question 23 :Reprendre les 5 trajets de la question 20 ci-dessus et comparer les chemins précédemment calculés avec ceux fournis par l'application de la RATP. Quelles différences constatez vous le cas échéant ?

Le premier trajet correspond parfaitement à notre résultat malgré une station qui n'apparait plus sur le trajet.

Le deuxième trajet correspond parfaitement à notre résultat malgré une station qui n'apparait plus sur le trajet.

Le troisième trajet correspond parfaitement à notre résultat.

Le quatrième trajet correspond partiellement. Les premiers (généralement les 3 premier) et derniers (généralement les 3 derniers) arrêts sont bon mais le reste ne l'est pas.

Le dernier trajet correspond partiellement. Les premiers (généralement les 3 premier) et derniers (généralement les 3 derniers) arrêts sont bon mais le reste ne l'est pas.