DUMONT Nathan,
GRAINE Mathis,
S2C

1. Représentation d’un graphe

Dans cette première partie, on a juste mis en place les bases du graphe avec des classes Java.
On représente les nœuds avec des Strings et les arcs avec un coût (et une ligne pour le métro plus tard).

→ Classes faites : Arc, Arcs, Graphe (interface), GrapheListe.

→ [Q1 à Q7] : tout a été fait comme demandé, avec un toString() pour afficher, un main pour tester, et des tests JUnit.

Exemple d'affichage :
A -> B(12.0) D(87.0)
B -> E(11.0)
C -> A(19.0)
D -> B(23.0) C(10.0)
E -> D(43.0)

2. Point fixe (Bellman-Ford)

On a implémenté l’algo de Bellman-Ford vu en TD. Il tourne jusqu’à ce que plus rien ne change.

Les valeurs sont mises à jour à chaque fois qu'on trouve un chemin plus court.

→ Classe : BellmanFord.java
→ Méthode principale : resoudre()
→ [Q8] : on a rédigé le pseudo-code dans le rapport.
Algorithme pointFixe(Graphe g, String depart)

```
Entrée :
- g : un graphe orienté pondéré avec des coûts positifs
- depart : nom du nœud de départ (type String)

Sortie :
- valeurs : un objet qui associe à chaque nœud :
   * sa distance minimale depuis le départ (L)
   * son parent sur le plus court chemin

Début
   Créer un objet Valeurs v

   Pour chaque nœud x de g.listeNoeuds() faire
      v.setValeur(x, +∞)
      v.setParent(x, null)
   Fin Pour

   v.setValeur(depart, 0)   // distance du départ à lui-même = 0

   booléen changement ← vrai

   Tant que changement faire
      changement ← faux

      Pour chaque nœud x de g.listeNoeuds() faire
         Pour chaque arc (x → y) de g.suivants(x) faire
            nouvelleValeur ← v.getValeur(x) + coût(x, y)

            Si nouvelleValeur < v.getValeur(y) alors
               v.setValeur(y, nouvelleValeur)
               v.setParent(y, x)
               changement ← vrai
            Fin Si
         Fin Pour
      Fin Pour
   Fin Tant que

   Retourner v
Fin
```

→ [Q9] : le code a été fait, fonctionne bien.
→ [Q10] : on a fait un main pour tester avec le graphe de la figure.
→ [Q11] : tests unitaires JUnit faits et validés.
→ [Q12] : on a aussi codé la méthode pour reconstruire un chemin (calculerChemin).

3. Dijkstra

Même principe que Bellman-Ford mais avec une sélection du nœud à plus petite distance à chaque fois.
Ça évite les re-calculs inutiles.

→ Classe : Dijkstra.java
→ Méthodes : resoudre() et resoudre2() (avec pénalité)
→ [Q13 à Q15] : tout a été fait comme dans le sujet, avec tests JUnit et un main.

4. Validation et comparaison

→ [Q16] : le chargement depuis fichier texte est codé dans GrapheListe (constructeur).
→ [Q17] : on a fait un programme ComparerGraphes.java qui mesure les temps de Bellman-Ford et Dijkstra sur 5 graphes.
→ Résultat : Dijkstra est souvent plus rapide, surtout si le graphe est grand.

5. Application : métro parisien

→ [Q18] : on a modifié la classe Arc pour ajouter le champ ligne.
→ [Q19] : LireReseau lit le fichier plan-reseau.txt (on a ignoré les noms des stations).
→ [Q20] : on a testé 5 trajets (départ/arrivée), affiché le chemin et les temps avec chaque algo.

Trajet	Départ	Arrivée	Temps BF (ms)	mps Dijkstra (m
1	1	10	2.6678	0.8060
2	3	20	0.0118	0.0046
3	5	15	0.0162	0.0040
4	8	25	0.0058	0.0034
5	2	18	0.0060	0.0037

→ [Q21] : on a ajouté une pénalité de +10 si la ligne change.
→ [Q22] : on a refait les trajets avec cette version.

Trajet	Départ	Arrivée	Temps BF (ms)	mps Dijkstra (m
1	1	10	2.0507	1.0953
2	3	20	0.0167	0.0057
3	5	15	0.0125	0.0056
4	8	25	0.0095	0.0087
5	2	18	0.0093	0.0050

→ [Q23] : on a comparé avec le site de la RATP : parfois les chemins sont différents car le site optimise aussi les correspondances et les horaires.

6. Conclusion

On a appris à manipuler des graphes en Java, à coder des algos classiques (Bellman-Ford, Dijkstra), et à structurer un projet.

Les difficultés : la gestion des indices dans GrapheListe, bien suivre les indices des lignes du métro, et la logique des tests JUnit.

Mais globalement, on a tout fait et tout fonctionne (même si c'est pas super optimisé).