IQ02E020 Exploration algorithmique d'un problème

SAE 2.02

Thème en 2023 : Graphes Orientés Valués

Contact : mikal.ziane@gmail.com (les adresses officielles sont en panne)

Attention : le non-respect des dates limites indiquées ci-dessous entrainera des points de pénalités sur la note.

Introduction

Dans le cadre de la SAE 2.02 vous devez réaliser en Java une application permettant de manipuler des graphes comme indiqué dans une série de tests unitaires fournis en annexe. Une application visée est la détermination du plus court chemin (en minutes) pour aller d'une station du métro parisien à une autre.

Vous vous répartirez en équipes de 4 étudiants (plus ou moins 1 si nécessaire) du même groupe de TP (201 par exemple) de façon à ce qu'il y ait 3 équipes par groupe : par exemple 4+4+3 = 11 ou 5+4+4 = 13.

Un projet privé github devra être créé pour l'équipe et votre chargé de TP devra être invité comme collaborateur. Un README.txt devra être défini avec la liste des membres de l'équipe et une synthèse de ce qui a été fait. Il n'y aura **pas** de rapport par ailleurs. Le chargé de TP devra avoir accès à tout le projet et notamment aux *commits*. Le Readme indiquant les membres de l'équipe et l'invitation du chargé de TD doivent être faites pour lundi 27 mars matin 8h.

La note de ce travail ne sera pas individualisée, si l'équipe fonctionne sans problème majeur, mais lors des DST un exercice porteront sur ce travail et la note (individuelle) aura le même coefficient.

Ce travail est composé d'une série d'exercices qui portent sur la notion de sous-typage et de polymorphisme qui sont vus dans l'enseignement Développement à Objets. Tout le développement sera fait en Java.

Une recette/soutenance aura lieu lors de la dernière séance de TP : le projet devra donc être rendu sur github jusqu'au **lundi 22 mai matin 8h**. Il n'y aura aucun report et la partie I devra être terminée pour le lundi 17 avril matin 8h.

Partie I Représentation des graphes orientés valués

La notion de sous-typage permet de masquer les différentes façons d'implémenter un graphe (matrice d'adjacence, liste d'adjacence etc.) tout en permettant de manipuler des graphes malgré tout.

Les différentes façons d'implémenter un graphe seront simplement codées chacune dans une classe dédiée qui implémenteront une interface **`IGraphe`** qui représente les graphes modifiables et qui implémenteront donc aussi sa super interface **`IGrapheConst`** qui représente les graphes non modifiables qui seront utilisés par Dijkstra. Les nœuds seront identifiés par une chaine de caractères. On suppose)ra qu'un graphe ne contient pas de valuation négative. Toutes les classes mentionnées ci-dessous doivent vérifier les tests unitaires qui seront donnés par ailleurs. Pour faciliter l'écriture de ces tests les classes de graphes ci-dessous auront, en plus du constructeur vide qui définit un graphe vide, un constructeur avec une chaine de la forme suivante "A-B(5), A-C(10), B-C(3), C-D(8), E:"qui décrit un graphe avec les sommets A,B,C,D,E et les arcs A vers B de valuation 5, A vers C de valuation

10 etc. Notez que E n'a pas d'arcs sortants. Bien entendu les noms des nœuds pourront faire plus d'un caractère.
Certaines des classes ci-dessous utilisent une classe `Arc` qui contient 3 données : un nœud source de type String, un nœud destination de type String et une valuation entière non négative.

Liste d'arcs

Écrivez une classe **`GrapheLArcs`** qui représente un graphe à l'aide d'une liste d'arcs. Les nœuds N qui n'ont aucun arc entrant ni sortant seront représentés par un arc factice de N vers un nœud dont le nom est la chaine vide "" et dont la valuation est 0.

```
private List<Arc> arcs;
```

Matrice d'adjacence

Écrivez une classe **`GrapheMAdj`** qui représente un graphe à l'aide d'une matrice d'adjacence.

```
private int[][] matrice;
private Map<String, Integer> indices;
```

Listes d'adjacence

Écrivez une classe **`GrapheLAdj`** qui représente un graphe par des listes d'adjacence :

```
private Map<String, List<Arc>> ladj;
```

Table de Hachage

Écrivez une classe **`GrapheHHAdj`** qui représente un graphe un peu comme le fait `GrapheLAdj` mais avec des tables de hachage imbriquées au lieu de listes d'adjacence.

```
private Map<String, Map<String, Integer>> hhadj;
```

Partie II Algorithme du plus court chemin de Dijkstra

Implémentez l'algorithme du plus court chemin de Dijkstra sur l'interface `IGrapheConst` **sans utiliser explicitement** les classes de graphes.
Des tests unitaires seront fournis ultérieurement et devront être exécutés avec succès pour toutes les classes de graphe mentionnées plus haut.

Partie III Étude comparative des différentes représentations

Le but de cette partie est de comparer l'efficacité en temps et en espace pour les classes de graphes implémentées lors de la première partie. Chaque représentation sera évaluée pour un ou plusieurs profils d'utilisation c'est-à-dire un certain nombre d'appels avec des graphes d'une certaine taille pour certaines méthodes de l'interface IGraphe.
En particulier il s'agira de

- mesurer le temps d'exécution de l'algorithme de Dijkstra sur de grands graphes,
- mesurer le temps nécessaire pour ajouter N nœuds au graphe.