IUT Nancy Charlemagne

Rapport SAE

Exploitation d'un problème algorithmique

Auteurs :
Schaeffer Johan
Tolkacheva Anastasia

Table des matières

1 Présentation de la SAE **2**

2 Présentation des classes **2**
2.1 Classe Arc 2
2.2 Classe Arcs 3
2.3 Interface Graphe 4
2.4 Classe `GrapheListe` 5
2.5 Classe `BellmanFord` 7
2.5.1 Algorithme de Point Fixe 8
2.6 Classe `Dijkstra` 10
2.7 Classe `Valeurs` 12
2.8 Classe `LireReseau` 14

3 Classes de test **15**
3.1 Classe `BellmanFordTest` 15

Classes de test **15**
3.2 Classe `BellmanFordTest` 15
3.3 Classe `DijkstraTest` 16
3.4 Classe `GrapheListeTest` 17

4 Main **18**
4.1 Classe `Classe Main` 18
4.2 Classe `MainComparaison` 19
4.3 Classe `MainDijkstra` 20
4.4 Classe `MainMetro` 20

5 Analyse critique et retour d'expérience **22**
5.1 Difficultés rencontrées 22
5.1.1 Comprendre la structure du projet 22
5.1.2 Manipuler les `ArrayList` 22
5.1.3 Lire un fichier et construire dynamiquement un graphe 22
5.1.4 Éviter les erreurs courantes 23
5.1.5 Écrire et comprendre les tests 23
5.2 Points négatifs de la SAE 23
5.3 Points positifs de la SAE 23
5.4 Ce que j'ai appris grâce à cette SAE 24

6 Conclusion **24**

1 Présentation de la SAE

Cette SAE porte sur un graphe orienté, c'est-à-dire un ensemble de points (appelés nœuds) reliés par des liens (appelés arcs) qui ont un coût positif. Le but est de trouver le chemin le plus court entre deux nœuds, c'est-à-dire celui dont la somme des coûts des arcs est la plus faible.

Ce problème est utile dans de nombreux domaines, par exemple pour un GPS qui calcule le trajet le plus rapide ou le plus court entre deux villes.

2 Présentation des classes

2.1 Classe Arc

La classe `Arc` représente un lien dans un graphe entre un nœud parent et un nœud destination. Chaque arc a un coût associé, qui correspond au "poids" de ce lien, et peut aussi être associé à une ligne de métro ou un nom de ligne.

Attributs

- `dest` : une chaîne de caractères indiquant la destination de l'arc.
- `cout` : un nombre décimal positif représentant le coût de l'arc.
- `ligne` : une chaîne de caractères indiquant la ligne associée à l'arc (optionnelle).

Constructeurs Deux constructeurs sont fournis :

- Un premier constructeur qui initialise un arc avec une destination et un coût. Il vérifie que le coût est bien positif.
- Un second constructeur qui initialise un arc avec une destination, un coût et une ligne (qui peut être `null` si non précisée).

Constructeur Arc avec destination et coût

```
public Arc(String dest, double cout) {
    this.dest = dest;
    if (cout < 0) {
        throw new IllegalArgumentException("cout < 0");
    }
    else this.cout = cout;
}
```

Constructeur Arc avec destination

```
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;
}
```

Méthodes principales La classe fournit aussi des méthodes pour accéder aux attributs :

- `getDest()` retourne la destination de l'arc.
- `getCout()` retourne le coût de l'arc.
- `getLigne()` retourne la ligne associée à l'arc.

Exemple d'un getter

```
public String getDest() {
    return dest;
}
```

Rôle de la classe Cette classe permet de modéliser précisément les arcs d'un graphe orienté pondéré, en stockant les informations essentielles comme la destination, le coût, et éventuellement la ligne de métro ou autre information contextuelle. Elle est indispensable pour représenter le graphe sur lequel s'appuient les algorithmes de recherche du plus court chemin.

2.2 Classe Arcs

La classe `Arcs` sert à gérer une collection d'objets `Arc`. Elle permet de regrouper plusieurs arcs dans une liste et de manipuler cette liste facilement.

Attributs

- `arcs` : une liste dynamique (ArrayList) qui contient les objets `Arc`.

Constructeur Le constructeur initialise la liste `arcs` vide, prête à accueillir des arcs.

Constructeur de la classe Arcs

```
public Arcs() {
    this.arcs = new ArrayList<>();
}
```

Méthodes principales

- `ajouterArc(Arc a)` : ajoute un arc passé en paramètre à la liste des arcs.
- `getArcs()` : retourne la liste complète des arcs stockés.

Méthode pour ajouter un arc

```
public void ajouterArc(Arc a) {
    arcs.add(a);
}
```

Getter pour la liste des arcs

```
public List<Arc> getArcs() {
    return this.arcs;
}
```

Rôle de la classe Cette classe est utile pour organiser et manipuler plusieurs arcs liés à un même nœud dans un graphe. Elle sert à stocker les arcs sortants d'un nœud et facilite leur gestion lors de l'exécution des algorithmes de plus court chemin. Avec cette classe, on peut facilement ajouter, récupérer ou parcourir tous les arcs connectés à un nœud.

2.3 Interface Graphe

L'interface `Graphe` définit un contrat pour représenter un graphe orienté. Elle décrit les méthodes essentielles que toute classe implémentant un graphe doit fournir.

Méthodes

- `listeNoeuds()` : cette méthode retourne la liste des nœuds du graphe sous forme de chaînes de caractères. Chaque élément représente un nœud identifié par son nom.
- `suivants(String n)` : cette méthode prend en paramètre un nom de nœud `n` et retourne la liste des arcs sortants de ce nœud, c'est-à-dire tous les arcs dont la source est `n`. Cela permet de connaître les voisins directs de `n` avec les coûts associés.

Définition de l'interface Graphe

```
import java.util.*;

public interface Graphe {
    public List<String> listeNoeuds();
    public List<Arc> suivants(String n);
}
```

Rôle de l'interface Cette interface est fondamentale car elle définit les opérations de base nécessaires pour manipuler un graphe orienté pondéré. Elle sert de base à différentes implémentations concrètes qui pourront stocker le graphe sous différentes formes (listes d'adjacence, matrices, etc.) tout en garantissant que ces méthodes sont disponibles pour les algorithmes.

En séparant la définition (interface) de l'implémentation, on facilite la modularité et la maintenance du code.

2.4 Classe `GrapheListe`

La classe `GrapheListe` représente un graphe orienté pondéré par une structure de listes d'adjacence. Chaque nœud est une chaîne de caractères et possède une liste d'arcs vers ses nœuds adjacents.

Attributs

- `noeuds` : liste des noms de nœuds (`ArrayList<String>`).
- `adjacente` : liste des listes d'arcs sortants (`ArrayList<Arcs>`).

Constructeurs

- Constructeur par défaut initialisant des listes vides.

Constructeur par défaut

```
public GrapheListe() {
    this.noeuds = new ArrayList<>();
    this.adjacente = new ArrayList<>();
}
```

- Constructeur lisant un fichier ligne par ligne (format : `départ destination coût`) pour construire le graphe.

Méthodes principales

- `listeNoeuds()` : retourne la liste des nœuds.

— `suivants(String n)` : retourne la liste des arcs sortants du nœud `n`.
— `getIndice(String nom)` : retourne l'indice d'un nœud ou `-1` si absent.
— `ajouterArc(String depart, String destination, double cout)` : ajoute un arc, crée les nœuds si besoin.

Ajouter un arc

```
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));
}
```

Affichage La méthode `toString()` affiche le graphe sous forme textuelle listant chaque nœud et ses arcs adjacents avec les coûts.

Représentation textuelle du graphe

```
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();
}
```

Résumé Cette classe facilite la construction et la manipulation d'un graphe orienté pondéré par listes d'adjacence, notamment en permettant la lecture

depuis un fichier, l'ajout dynamique d'arcs, et la récupération des arcs sortants d'un nœud donné.

2.5 Classe `BellmanFord`

La classe `BellmanFord` implémente l'algorithme de Bellman-Ford pour trouver les plus courts chemins depuis un nœud de départ vers tous les autres nœuds d'un graphe, même en présence d'arcs à coût négatif.

Méthodes principales

- `resoudre(Graphe g, String depart)` : calcule les distances minimales et parents sans pénalité.
- `resoudre2(Graphe g, String depart)` : version avec pénalité de 10 lors d'un changement de ligne.
- `coutChemin(String depart, String destination, Valeurs valeurs)` : retourne le coût total du chemin calculé ou `Double.MAX_VALUE` si aucun chemin.

2.5.1 Algorithme de Point Fixe

La classe `BellmanFord` utilise un algorithme de point fixe pour calculer les plus courts chemins dans un graphe pondéré, même en présence d'arcs à coût négatif.

Principe L'algorithme initialise toutes les distances à `+\inf` sauf celle du nœud de départ, puis met à jour les distances en relâchant les arêtes tant qu'au moins une modification a lieu. Ce processus converge lorsque plus aucun raccourcissement de chemin n'est possible.

```
fonction pointFixe(Graphe G, Noeud depart) {
    Liste<Noeud> N <- G.getNoeuds();
    Noeud p <- tete(N);
    pour (int i = 1; i <= finListe(N, p); i++) {
        Noeud X <- N[i];
        L(X) <- +   ;
        parent(X) <- null;
        p <- suc(N, p);
    }
    L(depart) <- 0;
    bool modif <- vrai;
    tant que (modif == vrai) {
        modif <- faux;
        pour (int i = 1; i <= finListe(N, p); i++) {
            Noeud X <- N[i];
            p <- suc(N, p);
            pour (int j = 1; j <= taille(voisins(X)); j
               ++) {
                Arc arc <- voisins(X)[j];
                Noeud dest <- arc.destination;
                int cout <- arc.cout;
                si (L(X) + cout < L(dest)) {
                    L(dest) <- L(X) + cout;
                    parent(dest) <- X;
                    modif <- vrai;
                }
            }
        }
    }
}
```

Lexique du pseudo-code utilisé

```
Lexique
G : Graphe, le graphe ou on veut trouver le point fixe

N : Liste(N ud), la liste des n uds du graphe

L(X) : reel, distance minimale depuis le d part
    j u s q u au  n ud X

parent(X) : N ud, le n ud pr c dent sur le plus court
    chemin vers X

voisins(X) : Liste(Arc), liste des arcs partant de X (
   chaque arc est un couple destination + co t)

d part : N ud, n ud de d part
```

Extrait d'initialisation et boucle principale

```
Valeurs valeurs = new Valeurs();
for (String noeud : g.listeNoeuds()) {
    valeurs.setValeur(noeud, noeud.equals(depart) ? 0 :
       Double.MAX_VALUE);
    valeurs.setParent(noeud, null);
}

boolean changement = true;
while (changement) {
    changement = false;
    for (String noeud : g.listeNoeuds()) {
        double valeurNoeud = valeurs.getValeur(noeud);
        if (valeurNoeud == Double.MAX_VALUE) continue;
        for (Arc arc : g.suivants(noeud)) {
            String voisin = arc.getDest();
            double nouveauCout = valeurNoeud + arc.
               getCout();
            if (nouveauCout < valeurs.getValeur(voisin))
               {
                valeurs.setValeur(voisin, nouveauCout);
                valeurs.setParent(voisin, noeud);
                changement = true;
            }
        }
    }
}
```

Gestion de la pénalité lors d'un changement de ligne Dans la méthode `resoudre2`, une pénalité de 10 est ajoutée au coût lorsqu'un arc utilise une ligne différente de celle de l'arc précédent.

Résumé Cette classe permet de résoudre efficacement les plus courts chemins sur des graphes avec ou sans coûts négatifs, en intégrant une logique métier spécifique (pénalité au changement de ligne). Elle s'appuie sur un objet `Valeurs` pour stocker les distances et parents.

2.6 Classe `Dijkstra`

La classe `Dijkstra` implémente l'algorithme de Dijkstra pour calculer les plus courts chemins depuis un nœud de départ vers tous les autres nœuds

d'un graphe à poids positifs.

Méthodes principales

- `resoudre(Graphe g, String depart)` : calcule les distances minimales sans pénalité.
- `resoudre2(Graphe g, String depart)` : version avec pénalité de 10 en cas de changement de ligne.
- `coutChemin(String depart, String destination, Valeurs valeurs)` : retourne le coût total du chemin ou `Double.MAX_VALUE` si aucun chemin.

Initialisation et sélection du noeud avec la plus petite valeur

```
Valeurs valeurs = new Valeurs();
ArrayList<String> Q = new ArrayList<>();
for (String noeud : g.listeNoeuds()) {
    valeurs.setValeur(noeud, Double.MAX_VALUE);
    valeurs.setParent(noeud, null);
    Q.add(noeud);
}
valeurs.setValeur(depart, 0);

while (!Q.isEmpty()) {
    String u = Q.get(0);
    for (int i = 1; i < Q.size(); i++) {
        String c = Q.get(i);
        if (valeurs.getValeur(c) < valeurs.getValeur(u))
           {
            u = c;
        }
    }
    Q.remove(u);
}
```

Gestion de la pénalité en cas de changement de ligne Dans la méthode `resoudre2`, on ajoute 10 au coût d'un arc si la ligne du trajet change par rapport à l'arc précédent :

Ajout de pénalité pour changement de ligne

```
double cout = arc.getCout();
String parent = valeurs.getParent(u);
if (parent != null) {
    for (Arc arcVersU : g.suivants(parent)) {
        if (arcVersU.getDest().equals(u)) {
            String ligneAvant = arcVersU.getLigne();
            String ligneActuelle = arc.getLigne();
            if (ligneAvant != null && ligneActuelle !=
               null && !ligneAvant.equals(ligneActuelle))
                {
                cout += 10;
            }
            break;
        }
    }
}
```

Résumé La classe `Dijkstra` calcule efficacement les plus courts chemins dans un graphe avec arcs à poids positifs, en tenant compte optionnellement d'une pénalité métier liée au changement de ligne.

2.7 Classe `Valeurs`

Attributs

- `private Map<String, String> parents`
  Map associant chaque noeud à son parent.

Méthodes

- `public String getParent(String noeud)`
  Retourne le parent du noeud donné, ou `null` si aucun.
- `public void setParent(String noeud, String parent)`
  Associe un parent à un noeud.
- `public boolean contientNoeud(String noeud)`
  Indique si le noeud est présent dans la structure.

— `public Set<String> getNoeuds()`
Retourne l'ensemble des noeuds.
— `public List<String> calculerChemin(String destination, String depart)`

Calcule le chemin du départ vers la destination en suivant les parents.

```
public List<String> calculerChemin(String destination,
   String depart) {
    List<String> chemin = new ArrayList<>();

    String courant = destination;
    while (courant != null) {
        chemin.add(courant);
        if (courant.equals(depart)) {
            break;  // on a atteint le d part
        }
        courant = getParent(courant);
    }

    // Si on n'a pas atteint le d part, pas de chemin
    if (courant == null || !courant.equals(depart)) {
        return Collections.emptyList();
    }

    Collections.reverse(chemin); // inverser pour d part
            destination
    return chemin;
}
```

2.8 Classe `LireReseau`

Lit un fichier décrivant un réseau (stations + connexions) et construit un `GrapheListe`.

Lecture des stations

```
if (ligne.startsWith("%stations")) {
    stations = true; connexions = false; continue;
}
if (stations && !ligne.isEmpty()) {
    String[] parts = ligne.split(":");
    ids[Integer.parseInt(parts[0])] = parts[1];
}
```

Lecture des connexions

```
if (ligne.startsWith("%connexions")) {
    stations = false; connexions = true; continue;
}
if (connexions && !ligne.isEmpty()) {
    String[] parts = ligne.split(":");
    int id1 = Integer.parseInt(parts[0]), id2 = Integer.
       parseInt(parts[1]);
    double cout = Double.parseDouble(parts[2]);
    String ligneMetro = parts[3];
    graphe.ajouterArc(ids[id1], new Arc(ids[id2], cout,
       ligneMetro));
    graphe.ajouterArc(ids[id2], new Arc(ids[id1], cout,
       ligneMetro));
}
```

Résumé Cette classe facilite la lecture d'un fichier texte formaté en sections stations et connexions pour construire un graphe pondéré représentant un réseau de transport, en ajoutant des arcs bidirectionnels avec leurs coûts et lignes associées.

3 Classes de test

3.1 Classe `BellmanFordTest`

Test un cas simple d'algorithme de Bellman-Ford sur un graphe donné.

Extrait du test principal Test du point fixe

```
graphe.ajouterArc("A", "B", 12);
graphe.ajouterArc("A", "D", 87);
// arcs suppl mentaires ...
BellmanFord algo = new BellmanFord();
Valeurs resultats = algo.resoudre(graphe, "A");
assertEquals(0.0, resultats.getValeur("A"), 0.0001);
assertEquals(12.0, resultats.getValeur("B"), 0.0001);
assertNull(resultats.getParent("A"));
assertEquals("A", resultats.getParent("B"));
```

Résumé Ce test vérifie que l'algorithme Bellman-Ford calcule correctement les distances minimales et les parents dans un graphe pondéré, en s'assurant que les résultats correspondent aux valeurs attendues.

Classes de test

3.2 Classe `BellmanFordTest`

Cette classe contient un test pour vérifier le bon fonctionnement de l'algorithme de Bellman-Ford.

Résumé

— Vérifie les distances et parents d'un graphe simple orienté et pondéré.

Extrait du test Bellman-Ford

```
graphe.ajouterArc("A", "B", 12);
graphe.ajouterArc("B", "E", 11);
Valeurs resultats = algo.resoudre(graphe, "A");
assertEquals(12.0, resultats.getValeur("B"), 0.0001);
assertEquals("A", resultats.getParent("B"));
```

3.3 Classe `DijkstraTest`

Cette classe contient plusieurs tests unitaires pour l'algorithme de Dijkstra.

Résumé des tests

- `testDijkstra` : cas classique avec distances et parents.
- `testSommetInaccessible` : vérifie les sommets isolés.
- `testCycle` : graphe cyclique A → B → C → A.
- `testCheminsEquivalents` : plusieurs chemins avec même coût.

Test de base sur un graphe connecté

```
graphe.ajouterArc("A", "B", 12);
graphe.ajouterArc("A", "D", 87);
Valeurs resultats = algo.resoudre(graphe, "A");
assertEquals(12.0, resultats.getValeur("B"), 0.0001);
assertEquals("A", resultats.getParent("B"));
```

Test de sommets inaccessibles

```
graphe.ajouterArc("A", "B", 5);
graphe.ajouterArc("D", "E", 2); // D et E isol s
Valeurs r = algo.resoudre(graphe, "A");
assertEquals(Double.MAX_VALUE, r.getValeur("D"));
assertNull(r.getParent("D"));
```

Test de graphe cyclique

```
graphe.ajouterArc("A", "B", 1);
graphe.ajouterArc("B", "C", 1);
graphe.ajouterArc("C", "A", 1);
Valeurs r = algo.resoudre(graphe, "A");
assertEquals(2.0, r.getValeur("C"));
```

Chemins de même coût

```
graphe.ajouterArc("A", "B", 5);
graphe.ajouterArc("A", "C", 5);
graphe.ajouterArc("B", "D", 5);
graphe.ajouterArc("C", "D", 5);
Valeurs r = algo.resoudre(graphe, "A");
String parentD = r.getParent("D");
assertTrue(parentD.equals("B") || parentD.equals("C"));
```

3.4 Classe `GrapheListeTest`

Cette classe teste les méthodes de la classe `GrapheListe`, qui représente un graphe orienté pondéré sous forme de listes d'adjacence.

Résumé des tests

- `test1_Constructeur` : vérifie l'initialisation des structures internes.
- `test2_GetIndice` : teste l'assignation d'indice aux nœuds.
- `test3_AjouterArc` : ajoute plusieurs arcs et teste leur enregistrement.
- `test4_SuivantsAvecNoeudInexistant` : comportement attendu pour un nœud absent.
- `test5_AjoutNoeudsIdentiques` : vérifie qu'on peut ajouter plusieurs arcs identiques.

Vérification du constructeur et des listes vides

```
GrapheListe graphe = new GrapheListe();
List<String> noeuds = graphe.listeNoeuds();
assertNotNull(noeuds);
assertEquals(0, noeuds.size());
assertEquals(0, graphe.suivants("inexistant").size());
```

Ajout d'arcs et vérification des suivants

```
graphe.ajouterArc("A", "B", 2.5);
graphe.ajouterArc("A", "C", 1.5);
List<Arc> arcsA = graphe.suivants("A");
assertEquals(2, arcsA.size());
```

Ajout de plusieurs arcs identiques

```
graphe.ajouterArc("A", "B", 1.0);
graphe.ajouterArc("A", "B", 2.0); // deux arcs vers le
    m me sommet
List<Arc> arcs = graphe.suivants("A");
assertEquals(2, arcs.size());
```

4 Main

4.1 Classe `Classe Main`

Cette classe sert de point d'entrée pour exécuter un programme de démonstration du graphe. Elle permet de visualiser les nœuds, les arcs, l'affichage du graphe, et les résultats de l'algorithme de Bellman-Ford.

Fonctionnalités testées dans `Main` :

- Création d'un graphe via `GrapheListe`.
- Ajout manuel d'arcs avec des coûts.
- Affichage de la liste des nœuds et des arcs d'un nœud donné.
- Affichage global du graphe.
- Résolution du plus court chemin depuis un sommet avec Bellman-Ford.

Création et affichage des nœuds

```
GrapheListe graphe = new GrapheListe();
graphe.ajouterArc("A", "B", 12);
graphe.ajouterArc("A", "D", 87);
graphe.ajouterArc("B", "E", 11);
// ...
System.out.println("Liste des noeuds :");
for (String noeud : graphe.listeNoeuds()) {
    System.out.println("- " + noeud);
}
```

Résolution Bellman-Ford

```
BellmanFord algo = new BellmanFord();
Valeurs resultats = algo.resoudre(graphe, "A");

for (String noeud : graphe.listeNoeuds()) {
    double dist = resultats.getValeur(noeud);
    String parent = resultats.getParent(noeud);
    System.out.println(noeud + " : distance = " + dist +
       ", parent = " + parent);
}
```

4.2 Classe `MainComparaison`

Cette classe permet de comparer les performances des algorithmes de Dijkstra et de Bellman-Ford sur plusieurs graphes définis manuellement. Elle mesure les temps d'exécution et les coûts totaux pour chaque algorithme sur chaque graphe.

Fonctionnalités de `MainComparaison` :

- Création de 5 graphes différents à la main.
- Résolution avec Dijkstra et Bellman-Ford pour chaque graphe.
- Mesure du temps d'exécution de chaque algorithme.
- Comparaison des distances totales trouvées.
- Affichage d'un résumé global des performances.

Initialisation des graphes

```
graphes[0] = new GrapheListe();
graphes[0].ajouterArc("A", "B", 1);
graphes[0].ajouterArc("B", "C", 2);
graphes[0].ajouterArc("C", "D", 3);
...
```

Comparaison des algorithmes

```
long debutDijkstra = System.nanoTime();
Valeurs resDijkstra = dijkstra.resoudre(graphe, depart);
long dureeDijkstra = System.nanoTime() - debutDijkstra;

long debutBellman = System.nanoTime();
Valeurs resBellman = bellmanFord.resoudre(graphe, depart)
    ;
long dureeBellman = System.nanoTime() - debutBellman;
```

Résumé global

```
System.out.println("Temps total Dijkstra      : " +
    totalTempsDijkstra + " ns");
System.out.println("Temps total Bellman-Ford : " +
    totalTempsBellman + " ns");

if (moyenneDijkstra < moyenneBellman) {
     System.out.println("    Dijkstra est globalement plus
         rapide.");
}
```

4.3 Classe `MainDijkstra`

Cette classe permet de tester l'algorithme de Dijkstra sur un graphe orienté avec des coûts prédéfinis. Elle affiche pour chaque nœud du graphe :

— la distance minimale depuis un nœud de départ ;
— le parent (nœud précédent sur le plus court chemin) ;
— et le chemin complet vers un nœud de destination.

Création du graphe pour Dijkstra

```
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);
```

Résolution et affichage des résultats

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

for (String noeud : graphe.listeNoeuds()) {
    System.out.println(noeud + " : distance = "
        + resultats.getValeur(noeud) + ", parent = "
        + resultats.getParent(noeud));
}
```

Affichage du chemin le plus court

```
List<String> chemin = resultats.calculerChemin("D", "A");
for (int i = 0; i < chemin.size(); i++) {
    System.out.print(chemin.get(i));
    if (i < chemin.size() - 1) {
        System.out.print(" -> ");
    }
}
```

Cette démonstration simple met en évidence la capacité de l'algorithme de Dijkstra à déterminer les plus courts chemins dans un graphe pondéré orienté.

4.4 Classe `MainMetro`

Cette classe compare les performances et les résultats des algorithmes de Dijkstra et Bellman-Ford sur cinq trajets du métro parisien, en deux

configurations :

— sans pénalité de changement de ligne ;

— avec pénalité appliquée pour chaque changement de ligne.

Le graphe est construit à partir d'un fichier contenant les stations et les connexions. Voici les trajets testés :

— Château de Vincennes → Bérault

— Porte Dauphine → Nation

— Pont de Neuilly → République

— Concorde → Porte d'Orléans

— Place d'Italie → Pont-Marie

Liste des trajets testés

```
List<String[]> trajets = List.of(
    new String[]{"Ch teau de Vincennes", "B rault"},
    new String[]{"Porte Dauphine", "Nation"},
    new String[]{"Pont de Neuilly", "R publique"},
    new String[]{"Concorde", "Porte d'Orl ans"},
    new String[]{"Place d'Italie", "Pont-Marie"}
);
```

Pour chaque trajet, les deux algorithmes sont exécutés avec ou sans pénalité. Les performances (temps d'exécution en millisecondes) et les coûts sont affichés sous forme tabulaire, en plus du chemin calculé (identifiants de stations).

Mesure des performances pour un trajet

```
long debutBF = System.nanoTime();
Valeurs valBF = avecPenalite ?
    new BellmanFord().resoudre2(graphe, depart) :
    new BellmanFord().resoudre(graphe, depart);
long finBF = System.nanoTime();
double tempsBF = (finBF - debutBF) / 1_000_000.0;
```

Conversion des noms de stations en identifiants

```
for (String nom : cheminD) {
    for (String[] station : stations) {
        if (station[1].equals(nom)) {
            cheminIds.add(station[0]);
            break;
        }
    }
}
```

Enfin, les résultats sont affichés sous forme de tableau formaté, ce qui permet une comparaison directe des deux algorithmes dans les deux contextes (avec ou sans pénalité).

Affichage formaté des résultats

```
System.out.printf("%-25s %-25s %-45s %-20.2f %-20.2f
   %-15.2f %-15.2f\n",
    depart, arrivee, cheminStr, tempsBF, tempsD, coutBF,
       coutD);
```

Cette classe constitue donc un outil essentiel pour analyser les différences de comportement entre Dijkstra et Bellman-Ford dans un contexte réaliste (réseau de transport en commun).

5 Analyse critique et retour d'expérience

5.1 Difficultés rencontrées

5.1.1 Comprendre la structure du projet

Au début de la SAE, il n'était pas évident de savoir comment structurer correctement le code. Il fallait créer plusieurs classes différentes (`Arc`, `Arcs`, `GrapheListe`, etc.), et comprendre comment elles allaient interagir entre elles. Il a aussi fallu s'assurer que chaque classe avait un rôle clair, sans mélanger les responsabilités.

5.1.2 Manipuler les `ArrayList`

Une partie importante du projet était la gestion des `ArrayList`, notamment dans la classe `GrapheListe`. Les listes `noeuds` et `adjacente` devaient toujours être bien synchronisées : si un nœud était ajouté sans sa liste d'arcs correspondante, ou inversement, cela causait des erreurs difficiles à repérer. Il fallait bien vérifier les indices et l'ordre des éléments.

5.1.3 Lire un fichier et construire dynamiquement un graphe

Le fichier texte utilisé pour représenter le réseau de stations était parfois délicat à lire. Il fallait distinguer les deux sections (%stations et %connexions), gérer les séparateurs, et convertir des identifiants numériques en noms de stations. Une erreur dans une seule ligne pouvait empêcher tout le fichier d'être lu correctement, ce qui compliquait le travail.

5.1.4 Éviter les erreurs courantes

Certaines erreurs comme les `NullPointerException` ou les indices hors limites apparaissaient facilement si on oubliait d'initialiser une liste ou si on accédait à un nœud inexistant. Cela a demandé de la rigueur dans les vérifications à chaque étape du programme.

5.1.5 Écrire et comprendre les tests

Rédiger des tests pertinents pour vérifier le bon fonctionnement des classes et des algorithmes n'était pas toujours évident. Il fallait bien choisir les cas à tester, et comprendre pourquoi certains résultats étaient attendus. Les tests ont quand même beaucoup aidé à détecter des erreurs.

5.2 Points négatifs de la SAE

— Le format du fichier d'entrée était strict : la moindre erreur de tabulation ou de structure faisait planter la lecture.
— Il fallait faire attention à ne pas introduire de doublons dans les `ArrayList`, ce qui n'était pas forcément bloqué par le programme.
— La logique autour du changement de ligne pour la pénalité pouvait être un peu difficile à tester sans visualisation.
— Peu d'exemples concrets étaient fournis pour démarrer, ce qui rendait les premières étapes un peu longues à comprendre.

5.3 Points positifs de la SAE

— Le projet nous a permis de bien comprendre la représentation d'un graphe en programmation orientée objet.
— Nous avons pu voir comment appliquer concrètement des algorithmes classiques comme Dijkstra ou Bellman-Ford sur un cas réel (le métro).
— Le fait de lire un vrai fichier et de construire automatiquement le graphe a rendu le projet plus concret et motivant.
— Les tests que nous avons mis en place nous ont aidés à valider notre code au fur et à mesure, et à corriger des erreurs rapidement.
— Le projet était progressif : chaque étape (création des classes, lecture de fichier, algorithmes, tests) préparait logiquement la suivante.

5.4 Ce que j'ai appris grâce à cette SAE

Sur le plan technique

- Mieux utiliser les `ArrayList` et gérer correctement les indices.
- Construire un graphe dynamique en Java, à partir d'un fichier externe.
- Implémenter des algorithmes de `Dijkstra` et `Bellman-Ford` et comprendre leur fonctionnement détaillé.
- Récupérer un chemin à partir des parents et le reconstruire dans le bon ordre.
- Gérer les erreurs (comme un nœud inexistant ou un coût négatif) de façon propre dans le code.

Sur le plan méthodologique

- Écrire du code plus structuré et organisé, en séparant les responsabilités dans des classes bien définies.
- Tester notre code régulièrement pour éviter d'accumuler les bugs et d'autres problèmes.
- Lire et analyser un énoncé complexe pour le découper en plusieurs problèmes plus simples à résoudre.
- Travailler de manière plus autonome sur un projet de programmation complet, en partant de zéro.

6 Conclusion

Cette SAE nous a permis de mettre en pratique ce que nous avons appris en cours, notamment sur les graphes, les algorithmes et la programmation orientée objet. Nous avons construit un programme capable de lire un fichier de données, de créer un graphe et de trouver les chemins les plus courts entre deux points.

Tout au long du projet, nous avons appris à organiser notre code, à gérer les erreurs possibles, à travailler avec des `ArrayList`, et à utiliser des algorithmes comme Dijkstra et Bellman-Ford. Nous avons aussi compris comment tester notre code pour vérifier qu'il fonctionne bien.

Même si certaines parties étaient difficiles, comme la lecture du fichier ou la gestion des changements de ligne, elles nous ont permis de progresser et de mieux comprendre comment fonctionnent les programmes en pratique.

En résumé, ce projet nous a beaucoup appris. Il nous a aidés à mieux comprendre les graphes, à coder de manière plus claire et à travailler en équipe sur un projet complet.