LELIEVRE Etienne

DEBRIS Gabriel

S2.02 Exploration algorithmique d'un problème

Compte-rendu

Question 17 :

Fichier	Bellman-Ford (ms)	Dijkstra (ms)
graphe1.txt	0.5	0.2
graphe2.txt	2.3	0.8
graphe3.txt	1.9	0.6
graphe4.txt	5.4	1.5
graphe5.txt	10.1	2.2

Dijkstra est plus rapide.

Cela s’explique car Bellman-Ford vérifie tous les arcs n - 1 fois, ce qui est plus coûteux.

Oui, car :

Dijkstra utilise une stratégie "gourmande", plus rapide si tous les coûts sont positifs

Bellman-Ford est plus lent généralement

Rendu

Pour représenter un graphe, nous avons implémenté la classe GrapheListe qui utilise une structure de type HashMap<String, List<Arc>> pour stocker les arcs partant de chaque sommet.

Cette classe contient les méthodes suivantes :

- ajouterArc(String depart, String destination, int cout)
- suivants(String sommet)
- getSommets()
- Constructeur depuis un fichier texte (type graphe simple)

Question 1)

Pour l'implémentation du graphe, nous avons choisi une liste d'adjacence car elle est plus adaptée aux graphes creux (peu d'arcs par sommet), ce qui est le cas pour notre usage (réseaux de transport).

Nous avons testé la lecture du fichier, l'ajout d'arcs et la méthode suivants() avec des fichiers simples comme celui-ci :

A B 5
B C 10
A C 20

Nous avons vérifié que tous les arcs étaient bien ajoutés et que les méthodes renvoyaient les bons résultats.

Nous avons implémenté l'algorithme de Bellman-Ford dans la classe AlgoBellmanFord.

Il contient les méthodes suivantes :

- initialiser(String sommetDepart)
- resoudre()
- getChemin(String destination)

Question 6)

Oui, on peut adapter la méthode du point fixe à un graphe orienté avec poids négatifs, tant qu'il n'y a pas de cycle de poids négatif.

Question 9)

Non, on n'a pas besoin d'inverser le graphe, car le sens des arcs est déjà pris en compte dans l'algorithme.

Nous avons testé cet algorithme sur plusieurs petits graphes avec sommets nommés en lettres (A, B, C…) et avons vérifié que les distances et les chemins calculés étaient corrects.

Dijkstra

Nous avons implémenté l'algorithme de Dijkstra dans la classe AlgoDijkstra.

Il contient les méthodes suivantes :

- initialiser(String sommetDepart)
- resoudre()
- getChemin(String destination)

Question 11)

L’algorithme de Dijkstra est incompatible avec les arcs de poids négatif, car une fois qu’un sommet est marqué comme traité, on ne le revisite plus, ce qui pourrait donner un mauvais résultat si un chemin plus court arrive après via un arc négatif.

On a comparé les performances de Bellman-Ford et Dijkstra sur des graphes plus gros (type réseau métro) pour observer les différences de temps.

Validation

Nous avons utilisé la classe LireReseau pour lire un fichier texte contenant les stations et connexions du métro parisien.

Format utilisé dans le fichier :

Stations:
1:Station A:0:0:1
2:Station B:0:1:1
3:Station C:0:2:2

Connexions:
1:2:4:1
2:3:6:2

Chaque connexion donne deux arcs (bidirectionnels). Nous avons aussi modifié la classe Arc pour ajouter un champ ligne, de type String.

Nous avons implémenté une version modifiée des algorithmes (méthode resoudre2) qui ajoute une pénalité de 10 au coût lors d’un changement de ligne.

Question 20)

Nous avons réalisé 5 tests entre différentes stations du métro, avec et sans pénalité de changement de ligne. Les temps de calcul étaient similaires pour les deux algorithmes, mais les chemins pouvaient différer à cause des pénalités.

Question 23)

En comparant avec le site de la RATP, on a constaté que nos chemins étaient proches, mais pas toujours identiques. La RATP propose parfois des trajets plus longs mais avec moins de correspondances. Notre algorithme pénalisé arrive à imiter ce comportement.