Chap 09. Graphes

Livre p 189 – Chap 11 Graphes
Livre p 207 – Chap 12 Parcours en profondeur et en largeur

1. Une structure abstraite : les graphes

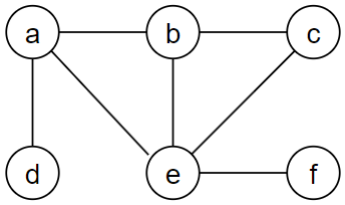
a. Définitions et vocabulaire

- Un graphe est un ensemble de **sommets** (ou nœuds) reliés entre eux par des liens (**arcs** ou **arêtes**).
  Remarque : Un arbre binaire est un cas particulier de graphe.
- Deux sommets reliés par un lien sont dits **adjacents** ou **voisins**.
- On parle de **graphe non orienté** lorsque les liens n'ont pas d'orientation (on parle alors d'arêtes), ils sont alors représentés par de simples traits entre deux sommets.
- On parle de **graphe orienté** lorsque les liens ont une orientation (on parle alors d'arcs), ils sont alors représentés par des flèches entre deux sommets.
- On parle de **graphe pondéré** lorsque les liens sont associés à des valeurs numériques **(poids)**.
- Une suite de sommets consécutifs reliés entre eux est appelée **chemin** (si le graphe est orienté) ou **chaîne** (si le graphe n'est pas orienté).
- On parle de **graphe connexe** s'il existe une chaîne passant par tous les sommets.
  Remarque : Pour un graphe orienté on parle de graphe fortement connexe s'il existe un chemin passant par tous les sommets, et graphe connexe s'il n'y a qu'une chaîne (sans les orientations)
- Un chemin (d'au moins deux sommets) ou une chaîne (d'au moins 3 sommets) partant d'un sommet et arrivant à ce même sommet est appelé un **cycle**.

b. Mesures d'un graphe

- L'**ordre** d'un graphe est le nombre de ses sommets.
- La **longueur** d'une chaîne (ou d'un chemin) est le nombre de liens qui la compose.
- La **distance** entre 2 sommets est la longueur de la chaîne (ou du chemin) le plus court qui les relie.
  Remarque : On verra dans le chapitre suivant l'algorithme de Dijkstra…

c. Parcours d'un graphe connexe



Il existe deux façons de parcourir un graphe à partir d'un sommet :

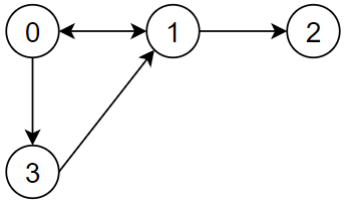
- Le parcours **en profondeur** (D.F.S. pour Depth First Search) :
  On initialise une pile avec le sommet de départ.
  Si la pile n'est pas vide, on dépile le premier élément,
  puis on empile tous ses voisins non vus et on recommence...
  Avantage : On va détecter les cycles éventuels au fur et à mesure.
  Exemple : a – b – c – e – f – d

	⇒	b	⇒	c	⇒		⇒		⇒		
		e		e		e		f			
a		d		d		d		d		d	

- Le parcours **en largeur** (B.F.S. pour Breadth First Search) :
  On initialise une file avec le sommet de départ.
  Si la file n'est pas vide, on défile le premier élément,
  puis on enfile tous ses voisins non vus et on recommence…
  Avantage : On va obtenir la distance de chaque sommet au sommet de départ au fur et à mesure.
  Exemple : a – b – e – d – c – f

a	⇒	d	e	b	⇒	c	d	e	⇒	f	c	d	⇒	f	c	⇒	f

2. Implémentation en Python



a. Matrice d'adjacence

Une matrice d'adjacence est un tableau à deux dimensions dans lequel la valeur située en ligne *i* et colonne *j* vaut :

- 1 (ou True) pour indiquer qu'il existe un lien de *i* vers *j*.
- 0 (ou False) pour indiquer qu'il n'y a pas de lien de *i* vers *j*.

Remarque : Si le graphe n'est pas orienté, la matrice d'adjacence est symétrique.
Inconvénients : Les sommets doivent être numérotés de 0 à $n-1$.
Il est difficile d'ajouter un sommet à un graphe existant.
Exemple : M = [[0, 1, 0, 1], [1, 0, 1, 0], [0, 0, 0, 0], [0, 1, 0, 0]] c'est-à-dire :

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

b. Listes d'adjacence

Les listes d'adjacences sont des listes contenant tous les sommets associés à un sommet donné.
On peut utiliser une liste de listes ou un dictionnaire de listes.
Inconvénient : L'utilisation d'une liste de listes nécessite que les sommets soient numérotés de 0 à $n-1$.
Avantages : L'utilisation d'un dictionnaire permet d'utiliser n'importe quelle étiquette pour les sommets.
On peut ajouter facilement un nouveau sommet à un graphe existant.
Exemple : liste = [[1, 3], [0, 2], [], [1]] ou dico = {0:[1, 3], 1:[0, 2], 2:[], 3:[1]}

c. Fonctions élémentaires sur les graphes

- **g.ajouter_arc(s1, s2)** : Permet d'ajouter un arc entre les sommets s1 et s2
  Attention : Pour un graphe non orienté, il faudra ajouter également l'arc entre s2 et s1.
- **g.arc(s1, s2)** : Renvoie True s'il existe un arc entre s1 et s2, False sinon.
- **g.voisins(s)** : Renvoie la liste des voisins du sommet s.

```
class Graphe:
    """un graphe représenté par une matrice d'adjacence,
       où les sommets sont les entiers 0,1,...,n-1"""

    def __init__(self, n):
        self.n = n
        self.adj = [[False] * n for _ in range(n)]

    def ajouter_arc(self, s1, s2):
        self.adj[s1][s2] = True

    def arc(self, s1, s2):
        return self.adj[s1][s2]

    def voisins(self, s):
        v = []
        for i in range(self.n):
            if self.adj[s][i]:
                v.append(i)
        return v
```

```
class Graphe:
    """un graphe comme un dictionnaire d'adjacence"""

    def __init__(self):
        self.adj = {}

    def ajouter_sommet(self, s):
        if s not in self.adj:
            self.adj[s] = set()

    def ajouter_arc(self, s1, s2):
        self.ajouter_sommet(s1)
        self.ajouter_sommet(s2)
        self.adj[s1].add(s2)

    def arc(self, s1, s2):
        return s2 in self.adj[s1]

    def sommets(self):
        return list(self.adj)

    def voisins(self, s):
        return self.adj[s]
```

d. Utilisation de la bibliothèque networkx

Attention : Ce module nécessite également la bibliothèque mathplotlib pour l'affichage, ainsi que la bibliothèque numpy si l'on souhaite utiliser les matrices d'adjacence.
(Tutoriel disponible ici : https://mmorancey.perso.math.cnrs.fr/TutorielPython_NetworkX.html)

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

M = np.array([[0, 1, 0, 1], [1, 0, 1, 0], [0, 0, 0, 0], [0, 1, 0, 0]])
G = nx.DiGraph(M)
nx.draw(G, with_labels=True)
plt.show()
```

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

dico = {0:[1, 3], 1:[0, 2], 2:[], 3:[1]}
G = nx.DiGraph(dico)
nx.draw(G, with_labels=True)
plt.show()
```