SAE2.02

Exploration Algorithmique Recherche de plus court chemin dans un graphe

1. **Représentation d’un graphe :**

Dans cette première partie nous avons créé la classe Arc qui permet de créer un arc, un arc 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 écris 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.

1. **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.

Question 8 :

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 pointfixe de la classe BellmanFord, cette méthode est une adaptation en java de l’algorithme que nous avons effectué précédemment. Puisque nous avons écris 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 créer 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.

1. **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 réaliser 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 a un autre. Et tout ce la en partant de plusieurs points différents. Pour les tests nous effectuons exactement les mêmes que pour le point fixe mais adapter pour cette méthode.

1. **Validation et expérimentation**

Question 16 :

Lorsque l’on effectue la méthode de point fixe en partant du point A et avec le graphe :

A -> B(12) D(87)

B -> E(11)

C -> A(19)

D -> B(23) C(10)

E -> D(43)

On obtient les valeurs :

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

Et Lorsque l’on effectue avec le même point de départ et le même graphe la méthode de Dijkstra on obtient :

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

On peut donc constater que peut importe la méthode choisis les résultats obtenus sont les mêmes. Ce qui est normal car ces deux méthodes effectuent la même chose mais avec deux méthodes différentes.

Question 17 :

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 log(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.