Di Renzo—Viller Aurelio, Bouaoukel Walid

SAÉ 2.02 – Exploration algorithmique d'un problème

2 Représentation d'un graphe

Dans cette partie nous avons fait le code qui nous permet de manipuler un graphe et ses nœuds. Les nœuds des graphes seront représentés par leurs noms.

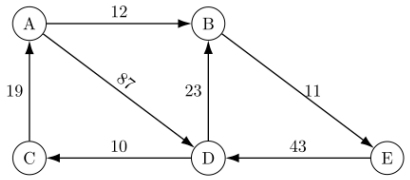
Nous créons la classe Arc, un objet composé d'un String dest, qui représente son nom et d'un double cout qui représente son cout.

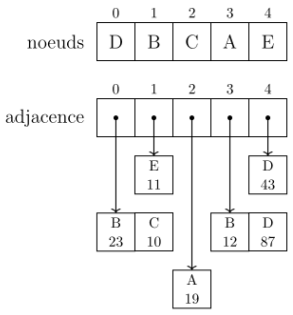
La classe Arcs regroupes sous forme de liste un ensemble d'objet de type Arc partant d'un nœud. Elle est composé des méthodes `void ajouterArc(Arc a)` et `List<Arc> getArcs()` pour gérer la liste.

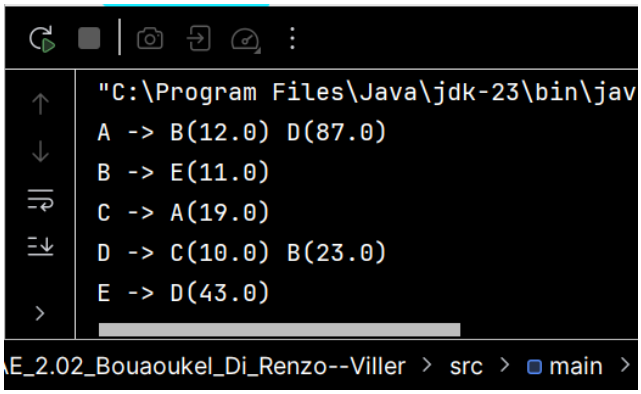
L'Interface Graphe qui nous permet d'implanter le TAD graphe. Elle est composé de 2 liste : `ArrayList<String> nœuds et ArrayList<Arcs> adjacence.`

La Classe GrapheListe implémente l'Interface Graphe, elle sert à représenter les données associer à un graphe.

Ensuite nous avons créé une classe main qui nous permet de représenter la figure 1 donné en exemple et l'affiché dans le terminal sous la forme demandée :







Pour finir sur cette partie nous avons créé des tests unitaires pour vérifier que le graphe est bien construit :

- Test_Arc() qui vérifie la création d'objets Arc.
- Test_Graphe() qui vérifie la création d'un objet Graphe qui est composé d'une liste d'Arc.

3 Calcul du plus court chemin par point fixe

Question 8 : Modalisation d'un algorithme de Point fixe.

Fonction pointFixe(Graphe g, Noeud depart) :
    noeuds ← listeNoeuds(g)
    i ← 0

    // Initialisation des distances et des parents
    Tant que i < taille(noeuds) faire
        n ← noeuds[i]
        L(n) ← +∞
        parent(n) ← null
        i ← i + 1
    Fin tant que

    L(depart) ← 0
    modif ← vrai

    // Boucle du point fixe
    Tant que modif est vrai ou faire
        modif ← faux
    i ← 0

    // Parcours de tous les nœuds
    Tant que i < taille(noeuds) faire
        x ← noeuds[i]
        voisins ← suivants(g, x)

        j ← 0

```
        // Parcours des voisins de x
        Tant que j < taille(voisins) faire
                arc ← voisins[j]
                n ← arc.noeud
                cout ← arc.cout
                valeur ← L(x) + cout

                // Mise à jour de la distance si un chemin plus court est trouvé
                Si valeur < L(n) alors
                        L(n) ← valeur
                        parent(n) ← x
                        modif ← vrai
                Fin si
                        j ← j + 1
        Fin tant que
                i ← i + 1
        Fin tant que
        Fin tant que
Fin
```

Lexique :

g : graphe, graphe orienté
depart : String, donne le nom du nœud de départ
noeuds: Liste de String, ce sont les élément du graphe.
L(n) : Int, valeur estimée du plus court chemin du nœud départ jusqu'à n.
parent(n) : String, le nœud précédant n sur le chemin trouvé.
voisin : arcs, sous forme de liste de couples <noeud, cout>.
modif : booléen, indique si une modification a été faite pendant une itération.
I et J : entier, indice d'itération

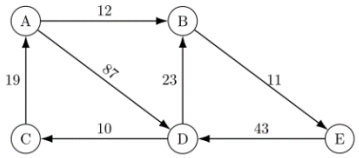
A l'aide de la classe Valeurs qui nous est fourni, nous avons créé une classe BellmanFord. Elle nous permet d'implanter l'algorithme du point fixe.

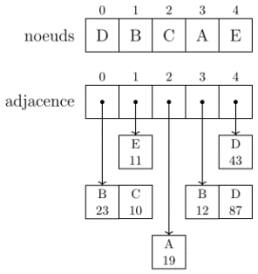
Question 10 :

Ensuite nous avons créé une class main qui applique l'algorithme du point fixe sur le graphe donné. Nous prenons comme point de départ C :

```
A -> B(12.0) D(87.0)
B -> E(11.0)
C -> A(19.0)
D -> C(10.0) B(23.0)
E -> D(43.0)

A ->  V:19.0 p:C
B ->  V:31.0 p:A
C ->  V:0.0 p:
D ->  V:85.0 p:E
E ->  V:42.0 p:B
```





Nous appliquons l'algorithme du point fixe pour vérifier le calcul du programme :

Sommet	A	B	C	D	E
Antécédent	C19	A12 D23		A87 E43	B11
Limite	∞	∞	0	∞	∞
	19	31	0	∞	42
	19	31	0	85	42
Point fixe	19	31	0	85	42

Nous retrouvons bien les mêmes valeurs calculer par l'algorithme, donc notre programme applique bien le calcul du point fixe.

Nous avons également crée un test, test_Bellman() qui vérifie que l'algorithme Bellman-Ford, appliqué à partir du sommet "A" d'un graphe orienté, renvoie les distances minimales correctes vers chaque autre sommet.

Ensuite nous avons écrit la méthode calculerChemin(String destination) dans la classe valeur pour retourner une liste de nœud passé en paramètre depuis le point de départ donnée précédemment.

En plus, nous avons créé en complément de cette méthode, la méthode static Object ToStringChemin(List<String> chemin) qui permet d'affiché dans le terminal le chemin parcouru étape par étape. Nous avons choisi comme point d'arrivé D :

```
A ->  V:19.0 p:C
B ->  V:31.0 p:A
C ->  V:0.0 p:
D ->  V:85.0 p:E
E ->  V:42.0 p:B

[D, E, B, A, C]

étape 1 : C
étape 2 : A
étape 3 : B
étape 4 : E
étape 5 : D
```

4 Calcul du meilleur chemin par Dijkstra

A partir de l'algorigramme fourni, nous l'avons traduit pour crée la class Dijkstra.

De la même manière que le test_Bellman(). Nous écrivons un test nommé test_Dijkstra qui vérifie que, depuis le sommet "A" d'un graphe orienté, l'algorithme renvoie les distances minimales correctes vers chaque autre sommet.

Comme pour avec la partie Bellman, nous avons ajouté du code dans notre class main pour tester le point de départ C :

```
A ->  V:19.0 p:C
B ->  V:31.0 p:A
C ->  V:0.0 p:
D ->  V:85.0 p:E
E ->  V:42.0 p:B

[D, E, B, A, C]
```

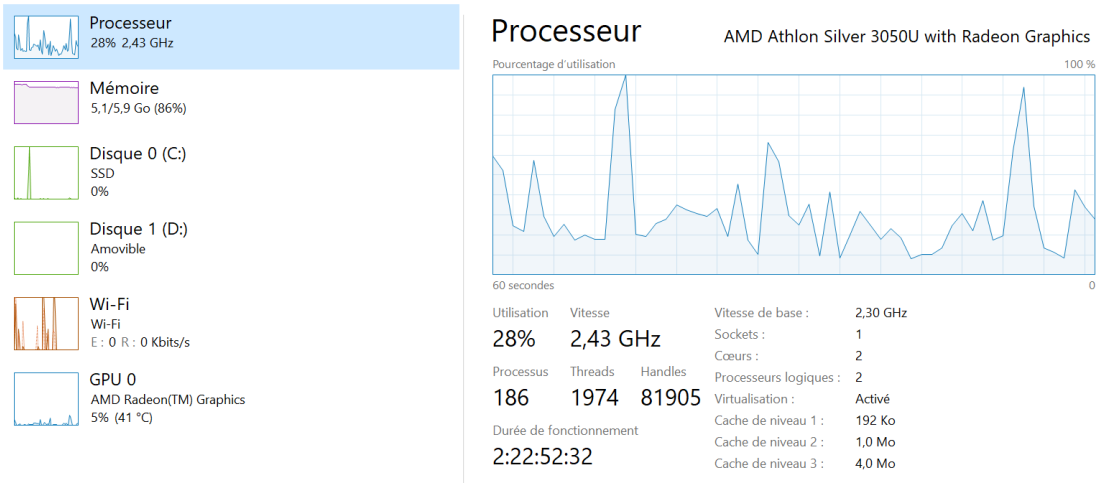
5 Validation et expérimentation

Nous avons ajouté un constructeur dans graphe liste qui prend en paramètre le nom du fichier contenant le descriptif.

Question 17 : Quel algorithme fonctionne le mieux expérimentalement ?

Pour cette question, nous avons créé une class mainComp permet de comparer les performances des algorithmes de plus court chemin avec 5 graphes.

Voici une capture d'écran des performances de l'ordinateur utilisé :

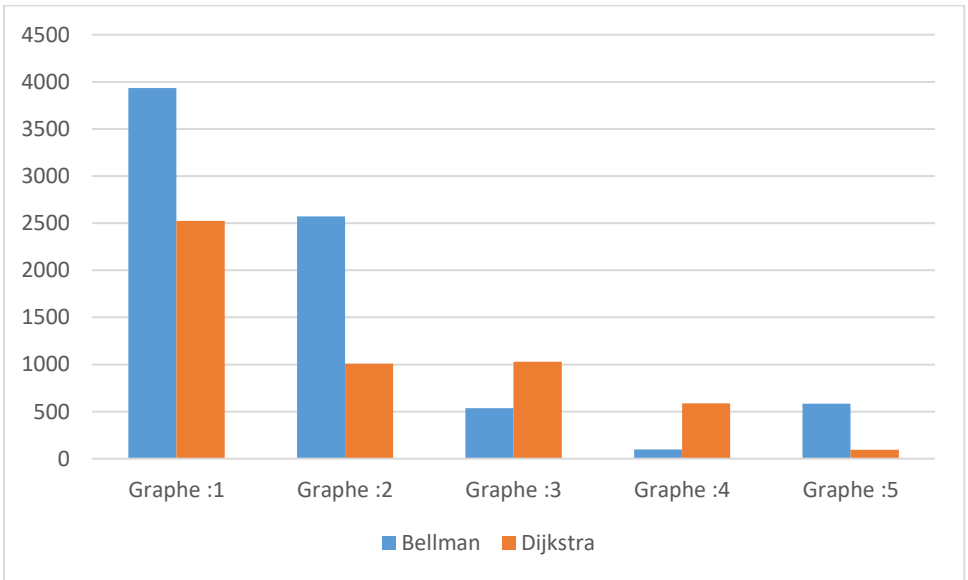


Dans cette expérience, nous avons comparé les performances des algorithmes de Bellman-Ford et de Dijkstra sur cinq graphes orientés différents. Pour chaque graphe, nous avons mesuré le temps d'exécution de chaque algorithme (en microsecondes) ainsi que les distances minimales calculées à partir d'un sommet de départ choisi.

Les résultats montrent que l'algorithme de Dijkstra est globalement plus rapide que celui de Bellman-Ford. Par exemple, dans le graphe 1, Dijkstra met 25 23 µs contre 39 34 µs pour Bellman-Ford. De manière générale, Dijkstra est le plus rapide dans trois graphes sur cinq (graphe 1, 2 et 5), tandis que Bellman-Ford est légèrement plus performant dans les deux autres (graphe 3 et 4), probablement en raison de la structure particulière de ces graphes.

Résultat obtenue :

	Graphe :1	Graphe :2	Graphe :3	Graphe :4	Graphe :5
Bellman	3934 µs	2572 µs	537 µs	99 µs	584 µs
Dijkstra	2523 µs	1010 µs	1029 µs	586 µs	95 µs



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

Dans cette partie, nous avons été amenés à modifier la classe Arc afin de pouvoir prendre en compte la ligne de métro à laquelle appartient chaque connexion entre deux stations. Pour cela, nous avons ajouté un nouveau champ ligne de type String à la classe Arc. Ce champ permet de stocker le numéro de la ligne associée à l'arc.

Nous avons réalisé une classe utilitaire nommée LireReseau. Cette classe contient une méthode statique Graphe lire(String fichier) qui permet de lire un fichier texte décrivant un réseau de métro, au format spécifié.

Ensuite, nous avons créé une classe MainMetro contenant une méthode main. Cette méthode sert à tester nos deux algorithmes de recherche de plus courts chemins (Bellman-Ford et Dijkstra) sur cinq trajets différents choisis dans le réseau de métro parisien.

Question 20 : tableau de comparaison :

Départ	Arrivée	Chemin	Temps calcul Bellman-Ford	Temps calcul Dijkstra
Blanche	**Assemblée Nationale**	**[Assemblée Nationale, Concorde, Madeleine, Saint-Lazare, Liège, Place de Clichy, Blanche]**	**12.9365 milliseconde**	**43.1825 milliseconde**
Charles de Gaulle, Étoile	**Richelieu Drouot**	**[Richelieu Drouot, Chaussée d'Antin, La Fayette, Havre Caumartin, Saint-Augustin, Miromesnil, Saint-Philippe du Roule, Franklin D. Roosevelt, George V, Charles de Gaulle, Étoile]**	**12.5186 milliseconde**	**9.1832 milliseconde**
Bastille	**Saint-Lazare**	**[Saint-Lazare, Havre Caumartin, Opéra, Pyramides, Châtelet, Hôtel de Ville, Saint-Paul, Le Marais, Bastille]**	**22.8299 milliseconde**	**8.8803 milliseconde**
Gare de Lyon	**Filles du Calvaire**	**[Filles du Calvaire, Saint-Sébastien-Froissart, Chemin Vert, Bastille, Gare de Lyon]**	**38.9879 milliseconde**	**9.3981 milliseconde**
Nation	**Louvre, Rivoli**	**[Louvre, Rivoli, Châtelet, Gare de Lyon, Reuilly Diderot, Nation]**	**6.0847 milliseconde**	**12.2435 milliseconde**

Après, nous avons modifié nos algorithmes pour tenir compte d'une pénalité liée au changement de ligne. Une seconde version des algorithmes à vue le jour, appelée resoudre2. Elle ajoute une pénalité de temps à chaque changement de ligne. Cette pénalité est appliquée lorsqu'on passe d'un arc à un autre avec un numéro de ligne différent.

Enfin, dans la méthode main de la classe MainMetro, nous avons complété notre programme pour avoir le choix d'exécuter les 5 trajets en utilisant les algorithmes qui pénalise ou non.

Question 23 – **Comparaison des trajets calculés avec ceux proposés par la RATP**

Dans cette étape, nous avons repris les cinq trajets de la question 20 et nous les avons comparés aux itinéraires proposés par l'application officielle de la RATP. Nous avons configuré l'application pour ne prendre en compte que le métro.

Comparaison :

Test 1 : Blanche → Assemblée Nationale :

- **Chemin calculé (Dijkstra & Bellman)** : Assemblée Nationale → Concorde → Madeleine → Saint-Lazare → Liège → Place de Clichy → Blanche.
- **Chemin RATP** : Plus direct, en empruntant la ligne 12.

Test 2 : Charles de Gaulle – Étoile → Richelieu Drouot

- **Chemin calculé** : Richelieu Drouot → … → Charles de Gaulle – Étoile
- **Chemin RATP** : Plus direct, via la ligne 1 ou la 9, sans autant de changements.

Test 3 : Bastille → Saint-Lazare

- **Chemin calculé** : Saint-Lazare → … → Bastille
- **Chemin RATP** : Plus long via la ligne 1 puis ligne 5.

Test 4 : Gare de Lyon → Filles du Calvaire

- **Chemin calculé** : Gare de Lyon → Bastille → Chemin Vert → Saint-Sébastien-Froissart → Filles du Calvaire.
- **Chemin RATP** : Similaire, passage sur ligne 8

Test 5 : Nation → Louvre – Rivoli

- **Chemin calculé** : Nation → Reuilly Diderot → Gare de Lyon → Châtelet → Louvre – Rivoli.
- **Chemin RATP** : Plus simple, passage par la ligne 1 directe.

On constate que les chemins proposés par nos algorithmes sont généralement valides, mais dans plusieurs cas, ils comportent plus de correspondances ou suivent des itinéraires moins directs que ceux de l'application RATP. Ces écarts peuvent s'expliquer principalement par le fait que nos algorithmes cherchent à minimiser une somme de temps, sans intégrer d'éléments comme la simplicité du trajet.

Bilan de notre SAE

Au terme de cette SAE consacrée à l'exploration algorithmique et à la recherche de plus courts chemins dans un graphe plusieurs enseignements se dégagent.

1. Concepts théoriques

Étudier et coder les algorithmes de Dijkstra et Bellman-Ford nous a permis de mieux comprendre leurs différences, leurs points forts.

Nous avons consolidé notre compréhension des graphes orientés, en particulier en représentant des stations de métro sous forme de nœuds et leurs liaisons comme des arcs.

2. Développement de compétences pratiques

Un autre aspect pratique important a été la lecture et le traitement de fichiers texte pour construire notre graphe du métro. Cela nous a familiarisés avec la gestion de fichiers, la manipulation de données et le traitement des erreurs.

3. Les difficultés rencontrées

Nous avons aussi été confrontés à des défis. Comprendre la complexité algorithmique pour ajouter des pénalités s'est révélé plus compliqué que prévu.

Travailler en binôme n'a pas toujours été simple non plus : il a fallu apprendre à bien nous répartir les tâches de manière efficace.

Conclusion :

Cette SAE nous a permis de lier théorie et pratique de manière concrète. La mise en œuvre des algorithmes de Dijkstra et Bellman-Ford sur des graphes complexes a contribué à développer davantage notre capacité à raisonner sur les algorithmes.

Elle nous a également appris à nous organiser et à prendre en compte les spécificités d'un problème réel avec le fonctionnement du métro parisien.

En somme, cette SAE nous a rappelé que derrière chaque solution algorithmique, il y a une logique à comprendre et des choix à justifier.