

Binôme:

Husser Yanis

Olsozwiak-Delandre Mathéo

SAE 2.02 - Exploration algorithmique d'un problème

Section 2 : Représentation d'un graphe

Dans cette partie, nous avons mis en place une structure simple pour représenter des graphes orientés avec des coûts associés à chaque arc. Les noeuds sont représentés simplement par des chaînes de caractères, ce qui permet de les manipuler facilement sans créer de classe spécifique pour eux.

Pour modéliser les arcs entre les noeuds, nous avons défini une classe Arc, qui contient le nom du noeud de destination et le coût du trajet. Chaque noeud peut avoir plusieurs arcs sortants, donc nous avons aussi créé une classe Arcs qui gère une liste d'objets Arc. Cela permet de regrouper les arcs liés à un même noeud.

L'ensemble du graphe est manipulé à travers une interface Graphe, qui impose deux méthodes essentielles : `listeNoeuds()` pour obtenir tous les noeuds du graphe, et `suivants(String n)` pour obtenir les arcs sortants d'un noeud donné. L'implémentation de cette interface est assurée par la classe `GrapheListe`, qui utilise deux listes parallèles : une pour stocker les noms des noeuds et une autre pour les objets Arcs correspondants. Quand un arc est ajouté, si le noeud de départ ou d'arrivée n'existe pas encore, il est automatiquement ajouté à la structure.

Une méthode `toString()` a également été mise en place pour pouvoir afficher facilement le contenu du graphe, avec un format lisible indiquant pour chaque noeud les destinations possibles et les coûts associés.

Pour s'assurer du bon fonctionnement de notre structure, nous avons mis en place plusieurs tests unitaires. Par exemple, nous avons vérifié que les noeuds sont bien ajoutés même s'ils ne sont pas présents au moment où on ajoute un arc. On a aussi testé la méthode `getIndice()` qui permet de retrouver la position d'un noeud dans la liste. On a ensuite vérifié que `listeNoeuds()` retourne bien tous les noeuds ajoutés, et que la méthode `suivants()` renvoie les bons arcs pour un noeud donné. Enfin, un test a été écrit pour s'assurer que la méthode `toString()` retourne une représentation correcte et complète du graphe construit.

Section 3 : Calcul du plus court chemin par point fixe

L'objectif de cette section était de résoudre le problème du plus court chemin dans un graphe en utilisant l'algorithme du point fixe.

Nous avons d'abord rédigé le pseudo-code de l'algorithme. Celui-ci commence par initialiser la distance de tous les noeuds à +infini, sauf celle du noeud de départ qui est mise à 0. Ensuite, tant qu'il y a des modifications indiquant ainsi une amélioration des plus courts chemins, l'algorithme parcourt tous les noeuds et leurs arcs sortants pour mettre à jour les distances et les parents.

Algorithme utilisé:

fonction pointFixe(Graphe g, Noeud depart):

debut

 liste <- g.listeNoeuds()

 taille <- liste.size()

 pour i de 0 a taille - 1 faire

 X <- liste[i]

 L(X) <- +infini

 fpour

 L(depart) <- 0

 changement <- vrai

 tant que changement = vrai faire

 changement <- faux

 pour i de 0 a taille - 1 faire

 X <- liste[i]

 arcs <- g.suivants(X)

 pour j de 0 a arcs.size() - 1 faire

 arc <- arcs[j]

 Y <- arc.getDestination()

 cout <- arc.getCout()

 si $L(X) + \text{cout} < L(Y)$ alors

$L(Y) <- L(X) + \text{cout}$

 parent(Y) <- X

 changement <- vrai

 fsi

 fpour

 fpour

 ftant

fin

Nous avons ensuite implémenté cet algorithme en Java dans la classe BellmanFord, via la méthode `resoudre(Graphe g, String depart)`. Cette méthode prend en entrée un graphe et le nom d'un nœud de départ, puis retourne un objet Valeurs. Cet objet contient, pour chaque nœud, la distance minimale depuis le départ ainsi que le nœud parent qui précède ce nœud sur le chemin optimal.

Pour stocker ces informations, nous avons créé une classe Valeurs qui associe à chaque nœud sa distance depuis le point de départ ainsi que son parent sur le chemin. Cette classe nous a aussi permis de manipuler les données comme les parents ou les longueurs de chemins associées à chaque noeud.

Une fois l'implémentation terminée, nous avons testé l'algorithme sur un exemple de graphe pour vérifier que les distances et les chemins calculés correspondaient bien aux attentes. Nous avons également rédigé un test unitaire pour automatiser cette vérification. Enfin, nous avons écrit la méthode `calculerChemin` qui, à partir d'un noeud donné en paramètre, renvoie la liste des noeuds à parcourir pour atteindre ce nœud par le chemin le plus optimal.