Rapport SAE Exploration algorithmique S2.2

- FUCHS Thomas
- COMTE Gabriel

Partie 1 - Représentation d'un graphe

Dans cette première partie nous nous sommes familiarisés avec la représentation d'un graphe sous la forme d'un type abstrait de donnée (TAD).

Pour cela nous avons tout d'abord développé plusieurs classes qui ont permis de modéliser les différents constituants d'un graphe, à savoir :

-la classe Arc caractérisée par ses attributs d'instance *destination* et *cout*

-l'interface Graphe (que nous avons par la suite décidé d'altérer afin de faire intervenir un labyrinthe qui correspond à un graphe un peu spécial).

-la classe Arcs qui est simplement une liste contenant plusieurs objets de type Arc.

-la classe GrapheListe qui implémente l'interface Graphe, permet de gérer un graphe en lui ajoutant des sommets (nœuds), arêtes (arcs) et une pondération (cout).

-la classe Main où est instancié un objet de type GrapheListe et qui permet d'afficher le graphe exemple de la figure 2 du sujet.

Pour cette partie, les tests ont uniquement consisté en une vérification de la bonne construction d'un graphe avec les classes :

-ArcsTest

-ArcTest

-GrapheListeTest

Partie 2 – Calcul du plus court chemin par point fixe

Algorithme de Bellman-Ford

Durant cette seconde partie nous avons dans un premier temps écrit l'algorithme de Bellman-Ford, puis nous l'avons testé dans le programme main de la classe Main précédemment créée.

Afin de tester l'algorithme de Bellman-Ford nous avons programmé une classe de tests :

AlgorithmeTest qui teste les deux algorithmes par le biais d'un tableau d'interface Algorithme.

Réponse à la question 8 :

Algorithme de point fixe :

```
fonction resoudre(Graphe g InOut, Noeud depart)

  Pour chaque noeud n dans g
    v.setValeur(n, +∞)
    v.setParent(n, null)
  FinPour

  v.setValeur(depart, 0)

  fixe <-- vrai
```

```
    TantQue fixe

        fixe <--faux

        Pour chaque noeud n dans g

            Pour chaque voisin m de n

                Si v.getValeur(n) + poids(n, m) < v.getValeur(m)

                    v.setValeur(m, v.getValeur(n) + poids(n, m))

                    v.setParent(m, n)

                FinSi

            FinPour

        FinPour

    FinTantQue

    Retourner v

FinFonction
```

Lexique :

v : Valeur, La classe Valeur est conçue pour représenter les fonctions L(X) et parent(X), où X est un nœud.

Elle associe des valeurs réelles et des chaînes de caractères représentant les parents à des nœuds donnés (sous forme de chaînes de caractères).

C'est également la valeur de retour de la méthode, qui affiche le point minimal.

g : Graphe InOut, il s'agit du graphe qui est passé en paramètre de la méthode résoudre, il contient la liste des noeuds et la liste des couples de valeurs (suivants, cout).

depart : String, représente le sommet de départ du graphe sous forme de chaine de caractères et est initialisé à 0

n : noeud, represente le noeud courant dans la liste de l'objet graphe g, il est sous forme de chaine de caractères

m : noeud voisin de n, c'est le sommet adjacent qui est directement accessible à partir du noeud n via un arc

poids : int, poids représente la longueur de l'arête qui relie n et m.

En java ce n'est pas une fonction, il est implémenté différemment au travers d'un getteur d'un objet de type Arc.

Partie 3 – Calcul du meilleur chemin

Algorithme de Dijkstra

Dans cette troisième partie nous avons implémenté l'algorithme de Dijkstra.

Il était donné dans le sujet et nous l'avons reporté dans la classe *Dijkstra.java,* puis juste en-dessous nous l'avons programmé.

Classes créées :

-Dijkstra classe dans laquelle se trouve l'algorithme

-MainDijkstra classe dans laquelle on utilise cet algorithme

Pour les tests unitaires, nous utilisons toujours la classe AlgorithmeTest dans laquelle nous testons par exemple :

-Le comportement d'un graphe vide

-Le comportement d'un graphe qui n'a qu'un seul arc

-Les pattern de sommets linéaires et circulaires

-Le backtracking vu en cours de graphes

-La résolution du plus court chemin

-La liste des nœuds et des voisins ainsi que la pondération des arcs

Partie optionnelle sur les labyrinthes

Comme il nous a été fourni sur arche des fichiers labyrinthes nous avons décidé de les utiliser quand bien même le sujet ne le demandait pas forcément afin de rajouter une expérimentation. Ce supplément d'informations nous a permis d'en arriver à la conclusion que dans le cas d'un parcours des algos dans un labyrinthe, qui est sensiblement un graphe non orienté, nous avons trouvé que l'algorithme de Bellman-Ford était plus

rapide. A l'inverse, l'algorithme de Dijkstra est plus efficace dans le cadre d'un graphe orienté. Les résultats seront mis plus en évidence dans la partie 4.

Pour mettre en place cette expérience, nous avons volontairement retiré l'interface Graphe car elle n'avait pas d'utilité dans notre conception du projet et l'avons remplacé par la classe abstraite buildListe qui permet de tirer profit d'héritage plutôt que d'une interface et donc de définir les méthodes directement dans les classes GrapheListes et LabyListes qui chargent les fichiers textes.

Partie 4 – Validation et expérimentation

Cette dernière partie est consacrée à la présentation des résultats de vitesse et d'efficacité des algorithmes de Dijkstra et Bellman-Ford.

Question 16 :

```
Temps d'execution de réolution des différents graphes de type : sae/src/main/resources/Graphes/avec l'algo de :class algorithme.algo
rithmes.Dijkstra 10608 ms

Temps d'execution de réolution des différents graphes de type : sae/src/main/resources/Graphes/avec l'algo de :class algorithme.algo
rithmes.BellmanFord 14888 ms
```

Question 17 :

On s'aperçoit que l'algorithme de Dijkstra est plus efficace temporellement de part sa complexité algorithmique qui est différente de celle de Bellman-Ford : O(nlogn +l) contre $O(n^2)$.

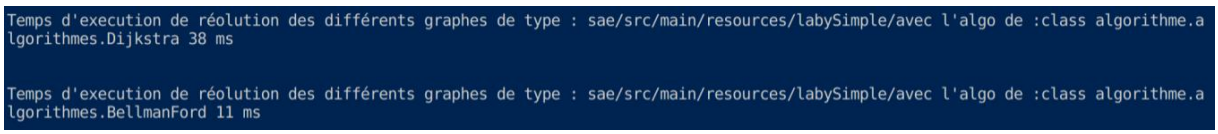
L'algorithme de Dijkstra fonctionne mieux lorsqu'un graphe est pondéré positivement et ne contient pas de cycles de poids négatifs.

Question 18 :

Dans la plupart des cas, si le graphe est orienté et à des valuations positives ou nulles, c'est Dijkstra qui sera meilleur. Toujours est-il que même si Bellman-Ford peut se montrer plus long dans le temps dans le cas de graphes denses avec beaucoup d'arêtes, il est généralement plus

approprié de manière générale de part sa détection de cycles de poids négatifs, là où Dijkstra se perd complètement.

On pourra aussi ajouter que grâce aux tests sur les labyrinthes, on voit que Bellman-Ford s'en sort mieux que Dijkstra :



Conclusion :

Lors de cette SAE nous avons pris connaissance d'un nouvel algorithme de recherche du plus court chemin que celui vu dans le module de maths.

Les difficultés rencontrées tournaient surtout autour des tests unitaires qui échouaient car nous n'avons pas suffisamment vérifié les valeurs qui pouvaient être inexistantes (null) ce qui engendrait l'exception NullPointer de Java.

Ce travail nous a permis de mieux comprendre le fonctionnement des algorithmes du plus court chemin et des graphes de manière générale, surtout quand on les applique à des cas concrets.

Thomas, s'est chargé de l'implémentation des graphes et des algorithmes en Java. Il a écrit quelques tests unitaires et a fait le rapport.

Tandis que Gabriel, s'est penché sur l'écriture de nombreux tests et du debugging des algos. Il a également fait la javadoc en entier et les commentaires du code. Il a par la suite créé les classes qui gèrent la liste des graphes et des labyrinthes, pour finir il a expérimenté les algos dans les différents cas de figure.