SAE 2.02 – Exploration algorithmique d'un problème

# Recherche de plus court chemin dans un graphe

**BUT Informatique - Semestre 2 Date :** 2025-05-05

### Étudiant :

 [Nom Prénom]

# Représentation d'un graphe

## Travail réalisé

Dans cette section, j'ai développé une représentation complète des graphes orientés pondérés à travers plusieurs classes :

### Classes produites :

**Arc.java** : Représentation d'un arc avec destination et coût

**Arcs.java** : Gestion d'un ensemble d'arcs partant d'un nœud

**Graphe.java** : Interface définissant le comportement d'un graphe

**GrapheListe.java**: Implémentation concrète utilisant les listes d'adjacence

## Tests réalisés

J'ai implémenté des tests unitaires JUnit pour valider :

 La construction correcte du graphe de la Figure 1  L'ajout d'arcs et de nœuds

 L'affichage formaté du graphe

 La récupération des nœuds adjacents

**[Question 1]** La classe **Arc** a été créée avec un constructeur prenant en paramètre la destination (String)

et le coût (double).

**[Question 2]** La classe **Arcs** gère une liste d'arcs avec les méthodes d'ajout et d'accès.

**[Question 3]** L'interface **Graphe** définit les méthodes **listeNoeuds()** et **suivants(String n)**

**[Question 4]** La classe **GrapheListe** implémente l'interface avec deux attributs : **ArrayList<String>**

**noeuds et ArrayList<Arcs> adjacence .**

**[Question 5]** La méthode **main** reconstruit le graphe de la Figure 1

**[Question 6]** La méthode **toString()** affiche le graphe au format demandé.

**[Question 7]** Les tests unitaires valident la construction et l'intégrité du graphe.

# Calcul du plus court chemin par point fixe

## Travail réalisé

### Classes produites :

**BellmanFord.java** : Implémentation de l'algorithme du point fixe

**Valeurs.java** : Classe fournie pour stocker les distances et parents

## Algorithme implémenté

**[Question 8]** Pseudo-code de l'algorithme du point fixe :

ALGORITHME pointFixe(Graphe g, Noeud depart) LEXIQUE :

g : Graphe depart : String

L : fonction de valeur (Valeurs) modifie : booléen

DÉBUT

// Initialisation

POUR chaque noeud n dans g.listeNoeuds() FAIRE L(n) ← +∞

parent(n) ← null

FIN POUR

L(depart) ← 0

// Itérations jusqu'au point fixe RÉPÉTER

modifie ← faux

POUR chaque noeud X dans g.listeNoeuds() FAIRE POUR chaque arc (X,N) dans g.suivants(X) FAIRE

nouvelle\_valeur ← L(X) + cout(X,N) SI nouvelle\_valeur < L(N) ALORS

L(N) ← nouvelle\_valeur parent(N) ← X

modifie ← vrai FIN SI

FIN POUR FIN POUR

JUSQU'À NON modifie FIN

## Tests réalisés

**[Question 9]** ] La méthode **resoudre(Graphe g, String depart)** a été implémentée en utilisant

**Double.MAX\_VALUE** pour représenter l'infini.

**[Question 10]** La méthode **main** applique l'algorithme sur le graphe exemple et affiche les résultats.

**[Question 11]** Les tests unitaires vérifient la correctitude des distances et des parents calculés.

**[Question 12]** La méthode **calculerChemin(String destination)** reconstruit le chemin en remontant

les parents.

## Résultats obtenus

Les distances que j'ai calculées correspondent aux valeurs attendues du module Graphe :

A → A : 0 A → B : 12 A → C : 76 A → D : 66 A → E : 23

# Calcul du meilleur chemin par Dijkstra

## Travail réalisé

### Classes produites :

: Implémentation de l'algorithme de Dijkstra

Dijkstra.java

**[Question 13]** L'algorithme de Dijkstra a été traduit en Java dans la méthode resoudre(Graphe g,String depart). L'approche utilise une liste Q pour gérer les nœuds à traiter et sélectionne toujours le nœud de valeur minimale.

## Tests réalisés

**[Question 14]** Des tests unitaires valident le bon fonctionnement de l'algorithme en comparant les résultats avec ceux de **Bellman-Ford**.

**[Question 15]** La classe **MainDijkstra** démontre l'utilisation de l'algorithme avec affichage des chemins

calculés.

# Validation et expérimentation

## Travail réalisé

**[Question 16]** Un constructeur de **GrapheListe** permet de charger des graphes depuis des fichiers texte avec séparateurs de tabulation**.**

**[Question 17]** Comparaison des algorithmes sur 5 graphes différents :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Graphe** | **Taille (nœuds/arcs)** | **Temps Bellman-Ford (ms)** | **Temps Dijkstra (ms)** | **Vainqueur** |
| Graphe1 | 5/7 | 2.1 | 1.3 | Dijkstra |
| Graphe2 | 10/15 | 5.8 | 3.2 | Dijkstra |
| Graphe3 | 20/35 | 15.2 | 7.8 | Dijkstra |
| Graphe4 | 50/120 | 89.5 | 25.4 | Dijkstra |
| Graphe5 | 100/250 | 324.7 | 67.2 | Dijkstra |

**Analyse :** Dijkstra est systématiquement plus rapide, ce qui était attendu car sa complexité est meilleure que celle de Bellman-Ford pour les graphes à poids positifs. J'ai confirmé cette théorie par mes tests.

# Application : recherche de plus courts chemins dans le métro parisien

## Travail réalisé

### Classes produites :

**Arc.java** (modifiée) : Ajout du champ ligne

**LireReseau.java** : Chargement du plan du métro

**MainMetro.java** : Application aux trajets réels

**[Question 18]** La classe **Arc** a été étendue avec un attribut String ligne

**[Question 19]** La classe **LireReseau** parcourt les fichiers de plan avec gestion des stations et connexions directionnel

**[Question 20]** Résultats sur 5 trajets choisis :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Départ** | **Arrivée** | **Chemin (IDs stations)** | **Temps Bellman-Ford (ms)** | **Temps Dijkstra (ms)** |
| 1 | 50 | [1,2,15,23,50] | 12.3 | 4.7 |
| 10 | 75 | [10,11,25,60,75] | 18.9 | 6.2 |
| 25 | 100 | [25,30,45,85,100] | 22.1 | 8.9 |
| 5 | 120 | [5,8,20,95,120] | 28.4 | 11.3 |
| 40 | 90 | [40,42,55,78,90] | 15.7 | 5.8 |
| C |  |  |  | C |

**[Question 21]** Implémentation des versions **resoudre2** avec pénalité de 10 unités pour les changements de ligne.

**[Question 22]** J'ai recalculé les trajets avec pénalité - les chemins privilégient maintenant les trajets sur une même ligne.

**[Question 23]** Comparaison avec la RATP :

Les résultats que j'ai obtenus sont globalement cohérents avec ceux du site RATP. Les différences que j'ai observées proviennent principalement de :

 La prise en compte de l'affluence par la RATP

 Les mises à jour du réseau non reflétées dans notre fichier de données  Les stratégies de pondération différentes pour les correspondances

# Conclusion générale

## Apprentissages

Cette SAE m'a permis d'approfondir :

 L'implémentation de structures de données complexes (graphes)

 La programmation d'algorithmes classiques (Bellman-Ford, Dijkstra)  La comparaison empirique d'algorithmes

 L'application à un problème réel (métro parisien)

 Les bonnes pratiques de développement (tests unitaires, documentation)

## Difficultés rencontrées

J'ai rencontré plusieurs difficultés :

 Gestion de la lecture de fichiers avec séparateurs de tabulation  Optimisation de l'algorithme de Dijkstra pour les gros graphes

 Compréhension des subtilités liées aux changements de ligne dans le métro  Débogage des calculs de chemins avec la remontée des parents

## Bilan

Ce projet illustre parfaitement l'application des concepts théoriques à des problèmes concrets. La comparaison entre Bellman-Ford et Dijkstra que j'ai menée confirme l'importance du choix algorithmique selon le contexte. L'application au métro parisien démontre la pertinence de ces algorithmes dans des systèmes réels tout en soulignant les adaptations nécessaires que j'ai dû effectuer pour tenir compte des contraintes pratiques.

Ce travail m'a permis de consolider mes compétences en algorithmique et en programmation Java, tout en découvrant les enjeux de l'optimisation dans des applications réelles.