**SAÉ 2.02 - Exploration algorithmique d’un problème**

**Binôme :** ATTENOT / RODRIGUES NETO  
**Semestre 2 - BUT Informatique**

**1. Représentation d’un graphe**

**Objectif**

L’objectif de cette première partie était de concevoir une représentation informatique d’un graphe orienté pondéré, en utilisant la structure de données la plus adaptée. Le but était de permettre une utilisation future par des algorithmes tels que Bellman-Ford ou Dijkstra.

**Travail réalisé**

Nous avons mis en œuvre une interface Graphe avec les méthodes suivantes :

public interface Graphe {

List<String> listeNoeuds();

List<Arc> suivants(String n);

}

Puis, la classe GrapheListe a été développée afin de représenter concrètement ce graphe par des listes d’adjacence. Les classes suivantes ont été produites :

* **Classe Arc** : représente un arc orienté avec destination, coût, et nom de ligne.
* **Classe GrapheListe** : gère les nœuds et arcs via une structure de type Map<String, List<Arc>>.

**[Question 1 à 7]** : Nous avons implémenté les méthodes demandées et vérifié le bon fonctionnement avec des graphes simples créés à la main.

**Difficultés rencontrées**

Le traitement du fichier metro-paris.txt s’est révélé complexe à cause des séparateurs et de la gestion des identifiants. Nous avons créé une méthode robuste LireReseau.lire(...) qui reconstruit le graphe à partir des identifiants et des noms de stations.

**2. Point fixe (Bellman-Ford)**

**Objectif**

L’objectif était d’implémenter l’algorithme de Bellman-Ford qui repose sur une mise à jour itérative des distances minimales, jusqu’à convergence (point fixe).

**Travail réalisé**

* **Classe Valeurs** : contient les distances (Map<String, Double>), parents (Map<String, String>) et lignes (Map<String, String>).
* **Classe BellmanFord** : contient les méthodes resoudre() (version classique) et resoudre2() (avec pénalité de changement de ligne).

**Code produit :**

* Calcul du plus court chemin
* Remontée des parents avec la méthode calculerChemin(String destination)

**[Question 9 à 12]** :

* L’algorithme a été testé sur de petits graphes ainsi que sur le graphe du métro.
* Les chemins ont été vérifiés grâce à la méthode calculerChemin.

**Difficultés**

* Les chemins étaient initialement vides. Cela venait de noms de stations mal chargés ou absents du graphe (non appel de ajouterNoeud).
* La logique de convergence a nécessité des tests rigoureux.

**Ce qui a été fait**

✅ Classes Valeurs, BellmanFord, méthodes resoudre et resoudre2 complètes

**3. Dijkstra (glouton)**

**Objectif**

Implémenter l’algorithme glouton de Dijkstra qui s’appuie sur le traitement du sommet ayant la distance minimale courante.

**Travail réalisé**

* **Classe Dijkstra** avec méthodes resoudre() et resoudre2()
* Méthode trouverNoeudValeurMinimale(...) pour sélectionner le prochain sommet à traiter.

**[Question 13 à 15]** :

* Nous avons strictement suivi le pseudo-code fourni dans l’énoncé.
* Le fonctionnement a été validé avec MainMetro, sur les mêmes graphes que pour Bellman-Ford.

**Ce qui a été fait**

✅ Résolution classique et avec pénalité de ligne, avec comparaison à Bellman-Ford

**Difficultés**

* Bien gérer la mise à jour des parents uniquement si la nouvelle distance est meilleure
* Contrôler les changements de ligne pour ajouter la pénalité à bon escient

**4. Validation**

**Objectif**

Réaliser des tests complets sur des trajets réels du métro parisien, avec affichage du chemin, du coût total et du temps de traitement pour chaque algorithme.

**Travail réalisé**

* Classe MainMetro.java pour comparer Bellman-Ford et Dijkstra (avec/sans pénalité)
* 5 trajets testés automatiquement via tableau String[][] trajets

**Exemple de trajets testés**

{ "Châtelet", "Nation" },

{ "Porte de Clignancourt", "Gare de Lyon" },

{ "Place d'Italie", "République" },

{ "Odéon", "Gare du Nord" },

{ "Montparnasse Bienvenue", "Bastille" }

**Affichage des résultats**

Chaque trajet affiche :

* Le chemin trouvé
* Le temps de calcul de chaque algorithme
* Les chemins étaient initialement vides (stations mal détectées), corrigé en rendant ajouterNoeud() public et nettoyant les noms dans metro-paris.txt

**Ce qui a été fait**

✅ Tests sur réseau réel avec extraction automatique des chemins

**5. Comparaison avec le site de la RATP**

**[Question 23]**

Nous avons comparé les trajets trouvés par nos algorithmes à ceux donnés par le site RATP (via l’outil IDFM).

**Constats :**

* Les chemins proposés sont souvent similaires.
* La version avec pénalité donne parfois des trajets différents (moins de changements).
* Le site RATP favorise parfois des correspondances pour minimiser le temps global.

**Exemple**

| **Trajet** | **Chemin RATP** | **Chemin Algorithme** |
| --- | --- | --- |
| Châtelet → Nation | Ligne 1 directe | Ligne 1 directe (identique) |
| Montparnasse → Bastille | L13 → L10 → L5 | L4 → L1 (différent mais cohérent) |

**6. Conclusion générale**

Cette SAE a été particulièrement formatrice.

**Compétences développées**

* Mise en œuvre d’algorithmes classiques
* Traitement de données issues d’un fichier complexe
* Comparaison objective de performances

**Ce que nous avons appris**

* Les structures de graphe sont au cœur des algorithmes de recherche de chemins
* L’importance de bien valider les données en entrée (stations, connexions)

**Difficultés rencontrées**

* Parsing du fichier metro-paris.txt : détection de sections, tabulations, accents
* Les trajets vides ont demandé une vérification minutieuse des structures internes

**Bilan**

* Le projet est **fonctionnel** pour les cas testés
* Les chemins sont corrects et cohérents avec les trajets RATP
* Le code est propre, structuré et modulaire

**Améliorations possibles**

* Ajouter une interface graphique pour visualiser les chemins
* Proposer des itinéraires alternatifs
* Améliorer la prise en compte du confort (moins de changements, affluence, accessibilité)

**Fichier rédigé par le binôme ATTENOT / RODRIGUES NETO - 2025**