06 - Les graphes

1 Définitions générales

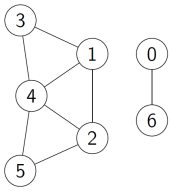
1.1 Graphes non orientés

Définition 1 - *Graphe non orienté, sommet, arête*
Un graphe non orienté $G = (S, A)$ est un couple formé de deux ensembles :

- un ensemble $S = \{s_0, \dots, s_{n-1}\}$ dont les éléments sont appelés **sommets** ou **noeuds** du graphe.
- un ensemble A formé de parties à deux éléments de S ; les éléments de A sont appelés les **arêtes** ou **arcs** du graphe.

Remarque* 1 *: Une arête peut relier un sommet à lui-même : de telles arêtes sont appelées **boucles**.
Dans tout ce cours, on travaillera avec des graphes sans boucles, on parle de graphes simples.

Exemple 1 : Décrire les ensembles S et A pour le graphe suivant :



Définition 2 - *Degré*
Soit $G = (S, A)$ un graphe non orienté, et s un sommet de ce graphe. On appelle **degré de *s*** le nombre d'arêtes qui le relient aux autres sommets.

Exemple 2 : Donner le degré de chaque sommet du graphe précédent.

Définition 3 - *Chemin, cycle*
Soit $G = (S, A)$ un graphe non orienté, et s et t deux sommets de ce graphe.

- Un **chemin** de longueur k qui relie s à t est une suite de sommets $\{c_0, \dots, c_k\}$ telle que $c_0 = s$, $c_k = t$ et pour tout i de $[\![1; k]\!]$, $\{c_{i-1}, c_i\}$ est une arête.
- Un **cycle** est un chemin dont les sommets de départ et d'arrivée sont identiques.

Exemple 3 : On reprend le graphe de l'exemple 1. Donner deux chemins de différentes longueurs, qui relient le sommet 2 au sommet 3. Et pour relier les sommets 0 et 1 ?

Définition 4 - *Connexité*
Un graphe est **connexe** si pour tout couple de sommets, il existe un chemin qui les relie.

Exemple 4 : Le graphe de l'exemple 1 est-il connexe ?

1.2 A quoi ça sert ?

A plein de choses ! Citons en vrac :

• les réseaux routiers ou ferroviaires (application type Waze ou Google Maps)

• le Web (sommets=pages internet, arête=lien entre deux pages)

• les réseaux sociaux (sommets=les individus, arête= connexion entre deux individus)

Une fois que l'on a modélisé la situation, on peut alors, grâce à la théorie des graphes, répondre à de nombreux problèmes : rechercher des plus courts chemins, établir une connexion entre des individus, etc...

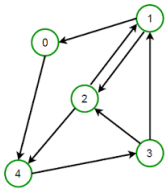
1.3 Graphes orientés

Définition 5 - *Graphe orienté, sommet, arête*
Un graphe orienté $G = (S, A)$ est un couple formé de deux ensembles :

- un ensemble $S = \{s_0, \ldots, s_{n-1}\}$ dont les éléments sont appelés **sommets** ou **noeuds** du graphe.
- un ensemble A formé de **couples** d'éléments de S ; les éléments de A sont appelés les **arêtes** ou **arcs** du graphe.

Remarque* 2 *: Les graphes orientés apparaissent naturellement sur nos exemples précédents : dans un réseau social Alice peut suivre Bob sans que Bob suive Alice ; dans les pages web une page peut contenir un lien vers une autre page sans que ce soit réciproque, etc.

Exemple 5 : Décrire les ensembles S et A pour le graphe suivant :



Définition 6 - *Degré sortant et degré entrant*
Soit $G = (S, A)$ un graphe orienté, et s un sommet de ce graphe.

- On appelle **degré sortant de *s***, noté le nombre d'arêtes dont l'extrémité initiale est s. On le note $d_+(s)$.
- On appelle **degré entrant de *s***, noté le nombre d'arêtes dont l'extrémité finale est s. On le note $d_-(s)$.

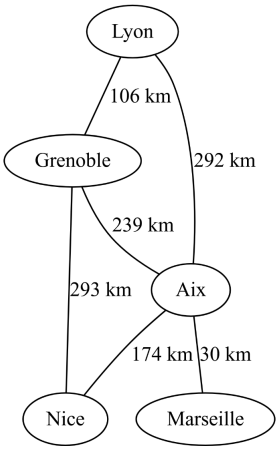
Exemple 6 : Donner le degré sortant et entrant de chaque sommet du graphe précédent.

1.4 Pondération d'un graphe

Dans le cas d'une carte routière, les arêtes sont les routes reliant deux nœuds routiers. Il peut être intéressant de faire figurer la longueur de cette route. Nous utiliserons pour ça des étiquettes apposées sur les arêtes.

Définition 7 - *Graphe pondéré*
Un graphe pondéré est un triplet $G = (S, A, w)$, où (S, A) est un graphe, et où $w : A \to \mathbb{R}$ est une fonction qui à chaque arête associe une étiquette (ou pondération).

Exemple 7 : (Données issues de Géoportail avec l'itinéraire le plus court)



2 Représentations d'un graphe simple

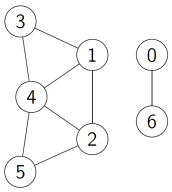
Nous allons décrire différentes manières d'implémenter les graphes, de manière à pouvoir les étudier efficacement.

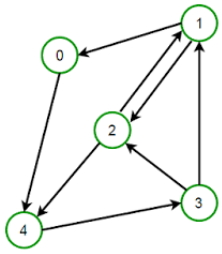
2.1 Matrice d'adjacence

On suppose que les sommets du graphe sont numérotés de 0 à $n-1$.

Définition 8
La matrice d'adjacence associée à un graphe est une matrice à n lignes et n colonnes, remplie de 0 et de 1. Le coefficient d'indice (i, j) vaut 1 si et seulement si il existe une arête du sommet i au sommet j.

Exemple 8 : Donner les matrices d'adjacence pour les deux graphes suivants :





Pour représenter la matrice d'adjacence d'un graphe sous Python, on utilisera un tableau `numpy` (déclaré ainsi : `M=np.array([[0,1],[1,0]])`).

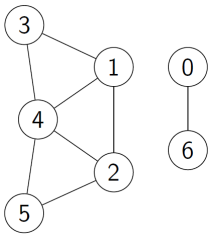
Remarque* 3 *(Avantages et inconvénients) :

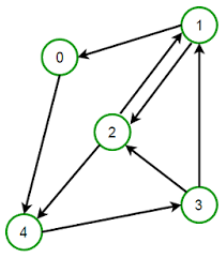
- La matrice d'adjacence permet d'avoir un accès rapide à l'existence d'un arc entre deux sommets (en effet l'accès aux éléments d'une matrice s'effectue en temps constant).
- La matrice contient beaucoup de 0 d'autant plus que le graphe est peu dense, i.e. avec peu d'arêtes) et prend donc beaucoup de place en machine. Une façon de contourner ce problème est d'utiliser une autre manière de coder la matrice, avec un dictionnaire par exemple (voir DS n°1 et l'exercice sur les matrices parcimonieuses).
- Pour déterminer s'il existe une arête reliant deux sommets donnés, il est nécessaire, dans le pire des cas, d'effectuer un parcours de la ligne correspondante de la matrice, et la complexité est donc linéaire.

2.2 Liste d'adjacence

Définition 9 - *Liste d'adjacence d'un sommet, d'un graphe*
• La liste d'adjacence d'un sommet est la liste de ses voisins (pour un graphe non orienté) ou de ses successeurs (pour un graphe orienté).
• La liste d'adjacence d'un graphe est la liste des listes d'adjacences de ses sommets (les sommets étant numérotés de 0 à $(n-1)$ s'il y a n sommets).

Exemple 9 : Construire la liste d'adjacence des deux graphes suivants :





Pour représenter la liste d'adjacence d'un graphe sous Python, on utilisera simplement une liste de listes.

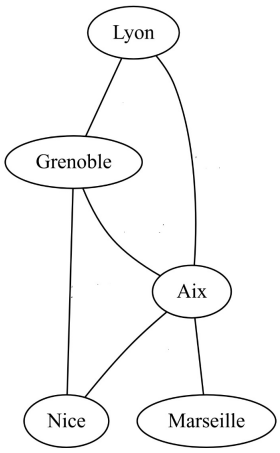
Remarque* 4 *(Avantages et inconvénients) :

- Pour des graphes peu denses, les listes d'adjacences seront plus courtes, et la mémoire utilisée plus faible que lors d'utilisation des matrices d'adjacences.
- Pour déterminer si un sommet a des voisins (ou un successeur), on peut utiliser l'opérateur `len` dont la complexité est en temps constant.
- Mais certaines propriétés du graphe ne peuvent s'obtenir qu'avec la matrice M d'adjacence, comme par exemple *"le coefficient (i, j) de la matrice M^p est le nombre de chemins de longueur p qui relient le sommet i au sommet j"* (pas au programme)

2.3 Dictionnaires

Si les sommets du graphe ne sont pas numérotés (par exemple ce sont des noms de villes), on peut alors utiliser un dictionnaire pour modéliser le graphe : chaque clé sera un sommet du graphe, et la valeur associée sera la liste d'adjacence du sommet.

Exemple 10 : Donner le dictionnaire associé au graphe suivant :



3 Premiers exercices d'implémentation de graphes en Python

Exercice 1 : 1. Ecrire une fonction `MatriceToListe`, ayant pour paramètre d'entrée la matrice d'adjacence d'un graphe (sous forme d'un tableau `numpy`), et qui retourne la liste d'adjacence de ce graphe.

2. Ecrire une fonction `ListeToMatrice`, ayant pour paramètre d'entrée la liste d'adjacence d'un graphe, et qui retourne la matrice d'adjacence de ce graphe.

Exercice 2 : Pour chacun des problèmes ci-dessous, vous écrirez deux fonctions : une qui prend en entrée la matrice d'adjacence d'un graphe, et l'autre qui prend en entrée la liste d'adjacence d'un graphe. Il est interdit ici d'utiliser les fonctions créées dans l'exercice précédent.

1. Déterminer le degré d'un sommet donné pour un graphe non orienté.
2. Déterminer si deux sommets donnés sont voisins, i.e. reliés par une arête. (dans le cas d'un graphe orienté, étant donnés deux sommets s et t, il s'agit de savoir s'il existe une arête partant de s et arrivant à t)

4 Parcours de graphes

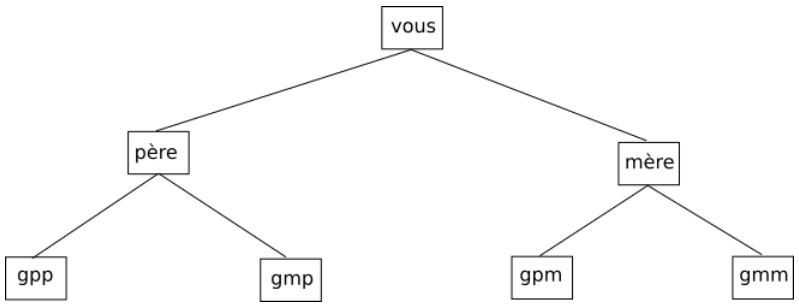
Parcourir efficacement les sommets d'un graphe en utilisant les arêtes qui les relient est essentiel pour par exemple :

- savoir s'il existe un chemin entre deux sommets donnés,
- déterminer le plus court chemin entre deux sommets donnés,
- savoir si un graphe non orienté est connexe,
- etc.

Nous allons tout d'abord expliquer les deux principes de parcours avec un graphe particulier : un arbre.

C'est un graphe sans cycle.

Voici par exemple un arbre généalogique :



Il y a essentiellement deux manières de parcourir cet arbre :

- **en profondeur** : on parcourt toute une branche généalogique avant de passer à la suivante. Ici, en choisissant les branches de gauche en premier, on obtient la succession de personnes : vous, père, gpp, gmp, mère, gpm, gmm.
- **en largeur** : on parcourt l'arbre génération par génération. Ici on obtient la succession de personnes : vous, père, mère,gpp, gmp, gpm, gmm.

Nous allons voir comment ces deux types de parcours se généralisent à un graphe quelconque. Nous garderons le vocabulaire de parcours en profondeur/largeur.

Remarque* 5 *: Dans les arbres il n'y a pas de cycles, on ne peut donc pas rencontrer deux fois le même sommet dans les parcours expliqués ci-dessus. Dans les graphes, ce n'est pas le cas. Il faudra donc veiller à ce que les sommets ne soient pas visités plusieurs fois (sinon notre parcours pourrait ne jamais s'arrêter).

4.1 Parcours en profondeur

4.1.1 Principe de l'algorithme

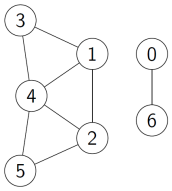
L'algorithme de parcours en profondeur d'un graphe consiste, partant d'un sommet de départ, à visiter les sommets du graphe en favorisant la génération de sommets suivants par rapport à la génération actuelle.

Il repose sur les étapes suivantes :

- créer une liste des sommets visités qui initialement est vide.
- créer une **pile** des sommets à traiter contenant initialement le sommet de départ.
- tant qu'il reste des sommets dans la pile :
  - extraire le sommet du haut de la pile.
  - si ce sommet n'est pas déjà dans la liste des sommets visités :
    - l'ajouter à la liste des sommets visités, et parcourir les sommets voisins.
    - pour chaque voisin : s'il ne fait pas partie des sommets déjà visités, l'ajouter à la pile des sommets à traiter.
- retourner la liste des sommets visités.

4.1.2 Illustration

Parcours en profondeur du graphe ci-dessous en partant du sommet 5 :



4.1.3 Implémentation en Python

Exercice 3 : Écrire une fonction de paramètre une liste `L` d'adjacence d'un graphe, et un sommet `d` de départ, et qui retourne la liste des sommets visités à partir de ce sommet de départ, en utilisant le principe du parcours en profondeur.

Amélioration de ce programme : à chaque fois que l'on a besoin de savoir si le sommet que l'on rencontre a déjà été visité, le parcours de la liste des sommets déjà visités a une complexité linéaire. On peut améliorer cela en marquant les sommets que l'on a déjà visité ainsi : on crée une liste `marquage` de longueur le nombre de sommets du graphe, qui au départ ne contient que les booléens `False` ; à chaque fois que l'on visite un sommet `s`, alors on passe `marquage[s]` à `True`. Pour savoir si un sommet `c` a déjà été visité, il suffit d'accéder à `marquage[c]`, ce qui se fait en complexité constante.

Exercice 4 : Améliorer la fonction précédente en suivant ce principe.

4.1.4 Application à la détection de connexité

Exercice 5 : Ecrire une fonction permettant de savoir si un graphe non orienté est connexe ou non. On utilisera la fonction de parcours en profondeur, en s'interrogeant sur le résultat de cette fonction lorsque le graphe est connexe ou lorsqu'il ne l'est pas.

4.2 Parcours en largeur

4.2.1 Principe de l'algorithme

L'algorithme de parcours en largeur d'un graphe consiste, partant d'un sommet de départ, à visiter les sommets du graphe en traitant d'abord tous les voisins d'un sommet donné avant de passer à la génération suivante.

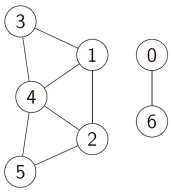
Il repose sur les étapes suivantes :

- créer une liste des sommets visités contenant initialement le sommet de départ.
- créer une **file** des sommets à traiter contenant initialement le sommet de départ.
- tant qu'il reste des sommets dans la file :
  - extraire le sommet en tête de file.
  - pour chaque voisin de ce sommet, s'il ne fait pas partie des sommets déjà visités :
    - l'ajouter à la liste des sommets visités
    - l'ajouter à la file des sommets à traiter.
- retourner la liste des sommets visités.

Le fait d'utiliser une file (alors qu'on utilisait une pile dans le parcours en profondeur) a pour conséquence que les nouveaux voisins sont traités après les anciens. On privilégie donc les sommets d'une même génération par rapport à ceux des générations suivantes.

4.2.2 Illustration

Parcours en largeur du graphe ci-dessous en partant du sommet 5 :



4.2.3 Implémentation en Python

Exercice 6 : Écrire une fonction de paramètre une liste `L` d'adjacence d'un graphe, et un sommet `d` de départ, et qui retourne la liste des sommets visités à partir de ce sommet de départ, en utilisant le principe du parcours en largeur.

N.B. *:*

- *Pour implémenter la structure de file, vous utiliserez des objets de type* **`deque`**, *(penser à les importer avec* **`from collections import deque`**), *et les méthodes associés* **`append`** *et* **`popleft`**.
- *Pour savoir si un sommet à déjà été visité, vous utiliserez un tableau de booléens, comme dans l'amélioration proposée de l'algorithme de parcours en profondeur.*

4.3 Plus courts chemins

4.3.1 Arêtes de poids constants

Etant donné deux sommets x et y d'un graphe, on cherche le chemin le plus court (i.e. celui qui contient le moins d'arêtes) dont le point de départ est x et le point d'arrivée est y.

Pour cela il suffit d'effectuer un parcours en largeur du graphe en démarrant depuis le point x.
En effet lors de ce type de parcours :
• on démarre de x ;
• puis on visite les voisins de x ;
• puis on visite les voisins des voisins de x ;
• ...
• jusqu'à avoir visité tous les sommets atteignables en partant de x.

On va donc tenir à jour, dans l'algorithme du parcours en largeur à partir de x, une liste **`dist`** qui contiendra les distances entre x et chacun des sommets du graphe. Initialement cette liste sera initialisée avec des $+\infty$ (ce qui signifie que le sommet n'a pas été relié à x), sauf **`dist[x]`** qui vaut 0.(en Python $+\infty$ se code **`float('inf')`**, ou bien **`np.inf`** si **`np`** désigne le module **`numpy`**)

Cette liste de distance sera tenue à jour à chaque tour de la boucle **`tant que`** en suivant le pseudo-code :

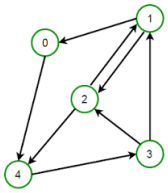
```
1 tant que la file n'est pas vide
2     s = la tête de file
3     pour tout voisin v de s
4         si v n'a pas été visité
5             dist[v]=dist[s]+1
6             ajouter v à la file
7             marquer v comme visité
```

Exercice 7 : Écrire une fonction de paramètre une liste `L` d'adjacence d'un graphe, et un sommet `d` de départ, et qui retourne la liste des distances de chacun des sommets à partir de ce sommet de départ.

N.B. : S'il n'existe pas de chemin entre le sommet de départ et un autre sommet, alors à la fin la distance correspondante dans la liste sera $+\infty$.

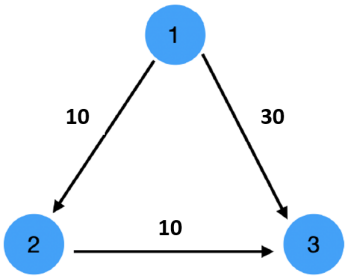
Exercice 8 : Cet algorithme fonctionne aussi sur un graphe orienté.

Tester votre programme sur le graphe suivant, en calculant les distances en partant du sommet 0.



4.3.2 Arêtes pondérées de poids positifs : l'algorithme de Dijkstra

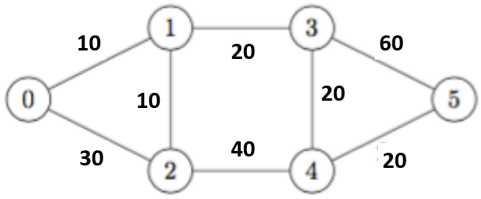
Lorsque nous avons à faire à un graphe pondéré, un simple parcours en largeur ne suffit pas à trouver le plus court chemin entre deux sommets.
Sur l'exemple ci-contre, la longueur du plus court chemin entre les sommets 1 et 3 est 20, mais le parcours en largeur nous donnerait 30.



Nous avons besoin de l'algorithme de Dijkstra (*Edsger Wybe Dijkstra est un mathématicien et informaticien néerlendais né en 1930 et mort en 2002*).

Tout d'abord, nous allons supposer que le graphe pondéré est implémenté de la façon suivante : nous allons utiliser une liste d'adjacence adaptée, où pour chaque sommet, la liste de ses voisins sera remplacée par un liste de couples (v,w), où v sera le numéro du voisin, et w le poids de l'arête correspondante.

Exercice 9 : Ecrire les instructions Python permettant de déclarer avec ce principe le graphe suivant :



Principe de l'algorithme de Dijkstra :

Paramètres d'entrée : un graphe pondéré implémenté sous la forme d'une liste d'adjacence, et un sommet de départ s.

En sortie : la fonction retourne la liste des distances minimales entre le sommet s et tous les sommets du graphe.

1. On crée la liste `S` de tous les sommets.

2. On crée la liste `dist` des distances, dont toutes les valeurs sont initialisées à $+\infty$, sauf `dist[s]` qui vaut 0.

3. Tant que `S` est non vide :

   3.1. On cherche le sommet v appartenant à S tel que `dist[v]` soit la plus petite valeur des distances entre s et tous les sommets **de S**

   3.2. Si cette distance est infinie, on sort de la boucle "Tant que" (aucun sommet de S n'est connecté à s)

   3.3. Pour tous les voisins u de v qui sont dans S, `dist[u]=min(dist[u],dist[v]+poids(v,u))`

   3.4. On enlève v de S.

Exercice 10 : Ecrire une fonction `Dijkstra` répondant au problème posé, i.e. prenant en entrée une liste d'adjacence comme décrit dans l'exercice 10, et un sommet de départ s, et retournant la liste des distances minimales du sommet s à tous les autres sommets.

Tester cette fonction avec l'exemple du graphe de l'exercice 10, en partant du sommet 0.