Rapport de la SAE 2.02

Par HAUTION Ilan et LOUNICI Ilyès

I) Représentation d'un graphe

La première partie de la SAE était centrée sur la création de classe afin de représenter un graphe. Nous avons créé des classes Arc, Arcs, GrapheListe, Main dans le répertoire src mais aussi un interface Graphe. Dans le répertoire test, nous avons créé TestGraphe permettant de tester la construction du graphe et les méthodes de la classe GrapheListe.
La classe Arc représente un arc entre deux noeuds mais aussi le coût associé
La classe Arcs gère une liste d'arcs partant d'un noeud défini
La classe GrapheListe implémente l'interface Graphe pour représenter un graphe avec des listes.
L'interface Graphe permet de definir les méthodes essentielles pour manipuler un graphe.

II) Calcul du plus court chemin par point fixe

Dans la seconde partie de cette SAE, nous avons commencé par écrire l'algorithme de la fonction pointFixe que voici (question 8) :

```
Fonction pointFixe(Graphe g, Noeud depart)
   v -> new Main.Valeur()
   Pour chaque n dans g.listeNoeuds() faire
       v.setValeur(n, +infini)
       v.setParent(n, null)
   FinPour

   v.setValeur(depart, 0)

   // Boucle principale
   valeurModifiee -> true
   Tant que valeurModifiee est vrai faire
       valeurModifiee -> false
       Pour chaque n dans g.listeNoeuds()
           Pour chaque a dans g.suivants(n)
               voisin -> a.getDest()
               nouvelleValeur -> v.getValeur(n) + a.getDist()
               Si nouvelleValeur < v.getValeur(voisin) Alors
                   v.setValeur(voisin, nouvelleValeur)
                   v.setParent(voisin, n)
                   valeurModifiee -> true
               FinSi
           FinPour
       FinPour
   FinTantQue
```

```
   Retourner v
FinFonction

lexique :
   Grpahe g : le graphique de base
   Main.Valeur v : le tableau de valeurs
   Noeud depart, n, voisin : la classe noeud n'existant pas ceci
est l'equivalent d'un String
   Arc a : les arcs suivant le noeud que l'on etudie
   Entier nouvelleValeur : variable temporaire pour modifier la
valeur si elle est plus courte
   Booléen valeurModifiee : cette variable permet de savoir si une
valeur a été modifiée et donc si le point fixe est atteint ou pas
```

Ensuite, nous avons créé la classe BellmanFord qui implémente l'algorithme du point fixe (question 9). et une classe Main qui applique l'algorithme du point fixe sur un graphe fourni pour construire le chemin le plus court (question 10). Nous avons aussi créé une classe test nommé ValeurTest dans laquel on verifie que l'algorithme du point fixe est correct (question 11).
Enfin, nous avons modifié la classe fournit Valeur afin d'ajouter la méthode calculerChemin (question 12).

III) Calcul du meilleur chemin par Dijkstra

Pour la troisième partie de la SAE, nous avons commencé par recopier et transposer en java l'algorithme Dijkstra fournit ainsi que la méthode résoudre. Nous avons aussi créé une interface Algorithme qui permet de choisir l'algorithme à utiliser (question 13). Ensuite, nous avons créé des tests (à la manière de la classe BellmanFord) (question 14) et nous avons écrit un programme principal nommé MainDijkstra (question 15).

IV) Validation et expérimentation

(Question 16) Les résultats obtenus par les deux algorithmes sont similaires et aucune différence dans le chemin n'est notifiable. (Question 17 et 18) Néanmoins, on remarque que l'algorithme de Dijkstra est nettement plus rapide à donner un résultat que celui du point fixe. On parle d'une différence d'environ 200% plus rapide pour l'algorithme de Dijkstra. Cela s'explique sûrement par le fait que Dijkstra comporte une sorte "d'ordre de priorité" et qu'elle s'exécute plus rapidement et en consommant moins de mémoire (même lors des graphes denses).
Pour finir, nous avons mis au point un constructeur de graphe qui, grâce à un nom de fichier, permet de créer un graphe automatiquement.