TP 21 Python – Représentations des graphes

Il existe deux méthodes principales pour représenter (et implémenter) un graphe :

- à l'aide d'une matrice d'adjacence, qui indique pour chaque couple de sommets s'ils sont reliés ou non.
- à l'aide de listes d'adjacence, qui donnent pour chaque sommet l'ensemble de ses voisins.

1. Matrice d'adjacence

1.A) Définition

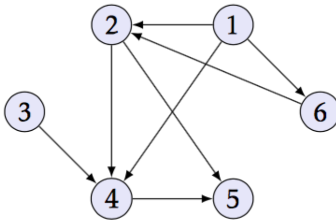
Définition

Soit $G = (S, A)$ un graphe (orienté ou non), où on écrit $S = \{s_1, \dots, s_n\}$ dans un **ordre précis**.
On appelle **matrice d'adjacence du graphe** G la matrice M carrée de taille n telle que pour tout $i, j \in [\![1, n]\!]$:

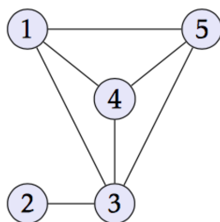
- $M_{i,j} = 1$ s'il existe une arête (ou un arc) allant de s_i à s_j
- $M_{i,j} = 0$ sinon.

Exemple

1. Quelle est la matrice d'adjacence associée au graphe orienté ci-dessous ?



2. Quelle est la matrice d'adjacence associée au graphe non orienté ci-dessous ?



3. Représenter un graphe dont la matrice d'adjacence est $M = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$.

Remarque. Dans un graphe **non orienté** :

- La matrice d'adjacence est toujours ....
- La matrice d'adjacence d'un graphe complet est ....
- Les voisins du sommet s_i sont les entiers $j \in [\![1, n]\!]$ tels que :

- On obtient le degré du sommet s_i par :

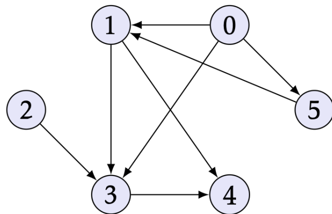
Dans un graphe **orienté** :

- La somme $\sum_{1 \leq i,j \leq n} M_{i,j}$ de tous les coefficients de la matrice d'adjacence correspond à ....
- On obtient le "degré entrant" d'un sommet par :
- On obtient le "degré sortant" d'un sommet par :

1.B) En Python

On représente ici un graphe G par sa matrice d'adjacence donnée dans une **liste de listes**.

Exercice 1. On considère le graphe ci-dessous.



1. Définir en Python ("à la main") la matrice A représentant ce graphe.
2. Prédire ce que doit renvoyer chacune des instructions suivantes, puis vérifier.

   a) `A[4][2]` b) `A[1]` c) `len(A[2])` d) `len(A)`
   e) `A[2][3]==0` f) `A[2][3]=0`

Exercice 2. Écrire une fonction `test(L)` qui prend en entrée une liste de listes L, et qui renvoie un booléen (`True` ou `False`) qui indique si L est bien la matrice d'adjacence d'un graphe non orienté, c'est-à-dire s'il s'agit bien d'une matrice carrée symétrique contenant des 0 et des 1.

Dans toute la suite, on identifie un graphe G à la liste de liste qui le représente. Si on note n l'ordre du graphe, on numérote les sommets de G de 0 à $n-1$ pour simplifier avec la numérotation de Python.

Exercice 3.

1. Écrire une fonction `sont_voisins(i,j,G)` qui prend en argument un graphe non orienté G et deux entiers i, j, et qui renvoie un booléen indiquant si les sommets i et j sont adjacents dans le graphe.
2. Écrire une fonction `liste_voisins(i,G)` qui prend en paramètres un graphe non orienté G et un entier i, et qui renvoie la liste de tous les voisins de i dans le graphe G.
3. Écrire une fonction `degré(i,G)` qui renvoie le degré du sommet i dans le graphe non orienté G.

Exercice 4.

1. Écrire une fonction `degré_orienté(k,G)` qui prend en argument un graphe orienté G et un entier k, et qui renvoie le degré entrant et le degré sortant du sommet k dans le graphe G.
2. Écrire une fonction `liste_degrés(G)` qui renvoie une liste de listes contenant pour chaque sommet, son degré entrant et son degré sortant.
3. Écrire les fonction `voisins_entrants(G)` et `voisins_sortants` qui renvoient chacune un **dictionnaire**, qui associe à chaque sommet de G la liste de ses voisins entrants ou sortants.

Exercice 5. Écrire une fonction `est_complet(G)` qui renvoie un booléen qui indique si un graphe donné en paramètre est complet ou non.

2. Listes d'adjacence

2.A) Définition

Définition

Soit $G = (S, A)$ un graphe (orienté ou non). On appelle **listes d'adjacence** l'ensemble des listes qui contiennent, pour chaque sommet, l'ensemble de ses voisins (sortants dans le cas d'un graphe).
Il s'agit donc d'un ensemble de n listes, une pour chaque sommet.

Exemple

1. Représenter le graphe orienté associé aux listes d'adjacences :

   $1 : \{2, 3, 4\}$ $\quad$ $2 : \{4\}$ $\quad$ $3 : \{3, 1\}$ $\quad$ $4 : \{3, 2\}$

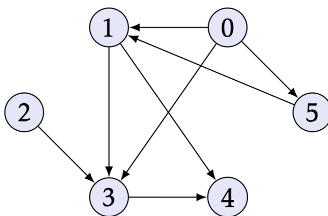
2. Donner les listes d'adjacence associées à chacun des deux graphes du premier exemple de ce TP.

2.B) Implémentation Python

En Python, on a plusieurs choix pour représenter les listes d'adjacences :

- On peut écrire une liste de listes, contenant dans l'ordre les listes d'adjacence de chaque sommet.
- On peut écrire un dictionnaire qui associe à chaque sommet la liste d'adjacence correspondante. C'est ce qu'on privilégie lorsque les sommets ne sont pas les entiers de 0 à $n - 1$.

Exercice 6. On reprend le graphe G suivant :



1. Définir en Python ("à la main") la liste L contenant les listes d'adjacences de G.
2. Définir en Python ("à la main") le dictionnaire d représentant G par ses listes d'adjacence.
3. Prédire ce que doit renvoyer chacune des instructions suivantes, puis vérifier.

   a) `L[1]` $\quad$ b) `d[4]` $\quad$ c) `len(d)` $\quad$ d) `len(A[0])`

Exercice 7. On représente ici les graphes par la liste des listes d'adjacence.

1. Quelle commande Python permet d'obtenir le nombre de sommets d'un graphe G ?
2. Écrire une fonction `taille(G)` qui prend en argument un graphe G et qui renvoie son nombre d'arêtes.

Exercice 8. On considère dans cet exercice des graphes *orientés* représentés par leur liste de listes d'adjacence.

1. Écrire une fonction `degré_sortant(i,G)` qui prend en entrée un graphe G et un entier i, et qui renvoie le degré sortant du sommet i.

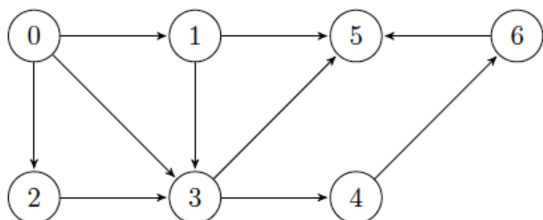
2. Écrire une fonction `voisins_entrants(i,G)` qui prend en entrée un entier i et un graphe G et qui renvoie la liste des sommets j tels qu'il existe un arc de j à i dans le graphe G.
3. En déduire une fonction `degré_entrant(i,G)`.
4. Qu'est-ce qui change si on utilise des **dictionnaires** de listes d'adjacences ?

3. Choix de la représentation

On a vu que l'on pouvait représenter un graphe en Python par une matrice d'adjacence ou par une liste d'adjacence.
Selon le type de graphe et selon ce que l'on souhaite faire, il peut être plus pertinent d'utiliser l'une ou l'autre de ces représentations, notamment pour des considérations de mémoire et de temps d'exécution, mais aussi pour la facilité de réaliser telle ou telle opération à partir du graphe.

De manière générale on retiendra que l'on privilégie les **listes d'adjacence lorsqu'il y a peu d'arêtes** (ou d'arcs), par rapport au nombre de sommets, on parle alors de graphes **creux** et qu'on choisit les matrices d'adjacence lorsqu'il y a beaucoup d'arêtes (ou d'arcs), on parle alors de graphes **denses**.

Exercice 9.



1. Donner le nombre total d'éléments dans la matrice d'adjacence du graphe, et faire de même pour les listes d'adjacence.
2. En terme d'espace mémoire, quelle représentation privilégier ?

Exercice 10. 1. Quel type de représentation choisir pour représenter le graphe dont les sommets sont les villes de France et les arêtes les routes les reliant ?

2. De manière générale, si un graphe non orienté possède n sommets et m arêtes, combien d'éléments contient la matrice d'adjacence ? la liste d'adjacence ?
3. Même question pour un graphe orienté.

Exercice 11. 1. Écrire une fonction Python `matrice_to_liste(R)` qui prend un graphe représenté par sa matrice d'adjacence et qui renvoie sa représentation par sa liste d'adjacence.

2. Écrire une fonction Python `liste_to_matrice(L)` qui prend un graphe représenté par sa liste d'adjacence et qui renvoie sa représentation par sa matrice d'adjacence.

Exercice 12. Quel type de représentation choisir pour…

1. donner la liste des voisins d'un sommet dans un graphe non orienté ?
2. déterminer si deux sommets sont voisins dans un graphe non orienté ?
3. déterminer le degré entrant d'un sommet dans un graphe orienté ?
4. ajouter un arc entre deux sommets donnés d'un graphe orienté ?
   Écrire la fonction associée.
5. supprimer une arc entre deux sommets donnés d'un graphe orienté ?
   Écrire la fonction associée. On renverra un message d'erreur si l'arc n'existe pas.
6. supprimer un sommet donné d'un graphe non orienté, ainsi que toutes les arêtes qui en sortent.
   Écrire la fonction associée.

4. Parcours de graphe

Il existe plusieurs manière de "parcourir" un graphe, c'est-à-dire de l'explorer **de proche en proche** en partant d'un sommet initial donné.
On voit ici le **parcours en largeur** et quelques unes de ses applications possibles.
L'idée est la suivante :

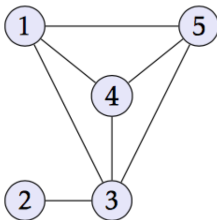
- On part d'un sommet initial s_0.
- On garde en mémoire une liste des sommets déjà visités.
- À chaque fois qu'on visite un sommet pour la première fois, on ajoute tous ses voisins à une "file d'attente", pour les visiter à leur tous plus tard.
- On s'arrête lorsque la file d'attente est vide.

Pourquoi est-il plus simple de passer par les listes d'adjacence ?

Compléter la fonction suivante pour qu'elle réalise un parcours en profondeur, à partir d'un graphe G représenté par le dictionnaire de ses listes d'adjacence, et d'un sommet initial s_i.

```
def parcours_en_largeur(G, s_i):
    visités = []
    file = [s_i]
    while file ............ :
        sommet = file[0]
        if sommet in visités:
            file = file[1:] # On retire le sommet de la file d'attente
        else:
            visités.append(sommet)
            file = .............................. # On retire le sommet et on ajoute tous ses voisins !
    return visités
```

Exercice 13. 1. Appliquer à la main la fonction ci-dessus au graphe suivant en partant du sommet 1. On donnera les listes `visités` et `file` à chaque étape de l'algorithme.



Vérifier votre résultat avec Python.

2. Que se passe-t-il si on applique la fonction à un graphe qui n'est pas connexe ?
3. (**) Adapter la fonction `parcours_en_largeur` pour qu'elle renvoie une liste de couples (s, d) où d est la **distance** entre s et le sommet initial s_i. On pourra commencer par l'instruction `file = [(s_i, 0)]`, et manipuler des couples (s, d) tout au long du programme.
4. (**) On modifie légèrement l'algorithme de la manière suivante : lorsqu'on visite un sommet pour la première fois, on ajoute tous ses voisins **au début de la file d'attente** au lieu de les ajouter à la fin. Reprendre la question 1, et expliquer à partir de cet exemple et d'autres (comme un arbre généalogique) qu'on parle alors de **parcours en profondeur**.