

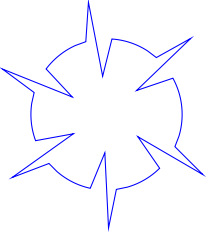
CHAPITRE	6	GÉNÉRALITÉS SUR LES GRAPHES	

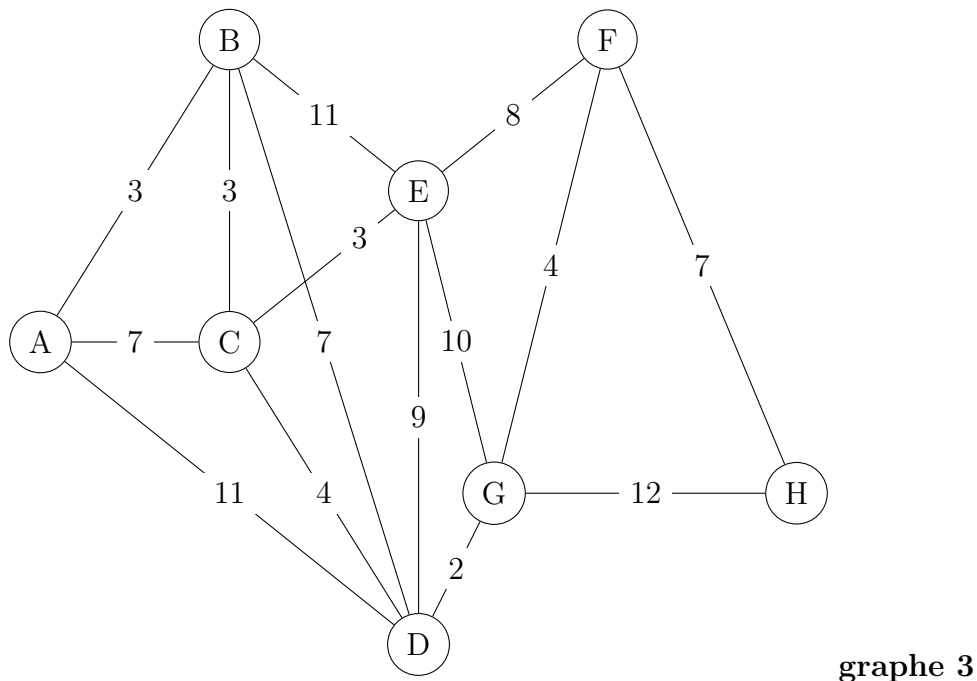
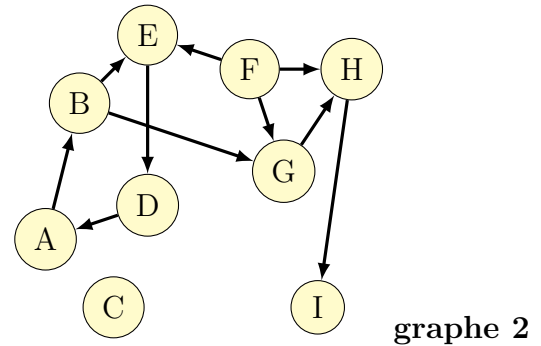
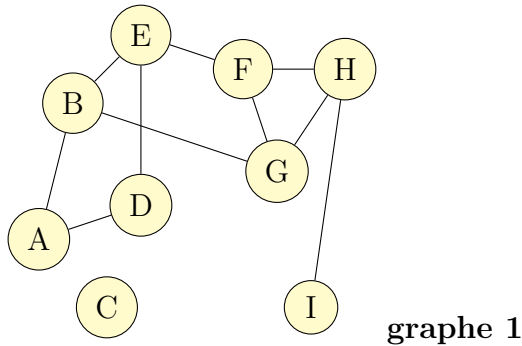
Table des matières

1	Définition	2
2	Interface et Implémentation	5
2.1	Interface	5
2.2	Quelles structures utilisées?	5
2.2.1	Matrices d'adjacence	5
2.2.2	Utilisation d'un dictionnaire	8
3	Exercices complémentaires	9
4	Représenter les graphes en Python	10

1 Définition



Un **graphe** est un ensemble d'éléments appelés **sommets**, reliés par des **arêtes** qui symbolisent une "relation" entre les deux entités représentées par les deux sommets.



Un graphe peut être *orienté* ou *non orienté*, *pondéré* ou non. Quelques exemples traditionnels de modélisation par des graphes :

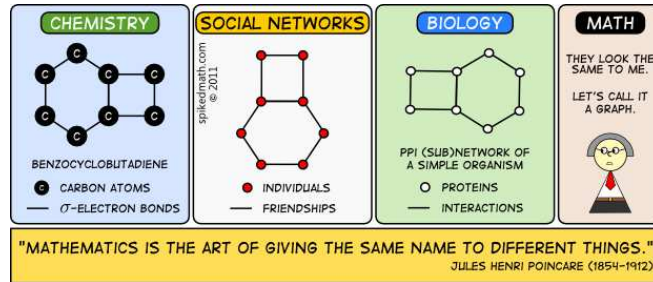
- ⇒ Un réseau routier peut être représenté par un graphe, où les routes qui relient chaque zone est un arc.
Si on s'intéresse au réseau routier d'une ville le graphe sera orienté pour représenter les rues à sens unique.
Le GPS utilise de tels graphes en rajoutant les temps de parcours, péages ... pour optimiser un chemin à prendre.
- ⇒ Un réseau informatique peut être représenté par un graphe, le protocole OSPF optimise lui aussi pour chaque routeur le chemin que doit prendre un paquet de données pour atteindre un point dans le réseau.
- ⇒ Un labyrinthe constitué de salles reliées entre elles peut être vu comme un graphe non orienté.

Les sommets sont des salles et sont reliés par un arc qui représente une porte permettant de relier ces deux salles.

⇒ Des jeux tel que le solitaire, jeu d'échecs, de dames peuvent être représentés par des graphes.

Les sommets sont l'emplacement des pièces et les arcs des "coups" à jouer modifiant ainsi la configuration.

⇒ Un arbre peut être vu lui aussi comme un graphe où chaque noeud constitue un sommet et les arcs sont les chemins menant aux racines des sous arbres gauche et droit.



L'ordre du graphe est le nombre de sommets présents dans le graphe.

Exercice n°1

Donner l'ordre des graphes précédents



Le degré d'un sommet est le nombre d'arêtes dont le sommet est une extrémité.

Exercice n°2

Donner le degré de chaque sommet du graphe 1

Sommet	A	B	C	D	E	F	G	H	I
Degré									



Un voisin d'un sommet A est un sommet qui lui est relié par une arête.

Si B est un voisin de A on dit alors que A est adjacent à B (et réciproquement). Un chemin du sommet A vers le sommet B est une séquence finie de sommets voisins qui relie le sommet

A à B .

On le notera sous la forme $A \rightarrow A_1 \rightarrow A_2 \rightarrow \dots \rightarrow B$



Un **cycle** est un chemin qui relie un sommet à lui même **sans passer deux fois par le même arc**.

Par exemple $A \rightarrow B \rightarrow E \rightarrow D \rightarrow A$ est un cycle, alors que $A \rightarrow B \rightarrow A$ n'en est pas un !.



La **longueur** d'un chemin est le nombre d'arêtes qui sépare les deux extrémités de ce chemin.

Par exemple la longueur du chemin $A \rightarrow B \rightarrow C \rightarrow D$ est 3.

On pourra aussi parler de la **distance** entre les sommets.



Une arête AB est **orientée** d'un sommet A vers un sommet B .

Si toutes les arêtes d'un graphe de type AB sont complétées par une arête de type BA alors le graphe est dit **non orienté**.

S'il existe au moins une arête non complétée de la sorte, le graphe sera dit **orienté**



Un graphe est dit **complet** si chaque sommet est relié à tous les autres sommets du graphe.

Exercice n°3

Dessiner les graphes complets d'ordre 3 et 4



Un graphe non orienté est dit **connexe** si pour toute paire de sommet $\{A; B\}$ il existe au moins **un chemin** reliant les deux sommets.

Nous concernant, tous les graphes seront connexes !

2 Interface et Implémentation

2.1 Interface

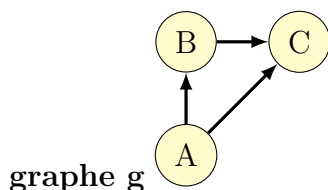
Les primitives permettent d'écrire les algorithmes indépendamment de l'implémentation.

Un graphe est un ensemble de sommets et de relation entre eux. Il faut donc au minimum les primitives suivantes :

<code>creerGrapheVide</code>	Crée un graphe vide
<code>estVide</code>	Renvoie True si le graphe est vide, False sinon
<code>ajouterSommet</code>	Ajoute le sommet valeur au graphe
<code>supprimerSommet</code>	Supprime le sommet valeur du graphe
<code>ajouterArete</code>	Ajoute une arête du graphe, du sommet A au sommet B
<code>supprimerArete</code>	Supprime l'arête du graphe, de A à B

À ces primitives, on ajoutera, selon les besoins, des fonctions de lecture des propriétés du graphe (ordre), de recherche, de calcul de distance, de parcours ...

Par exemple, quelles instructions sont nécessaires à la construction du graphe **g** ci-dessous ?



```
g = creerGrapheVide()
ajouterSommet(g,A)
ajouterSommet(g,B)
ajouterSommet(g,C)
ajouterArete(g,A,B)
ajouterArete(g,B,C)
ajouterArete(g,A,C)
```

2.2 Quelles structures utilisées ?

Comment peut-on représenter un graphe en machine, plus précisément en Python ? Nous allons développer cette question en présentant les deux habituelles implémentations choisies.

2.2.1 Matrices d'adjacence

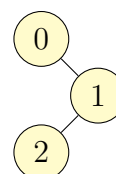
On dispose d'un graphe d'ordre N , on décide alors d'associer des entiers de 0 à $N - 1$ pour nommer les sommets.

Pour représenter les relations entre sommets, on construit une matrice M (tableau) de taille $N \times N$ contenant des booléens (par exemple) en respectant la règle suivante :

- ⇒ Si $M[i][j] = 1$ alors on a l'arête $i \rightarrow j$
- ⇒ Si $M[i][j] = 0$ alors l'arête $i \rightarrow j$ n'existe pas.

On considère le graphe **g** d'ordre 3 avec les relations suivantes avec à gauche le tableau permettant de construire la matrice d'adjacence .

graphe g



On obtient alors le tableau suivant :

	0	1	2
0	0	1	0
1	1	0	1
2	0	1	0

La matrice adjacente au graphe \mathbf{g} est alors $\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$

Exercice n°4

Construire un graphe dont la matrice d'adjacence est donnée ci-dessous

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

La matrice adjacente du graphe définit donc parfaitement celui-ci, et on va dans un premier temps choisir une *double liste* pour **implémenter** la structure de graphes. Pour le graphe précédent, on obtiendrait donc :

1. Création du tableau par défaut (sans liens) `M = [[0]*3 for _ in range(3)]`
2. Création des chemins : `M[0][1] = 1, M[1][0] = 1, M[1][2] = 1` et `M[2][1] = 1`

Exercice n°5

- ❶ Implémenter le graphe `G` d'ordre 5 dont la matrice M est donnée précédemment.
- ❷ Implémenter le graphe non orienté `alea` d'ordre 10 composés d'arêtes choisies au hasard.
- ❸ Implémenter le graphe non orienté `complet_6` qui est d'ordre 6 et complet !

On pourra ainsi utiliser ces graphes pour tester les programmes suivants...

Exercice n°6

En choisissant l'implémentation d'une matrice d'adjacence comme dans l'exemple précédent, programmer en Python les fonctions suivantes :

1. `ordre(mat)` qui prend comme argument une matrice d'adjacence et renvoie l'ordre du graphe associé.
2. `degre(mat, s)` qui prend comme argument une matrice d'adjacence et un entier `s` qui représente un sommet du graphe et renvoie le degré de ce sommet.
3. `existeArete(mat, (a,b))` qui prend comme argument une matrice d'adjacence et un tuple de deux entiers représentant chacun un sommet du graphe, et qui renvoie `True` s'il existe une arête sur le graphe menant du sommet `a` au sommet `b`, `False` sinon.
4. `supprimeArete(mat, (a,b))` et `supprimeSommet(mat, s)`.

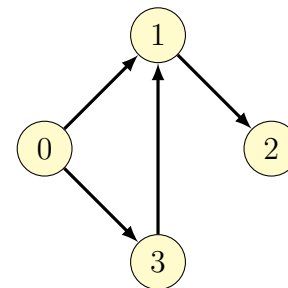
Exercice n°7

On souhaite encapsuler les opérations sur les graphes orientés dans une classe `Graphe`.

Compléter alors les méthodes de cette classe.

```
1 class Graphe():
2     """un graphe représenté par une
3     ↪ matrice
4     ↪ d'adjacence, où les sommets sont
5     ↪ représentés
6     ↪ par les entiers 0,1,...,n-1"""
7
8     def __init__(self, n):
9         self.ordre = n
10        self.adj = [[False]*n for _
11                    ↪ in range(n)]
12
13        def ajouter_arete(self, s1, s2):
14            """crée l'arete s1s2"""
```

A l'aide de cette classe, créer l'instance `g` qui représente le graphe suivant :
graphe g



2.2.2 Utilisation d'un dictionnaire

Sur de "petits" graphes la matrice est pertinente, mais sur de plus grands graphes, la taille de la matrice est n^2 avec des booléens (0 ou 1 sont des booléens...) qui prennent de la place en mémoire et cela même s'il y a peu de chemins.

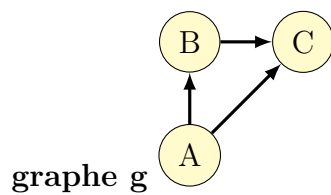
On change ici de modèle de représentation avec l'utilisation d'un *dictionnaire*.

De plus il n'est plus ici obligé de passer par des entiers pour représenter les sommets puisque le type de clés pour un dictionnaire est quelconque.



Dans cette nouvelle représentation, un graphe est un *dictionnaire*, qui associe à chaque sommet l'ensemble de ses voisins.

Par exemple on pourra représenter le graphe suivant



sous la forme `g = {'A': ['B', 'C'], 'B': ['C'], 'C': []}`

Exercice n°8

Avec le choix de cette implémentation en dictionnaire,

- ❶ créer les primitives `creer_graphe(sommets)`, `ajouter_arete(graphe, s1, s2)`, `sommets(graphe)` et `voisins(graphe, sommet)`.
- ❷ Enrichir cette interface avec les méthodes `ordre`, `degre` et `existeArete` rencontrées ci-dessus.

On décide maintenant de reprogrammer la classe Graphe de l'exercice précédent.

```
1 class Graphe():
2     """un graphe comme dictionnaire de
   ↪ voisins"""
3     def __init__(self):
4         self.adj = {}
```

Exercice n°9

Programmer les méthodes

1. `ajouterSommet(self, s)` qui ajoute au graphe un nouveau sommet `s`.
2. `ajouterArete(self, s1, s2)` qui ajoute si nécessaire le(s) sommet(s) manquant et crée le lien $s1 \rightarrow s2$
3. `arete(self, s1, s2)` qui renvoie `True` si le lien $s1 \rightarrow s2$ existe et `False` sinon.
4. `sommets(self)` qui renvoie la liste des sommets du graphe.
5. `voisins(self, s)` qui renvoie la liste (sous forme d'ensemble) des voisins du sommet `s`.

3 Exercices complémentaires

Exercice n°10

Écrire deux fonctions Python permettant respectivement de passer d'une liste de voisins à une matrice d'adjacence et réciproquement

Exercice n°11

Les théorèmes d'Euler permettent d'identifier la présence de *chaîne eulérienne* et de *cycle eulérien* dans des graphes connexes.

Coder les méthodes ou fonctions Python qui retourne `True` si un graphe quelconque `G` contient une chaîne eulérienne, un cycle eulérien et `False` sinon.

4 Représenter les graphes en Python

Les bibliothèques `networks` et `matplotlib` permettent la construction du graphe.



La fonction `creer_graphe_non_orienté` crée un graphe non orienté utilisable par la bibliothèque `networks` pour le représenter.

Et voici le code qui permet de dessiner le graphe `mon_graphe` implémenté sous la forme d'un dictionnaire :

```
1 import matplotlib.pyplot as plt
2 import networkx as nx
3 def cree_graphe_non_orienté_nx(dictionnaire: dict) -> nx.Graph:
4     """
5     Cette fonction permet de transformer une représentation en dictionnaire
6     ↪ en
7     une représentation «complexe» d'un objet graphe orienté.
8
9     - Précondition : l'entrée est un dictionnaire
10    - Postcondition : la sortie est un graphe orienté (Graph) de Networkx
11    """
12    Gnx = nx.Graph()
13    for sommets in dictionnaire.keys():
14        Gnx.add_node(sommets) # Creation des sommets
15    for sommet in dictionnaire.keys():
16        for sommets_adjacents in dictionnaire[sommet]:
17            Gnx.add_edge(sommet, sommets_adjacents) # Creation des arcs
18    return Gnx
19
20 plt.cla() # Pour effacer les figures précédentes
21 mon_graphe = {0: [1, 2], 1: [0, 2, 3], 2: [0, 1, 3], 3: [1,2]}
22 G = cree_graphe_non_orienté_nx(mon_graphe)
23 nx.draw(G, with_labels = True) # Pour une représentation classique
24 plt.show()
```