Complément au TP 2 — Précisions sur la manipulation des graphes

Ce court document donne quelques précisions sur les exercices de la feuille de TP 2, afin de lever toute ambiguïté sur le travail qu'il faut avoir fait pour pouvoir enchaîner facillement sur les exercices qui viendront.

Précisions sur l'exercice 1

La première fonction, qui lit la matrice d'adjacence du graphe doit avoir, peu ou prou, le profil suivant (qui est légèrement modifié par rapport à ce qui est écrit dans l'énoncé initial, ici, on suppose que la taille de la matrice a été déterminée avant l'appel à cette fonction) :

- **public static int[][]** *getMatrix (***int** *n)*

La seconde fonction, quant à elle, a un argument qui est un entier entre 1 et n (un numéro de sommet), et elle utilise la matrice d'adjacence lue avec l'autre fonction, pour calculer la liste d'adjacence du sommet fourni (cela est fait en parcourant la bonne ligne de la matrice à la recherche des coefficients 1) ; elle peut avoir le profil suivant (mais il y a d'autres possibilités) :

- **public static int[]** *computeAdjacentyList (***int** *x,* **int[][]** *matrix)*

Enfin, le programme principal utilise ces deux fonctions, d'abord pour lire la matrice d'adjacence d'un graphe, puis, répétitivement, lire un entier (représentant un numéro de sommet) et afficher la liste d'adjacence de ce sommet. Noter que l'on ne demande pas de calculer une structure contenant toutes les listes d'adjacence, mais seulement calculer, à la volée, une liste d'adjacence et de l'afficher. Le stockage et la gestion des listes d'adjacence sera le thème des exercides qui suivent dans cette feuille 2.

Il est indiqué dans l'énoncé que la numérotation des sommets est de 1 à n (qui est l'ordre du graphe), il n'est pas totalement incongru (même si ce n'est pas la meilleure méthode) que les tableaux utilisés (lorsqu'ils sont indexés par les numéros des sommets) soient indexés de 0 à n (donc de taille $n+1$), la case d'indice 0 étant inutilisée. Dans ce cas, on prendra bien soin de vérifier que lorsqu'une fonction est censée recevoir un numéro de sommet, ce numéro soit bien entre 1 et n, de manière à éviter l'erreur consistant à donner un numéro de sommet qui est 0.

Précisions sur les exercices 2 et 3

Ces deux exercices peuvent être faits en les mélangeant, pas obligatoirement dans l'ordre 2 puis 3. Ils concernent tous les deux essentiellement la représentation et la manipulation des graphes par leurs listes d'adjacence.

Dans la classe *GraphSimple*, on doit trouver au moins un constructeur, quelques attributs privés, et quelques méthodes assez élémentaires ; dans ce qui suit, les noms des entités publiques sont imposés, ceux des entités privées sont laissés à votre libre choix :

- il faudra peut-être un attribut permettant de représenter une matrice d'adjacence, donc de type **int [][]** ; ce n'est pas une obligation, c'est juste parfois une facilité (confer la fin de cette description) ; il faudra prévoir que cette matrice ne soit pas calculée pour un graphe donné (si un graphe a quelques milliers de sommets et quelques dizaines de milliers d'arêtes, on peut envisager de le représenter par ses listes d'adjacence, en revanche on sera très circonspect à l'idée de représenter sa matrice d'adjacence, qui sera remplie essentiellement de 0) ;
- un attribut qui permet de représenter le tableau des listes d'adjacence, qui sera également de type **int [][]**, j'espère que vous avez compris, dans les exercices des feuilles précédentes comment on manipulait ce genre de structure ; pour qu'un graphe soit bien défini, il faudra que cet attribut ait été initialisé (alors que, comme on l'a dit précédemment, ce ne sera pas le cas de la matrice d'adjacence) ;
- **public** *GraphSimple*(**int** n), est le constructeur obligatoire, il prépare toute la structure

de l'objet graphe (prépare les attributs, en particulier), et fournit un graphe nul d'ordre n (c'est-à-dire un graphe ayant n sommets et zéro arête) ;
- **public int** *order*() fournit l'ordre de ce graphe (i.e. le nombre de ses sommets) ;
- **public boolean** *isVertex*(**int** x) indique si x est le numéro d'un sommet de ce graphe (i.e. est un entier compris entre 1 et *order*() ;
- **public boolean** *isEdge*(**int** x, **int** y) indique si (x, y) est une arête de ce graphe ; bien entendu, x et y doivent être des sommets de ce graphe ;
- **public int** *degree*(**int x**) donne le degré du sommet x de ce graphe ;
- **public void** *setAdjacencyList* (**int** x, **int[] neighborhood**) définit la liste d'adjacence du sommet x avec le contenu de du tableau *neighborhood* (qui contient donc certains entiers entre 1 et n) ; c'est avec cette méthode principalement que l'on peut enrichir le graphe avec de nouvelles arêtes (en particulier, lorsqu'on le construit initialement à partir de données lues sur un flot d'entrée) ; il est à noter que cette méthode doit recopier le tableau *neighborhood* afin d'éviter de vicieux problèmes de partage de structure) ;
- **public int[]** *getAdjacencyList* (**int** x) fournit la liste d'adjacence du sommet x ; contrairement à la précédente méthode, celle-ci ne fait pas de recopie de la liste d'adjacence, il est donc conseiller de ne pas toucher au contenu du tableau renvoyé par la fonction et de ne s'en servir que pour itérer dans les voisins d'un sommet donné ;

Pour faciliter l'itération dans l'ensemble des sommets, on aura le choix entre faire une boucle de 1 à n (ce qui peut nécessiter des ajustements au niveau des indices de certains tableaux) ou bien de disposer d'un tableau contenant la liste des sommets (donc la liste des entiers de 1 à n rangée dans un tableau) dans lequel on peut itérer en utilisant la forme élaborée des boucles **for** ; dans ce dernier cas, il faudra prévoir
- un attribut représentant l'ensemble des sommets (un tableau des entiers de 1 à n) ;
- **public int[]** *vertexSet*() qui donne l'ensemble des sommets du graphe sous la forme d'un tableau d'entiers ; c'est la méthode permettant de récupérer le tableau contenant la liste des sommets afin de pouvoir itérer dans cette liste.

La manipulation de la matrice d'adjacence doit être menée avec précautions. Si le graphe est défini initialement par sa matrice d'adjacence, alors, lors de cette définition, on en profite pour calculer immédiatement les listes d'adjacence de ses sommets. En revanche, si le graphe est défini par les listes d'adjacence de ses sommets, on ne calcule pas la matrice d'adjacence du graphe et cette matrice ne sera calculée que lorsque l'on sollicitera l'obtention de cette matrice. Ceci explique le pourquoi de certaines particularités des entités suivantes :
- **public void** *setAdjacencyMatrix*(**int [][]** *matrix*) définit la matrice en paramètre comme la matrice d'adjacence du graphe ; l'argument *matrix* sera partagé, en ce sens qu'il n'y aura pas de recopie dans l'attribut représentant en interne la matrice d'adjacence de ce graphe ; si la représentation du graphe par listes d'adjacence n'a pas encore été définie, cette méthode est suivie automatiquement de la détermination des listes d'adjacence du graphe pour utilisations ultérieures ;
- **public int[][]** *getAdjacencyMatrix*() fournit la matrice d'adjacence de ce graphe ; il est à noter que cette méthode induit un partage de structure qui peut être dangereux ; si la matrice d'adjacence du graphe n'a pas été définie auparavant, cette méthode provoque le calcul de la matrice d'adjacence à partir de ses listes d'adjacence ;
- ce qui est dit sur l'existence ou pas de l'une ou l'autre des représentations implique qu'il faut gérer, de manière interne, des indicateurs permettant de savoir si la représentation par listes d'adjacence est définie et si la représentation par matrice d'adjacence est définie (deux booléens, donc) ;
- tout ceci implique également qu'en interne (à voir !), on aura deux fonctions permettant de calculer une représentation à partir de l'autre ; mais tout ceci a déjà été préparé dans les exercices précédents.