Drouvot Melody S2A
Duhoux Olivier

SAE 2.2 Graphe
Exploration algorithmique d'un problème

Représentation d'un graphe

Tout d'abord, nous avons écrit la classe Arc permettant de représenter un arc avec son coût et son noeud de destination puis par la suite la classe Arcs qui permet de créer des listes d'objets arc correspondants aux arcs partant d'un noeud. Ces deux classes disposent de getters.

La classe TestArc permet de tester le bon fonctionnement de la construction d'un Arc, l'utilisation des getters et aussi le bon renvoi d'une exception lors de la tentative de création d'un Arc avec un coût négatif.

La classe TestArcs vérifie le fonctionnement du constructeur de Arcs, sa méthode d'ajout d'Arc et son getter.

Ensuite, afin de pouvoir représenter des graphes nous avons écrit la classe GrapheListe qui implémente notre interface Graphe. Cette classe permet de représenter un graphe via une liste de noeuds (String représentant le nom de chaque noeud) et une liste d'arcs (Arcs représentant les listes d'arcs partant de chaque noeud).
La méthode ajouterArc permet d'ajouter un arc au graphe en prenant deux noeuds (String) et un coût (réel) en paramètres. Si un des noeuds n'est pas déjà présent dans le graphe, il est ajouté avant d'ajouter l'arc.
La méthode suivants permet à partir d'un noeud (String) donné en paramètre de récupérer la liste des arcs partant de ce noeud.
La méthode toString affiche tous les noeuds suivis de leurs successeurs ainsi que les coûts des arcs y menant.

La classe de tests TestGraphe vérifie le fonctionnement d'ajout d'arcs, de la méthode suivants et des getters. Nous avons aussi écrit des tests vérifiant que la construction d'un graphe à partir d'un fichier qui fonctionne correctement mais aussi des tests qui vérifient que les tentatives d'ouvertures de fichiers incorrects ou n'existant pas génèrent bien des exceptions.

Calcul du plus court chemin par point fixe

Question 8 :
En utilisant le TAD fourni, écrire l'algorithme de la fonction point Fixe(Graphe g InOut, Noeud départ) qui modifie les valeurs (parent et distance) associées aux nœuds du graphe pour trouver le chemin le plus court partant du nœud de départ passé en paramètre (en n'oubliant pas le lexique).
On supposera qu'on peut directement changer les valeurs de L(X) et parent(X)

par une simple affectation comme L(X) ← +∞

Notre algorithme de point fixe en pseudo code :

```
fonction PointFixe(Graphe g InOut, Noeud depart)
début
pour chaque sommet u de s faire
        u.valeur <- infini
        u.parent <- indefini
fin pour
depart <- 0
pour i de 1 à tailles(S) -1 faire
        pour chaque arc(u,v) < v.valeur alors
                si u.valeur + poids(u,v) < v.valeur alors
                        v.valeur <- u.valeur + poids(u,v)
                        v.parent <- u
                fin si
        fin pour
fin pour
retourner g
```

Lexique :
g : Graphe, le graphe à résoudre
S : Liste, liste des sommets du graphe g
A : Liste, liste des arcs du graphe g
depart : Noeud, sommet de départ

Ensuite nous avons dû traduire cet algorithme en code Java dans la classe BellmanFord. Cette classe possède donc la méthode résoudre, prenant en paramètre un Graphe g et un sommet de départ. Et retourne un objet valeur contenant le graphe correctement construit avec les chemins les plus courts vers chaque sommet.

Par la suite, nous avons dû écrire un Main qui applique l'algorithme de point fixe, en créant donc un GrapheListe et lui ajoutant les arcs qui le composent. Sur ce graphe nous avons donc utilisé l'algorithme de BellmanFord avec comme noeud de départ le noeud "A".
Les valeurs renvoyées :
A -> V:0.0 p:null
B -> V:12.0 p:A
C -> V:76.0 p:D
D -> V:66.0 p:E
E -> V:23.0 p:B

correspondent à ce que nous avons calculé dans le module Graphe.

Pour les tests nous avons testé de nombreux cas possible, d'abord la méthode résoudre normalement, puis sur un graphe vide et enfin sur un graphe possédant une boucle.

Pour calculer le chemin le plus court, nous avons dû écrire la méthode List<String> calculerChemin(String destination) qui permet dans un objet Valeur de trouver le chemin le plus court vers le sommet de destination. Si le sommet demandé n'est pas dans le graphe, alors la valeur des autres sommets sera +infini.

Calcul du plus court chemin par Djikstra

Nous avons en premier lieu dû traduire l'algorithme fourni en code Java, nous avons eu certaines difficultés pour y parvenir. Une fois cela fait nous avons pû écrire les tests sur cette classe. Nous avons testé en premier si la méthode résoudre fonctionnait, puis différents cas: si le graphe est vide, si le sommet demandé est en fin du graphe, si le graphe possède une boucle, et la méthode calculerChemin sur ce graphe.

Pour le MainDjikstra nous avons pu montrer l'utilisation de ces algorithmes avec le calcul de différents chemins les plus courts.

Validation et expérimentation

Nos résultats montrent que l'algorithme de Djikstra est plus efficace que l'algorithme de BellmanFord. En les utilisant sur la centaine de graphes fournis, le temps d'exécution de Djikstra est de 9.34 secondes contre 9.5 pour BellmanFord, la différence est plutôt faible. On peut supposer que l'algorithme de Djikstra est plus rapide car il fait moins d'itérations pour trouver le chemin le plus court vers un sommet.

Le constructeur GrapheListe(String nomfichier) permet de grandement simplifier les tests, que ce soit pour vérifier les algorithmes ou bien pour tester leurs performances. Grâce à cela nous avons pu boucler sur un répertoire de graphes pour trouver l'algorithme le plus rapide des deux.

Conclusion

Cette SAE nous a permis d'apprendre différentes choses. En premier, la partie mathématique nous a permis de découvrir un algorithme plus efficace que celui de BellmanFord. La partie code et implémentation Java nous as fait réfléchir a comment traduire efficacement un algorithme en pseudo code en code Java. L'algorithme de BellmanFord nous a posé moins de difficultés que l'algorithme de Dijkstra. C'est la partie qui nous as pris le plus de temps afin de vérifier que le code fonctionnait bien.

Grâce à GitHub nous avons pu travailler en même temps sur différentes parties du projet. Nous nous sommes réparti les tâches, Olivier faisait l'algorithme de BellmanFord tandis que Melody faisait l'algorithme de Dijkstra. Si l'un de nous deux avait un problème, l'autre pouvait venir l'aider à tout moment. Celà nous a permis de gagner du temps sur cette SAE.