TP - Représentations d'un graphe

Objectifs :

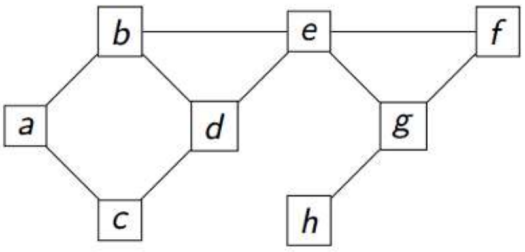
- ▷ Comment représenter un graphe ?
- ▷ Passer d'une représentation d'un graphe à une autre.
- ▷ Choisir la représentation d'un graphe adaptée à la situation

1 Avec un dictionnaire

Un graphe est caractérisé par sa matrice d'adjacence composée de 1 et de 0 selon que deux sommets sont ou ne sont pas reliés par une arête.

Une façon d'encoder un graphe sous Python est d'utiliser un **dictionnaire** qui sera la représentation de sa matrice d'adjacence.

Les clés seront les sommets du graphe et leur valeur sera la liste des sommets adjacents. Prenons par exemple ce graphe :



	a	b	c	d	e	f	g	h
a	0	1	1	0	0	0	0	0
b	1	0	0	1	1	0	0	0
c	1	0	0	1	0	0	0	0
d	0	1	1	0	1	0	0	0
e	0	1	0	1	0	1	1	0
f	0	0	0	0	1	0	1	0
g	0	0	0	0	1	1	0	1
h	0	0	0	0	0	0	1	0

Ce qui donne :

```
G = dict()
G['a'] = ['b','c']
G['b'] = ['a','d','e']
G['c'] = ['a','d']
G['d'] = ['b','c','e']
G['e'] = ['b','d','f','g']
G['f'] = ['e','g']
G['g'] = ['e','f','h']
G['h'] = ['g']
```

Pour la matrice d'adjacence, on peut l'écrire à la main, en utilisant une liste de liste :

```
A1 = [[0,1,1,0,0,0,0,0],
      [1,0,0,1,1,0,0,0],
      [1,0,0,1,0,0,0,0],
      [0,1,1,0,1,0,0,0],
      [0,1,0,1,0,1,1,0],
      [0,0,0,0,1,0,1,0],
      [0,0,0,0,1,1,0,1],
      [0,0,0,0,0,0,1,0]]
```

Ou bien, on peut utiliser le code ci-après pour fabriquer cette matrice d'adjacence à partir de la liste des sommets et du dictionnaire G :

```
liste = ['a','b','c','d','e','f','g','h']
n = len(liste)
A = [[0]*n for i in range(n)]

for i in range(n):
    for j in range(n)
        if liste[j] in G[liste[i]]:
            A[i][j] = 1
```

1.1 Quelques fonctions pour exploiter le graphe

Voici quelques rappels sur les dictionnaires :

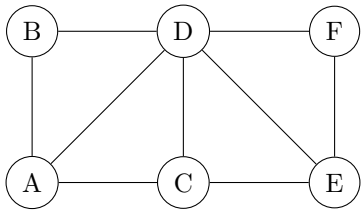
```
print(G.keys())                  #affiche les cles
print(G.values())                #affiche les valeurs
print(len(G))                    #affiche le nombre de cles
print(G['e'])                    #affiche la value de la cle 'e'

# G.keys() et G.values() sont iterables
for el in G.values()
    print(el)                    #affiche les valeurs du dictionnaire

for el in G.keys()
    print(key,G[key])            #affiche les cles et les valeurs des cles
```

Exercice 1 : Tester tous le programmes donnés en exemple ci-dessus. Implémenter le graphe G et tester ces différents affichages.

Exercice 2 : Écrire des fonctions permettant d'obtenir les informations suivantes sur le graphe G :

- le nombre de sommets du graphe
- le nombre d'arêtes du graphe
- le degré d'un sommet
- le sommet de plus haut degré
- les voisins d'un sommet

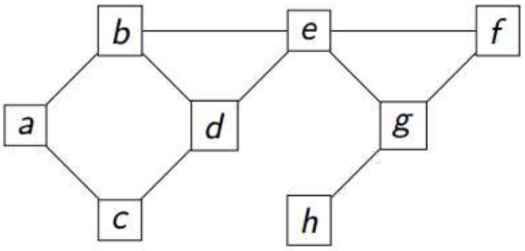
Exercice 3 : Implémenter le graphe du réseau social du cours et faire afficher celui qui a le plus d'amis.



2 Avec une bibliothèque

La bibliothèque `networkX` permet de manipuler les graphes.

Prenons le même exemple que précédemment :



	a	b	c	d	e	f	g	h
a	0	1	1	0	0	0	0	0
b	1	0	0	1	1	0	0	0
c	1	0	0	1	0	0	0	0
d	0	1	1	0	1	0	0	0
e	0	1	0	1	0	1	1	0
f	0	0	0	0	1	0	1	0
g	0	0	0	0	1	1	0	1
h	0	0	0	0	0	0	1	0

On importe le module :

```
import networkx as nx
```

On crée un graphe vide :

```
#creation du graphe
g1 = nx.Graph()
```

On ajoute les sommets appelés *node* :

```
#creation des sommets
g1.add_node('a')
g1.add_node('b')
g1.add_node('c')
g1.add_node('d')
g1.add_node('e')
g1.add_node('f')
g1.add_node('g')
g1.add_node('h')
```

On ajoute les arêtes appelés *edge* :

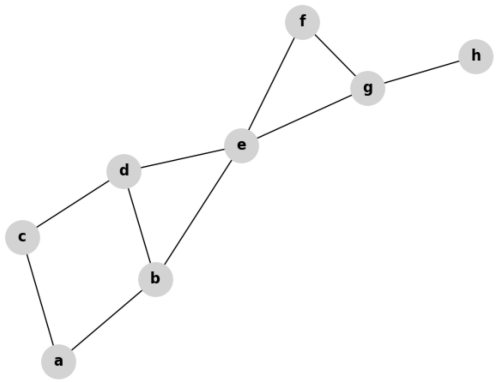
```
#creation des aretes
g1.add_edge('a','b')
g1.add_edge('a','c')
g1.add_edge('b','d')
g1.add_edge('b','e')
g1.add_edge('c','d')
g1.add_edge('d','e')
g1.add_edge('e','g')
g1.add_edge('e','f')
g1.add_edge('g','f')
g1.add_edge('g','h')
```

On peut visualiser le graphe grâce à la librairie `maplotlib` et la méthode `draw`.

Ici, on a configuré l'affichage pour que :

- les étiquettes des sommets soient bien affichées,
- la taille des sommets soit de 800,
- la couleur de fond des sommets soit gris clair.

```
import matplotlib.pyplot as plt
nx.draw(g1,with_labels=True,font_weight='bold',node_size=800,
    node_color='lightgrey')
plt.show()
```



On peut également le faire avec des listes de sommets et d'arêtes :

```
import networkx as nx
import matplotlib.pyplot as plt

#creation du graphe a partir de listes
liste1 = ['a','b','c','d','e','f','g','h']
g2 = nx.Graph()
g2.add_nodes_from(liste1)

liste2=[('a','b'),('a','c'),('b','d'),('b','e'),('c','d'),('d','e'),('e','g')
        ,('e','f'),('g','f'),('g','h')]
g2.add_edges_from(liste2)

nx.draw(g2,with_labels=True,font_weight='bold',node_size=800,
    node_color='lightgrey')
plt.show()
```

Pour la matrice d'adjacence, `networkx` propose une méthode `nx.adjacency_matrix(g2)` qui stocke les coefficients a_{ij} de la matrice d'adajcence.

Il suffit alors de remplir un tableau avec ces coefficients.

```
B = nx.adjacency_matrix(g2)
print(B[0,0])

n = len(liste1)
A = [[0]*n for i in range(n)]

for i in range(n):
    for j in range(n):
        A[i][j] = B[i][j]

print(A)
```

2.1 Exploiter la bibliohèque

La documention de `networkX` est divisée en section. Il existe notamment :

— une section pour obtenir les méthodes sur les sommets et les arêtes
— une section pour obtenir les algorithmes disponibles

On pourra aussi consulter le tutoriel de `networkX`.

Voici quelques unes des méthodes disponibles :

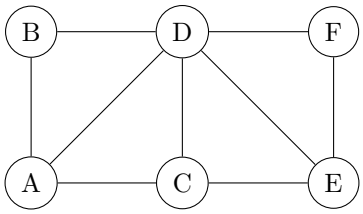
— degrés d'un sommet du graphe g : `g.degree('a')`
— nombre de sommets du graphe g : `g.number_of_nodes()`
— nombre d'arêtes du graphe g : `g.number_of_edges()`
— liste des précédesseurs du sommet i : `g.predecessors(i)`
— liste des successeurs du sommet i : `g.successors(i)`
— liste des voisins du sommet i : `g.neighbors(i)`

Exercice 4 : Tester tous le programmes donnés en exemple ci-dessus.

Avec `networkX`, chercher les méthodes pour obtenir les informations suivantes sur le graphe G :

— le nombre de sommet du graphe
— le nombre d'arêtes du graphe
— le degré d'un sommet
— le sommet de plus haut degré
— les voisins d'un sommet

Exercice 5 : Implémenter le graphe du réseau social du cours et faire afficher celui qui a le plus d'amis.



Exercice 6 : *Secret Santa*

Un groupe de n amis décide de se faire des cadeaux pour Noël.

Il vont adopter la stratégie *Secret Santa* : chacun va faire un seul cadeau et recevoir un seul cadeau. Evidemment, on ne doit pas se faire un cadeau à soi-même.

La situation peut être modélisée par un graphe orienté :

— Les noeuds sont les prénoms des amis.
— Le fait qu'une personne A donne un cadeau à une personne B sera représenté par un arc partant de A et allant à B.
— On dit qu'un graphe est valide si à chaque noeud arrive un arc et un seul, de chaque noeud part un arc et un seul, il n'y a pas de boucle sur un noeud.

1. Représenter les graphes dont on donne les matrices d'adjacences ci-dessous, et dire lesquels sont valides.

   Graphe 1 pour les noeuds `A`, `B`, `C`, `D`, `E` et `F` :

```
matrice_1 = [
[0, 1, 0, 0, 0, 0],
[0, 0, 1, 0, 0, 0],
[1, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 1, 0],
[0, 0, 0, 0, 0, 1],
[0, 0, 0, 1, 0, 0]
]
```

Graphe 2 pour les noeuds `A`, `B`, `C`, `D` :

```
matrice_2 = [
[0, 1, 0, 0],
[0, 0, 1, 0],
[0, 1, 0, 0],
[1, 0, 0, 0]
]
```

2. Ecrire une fonction `valide` qui prend en paramètre une liste de listes `matrice` et renvoie `True` si elle est valide, et `False` sinon.

   Pour être sûr, on procède ainsi : on crée une liste d'amis, par exemple `['A', 'B', 'C', 'D', 'E', 'F']`. Chaque ami donne au suivant, et reçoit du précédant dans la liste, sauf le premier qui reçoit du dernier, et le dernier qui donne au premier.
3. Ecrire la fonction `cree_matrice` qui prend en palamètre la liste d'amis `amis`, et renvoie la matrice d'adjacence du graphe construit en suivant ce principe.
4. Ecrire la fonction `cree_dico` qui prend en palamètre la liste d'amis `amis` qui a été mélangée, et renvoie la liste d'adjacence du graphe construit en suivant le principe de la question 3.