Tolkacheva Anastasia

Schaeffer Johan

**Rapport Algorithmes de parcours de graphes**

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

**Classe Arc (Question 1) :**

Nous avons créé une classe Arc pour représenter un arc dans un graphe orienté. Un arc contient deux informations :

* le nom du nœud de destination (une chaîne de caractères) ;
* le coût de l'arc (un nombre réel positif).

Cela permet de connaître la direction de l'arc et son poids, utile pour les algorithmes de plus court chemin.

**Classe Arcs (Question 2) :**

La classe Arcs regroupe une liste d'objets Arc qui partent d'un même nœud. Elle fournit :

* une méthode ajouterArc(Arc a) pour ajouter un arc ;
* une méthode getArcs() pour récupérer la liste des arcs.

**Interface Graphe (Question 3) :**

Pour généraliser la structure d'un graphe, nous avons défini une interface Graphe avec deux méthodes essentielles :

* listeNoeuds() : retourne la liste des nœuds ;
* suivants(String n) : retourne la liste des arcs partant d'un nœud.

**Classe GrapheListe (Question 4) :**

Nous avons implémenté l'interface Graphe dans la classe GrapheListe. Cette classe utilise deux listes parallèles :

* une liste de nœuds (chaînes de caractères) ;
* une liste d'objets Arcs correspondant aux arcs partant de chaque nœud.

Une méthode ajouterArc(String depart, String destination, double cout) permet d'ajouter un arc et crée les nœuds si besoin.

**Test manuel (Question 5) :**

Dans une méthode main, nous avons créé un petit graphe d'exemple (comme celui du sujet) pour tester l'ajout d'arcs et la structure du graphe.

**Affichage (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) :**

Nous avons écrit des tests pour vérifier que :

* la liste des nœuds est correcte ;
* les arcs partants de chaque nœud sont corrects ;
* la méthode toString() affiche bien le graphe.

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

**Dijkstra (Question 8) :**

L'algorithme de Dijkstra est implémenté dans la classe Dijkstra. Il permet de calculer les plus courts chemins à partir d'un nœud source, en supposant que tous les coûts sont positifs.

Le principe :

1. Initialisation des distances à l'infini, sauf pour le nœud de départ (0).
2. Parcours des nœuds par ordre de distance minimale.
3. Mise à jour des voisins si un chemin plus court est trouvé.

**Bellman-Ford (Question 9) :**

L'algorithme de Bellman-Ford, codé dans la classe BellmanFord, fonctionne aussi pour les coûts négatifs. Il répète la mise à jour des distances jusqu'à ce qu'aucun changement ne soit détecté.

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

**Utilisation réelle (Question 10) :**

Nous avons testé les deux algorithmes sur le fichier plan-reseau.txt, représentant le réseau du métro parisien. La classe LireReseau lit ce fichier et construit un GrapheListe.

Des trajets ont été testés, et les chemins, ainsi que le temps d'exécution (en nanosecondes), ont été affichés pour chaque algorithme.

Exemple d'affichage :

Départ Arrivée Chemin (ID stations) Temps Dijkstra Temps Bellman-Ford

---------------------------------------------------------------------------------------------

1 17 [1, 2, 3, ..., 17] 2000000 ns 5000000 ns

**Changement de ligne (Question 11) :**

Nous avons implémenté une version modifiée de chaque algorithme (resoudre2) qui ajoute une pénalité de 10 en cas de changement de ligne de métro. Cela permet de simuler les correspondances.

Le changement de ligne est détecté grâce à la comparaison des noms de lignes dans les objets Arc.

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

Nous avons comparé les temps d’exécution de Dijkstra et Bellman-Ford sur différents trajets. Dijkstra est généralement plus rapide, car il s’arrête dès que tous les plus courts chemins sont trouvés. Bellman-Ford, lui, répète plusieurs fois les vérifications, ce qui rallonge le temps de calcul.

**Comparaison des chemins (Question 13) :**

Nous avons aussi comparé les chemins retournés par les deux algorithmes. Dans la majorité des cas, ils trouvent les mêmes chemins. Cependant, en ajoutant la pénalité de changement de ligne, les chemins peuvent différer, car certains trajets avec correspondances deviennent plus coûteux.

**Influence de la pénalité (Question 14) :**

Nous avons observé qu’ajouter une pénalité de 10 pour les changements de ligne influence fortement le chemin choisi. Par exemple, un trajet avec correspondance peut devenir moins intéressant qu’un chemin plus long mais sur une même ligne.

**Simulation d’un trajet (Question 15) :**

Nous avons utilisé le programme pour simuler un trajet entre deux stations données. Le chemin est affiché sous forme de liste de noms ou d’ID de stations. Cela permet de visualiser l’itinéraire choisi par l’algorithme.

**Affichage détaillé (Question 16) :**

Nous avons ajouté un affichage plus détaillé du chemin, incluant les stations traversées, leur ligne et le coût total. Cela permet une meilleure compréhension du résultat fourni par l’algorithme.

**Robustesse du programme (Question 17) :**

Le programme gère les cas où il n’y a pas de chemin entre deux stations (par exemple si elles ne sont pas connectées). Dans ce cas, aucune erreur ne se produit, et une liste vide ou un message explicatif est affiché.

**Lecture du fichier réseau (Question 18) :**

La classe LireReseau lit correctement les stations (avec leur nom et ID) et les connexions (avec temps et ligne). Nous avons corrigé des erreurs liées aux identifiants manquants ou aux noms incorrects pour garantir une lecture fiable.

**Construction automatique du graphe (Question 19) :**

Le graphe est construit automatiquement depuis le fichier. Chaque station est identifiée et reliée à ses voisines selon les données du fichier. Cela évite toute saisie manuelle des données.

**Généralisation (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.

**Analyse des résultats (Question 21) :**

Nous avons analysé les résultats obtenus sur plusieurs trajets. Les chemins trouvés sont pertinents, et l’ajout de la pénalité améliore la simulation du comportement d’un utilisateur réel (préférer un trajet direct sans changement).

**Améliorations possibles (Question 22) :**

Le programme pourrait être amélioré en ajoutant une interface graphique, en permettant une saisie dynamique des stations, ou en affichant les trajets sur un plan. On pourrait aussi prendre en compte le temps d’attente ou la fréquence des métros pour rendre les calculs encore plus réalistes.

**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.