Graphes

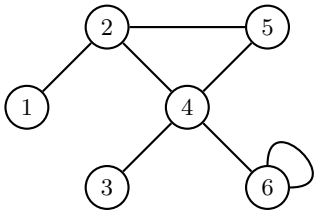
Vocabulaire

Graphe non orienté

Un **graphe non orienté** est un couple $G = (S, A)$ où S est un ensemble fini et A est un ensemble de paires $\{x, y\}$ où $x, y \in S$ avec $x \neq y$. Les éléments de S sont appelés **sommets** ou **nœuds**, les éléments de A sont appelés **arêtes**. Si $\{x, y\}$ est une arête, on dit que x et y sont **voisins**.

On autorise parfois les **boucles**, c'est-à-dire des arêtes qui relient un sommet à lui-même.

Exemple : soient $S = \{1, 2, 3, 4, 5, 6\}$ et $A = \{\{1, 2\}, \{2, 4\}, \{2, 5\}, \{3, 4\}, \{4, 5\}, \{4, 6\}, \{6, 6\}\}$. Le graphe (S, A) peut représenté graphiquement comme suit :



Le **degré** d'un sommet est le nombre d'arêtes qui partent de ce sommet. Le degré du sommet s est noté $d(s)$. Ainsi, dans l'exemple ci-dessus, on a $d(1) = 1$, $d(2) = 3$, $d(3) = 1$, $d(4) = 4$, $d(5) = 2$ et $d(6) = 3$ (la boucle est comptée deux fois).

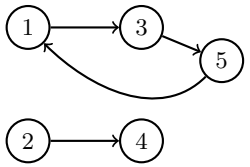
Une **chaîne** (ou un **chemin**) est une suite finie d'arêtes consécutives. La **longueur** d'une chaîne est le nombre d'arêtes qui la composent. Une chaîne est **simple** si ses arêtes sont deux à deux distinctes. Un **cycle** est une chaîne ayant même sommet de départ et d'arrivée. Dans l'exemple précédent $(\{1, 2\}, \{2, 4\}, \{4, 3\})$ (qu'on notera simplement $(1, 2, 4, 3)$) est une chaîne simple de longueur 3. La chaîne $(2, 5, 4, 2)$ est un cycle de longueur 3.

Un graphe est **connexe** si pour tout couple de sommets distincts il existe une chaîne qui les relie. Le graphe ci-dessus est connexe.

Graphe orienté

Un **graphe orienté** est un couple $G = (S, A)$ où S est un ensemble fini et A est un ensemble de couples (x, y) où $x, y \in S$ (autrement dit un sous-ensemble de $S \times S$). Les éléments de A sont appelés **arcs**.

Exemple : soient $S = \{1, 2, 3, 4, 5\}$ et $A = \{(1, 3), (2, 4), (3, 5), (5, 1)\}$. Le graphe (S, A) peut représenté graphiquement comme suit :

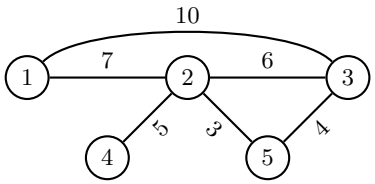


Le **degré sortant** d'un sommet s, noté $d_+(s)$, est le nombre d'arcs qui partent de s, le **degré entrant** de s, noté $d_-(s)$, est le nombre d'arcs qui y arrivent. Le **degré** de s est $d(s) = d_+(s) + d_-(s)$. Dans l'exemple ci-dessus, on a $d_+(1) = d_-(1) = 1$ (donc $d(1) = 2$), $d_+(2) = 1$ et $d_-(2) = 0$ (donc $d(2) = 1$), etc.

Un **chemin** est une suite finie d'arcs consécutifs. La **longueur** d'un chemin est le nombre d'arcs qui le composent. Un chemin est **simple** si ses arcs sont deux à deux distincts. Un **circuit** (ou **cycle orienté**) est un chemin ayant même sommet de départ et d'arrivée. Dans l'exemple précédent $(1, 3, 5, 1)$ est un circuit de longueur 3.

Graphe pondéré

Un **graphe pondéré** est un graphe (orienté ou non) dont les arcs ou les arêtes sont affectées d'un nombre réel appelé **poids**.



Le **poids** d'une chaîne ou d'un chemin d'un graphe pondéré est la somme des poids de ses arêtes ou de ses arcs. Par exemple, sur le graphe ci-dessus, le poids du chemin $(1, 2, 3, 5)$ est $7 + 6 + 4 = 17$.

Exemples d'applications

On peut représenter par un graphe de nombreuses situations concrètes :

– Réseau de transports : les sommets sont, selon le moyen de transport considéré, des villes, des gares, des aéroports, des stations de métro, des arrêts de bus, etc. et les arêtes les routes, voies ferrées, voies aériennes, etc. qui les relient. On peut pondérer le graphe par les distances ou les temps de parcours entre les sommets.

– Réseau informatique : les sommets sont les différents équipements (ordinateurs, routeurs, etc.) et les arêtes leurs moyens de communiquer.

– Réseau social : graphe non orienté dont les sommets sont les utilisateurs et les arêtes les liens d'amitié entre eux, ou graphe orienté dont les arcs traduisent la notion de "follower".

– Théorie des jeux : aux échecs par exemple, graphe orienté dont les sommets sont les configurations possibles de l'échiquier et les arcs les mouvements des pièces qui permettent de passer d'une configuration à une autre.

– Ordonnancement de tâches : graphe orienté dont les sommets sont les tâches à effectuer et les arcs traduisent le fait qu'une tâche doit être effectuée avant une autre.

– Graphe du web : graphe orienté dont les sommets sont les pages web et les arcs les liens entre ces pages.

Représentation informatique

Listes d'adjacence

On peut représenter un graphe en donnant pour chaque sommet la liste de ses voisins (pour un graphe non orienté) ou la liste des extrémités des arcs partant de ce sommet (pour un graphe orienté). En Python on peut implémenter ces **listes d'adjacence** à l'aide d'un dictionnaire dont les clés sont les sommets. Pour le premier graphe on obtient :

```
G = {1: [2], 2: [1, 4, 5], 3: [4], 4: [2, 3, 5, 6], 5: [2, 4], 6: [4, 6]}
```

Pour le deuxième graphe :

```
G = {1: [3], 2: [4], 3: [5], 4: [], 5: [1]}
```

Si le graphe est pondéré on peut remplacer les listes par des dictionnaires associant aux voisins du sommet le poids des arêtes correspondantes. Pour le troisième graphe :

```
G = {1: {2: 7, 3: 10}, 2: {1: 7, 3: 6, 4: 5, 5: 3}, 3: {1: 10, 2: 6, 5: 4},
     4: {2: 5}, 5: {2: 3, 3: 4}}
```

Cette représentation par listes d'adjacence est intéressante lorsque le nombre d'arêtes est relativement petit.

Matrice d'adjacence

On peut également représenter un graphe, orienté ou non, par une matrice. Si on numérote les sommets de 1 à n, la **matrice d'adjacence** du graphe est $M = (m_{ij})_{1 \leqslant i,j \leqslant n}$ où

$$m_{ij} = \begin{cases} 1 & \text{s'il existe une arête entre } i \text{ et } j \text{ (ou un arc de } i \text{ vers } j) \\ 0 & \text{sinon} \end{cases}.$$

Noter que si le graphe n'est pas orienté, sa matrice d'adjacence est symétrique.

Pour le premier graphe on obtient :

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

Pour le deuxième graphe :

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

On peut montrer que le $(i,j)^e$ coefficient de la matrice M^p est le nombre de chemins de longueur p qui relient les sommets i et j.

Si le graphe est pondéré, m_{ij} est le poids de l'arête ou de l'arc correspondant. Ainsi pour le troisième graphe on obtient :

$$M = \begin{pmatrix} 0 & 7 & 10 & 0 & 0 \\ 7 & 0 & 6 & 5 & 3 \\ 10 & 6 & 0 & 0 & 4 \\ 0 & 5 & 0 & 0 & 0 \\ 0 & 3 & 4 & 0 & 0 \end{pmatrix}$$

Pour un graphe pondéré, on posera parfois $m_{ii} = 0$ et $m_{ij} = -1$ si les sommets i et j ne sont pas reliés. Ce sera souvent le cas si les poids représentent des distances.

Parcours d'un graphe

Parcourir un graphe consiste à visiter ses sommets l'un après l'autre à partir d'un sommet initial. Il y a différentes manières de procéder.

Parcours en profondeur (DFS - Depth First Search)

L'idée est de suivre les chemins dans le graphe le plus loin possible en marquant les sommets qu'on visite. Si on est bloqué, on revient en arrière. Plus précisément, l'algorithme est le suivant :

On part du sommet initial. À chaque étape, on marque le sommet où on est et on se place sur un de ses voisins non encore marqué. S'il n'y en a pas, alors on revient au dernier sommet précédemment visité qui a encore des voisins non marqués.

Avec le premier graphe en partant du sommet 1 :

- On marque le sommet 1 comme visité. Il a un seul voisin : le sommet 2.
- On se place au sommet 2 et on le marque. Il a deux voisins : 4 et 5.
- On se place au sommet 4 et on le marque. Il a trois voisins non visités : 3, 5 et 6.
- On se place au sommet 3 et on le marque. Il n'a pas de voisins non visités.
- On revient au sommet 4. Il a deux voisins non visités : 5 et 6.
- On se place au sommet 5 et on le marque. Il n'a pas de voisins non visités.
- On revient au sommet 4. Il a un voisin non visité : 6.
- On se place au sommet 6 et on le marque. Il n'a pas de voisins non visités.
- On revient en 4. Il n'a pas de voisins non visités.
- On revient en 2. Il n'a pas de voisins non visités.
- On revient en 1. Il n'a pas de voisins non visités donc le parcours est terminé.

On a ainsi parcouru le graphe dans l'ordre 1, 2, 4, 3, 5 et 6.

Implémentation d'un parcours en profondeur

Pour marquer les sommets visités au cours du parcours d'un graphe, on peut utiliser un ensemble (objet de type `set` pour lequel l'ajout d'un élément (avec `add`) et le test d'appartenance (avec `in`) sont en $O(1)$) ou un dictionnaire.

Un parcours en profondeur se programme très simplement de manière récursive. La fonction suivante reçoit un graphe (représenté par un dictionnaire associant à chaque sommet sa liste d'adjacence) et le sommet de départ, et affiche les points visités par un parcours en profondeur. Le troisième paramètre `visites` est un ensemble qui contient les sommets déjà visités.

```
def DFS(G, s, visites):
    '''Parcourt le graphe G en profondeur en affichant les sommets visités.
       G est le dictionnaire des listes d'adjacence, s le sommet de départ.
       visites est l'ensemble des sommets déjà visités.'''
    visites.add(s)
    print(s, end=' ')
    for voisin in G[s]:
        if voisin not in visites:
            DFS(G, voisin, visites)
```

Avec le premier graphe :

```
>>> G = {1: [2], 2: [1, 4, 5], 3: [4], 4: [2, 3, 5, 6], 5: [2, 4], 6: [4, 6]}
>>> DFS(G, 1, set())   # set() = ensemble vide
1 2 4 3 5 6
```

Pour programmer un parcours en profondeur de manière itérative on utilise une **pile** (*stack* en anglais) : c'est une structure de données de type **LIFO** (Last In First Out) à laquelle on peut ajouter des éléments un par un (empilage) et les retirer un par un (dépilage) en $O(1)$. En Python on peut utiliser les listes avec `append` pour empiler et `pop` pour dépiler. On peut ainsi programmer un parcours en profondeur de la manière suivante :

```
def DFS_iter(G, s):
    visites = set()
    pile = [s]
    while pile: # tant que la pile n'est pas vide
        s = pile.pop()
        if s not in visites:
            visites.add(s)
            print(s, end=' ')
            for voisin in G[s]:
                pile.append(voisin)
```

Pour comprendre le fonctionnement de cette fonction, il faut l'appliquer à la main sur un exemple en observant l'évolution de la pile.

Avec le premier graphe :

```
>>> G = {1: [2], 2: [1, 4, 5], 3: [4], 4: [2, 3, 5, 6], 5: [2, 4], 6: [4, 6]}
>>> DFS_iter(G, 1)
1 2 5 4 6 3
```

On notera que les sommets ne sont pas parcourus dans le même ordre que précédemment. Cela vient du fait que les voisins d'un sommet sont retirés de la pile dans l'ordre inverse de celui dans lequel on les a ajoutés. Si on veut obtenir le même ordre que pour la fonction récursive, il suffit de remplacer `G[s]` par `G[s][::-1]`.

Parcours en largeur (BFS - Breadth First Search)

On visite les voisins du sommet de départ, puis les voisins de ses voisins, et ainsi de suite, en marquant à chaque fois les sommets visités.

Avec le premier graphe en partant du sommet 1 :

– On marque le sommet 1 comme visité. Il a un seul voisin : le sommet 2. On le marque.

– Le sommet 2 a deux voisins : 4 et 5. On les marque.

– Le sommet 4 a deux voisins non encore marqués : 3 et 6. On les marque.

– Le sommet 5 n'a pas de voisins non encore marqués.

– Le sommet 3 n'a pas de voisins non encore marqués.

– Le sommet 6 n'a pas de voisins non encore marqués.

On a ainsi parcouru le graphe dans l'ordre 1, 2, 4, 5, 3 et 6.

Implémentation d'un parcours en largeur

Pour programmer un parcours en largeur de manière efficace, on utilisera une **file** (*queue* en anglais). C'est une structure de données de type **FIFO** (First In First Out) à laquelle on peut ajouter des éléments un par un par la droite (enfilage) et les retirer un par un par la gauche (défilage) en $O(1)$. Les listes de Python ne sont pas adaptées, il vaut mieux utiliser le type `deque` (pour *double-entry queue*) du module `collections`.

```
>>> from collections import deque
>>> file = deque([])
>>> file.append(5)   # enfilage
>>> file.append(8)
>>> file
deque([5, 8])
>>> file.popleft()   # défilage
5
>>> file
deque([8])
>>> file.append(12)
>>> file.append(6)
>>> file
deque([8, 12, 6])
>>> file.popleft()
8
>>> file.popleft()
12
>>> file.popleft()
6
>>> file
deque([])
```

On peut ainsi programmer le parcours en largeur d'un graphe de la manière suivante :

– On met dans la file le sommet de départ.

– À chaque étape, on enlève le premier élément de la file, on visite ses voisins non encore marqués, on les marque et on les met dans la file.

– L'algorithme s'arrête lorsque la file est vide.

Comme pour le parcours en profondeur, on utilise un ensemble pour retenir les sommets visités.

On obtient ainsi la fonction suivante (où `G` est le dictionnaire des listes d'adjacence et `s` le sommet de départ) :

```
def BFS(G, s):
    '''Parcourt le graphe G en largeur en affichant les sommets visités.
       G est le dictionnaire des listes d'adjacence, s le sommet de départ.'''
    visites = {s}
    file = deque([s])
    while file:  # tant que la file n'est pas vide
        s = file.popleft()
        print(s, end=' ')
        for voisin in G[s]:
            if voisin not in visites:
                file.append(voisin)
                visites.add(voisin)
```

Avec le premier graphe :

```
>>> G = {1: [2], 2: [1, 4, 5], 3: [4], 4: [2, 3, 5, 6], 5: [2, 4], 6: [4, 6]}
>>> BFS(G, 1)
1 2 4 5 3 6
```

Comme précédemment, pour comprendre le fonctionnement de cette fonction, il faut l'appliquer à la main sur un exemple en observant l'évolution de la file.

Algorithme de Dijkstra

Principe de l'algorithme

Dans ce paragraphe on travaille avec des graphes pondérés (orientés ou non) à poids positifs. On parlera de longueur d'une arête/d'un chemin plutôt que de poids. On appelle distance entre deux sommets la longueur du plus court chemin qui les relie.

L'algorithme de Dijkstra permet de déterminer le plus court chemin entre deux sommets d'un tel graphe. L'idée est de calculer pour chaque sommet s sa distance $d(s)$ au point de départ de la manière suivante :

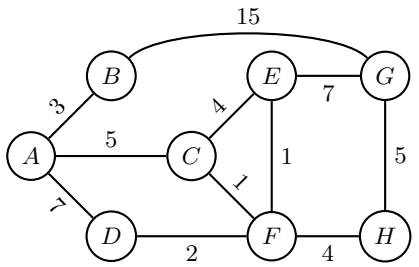
– On part du sommet de départ s_0. On pose $d(s_0) = 0$.

– À chaque étape, on prend le sommet s non visité dont la distance $d(s)$ au sommet de départ est la plus petite et on le marque comme visité. Pour chacun de ses voisins v non encore visités, on calcule la somme S de $d(s)$ et de la longueur de l'arête entre s et v. Si cette somme est plus petite que $d(v)$, ou si $d(v)$ n'a pas encore été définie, on pose $d(v) = S$.

– L'algorithme s'arrête quand le sommet d'arrivée est visité, ou quand il n'y a plus de sommets à visiter.

On obtient ainsi la distance entre les sommets de départ et d'arrivée. Si on veut également le chemin le plus court qui les relie, on peut soit retenir à chaque étape le sommet s avec lequel on a calculé $d(v)$, soit le retrouver après coup (cf exemple).

Cherchons par exemple le plus court chemin entre les sommets A et G du graphe suivant :



Le sommet de départ est A. On pose $d(A) = 0$.

Le sommet non visité le plus proche de A est A lui-même. On le marque comme visité. Les voisins de A sont B, C et D. On pose $d(B) = 3$, $d(C) = 5$ et $d(D) = 7$.

Le sommet non visité le plus proche de A est B. On le marque comme visité. Son seul voisin non visité est G : on pose $d(G) = d(B) + 15 = 18$.

Le sommet non visité le plus proche de A est maintenant C. On le marque comme visité. Ses voisins non visités sont E et F : on pose $d(E) = d(C) + 4 = 9$ et $d(F) = d(C) + 1 = 6$.

Le sommet non visité le plus proche de A est maintenant F. On le marque comme visité. Ses voisins non visités sont D, E et H. On ne modifie pas $d(D)$ car $d(F) + 2 = 8 \geqslant d(D)$. En revanche $d(F) + 1 = 7 < d(E)$ donc on modifie $d(E)$ en posant $d(E) = 7$. Enfin on pose $d(H) = d(F) + 4 = 10$.

Les sommets non visités les plus proches de A sont maintenant D et E. On choisit D. On le marque comme visité. Il n'a pas de voisins non visités.

Le sommet non visité le plus proche de A est maintenant E. On le marque comme visité. Son seul voisin non visité est G et $d(E) + 7 = 14 < d(G)$ donc on modifie $d(G)$ en posant $d(G) = 14$.

Le sommet non visité le plus proche de A est maintenant H. On le marque comme visité. Son seul voisin non visité est G mais $d(H) + 5 = 15 \geqslant d(G)$ donc on ne modifie pas $d(G)$.

Le sommet non visité le plus proche de A est maintenant G : c'est le sommet d'arrivée donc l'algorithme s'arrête.

On a ainsi déterminé que la distance de A et G est 14. Pour trouver le chemin correspondant, il suffit de retrouver les prédécesseurs de chaque sommet :

– On a $d(E) + 7 = d(G)$ donc avant G on était en E.

– On a $d(E) = d(F) + 1$ donc avant E on était en F.

– On a $d(F) = d(C) + 1$ donc avant F on était en C.

– On a $d(C) = d(A) + 5$ donc avant C on était en A.

Le chemin le plus court entre A et G est donc (A, C, F, E, G). Bien évidemment on aurait pu aussi noter à chaque modification de d le prédécesseur correspondant (c'est ce qu'on fera au paragraphe suivant).

Implémentation

Le graphe pondéré sera représenté par un dictionnaire associant à chaque sommet le dictionnaire associant aux voisins du sommet le poids des arêtes correspondantes.

On va utiliser un dictionnaire pour noter les distances au sommet de départ et un ensemble pour noter les sommets visités.

La recherche du sommet le plus proche du départ peut se faire de manière efficace en utilisant une **file de priorité** : c'est une structure de données analogue à une liste à laquelle on peut ajouter des éléments et en extraire le plus petit avec une complexité logarithmique. Les files de priorité peuvent être implantées en Python avec le module `heapq` : la fonction `heappop` permet de retirer le plus petit élément de la file et la fonction `heappush` d'y ajouter un élément.

```
>>> from heapq import heappop, heappush
>>> file = []
>>> heappush(file, 10)
>>> heappush(file, 3)
>>> heappush(file, 5)
>>> file
[3, 10, 5]  # le plus petit se retrouve en tête
>>> x = heappop(file)
>>> x
3
>>> file
[5, 10]     # le plus petit se retrouve en tête
```

On va ainsi placer dans une file de priorité les couples (distance, sommet) à mesure qu'ils sont créés (quand on place des couples dans une telle file, le minimum est pris par rapport au premier élément du couple). On pourra ainsi récupérer efficacement à chaque étape le sommet le plus proche du départ.

Enfin, pour récupérer le chemin le plus court entre les sommets de départ et d'arrivée, on va ajouter un dictionnaire associant à chaque sommet son prédécesseur qu'on met à jour en même temps que le dictionnaire des distances. Une fois l'algorithme terminé, on crée le chemin en partant du sommet d'arrivée et en prenant son prédécesseur, puis le prédécesseur de celui-ci, etc., jusqu'à ce qu'on arrive au départ.

On obtient ainsi la fonction suivante, qu'on appliquera à la main sur un exemple pour bien comprendre son fonctionnement. Noter que s'il n'y a pas de chemin entre les sommets de départ et d'arrivée, la fonction lève une assertion.

```
from heapq import heappop, heappush

def plus_court_chemin(G, depart, arrivee):
    '''Renvoie le plus court chemin entre les sommets de départ et d'arrivée
       obtenu par l'algorithme de Dijkstra, s'il existe.
       G est un dictionnaire qui à chaque sommet associe un dictionnaire qui
       associe à ses voisins le poids de l'arête/arc qui les relie.'''
    d = {depart: 0}      # dictionnaire des distances au sommet de départ
    pred = {}            # dictionnaire des prédécesseurs
    visites = set()
    file = [(0, depart)]

    # Construction du dictionnaire des distances
    while file and arrivee not in visites:
        dmin, smin = heappop(file)   # point le plus proche du départ
        if smin not in visites:
            visites.add(smin)
            for v in G[smin]:
                S = dmin + G[smin][v]
                if v not in d or S < d[v]:
                    d[v] = S
                    pred[v] = smin
                    heappush(file, (S, v))

    # Recherche du plus court chemin (s'il existe)
    assert arrivee in visites, "Pas de chemin"
    chemin = [arrivee]
    s = arrivee
    while s != depart:
        s = pred[s]
        chemin.append(s)
    return chemin[::-1]
```

Avec le graphe précédent :

```
>>> G = {'A': {'B': 3, 'C': 5, 'D': 7}, 'B': {'A': 3, 'G': 15},
     'C': {'A': 5, 'E': 4, 'F': 1}, 'D': {'A': 7, 'D': 2},
     'E': {'C': 4, 'F': 1, 'G': 7}, 'F': {'C': 1, 'D': 2, 'E': 1, 'H': 4},
     'G': {'B': 15, 'E': 7, 'H': 5}, 'H': {'F': 4, 'G': 5}}
>>> plus_court_chemin(G, 'A', 'G')
['A', 'C', 'F', 'E', 'G']
```