Les Graphes

1 Définitions

1.1 Graphe non orienté

Un **graphe non orienté** G est la donnée d'un couple $G = (S, A)$ tel que :

- S est un ensemble fini de **sommets**,
- A est un ensemble de couples non ordonnés de sommets $\{s_i, s_j\} \in S^2$, les **arêtes**.

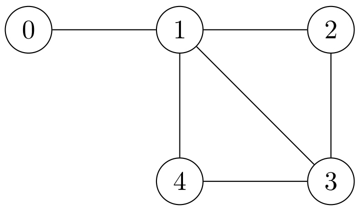


FIGURE 1 – Un graphe à cinq sommets et six arêtes

- Deux sommets sont **adjacents** (ou **voisins**) si ils sont reliés par une arête, dans l'exemple précédent les sommets 2 et 3 sont adjacents.
- Le **degré** d'un sommet est le nombre d'arête partant de ce sommet, dans la figure 1, le sommet 4 a pour degré 2.
- Une **chaîne** entre deux sommets A et B est une suite d'arête menant de A à B. Dans l'exemple précédent, les sommets `0` et `3` sont reliés par la chaîne `0 - 1 - 3`.
- Un **circuit** est une chaîne qui relie un sommet à lui-même sans emprunter deux fois la même arête. La chaîne `1 - 3 - 2 - 1` est un circuit.

1.2 Graphe orienté

Un **graphe orienté** G est la donnée d'un couple $G = (S, A)$ tel que :

- S est un ensemble fini de **sommets**,
- A est un ensemble de couples ordonnés de sommets $(s_i, s_j) \in S^2$, les **arcs**.

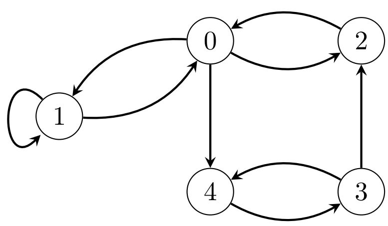


FIGURE 2 – Un graphe à cinq sommets et neufs arcs

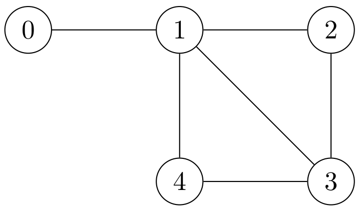
- Le sommet B est **adjacent** (ou **voisin**) du sommet A lorsqu'il existe un arc de A vers B. Dans l'exemple précédent le sommet 4 est un voisin du sommet 0.

- Le **degré** (ou degré sortant) d'un sommet est le nombre d'arc partant de ce sommet, dans la figure 2, le sommet 4 a pour degré 1.
- Un **chemin** entre deux sommets A et B est une suite d'arcs menant de A à B. Dans l'exemple précédent, les sommets `1` et `3` sont reliés par le chemin `1 -> 0 -> 2 -> 3`.
- Un **cycle** est un chemin qui relie un sommet à lui-même sans emprunter deux fois le même arc. La chemin `0 -> 4 -> 3 -> 2 -> 0` est un cycle.

2 Différentes représentations d'un graphe

2.1 Matrice d'adjacence

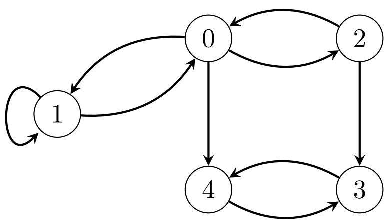
Dans cette première représentation, on suppose que les sommets des graphes sont numérotés par les entiers de 1 à $n-1$ où n est la taille du graphe.
La **matrice d'adjacence** d'un graphe, est la matrice (tableau de nombres) où la valeur de la case se trouvant à la i^{me} ligne et j^{me} colonne vaut 1 si il y a un arc entre les sommets i et j et 0 sinon.



$$
\begin{array}{c} \\ 0 \\ 1 \\ M = 2 \\ 3 \\ 4 \end{array}
\begin{array}{c} \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix} \end{array}
$$

Figure 3 – Matrice d'adjacence d'un graphe non orienté

Remarque : La matrice d'ajacence d'un graphe non-orienté est symétrique.

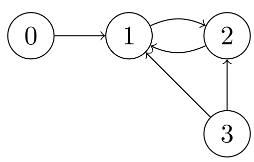


$$
\begin{array}{c} \\ 0 \\ 1 \\ M = 2 \\ 3 \\ 4 \end{array}
\begin{array}{c} \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}
$$

Figure 4 – Matrice d'adjacence d'un graphe orienté

Le programme 1 ci-dessous encapsule une telle matrice d'adjacence dans une classe `Graph` Le constructeur de cette classe prend comme argument le nombre de sommet. Une méthode `ajouter_arc` permet d'ajouter des arcs entre les sommets. Ainsi on peut écrire :

```
g = Graphe(4)
g.ajouter_arc(0, 1)
g.ajouter_arc(1, 2)
g.ajouter_arc(2, 1)
g.ajouter_arc(3, 1)
g.ajouter_arc(3, 1)
```



pour obtenir le graphe de droite.

Programme 1 – Graphe représenté par une matrice d'adjacence

```python
class Graphe:
    '''un graphe represente par une matrice d'adjacence,
    ou les sommets sont les entiers 0, 1, ..., n-1'''
    def __init__(self, n):
        self.n = n
        self.adj = [[0 for j in range(n)] for i in range(n)]

    def ajouter_arc(self, s1, s2):
        self.adj[s1][s2] = 1

    def sommets(self):
        return [s for s in range(self.n)]

    def arc(self, s1, s2):
        return self.adj[s1][s2] == 1

    def voisins(self, s):
        v = []
        for i in range(self.n):
            if self.adj[s][i] == 1:
                v.append(i)
        return v
```

2.2 Liste d'adjacence

Soit G un graphe d'ordre n où les sommets sont numérotés de 0 à $n-1$.
La représentation par **listes d'adjacence** de G consiste en un tableau T de n listes, une pour chaque sommet de G.
Pour chaque sommet i, la liste d'adjacence $T[i]$ est une liste (chaînée) de tous les sommets j tel qu'il existe un arc (resp. arête) du sommet i vers le sommet j.

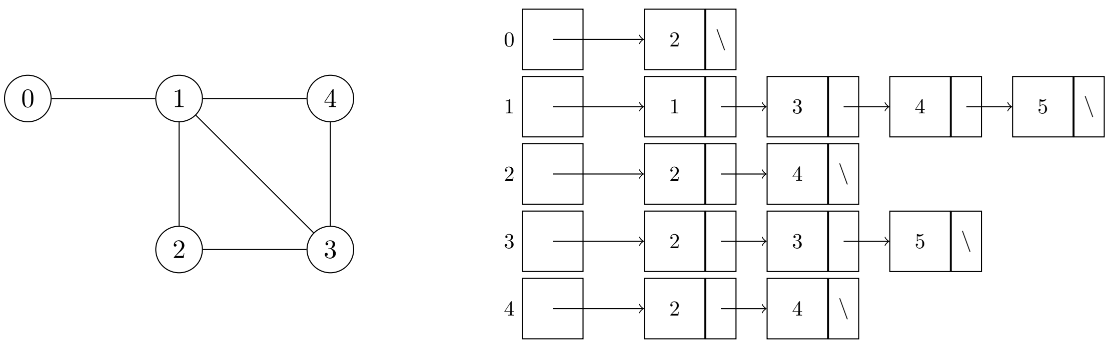


FIGURE 5 – Liste d'adjacence d'un graphe non orienté

Nous allons, pour cette nouvelle représentation, utiliser un *dictionnaire* Python. Chaque sommet sera une clef de ce dictionnaire qui a pour valeur la liste des sommets voisins. Le programme 2 encapsule un tel dictionnaire d'adjacence dans une classe `Graphe` qui permet d'écrire

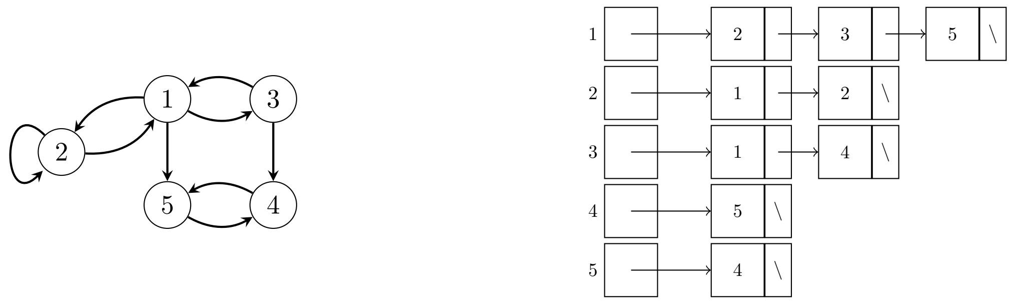
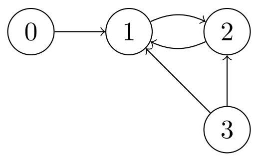


Figure 6 – Liste d'adjacence d'un graphe orienté

```
g = Graphe()
g.ajouter_arc(0, 1)
g.ajouter_arc(1, 2)
g.ajouter_arc(2, 1)
g.ajouter_arc(3, 1)
g.ajouter_arc(3, 1)
```



pour obtenir le graphe de droite.

Programme 2 – Graphe représenté par une liste d'adjacence

```
class Graphe:
    '''un graphe represente par un dictionnaire d'adjacence'''
    def __init__(self):
        self.adj = {}

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

    def ajouter_arc(self, s1, s2):
        self.ajouter_sommet(s1)
        self.ajouter_sommet(s2)
        if s2 not in self.adj[s1]:
            self.adj[s1].append(s2)

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

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

3 Algorithmes de parcours d'un graphe

3.1 Parcours en profondeur

Programme 3 – Parcours en profondeur

```
def parcours_profondeur(g, s):
    vus = []
    parcours_rec(g, s, vus)
    return vus

def parcours_rec(g, s, vus):
    '''parcours en profondeur depuis le sommet s'''
    if s not in vus:
        vus.append(s)
        for v in g.voisins(s):
            parcours(g, v, vus)
```

Programme 4 – Parcours en profondeur version itérative

```
def parcours_profondeur_it(g, s):
    '''parcours en profondeur depuis le sommet s'''
    vus = []
    vus.append(s)
    pile = Pile()
    pile.empiler(s)
    while not pile.est_vide():
        s = pile.consulter()
        #Liste des voisins qui n'ont pas encore ete visite
        non_vus = [voisin for voisin in g.voisins(s) if voisin not in vus]
        #Si un voisin n'a pas encore ete visite, il est empile et on le visite
        if non_vus:
            voisin = non_vus[0]
            visite.append(voisin)
            pile.empiler(voisin)
        #Si tous les voisins sont visites, le sommet est depile
        else:
            pile.depiler()
```

3.2 Parcours en largeur

Programme 5 – Parcours en largeur

```python
def parcours_largeur(g, s):
    vus = []
    vus.append(s)
    file = File()
    file.ajouter(s)
    while not file.est_vide() :
        u = file.consulter()
        for v in g.voisins(u):
            if v not in vus:
                vus.append(v)
                file.ajouter(v)
        file.retirer()
    return vus
```


Auteur : Pascal Seckinger