Chapitre 3 : Un peu de graphes

I Qu'est-ce qu'un graphe ?

I.1 Des sommets et des arcs

Définition I.1 Un **graphe orienté** $G(S, A)$ est la donnée d'un ensemble fini S et d'une partie A de $S \times S$.
Les éléments de S sont appelés **sommets** (ou **noeuds**) du graphe.
Les éléments de A sont appelés **arcs** du graphe.

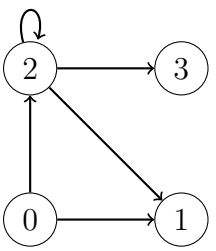
■ **Exemple 1** On peu représenter le graphe $G(\{1,2,3,4\}, \{(1,3),(3,1),(1,2),(3,4),(4,4)\})$ graphiquement : ■

Définition I.2 Un **graphe non-orienté** $G(S, A)$ est la donnée d'un ensemble fini S et d'un ensemble A de parties de S à 2 éléments.
Les éléments de A sont appelés **arêtes** du graphe.

■ **Exemple 2** Le graphe non-orienté complet d'ordre 4, est le graphe ayant 4 sommets qui sont tous reliés les uns aux autres : ■

Définition I.3
- Si $G(S, A)$ est un graphe orienté, pour tout arc $(s, s') \in A$, on dit que s est un **précédesseur** de s' et que s' est un **successeur** de s.
  Pour tout sommet $s \in S$ on note
  - ▷ $d_+(s)$ le nombre de successeurs de s : c'est le **degré sortant** de s ;
  - ▷ $d_-(s)$ le nombre de prédécesseurs de s : c'est le **degré entrant** de s.
- Si $G(S, A)$ est un graphe non orienté, pour tout sommet $s \in S$, on dit que s' est un **voisin** de s si $\{s, s'\} \in A$. On note $d(s)$ le nombre de voisins de s : c'est le **degré** de s.

■ **Exemple 3** 1. Dans le graphe orienté suivant :



2. Si on oublie les flèches et la boucle dans le graphe précédent :

■

I.2 Listes et matrices d'adjacence

On peut représenter un graphe par ses **listes d'adjacence** : on donne S et pour chaque élément s de S, on donne la liste des sommets tels qu'il existe un arc (ou une arête) de s à cet élément.

■ **Exemple 4** 1. Reprenons le graphe de l'exemple précédent. Ses listes d'adjacence sont :

2. Le graphe non orienté obtenu à partir de l'exemple précédent a pour listes d'adjacence :

■

■ **Remarque 1** Pour implémenter cette représentation sous python, on pourra par exemple utiliser un dictionnaire dont les clés sont les sommets et les valeurs sont les listes d'adjacence associées. ■

On peut aussi représenter un graphe par sa **matrice d'adjacence** : on numérote les sommets de 1 à n et on obtient une matrice carré de taille n en mettant un 1 en position (i, j) s'il y a un arc (ou une arête) joignant le sommet i au sommet j et 0 sinon.

■ **Exemple 5** 1. La matrice d'adjacence du graphe orienté précédent est :

2. La matrice d'adjacence du graphe non orienté précédent est :

■

■ **Remarque 2** Si le graphe n'est pas orienté, alors la matrice d'adjacence est symétrique avec des 0 sur la diagonale. ■

Définition I.4

- Un arc qui part et arrive au même sommet est une **boucle**.
- Un **chemin** (resp. **chaîne**) est une suite finie $(s_0, \dots, s_k)$ d'éléments de S telle que pour tout $i \in [\![0, k-1]\!]$, il existe un arc (resp. une arête) entre s_i et s_{i+1} : $(s_i, s_{i+1}) \in A$ (resp. $\{s_i, s_{i+1}\} \in A$).
  La **longueur** du chemin est le nombre $k-1$ d'arcs utilisés pour relier s_0 à s_k.
  Le chemin (resp. une chaîne) est dit **simple** s'il ne passe pas deux fois par un même arc (resp. arête).
- On dit que le graphe non-orienté $G(S, A)$ est **connexe** si pour tout couple de sommets de G il existe une chaîne les reliant.
- Un **circuit** (resp. **cycle**) est un chemin (resp. chaîne) simple $(s_0, \dots, s_k)$ tel que $s_0 = s_k$.

■ **Exemple 6** 1. Dans le graphe orienté précédent, on a 2 chemins de longueur 2 :
Il n'y a pas de circuit.

2. Dans le graphe non orienté correspondant, on a 1 chemin simple de longueur 3 et ... chemins simples de longueur 2.
   Il y a un cycle :

■

Théorème I.1 Soit M la matrice d'adjacence du graphe G. Pour tout entier $n \geq 1$, le coefficient (i, j) de M^n est le nombre de chemins de longueur n joignant le sommet i au sommet j.

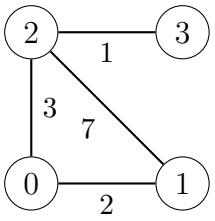
■ **Exemple 7** Prenons la matrice d'adjacence du graphe non orienté précédent. Mise au carré : ■

I.3 Graphes pondérés

Définition I.5 Un graphe $G(S, A)$ **pondéré** est un graphe dont les arcs (ou les arêtes) sont etiquetés par un nombre entier appelé **poids**.
On peut représenter cette donnée supplémentaire par une fonction de A vers $\mathbb{Z}$.
Le **poids** d'un chemin dans un graphe pondéré est la somme des poids des arcs qui forment ce chemin.

On peut encore utiliser la représentation par listes d'adjacence, en précisant pour chaque voisin, le poids de l'arc permettant d'y accéder.
On peut aussi utiliser une matrice d'adjacence en mettant en position (i, j) le poids de l'arc reliant i à j (ou 0 s'il n'y en a pas).

■ **Exemple 8** La matrice d'adjacence du graphe :



■

Beaucoup de problèmes concrets se résument à chercher un chemin de poids minimal reliant deux sommets dans un graphe :

- élaboration d'un itinéraire ;
- mise en connexion de deux machines dans un réseau ;
- tournées de distibution ;
- classement de pages web ;
- bio-informatique, etc...

II Algorithmes sur les graphes

II.1 Piles et files

Définition II.1 Une **pile** (ou **stack**) est une structure de donnée **linéaire** (les éléments sont rangés sur une ligne) qui permet de stocker des éléments en utilisant le principe **LIFO** : Last In First Out.
Le dernier élément ajouté (ou **empilé**) est le **sommet** : c'est le seul élément accessible. On y accède en le retirant (ou en le **dépilant**) de la pile.

■ **Remarque 3** En python, on peut réaliser la structure de pile en utilisant des listes : on ne peut utiliser que l'initialisation à `[]`, les méthodes `append` et `pop` et la fonction `len`. ■

Définition II.2 Une **file** (ou **queue**) est une structure de donnée linéaire qui permet de stocker des éléments en utilisant le principe **FIFO** : First In First Out.
Le premier élément ajouté (ou **enfilé**) est la **tête** : c'est le seul élément accessible. On y accède en le retirant (ou en le **défilant**) de la file.

■ **Remarque 4** En python, on pourrait implémenter les files avec des listes : en utilisant `append` pour enfiler et `pop(0)` pour défiler. Toutefois, cette dernière instruction est assez coûteuse en complexité.
Il y a un module qui implémente efficacement la structure de file : `collections.deque`. Celle-ci fonctionne comme les listes avec les méthodes `append` et `popleft` et la fonction `len`. ■

■ **Exemple 9** Voir le fichier python associé. ■

II.2 Parcours d'un graphe

Le parcours d'un graphe permet de répondre à divers problèmes qui se posent sous la même forme : il s'agit de trouver un ensemble de sommets ou de chemins vérifiant une certaine propriété. Pour cela, on va voir deux façons de procéder :

- le **parcours en profondeur** : à partir d'un sommet, on passe à un de ses voisins puis à un voisin de celui-ci, et ainsi de suite. S'il n'y a pas de voisin, on revient au sommet précédent et on passe à un autre voisin et on recommence ;
- le **parcours en largeur** : à partir d'un sommet, on visite tous ses voisins, puis tous les voisins de ses voisins et ainsi de suite.

II.2.1 Parcours en profondeur

On utilise une pile pour stocker les sommets à visiter :

- au départ, le sommet de départ est placé dans la pile ;
- tant que tous les sommets du graphe n'ont pas été visité, on dépile et les voisins du sommet sont empilés.

On a besoin d'une structure de donnée qui permet de vérifier qu'un sommet n'a pas encore été visité avant de le visiter.

Voici le pseudo-code correspondant :

```
#Entrée : graphe, origine
attente <- [origine] #une pile
visites <- [] #un dictionnaire
tant que attente est non vide
    sommet <- depile(attente)
    si sommet pas dans visites
        inserer(visites, sommet)
        pour tout voisin dans graphe[sommet]
            empiler(attente, voisin)
```

■ **Exemple 10** Voici un algorithme qui permet de déterminer si un graphe est connexe : le graphe est donné par ses listes d'adjacences.

```
def est_connexe(G:dict, origine) -> bool:
    '''
    Entrée : G est un graphe donné par le dictionnaire de ses listes d'adjacences,
    origine est un sommet de G.
    Sortie : renvoie True si G est connexe, False sinon
    '''
    visites = {}
    attente = [origine]
    while
        sommet =
        if

            for

    if
        return True
    else:
        return False
```

Un variant de boucle est :
La boucle `while` s'arrête lorsque

■

II.2.2 Parcours en largeur

On utilise cette fois une file :

- au départ, le sommet de départ est placé dans la file ;
- tant que tous les sommets du graphe n'ont pas été visités, on défile et les voisins du sommet sont enfilés.

Ainsi, on va visiter tous les voisins à distance 1 de l'origine, puis tous les voisins à distance 2, etc...
Voici le pseudo-code correspondant :

```
#Entrée : graphe, origine
attente <- [origine] #une file
visites <- [origine] #un dictionnaire
tant que attente est non vide
    sommet <- defile(attente)
    pour tout voisin dans graphe[sommet]
        si voisin pas dans visites
            enfiler(attente, voisin)
            inserer(visites, voisin)
```

■ **Exemple 11** Voici un algorithme qui permet de déterminer s'il y a un cycle dans un graphe non orienté : le graphe est donné par ses listes d'adjacences. Cette fois, on sauvegarde aussi la distance des sommets visités à l'origine. Ainsi, si on atteint un même sommet à partir de deux sommets qui sont plus proches de l'origine, on a trouvé un cycle.

```
from collections import deque #pour les files
def cycle(G:dict, origine) -> bool:
    '''
    Entrée : G est un graphe connexe donné par le dictionnaire
    de ses listes d'adjacences,
    origine est un sommet de G.
    Sortie : renvoie True s'il y a un cycle dans le graphe et False sinon
    '''
    distances = {s: None for s in G}
    distances[origine] = 0
    attente = deque()
    attente.append(origine)
    while
        sommet =
        for voisin in G[sommet]:
            if distances[voisin] == None:


            elif distances[voisin] >= distances[sommet]:
                return True
    return False
```

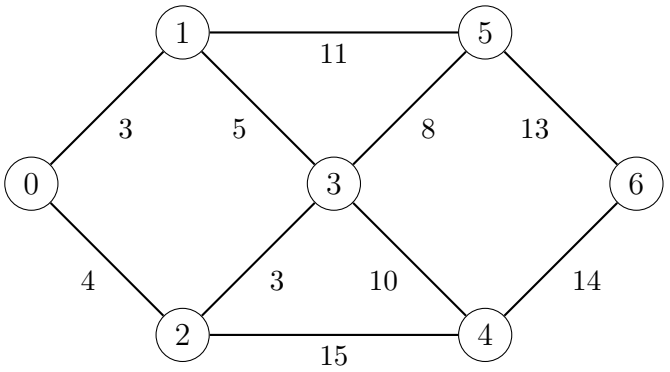
■

II.3 Algorithme de Dijkstra

II.3.1 Principe et exemple

L'algorithme de Dijkstra permet de trouver le chemin le plus court entre deux sommets d'un graphe non orienté pondéré lorsque tous les poids sont positifs. Il suit le principe d'un parcours en largeur, mais au lieu de prendre le sommet suivant dans la file, on traite d'abord celui dont la distance à l'origine est la plus petite. Ainsi, une fois qu'un sommet est traité, sa distance à l'origine ne peut plus diminuer (car les poids sont positifs).
Dit autrement : si pour aller de A à B le chemin le plus court passe par I, alors la partie du chemin entre A et I est le plus court chemin entre A et I et la partie du chemin entre B et I est le plus court chemin entre B et I. On optimise donc le choix à chaque étape : c'est un algorithme glouton.

■ **Exemple 12** Appliquons l'algorithme sur un exemple :



On peut regrouper les résultats successifs dans un tableau :

Étape	0	1	2	3	4	5	6
0	0	∞	∞	∞	∞	∞	∞
1							
2							
3							
4							
5							
6							

■

II.3.2 Le pseudo-code

On utilise une variante des files : les files de priorité. Les défilements des éléments ne sont plus décidés par leur ordre d'enfilement, mais par un entier naturel associé : leur priorité.
Pour nous, cette priorité est la distance du sommet à l'origine : on veut traiter en priorité le sommet restant dont la distance à l'origine est la plus faible. On peut aussi mettre à jour la priorité des éléments au fur et à mesure de l'avancement.

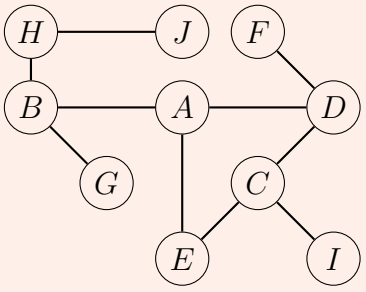
```
#Entrée : graphe (listes d'adjacence avec poids), origine
#Sortie : renvoie le dictionnaire (sommet, distance à l'origine)
distances <- dictionnaire (sommet, infini)
d[origine] = 0
attente <- [(origine, 0)] #file de priorité
tant que attente est non vide
    sommet <- defiler_valeur_min(attente)
    pour tout (voisin, poids) dans graphe[sommet]
        poids_voisin <- distances[voisin]
        si distances[sommet] + poids < poids_voisin
            distances[voisin] = distances[sommet] + poids
            si poids_voisin = infini
                enfiler(attente, (voisin,distances[voisin]))
            sinon
                changer_priorite(attente, voisin, distances[voisin])
retourner distances
```

Exercices

Exercice 1 — Des matrices d'adjacence. Dessiner les graphes associés aux matrices d'adjacence :

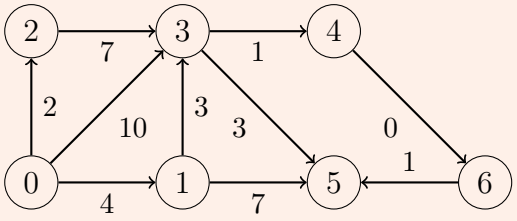
1. Non orienté : $\begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}$
2. Orienté : $\begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$

Exercice 2 — Un graphe non orienté. Voici un graphe non orienté :



1. Donner les trois représentations du graphe vues dans le cours.
2. Réaliser à la main un parcours en profondeur.
3. Réaliser à la main un parcours en largeur.

Exercice 3 — Un graphe orienté. Voici un graphe orienté pondéré :



1. Combien le graphe a-t-il de sommets et d'arcs ?
2. Donner les trois représentations du graphe vues dans le cours.
3. Réaliser à la mains un parcours en profondeur depuis le sommet 0 en donnant la priorité parmi les voisins d'un sommet aux petits numéros d'abord.
4. Réaliser à la mains un parcours en largeur depuis le sommet 0 en donnant la priorité parmi les voisins d'un sommet aux petits numéros d'abord.
5. Déterminer un chemin de poids minimal depuis le sommet 0 vers chaque autre sommet.

Exercice 4

1. On considère l'algorithme suivant écrit en pseudo-code :

Algorithme 1 : Mystère

```
Entrée :
    P : pile d'entiers contenant au moins deux éléments ;
Sortie :
    None ;
Déclarer :
    aux1, aux2 : entiers ;
Début :
    aux1 = depiler(P)
    aux2 = depiler(P)
    empiler(P, aux1)
    empiler(P, aux2)
Fin
```

   (a) Exécuter cet algorithme avec la pile P = (1, 2, 3, 4), où 1 est le sommet de la pile.

   (b) Que fait cet algorithme ?

2. On considère l'algorithme suivant écrit en pseudo-code :

Algorithme 2 : Mystère

```
Entrée :
    P : pile d'entiers rangée par ordre croissant du sommet à la base ;
    x : entier ;
Sortie :
    None ;
Déclarer :
    Buff : pile vide ;
    continuer : booléen ;
    y : entier ;
Début :
    continuer = True
    tantque continuer et (P n'est pas vide) faire
        y = depiler(P)
        si y ≥ x alors
            empiler(P, y)
            continuer = False
        sinon
            empiler(Buff, y)
        fin si
    fin tantque
    empiler(P, x)
    tantque Buff n'est pas vide faire
        y = depiler(Buff)
        empiler(P, y)
    fin tantque
Fin
```

   (a) Exécuter cet algorithme avec la pile P = (1, 2, 3, 5) et l'entier x = 4.

   (b) Que fait cet algorithme ?