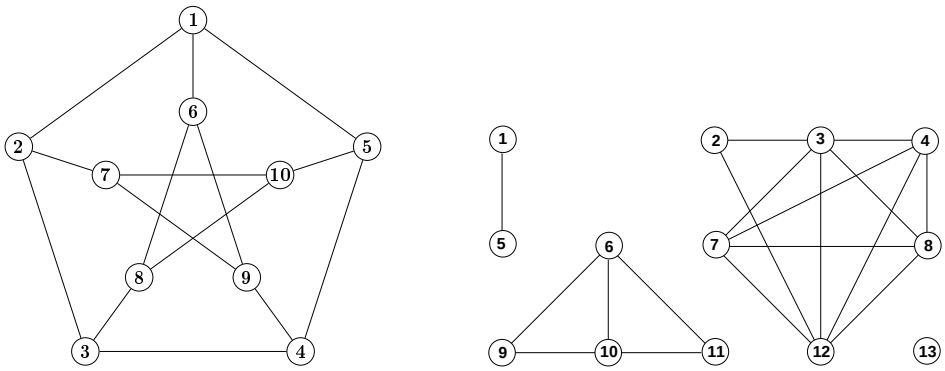
**Mathématiques pour l'informatique** **Licence 3 d'informatique**

Travaux pratiques n° 2 — Graphes : premiers pas

Sur l'ENT, à l'endroit habituel, vous trouverez, outre ce document (!), quelques exemples de graphes pouvant être utilisés lors des tests. Ces graphes sont fournis dans deux formats : matrice d'adjacence et suite de listes d'adjacence, avec parfois une petite spécification informelle.

Il y a au moins deux façons usuelles de coder un graphe : par listes d'adjacence ou par matrice d'adjacence. Dans tout ce qui suit, nous coderons les sommets des graphes par les **entiers à partir de 1** (et pas à partir de 0, bien que l'indexation naturelle des tableaux en Java soit faite à partir de 0). Nous allons ici, étudier un petit peu ces deux codages et mettre en œuvre des algorithmes de base qui nous permettront d'étudier quelques graphes plus tard. Les fonctions demandées dans les exercices qui suivent sont parfois très proches de fonctions déjà réalisées dans la feuille 1 (attention toutefois à la correspondance entre numéros de sommets et indices des tableaux utilisés). Sur la figure 1, nous donnons deux exemples de graphes pour illustrer le propos (à gauche, on voir le graphe de Petersen, et à droite un graphe sans nom).




**a.** Le graphe de Petersen **b.** Un graphe non connexe

**Figure 1.** Deux exemples de graphes non orientés

### *I. Codage par matrice d'adjacence*

Soit $G$ un graphe à $n$ sommets numérotés de 1 à $n$. Sa matrice d'adjacence est la matrice carrée de taille $N$ à valeurs booléennes (pour nous, les valeurs 0 et 1) donnée par $A = (a_{ij})$, $a_{ij}$ valant *true* ou *false*, et où $a_{ij}$ = true si et seulement si il existe une arête entre les sommets $i$ et $j$. Le type des matrices d'adjacence de nos graphes sera le type Java `int[][]` (où le nombre de lignes est égal au nombre de colonnes, c'est donc une matrice). Sur la figure 2, on voit le codage, par matrice d'adjacence, des deux graphes de la figure 1.

$$\begin{pmatrix} 0&1&0&0&1&1&0&0&0&0\\ 1&0&1&0&0&0&1&0&0&0\\ 0&1&0&1&0&0&0&1&0&0\\ 0&0&1&0&1&0&0&0&1&0\\ 1&0&0&1&0&0&0&0&0&1\\ 1&0&0&0&0&0&0&1&1&0\\ 0&1&0&0&0&0&0&0&1&1\\ 0&0&1&0&0&1&0&0&0&1\\ 0&0&0&1&0&1&1&0&0&0\\ 0&0&0&0&1&0&1&1&0&0 \end{pmatrix}$$

Le graphe **1.a** (de Petersen)

$$\begin{pmatrix} 0&0&0&0&1&0&0&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0&0&0&0&1&0\\ 0&1&0&1&0&0&1&1&0&0&0&1&0\\ 0&0&1&0&0&0&1&1&0&0&0&1&0\\ 1&0&0&0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&1&1&1&0&0\\ 0&0&1&1&0&0&0&1&0&0&0&1&0\\ 0&0&1&1&0&0&1&0&0&0&0&1&0\\ 0&0&0&0&0&1&0&0&0&1&0&0&0\\ 0&0&0&0&0&1&0&0&1&0&1&0&0\\ 0&0&0&0&0&1&0&0&0&1&0&0&0\\ 0&1&1&1&0&0&1&1&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&0&0&0&0 \end{pmatrix}$$

Le graphe **1.b**

**Figure 2.** Les matrices d'adjacence des deux graphes de la figure 1

**Exercice 1. Représentation par matrice d'adjacence**

Écrire une fonction qui lit la matrice d'adjacence d'un graphe sur le flot d'entrée (les données sont : l'ordre $n$ du graphe, puis les coefficients 0 ou 1 de la matrice d'adjacence). Puis écrire une seconde fonction qui a pour argument un entier $x$ compris entre 1 et $n$ représentant l'un des sommets du graphe, et qui retourne la liste d'adjacence du sommet $x$ en utilisant la matrice d'adjacence, précédemment lue.

Organiser ces deux fonctions en un programme qui commence par lire la description d'un graphe (donné par sa matrice d'adjacence), sur l'entrée standard ou bien dans un fichier dont le nom est donné en argument de la ligne de commande, puis qui lit répétitivement sur l'entrée standard un entier $x$ compris entre 1 et $n$ et affiche la liste d'adjacence du sommet $x$ (le programme s'arrête lorsque l'on fournit l'entier 0 au lieu de donner le numéro d'un sommet). Évidemment, ce programme doit utiliser les deux fonctions élaborées précédemment.

Remarque : le seul point d'attention des fonctions demandées est la numérotation des sommets du graphe : pour l'utilisateur ce sont des entiers de 1 à $n$, mais dans le programme, tout est stocké dans des tableaux indexés à partir de 0, il faut donc faire très attention au traitement des indices des tableaux.

Petite question subsidiaire : que l'on traite ainsi des graphes orientés ou des graphes non orientés, y a-t-il une différence dans les algorithmes utilisés, que ce soit pour la lecture ou le stockage de la matrice d'ajdacence, ou bien pour la détermination des listes d'adjacence ou leur affichage ?

Dans l'exercice précédent, si l'on part de l'un des deux graphes de la figure 1, les listes d'adjacence qui seront affichées doivent être conformes à ce que l'on voit sur la figure 3.

### *II. Codage par listes d'adjacence*

Soit $G$ un graphe (orienté ou non) sur $n$ sommets. La représentation par listes d'adjacence revient à coder la liste d'adjacence de chaque sommet comme on vient de le faire dans l'exercice 1 et à stocker les différentes listes d'adjacence dans un unique tableau qui permet de retrouver instantanément la liste d'adjacence du sommet $x$ (stockée à l'indice $x-1$ puisque les sommets sont les entiers de 1 à $n$ et que les tableaux sont indexés à partir de 0). Ce que l'on nomme ici « liste » d'adjacence est une suite de sommets, qui peut être réalisée par un tableau, par une liste, ou par toute autre structure adaptée (par exemple, un arbre binaire) ; pour ce qui nous intéresse ici nous coderons ces listes par des tableaux Java. Ainsi, le type qui représentera un graphe par listes d'adjacence sera là encore le type `int[][]` mais ici, contrairement à la représentation par matrice d'adjacence, les tableaux élémentaires stockés dans le tableau principal ne seront pas tous de même longueur.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 : | 2 | 5 | 6 | 6 : | 1 | 8 | 9 | | | | |
| 2 : | 1 | 3 | 7 | 7 : | 2 | 9 | 10 | | | | |
| 3 : | 2 | 4 | 8 | 8 : | 3 | 6 | 10 | | | | |
| 4 : | 3 | 5 | 9 | 9 : | 4 | 6 | 7 | | | | |
| 5 : | 1 | 4 | 10 | 10 : | 5 | 7 | 8 | | | | |

Le graphe **1.a** (de Petersen)

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 : | 5 | | | | | | | 8 : | 3 | 4 | 7 | 12 |
| 2 : | 3 | 12 | | | | | | 9 : | 6 | 10 | | |
| 3 : | 2 | 4 | 7 | 8 | 12 | | | 10 : | 6 | 9 | 11 | |
| 4 : | 3 | 7 | 8 | 12 | | | | 11 : | 6 | 10 | | |
| 5 : | 1 | | | | | | | 12 : | 2 | 3 | 4 | 7 | 8 |
| 6 : | 9 | 10 | 11 | | | | | 13 : | | | | |
| 7 : | 3 | 4 | 8 | 12 | | | | | | | | |

Le graphe **1.b**

**Figure 3.** La représentation par listes d'adjacence des deux graphes de la figure 1

Par exemple, si l'on prend le graphe de la figure 1.b, sa matrice d'adjacence est une matrice à 13 lignes et 13 colonnes, et sa représentation par listes d'adjacence sera un tableau de 13 cases, chaque cas contenant elle-même un tableau, tous ces tableaux n'étant généralement pas de même taille. Si l'on note $G$ le tableau représentant ce graphe par listes d'adjacence, on a alors $G[0] = [5]$ (c'est la liste d'adjacence du sommet 1), $G[1] = [3, 12]$ (c'est la liste d'adjacence du sommet 2), ..., $G[11] = [2, 3, 4, 7, 8]$ (c'est la liste d'adjacence du sommet 12), et pour finir $G[12] = []$ (c'est la liste d'adjacence du sommet 13). Nous allons maintenant créer un programme permettant de manipuler un graphe représenté sous forme de listes d'adjacence.

**Exercice 2. Une collection de fonctions d'entrées-sorties pour les graphes**

On définit ici une classe contenant une série de fonctions *static* (donc des méthodes de classe) permettant de faire des entrées-sorties relativement élémentaires sur des graphes que l'on veut représenter par listes d'adjacence. Cette classe doit s'appeler *GraphSimpleIO*.

**a.** Écrire une fonction nommée *Initialize* qui initialise un *Scanner* pour que l'on puisse lire le flot d'entrée ; ce *Scanner* sera stocké comme une variable statique dans la classe (un attribut de classe) de manière à ce que les autres fonctions de la classe puissent y accéder sans devoir en reconstruire un à chaque opération.

**b.** Écrire une fonction *getMatrix* qui a un argument entier $n$ représentant l'ordre d'un graphe, qui lit sur le flot d'entrée la matrice d'adjacence du graphe et renvoie cette matrice.

**c.** Écrire une fonction *getGraph* qui a un argument *graph* de type `int[][]`, a priori non initialisé, mais dont on connaît tout de même le nombre de cases dans la première dimension (en clair, cela signifie que l'on peut utiliser *graph.length*) et qui lit sur le flot d'entrée une suite de listes d'entiers, chaque liste représentant la liste d'adjacence de l'un des sommets du graphe. Pour être plus précis, si $n$ est la première dimension du tableau *graph*, alors ce que l'on va lire sur le flot d'entrée pour remplir le tableau de listes d'adjacence est une suite de listes d'entiers décrite de la manière suivante (l'exemple donné correspond au graphe de la figure 1.b) :

```
 1  5  0
 2  3 12  0
 3  2  4  7  8 12  0
 4  3  7  8 12  0
 5  1  0
 6  9 10 11  0
 7  3  4  8 12  0
 8  3  4  7 12  0
 9  6 10  0
10  6  9 11  0
11  6 10  0
12  2  3  4  7  8  0
13  0
```

Dans cette description on a une suite de $n$ lignes[1] (ici $n = 13$, c'est l'ordre du graphe), et chaque ligne commence par le numéro du sommet considéré, suivi de la suite des sommets adjacents à ce sommet[2], cette suite étant terminée par 0.

Pour implémenter la fonction précédente, étant donné que l'on ne connaît pas à l'avance la longueur de chacune des listes d'adjacence, il faut lire et stocker une liste d'adjacence dans un tableau suffisamment grand, puis quand on a terminé la lecture de la liste d'adjacence, allouer un tableau d'entiers de la bonne taille, y recopier ce qu'on a lu, et stocker le tableau ainsi obtenu au bon indice dans le paramètre *graph* de la fonction. Question : connaît-on une borne maximum à la longueur des listes d'adjacence des graphes manipulés ?

**d.** Écrire une fonction, nommée *printMatrix*, qui affiche sur le flot de sortie la matrice d'adjacence d'un graphe.

**e.** Écrire une fonction, nommée *printGraph*, qui a comme argument un graphe représenté sous forme de suite de listes d'adjacence, et qui affiche sur le flot de sortie la représentation du graphe sous forme de suite des listes d'adjacence des sommets du graphe.

**f.** Enfin, écrire une fonction, nommée *rawPrintGraph*, qui a comme argument un graphe représenté sous forme de suite de listes d'adjacence, et qui affiche sur le flot de sortie la représentation du graphe sous la forme décrite dans la question c (ce qui signifie que ce qui est affiché peut être lu par la fonction de la question c, directement, sans rien changer).

1. La présentation en lignes n'a aucune importance, ce qui compte c'est la succession des nombres entiers, la présentation en lignes est seulement destinée à permettre à quelqu'un de lire facilement cette description, c'est tout !
2. Par conséquent, on voit que l'on peut fournir ces lignes dans n'importe quel ordre, pas forcément par ordre croissant des sommets.

**g.** Par commodité, écrire une fonction nommée *getInt*, sans argument, qui lit un entier sur le flot d'entrée, on pourra alors l'utiliser lorsque, dans un programme, on aura besoin de lire un entier isolé (sans se préoccuper des instances de *Scanner* ou d'autres choses comme cela).

**h.** Lorsque tout ceci sera fait, ajoutez dans la classe *GraphSimpleIO* le constructeur défini par : `private GraphSimpleIO() {}`. Comprenez-vous à quoi cela sert ?

**Exercice 3. Réalisation d'un graphe simple**

Nous allons maintenant nous occuper de l'implémentation d'une structure de graphe simple, représenté par suite de listes d'adjacence. Nous allons donc écrire une classe, nommée *GraphSimple* (essayez de respecter l'orthographe pour tous les noms imposés dans l'énoncé) qui permettra de créer des objets de ce type désignant des graphes que l'on pourra alors manipuler. Un graphe sera donc représenté par une structure du type `int[][]` dont la première dimension sera l'ordre du graphe (le nombre de ses sommets).

**a.** Dans la classe *GraphSimple*, on trouvera donc au moins un attribut d'instance qui est le tableau d'entiers, à deux dimensions (tableau des listes d'adjacence) ; faire cela, et ajouter un constructeur dont l'unique argument sera un entier $n$ désignant l'ordre du graphe, qui initialise la première dimension du tableau des listes d'adjacence.

**b.** Ajouter à la classe une méthode *setAdjacencyList* qui a pour arguments un entier $x$ désignant un sommet du graphe (attention : les sommets sont numérotés à partir de 1) et un tableau d'entiers représentant la liste d'adjacence de ce sommet et qui affecte cette liste d'adjacence à la bonne position dans l'attribut qui est le tableau des listes d'adjacence. Une fois ceci fait, écrire également la méthode *getAdjacencyList* qui a pour unique argument un entier $x$ représentant un sommet du graphe et qui renvoie la liste d'adjacence de ce sommet, sous la forme d'un tableau d'entiers, évidemment.

**c.** Ensuite, ajouter les fonctions naturelles suivantes : la méthode *order* qui donne l'ordre du graphe, la méthode *degree* qui donne le degré d'un sommet désigné, le prédicat *isVertex* qui vérifie si un entier représente un sommet du graphe (s'il est supérieur ou égal à 1 et inférieur ou égal à l'ordre du graphe), et enfin, la méthode (pas efficace) indiquant s'il existe une arête entre les sommets $x$ et $y$.

**d.** Écrire ensuite une méthode *toMatrix* permettant de transformer un graphe représenté par un tableau de listes d'adjacence en une matrice d'adjacence pour ce même graphe ; ceci fait, écrire la fonction réciproque, *fromMatrix*, qui prend une matrice d'adjacence en argument et qui construit le graphe correspondant (dans la représentation par listes d'adjacence).

**e.** Ceci fait, écrire un petit programme permettant de tester cette classe, en utilisant les fonctions définies dans la classe *GraphSimpleIO*. Bien vérifier toutes les fonctions réalisées en apportant un soin tout particulier à vérifier la gestion des indices dans les tableaux utilisés (n'oublions pas que les sommets sont numérotés [pour l'utilisateur] de 1 à $n$ [ce qui permet d'utiliser 0 comme une valeur sentinelle] alors que les tableaux sont indexés à partir de 0). Il faut absolument que ces problèmes d'indexation soient réglés une fois pour toutes afin de ne pas avoir à s'en soucier lorsqu'il faudra implémenter des algorithmes sur les graphes (qui sont souvent un peu élaborés, voire complexes).