Dieudonné Quentin
Herrmann Vivien

Rapport de SAE 2.2 :
Exploration Algorithmique Recherche de plus court chemin dans un graphe

I) Représentation d'un graphe

Question 1 :

Nous avons commencé par créer une classe Arc représentant un Arc à travers dest : le nœud destination d'un Arc donné et coût : le coût de l'Arc. Pour gérer ces attributs de manière efficace nous avons choisi de créer un constructeur et des getters.

Question 2 :

Nous avons ensuite créé une classe Arcs qui gère la bonne construction et le bon fonctionnement des liste d'Arcs. En effet les Arcs sont stockés dans l'attribut, construits et contient des méthodes permettant de le modifier (méthode ajouterArc) et y accéder de manière flexible (méthode getsArcs).

Question 3 :

Nous avons créé une interface Graphe qui est implémentée dans GrapheListe. Cette interface contient différentes méthodes :
une méthode getIndice, une methode ajouterArc, une methode getAdjacence ainsi qu'une méthode toString

Question 4 :

Nous avons ensuite créé la classe GrapheListe permettant de représenter un graphe sous forme de liste d'Adjacence. De cette manière, elle permet d'ajouter et de gérer des nœuds et des arcs de manière dynamique. Nous avons notamment conçu une méthode ajouterArc qui vérifie que pour un arc donné (sachant le nœud de départ et de destination et le coût de l'arc) il existe bien et sinon l'ajoute et met à jour la liste d'Arcs du nœud de départ.

Question 5 :

Nous avons créé une classe Main contenant un main qui déclare un graphe et ajoute des arcs à ce graphe en utilisant la méthode ajouterArc.

Question 6 :

Après avoir créé un main dans la classe Main nous avons créé une méthode toString dans la classe Graphe qui permet de représenter le graphe sous une forme simplifiée en donnant les nœuds adjacents pour chaque nœud du graphe. Et nous avons ensuite utilisé cette méthode dans le main pour générer la représentation dans la console.

Question 7 :

Nous avons ensuite créé une classe de test avec une méthode de test pour tester la bonne construction du graphe. Nous initialisons le graphe, ajoutons les arcs puis vérifions les indices pour des nœuds donnés et vérifions l'adjacence et les coûts des arcs.

II) Point fixe

Question 8 :

Nous avons écrit l'algorithme de la fonction pointfixe qui permet de trouver le point fixe en mettant à jour les valeurs de chaque nœud tant que tous les nœuds ne sont pas à jour. Afin de reconstruire le chemin optimal, il stocke pour chaque nœud son parent, c'est-à-dire le nœud précédemment utilisé pour atteindre ce nœud avec la meilleure valeur. Chaque fois qu'une meilleure valeur est trouvée pour un nœud, le parent de ce nœud est mis à jour.

Question 9 :

Nous avons traduit l'algorithme en java pour construire la méthode résoudre dans la classe BellmanFord. Nous avons initialisé les distances et les parents pour chaque nœud du graphe. Ensuite, nous avons itéré sur tous les nœuds et leurs arcs pour mettre à jour les distances et les parents tant qu'il y avait des changements. Enfin, nous avons retourné un objet Valeur contenant les distances et les parents après convergence.

Question 10 :

Nous avons ajouté une partie dans la Classe Main pour tester d'avoir le bon affichage en résolvant l'algorithme de Bellman Ford grâce à la méthode résoudre de la classe Bellman Ford en reprenant le graphe utilisé pour tester le bon affichage du graphe dans la console. Ici nous affichons les plus courts chemins jusqu'à tous les nœuds du graphe en fournissant le graphe et le nœud de départ.

Question 11 :

Nous avons ensuite ajouté une méthode de test à la classe TestGraphe, testPointFixe testant le bon fonctionnement et la véracité des résultats d'exécution de la méthode. Nous avons vérifié les valeurs des plus courts chemins et vérifié les nœuds parents pour chacun des nœuds du graphe.

Question 12 :

Nous avons créé une méthode calculerChemin dans la classe Valeur qui permet de remonter la parenté d'un noeuds tant qu'il existe un parent (avec getParent). Cette méthode nous permet d'obtenir le chemin du noeuds de départ jusqu'au noeuds passé en paramètre.

III) Dijkstra

Question 13 :

Nous avons élaboré une méthode resoudre de la classe Dijkstra à partir de l'algorithme donné. Pour cela nous avons bien compris le fonctionnement de cet algorithme, il commence par l'initialiser des sommets, la fixation du point de départ à 0, extraction du sommet avec la valeur minimale de la liste des nœuds à traiter, réévaluation des valeurs et parents des voisins pour chaque sommet traité, répétition du processus jusqu'à ce que la liste des nœuds à traiter soit vide.

Question 14 :

Nous avons ensuite ajouté une méthode de test à la classe TestGraphe, testDijkstra testant le bon fonctionnement et la véracité des résultats d'exécution de la méthode. Nous avons vérifié les valeurs des plus courts chemins et vérifié les nœuds parents pour chacun des nœuds du graphe.

Question 15

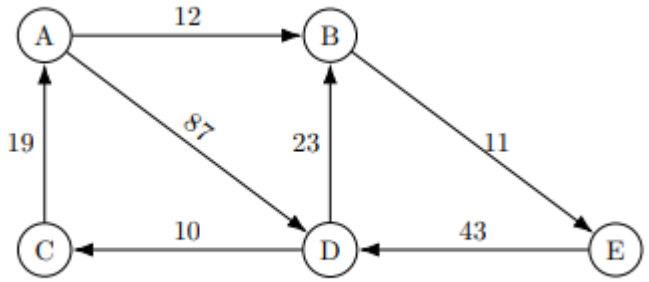
Ensuite, nous avons procédé à la création de la Classe MainDijkstra qui consiste à partir d'un graphe donné de calculer le chemin le plus court en s'appuyant sur la méthode résoudre de la classe Dijkstra et ensuite à afficher les chemins pour des nœuds donnés

IV) Validation

Question 16 :

<u>Algorithme de Bellman Ford:</u>

Nous avons utilisé l'algorithme de Bellman Ford pour calculer le plus court chemin de ce graphe:



Le plus court chemin obtenu est celui-ci:

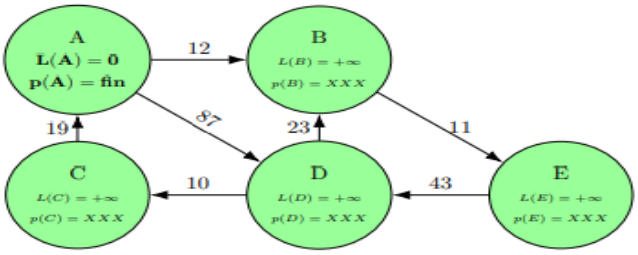
A -> V:0.0 p:null
B -> V:12.0 p:A
C -> V:76.0 p:D
D -> V:66.0 p:E
E -> V:23.0 p:B

durée d'exécution: 1200711 nanosecondes

avec p représentant le parent de chaque noeud

Algorithme de Dijkstra:

Nous avons utilisé l'algorithme de Dijkstra pour calculer le plus court chemin du même graphe :



Le plus court chemin obtenu est celui-ci:

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

durée d'exécution: 1049936 nanosecondes

avec p représentant les parents

Nous avons ensuite calculé les plus courts chemins d'un noeud donné a un autre et avons obtenu ceci :

Chemin de A à B : [A, B]

Chemin de A à C : [A, B, E, D, C]
Chemin de A à D : [A, B, E, D]
Chemin de A à E : [A, B, E]

Question 17:

En analysant les résultats, nous pouvons constater que les deux algorithmes nous donnent la même réponse. Cependant, en ajoutant quelques lignes permettant de mesurer le temps d'exécution de chacun des algorithmes dans les main, on se rend compte que l'algorithme de Dijkstra met moins de temps à s'exécuter que l'algorithme de Bellman Ford

Question 18:

L'algorithme le plus efficace dépend du graphe pour lequel on veut l'utiliser :
Si le graphe a des coûts d'arc négatifs, alors Bellman Ford est indispensable car Dijkstra ne peut pas gérer ces cas. Dans les autres cas, Dijkstra devrait être le plus efficace car il est le plus rapide. Cependant, Si nous le faisons à la main, Bellman Ford est plus rapide que Dijkstra car il demande moins de calculs que ce dernier

Question 19:

Nous avons donc créé un constructeur prenant en paramètre un nom de fichier qui comprend le descriptif d'un graphe, ce qui nous a permis de faire une boucle qui lit les graphes numérotés de 11 à 15 50 fois chacun, calculant à chaque fois le temps d'exécution et nous avons pris la moyenne à la fin, ce qui nous a permis de nous rendre compte que dijkstra est plus rapide que bellman ford