Lycée Pierre-Gilles de Gennes 2024-2025

TP5 Python

Graphes

1 Qu'est-ce qu'un graphe ?

Si l'on dispose de n objets, un graphe (dans sa version la plus simple) G sur ces n objets est une structure reflétant le fait que deux de ces objets sont reliés ou pas.
La structure de graphe est une des structures les plus fondamentales de l'informatique. Elle est très utilisée par exemple en bio-informatique dans l'analyse de génomes.
On rencontre des graphes de manière évidente et explicite lorsque l'on traite de réseaux routiers, de réseaux électriques mais aussi de manière cachée et implicite lorsque l'on traite de jeux. Dans ce dernier cas, les configurations du jeu sont les sommets du graphe et on relie un sommet à un autre si les règles du jeu permettent de passer de la première configuration à l'autre.

1.1 Graphe non orienté

Définition .1

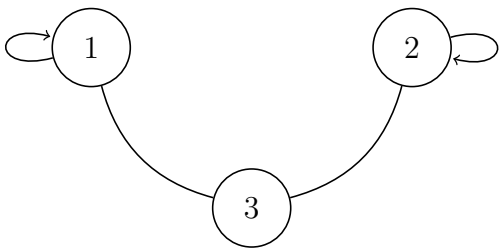
Un *graphe non orienté* G est la donnée d'un ensemble de *sommets* reliés par des *arêtes*.
On note ainsi $G = (S, A)$ où S est l'ensemble des sommets et A l'ensemble des arêtes, représentées par une paire de sommets.

Exemple :

$G = (S, A)$ avec :

$$S = \{1, 2, 3\}$$

$$A = \{\{1\}, \{1, 3\}, \{2, 3\}, \{2\}\}$$



1.2 Graphe orienté

Définition .2

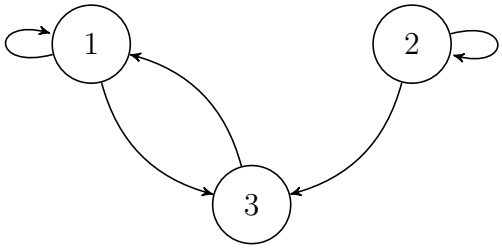
Un *graphe orienté* G est la donnée d'un ensemble de *sommets* reliés par des *arcs orientés*.
On note ainsi $G = (S, A)$ où S est l'ensemble des sommets et A l'ensemble des arêtes, représentées par un couple de sommets.

Exemple :

$G = (S, A)$ avec :

$$S = \{1, 2, 3\}$$

$$A = \{(1,1), (1,3), (2,2), (2,3), (3,1)\}$$

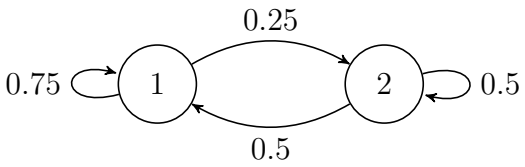


1.3 Graphe pondéré

Définition .3

Un graphe *pondéré* est un graphe (orienté ou non) dans lequel chaque arête (ou arc orienté) porte un nombre appelé un *poids*.

Exemple :



2 Représentation des graphes en Python

On considère dans cette partie un **graphe orienté possédant n sommets numérotées numérotés de 0 à $n-1$.**
Il existe deux approches principales pour représenter un graphe en Python : par une matrice d'adjacence ou par une liste d'adjacence.
Il est à noté que tout ce qui va suivre s'applique également aux graphes non-orienté : il suffit de considérer un graphe non orienté comme un graphe orienté « symétrique » (s'il y a un arc de x vers y alors il y a un arc de y vers x).

2.1 Représentation par matrice d'adjacence

Définition .4

La *matrice d'adjacence* de G est la matrice $M = (m_{ij})_{i,j \in [\![0,n-1]\!]}$ définie par :

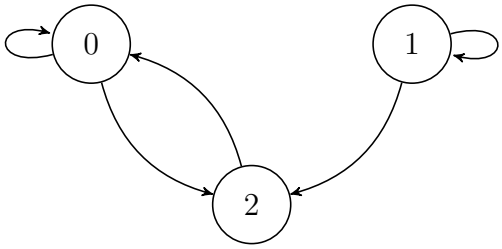
$$\forall (i,j) \in [\![0, n-1]\!]^2, \quad m_{ij} = \begin{cases} 1 & \text{si un arc relie le sommet i au sommet j} \\ 0 & \text{sinon.} \end{cases}$$

Si le graphe est pondérée, on remplace les coefficients par la valeur du poids de l'arc correspondant.

Exemple :

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

En python on peut implémenter M comme une liste de liste ou en tableau numpy avec la commande `np.array`.



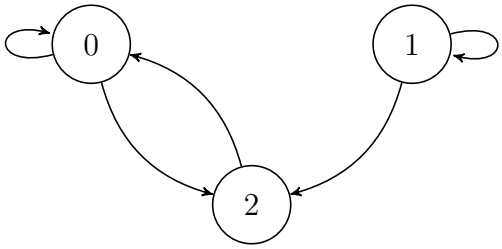
2.2 Représentation par liste d'adjacence

Définition .5

La *la liste d'adjacence* associée à G est la liste L à n éléments telle que pour tout $i \in [\![0, n-1]\!]$, $L[i]$ contient la liste des sommets j pour lesquels un arc va de i à j.

Exemple :

$L = [[0, 2], [1, 2], [0]]$



2.3 La structure de dictionnaire

La structure de dictionnaire est une structure utile pour implémenter les graphes.

Définition .6 (Dictionnaire)

Un dictionnaire est une liste d'objets de type quelconque dont l'indiçage se fait à l'aide d'identifiants appelés *clés*.
Contrairement aux listes qui sont délimitées par des crochets, on utilise des accolades pour les dictionnaires.

- un dictionnaire vide se déclare en utilisant la commande `D={}` ;
- on ajoute un élément a `D` en utilisant la commande :

```
D[cle]=valeur
```

- on peut récupérer l'ensemble des clés avec la commande `D.keys()` et il s'agit d'un objet sur lequel on peut boucler.

Exemple : on veut créer un dictionnaire récapitulant le nombre de pattes de différents animaux :

`Pattes = {"Chat" : 4 , "Poule" : 2 , "Boa" : 0}`.

La commande

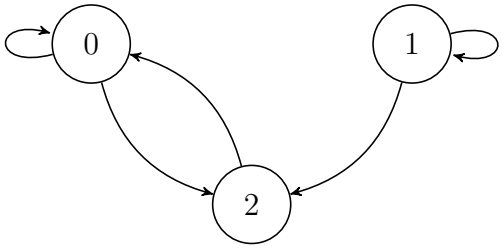
- `Pattes["Chat"]` renvoie 4 (le nombre de pattes d'un chat en bonne santé) ;
- `Pattes["Lapin"]=4` permet d'ajouter l'entrée `"Lapin"` au dictionnaire en spécifiant le nombre de pattes,
- `Pattes.keys()` renvoie l'ensemble des clés (animaux) présents dans le dictionnaire,
- `for k in Pattes.keys(): print("un",k,"possède", Pattes[k], "patte(s)")` affiche :

```
un Chat possède 4 patte(s)
un Poule possède 2 patte(s)
un Boa possède 0 patte(s)
un Lapin possède 4 patte(s)
```

Dans le contexte des graphes, on peut utiliser les dictionnaires pour implémenter la liste d'adjacence : les clés seront les sommets et la valeur associée à une clé la liste des sommets adjacents à la clé.

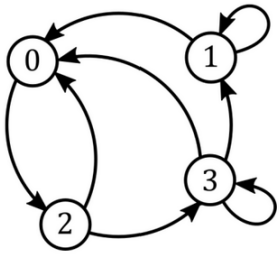
Exemple :

$L = \{0 : [0, 2], 1 : [1, 2], 2 : [0]\}$



2.4 Exercice

Exercice 1. Soit G le graphe orienté représenté par la figure ci-dessous :



1. Créer la matrice d'adjacence de G à l'aide d'un tableau numpy.
2. Créer le dictionnaire associé à G.

Exercice 2.

1. Soit G le graphe orienté dont la matrice d'adjacence est $\begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}$.

   Donner la liste d'adjacence sous forme de dictionnaire.
2. Soit G le graphe orienté associé au dictionnaire :

$$\{0 : [2], 1 : [1, 2, 3], 2 : [], 3 : [0, 1]\}.$$

   Donner la matrice d'adjacence.

Exercice 3. Soit d et n deux entiers naturels non nuls tels que d divise n. On définit $G_{d,n}$ le graphe non-orienté de sommets $0,\ldots, n-1$ tel que pour tout $i, j \in [\![0, n-1]\!]$ les sommets i et j sont reliés par une arête si et seulement si $i \neq j$ et d divise $j - i$.

1. Dessiner le graphe correspondant à $n = 6$ et $d = 3$.
2. Écrire une fonction $G(d, n)$ qui renvoie la matrice d'adjacence de $G_{d,n}$.

3 Parcours d'un graphe en largeur

Soit G un graphe non-orienté. On s'intéresse aux questions suivantes :

1. Est-ce que deux sommets donnés sont reliés par un chemin ?
2. Le graphe est-il connexe (c'est-à-dire deux sommets sont-ils toujours joignables par un chemin d'arêtes) ?
3. Quels sommets sont joignables à partir d'un sommet donné ?

Une façon de répondre à ces questions est de parcourir le graphe. Pour ce faire, il existe plusieurs algorithmes dont celui du *parcours en largeur.*

Principe général : en partant du sommet de départ (appelé x), on commence par explorer tous ses voisins, puis les voisins de ces voisins etc.

Implémentation :

— Les sommets déjà explorés / à explorer sont recensés dans une liste L qui se comporte comme une file d'attente. On examine les sommets en tête de liste les uns après les autres en regardant leurs voisins : à chaque fois que l'on découvre un nouveau sommets, on l'ajoute en queue de liste.

— On initialise $L = \{x\}$ et une variable $i = 0$.

  A tout instant du processus, i désignera la position de la « tête de lecture » c'est-à-dire l'indice de la liste L du sommet dont on est en train de recenser les voisins de sorte que :

  — les sommets d'indices 0 à $i - 1$ ont déjà été examinés ;

  — les sommets d'indices $> i$ sont les sommets en attentes dans L qui seront examinés après.

  — Pour chaque sommet examiné, on recense tous ses voisins et on ajoute à L, en queue de liste, tous ceux qui n'ont pas déjà été explorés.

  — Le processus s'arrête dès que l'on a exploré tous les sommets possibles.

Travail demandé

1. Écrire une fonction `GrapheAleatoire(n)` qui prend en entrées un entier $n \in \mathbb{N}^*$ et renvoie un graphe non orienté à n sommets sous forme d'une matrice d'adjacence créée aléatoirement
2. Écrire une fonction `Parcours(G,x)` qui prend en entrées un graphe non orienté G et un sommet x et renvoie la liste des sommets accessibles depuis x.
3. Tester ce programme avec des graphes générés aléatoirement et avec le graphe dont la matrice est :
$$M = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$
4. Tester le programme avec les graphes $G_{d,n}$ pour différentes valeurs de n et d.
5. Écrire une fonction `Connexe(G)` qui prend en entrées un graphe non orienté G et renvoie `True` si le graphe est connexe et `False` sinon.

4 Marche aléatoire sur un graphe

Une marche aléatoire sur un graphe orienté consiste à partir d'un sommet donné, puis à chaque étape, à se déplacer sur un sommet adjacent choisi au hasard (uniformément pour l'instant).
Par commodité on supposera qu'il y a toujours au moins un arc partant de chaque sommet.

Travail demandé

1. (a) Écrire une fonction `position(G,x,n)` qui :
   - prend en entrée un sommet x d'un graphe G, donné sous forme de dictionnaire ;
   - simule n étapes d'une marche aléatoire sur G partant de x ;
   - renvoie la position atteinte à l'issue de ces n déplacements.

   Commande utile : si `L` est une liste, `choice(L)` renvoie un élément de `L` choisi au hasard. Importez cette commande en tapant `from random import choice`.

   (b) Adapter la fonction précédente pour créer une fonction `trajet(G,x,n)` qui simule n en partant de x, mais renvoie cette fois le trajet complet effectué (sous forme de liste des sommets parcourus). Testez votre fonction sur différents exemples !
2. Bidule le hamster mène une vie simple (mais pleinement satisfaisante) : il partage son temps entre ses trois activités favorites : dormir, manger et faire de la roue. Au début de la journée, il commence toujours par manger ; puis, à chaque heure, il change d'activité selon les critères suivants.
   - Si, à l'heure n, il est en train de dormir, alors à l'heure suivante il peut adopter n'importe laquelle de ses trois occupations préférées (de façon équiprobable).

— Si, à l'heure n, il est en train de manger, alors à l'heure suivante il va dormir ou aller se dépenser dans la roue (de façon équiprobable).
— Si, à l'heure n, il est en train de faire de la roue, alors à l'heure suivante il va récupérer de cet effort soit en dormant, soit en mangeant (de façon équiprobable).

On s'intéresse ici à l'évolution du comportement de Bidule. On souhaite notamment déterminer s'il a tendance à passer, en moyenne, plus de temps sur certaines activités que d'autres.

(a) Représenter le comportement de Bidule par un graphe (on notera "0" l'état où il dort, "1" celui où il mange et "2" celui où il fait de la roue). On indiquera par une flèche chacune des transitions possibles, sans chercher à pondérer ces flèches. Stockez ce graphe, sous forme de dictionnaire, dans une variable `hamsterG`.

(b) Adapter la fonction de la question 1a pour créer une fonction `premierDodo()` qui simule le comportement du hamster et renvoie le nombre d'heures qui se sont écoulées lorsqu'il s'endort pour la première fois.

(c) Écrire une fonction `frequences(n)` qui simule le comportement de Bidule pendant n heures, et renvoie une liste $[a, b, c]$ contenant les fréquences auxquelles il s'est livré à chacune de ses trois occupations favorites.

5 Algorithme de Dijkstra

On considère un graphe **non orienté** et **pondéré** avec des poids positifs qui représentent la distance entre deux sommets (on peut penser à un réseau ferroviaire où les sommets représentent des villes et les arêtes représentent la présence d'une ligne de chemin de fer). On supposera qu'entre deux sommets il y **au plus** une arête.
Un tel graphe sera représenté par sa matrice de poids $M = (m_{ij})_{i,j\in[\![0,n-1]\!]}$ définie par :

$$\forall (i,j) \in [\![0,n-1]\!]^2, \quad m_{ij} = \begin{cases} 0 & \text{si } i = j \\ \text{poids de l'arc } (i,j) & \text{si un arc relie le sommet i au sommet j} \\ +\infty & \text{sinon.} \end{cases}$$

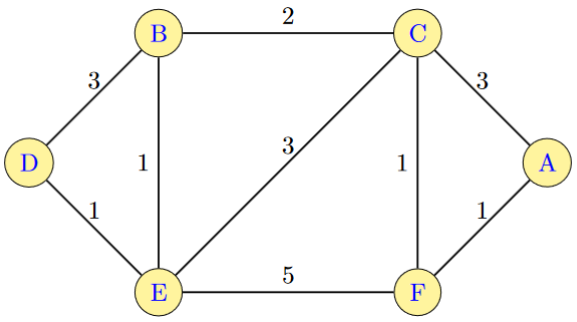
On s'intéresse au problème du plus court chemin dans un tel graphe : à savoir, déterminer la longueur du plus court chemin menant d'un sommet s à chaque autre sommet du graphe.
L'algorithme de Dijkstra permet de résoudre ce problème.
On va décrire cet algorithme en illustrant son utilisation sur le graphe ci-dessous :

1. **Initialisation :** on initialise une liste `d` dont la taille est le nombre de sommet de la façon suivante :

   $d[s] = 0$; $d[v] =$ poids de l'arc (s, v) s'il y a un arc de s à v ; $d[v] = +\infty$ sinon.

   Avec le graphe exemple on obtient en partant de $s = D$:

D	B	E	C	F	A
0	3	1	$+\infty$	$+\infty$	$+\infty$



2. **Principe de relâchement.**
   L'opération de relâchement d'un arc (u, v) consiste à savoir s'il est possible, en passant par u, d'améliorer le plus court chemin jusqu'à v : si oui, il faut mettre à jour la valeur de $d[v]$. Plus précisément, on effectue le test suivant :

$$d[u] + \text{"poids de } (u,v)\text{"} \overset{?}{<} d[v].$$

   Si le test est négatif, c'est que passer par u ne raccourcit pas le chemin actuel et on ne modifie donc pas $d[v]$.
   Si le test est positif cela signifie qu'on a trouvé un chemin plus court que le précédent pour aller de s à v, à savoir :
   - on va de s à u par le plus court chemin connu à ce jour ;
   - on passe de u à v par l'arc (u, v).

   On met donc à jour la valeur de $d[v]$:

$$d[v] = d[u] + \text{"poids de } (u,v)\text{"}.$$

   Choix de u : à chaque étape, on choisit le sommet u qui vérifie les propriétés suivantes :
   - u n'a pas encore été sélectionné,
   - $d[u]$ est le plus faible parmi tous les sommets non encore sélectionnés.
3. **Fin de l'algorithme :** une fois tous les sommets visités, on trouvé tous les plus courts chemin de s à v .

On reprend le graphe donné en exemple en partant de $s = D$. Les étapes de l'algorithme sont les suivantes :

— **Initialisation** :

D	B	E	C	F	A
0	3	1	$+\infty$	$+\infty$	$+\infty$

— **Étape 1** : on prend $u = E$.

D	B	E	C	F	A
0	2	1	4	6	$+\infty$

— **Étape 2** : on prend $u = B$.

D	B	E	C	F	A
0	2	1	4	6	$+\infty$

— **Étape 3** : on prend $u = C$.

D	B	E	C	F	A
0	2	1	4	5	7

— **Étape 4** : on prend $u = F$.

D	B	E	C	F	A
0	2	1	4	5	6

— **Étape 4** : on prend $u = A$.

D	B	E	C	F	A
0	2	1	4	5	6

— **Fin :** on a visité tous les sommets l'algorithme est fini.

Travail demandé

1. Effectuer l'algorithme à la main avec le même graphe mais en partant du sommet $s = E$.
2. Programmer une fonction `Dijkstra`(G,s) qui prend en entrée un graphe pondéré décrit par sa matrice de poids et un sommet de départ et renvoie la liste des longueurs plus courts chemins partant de s.
   Commande utile : `np.inf` pour $+\infty$.