Benjamin SCHEFFER,
Baptiste DELABORDE

SAE2.02

Exploration Algorithmique
Recherche de plus court chemin dans un graphe

Benjamin SCHEFFER,
Baptiste DELABORDE

1. **Représentation d'un graphe :**

Dans cette première partie nous avons créé la classe Arc qui permet de créer un arc, il est représenté par sa destination et sa valeur.

Dans un deuxième temps nous avons créé une classe Arcs qui est utilisé pour regrouper les arcs dans une même liste afin de pouvoir attribuer une liste d'arc a un nœud dans un graphe.

Ensuite nous avons fait la classe GrapheListe qui est composé d'une liste de nœuds et d'une liste d'Arcs cette seconde liste représente tous les arcs qui partent du nœuds auquel il est attribué. Lors de la création de cette classe nous avons dû faire une méthode qui s'appelle ajouterArc, c'est la méthode qui nous a pris le plus de temps lors de cette partie car elle était conséquente, elle a donc nécessité plus de temps et plus de réflexion. Ce qui a rendu cette partie un peu plus compliquée était le fait de bien comprendre comment fonctionner cette classe avec sa liste de nœuds et sa liste d'adjacent (Arcs).

Après cela nous avons écrit un main qui permet de représenté un graphe et ses arcs, ce main représente le graphe de manière à écrire le nœuds en suite une flèche et les arc associé à ce nœuds.

Et pour conclure cette partie on écrit plusieurs tests afin de vérifier que le graphe se construisait bien, donc pour cela on a vérifié que les nœuds étaient bien ajoutés à la liste de nœuds et nous avons vérifié que les arcs c'était bien ajouter à la liste d'arc de chaque nœud.

2. **Calcul du plus court chemin par point fixe**

Dans cette seconde partie de la SAE nous avons dû écrire l'algorithme de la méthode point fixe. Cet algorithme permet de voir à partir d'un nœud du graphe le chemin le plus court qui relis ce nœud aux autres.

<u>Question 8 :</u>

```
fonction pointFixe(Graphe g InOut,Noeud depart)
Début
        pour i = 0 i < g.listeNoeud().size() i ++
                si(L(i) = depart)alors
                        l(g.listeNoeud().get(i))<=0;
                sinon
                        L(g.listeNoeud().get(i)) <= +l'infinit
                fsi
        fpour
        boolean arret <- false
        tant que nonarret faire
                arret <- true
                pour i = 0; i<g.listeNoeud().size(); i ++ faire

                        pour j = 0 ; j<g.listeNoeud().get(i).size() ;j++ faire
                                noeud <-g.suivants(g.listeNoeud().get(i)).get(j).getDestination());
                                int min = l(noeud);
                                l(n)=l(g.listeNoeuds().get(i))+g.suivants(ListeNoeuds.get(i)
                                        .getDestination()).get(j).getCout())

                                si l(n) < min alors

                                        l(g.suivants(ListeNoeuds.get(i).getDestination()).get(j)
                                        .getDestination())<- l(n)
                                        Parent(g.suivants(ListeNoeuds.get(i).getDestination()).get(j)
                                        .getDestination())<=l(g.listeNoeuds().get(i))
                                        arret <= faux
                                fsi
                        fpour
                fpour
```

```
        ftant que

fin

Lexique :
l(x):double permet d'avoir la valeur d'un noeud
Parent (x) :Noeud permet d'avoir le noeud parent du noeud x
arret : boolean condition d'arret de la boucle tant que
ListeNoeud() : list liste des noeuds du graphe
suivants(x) : list liste des noeuds adjacents du noeud
min : entier valeur minimal des adjacent d'un noeud
fin lexique
```

Cet algorithme nous a permis de réaliser la méthode de point fixe de la classe BellmanFord, cette méthode est une adaptation en java de l'algorithme que nous avons effectué précédemment. Puisque nous avons écrit l'algorithme avant de réaliser la méthode cela été donc beaucoup plus simple, ce qui nous a permis de réaliser cette partie sans trop de difficulté.

Après avoir écrit cette classe nous avons créé un main qui s'appelle MainBellmanFord, ce main permet d'afficher le graphe, le nœuds de départ et le chemin le plus court qui permet d'aller d'un point à un autre.

Puis nous avons effectué plusieurs tests afin de vérifier que ce soit les bonnes valeurs pour chaque sommet et que le chemin qui permet d'aller d'un point à un autre soit également le bon.

3. Calcul du meilleur chemin par Dijkstra

Pour cette troisième partie on écrit l'équivalent de la méthode de pont fixe de Bellman Ford mais avec la méthode de Dijkstra. L'algorithme nous a était donné donc nous avions juste à l'adapter en java. Par conséquent cette partie ne nous a pas posé énormément de difficultés.

De la même manière que pour la méthode de point fixe nous avons réalisé un main et une suite de tests, le main permet d'afficher le graphe, le nœud de départ, le chemin le plus court qui permet d'aller d'un point à un autre. Et tout cela en partant de plusieurs points différents. Pour les tests nous effectuons exactement les mêmes que pour le point fixe mais adapter pour cette méthode.

4. Validation et expérimentation

Question 16 :
Pour vérifier quelle méthode est la plus efficace nous avons testé ces méthodes sur plusieurs graphes à la suite et nous avons calculé le temps que chaque méthode mettrai pour les résoudre. Pour la méthode de Bellman Ford nous avons obtenu environ 60 secondes pour résoudre tous les graphes et pour la méthode de Dijkstra nous obtenons environ 30 secondes pour résoudre ces graphes.

Question 17 :
D'après les résultats obtenus on constate que la méthode de Dijkstra résous le même nombre de graphe que pour la méthode de Bellman Ford en deux fois moins de temps, on obtient cette différence car la méthode de Dijkstra fait beaucoup moins d'itération que la méthode de Bellman Ford.

Benjamin SCHEFFER,
Baptiste DELABORDE

Question 18 :

La méthode la plus efficace est la méthode de Dijkstra car pour calculer la complexité de la méthode de Bellman Ford il faut faire (sommet x arrêtes) alors que la complexité de la méthode de Dijkstra la complexité vaut (sommet + arrêtes) x ln(sommet) < (sommet x arrêtes). Donc la méthode de Dijkstra effectue moins d'itération que la méthode de point fixe. C'est pour cela que la méthode de Dijkstra est plus efficace.

Conclusion de cette SAE :

Cette SAE nous a permis d'amélioré notre compétence de travail en groupe, elle nous a également permis d'amélioré notre organisation tant dans la répartition des taches mais aussi l'organisation de nos classes et pour finir elle nous a aussi permis de bien comprendre le fonctionnement des méthodes de Dijkstra et de point fixe.