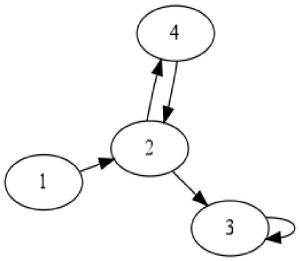
Fiche 9 - Graphes

I. Le vocabulaire

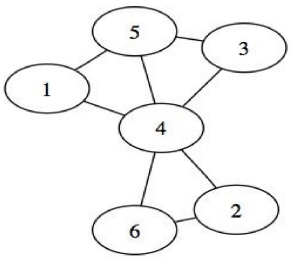
Définition – Graphe
*Un **graphe** est un couple $G = (S, A)$ où S est un ensemble fini de points, appelés **sommets** ou **noeuds**, et A un sous-ensemble de $S \times S$, vu comme des liens entre les points.*

- *L'**ordre** d'un graphe est le cardinal de S.*
- *Une **boucle** est une lien d'un sommet vers lui-même, c'est-à-dire un élément de A du type (s, s).*

Remarque – Un lien entre deux sommets peut être à double sens ou à sens unique, cela donne la notion de graphe orienté.



G_1 est orienté



G_2 est non orienté

Définition – Graphe non orienté
*Le graphe est dit **non orienté** lorsque la relation binaire A est symétrique : pour tout $s_1, s_2 \in S$,*

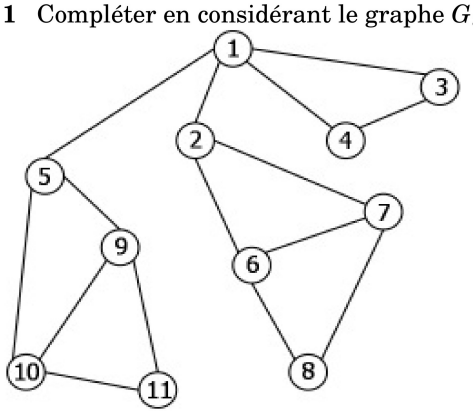
$$(s_1, s_2) \in A \iff (s_2, s_1) \in A$$

- *Les liens sont appelés **arêtes**.*
- *Une **chaîne** entre deux sommets $s_1, s_2 \in S$ est une suite finie d'arêtes allant de s_1 à s_2.*
- *La **longueur** d'une chaîne est le nombre de ses arêtes.*
- *Un **cycle** est une chaîne dont les deux sommets extrémités sont identiques.*
- *Deux sommets $s_1, s_2 \in S$ sont dit **voisins** si $(s_1, s_2) \in A$*
- *Le **degré** d'un sommet $s \in S$, noté $d(s)$, est le nombre de ses voisins*

$$d(s) = \text{Card}\{v \in S, (s, v) \in A\}$$

- *Un graphe non orienté est **connexe** s'il est d'un seul tenant, c'est-à-dire pour tout couple $(s_1, s_2) \in S^2$, il existe une chaîne de s_1 à s_2.*

Exercice 1 Compléter en considérant le graphe G_3 :



- L'ordre de G_3 est : ...
- Donner une chaîne entre s_8 et s_9 et préciser sa longueur :

...

- Donner un cycle : ...
- Donner les degrés suivants :

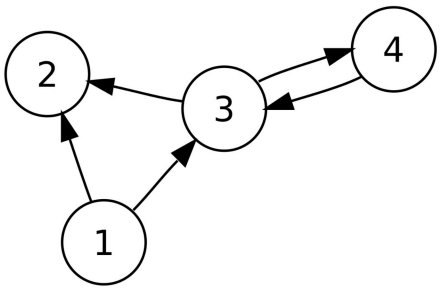
$$d(s_8) = \ldots \qquad d(s_9) = \ldots \qquad d(s_1) = \ldots$$

- Donner des exemples de graphes connexes et non connexes.

Définition – Graphe orienté
*Le graphe est dit **orienté** lorsque la relation binaire n'est pas symétrique.*

- *Les liens sont appelés **arcs**.*
- *Un **chemin** d'un sommet $s_1 \in S$ à un sommet $s_2 \in S$ est une suite finie d'arcs allant de s_1 à s_2.*
- *La **longueur** d'un chemin est le nombre de ses arcs.*
- *Un **circuit** est un chemin dont les deux sommets extrémités sont identiques.*
- *Considérant deux sommets $s_1, s_2 \in S$, s_2 est un voisin orienté de s_1 si $(s_1, s_2) \in A$.*
- *Les degrés orientés d'un sommet sont définis par :*
  *le **degré sortant** : $d_+(s) = \text{Card}\{v \in S, (s, v) \in A\}$*
  *le **degré entrant** : $d_-(s) = \text{Card}\{v \in S, (v, s) \in A\}$*

Exercice 2 Compléter en considérant le graphe G_4 :



- Donner un chemin entre s_1 et s_2 de longueur 4 :

...

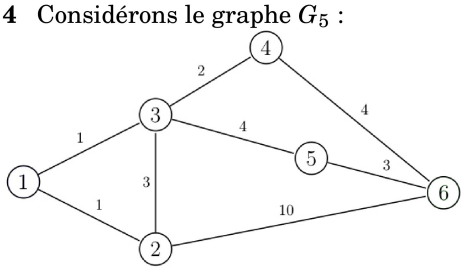
- Donner un circuit : ...
- Donner les degrés suivants :

$$d_-(s_2) = \ldots \qquad d_+(s_2) = \ldots \qquad d_-(s_3) = \ldots \qquad d_+(s_3) = \ldots$$

Exercice 3 Donner une relation entre le nombre de liens et les degrés d'un graphe $G(S, A)$.

Définition – Graphe pondéré
*On appelle graphe **pondéré** lorsque à chaque lien entre deux sommets est associée une valeur, appelée **poids**.*
Le poids d'une chaînes d'un graphe non orienté [resp. d'un chemin d'un graphe orienté] est la somme du poids de ses arêtes [resp. ses arcs].

Exercice 4 Considérons le graphe G_5 :



Quel est la chaîne de poids minimal entre s_2 et s_{10} ?

➜ Exemples de situations modélisées par un graphe :

▶ réseau de transport :
- les sommets peuvent être les intersections, les villes, les aéroports ;
- les arêtes sont les routes (les arcs en cas de sens unique), les vols ;
- les poids peuvent être le temps, la distance, le coût.

▶ réseau de distribution d'électricité :
- les sommets sont les sites de production, de transformation et de consommation ;
- les arcs ou arêtes sont les lignes électriques ;
- les poids sont les distances.

▶ réseaux sociaux :
- les sommets sont les personnes ;
- les arêtes sont les amitiés ;
- les poids sont les nombres d'interactions.

▶ graphe du web :
- les sommets sont les pages ;
- les arcs sont les liens hypertextuels.

II. Représentation d'un graphe

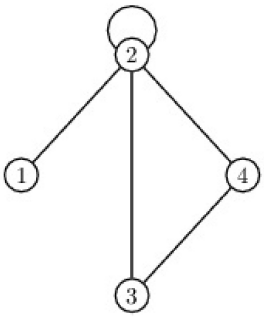
II.1. Matrice d'adjacence

Définition – Matrice d'adjacence
Soit $G=(S,A)$ un graphe où $\text{Card}(S)=n\geq 2$. *On appelle* ***matrice d'adjacence*** *de G la matrice $M\in\mathcal{M}_n(\mathbb{R})$ telle que*

$$\forall i,j\in[\![1,n]\!],\qquad m_{i,j}=\begin{cases}1 & si\ (s_i,s_j)\in A\\ 0 & sinon\end{cases}$$

La matrice d'adjacence d'un graphe donne une représentation de A : elle indique sur l'existence d'un arc en deux sommets.

Exercice 5 Donner la matrice d'adjacence du graphe G_1 et celle du graphe G_6 suivant :



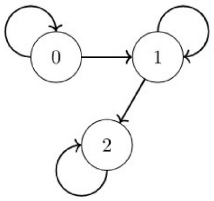
Que dire d'un graphe dont la matrice d'adjacente est symétrique.

Proposition – *Soit M la matrice d'adjacente du graphe $G=(S,A)$ orienté. Pour tout $n\in\mathbb{N}$, $\mathscr{C}_{i,j}(M^n)$ est le nombre de chemins de longueur n allant du sommet s_i au sommet s_j.*

Exercice 6 Faire la démonstration par récurrence.

Exercice 7 *Nombre de chemins de longueur fixée*

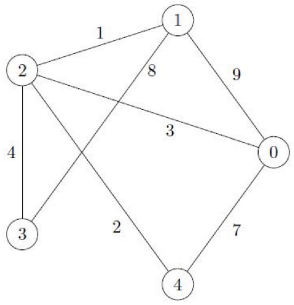
1. Donner une fonction `nb_chemins(n,i,j,M)` qui retourne le nombre de chemins de longueur `n` allant du sommet s_i au sommet s_j du graphe de matrice d'adjacence `M` d'une graphe orienté.
Rappel, la fonction `np.dot( )` réalise le produit matriciel avec des matrices de type `array`.
2. Considérer le graphe suivant :



Montrer qu'il y a 1225 chemins de longueur 50 entre s_0 et s_2 :
(i) avec la fonction `nb_chemins`
(ii) par dénombrement.

Remarque – La matrice d'adjacence peut être adaptée en une **matrice de poids** contenant le poids des arcs dans le cas d'un graphe pondéré.
On peut prendre différentes conventions en fonction de la représentation des poids :
- 0 pour les boucles ;
- `None` ou `inf` lors qu'il n'y a pas d'arc.

Exercice 8 On considère le graphe suivant, où le nombre situé sur l'arête joignant deux sommets est leur distance, supposée entière :



1. Définir la matrice $M\in\mathcal{M}_5(\mathbb{R})$, *matrice de distances* du graphe `G`, définie par :
- m_{ij} représente la distance entre les sommets i et j ;
- la distance vaut -1 si les sommets ne sont pas reliés ;
- la distance entre un sommet et lui-même vaut 0.

2. Écrire une suite d'instructions permettant de dresser à partir de la matrice `M` la liste des voisins du sommet 4.
3. Écrire une fonction `voisins`, d'argument un sommet i, renvoyant la liste des voisins du sommet i.
4. Écrire une fonction `degre`, d'argument un sommet i, renvoyant le nombre des voisins du sommet i, c'est-à-dire le nombre d'arêtes issues de i.
5. Écrire une fonction `longueur`, d'argument une liste `L` de sommets de `G`, renvoyant la longueur du trajet décrit par cette liste `L`, c'est-à-dire le poids de cette chaîne.
Si le trajet n'est pas possible, la fonction renvoit -1.

II.2. Listes d'adjacence

Définition – Listes d'adjacence
Soit $G = (S, A)$ un graphe. La liste d'adjacence d'un sommet est la liste de ses voisins orientés (sortants).

Remarques –

1. Les listes d'adjacentes peuvent être rassemblées dans une liste ou dans un dictionnaire.
2. Les listes d'adjacences sont adaptées aux graphes creux car la représentation est compacte lorsqu'il y a peu d'arcs, tandis que les matrices d'adjacentes sont adaptées aux graphes denses.
3. On peut adaptée les listes d'adjacences afin d'inclure le poids des arcs.

Exercice 9 Considérant la liste des listes d'adjacentes, `L`, d'un graphe, donner les deux fonctions suivantes :
(i) `matrice(L)` retourne la matrice d'adjacence associée,
(ii) `degre(s,L)` retourne le couple du degré entrant et du degré sortant du sommet `s` du graphe associé à `L`.

II.3. Structure de dictionnaire

Le type `dict` permet de définir un dictionnaire : des relations à sens unique entre des clés et des valeurs :

- les valeurs et les clés peuvent être de type différent ;
- il n'y a pas d'ordre entre les relations ;
- une relation est identifiée par sa clé.

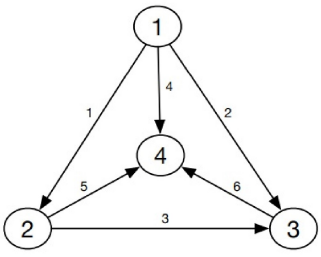
Notation	Description
`D={}`	Définition d'un dictionnaire `D` vide
`D[c]=v`	Ajout de la relation `c:v` de clé `c` et de valeur `v`
`D[k]`	Retourne la valeur associée à la clé `k`
`del D[k]`	Efface la relation de clé `k`
`len(D)`	Taille : nombre de relations
`k in D`	Teste la présence de la clé `k`
`D.keys()`	Séquence des clés
`D.values()`	Séquences des valeurs
`D.items()`	Séquence des relations (type `tuple`)
`D.copy()`	Copie de `D`

```
# definition d'un dictionnaire
>>> dico={'un':2,3:[4,5],'six':'sept'}
# ajout ou modification d'une relation
>>> dico[8]='neuf'
>>> dico
{'six': 'sept', 8: 'neuf', 'un': 2, 3: [4, 5]}
# Taille d'un dictionnaire : nombre de relations
>>> len(dico)
4
# liste de cles
>>> C=list(dico.keys());C
['six', 8, 'un', 3]
# les valeurs
>>> dico.values()
dict_values(['sept', 'neuf', 2, [4, 5]])
```

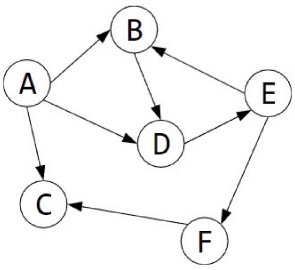
```
#acces a une valeur
>>> dico['six']
'sept'
>>> dico[2] # 2 est une valeur et non une cle !!!
KeyError: 2
# test d'appartenance d'une cle
>>> 'un' in dico
True
# suppression d'une relation
>>> del dico[4]
>>> dico
{'six': 'sept', 8: 'neuf', 'un': 2}
# liste des couples des relations
>>> list(dico.items())
[('six', 'sept'),(8,'neuf'),('un', 2)]
```

Exercice 10 Donner une fonction `comptage(L)` avec `L` une liste, qui retourne un dictionnaire permettant de dénombrer le nombre d'occurrences de chaque élément de `L`.

Exercice 11 Donner la matrice d'adjacence, la liste des listes d'adjacences et le dictionnaire des listes d'adjacence des graphes suivants :



Graphe G_7



Graphe G_8

Quelle représentation vous semble la mieux indiquée ?

III. Parcours d'un graphe

L'exploration d'un graphe est un enjeu majeur, par exemple :

- pour rechercher les sommets atteignable à partir d'un sommet donnée, autrement dit identifier une composante connexe ;
- pour recherche le plus court chemin (ou celui de plus petit poids) en deux sommets, autrement dit résoudre un problème de minimisation.

Le travail peut être penser récursivement : on considère les voisins du sommet initial, puis les voisins des voisins et ainsi de suite. On peut différencier deux schémas :

- ▶ approche DFS (Deep First Search), ou approche en profondeur : on privilégier le premier voisin, le premier voisin d'un premier voisin, etc.
- ▶ approche BFS (Breadth First Search), ou approche en largeur : on privilégier les premiers voisins avant d'explorer les voisins des voisins.

III.1. Notion de Pile et de File

Le stockage de données est une problématique incontournable en informatique. Plusieurs solutions existent :

- La liste (ou le tableau) permet d'accéder directement à n'importe quel élément en connaissant sa position. Sa

taille est fixée à l'avance : il convient d'anticiper le nombre de données, en particulier le majorer.
En PYTHON, une liste est redimensionnable.

• La **pile** de capacité finie (ou non) répond à une autre logique, à l'image d'une pile d'assiettes, on peut ajouter une assiette au sommet de la pile ou en retirer une : c'est un modèle **LIFO** (en anglais *last in, first out*), le dernier élément introduit est le premier à sortir.
Dans une *pile*, on ne peut accéder qu'au dernier élément et non à n'importe quel.

• La **file** est du même type qu'une *pile*, mais c'est un modèle **FIFO** (en anglais *first in, first out*), le premier élément introduit est le premier à sortir.

➜ Les trois fonctions de base d'une pile sont :

Fonction	Description
`creer_pile()`	Renvoie un pile p vide
`depiler(p)`	dépile et renvoie le sommet de p : **pop**
`empiler(p,v)`	empile la valeur v sur la pile p : **push**

On peut aussi introduire les fonctions :

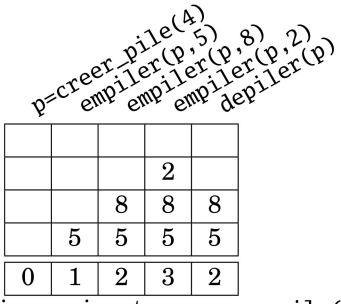
Fonction	Description
`est_vide(p)`	Teste si la pile p est vide
`taille(p)`	Renvoie le nombre d'éléments de p
`sommet(p)`	renvoie le sommet p sans la modifier

➜ Les fonction pour une file sont : `creer_file()`, `enfiler(p,v)`, `defiler(p)`, etc.

Exercice 12 *Modèle de Piles à capacité finie c*
On considère une liste p de la longueur fixe : `c+1`.

• La hauteur du sommet est renseignée dans la première case : `p[0]`.

• Les éléments sont rangés dans l'ordre où ils ont été empilés.



			2	
		8	8	8
	5	5	5	5
0	1	2	3	2

Définir les fonctions suivantes : `creer_pile(c)` (de capcaité `c`), `depiler(p)`, `empiler(p,v)`, `taille(p)`, `est_vide(p)`, `sommet(p)`

Exercice 13 *Modèle de Piles non bornée*
On considère une liste initialement vide.

• Les éléments sont rangés dans l'ordre où ils ont été empilés.

• La longueur de la liste correspond à la hauteur de la pile.

Définir les mêmes fonctions que pour les piles bornées.
Utiliser `p.append(v)` et `p.pop()`.

Remarques –

1. De façon générale, un objet *pile* ou *file* peut être représenté en PYTHON par une liste, mais il existe un type plus adapté et donc plus efficace dans le module dédié `collections.deque`. Nous continuerons à utiliser les listes ... mais sans exploiter la possibilité d'accéder à n'importe quel élément à partir de sa position.
2. L'objet *pile* est directement associé à l'empilement des appels dans une fonction récursive.

III.2. DÉTECTION DE CYCLE

Objectif : Détecter si un graphe possède un cycle

Exercice 14 Proposer une solution pour un graphe orienté en considérant l'exercice sur le nombre de chemins de longueur fixée entre deux sommets donnés.

➜ Voici un algorithme de détection de cycle à partir d'un sommet s donné, dans un graphe orienté. Il utilise la notion de *file*. Les outils sont :

• une liste `sommets` qui renseigne si le sommets numéroté i est atteignable à partir de s : `sommets[i]` prend les valeurs `True` ou `False` ;

• un file `file` des sommets (voisins) intermédiaires restant à parcourir.

Entrée : $s \in S$ un sommet et
M la matrice d'adjacence du graphe $G(S,A)$

```
n le nombre de sommets
file une file initialisée avec s
sommets une liste de False de longueur n
Tant que file n'est pas vide faire
    p le sommet défilé de file
    Pour i parcourant les sommets de G faire
        Si i est un voisin sortant de p alors
            Si i = s alors
                Retourner True
            SinonSi i est non encore atteint alors
                enfiler i à file
                mettre à jour sommets
            FinSi
        FinSi
    FinPour
FinTantQue
Retourner False
```

Sortie : `True/False`

Exercice 15
1. Donner le script de cet algorithme.
2. Que reste-t-il à faire pour répondre à l'objectif ?
3. Est-ce un parcours en largeur ou en profondeur ?
4. Quel effet à le changement de la *file* en *pile* ?
5. Proposer une variante qui permettent de retourner un cycle de G, s'il en existe un.

III.3. DÉTECTION DE CONNEXITÉ

Objectif : Détecter si un graphe non orienté est connexe

Exercice 16 En vous inspirant de l'algorithme précédent, proposer une fonction qui détermine si un graphe non orienté est connexe.