TP de Python numéro 9 : Graphes

*Semaine du **jeudi 12 février**.*

Dans ce TP, on étudie les deux représentations classiques des graphes en informatique (par matrice d'adjacence ou par liste des adjacences) et on programme les fonctions de conversion d'une représentation vers l'autre. Ensuite, on utilise les résultats du programme de mathématiques liés aux chaînes, aux chemins et à la connexité pour écrire des algorithmes classiques sur ces notions.

Dans tout ce TP, les graphes seront orientés ou non, et auront pour ensemble de sommets un ensemble de la forme $[\![0, n-1]\!]$ (où $n \in \mathbb{N}^*$ est l'ordre du graphe) de sorte que ces sommets sont naturellement numérotés, numérotation qu'on utilisera pour parler de *la* matrice d'adjacence d'un graphe considéré.

Cette numérotation à partir de 0 est plus simplement compatible avec la numérotation de `Python`.

Remarque : tout ce qui est expliqué dans ce TP est autant valide pour les graphes orientés que pour les graphes non orientés, mais pour utiliser ces résultats, il faut tout de même préciser le cadre (orienté ou non) dans lequel on se place.

I. Représentations informatiques des graphes

Il existe deux moyens classiques de représenter un graphe en informatique : en utilisant la matrice d'adjacence, ou en utilisant la liste des adjacences.

1. Représentation par la matrice d'adjacence

On peut tout d'abord représenter un graphe en donnant sa matrice d'adjacence. Par exemple, Considérons les graphes $\mathcal{G}$ et $\mathcal{H}$ représentés ci-dessous :

Notons $M_{\mathcal{G}}$ et $M_{\mathcal{H}}$ leurs matrices d'adjacence respectives. Alors :

$$M_{\mathcal{G}} = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix} \text{ et } M_{\mathcal{H}} = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

et on peut reconstruire ces graphes à partir de ces matrices. On peut donc représenter $\mathcal{G}$ et $\mathcal{H}$ en Python en donnant leurs matrices d'adjacences `MG` et `MH` données par :

```
MG=[[0,1,1,0],[1,0,0,1],[1,0,0,1],[0,1,1,0]]
MG=np.array(MG)
MH=[[0,1,1,0],[0,0,0,0],[0,0,0,1],[0,0,0,0]]
MH=np.array(MH)
```

Exercice 1.
1. Représenter le graphe (non orienté) complet K_5 d'ordre 5. Donner sa matrice d'adjacence. Définir cette matrice `K5` en Python, sans saisir ses coefficients à la main.
2. Écrire le code d'une fonction Python d'entête `def MatriceK(n):` prenant en entrée un entier naturel non nul `n` et renvoyant en sortie la matrice d'adjacence du graphe complet d'ordre `n`.

Exercice 2. 1. Écrire une fonction Python d'entête `def DegSortants(M):` prenant en entrée la matrice d'adjacence `M` d'un graphe orienté G et renvoyant en sortie la liste (dans l'ordre de ces sommets) des degrés sortants des sommets de $\mathcal{G}$.

2. Coder de même une fonction d'entête `def DegEntrant(M):` prenant en entrée la matrice d'adjacence `M` d'un graphe orienté G et renvoyant la liste des degrés entrants des sommets de $\mathcal{G}$.

3. Tester ces deux fonctions sur le graphe $\mathcal{H}$ de l'exemple ci-dessus.

2. Représentation par la liste des adjacences

Plutôt que d'utiliser leur matrice d'adjacence, on peut représenter des graphes en donnant leur *liste des adjacences*.

Soit s un sommet d'un graphe $\mathcal{G}$. **La liste des adjacences** du sommet s est la liste, qu'on notera $V(s)$, formée par les sommets

- adjacents à s si $\mathcal{G}$ est non orienté,
- qui sont le but d'une arête d'origine s si $\mathcal{G}$ est orienté.

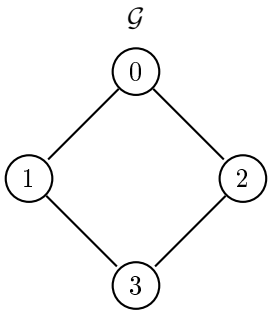
Pour représenter un graphe $\mathcal{G}$ d'ordre n, on donne alors la liste

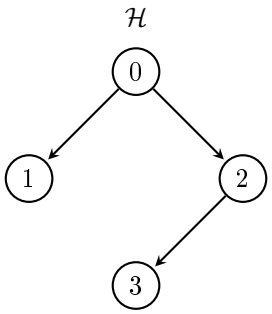
$$V = [V(0), V(1), \dots, V(n-1)]$$

de ces liste d'adjacence.

Cette liste V est appelée **la liste d'adjacence** du graphe $\mathcal{G}$ (c'est une liste de listes). Il est clair qu'on peut retrouver un graphe $\mathcal{G}$ à partir de sa liste des adjacences (on peut reconstituer son ordre et toutes ses arêtes).

Par exemple, reprenons les exemples précédents.





- **Dans le graphe $\mathcal{G}$ non orienté**, les listes d'adjacence `V0,V1` des sommets `0` et `1` sont données par

```
V0=[1,2] # le sommet 0 est adjacent aux sommets 1 et 2
V1=[0,3] # le sommet 1 est adjacent aux sommets 0 et 3
```

et la liste d'adjacence de $\mathcal{G}$ est la liste `V` donnée par :

```
V=[[1,2],[0,3],[0,3],[1,2]]
```

- **Dans le graphe $\mathcal{H}$ orienté**, les listes d'adjacence `V0,V1` des sommets `0` et `1` sont données par

```
V0=[1,2] # Les arêtes de source 0 ont pour but 1 et 2
V1=[] # Aucune arête n'a pour source le sommet 1
```

et la liste des adjacences de $\mathcal{G}$ est la liste `V` donnée par :

```
V=[[1,2],[],[3],[]]
```

Exercice 3. 1. Déterminer à la main la liste d'adjacence du graphe complet (non orienté) K_4 d'ordre 4.

2. A l'aide d'une boucle, définir en Python la liste d'adjacence du sommet 0.

3. Sans la renter à la main, définir en python la liste des adjacences de K_4.

4. Écrire le code d'une fonction python d'entête `def ListeAdjacenceK(n):` prenant en entrée un entier naturel non nul `n` et renvoyant en sortie la liste des adjacences du graphe complet d'ordre `n`.

Exercice 4.

1. Écrire une fonction Python d'entête `def estArete(V,i,j):` prenant en entrée la liste des adjacences d'un graphe (orienté ou non), ainsi que deux sommets `i` et `j` de ce graphe, et renvoyant en sortie `True` si ce graphe comporte une arête de `i` vers `j`, et `False` sinon.
2. Écrire une fonction Python d'entête `def AfficheDescriptionNO(V):` prenant en entrée la liste des adjacences d'un graphe `G` non orienté, et affichant (sans sortie) de manière lisible l'ordre de `G` et la liste des degrés des sommets de `G`.
3. Écrire une fonction Python d'entête `def AfficheDescriptionO(V):` prenant en entrée la liste des adjacences d'un graphe `G` orienté, et affichant (sans sortie) de manière lisible l'ordre de `G`, la liste des degrés sortants des sommets de `G`, et la liste des degrés entrants des sommets de `G`.

3. Fonctions de conversion

On dispose de deux manières de représenter un graphe : par sa matrice d'adjacence, ou par sa liste des adjacences. Dans tous les cas, l'information retenue permet de reconstruire le graphe.

On peut alors avoir besoin de passer d'une représentation à l'autre.

Remarque. Même s'il existe des différences entre le cas des graphes orientés et celui des graphes non orientés, les fonctions demandées dans les deux exercices suivants doivent fonctionner indifféremment dans ces deux cas.

Exercice 5.

1. Écrire la matrice d'adjacence du graphe orienté ayant pour liste des adjacences :

   `[[1,2,3],[0,2],[0,4],[2,4],[0,1,2,3]]`

2. Écrire une fonction python d'entête `def conversion_Liste_vers_Matrice(V):` prenant en entrée une liste `V` qui est la liste des adjacences d'un graphe (orienté ou non), et renvoyant en sortie la matrice d'adjacence de ce graphe.
3. Tester votre fonction à l'aide des fonctions `ListeAdjacenceK` et `MatriceK` des exercices précédents.

Pour la 2., on pourra partir du code incomplet suivant :

```
def conversion_Liste_vers_Matrice(V):
    ordre =... #ordre du graphe considéré
    M=np.zeros((ordre,ordre)) #matrice d'adjacence à remplir
    for i in range(ordre): #pour chaque sommet i :
        #remplissage de la i ième ligne de M,
        #en utilisant la liste V[i]
        (...)
    return(M)
```

Exercice 6.

1. Écrire la liste d'adjacence du graphe non orienté ayant la matrice d'adjacence :

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

2. Écrire une fonction Python d'entête `def conversion_Matrice_vers_Liste(M):` prenant en entrée une matrice `M` qui est la matrice d'adjacence d'un graphe (orienté ou non), et renvoyant en sortie la liste d'adjacence de ce graphe.
3. Tester votre fonction à l'aide des fonctions `ListeAdjacenceK` et `MatriceK` des exercices précédents.

II. Matrice d'adjacence, chaînes et chemins, connexité

1. Distance entre deux sommets dans un graphe non orienté

Définition 7. Soit G un graphe non orienté.
Pour tous sommets s et s' de G, on appelle ***distance (combinatoire) de s à s'*** et on note $\mathrm{d}(s,s')$ l'élément de $\mathbb{R}_+ \cup \{+\infty\}$ donné par :

- $\mathrm{d}(s,s')$ est la longueur de la plus courte chaîne de s vers s' si s et s' sont reliés par une chaîne,
- $\mathrm{d}(s,s') = +\infty$ si s et s' ne sont par reliés par une chaîne.

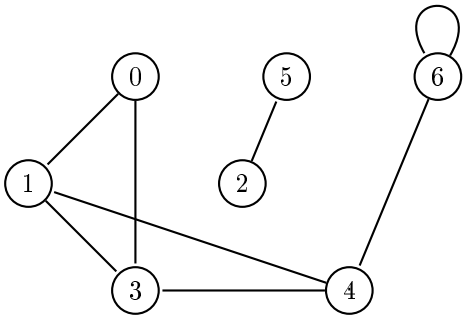
On définit ainsi une application $\mathrm{d} : S^2 \to \mathbb{R}_+ \cup \{+\infty\}$.

Remarque. Dans certains cas, il existe d'autres notions de distance sur les sommets d'un graphe : dans ce cas, celle vue ici est désignée comme étant la distance combinatoire.

Remarque. La distance d'un sommet s à lui même est donc 0, car il existe une chaîne de longueur 0 de s vers s : la chaîne $(s) \in S^1$.

Exemple 8. Dans le graphe G ci-dessous...

G



- $\mathrm{d}(0,3) = \ldots$
- $\mathrm{d}(0,4) = \ldots$
- $\mathrm{d}(5,2) = \ldots$
- $\mathrm{d}(5,0) = \ldots$
- $\mathrm{d}(6,1) = \ldots$
- $\mathrm{d}(6,6) = \ldots$

Pour travailler avec $+\infty$, on adopte les conventions "naturelles" suivantes :

- $(+\infty) + (+\infty) = +\infty$
- $a + (+\infty) = +\infty$ pour tout réel a,
- $a \leq +\infty$ pour tout réel a,
- $+\infty \leq +\infty$.

Proposition 9. *Soit $G = (S, A)$ un graphe non orienté. Soient s et s' deux sommets de G. Alors :*

1. $\mathrm{d}(s,s') \geq 0$ *(positivité)*
2. $\mathrm{d}(s,s') = 0 \iff s = s'$ *(propriété dite de séparation).*
3. $\mathrm{d}(s,s') = \mathrm{d}(s',s)$ *(propriété dite de symétrie).*
4. *Pour tout autre sommet s'',* $\mathrm{d}(s,s'') \leq \mathrm{d}(s,s') + \mathrm{d}(s',s'')$ *(on dit que* d *vérifie l'inégalité triangulaire).*

Exercice 10. Démontrer ces propriétés.

Remarque. Si le graphe G considéré est connexe, on dit alors que d définit une distance sur l'ensemble des sommets de G (la notion générale de distance considère une application à valeurs dans $\mathbb{R}_+$, donc non infinie).

L'infini de numpy

Supposant `numpy` importé avec l'alias `np`, la commande `np.inf` sert à représenter $+\infty$ en accord avec les conventions ci-dessus pour la somme et la comparaison.

Exemple 11. Exécuter et observer, dans l'invite de commande, les lignes suivantes.

```
>>>import numpy as np
>>>print(np.inf)
>>>np.inf+3
>>>np.inf+np.inf
>>>np.inf-1
>>>np.inf<2
>>>2 <= np.inf
```

2. Informatique

Pour cette partie, on représentera un graphe à l'aide de sa matrice d'adjacence. On utilisera les commandes Python au programme sur les matrices, vues lors du TP précédent.

Exercice 12. Les fonctions demandées dans cet exercice doivent fonctionner indifféremment pour un graphe orienté ou non orienté.

1. Écrire le code d'une fonction Python d'entête `def NombreChemins(M,i,j,k):` prenant en entrée :
   - la matrice d'adjacence `M` d'un graphe $\mathcal{G}$
   - les numéros `i` et `j` de deux de ses sommets,
   - et un entier `k`,

   et renvoyant en sortie le nombre de chaînes de longueur `k` (ou chemins si $\mathcal{G}$ est orienté) du sommet `i` vers le sommet `j`.
2. Écrire le code d'une fonction Python d'entête `def distance(M,i,j):` prenant en entrée la matrice d'adjacence `M` d'un graphe $\mathcal{G}$ et les numéros `i` et `j` de deux de ses sommets, et renvoyant en sortie :
   - la longueur du plus court chemin (ou de la plus courte chaîne dans le cas non orienté) du sommet `i` vers le sommet `j` si un tel chemin (ou une telle chaîne) existe,
   - `np.inf` s'il n'existe pas de chemin (ou de chaîne) de `i` vers `j`

   On pensera au lemme utilisé dans la démonstration du théorème liant la connexité d'un graphe à sa matrice d'adjacence : dans un graphe d'ordre n, s'il existe un chemin (ou une chaîne) d'un sommet i vers un sommet j, alors il existe un chemin de i vers j de longueur au plus $n-1$.
3. Tester vos fonctions sur la matrice d'adjacence du graphe de l'exemple 8 avec quelques sommets.

   Conseil : saisissez à la main la liste des adjacences de ce graphe, et convertissez celle-ci à l'aide de votre fonction de l'exercice 5.

Remarque : le code précédent ne donne pas le plus court chemin (s'il existe) entre deux sommets, juste sa longueur. Ce problème sera abordé à nouveau dans un TP ultérieur, où on abordera l'algorithme de Dijkstra.

3. Connexité, classe de connexité d'un sommet

Tout d'abord, un certain théorème du cours permet de facilement résoudre l'exercice suivant.

Exercice 13. Écrire le code d'une fonction Python d'entête `def estConnexe(M):` prenant en entrée la matrice d'adjacence `M` d'un graphe $\mathcal{G}$ (orienté ou non) et renvoyant en sortie `True` si $\mathcal{G}$ est connexe ("fortement" dans le cas orienté), et `False` sinon.

Voyons une autre notion, liée à la notion de composante connexe d'un graphe.

Définition 14. Soit G un graphe **non orienté**, et s un sommet de G.
On appelle **classe de connexité** de s dans G l'ensemble, noté $\mathcal{C}(s)$ dans ce TP, des sommets de G reliés à s par une chaîne.

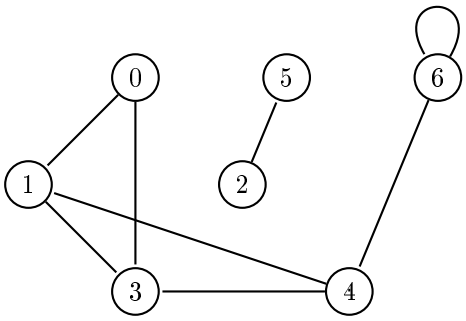
Remarque. Autrement dit, si $G=(S,A)$ est un graphe non orienté, alors pour tout $s\in S$:

$$\mathcal{C}(s)=\{s'\in S\,|\,\mathrm{d}(s,s')\in\mathbb{R}\}$$

Remarque. La classe de connexité d'un sommet s (dans un graphe G fixé) est donc l'ensemble des sommets de G intervenant dans la composante connexe de G contenant s.

Exemple 15. Dans le graphe G ci-dessous (de l'exemple 8)...

G



On observe les classes de connexité suivantes :

$$\mathcal{C}(1) = \{0, 1, 3, 4, 6\}$$

et

$$\mathcal{C}(2) = \{2, 5\}.$$

Exercice 16.
1. A l'aide de la fonction de l'exercice 12, écrire une fonction d'entête `def ClasseConnexite(M,s):` prenant en entrée la matrice d'adjacence `M` d'un graphe non orienté G et un sommet `s` de G, et renvoyant en sortie la classe de connexité $\mathcal{C}$(`s`) du sommet `s`, sous forme d'une liste.
2. Testez votre fonction sur le graphe de l'exemple 8.

On dispose d'un algorithme plus intéressant pour établir cette liste des adjacences, pour les raisons suivantes

- il est plus économe en calcul, donc plus approprié dans le cas d'énormes graphes,
- il se généralise à des situations plus compliquées, où l'on ne peut avoir recours à la matrice d'adjacence.

Pour établir la classe de connexité $\mathcal{C}(s)$ d'un sommet s donné d'un graphe (non orienté) G, on suit l'algorithme suivant :

Algorithme

1. **Initialisation des variables :** On définit deux listes `T` ("sommets à traiter") et `C` ("classe de connexité"). Initialement, `T=[`s`]` et `C=[]`.
2. **Itérations :** Tant que `T` n'est pas la liste vide, on "traite" son premier élément :
   - On considère le premier élément `t` de `T`.
   - On détermine la liste `V` des sommets adjacents à `t` qui ne sont ni dans `T` ni dans `C`.
   - On ajoute les sommets de la liste `V` à la liste `T`.
   - On supprime `t` de la liste `T` et on ajoute `t` à la liste `C`.
3. A la fin de cette boucle (qui s'arrête : pourquoi ?), `C` contient la liste de sommets dans la classe de connexité de s (sans répétitions).

Exercice 17. Appliquer cet algorithme à la main sur le graphe de l'exemple 8 pour déterminer la classe de connexité du sommet 6.

Exercice 18. Écrire le code d'une fonction Python `ClasseConnexite2(L,s):` prenant en entrée la liste des adjacences `L` d'un graphe (non orienté) G et un sommet `s` de G, et renvoyant en sortie la classe de connexité de `s` (dans G) sous forme de liste. On pourra s'inspirer du code à trou en début de page suivante. Puis, vérifier cette fonction sur le graphe de l'exercice 8.

```python
def ClasseConnexite2(L,s):
    # Initialisation des variables
    T=[s]
    C=[]
    # Itérations
    while len(T)!=0:
        # Traitement du premier élément de T
        t=T[0]
        # Détermination des voisins de t ni dans T ni dans C
        V=[ v for v in ... if not(... ...  ...) ]
        # Ajout des éléments de V à la fin de T
        T=...
        # Fin du traitement de t
        ... # suppression de t dans T
        ... # ajout de t dans C
    return(C)
```

III. Un exercice sur les graphes bipartis

Exercice 19. Pour simplifier, dans cet exercice, les sommets des graphes $K_{n,m}$ considérés sont les entiers de 1 à $n+m$. On prendra garde, pour les questions informatiques, au décalage de 1 occasionné par l'habitude des informaticiens de commencer leurs décomptes à 0...

1. Rappeler la définition d'un graphe biparti.

   Pour tout $(n,m) \in (\mathbb{N}^*)^2$, on pose $S_{n,m} = [\![1, n+m]\!]$, $G_{n,m} = [\![1,n]\!]$, $D_{n,m} = [\![n+1, n+m]\!]$, et $A_{n,m} = \{\{i,j\}, i \in G_{n,m}, j \in D_{n,m}\}$.

   On pose enfin $K_{n,m} = (S_{n,m}, A_{n,m})$. On dit que $K_{n,m}$ est un graphe biparti complet.

2. Représenter $K_{2,4}$ et $K_{3,1}$. Donner la matrice d'adjacence et la liste des adjacences de ces graphes (avec la numérotation induite par les sommets).

3. Justifier la terminologie de "graphe biparti complet".

4. Définir en Python, et sans rentrer ces données à la main, la matrice d'adjacence et la liste des adjacences de $K_{2,3}$.

5. Écrire une fonction `Python` d'entête `def MatriceBiparti(n,m):` prenant en entrée des entier `n,m` (non nuls) et renvoyant en sortie la matrice d'adjacence de $K_{n,m}$.

6. Écrire une fonction `Python` d'entête `def ListeBiparti(n,m):` prenant en entrée des entier `n,m` (non nuls) et renvoyant en sortie la liste des adjacences de $K_{n,m}$.

7. Donner le nombre d'arêtes de $K_{n,m}$ en fonction de n et m.

8. Afficher, pour quelques valeurs de n et m et de manière lisible, les 6 premières puissances de la matrice d'adjacence de $K_{n,m}$.

9. Conjecturer un résultat portant sur la longueur des chaînes fermées de $K_{n,m}$.

10. Démontrer cette conjecture.

11. Conjecturer un résultat portant sur le nombre de chaînes de longueur impaire de $K_{n,m}$ entre deux sommets donnés de ce graphe.

12. Démontrer cette conjecture.