Institut Villebon - Georges Charpak
T.P. 1

Savoir faire :	• Connaitre les différents types de représentation d'un graphe.
	• Savoir implémenter une structure de graphe en Python.
	• Savoir effectuer un parcours de graphe.
	• Savoir implémenter l'algorithme de calcul des composantes connexes.

1 Condition de rendu du T.P.

Ce T.P. sera à rendre par mail à olivier.bouillot@villebon-charpak.fr, au plus tard le mardi 24 novembre à 23h59. Il sera à réaliser dans un premier temps seul, pendant la séance, puis à finir en binôme avec un membre de l'autre groupe.

Votre mail d'envoi aura pour sujet : [UE 5i4] Rendu de TP 1 + `prenom_1` + `prenom_2`, où `prenom_1` et `prenom_2` seront bien entendu remplacé par les prénoms des membres du binômes. Il contiendra une et une seule pièce jointe qui sera une archive nommée `nom_1_nom_2_-_TP_1.zip`, où `nom_1` et `nom_2` correspondent aux noms de familles des membres du groupe, `nom_1` étant avant `nom_2` dans le dictionnaire.

Veillez bien à ne pas modifier les noms donnés dans l'énoncé, car une partie de votre T.P. sera corrigé automatiquement : si vous ne respectez pas cette consigne, vous aurez alors 0, car aucun des tests que votre code subira ne passera...

Enfin, n'oubliez pas de respecter des règles d'hygiènes correctes pour votre code : cela sera hautement pris en compte dans la notation. Voir **??** et **??**.

2 Représentation d'un graphe

Un graphe orienté est un couple d'ensemble constitué de l'ensemble de ses sommets et de l'ensemble de ses arêtes.En machine, pour représenter un graphe, on pourrait alors simplement stocker ces deux ensembles. Cependant, cela entraîne un problème d'efficacité :

⇝ l'accès aux arêtes incidentes à un sommet demandera systématiquement une recherche ;

⇝ l'accès aux sommets adjacents à un sommet demandera systématiquement une recherche ;

D'ailleurs, dans un graphe, ce sont les relations d'incidence et d'adjacence qui sont en fait importantes. Il y a alors plusieurs structures de données possibles pour stocker en mémoire un graphe :

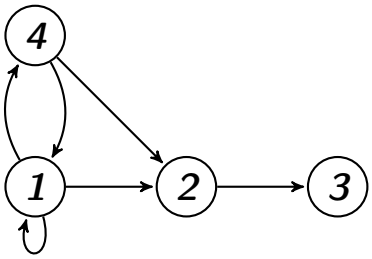
- la matrice d'adjacence ;
- la matrice d'incidence ;
- la matrice d'incidence ;
- la liste d'incidence.

2.1 A partir de la matrice d'adjacence

Un graphe $G = (S, A)$ orienté à n sommets (assimilés comme étant les entiers de 1 à n) peut être représenté par une matrice M de taille $n \times n$, appelée *matrice d'adjacence* et définie par :

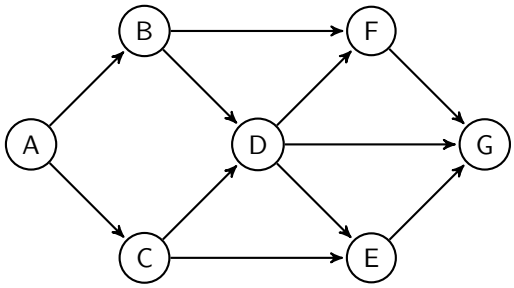
- les indices des lignes et des colonnes représentent les sommets de G.
- $M_{i,j}$ est égale au nombre d'arêtes reliant les sommets i et j.

Exemple 1. *La matrice d'adjacence du graphe suivant vaut :*



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

▸ **Exercice 1.** Donner la matrice d'adjacence du graphe suivant :



Le résultat sera noté dans un fichier nommé `matrice_adjacence.txt`, une ligne de la matrice sur une ligne du fichier, les nombres étant simplement espacés et sans espace à la fin d'une ligne !

Le contenu du fichier serait le suivant si la matrice était celle de l'exemple précédent :

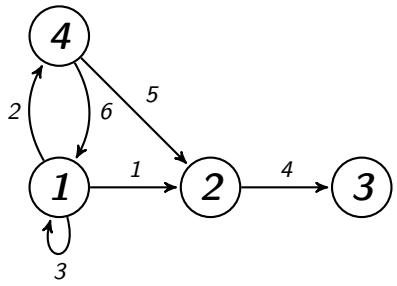
```
1 1 0 1
0 0 1 0
0 0 0 0
1 1 0 0
```

2.2 A partir de la matrice d'incidence

Un graphe $G = (S, A)$ orienté à n sommets (assimilés comme étant les entiers de 1 à n) et p arêtes (assimilées comme étant les entiers de 1 à p) peut être représenté par une matrice M de taille $n \times p$, appelée *matrice d'incidence* et définie par :

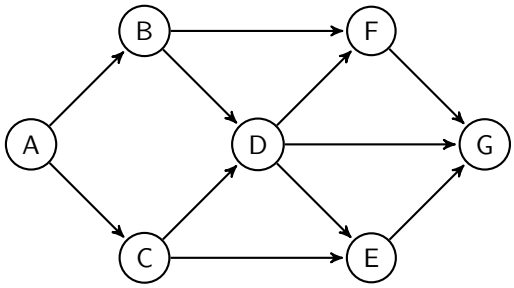
- les indices des lignes et des colonnes représentent respectivement les sommets de G et les arêtes de G.
- $M_{i,j}$ vaut : $\begin{cases} 1 & \text{lorsque le sommet } i \text{ est l'origine de l'arête } j. \\ -1 & \text{le sommet } i \text{ est la destination de l'arête } j. \\ 0 & \text{sinon.} \end{cases}$

Exemple 2. *La matrice d'incidence du graphe suivant vaut :*



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

▸ **Exercice 2.** Donner la matrice d'incidence du graphe suivant :



Le résultat sera noté dans un fichier nommé `matrice_incidence.txt`, une ligne de la matrice sur une ligne du fichier, les nombres étant simplement espacés et sans espace à la fin d'une ligne !

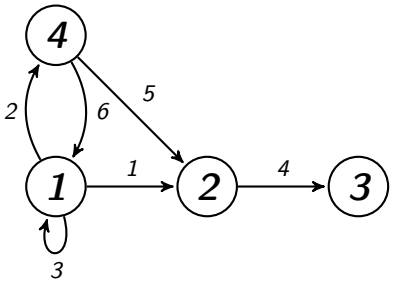
Le contenu du fichier serait le suivant si la matrice était celle de l'exemple précédent :

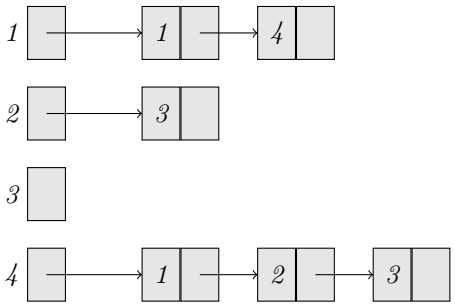
```
1 1 0 0 0 -1
-1 0 0 1 -1 0
0 0 0 -1 0 0
0 -1 0 0 1 1
```

2.3 A partir de la liste d'adjacence

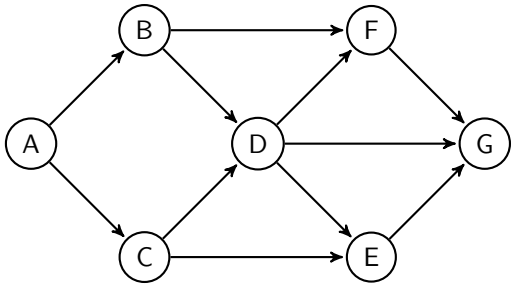
Un graphe $G = (S, A)$ orienté peut être représenté par la liste chainée des voisins de chaque sommet. Celle-ci s'appelle la liste d'adjacence de G.

Exemple 3. *La matrice d'incidence du graphe suivant vaut :*





▸ **Exercice 3.** Donner la liste d'adjacence du graphe suivant :



Le résultat sera noté dans un fichier nommé `liste_adjacence.txt`, une ligne par liste chainée, la liste chainée étant représentée par le sommet concernée suivi immédiatement de `:` puis de la suite des sommets que la liste chainée représente, ceux-ci étant étant simplement espacés et sans espace à la fin d'une ligne !

Le contenu du fichier serait le suivant si la liste d'adjacence était celle de l'exemple précédent :

1:1 4
2:3
3:
4:1 2 3

N.B. : L'ordre, à l'intérieur des listes chainées est l'ordre alphabétique, ou l'ordre sur des entiers, selon que les sommets soient nommés par des chaines de caractères ou des entiers.

3 Implémentation des graphes

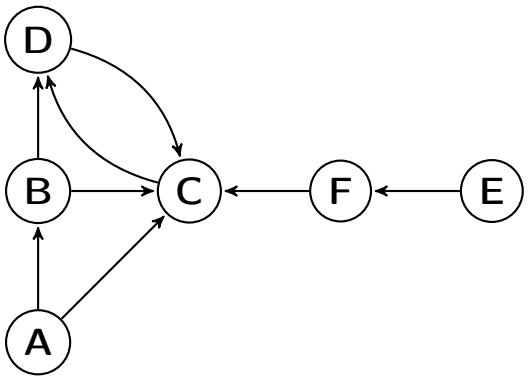
On opte ici pour une écriture des graphes sous forme de liste d'incidence. Pour cela, on suivra le conseil de Guido Van Rossum, le créateur de Python :

> "Few programming languages provide direct support for graphs as a data type, and Python is no exception. However, graphs are easily built out of lists and dictionaries. For instance, here's a simple graph (I can't use drawings in these columns, so I write down the graph's arcs):
>
> A –> B
> A –> C
> B –> C
> B –> D
> C –> D
> D –> C
> E –> F
> F –> C
>
> This graph has six nodes (A-F) and eight arcs. It can be represented by the following Python data structure:

```
graph = {'A': ['B', 'C'],
         'B': ['C', 'D'],
         'C': ['D'],
         'D': ['C'],
         'E': ['F'],
         'F': ['C']}
```

> This is a dictionary whose keys are the nodes of the graph. For each key, the corresponding value is a list containing the nodes that are connected by a direct arc from this node. This is about as simple as it gets (even simpler, the nodes could be represented by numbers instead of names, but names are more convenient and can easily be made to carry more information, such as city names)."

Le graphe considéré par Guido est le suivant :



▸ **Exercice 4.** Ecrire une classe `Graph` ayant trois attributs :

- un ensemble `nodes` représentant l'ensemble des noeuds du graphe ;
- une liste `edges` représentant la liste des arêtes du graphe ;
- un dictionnaire `adjacency_list` représentant la liste d'adjacence du graphe (comme décrit par Guido van Rossum).

Le constructeur doit uniquement permettre de créer le graphe vide.

Rappels : La classe `Graph` doit contenir une docstring, tout comme son constructeur et toutes les méthodes qu'elle contiendra. Celles-ci devront être en **anglais**.

▸ **Exercice 5.** Ajouter à la classe `Graph` une méthode `add_a_node(self, node_name)` permettant d'ajouter le sommet `node_name` au graphe courant.
On fera attention à ne pas recréer de sommet si celui-ci est déjà crée.

▸ **Exercice 6.** Ajouter à la classe `Graph` une méthode `add_an_edge(self, from_node, to_node):` permettant d'ajouter une arête au graphe courant entre les sommets `from_node` et `to_node`.
On fera attention à ne pas créer l'arête si les sommets passés en paramètres ne sont pas des sommets du graphe courant.

▸ **Exercice 7.** Dans un fichier `graph_example.py`, créer le graphe correspondant au graphe mentionné par Guido Van Rossum.

N'oubliez pas de commenter votre fichier de tests.

▸ **Exercice 8.** Ecrire la méthode `__str__(self)` permettant de représenter visuellement un graphe à la manière dont Guido Van Rossum l'a fait dans la citation précédente.

Précisément, si le graphe créée dans l'exercice précédent est disponible dans la variable `guido_s_graph`, alors la commande

```
print(guido_s_graph)
```

devra afficher précisemment en console :

```
************************
* Affichage d'un graphe *
************************
Nodes:
------

A, E, C, D, F, B

Edges:
------

A ---> B
A ---> C
B ---> D
B ---> C
C ---> D
D ---> C
E ---> F
F ---> C
==========================
```

4 Parcours d'un graphe

Un *parcours de graphe* est un algorithme consistant à explorer les sommets d'un graphe de proche en proche à partir d'un sommet initial. Ces algorithmes reviennent à de nombreuses reprises en théorie des graphes, notamment dans : la résolution des problèmes suivant :

- Connexité et forte connexité
- Existence d'un circuit ou d'un cycle (ce qu'on appelle tri topologique)
- Calcul des plus courts chemins (notamment l'algorithme de Dijkstra)

Il existe de nombreux algorithmes de parcours, les plus classiques étant le *parcours en largeur* et le *parcours en profondeur*. Pour pouvoir les implémenter simplement, nous allons avoir besoin de deux structures de données extrêmement importante en informatique : les *piles* et les *files*.

4.1 Notion de pile et de file

Les notions de *pile* et de *file* sont deux structures de données abstraites importantes en informatique. On limite la présentation de ces notions aux besoins des parcours de graphes envisagés ci-après.

La structure de pile (aussi appelée structure LIFO : last in, first out) est celle d'une pile d'assiettes:

- Pour ranger les assiettes, on les empile les unes sur les autres.
- Lorsqu'on veut utiliser une assiette, c'est l'assiette qui a été empilée en dernier qui est utilisée.

La structure de file (aussi appelée structure FIFO : first in, first out) est celle d'une file d'attente à un guichet :

- Les nouvelles personnes qui arrivent se rangent à la fin de la file d'attente.
- La personne servie est celle qui est arrivée en premier dans la file.

4.2 Parcours en largeur ou "breadth first search"

Lors d'un parcours en largeur d'un graphe, on traite en priorité les sommets découverts en premier. On utilisera donc une structure de file :

1. On enfile le sommet de départ.
2. On visite les voisins de la tête de file. On les enfile (en les numérotant au fur et à mesure de leur découverte) s'ils ne sont pas déjà présents dans la file, ni déjà passés dans la file.
3. On défile (c'est à dire que l'on supprime de la file l'élément en tête de la file).
4. On recommence les points 2 et 3 tant que c'est possible (c'est à dire tant que la file n'est pas vide).

L'algorithme de parcours en largeur va visiter en premier lieu tous les sommets du graphe à distance 1 du départ, puis tous les sommets à distance 2 du départ, puis tous les sommets à distance 3, $\cdots$ (c'est en fait cette propriété qui donne son nom à ce type de parcours).

Voici un exemple de parcours en largeur d'un arbre. En noir se trouvent les sommets traités, en gris sont les sommets situés sur la pile, et en blanc les sommets encore non découvert par le parcours.