TP 1 - Représentation de graphes
Matrice d'adjacence et Listes d'adjacence

Gilles Simonin

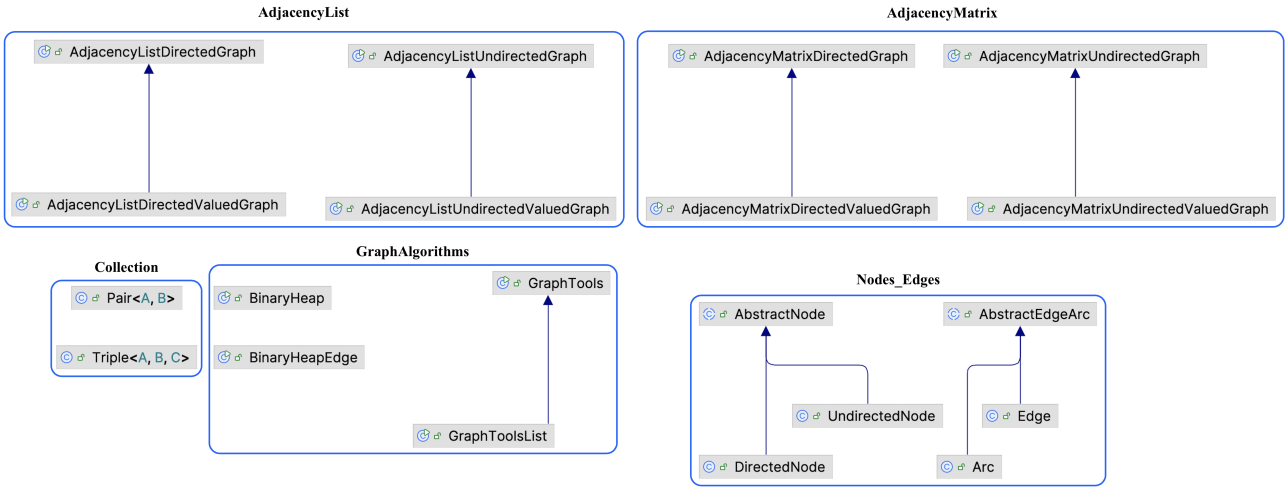
Résumé

Votre travail lors de ce TP est de jouer avec la représentation d'un graphe orienté (ou non) et pondéré (ou non) par sa matrice d'adjacence, puis par une liste d'adjacence. Lors de touts les TPs basés sur le framework donné, nous considérerons seulement des graphes simples.

1 Importation de projet

Question 1 *Importez dans votre workspace sous Eclipse le projet se trouvant sur github en ajoutant l'option copier dans le workspace. Ce framework composées de classes java représente l'espace dans lequel vous avez différentes méthodes à compléter au fil du TP.*

Question 2 *Afin de voir si tout marche bien, commencez par ouvrir dans le package AdjacencyMatrix la classe "AdjacencyMatrixUndirectedGraph". Cliquez sur le menu Run en haut d'eclipse puis choisissez Run. Sinon vous pouvez lancer les tests en cliquant sur le bouton rond vert ayant un triangle blanc (play). Vous devez voir dans la console en base de votre eclipse les tests pré-codés pour votre TP. Pour le moment il y a juste des graphes affichés sous forme de matrice.*
Le diagramme UML de la figure 1 représente le framework qui vous est proposé pour ces TPs.



2 Représentation d'un graphe par une matrice d'adjacence

Question 3 *Dans le package AdjacencyMatrix, nous vous proposons un implémentation quasi-complète de la structure de graphe en utilisant la représentation sous forme de matrice d'adjacence : Dans une premier temps, observez en haut de la classe "AdjacencyMatrixUndirectedGraph" les lignes suivantes :*

- *private int nbNodes ; // nombre de sommets*
- *private int nbEdges ; // nombre d'arêtes*

— *private int[][][] matrix ; // matrice d'adjacence*

Ces trois lignes définissent les variables d'instance de la classe. C'est à dire que vous pouvez les utiliser dans toutes les méthodes de votre programme. Et principalement la variable matrix qui représente votre graphe sous forme de matrice, où $matrix[i][j]$ désigne la valeur dans la case de la ligne i et de la colonne j. Pour $nbNodes$ sommets dans votre graphe (représenté par la variable $nbNodes$), les lignes de la matrice vont de 0 à $(nbNodes - 1)$ en informatique. D'ailleurs il est souvent préférable de numéroter les sommets d'un graphe à partir de 0 pour se simplifier le codage et les affichages.

Pour représenter la matrice d'adjacence, les modifications du graphe et accès aux informations passeront par cette matrice. Une fois cette compréhension faite, il vous est demandé de compléter des ***méthodes*** *de bases pour les graphes non orientés :*
— `isEdge(int x, int y)`,
— `removeEdge(int x, int y)`,
— `addEdge(int x, int y)`.
Puis les méthodes de bases pour les graphes orientés :
— `isArc(int from, int to)`,
— `removeArc(int from, int to)`,
— `addArc(int from, int to)`.

Enfin vous testerez vos méthodes en complétant la méthode principale `main`.

Question 4 *Vous continuerez en codant la méthode d'inversion de matrice "computeInverse()". Cette méthode retourne une nouvelle matrice déjà défini dans le code de la méthode en se basant sur les données déjà placées dans votre variable de classe : matrix.*
Votre travail consiste à inverser les éléments de manière symétrique. Les valeurs sur les cases [i][j] et [j][i] sont échangées.

3 Implémentation sous forme de liste d'adjacence

Nous proposons ici de mettre en oeuvre une implémentation de la structure de graphe en utilisant la représentation sous forme de liste d'adjacence. Dans une premier temps, observez en haut de la classe `AdjacencyListUndirectedGraph` les lignes suivantes :
— protected $List<UndirectedNode>$ *nodes* ; // liste des sommets du graphe
— protected $List<Edge>$ *edges* ; // liste des arêtes du graphe
— private int nbNodes ; // nombre de sommets
— private int nbEdges ; // nombre d'arêtes

Dans cette implémentation sous forme de liste d'adjacence, tout graphe $G = (V, E)$ est représenté par sa liste de sommets V et sa liste d'arêtes E. **Chaque sommet $v \in V$ aura sa liste d'arêtes incidentes $\delta(v)$, liste permettant d'avoir accès aux sommets voisins.**
Afin de mieux gérer les informations sur les sommets et les arêtes/arcs, des objets "Node" et "Edge" sont créés (avec des versions orientées et non orientées) dans le package *Nodes_Edges*.

3.1 Edges/Arcs

Prenons la classe `Edge` (équivalent pour `Arc`) qui hérite de la classe abstraite `AbstractEdgeArc`. Une arête (ou arc) est définie par ses sommets extrémités, et par un poids si besoin (par défaut égal à 0 si non pondéré). La classe `Edge` possède une méthode "equals" qui permet de comparer deux arêtes ayant les mêmes extrémités (quel que soit leur ordre).
Voici les méthodes utiles dans `Edge` (et `AbstractEdgeArc` par héritage) :
— `getFirstNode()` ; // Retourne le premier sommet
— `getSecondNode()` ; // Retourne le second sommet
— `getWeight()` ; // Retourne le poids de l'arête

3.2 Nodes

Les sommets (Nodes), sont définis selon le type de représentation du graphe que l'on a. Pour les graphes non orientés sous forme de liste d'adjacence, les sommets sont représentés par la classe `UndirectedNode`. Pour les graphes orientés, il faut utiliser la classe `DirectedNode`. Ces deux classes héritent de la classe abstraite `AbstractNode` où un label (de type *int*) indique le numéro du sommet.
Dans la classe `UndirectedNode`, tout sommet v possède une liste d'arêtes incidentes $\delta(v)$, permettant ainsi d'avoir accès au voisinage du sommet.
Analysez le code et en particulier la ligne suivante :

- private $List < Edge >$ *incidentEdges* ; // Liste d'arêtes dont une extrémité est le sommet "this".
- *public void addEdge*(*Edge* $e1$) ; // Méthode permettant d'ajouter une arête e_1 à une liste $\delta(v)$ si v est une extrémité du sommet. L'arête est toujours ajoutée de telle manière que v est le premier sommet de l'arête.

Ainsi, pour chaque arête dans $\delta(v)$, le premier sommet (méthode `getFirstNode`() de *Edge*) sera toujours le sommet v. Et pour récupérer les voisins de v, il faudra parcourir $\delta(v)$ et pour chaque arête appeler la méthode `getSecondNode`().

3.3 Code à compléter

Question 5 *Complétez la classe* `AdjacencyListUndirectedGraph` *qui représente un graphe non orienté par sa liste de voisins. Il vous faudra compléter les méthodes suivantes :*

- `isEdge(UndirectedNode x, UndirectedNode y)`,
- `removeEdge(UndirectedNode x, UndirectedNode y)`
- `addEdge(UndirectedNode x, UndirectedNode y)`
- `toAdjacencyMatrix`()

Enfin vous testerez ces méthodes en complétant la méthode principale `main`().

Question 6 *De la même manière, complétez la classe* `AdjacencyListDirectedGraph` *qui permet de représenter un graphe orienté par ses listes de successeurs (List < Arc > arcSucc) et prédécesseurs (List<Arc> arcPred) via la classe* `DirectedNode`.

Complétez les méthodes équivalentes à celle de la question 5, ainsi que la méthode `computeInverse`().

4 Vers d'autres représentations

Cette partie est à faire chez soi, nous proposons ici de mettre en oeuvre différentes implémentations de la structure de graphe par rapport à la gestion des poids d'arêtes/arcs dans les graphes. Pour les matrices et les listes d'adjacence, complétez les sous-classes liées contenants le mot "Valued".