Programmation Orientée Objet

TP collections – Les pangolins à la conquête de l'espace

Ensimag 2ème

1 Vers l'infini et au-delà

N'avons-nous pas tous été émus lorsque l'Humanité envoya dans l'espace en 2021 l'homme le plus riche de la planète ? N'avons-nous pas tous été déçus de le voir revenir sans encombre, dix minutes plus tard ? Rien de tout cela n'aurait été possible sans la contribution des pangolins cartographes, envoyés dans l'espace pour paver le chemin de ce bon Jeff. Ce TP retrace une partie de cette épopée des temps modernes, en vous mettant dans la peau d'un ingénieur responsable du développement d'un programme de cartographie spatiale (on n'allait quand même pas utiliser Google Maps pour diriger le PDG d'Amazon !).

Bon, en vrai, on va écrire du code Java pour représenter des *graphes orientés étiquetés et extensibles*. Et on n'utilisera pas de pangolin non plus[1]. Mais restez quand même, et si ça se trouve vous apprendrez deux-trois trucs sur les collections !

Pour rappel, un graphe orienté est une structure mathématique constituée d'un ensemble $\mathcal{N}$ de sommets (ou nœuds), accompagné d'un ensemble de couples de $\mathcal{N} \times \mathcal{N}$: les arcs. Nous supposons que les sommets sont étiquetés par des chaînes de caractères, c'est-à-dire qu'à tout sommet est associé une chaîne de caractères (`String`) unique. De plus, on suppose qu'il ne peut pas y avoir plusieurs sommets de même étiquette.

Par « extensible », on entend des graphes auxquels on peut ajouter dynamiquement des sommets et des arcs. Pour les arcs, à vous de définir si vous autorisez ou non d'avoir plusieurs arcs entre deux sommets (multigraphe ou non).

On s'intéresse ici à :

— la structure d'un graphe ;
— savoir l'afficher ;
— déterminer si un chemin existe entre deux sommets.

On introduit donc l'`interface Graphe` suivante :

```
1  public interface Graphe {
2
3      /**
4       * Ajoute un nouveau sommet d'étiquette label.
5       * Si un sommet de même étiquette existe déjà, cette méthode ne fait
            rien.
6       */
7      public void ajouteSommet(String label);
8
9      /**
10      * Ajoute un nouvel arc entre deux sommets.
11      * Si les sommets n'existent pas encore, ils sont ajoutés au graphe.
12      * Si un arc existe déjà entre ces deux sommets, on ne fait rien
13      * (sauf en cas de multi-graphe).
14      */
```

1. On a effectivement vite conclu que ça ne servait à rien, parce que chacun sait que dans l'espace, personne ne vous entend crier().

```java
    public void ajouteArc(String labelDepart, String labelArrivee);


    /**
     * Retourne une chaîne contenant la liste des sommets, puis celle
        des arcs.
     */
    @Override
    public String toString();

    /**
     * Cherche l'existence d'un chemin entre deux sommets du graphe.
     * @return true si un chemin existe
     */
    public boolean existeChemin(String labelDepart, String labelArrivee)
        ;
}
```

Une classe `TestGraphe.java` est fournie pour construire et tester le (multi-)graphe de la figure 1 à partir de la spécification ci-dessus.

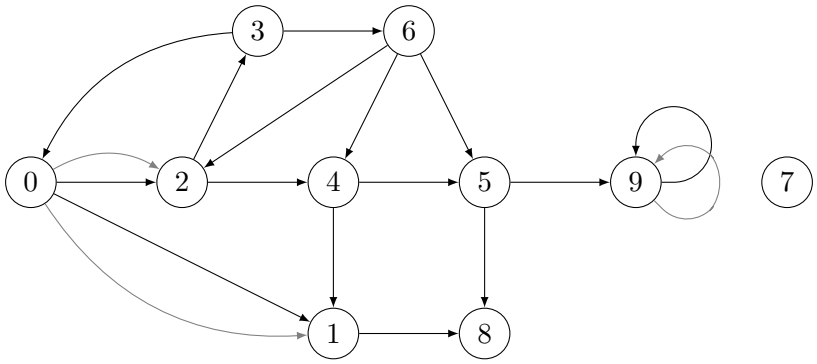

FIGURE 1 – Exemple de graphe orienté construit dans `TestGraphe.java`. En gris les arêtes ajoutées si l'on considère un modèle de multigraphe.

2 Codage par successeurs

Il existe différentes façons de représenter des graphes en mémoire. Les graphes extensibles reposent toujours sur des codages par successeurs, et non des tables d'adjacences statiques. Ceci signifie que le codage du graphe en mémoire va ressembler un peu au graphe lui-même : un sommet donné contient des liens (références) vers chacun de ses sommets successeurs. Un tel codage est avantageux pour implémenter des algorithmes de parcours vers l'avant (recherche de chemin, par exemple).

Cette représentation repose ici sur deux les deux classes suivantes :

— un `Sommet` est défini par son étiquette et ses successeurs ;

— un `GrapheSuccesseurs` contient simplement l'ensemble des sommets du graphe. Cette classe doit réaliser l'interface `Graphe` ci-dessus, et donc définir toutes ses méthodes.

Question 1 Définir les attributs des classes et écrire les méthodes `ajouteSommet` et `ajouteArc`. Bien entendu, vous devez ici utiliser les collections Java adéquates (choix à justifier !).

Question 2 Redéfinir la méthode `toString()` de la classe `Graphe` pour afficher l'ensemble des sommets du graphe, puis l'ensemble des arcs.

Question 3 Redéfinir la méthode `existeChemin` qui cherche s'il existe un chemin entre deux sommets.

Cette recherche nécessite de résoudre le problème de *marquage* des sommets, pour éviter de tourner en rond en cas de cycle... Pour savoir si un sommet a déjà été traité, plusieurs approches sont possibles :

— garder une information au niveau de chaque sommet (qui doit être initialisée puis mise à jour)

— maintenir une collection contenant les sommets déjà parcourus.

A vous de tester une ou ces deux méthodes ; laquelle vous paraît la plus intéressante ?

3 Autre représentation

Une autre représentation possible est de décrire entièrement la structure au niveau du graphe lui-même. Celui-ci contient alors une association (clé, valeur) entre chaque sommet (la clé) et l'ensemble de ses successeurs (la valeur).

Question 4 Implanter cette deuxième représentation dans une nouvelle classe réalisant elle aussi l'interface `Graphe`. Ainsi, vous pourrez la tester avec le même programme de test que précédemment.

Quels sont les avantages/inconvénients par rapport à la première solution ?