Tolkacheva Anastasia

Schaeffer Johan

**Rapport Algorithmes de parcours de graphes**

**2. Représentation d’un graphe (2h)**

**Question :**

La classe Arc représente une **liaison orientée** entre deux nœuds dans un graphe pondéré, comme une ligne de métro allant d’une station à une autre.

Chaque objet de la classe Arc contient :

* La **destination** du nœud (station d’arrivée) ;
* Le **coût** du trajet (exprimé en minutes ou autre unité de poids) ;
* La **ligne** de métro concernée (utile pour gérer les correspondances).

Cette classe est centrale pour les algorithmes de plus court chemin (Dijkstra, Bellman-Ford), car elle modélise les **chemins possibles entre deux stations**, avec des coûts précis.

**📦 Attributs de la classe**

private String dest;

private double cout;

private String ligne;

* dest : nom du nœud de destination (ex. : "Châtelet").
* cout : coût associé à l’arc (ex. : 12.5 minutes).
* ligne : numéro ou nom de la ligne de métro (ex. : "1", "6"). Peut être null pour les cas simples.

**🏗️ Constructeurs**

**1. Constructeur de base (sans ligne) :**

public Arc(String dest, double cout) {

this.dest = dest;

if (cout < 0) {

throw new IllegalArgumentException("cout < 0");

} else {

this.cout = cout;

}

}

* Sert pour les graphes simples, sans notion de ligne (par exemple un graphe abstrait).
* Vérifie que le coût est **positif ou nul**, car un coût négatif n’a pas de sens dans certains algorithmes comme Dijkstra.

**2. Constructeur avec ligne :**

public Arc(String dest, double cout, String ligne) {

this.dest = dest;

if (cout < 0) {

throw new IllegalArgumentException("cout < 0");

}

this.cout = cout;

this.ligne = ligne;

}

* Ajoute la **ligne de métro** comme information supplémentaire.
* Très utile dans le cas du **réseau du métro parisien**, notamment pour :
  + détecter les **changements de ligne** ;
  + ajouter une **pénalité de correspondance** dans les versions modifiées des algorithmes.

**🧰 Méthodes publiques**

public String getDest() { return dest; }

public double getCout() { return cout; }

public String getLigne() { return ligne; }

* getDest() : retourne la destination de l’arc.
* getCout() : retourne le poids (temps) du trajet.
* getLigne() : retourne la ligne de métro utilisée (null si absente).

Ces méthodes sont utilisées dans toutes les **parcours d’arcs** lors de l’exécution des algorithmes.

**🧠 Pourquoi cette classe est essentielle**

La classe Arc est une **brique de base du graphe orienté**. Chaque algorithme de plus court chemin repose sur :

* les arcs disponibles à partir d’un nœud ;
* leur coût ;
* et parfois leur nature (ligne).

Sans cette structure bien conçue, impossible de :

* parcourir correctement un graphe ;
* appliquer des pénalités de changement de ligne ;
* ou lire efficacement les connexions dans un fichier réseau.

**Question 2 :**

La classe Arcs représente un **ensemble d'arcs sortants d’un même nœud**.  
Dans un graphe orienté, chaque sommet peut être relié à plusieurs autres sommets : on a donc besoin de pouvoir stocker une **liste d'arcs** associés à un sommet donné.

C’est cette classe qui permet cela, en encapsulant une simple **liste d’objets Arc**.

**📦 Attribut de la classe**

private List<Arc> arcs;

* Il s’agit d’une liste Java générique (ArrayList<Arc>) qui contient les arcs sortants (destination + coût + ligne) d’un nœud donné.
* Chaque objet Arc représente une liaison vers une autre station.

**🏗️ Constructeur**

public Arcs(){

this.arcs = new ArrayList<>();

}

* Ce constructeur **initialise une liste vide** d’arcs.
* Chaque nœud du graphe est donc associé à un objet Arcs, ce qui rend possible de construire dynamiquement les connexions du graphe.

**🧰 Méthodes publiques**

**➤ Méthode ajouterArc**

public void ajouterArc(Arc a){

arcs.add(a);

}

* Permet d’**ajouter un nouvel arc** (connexion vers un autre nœud).
* Appelée depuis GrapheListe.ajouterArc(...) à chaque fois qu’une nouvelle liaison est ajoutée entre deux stations.
* Cette méthode est **très souvent utilisée lors de la lecture du fichier** plan-reseau.txt, où chaque ligne de connexion crée un Arc qu’on ajoute ensuite ici.

**➤ Méthode getArcs**

public List<Arc> getArcs(){

return this.arcs;

}

* Retourne **la liste complète des arcs sortants** d’un nœud.
* Elle est indispensable pour tous les parcours de graphe (Dijkstra, Bellman-Ford), car ces algorithmes doivent consulter les arcs pour propager les distances.

**🧠 Pourquoi cette classe est importante**

* Elle sert à **structurer les arcs sortants d’un nœud** proprement.
* Elle évite de mélanger la logique de graphe avec la gestion des listes.
* Elle est indispensable pour maintenir une représentation **cohérente, claire et modulaire** du graphe.

**Question 3 :**

L’interface Graphe a été conçue dans le but de **généraliser la structure d’un graphe** et de **rendre les algorithmes indépendants de l’implémentation** utilisée.

Autrement dit, au lieu d’écrire un algorithme qui dépend de GrapheListe, on l’écrit pour un Graphe en général. Cela permet d'appliquer les mêmes algorithmes (Dijkstra, Bellman-Ford...) à **n'importe quel type de graphe**, tant qu’il implémente l’interface.

**📄 Code de l’interface**

import java.util.\*;

public interface Graphe {

/\*\*

\* Retourne la liste de tous les noeuds du graphe.

\* @return liste des noms des noeuds.

\*/

public List<String> listeNoeuds();

/\*\*

\* Retourne la liste des arcs sortants depuis un noeud donné.

\* @param n le nom du noeud

\* @return liste des arcs partant de ce noeud.

\*/

public List<Arc> suivants(String n);

}

**🔍 Explication des deux méthodes**

**➤ listeNoeuds()**

* **But :** Obtenir tous les sommets présents dans le graphe.
* **Retourne :** Une List<String> contenant tous les noms des sommets (stations de métro dans notre cas).
* **Utilité :** Permet aux algorithmes de parcourir tous les sommets pour initialiser les distances (par exemple dans Dijkstra ou Bellman-Ford).

**➤ suivants(String n)**

* **But :** Obtenir tous les arcs sortants à partir du sommet n.
* **Retourne :** Une List<Arc> représentant toutes les destinations accessibles directement depuis le nœud n, avec les coûts associés.
* **Utilité :** Essentiel pour le cœur des algorithmes, lorsqu'on veut explorer les voisins d’un sommet.

**Question 4 :**

La classe GrapheListe est une **implémentation concrète** de l’interface Graphe. Elle utilise deux structures principales :

* Une liste noeuds contenant les noms de tous les nœuds (sommets),
* Une liste adjacente contenant, pour chaque nœud, une liste d’arcs sortants représentée par un objet Arcs.

Ces deux listes sont **parallèles** : l’indice d’un nœud dans noeuds correspond à l’indice des arcs qui partent de lui dans adjacente.

**🔨 Constructeurs**

**🔹 Constructeur vide :**

public GrapheListe() {

this.noeuds = new ArrayList<>();

this.adjacente = new ArrayList<>();

}

🧠 **Rôle :** Initialise un graphe vide. Ce constructeur est utile lorsqu’on veut ajouter les arcs manuellement ou via un autre programme (comme LireReseau).

**🔹 Constructeur depuis un fichier :**

public GrapheListe(String nom) {

this.noeuds = new ArrayList<>();

this.adjacente = new ArrayList<>();

try {

BufferedReader br = new BufferedReader(new FileReader(nom));

String ligne = br.readLine();

while (ligne != null) {

String[] separation = ligne.trim().split("\t");

String depart = separation[0];

String destination = separation[1];

double cout = Double.parseDouble(separation[2]);

this.ajouterArc(depart, destination, cout);

ligne = br.readLine(); // manquait dans ta version

}

} catch (FileNotFoundException e) {

System.out.println("Le fichier n'est pas trouvé");

} catch (IOException e) {

System.out.println("Erreur de lecture du fichier");

}

}

🧠 **Rôle :** Lit un fichier texte (chaque ligne représente un arc sous forme A B 10) et ajoute dynamiquement les arcs.  
⚠️ **Difficulté :** Il fallait penser à appeler readLine() à chaque itération sinon on crée une boucle infinie.

**🔧 Méthodes principales**

**🔹 listeNoeuds()**

public List<String> listeNoeuds() {

return noeuds;

}

📌 **But :** Retourne tous les sommets du graphe.  
💡 **Utilisé dans** : les algorithmes pour initialiser les distances.

**🔹 suivants(String n)**

public List<Arc> suivants(String n) {

int i = getIndice(n);

if (i == -1) return new ArrayList<>();

return adjacente.get(i).getArcs();

}

📌 **But :** Donne la liste des arcs qui partent du nœud n.  
⚠️ **Difficulté :** Il faut s’assurer que le nœud existe, sinon on retourne une liste vide.

**🔹 getIndice(String nom)**

public int getIndice(String nom) {

return noeuds.indexOf(nom);

}

📌 **But :** Trouver l’indice d’un nœud dans la liste noeuds.  
💡 **Important pour accéder à l’indice correspondant dans adjacente**.

**🔹 ajouterArc(String depart, String destination, double cout)**

public void ajouterArc(String depart, String destination, double cout){

int indiceDepart = getIndice(depart);

if (indiceDepart == -1) {

noeuds.add(depart);

adjacente.add(new Arcs());

indiceDepart = noeuds.size() - 1;

}

if (getIndice(destination) == -1) {

noeuds.add(destination);

adjacente.add(new Arcs());

}

adjacente.get(indiceDepart).ajouterArc(new Arc(destination, cout));

}

📌 **But :** Ajoute un arc dans le graphe. Si les nœuds n'existent pas encore, ils sont créés.  
✅ **Très pratique** pour construire un graphe dynamiquement.  
⚠️ **Difficulté rencontrée :** vérifier que depart et destination n’existent pas déjà pour éviter des doublons dans noeuds.

**🔹 Surcharge ajouterArc(String depart, Arc arc)**

public void ajouterArc(String depart, Arc arc) {

int indiceDepart = getIndice(depart);

if (indiceDepart == -1) {

noeuds.add(depart);

adjacente.add(new Arcs());

indiceDepart = noeuds.size() - 1;

}

if (getIndice(arc.getDest()) == -1) {

noeuds.add(arc.getDest());

adjacente.add(new Arcs());

}

adjacente.get(indiceDepart).ajouterArc(arc);

}

📌 **But :** Ajouter un arc complet, utile quand on connaît déjà le cout et la ligne.  
💡 Utilisé par LireReseau.

**🔹 toString()**

public String toString(){

StringBuilder sb = new StringBuilder();

for (int i = 0; i < noeuds.size(); i++) {

sb.append(noeuds.get(i)).append(" -> ");

for (Arc arc : adjacente.get(i).getArcs()) {

sb.append(arc.getDest()).append("(").append(arc.getCout()).append(") ");

}

sb.append("\n");

}

return sb.toString();

}

📌 **But :** Affiche le graphe de manière lisible.  
💡 **Très utile** pour tester visuellement que le graphe est correct.

**Question 5 :**

**📋 Objectif :**

Avant d’utiliser le graphe sur des données complexes comme le réseau du métro parisien, il est important de **tester sa structure avec un petit exemple simple**.  
Cela permet de :

* Vérifier que l’ajout d’arcs fonctionne correctement.
* Tester si les nœuds sont bien créés automatiquement.
* Contrôler que l'affichage (toString) renvoie un graphe lisible.

**✅ Code de test utilisé**

import java.util.ArrayList;

import java.util.List;

public class Main {

public static void main(String[] args) {

// Création du graphe pour tester

GrapheListe graphe = new GrapheListe();

// Ajout des arcs (on crée les liens entre les nœuds)

graphe.ajouterArc("A", "B", 12);

graphe.ajouterArc("A", "D", 87);

graphe.ajouterArc("B", "E", 11);

graphe.ajouterArc("C", "A", 19);

graphe.ajouterArc("D", "C", 10);

graphe.ajouterArc("D", "B", 23);

graphe.ajouterArc("E", "D", 43);

// Test d'affichage des nœuds

System.out.println("Liste des noeuds :");

List<String> noeuds = graphe.listeNoeuds();

for (String n : noeuds) {

System.out.println("- " + n);

}

// Test des arcs partant de A

System.out.println("\nArcs partant de A :");

List<Arc> arcsA = graphe.suivants("A");

for (Arc a : arcsA) {

System.out.println("→ " + a.getDest() + " (" + a.getCout() + ")");

}

// Test affichage complet du graphe

System.out.println("\nTest Affichage Graphe :");

System.out.println(graphe);

}

}

**🧠 Explication du test**

**✅ Création du graphe**

GrapheListe graphe = new GrapheListe();

➡️ On initialise un graphe vide pour y ajouter des arcs.

**✅ Ajout d’arcs**

graphe.ajouterArc("A", "B", 12);

graphe.ajouterArc("A", "D", 87);

graphe.ajouterArc("B", "E", 11);

graphe.ajouterArc("C", "A", 19);

graphe.ajouterArc("D", "C", 10);

graphe.ajouterArc("D", "B", 23);

graphe.ajouterArc("E", "D", 43);

➡️ On crée un petit graphe dirigé en connectant les sommets avec leurs poids respectifs.  
🧠 **Avantage** : on vérifie ici que ajouterArc() ajoute aussi les nœuds automatiquement.

**✅ Affichage de la liste des nœuds**

List<String> noeuds = graphe.listeNoeuds();

for (String n : noeuds) {

System.out.println("- " + n);

}

➡️ Cela permet de vérifier que tous les nœuds ont été créés.  
✅ On doit voir A, B, D, E, C (même s’ils n'ont pas de successeurs, ils apparaissent).

**✅ Affichage des arcs de A**

List<Arc> arcsA = graphe.suivants("A");

for (Arc a : arcsA) {

System.out.println("→ " + a.getDest() + " (" + a.getCout() + ")");

}

➡️ On teste ici si suivants("A") fonctionne bien et retourne les bons arcs (A → B, A → D).

**✅ Affichage complet du graphe**

System.out.println(graphe);

➡️ Utilise la méthode toString() de GrapheListe pour voir l’ensemble du graphe sous forme :

A -> B(12.0) D(87.0)

B -> E(11.0)

C -> A(19.0)

D -> C(10.0) B(23.0)

E -> D(43.0)

Cela permet de :

* Vérifier que chaque arc est bien relié à sa destination,
* Visualiser rapidement la structure complète du graphe

**Question 6 :**

La méthode toString() de GrapheListe affiche le graphe sous la forme suivante :

A -> B(12) D(87)

B -> E(11)

C -> A(19)

D -> B(23) C(10)

E -> D(43)

Cela permet de vérifier visuellement les arcs de chaque nœud.

**Tests (Question 7) :**

**🎯 Objectif :**

Nous avons écrit une série de **tests unitaires** avec JUnit pour vérifier que la classe GrapheListe fonctionne correctement. Ces tests permettent de :

* S'assurer que la structure du graphe est bien initialisée ;
* Vérifier que les arcs sont correctement ajoutés ;
* Tester que la méthode suivants() retourne les bons arcs ;
* Valider que le graphe se comporte comme attendu même dans des cas limites (nœuds inconnus, doublons...).

**Test 1 : Constructeur vide et cas d’un nœud inexistant**

@Test

public void test1\_Constructeur() {

GrapheListe graphe = new GrapheListe();

List<String> noeuds = graphe.listeNoeuds();

assertNotNull(noeuds, "La liste des noeuds ne doit pas être nulle");

assertEquals(0, noeuds.size(), "La liste des noeuds doit être vide");

List<Arc> arcs = graphe.suivants("inexistant");

assertNotNull(arcs, "La liste des arcs ne doit pas être nulle même pour un nœud inexistant");

assertEquals(0, arcs.size(), "La liste des arcs doit être vide pour un nœud inexistant");

}

🧠 **Pourquoi c’est important** :

* Vérifie que la structure du graphe est bien initialisée ;
* Gère le cas où un utilisateur demande les voisins d’un nœud qui n'existe pas ;
* Évite les erreurs de NullPointerException.

**🔧 Test 2 : Vérification des indices des nœuds**

@Test

public void test2\_GetIndice() {

GrapheListe graphe = new GrapheListe();

graphe.ajouterArc("A", "B", 1.0);

assertEquals(0, graphe.getIndice("A"));

assertEquals(1, graphe.getIndice("B"));

assertEquals(-1, graphe.getIndice("X"), "Le noeud 'X' n'existe pas donc l'indice doit être -1");

}

🧠 **Pourquoi c’est important** :

* Confirme que les nœuds sont bien ajoutés ;
* Teste le bon fonctionnement de la méthode getIndice().

**🔧 Test 3 : Ajout d’arcs et vérification des voisins**

@Test

public void test3\_AjouterArc() {

GrapheListe graphe = new GrapheListe();

graphe.ajouterArc("A", "B", 2.5);

graphe.ajouterArc("A", "C", 1.5);

graphe.ajouterArc("C", "D", 4.0);

List<Arc> arcsA = graphe.suivants("A");

assertEquals(2, arcsA.size(), "'A' devrait avoir deux arcs");

List<Arc> arcsC = graphe.suivants("C");

assertEquals(1, arcsC.size());

assertEquals("D", arcsC.get(0).getDest());

assertEquals(4.0, arcsC.get(0).getCout());

}

🧠 **Pourquoi c’est important** :

* Vérifie que plusieurs arcs peuvent être ajoutés depuis un même nœud ;
* Teste la cohérence des arcs créés : destination et coût.

**🔧 Test 4 : Appel de suivants() sur un nœud inexistant**

@Test

public void test4\_SuivantsAvecNoeudInexistant() {

GrapheListe graphe = new GrapheListe();

List<Arc> arcs = graphe.suivants("Inconnu");

assertNotNull(arcs, "Doit retourner une liste vide (pas null)");

assertEquals(0, arcs.size(), "Doit retourner une liste vide si le noeud n'existe pas");

}

🧠 **Pourquoi c’est important** :

* Vérifie que le graphe est robuste aux erreurs d’entrée ;
* Confirme qu’on ne génère pas d’exception en consultant un nœud absent.

**🔧 Test 5 : Ajout d’arcs en doublon**

@Test

public void test5\_AjoutNoeudsIdentiques() {

GrapheListe graphe = new GrapheListe();

graphe.ajouterArc("A", "B", 1.0);

graphe.ajouterArc("A", "B", 2.0); // ajoute un deuxième arc du même type

List<Arc> arcs = graphe.suivants("A");

assertEquals(2, arcs.size(), "Deux arcs de A vers B doivent être présents");

}

🧠 **Pourquoi c’est important** :

* Vérifie que l’ajout de plusieurs arcs identiques est bien accepté ;
* Utile dans un contexte où plusieurs trajets différents existent entre deux stations (par exemple, deux lignes de métro différentes entre A et B).

**Question 7 :**

Ces tests ont permis de valider **la fiabilité et la cohérence de la structure GrapheListe**, avant d’implémenter les algorithmes de Dijkstra et Bellman-Ford.

**💡 Difficultés rencontrées**

* **Comprendre comment tester des listes internes** (nœuds/arcs), surtout sans accès direct aux structures internes ;
* **Distinguer les cas limites** (nœud inexistant, doublon) pour éviter des bugs silencieux ;
* **Maintenir la cohérence** entre noeuds et adjacente, qui sont deux listes parallèles : tout ajout dans l’une doit être suivi dans l’autre.

**3. Algorithmes de plus court chemin (4h)**

**🎯 Objectif :**

**Trouver le plus court chemin entre un nœud de départ et tous les autres nœuds dans un graphe orienté à poids positifs.**

**📌 Principe général (en 3 étapes) :**

1. **Initialisation :**
   * **Toutes les distances sont initialisées à l’infini sauf pour le nœud de départ dont la distance est 0.**
2. **Choix du nœud minimum :**
   * **À chaque itération, on sélectionne le nœud non traité ayant la plus petite distance actuelle.**
3. **Mise à jour des voisins :**
   * **Pour chaque voisin v de ce nœud, si le chemin via ce nœud est plus court, on met à jour la distance et le parent de v.**

**Question 9 :**

**🎯 Objectif :**

L’algorithme de Bellman-Ford est un algorithme de plus court chemin qui permet de :

* Gérer des **poids positifs et négatifs** ;
* Trouver les **plus courts chemins** depuis un **nœud de départ** vers tous les autres nœuds ;
* Résoudre certains cas où **Dijkstra échoue** (poids négatifs).

**🧱 Structure de la classe BellmanFord**

public class BellmanFord {

public Valeurs resoudre(Graphe g, String depart) { ... }

public Valeurs resoudre2(Graphe g, String depart) { ... }

public double coutChemin(String depart, String destination, Valeurs valeurs) { ... }

}

**🔧 Méthode resoudre(Graphe g, String depart)**

Cette méthode implémente la version **classique** de Bellman-Ford, **sans pénalité**.

public Valeurs resoudre(Graphe g, String depart)

**▶ Étapes du traitement :**

1. **Initialisation** :
   * Tous les nœuds ont une distance infinie (Double.MAX\_VALUE), sauf le point de départ qui a une distance de 0.
2. **Boucle de point fixe** :
   * Tant qu’on améliore les distances, on continue.
   * Pour chaque nœud u, on regarde tous ses arcs (u → v), et si on trouve un chemin plus court, on met à jour :
     + La distance de v,
     + Le parent de v.

**✅ Avantage :**

* Peut traiter des **coûts négatifs**, contrairement à Dijkstra.

**⚠ Inconvénient :**

* **Plus lent**, car il répète l’algorithme jusqu’à stabilisation (point fixe).

Avantage : il gère les coûts négatifs. Inconvénient : il est plus lent que Dijkstra.

**Utilisation réelle (Question 10) :**

Tester le **fonctionnement de Bellman-Ford** sur un graphe **réalisé à la main** avec des nœuds et des arcs, afin de s’assurer que les deux algorithmes :

* donnent des résultats cohérents ;
* retournent les bons chemins les plus courts ;
* réagissent bien à différents types de topologie.

**🧠 Pourquoi ce test est important :**

1. **Validation de la logique interne** :
   * S’assurer que l’algorithmes trouve bien les **plus courts chemins** sur un graphe simple.
   * Vérifier que les **parents** et **valeurs de coût** sont bien mis à jour.
2. **Visualisation des résultats** :
   * Grâce à System.out.println, on peut voir clairement :
     + le coût minimal depuis le point de départ ;
     + le parent de chaque nœud (utile pour reconstruire le chemin).

**Question 11 :**

**🎯 Objectif :**

Dans cette première version de la question 11, nous voulons simplement vérifier que l’algorithme **Bellman-Ford** fonctionne correctement **sur un petit graphe classique**, sans notion de métro ou de changement de ligne.  
C’est une étape **intermédiaire importante** avant d’ajouter les pénalités liées au changement de ligne.

**🧪 Test unitaire de Bellman-Ford (sans pénalité)**

Nous avons écrit un test dans une classe BellmanFordTest pour valider :

* que **les distances minimales** sont correctement calculées ;
* que **les parents** sont bien mis à jour.

**🔧 Graphe utilisé pour le test :**

Voici les arcs du graphe que nous avons créé manuellement dans le test :

A → B (5)

A → C (3)

B → C (2)

B → D (6)

C → D (7)

D → E (4)

E → B (1)

Il s'agit d'un **graphe orienté avec des poids positifs** (mais pas de lignes de métro ici).  
Il contient un **cycle (E → B)** mais sans coût négatif, donc **Bellman-Ford peut le gérer** sans problème.

**🧪 Code du test :**

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.\*;

public class BellmanFordTest {

@Test

public void testBellmanFordSimpleGraphe() {

GrapheListe graphe = new GrapheListe();

// Construction d’un graphe simple avec des coûts

graphe.ajouterArc("A", "B", 5);

graphe.ajouterArc("A", "C", 3);

graphe.ajouterArc("B", "C", 2);

graphe.ajouterArc("B", "D", 6);

graphe.ajouterArc("C", "D", 7);

graphe.ajouterArc("D", "E", 4);

graphe.ajouterArc("E", "B", 1); // cycle sans poids négatif

BellmanFord algo = new BellmanFord();

Valeurs valeurs = algo.resoudre(graphe, "A");

// Vérifie les distances calculées

assertEquals(0.0, valeurs.getValeur("A"), 0.001); // A est le départ

assertEquals(5.0, valeurs.getValeur("B"), 0.001); // A → B

assertEquals(3.0, valeurs.getValeur("C"), 0.001); // A → C

assertEquals(10.0, valeurs.getValeur("D"), 0.001); // A → C → D

assertEquals(14.0, valeurs.getValeur("E"), 0.001); // A → C → D → E

// Vérifie les parents (chemin suivi)

assertNull(valeurs.getParent("A")); // A n’a pas de parent

assertEquals("A", valeurs.getParent("B")); // A → B

assertEquals("A", valeurs.getParent("C")); // A → C

assertEquals("C", valeurs.getParent("D")); // C → D

assertEquals("D", valeurs.getParent("E")); // D → E

}

}

**🔍 Explication du fonctionnement de l’algorithme (sans pénalité) :**

* **Initialisation** : on met toutes les distances à +∞, sauf le départ (A) qui est à 0.
* Ensuite, on **répète les mises à jour** tant que l'on peut améliorer une distance :
  + Par exemple, si on est à C et que C → D coûte 7, alors si L(C) + 7 < L(D), on met à jour.
* Le cycle E → B ne crée pas de boucle infinie car Bellman-Ford s’arrête **dès qu’il n’y a plus d’amélioration**.

**❗ Pourquoi ce test est important :**

* Il permet de **valider le cœur de l’algorithme** de Bellman-Ford.
* C’est une **base stable** avant d’ajouter la logique du changement de ligne (dans resoudre2()).
* Il montre que notre classe Valeurs gère correctement à la fois :
  + les **distances minimales** ;
  + et les **chemins (parents)**.

**✅ Résultat attendu :**

Si l'algorithme est correct, le test doit **passer sans erreur** et prouver que :

* les distances sont bien calculées ;
* le parcours du graphe suit les arcs avec les **meilleurs coûts cumulés**.

**Comparaison des performances (Question 12) :**

**🧠 Principe de la méthode calculerChemin :**

* On part du nœud **destination** ;
* On regarde son **parent** ;
* Puis le parent de son parent… jusqu’à atteindre le **départ** ;
* À chaque étape, on ajoute le nœud courant à une **liste temporaire** ;
* On inverse la liste à la fin (car elle est construite à l’envers).

**✅ Exemple attendu :**

Si on a appliqué l’algorithme en partant du nœud A, et que les parents sont définis ainsi

C → D

D → E

E → B

B → A

alors calculerChemin("C", "A") renverra :  
[A, B, E, D, C]

**🧪 Méthode calculerChemin(String destination, String depart) :**

Voici la méthode incluse dans la classe Valeurs :

public List<String> calculerChemin(String destination, String depart) {

List<String> chemin = new ArrayList<>();

String courant = destination;

while (courant != null) {

chemin.add(courant); // On ajoute le nœud courant au chemin

if (courant.equals(depart)) {

break; // On s’arrête si on est revenu au départ

}

courant = getParent(courant); // On passe au parent

}

// Si on a terminé sans avoir atteint le départ, alors le chemin est invalide

if (courant == null || !courant.equals(depart)) {

return Collections.emptyList(); // Aucun chemin possible

}

Collections.reverse(chemin); // Le chemin est construit à l’envers, on l’inverse

return chemin;

}

**⚠️ Cas à traiter :**

* **Si aucun chemin n’existe (par exemple, le nœud de destination est isolé ou inatteignable), la méthode retourne une liste vide.**

**🧱 Difficultés rencontrées :**

* **Faire attention aux cas où il n’existe aucun parent pour un nœud (null).**
* **Vérifier que l’on ne crée pas un chemin infini ou incorrect si les parents sont mal définis.**
* **Bien inverser le chemin à la fin : c’est un détail essentiel pour qu’il soit dans le bon ordre.**

**Question 13 :**

L’algorithme de Dijkstra permet de trouver les plus courts chemins à partir d’un nœud de départ, en supposant que tous les arcs ont des poids **positifs**.

**Principe :**

1. Initialisation : distance de tous les sommets à l’infini sauf celui de départ qui vaut 0.
2. Sélection du sommet non encore traité ayant la plus petite distance.
3. Mise à jour des voisins si un chemin plus court est trouvé.
4. Répéter jusqu’à ce que tous les sommets aient été traités.

**Implémentation :**

public Valeurs resoudre(Graphe g, String depart) {

Valeurs valeurs = new Valeurs(); // distances et parents

ArrayList<String> Q = new ArrayList<>(); // sommets non encore traités

for (String noeud : g.listeNoeuds()) {

valeurs.setValeur(noeud, Double.MAX\_VALUE);

valeurs.setParent(noeud, null);

Q.add(noeud);

}

valeurs.setValeur(depart, 0); // distance au départ = 0

while (!Q.isEmpty()) {

String u = Q.get(0);

for (String s : Q) {

if (valeurs.getValeur(s) < valeurs.getValeur(u)) u = s;

}

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;

}

**Difficultés rencontrées :**

* Gérer efficacement la sélection du nœud avec la plus petite valeur. Sans PriorityQueue, cela reste en O(n²).
* Bien initialiser tous les parents et valeurs dès le départ pour éviter des NullPointerException.

**Question 14 :**

import java.util.List;  
  
*/\*\*  
 \* Programme principal pour tester l'algorithme de Dijkstra.  
 \* Il crée un graphe par défaut, calcule les plus courts chemins depuis un nœud de départ,  
 \* affiche les distances et parents de chaque nœud, puis affiche un chemin complet.  
 \*/*public class MainDijkstra {  
  
 */\*\*  
 \* Entrée du main  
 \*  
 \* @param args arguments de la ligne de commande (non utilisé)  
 \*/* public static void main(String[] args) {  
 // Création d'un graphe avec des arcs et leurs coûts  
 GrapheListe graphe = new GrapheListe();  
 graphe.ajouterArc("A", "B", 12);  
 graphe.ajouterArc("A", "D", 87);  
 graphe.ajouterArc("B", "E", 11);  
 graphe.ajouterArc("C", "A", 19);  
 graphe.ajouterArc("D", "C", 10);  
 graphe.ajouterArc("D", "B", 23);  
 graphe.ajouterArc("E", "D", 43);  
  
 // Noeud de départ pour Dijkstra  
 String depart = "A";  
  
 // Exécution de l'algorithme de Dijkstra sur le graphe  
 Dijkstra algo = new Dijkstra();  
 Valeurs resultats = algo.resoudre(graphe, depart);  
  
 // Affichage des distances et parents pour chaque noeud du graphe  
 for (String noeud : graphe.listeNoeuds()) {  
 System.*out*.println(noeud + " : distance = " + resultats.getValeur(noeud) + ", parent = " + resultats.getParent(noeud));  
 }  
  
 // Affichage du chemin complet de depart à destination  
 String destination = "D";  
 System.*out*.print("Chemin de " + depart + " à " + destination + " : ");  
 List<String> chemin = resultats.calculerChemin(destination, depart);  
  
 // Affichage pas à pas des noeuds du chemin avec une flèche entre eux  
 for (int i = 0; i < chemin.size(); i++) {  
 System.*out*.print(chemin.get(i));  
 if (i < chemin.size() - 1) {  
 System.*out*.print(" -> ");  
 }  
 }  
 System.*out*.println(); // Retour à la ligne à la fin de l'affichage du chemin  
 }  
}

**Question 15 :**

**🎯 Objectif du programme**

Le but de cette classe MainDijkstra est de :

1. Créer un graphe simple.
2. Exécuter l’algorithme de Dijkstra à partir d’un point de départ.
3. Afficher :
   * La distance minimale entre le départ et chaque nœud.
   * Le parent de chaque nœud dans le chemin optimal.
   * Le chemin complet entre un point de départ et un point d’arrivée.

**🧱 Construction du graphe**

graphe.ajouterArc("A", "B", 12);

graphe.ajouterArc("A", "D", 87);

graphe.ajouterArc("B", "E", 11);

graphe.ajouterArc("C", "A", 19);

graphe.ajouterArc("D", "C", 10);

graphe.ajouterArc("D", "B", 23);

graphe.ajouterArc("E", "D", 43);

➡️ Le graphe simulé contient des chemins directs et indirects. Il modélise des connexions avec différents coûts, ce qui permet de tester le calcul de distances optimales.

**🏁 Départ de l’algorithme**

String depart = "A";

Dijkstra algo = new Dijkstra();

Valeurs resultats = algo.resoudre(graphe, depart);

➡️ On utilise le sommet A comme point de départ, puis on exécute l’algorithme de Dijkstra. Le résultat est stocké dans un objet Valeurs.

**📊 Affichage des résultats (distances + parents)**

for (String noeud : graphe.listeNoeuds()) {

System.out.println(noeud + " : distance = " + resultats.getValeur(noeud) + ", parent = " + resultats.getParent(noeud));

}

➡️ Ce code affiche, pour chaque nœud :

* La **distance minimale** trouvée depuis le départ.
* Le **parent** de ce nœud (c’est-à-dire, le sommet juste avant dans le chemin optimal).

**Exemple de sortie :**

A : distance = 0.0, parent = null

B : distance = 12.0, parent = A

E : distance = 23.0, parent = B

D : distance = 66.0, parent = E

C : distance = 76.0, parent = D

**🧭 Affichage du chemin optimal**

String destination = "D";

List<String> chemin = resultats.calculerChemin(destination, depart);

➡️ Cette ligne calcule le chemin entre le sommet de départ "A" et la destination "D". On utilise pour cela la méthode calculerChemin de la classe Valeurs.

**➡️ Affichage formaté du chemin**

for (int i = 0; i < chemin.size(); i++) {

System.out.print(chemin.get(i));

if (i < chemin.size() - 1) {

System.out.print(" -> ");

}

}

➡️ Résultat attendu dans la console :

Chemin de A à D : A -> B -> E -> D

**⚠️ Cas gérés automatiquement**

* Si le chemin n’existe pas, calculerChemin() retourne une liste vide.
* Si le sommet n’a pas de parent, cela indique qu’il est isolé ou inaccessible.

**🧠 Ce que ce programme nous apprend**

* À utiliser un algorithme de plus court chemin dans un cas concret.
* À construire un graphe dynamiquement avec des coûts personnalisés.
* À interpréter les résultats produits par Dijkstra.
* À reconstruire un chemin complet grâce aux **parents** stockés dans Valeurs.

**🛠️ Difficultés rencontrées**

* Erreurs fréquentes lors de l'inversion des nœuds départ et destination.
* Besoin d'une méthode robuste (calculerChemin) pour reconstruire un chemin valide ou vide si le trajet est impossible.
* Affichage parfois confus si les noms des nœuds sont proches ou si le graphe contient des cycles.

**Question 16 :**

Dans cette question, il s'agit de :

* Lire un fichier texte pour **construire automatiquement un graphe** ;
* Pouvoir ensuite **afficher ou manipuler ce graphe** dans les algorithmes ;
* Comprendre comment ce constructeur fonctionne et pourquoi il est essentiel.

**🧱 Code concerné (constructeur GrapheListe(String nom))**

public GrapheListe(String nom){

this.noeuds = new ArrayList<>();

this.adjacente = new ArrayList<>();

try {

BufferedReader br = new BufferedReader(new FileReader(nom));

String ligne = br.readLine();

while(ligne != null){

String[] separation = ligne.trim().split("\t");

String depart = separation[0];

String destination = separation[1];

double cout = Double.parseDouble(separation[2]);

this.ajouterArc(depart, destination, cout);

ligne = br.readLine(); // ⚠️ À ne pas oublier sinon boucle infinie !

}

} catch (FileNotFoundException e) {

System.out.println("Le fichier n'est pas trouvé");

} catch (IOException e) {

System.out.println("Erreur de lecture du fichier");

}

}

**📝 Explication ligne par ligne**

**1. Initialisation des listes :**

this.noeuds = new ArrayList<>();

this.adjacente = new ArrayList<>();

➡️ On commence par créer deux listes vides :

* noeuds : pour stocker les noms des stations ;
* adjacente : pour stocker leurs arcs sortants.

**2. Lecture du fichier ligne par ligne :**

BufferedReader br = new BufferedReader(new FileReader(nom));

String ligne = br.readLine();

➡️ On lit le fichier ligne par ligne avec BufferedReader.

**3. Traitement de chaque ligne :**

while(ligne != null){

String[] separation = ligne.trim().split("\t");

➡️ Chaque ligne est supposée être au format :

StationDépart StationArrivée Coût

On la découpe avec split("\t") pour extraire :

* depart
* destination
* cout

**4. Ajout de l’arc :**

this.ajouterArc(depart, destination, cout);

➡️ Cette méthode ajoute les deux stations si elles n'existent pas et crée un **arc orienté** entre elles.

⚠️ **Important** : cet ajout n’est pas symétrique ! Il ne crée **qu’un arc dans un seul sens**.

**5. Prochaine ligne :**

ligne = br.readLine();

➡️ Cela permet de lire la prochaine ligne, sinon la boucle serait infinie.

**6. Gestion des erreurs :**

} catch (FileNotFoundException e) {

System.out.println("Le fichier n'est pas trouvé");

} catch (IOException e) {

System.out.println("Erreur de lecture du fichier");

}

➡️ On capture deux erreurs possibles :

* Fichier introuvable ;
* Problème pendant la lecture (encodage, caractères spéciaux, etc).

**🔍 Pourquoi ce constructeur est-il important ?**

✅ Il permet de **charger dynamiquement un graphe depuis un fichier** sans coder manuellement les arcs.

✅ Il facilite les **tests sur des graphes plus grands** que ceux qu'on pourrait écrire à la main.

✅ Il est réutilisable pour tous les algorithmes (Dijkstra, Bellman-Ford, etc.).

**✅ Amélioration possible (facultative)**

Si le graphe est **non orienté**, il peut être utile d’ajouter une ligne supplémentaire pour créer aussi l’arc inverse :

this.ajouterArc(destination, depart, cout); // si le graphe est non orienté

**🧠 Ce que cette question nous a appris**

* **Lire un fichier texte** pour créer une structure de données exploitable.
* **Automatiser la création d’un graphe** à partir d’entrées utilisateur ou d’un fichier du monde réel (ex : plan du métro).
* **Gérer les erreurs**

Haut du formulaire

Bas du formulaire

**Question 17 :**

Ce programme permet de **comparer les performances** des deux algorithmes suivants sur plusieurs graphes :

* **Dijkstra** : efficace pour les graphes sans coûts négatifs.
* **Bellman-Ford** : plus général, accepte les coûts négatifs.

Les comparaisons se font sur :

* ⏱️ **Le temps d'exécution** (en nanosecondes),
* 💰 **Le coût total des chemins trouvés**.

**📂 Structure du programme**

**🧱 Étape 1 : Création de 5 graphes**

GrapheListe[] graphes = new GrapheListe[5];

Chaque graphe est défini à la main avec ajouterArc(...), pour varier les cas :

* **Graphe 1** : chaîne simple A → B → C → D
* **Graphe 2** : avec un cycle
* **Graphe 3** : graphe plus dense avec retour vers A
* **Graphe 4** : deux composants déconnectés (A-B, C-D)
* **Graphe 5** : deux chemins de même coût vers un sommet

➡️ Ces graphes permettent de tester des cas simples, des cycles, des déconnexions et des équivalents.

**📏 Étape 2 : Mesure du temps d’exécution**

Pour chaque algorithme, on utilise :

long debut = System.nanoTime();

// appel à l’algorithme

long fin = System.nanoTime();

long duree = fin - debut;

Les temps sont stockés et additionnés pour obtenir :

* **temps total** (pour 5 graphes)
* **temps moyen**

**📈 Étape 3 : Affichage des résultats**

Pour chaque graphe :

* Le **résultat de Dijkstra et Bellman-Ford** est affiché (distance pour chaque nœud).
* Le **temps d’exécution** est affiché.
* On calcule la **somme des distances** pour chaque algorithme.
* On compare les deux : qui donne un chemin moins coûteux ?

if (sommeDijkstra < sommeBellman) {

System.out.println("→ Dijkstra a été meilleur");

}

**📊 Étape 4 : Résumé global**

À la fin :

* Moyenne des temps d’exécution sur tous les graphes.
* Indication de **quel algorithme est plus rapide** globalement :

if (moyenneDijkstra < moyenneBellman) {

System.out.println("Dijkstra est plus rapide.");

}

**🧠 Intérêt pédagogique**

* Permet de **valider expérimentalement** les propriétés théoriques :
  + Dijkstra est souvent plus rapide (car ne répète pas autant de passes).
  + Bellman-Ford est plus lent, mais plus **robuste** (supporte plus de cas).
* Introduit la **mesure de performances** en Java.
* Utilise des notions importantes comme :
  + Tableaux de graphes,
  + Appels d’algorithmes en boucle,
  + System.nanoTime() pour la précision.

**✅ Résultat obtenu**

Globalement, Bellman-Ford est plus rapide. (niveau temps)

Cela peut être dû :

* À la **taille réduite** des graphes testés,
* Ou à l’**absence d’optimisations** dans l’implémentation de Dijkstra (par exemple, pas d’utilisation de file de priorité).

**📌 Améliorations possibles**

* Ajouter un **mode avec pénalité de changement de ligne** (resoudre2()).
* Ajouter un test avec **coûts négatifs** pour voir que Dijkstra échoue.
* Utiliser des **graphes plus grands**, voire aléatoires.

**Question 18 :**

On a fait un constructeur pour cela

Ce constructeur permet de **créer un objet Arc complet** avec trois paramètres :

public Arc(String dest, double cout, String ligne)

**Paramètres :**

* dest : le **nom du sommet de destination** (chaîne de caractères).  
  ⮕ C’est le sommet vers lequel va l’arc.
* cout : le **coût ou poids de l’arc** (par exemple, temps ou distance).  
  ⮕ Doit être **positif ou nul**.
* ligne : une **information additionnelle** sur la ligne de métro.  
  ⮕ Utile pour les **algorithmes avec pénalité de changement de ligne**.

**Question 19 :**

Cette classe permet de **lire un fichier de données** décrivant :

* Une liste de **stations** (avec un identifiant unique et un nom),
* Une liste de **connexions** (entre deux stations avec un coût et une ligne de métro),  
  et de construire un objet de type GrapheListe.

**🔸 Méthode principale : public static GrapheListe lire(String fichier)**

Cette méthode lit le fichier, analyse chaque ligne, et ajoute les arcs au graphe.

**Signature :**

public static GrapheListe lire(String fichier) throws IOException

* fichier : chemin du fichier à lire.
* throws IOException : signale une erreur d'accès au fichier.

**🧱 Étapes de traitement dans le code**

**1. Initialisation :**

GrapheListe graphe = new GrapheListe();

String[] ids = new String[100000]; // tableau associant id -> nom station

* On crée un graphe vide.
* On crée un tableau ids qui associera l’**id numérique** d’une station à son **nom** (par exemple : ids[3] = "République").

**2. Lecture ligne par ligne :**

BufferedReader lecteur = new BufferedReader(new FileReader(fichier));

Deux drapeaux sont utilisés :

* lireStations pour savoir si on lit des lignes de stations.
* lireConnexions pour savoir si on lit des lignes de connexions.

**3. Détection de la section à lire :**

if (ligne.startsWith("%")) {

if (ligne.toLowerCase().contains("stations")) {

lireStations = true;

lireConnexions = false;

} else if (ligne.toLowerCase().contains("connexions")) {

lireStations = false;

lireConnexions = true;

}

continue;

}

➡️ Les lignes commençant par % sont des **marqueurs de section**, pas des données.

**4. Lecture des stations :**

if (lireStations && !ligne.isEmpty()) {

String[] parts = ligne.split(":");

int id = Integer.parseInt(parts[0]);

String nom = parts[1];

ids[id] = nom;

}

* On lit une station sous forme id:nom:x:y:....
* On enregistre dans le tableau ids la correspondance entre id et nom.

**5. Lecture des connexions :**

if (lireConnexions && !ligne.isEmpty()) {

String[] parts = ligne.split(":");

int id1 = Integer.parseInt(parts[0]);

int id2 = Integer.parseInt(parts[1]);

double cout = Double.parseDouble(parts[2]);

String ligneMetro = parts[3];

String nom1 = ids[id1];

String nom2 = ids[id2];

➡️ Une ligne de connexion est du type id1:id2:temps:ligne.

Si les noms des stations sont bien trouvés dans ids, on ajoute **deux arcs** au graphe (car le métro est **bidirectionnel**).

graphe.ajouterArc(nom1, new Arc(nom2, cout, ligneMetro));

graphe.ajouterArc(nom2, new Arc(nom1, cout, ligneMetro));

**✅ Exemples d'entrées du fichier texte**

% Stations

1:Gare de Lyon:0:0:

2:Bastille:1:0:

% Connexions

1:2:2.5:1

Résultat :

* ids[1] = "Gare de Lyon"
* ids[2] = "Bastille"
* Création d'un arc de "Gare de Lyon" vers "Bastille" (coût 2.5) sur la ligne 1.

**⚠️ Remarques techniques**

* **Pas de Map utilisée** volontairement pour respecter vos consignes (aucune HashMap).
* Le tableau String[] ids est plus rapide mais **nécessite un maximum arbitraire (100000)**.
* Le programme suppose que les identifiants sont **numériques** et **commencent à 0** ou plus.
* Les erreurs sont **ignorées silencieusement** si les noms sont null.

**🧠 Pourquoi cette classe est essentielle ?**

* Elle permet de **traduire un fichier texte** en **structure Java utilisable par les algorithmes**.
* Elle permet de **généraliser l’approche** à d’autres réseaux, en lisant simplement un autre fichier.

**Question 20 :**

Notre programme peut fonctionner avec d’autres réseaux que celui du métro parisien, à condition de fournir un fichier texte dans le bon format. Cela le rend adaptable à différents cas réels de transport ou de graphes.

**Question 21 :**

**🔧 Méthode resoudre2(Graphe g, String depart)**

Cette méthode est une **version modifiée** de Bellman-Ford qui ajoute une **pénalité** de 10 en cas de **changement de ligne** de métro.

public Valeurs resoudre2(Graphe g, String depart)

**▶ Pourquoi une pénalité ?**

* Pour **modéliser un comportement réaliste** : un usager préfère **éviter les correspondances**, même si cela rallonge un peu le trajet.

**▶ Comment est-elle détectée ?**

1. On regarde le **parent** du nœud courant u.
2. On récupère la **ligne du métro** empruntée pour aller du parent à u.
3. On la compare à la **ligne** entre u et v.
4. Si les lignes sont **différentes**, on ajoute +10 au coût de l’ar

**Question 22 :**

**🎯 Objectif de la classe MainMetro**

Cette classe sert à **tester les performances et les chemins** générés par les deux algorithmes sur **le vrai réseau du métro parisien**, défini dans plan-reseau.txt.

Elle affiche :

* Les trajets choisis entre plusieurs stations.
* Le **temps d’exécution** de Dijkstra et Bellman-Ford.
* Le **coût total du chemin** (temps de trajet).
* La **différence avec ou sans pénalité** de changement de ligne.

**🧱 Structure générale du code**

**1. Chargement du graphe**

GrapheListe graphe = LireReseau.lire(fichier);

➡️ Le graphe du métro est reconstruit automatiquement depuis le fichier texte. Il contient :

* Les noms de stations,
* Les connexions (arcs),
* Les lignes de métro.

**2. Lecture des stations**

ArrayList<String[]> stations = lireStations(fichier);

➡️ Cette méthode lit uniquement la partie % Stations du fichier pour récupérer :

* Les identifiants des stations,
* Le nom associé à chaque id.

Cela servira à **afficher les chemins en termes d’IDs**, comme demandé.

**3. Liste des trajets à tester**

List<String[]> trajets = List.of(

new String[]{"Château de Vincennes", "Bérault"},

...

);

➡️ 5 trajets sont prédéfinis pour tester les algorithmes de manière réaliste.

**4. Comparaison sans et avec pénalité**

afficherResultats(graphe, stations, trajets, false);

afficherResultats(graphe, stations, trajets, true);

➡️ Les deux appels permettent de comparer l’impact **d’un changement de ligne** :

* Sans pénalité (chemins théoriquement les plus courts),
* Avec pénalité de 10 (chemins qui évitent les correspondances si possible).

**🔍 Focus sur la méthode afficherResultats(...)**

Cette méthode est le cœur du test.

**Étapes :**

**a) Pour chaque trajet :**

for (String[] trajet : trajets)

**b) Exécution de Bellman-Ford**

Valeurs valBF = (avecPenalite) ? bf.resoudre2(graphe, depart) : bf.resoudre(graphe, depart);

➡️ Selon le booléen avecPenalite, on utilise la version classique ou celle avec **pénalité de correspondance**.

**c) Exécution de Dijkstra**

Même principe :

Valeurs valD = (avecPenalite) ? d.resoudre2(graphe, depart) : d.resoudre(graphe, depart);

**d) Conversion du chemin en liste d’IDs**

for (String nom : cheminD) {

for (String[] station : stations) {

if (station[1].equals(nom)) {

cheminIds.add(station[0]);

}

}

}

➡️ Cette boucle transforme les noms de stations du chemin en IDs lisibles (comme demandé dans le sujet).

**e) Affichage formaté**

System.out.printf("%-25s %-25s %-45s %-20.2f %-20.2f %-15.2f %-15.2f\n",

depart, arrivee, cheminStr, tempsBF, tempsD, coutBF, coutD);

Question 23 :

1. Premier trajet : 1min -> Conforme + identique
2. Deuxième: 22mn -> Elevé par rapport à notre constat avec les algorithmes + passe par plus de changements
3. Troisième: 27mn -> Pareil que pour le deuxième + pareil que le 2
4. Quatrième: 19mn -> Pareil pour le 2e encore
5. Cinquième :8mn -> Conforme niveau temps + mais passe par plus de changements avec nos algos

**4. Conclusion**

Ce projet nous a permis de comprendre la représentation d'un graphe en Java, ainsi que deux algorithmes classiques de recherche de plus court chemin : Dijkstra et Bellman-Ford.

Nous avons pu appliquer ces algorithmes à un cas réel (le métro parisien), tout en analysant leur comportement, leur temps d'exécution et leur pertinence selon les données.

Nous avons également vu l'intérêt d'adapter un algorithme aux contraintes du problème, ici le changement de ligne.

Ce projet nous a appris à modéliser un problème complexe, à le décomposer en classes et méthodes Java, et à analyser les résultats pour en tirer des conclusions.