Binôme:
Husser Yanis
Olsozwiak-Delandre Mathéo

SAE 2.02 - Exploration algorithmique d'un problème

Section 2 : Représentation d'un graphe

Dans cette partie, nous avons mis en place une structure simple pour représenter des graphes orientés avec des coûts associés à chaque arc. Les noeuds sont représentés simplement par des chaînes de caractères, ce qui permet de les manipuler facilement sans créer de classe spécifique pour eux.

Pour modéliser les arcs entre les noeuds, nous avons défini une classe Arc, qui contient le nom du noeud de destination et le coût du trajet. Chaque noeud peut avoir plusieurs arcs sortants, donc nous avons aussi créé une classe Arcs qui gère une liste d'objets Arc. Cela permet de regrouper les arcs liés à un même noeud.

L'ensemble du graphe est manipulé à travers une interface Graphe, qui impose deux méthodes essentielles : listeNoeuds() pour obtenir tous les noeuds du graphe, et suivants(String n) pour obtenir les arcs sortants d'un noeud donné. L'implémentation de cette interface est assurée par la classe GrapheListe, qui utilise deux listes parallèles : une pour stocker les noms des noeuds et une autre pour les objets Arcs correspondants. Quand un arc est ajouté, si le noeud de départ ou d'arrivée n'existe pas encore, il est automatiquement ajouté à la structure.

Une méthode toString() a également été mise en place pour pouvoir afficher facilement le contenu du graphe, avec un format lisible indiquant pour chaque noeud les destinations possibles et les coûts associés.

Pour s'assurer du bon fonctionnement de notre structure, nous avons mis en place plusieurs tests unitaires. Par exemple, nous avons vérifié que les noeuds sont bien ajoutés même s'ils ne sont pas présents au moment où on ajoute un arc. On a aussi testé la méthode getIndice() qui permet de retrouver la position d'un noeud dans la liste. On a ensuite vérifié que listeNoeuds() retourne bien tous les noeuds ajoutés, et que la méthode suivants() renvoie les bons arcs pour un noeud donné. Enfin, un test a été écrit pour s'assurer que la méthode toString() retourne une représentation correcte et complète du graphe construit.

Section 3 : Calcul du plus court chemin par point fixe

L'objectif de cette section était de résoudre le problème du plus court chemin dans un graphe en utilisant l'algorithme du point fixe.

Nous avons d'abord rédigé le pseudo-code de l'algorithme. Celui-ci commence par initialiser la distance de tous les noeuds à +infini, sauf celle du noeud de départ qui est mise à 0. Ensuite, tant qu'il y a des modifications indiquant ainsi une amélioration des plus courts chemins, l'algorithme parcourt tous les noeuds et leurs arcs sortants pour mettre à jour les distances et les parents.

Question 8 - Algorithme utilisé:

```
fonction pointFixe(Graphe g, Noeud depart):
debut
        liste <- g.listeNoeuds()
        taille <- liste.size()

        pour i de 0 a taille - 1 faire
                X <- liste[i]
                L(X) <- +infini
        fpour

        L(depart) <- 0

        changement <- vrai
        tant que changement = vrai faire
                changement <- faux
                pour i de 0 a taille - 1 faire
                        X <- liste[i]
                       arcs <- g.suivants(X)
                       pour j de 0 a arcs.size() - 1 faire
                        arc <- arcs[j]
                        Y <- arc.getDestination()
                        cout <- arc.getCout()
                                si L(X) + cout < L(Y) alors
                                L(Y) <- L(X) + cout
                                parent(Y) <- X
                                changement <- vrai
                                fsi
                       fpour
                fpour
        ftant
fin
```

Nous avons ensuite implémenté cet algorithme en Java dans la classe BellmanFord, via la méthode resoudre(Graphe g, String depart). Cette méthode prend en entrée un graphe et le nom d'un noeud de départ, puis retourne un objet Valeurs. Cet objet contient, pour chaque noeud, la distance minimale depuis le départ ainsi que le noeud parent qui précède ce noeud sur le chemin optimal.

Pour stocker ces informations, nous avons créé une classe Valeurs qui associe à chaque noeud sa distance depuis le point de départ ainsi que son parent sur le chemin. Cette classe nous a aussi permis de manipuler les données comme les parents ou les longueurs de chemins associées à chaque noeud.

Une fois l'implémentation terminée, nous avons testé l'algorithme sur un exemple de graphe pour vérifier que les distances et les chemins calculés correspondaient bien aux attentes. Nous avons également rédigé un test unitaire pour automatiser cette vérification. Enfin, nous avons écrit la méthode calculerChemin qui, à partir d'un noeud donné en paramètre, renvoie la liste des noeuds à parcourir pour atteindre ce noeud par le chemin le plus optimal.

Section 4 : Algorithme de Dijkstra

Dans cette section, nous avons implémenté l'algorithme de Dijkstra afin de résoudre le problème du plus court chemin dans un graphe orienté pondéré mais cette fois-ci avec un autre algorithme.

Nous avons commencé par recopier l'algorithme tel qu'il est présenté dans le sujet, puis nous l'avons traduit en Java dans une classe Dijkstra, à travers une méthode resoudre(Graphe g, String depart) respectant la même signature que celle utilisée pour Bellman-Ford. Cette méthode retourne également un objet de type Valeurs, contenant les distances minimales depuis le noeud de départ et les parents pour reconstruire les chemins optimaux.

Question 13 - Algorithme utilisé:

```
Entrées :
   G un graphe orienté avec une pondération positive des arcs
   (coût ou poids)
   A un sommet (départ) de G

   Début
   Q <- {} // utilisation d'une liste de noeuds à traiter
   Pour chaque sommet v de G faire
      v.valeur <- Infini
      v.parent <- Indéfini
      Q <- Q U {v} // ajouter le sommet v à la liste Q
   Fin Pour

   A.valeur <- 0
   Tant que Q est un ensemble non vide faire
      u <- un sommet de Q telle que u.valeur est minimal
      // enlever le sommet u de la liste Q
      Q <- Q \ {u}
      Pour chaque sommet v de Q tel que l'arc (u,v) existe faire
         d <- u.valeur + poids(u,v)
         Si d < v.valeur
            // le chemin est plus interessant
            Alors v.valeur <- d
                  v.parent <- u
         Fin Si
      Fin Pour
   Fin Tant que
Fin
```

A chaque étape on sélectionne le nœud non traité avec la plus petite distance connue, et on met à jour ses voisins si un chemin plus court est trouvé.

L'algorithme a été codé dans la classe Dijkstra, via la méthode resoudre(Graphe g, String depart), identique à celle utilisée pour Bellman-Ford. Elle retourne un objet Valeurs contenant les distances et les parents.

Nous avons testé cette méthode en créant une classe MainDijkstra, avec le même graphe que dans la section précédente. Cela nous a permis de vérifier les distances calculées et de reconstruire les chemins grâce à calculerChemin. Un test unitaire a également été rédigé pour s'assurer de la validité des résultats.

Section 5 : Validation et expérimentation

Dans cette section, on a d'abord implémenté un nouveau constructeur dans la classe GrapheListe pour pouvoir charger automatiquement un graphe à partir d'un fichier texte. Ensuite, on a créé une classe dédiée à la comparaison des deux algorithmes, ComparaisonAlgos qui nous a permis d'analyser les temps d'execution de chacun dans différents cas .

Question 17 - Analyse comparative:

Pour comparer nos deux algorithmes, on les a testés sur 5 graphes complets de tailles différentes allant d'un graphe contenant 10 sommets jusqu'à un graphe contenant 100 sommets, puis on a mesuré le temps d'exécution de chaque algorithme sur chacun d'entre eux.

Sur les petits graphes, les temps étaient assez proches, voire parfois plus rapides pour Bellman-Ford. Par exemple, sur le graphe à 10 sommets, Dijkstra a mis 2 868 400 nanosecondes, contre 756 500 nanosecondes pour Bellman-Ford.

Mais plus on augmentait le nombre de sommets, plus on voyait que Dijkstra allait plus vite. À partir du graphe à 50 sommets, ça commençait déjà à se voir, et sur celui à 100 sommets la différence est devenue bien plus marquée. Sur le graphe complet de 100 points, l'algorithme de Dijkstra a mis environ 9 millions de nanosecondes à terminer, alors que Bellman-Ford en a mis plus de 15 millions.

Cela nous montre que Dijkstra est plus efficace sur les grands graphes. En contrepartie ce dernier est contraint sur le fait de devoir obligatoirement opérer sur des graphes possédants des arcs positifs ce qui justifie l'efficacité supérieure de l'algorithme.

Résultats obtenus:

Graphe : graphes/graphe_10.txt

Temps Dijkstra : 2868400 nanosecondes

Temps BellmanFord : 756500 nanosecondes

Graphe : graphes/graphe_30.txt

Temps Dijkstra : 2369300 nanosecondes

Temps BellmanFord : 2686400 nanosecondes

Graphe : graphes/graphe_50.txt

Temps Dijkstra : 3232200 nanosecondes

Temps BellmanFord : 5544000 nanosecondes

Graphe : graphes/graphe_80.txt

Temps Dijkstra : 6479600 nanosecondes

Temps BellmanFord : 9734600 nanosecondes

Graphe : graphes/graphe_100.txt

Temps Dijkstra : 9059300 nanosecondes

Temps BellmanFord : 15635400 nanosecondes

Section 6 : Application : recherche de plus courts chemins dans le métro parisien

Dans cette partie, on a modifié la classe Arc pour ajouter un attribut ligne, qui stocke le numéro de la ligne de métro. On a aussi modifié son constructeur pour qu'il prenne ce paramètre.

Dans GrapheListe, on a ajouté une deuxième méthode ajouterArc qui prend en paramètre une ligne. L'ancienne méthode est toujours là, mais elle met une ligne par défaut à 1 de sorte gérer cette nouvelle feature tout en conservant le choix d'utiliser la méthode ajouterArc avec ou sans ligne.

On a ensuite créé une classe LireReseau qui lit un fichier avec la structure du métro. Le fichier contient les stations, puis les connexions. On passe les stations pour ensuite lire les connexions et ajouter les arcs dans le graphe. Chaque ajout se fait donc dans les 2 sens comme demandé dans la consigne.

Question 20 - Création du Main Métro:

Les départs et arrivées sont donnés par ID choisis et les temps sont donnés en Nanoseconde.

Départ	Arrivée	Chemin	Temps Calcul Bellman-Ford	Temps Calcul Dijkstra
1	15	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]	3950600ns	10890500ns
7	263	[7, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 259, 260, 261, 262, 263]	3243800ns	5886500ns
25	136	[25, 24, 23, 22, 21, 48, 47, 46, 45, 44, 43, 42, 41, 40, 39, 142, 141, 140, 139, 138, 137, 136]	6255300ns	3761800ns
39	87	[39, 143, 144, 80, 81, 82, 63, 83, 84, 15, 85, 86, 87]	4563800ns	4349800ns
198	150	[39, 143, 144, 80, 81, 82, 63, 83, 84, 15, 85, 86, 87]	4563800ns	4349800ns

Trajets utilisés:

Trajet 1
Grande Arche de la Défense vers Châtelet

Trajet 2
Charles de Gaulle – Étoile vers Mairie d'Issy

Trajet 3
Château de Vincennes vers La Courneuve

Trajet 4
Stalingrad vers Odéon

Trajet 5
Pont de Sèvres vers Pont-Neuf

On a ensuite ajouté une version 2 des deux algorithmes avec resoudre2. Dans ces versions, on ajoute un malus de 10 quand on change de ligne. Pour faire ça, on garde en mémoire la ligne utilisée pour arriver à une station. Si on change de ligne, on ajoute le malus au coût. Le reste du code ne change presque pas.

Question 22 - MainMetro avec resoudre2:

Départ	Arrivée	Chemin	Temps Calcul Bellman-Ford	Temps Calcul Dijkstra
1	15	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]	10679700ns	28509200ns
7	263	[7, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 259, 260, 261, 262, 263]	7907800ns	16859700ns
25	136	[25, 24, 23, 22, 21, 48, 47, 46, 45, 44, 43, 42, 41, 40, 39, 142, 141, 140, 139, 138, 137, 136]	22953300ns	11814800ns
39	87	[39, 143, 144, 80, 81, 82, 63, 83, 84, 15, 85, 86, 87]	11836300ns	10519300ns
198	150	[198, 199, 200, 201, 202, 203, 226, 227, 228, 229, 230, 120, 231, 232, 233, 234, 235, 87, 86, 85, 15, 150]	10105800ns	8907400ns

Pour cette question, on a relancé les 5 trajets avec les versions de Dijkstra et Bellman-Ford qui prennent en compte un malus de 10 quand on change de ligne.

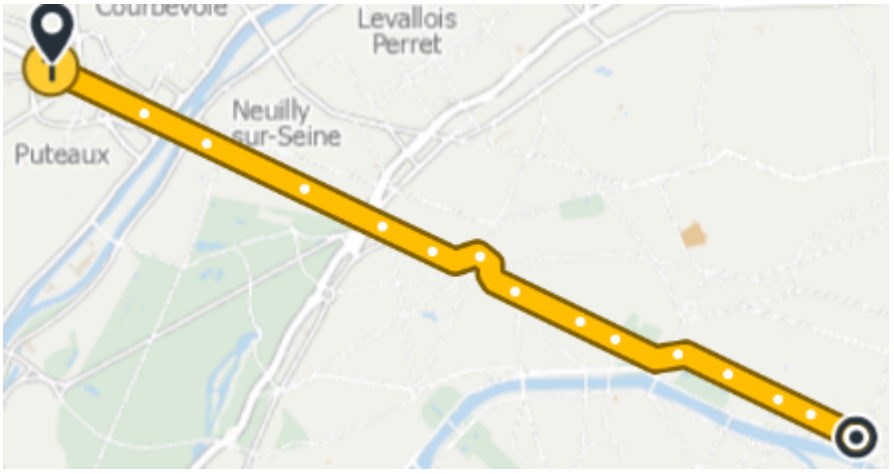
Les temps de calcul sont un peu plus longs qu'avant, ce qui est logique vu qu'on ajoute une vérification en plus à chaque étape. Dijkstra reste quand même plus rapide que Bellman-Ford dans les cas logiques ou il y a assez de matière pour comparer les 2 algorithmes convenablement.

Sur les 5 trajets, un seul a changé de chemin, le dernier. Là, l'algorithme a évité un changement de ligne en passant par un chemin un peu plus long. Pour les 4 autres, le chemin est resté le même, donc soit il n'y avait pas de changement de ligne, soit c'était plus rapide de quand même changer.

Question 23 - Création du Main Métro:

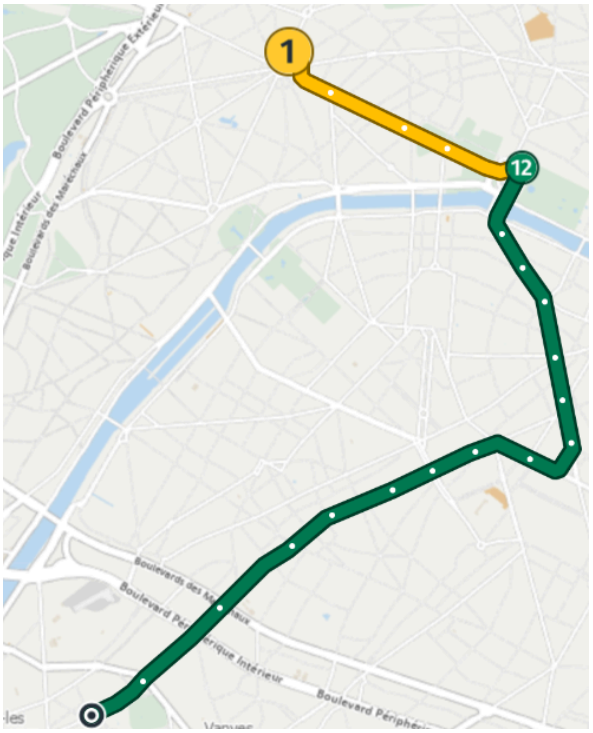
Premier Trajet:

Pour le premier trajet, je peux constater que les chemins empruntés par la RATP pour aller de la station 1 à 15 sont les mêmes que ceux trouvés par les algorithmes de chemins minimaux utilisant la méthode resoudre 2.



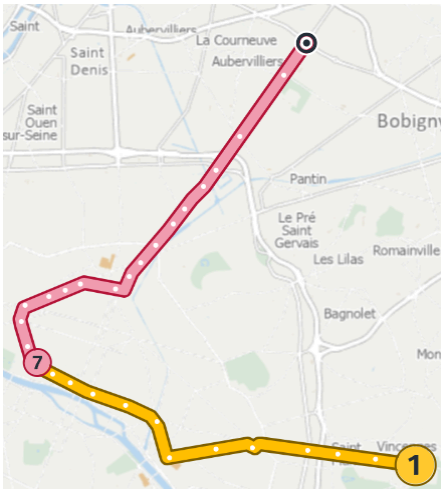
Second Trajet:

Pour le deuxième trajet (de Charles de Gaulle – Étoile à Mairie d'Issy), le site de la RATP propose un chemin différent de celui trouvé par notre programme. Il fait passer par Concorde avec un changement pour la ligne 12, alors que notre programme emprunte un autre itinéraire en passant dans un premier temps par la ligne 6 puis en allant sur la ligne 12. Le nombre d'arrêts est à peu près le même, mais les chemins ne sont pas identiques. C'est sûrement lié à la gestion du changement de ligne que l'on utilise qui ne représente pas exactement la réalité ou notre système exclut le fait de marcher ce que la RATP inclut dans quasiment tous ses trajets incluants plusieurs lignes



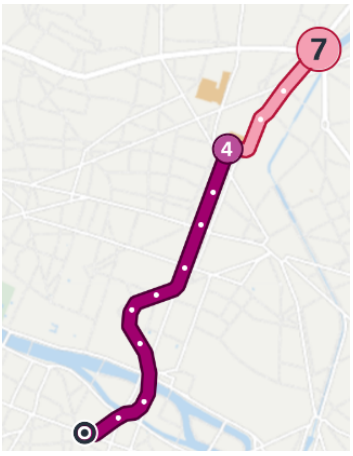
Troisième Trajet:

Sur ce trajet, la RATP commence par la ligne 1 puis fait un changement pour prendre la ligne 7. De notre côté, notre programme passe par la ligne 1 puis la ligne 2 puis la ligne 7 , ce qui donne un itinéraire plus direct. Ici encore, la RATP emprunte un chemin différent. Le malus de temps n'est surement pas réaliste dans le cas ou un changement de voie ne dure pas vraiment 10 minutes étant donné des horaires de métros.



Quatrième Trajet:

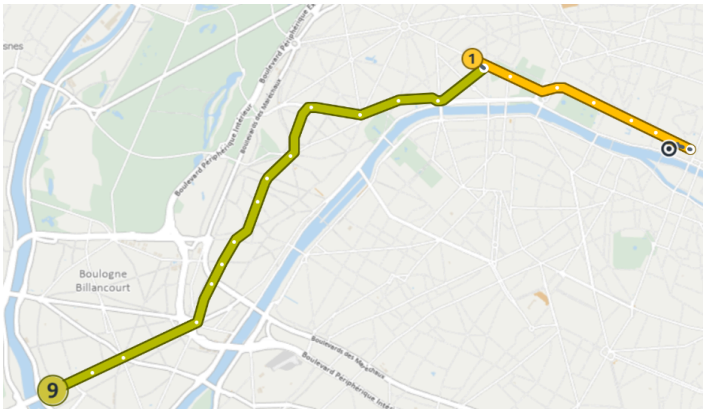
Sur ce trajet, la RATP propose un changement en passant de la ligne 7 à la ligne 4. Il s'agit ici du même chemin réalisé par la RATP et notre programme. Nous empruntons les mêmes arrêts ainsi que les mêmes lignes.



Cinquième Trajet:

Sur ce trajet, la RATP propose un changement en passant de la ligne 9 à la ligne 1. Contrairement au trajet proposé par la RATP, notre algorithme nous fait passer par la ligne 9, puis la ligne 10, suivi de la ligne 4 pour au final finir en passant par la ligne 7.

Encore une fois on estime que le changement de voie n'est pas assez réaliste comparé à ce que peut proposer la RATP avec des algorithmes plus aboutis prenants en comptes de nombreux détails afin de proposer un chemin qui sera au plus court en plus d'être au plus pratique avec seulement 2 lignes empruntées.



Conclusion:

Ce projet nous a permis de mieux comprendre la manière de représenter un graphe et d'appliquer des algorithmes classiques comme Bellman-Ford et Dijkstra à un cas concret. Même si notre système n'était pas parfait, il a quand même réussi à proposer deux trajets pertinents dans le métro parisien, ce qui montre que notre approche globale tenait la route. Cela dit, on a aussi constaté des écarts par rapport à la réalité, notamment à cause de certaines limites dans notre modélisation notamment en comparant avec les résultats observés sur le site officiel de la RATP.

Entre les deux algorithmes testés, Dijkstra s'est révélé plus efficace que Bellman-Ford la plupart du temps, quand le graphe était suffisamment fournis pour permettre à la différence de se creuser. Bellman-Ford reste néanmoins utile d'autant plus en prenant en compte les limitations de Dijkstra qui est limité à des graphes ne comportant que des arcs à poids positif.

L'ajout d'un malus pour les changements de ligne était une bonne idée pour se rapprocher du comportement réel d'un voyageur, mais il a aussi compliqué certains trajets, montrant à quel point il est difficile d'imiter exactement le fonctionnement d'un réseau comme celui de la RATP.

Au final, même si notre système reste perfectible, on est assez fiers d’avoir pu construire un outil qui fonctionne, qui s’appuie sur des concepts appris en cours, et qui donne des résultats plutôt crédibles.