TP 19
Graphes

Les *graphes* sont des structures de données (comme les listes, tableaux, dictionnaires, ...) énormément utilisées en informatique et en mathématiques car ils sont assez simples à manipuler et modélisent de très nombreuses situations.

I Notion de graphe

Un graphe est tout simplement donné par un ensemble de sommets, reliés entre eux par des arêtes :

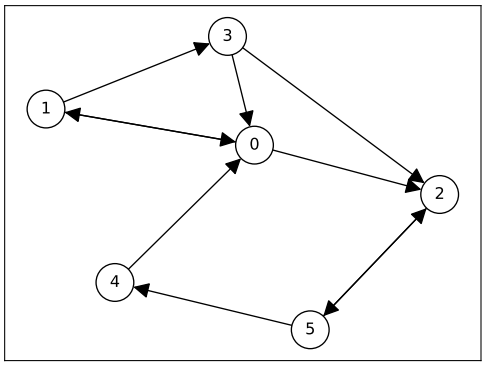

FIGURE 1 – Un graphe

Ce genre de dessin modélise de très nombreuses situations différentes :

1. Une carte géographique, où les sommets sont des villes et les arêtes sont des routes reliant ces villes. La carte du métro parisien est assurément un graphe, de même que la carte des lignes de train en France.
2. Un réseau social, où les sommets sont des personnes et une arête entre deux personnes signifie que ces personnes sont amies, ou bien (suivant le sens de la flèche) que l'un est un *follower* de l'autre. Les gros réseaux sociaux ont absolument besoin d'algorithmes efficaces opérant sur des graphes avec des millions d'informations.
3. Un jeu de stratégie, où chaque sommet représente un état possible du jeu, et une arête représente une façon de passer d'un état à un autre. Jouer une partie revient à démarrer sur un sommet initial puis passer d'un sommet à un autre via des arêtes, jusqu'à arriver sur un sommet représentant une partie gagnée. Un jeu de labyrinthe peut se représenter par un graphe dont le but est d'arriver au sommet final, en partant d'un sommet initial donné.

La formalisation mathématique est celle-ci, prenant en compte la petite flèche dessinée sur les arêtes qui correspond à une orientation :

Définition 1. Un **graphe orienté** est la donné d'un ensemble fini S (ensemble des **sommets**) et d'un sous-ensemble $A \subset S \times S$ (ensemble des **arêtes**). Si x, y sont deux sommets, la condition $(x, y) \in A$ signifie qu'il y a une arête de x vers y.

Une autre variante de graphe est celle où on ne s'occupe pas du sens des arêtes, qu'on représente donc comme un simple trait entre sommets : deux sommets x et y sont ou bien ne sont pas connectés, peu importe dans quel ordre.

Définition 2. Un **graphe non orienté** est un graphe $G = (S, A)$ vérifiant la condition :

$$\forall (x, y) \in S^2, \quad (x, y) \in A \iff (y, x) \in A \tag{1}$$

Autrement dit on interprète la condition $(x, y) \in A$ comme signifiant qu'il existe une arête entre les sommets x et y, et ceci est équivalent à dire qu'il existe une arête entre les sommets y et x.

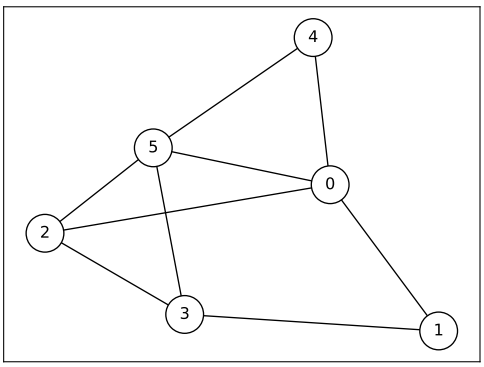

FIGURE 2 – Un graphe non orienté, représenté sans flèches

II Représentation informatique des graphes

Il y a plusieurs façons de représenter un graphe orienté $G = (S, A)$ en Python. On suppose qu'on a d'abord numéroté les N sommets de 0 à $N-1$, ainsi $S = \{0, 1, \ldots, N-1\}$.

1. Par **liste d'adjacence** : on donne une liste `L` où pour chaque indice de sommet `i`, `L[i]` est la **liste** des sommets auxquels mène une arête issue de `i`. Mathématiquement c'est la liste des $\{ j \in S \mid (i,j) \in A \}$.
2. Par **matrice d'adjacence** : on donne une liste de listes `M` représentant une matrice carrée, où `M[i][j]` vaut 1 s'il y a une arête du sommet i vers le sommet j et 0 sinon.

Prenons par exemple le graphe très simple

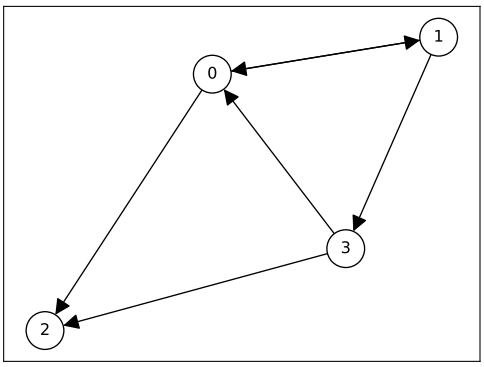

FIGURE 3 – Un exemple

Alors vérifiez que la liste d'adjacence est

```
L = [[1, 2], [0, 3], [], [0, 2]]
```

et que la matrice d'adjacence est

```
M = [[0, 1, 1, 0], [1, 0, 0, 1], [0, 0, 0, 0], [1, 0, 1, 0]]
```

3. Si on ne souhaite pas numéroter les sommets, alors on peut leur donner un nom et travailler comme en 1 mais avec à la place des **dictionnaires d'adjacence** : un dictionnaire où chaque clé est un sommet et la valeur correspondante est la liste des sommets auquel il est relié. Voici par exemple un bout de graphe représentant des lignes de train en France :

```
trains = {"Paris": ["Angers", "Le Havre", "Troyes", "Lyon"], "Angers": ["Paris"],
    "Le Havre": ["Paris"], "Troyes": ["Paris"], "Lyon": ["Paris", "Grenoble", "Marseille"],
    "Grenoble": ["Lyon"], "Marseille": ["Lyon", "Toulon"], "Toulon": ["Marseille"]}
```

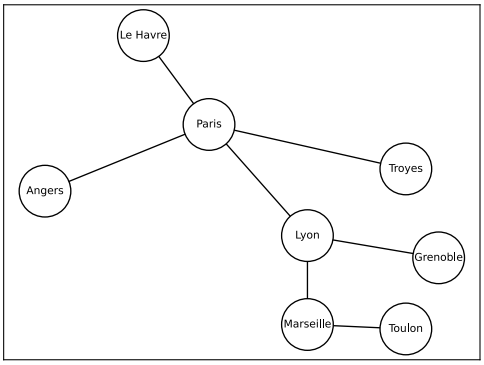

FIGURE 4 – Carte de France...

En fait, chaque représentation possible a ses avantages et ses inconvénients...

Exercice 1. Écrire des fonctions

1. `liste_vers_matrice(L)` : convertit un graphe représenté par une liste d'adjacence, vers une matrice d'adjacence. On aura besoin de savoir créer une matrice nulle, et d'une double boucle pour itérer sur les listes d'adjacence de chaque sommet.
2. `matrice_vers_liste(M)` : convertit un graphe représenté par une matrice d'adjacence, vers une liste d'adjacence. On aura besoin de créer une liste nulle, puis d'une double boucle pour parcourir la matrice d'adjacence et on utilisera `append` pour ajouter au fur et à mesure les sommets dans la liste d'adjacence.

Exercice 2. Écrire une fonction `est_non_oriente(M)` prenant en argument une matrice d'adjacence `M` et qui renvoie `True` si cela représente bien un graphe non orienté, et `False` sinon.

III Chemins

Un thème important est d'étudier les chemins dans un graphe, en passant d'un sommet au suivant via une arête.

Définition 3. Soit $G = (S, A)$ un graphe. Soient $x, y \in S$ deux sommets. Un **chemin** dans G de x à y est la donnée d'une suite de sommets $s_0, \dots, s_n$ tels que :

1. $s_0 = x$,
2. $s_n = y$,
3. $\forall i \in [\![0, n-1]\!]$, $(s_i, s_{i+1}) \in A$.

Autrement dit il s'agit de passer de sommets en sommets en passant à chaque fois sur une arête, partant de x pour arriver à y. Ici le nombre $n-1$ est la **longueur** du chemin ($n-1$ est le nombre d'arêtes parcourues).

Définition 4. Une **boucle** est une arête d'un sommet à lui-même, (x, x). Plus généralement un **cycle** est un chemin partant d'un sommet et arrivant à lui-même.

Exercice 3. On représente un chemin par une liste `C` des sommets à parcourir, dans l'ordre, dans un graphe représenté par une matrice d'adjacence `M`. Écrire une fonction `est_chemin_possible(C, M)` qui renvoie `True` si ce chemin est bien possible (c'est à dire s'il existe bien une arête de `C[i]` à `C[i+1]`, pour tout i) et `False` sinon.

Tester avec l'exemple de la figure 5 en écrivant sa matrice d'adjacence. Vérifier que le chemin (qui est un cycle) $(0, 2, 3, 4, 3, 1, 0)$ est bien possible, mais que $(0, 2, 3, 2, 0)$ ne l'est pas.

Exercice 4 (Mathématiques)**.** Soit G un graphe orienté et soit M sa matrice d'adjacence. Montrer par récurrence que pour tout $p \in \mathbb{N}$ le coefficient (i, j) de M^p donne le nombre de chemins de longueur exactement p reliant le sommet i au sommet j.
On se concentrera sur le cas $n = 2$. On peut en fait démarrer avec $p = 0$, en considérant qu'il existe toujours un chemin de longueur 0 d'un sommet à lui-même.

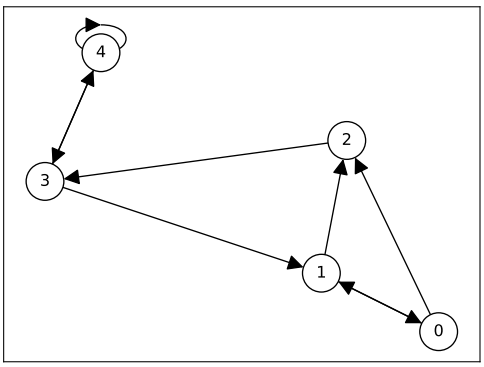

FIGURE 5 – Exemple avec une boucle sur le sommet 4

IV Pondération

Dans de nombreuses situations on veut attacher plus d'information sur les graphes. Par exemple sur le graphe représentant des lignes de train, on veut attacher sur chaque arête un nombre qui indique la distance entre les deux villes. La distance totale associée à un chemin est alors la somme des distances sur les arêtes par lesquelles passe le chemin. Un exemple d'une question extrêmement importante est de trouver le plus court chemin entre deux points.

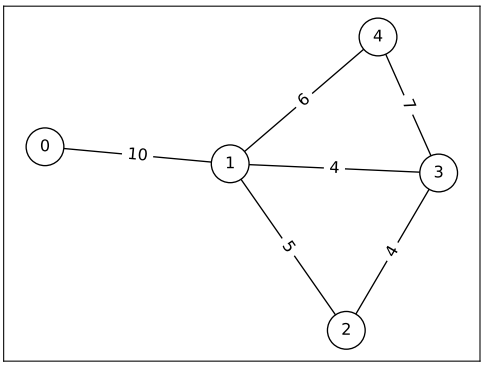

FIGURE 6 – Un graphe pondéré

Une façon simple de représenter cette situation en Python est de numéroter les sommets, puis d'utiliser une modification de la matrice d'adjacence où `M[i][j]` est tout simplement égal à la distance entre les sommets i et j, ou bien 0 si les deux sommets ne sont pas reliés. Attention alors dans les programmes à bien tester si le coefficient est nul ou non ! Une valeur mathématiquement cohérente serait plutôt $+\infty$ (deux points à distance 0 sont collés, alors que deux points à distance $+\infty$ ne sont pas reliés du tout).

Exercice 5. Écrire une fonction `longueur(C, M)` prenant en argument un chemin comme dans la partie précédente (une liste de sommets parcourus dans l'ordre) et une matrice d'adjacence, et qui calcule la longueur totale du chemin.

On pourra tester la fonction en écrivant la matrice du graphe pondéré ci-dessus (figure 6), et donner la longueur du chemin $(0, 1, 3, 4, 1, 2)$.

V Parcours aléatoire

Exercice 6. Le chat passe son temps entre 4 activités : dormir sur le canapé (C), manger (M), sortir (S), et… dormir sur le lit (L). Son parcours d'une journée typique est le suivant :

— S'il dort sur le canapé, alors ensuite il peut aller manger ou sortir ou bien passer sur le lit.

— S'il mange, alors ensuite il peut sortir ou bien dormir sur le lit.

— S'il sort, alors ensuite il peut dormir sur le canapé ou le lit ou bien manger.

— S'il dort sur le lit, alors il peut rester sur le lit ou bien passer sur le canapé.

On représente cette situation par un graphe orienté à 4 sommets, représentant les 4 activités possibles du chat, et des flèches indiquant qu'il peut passer d'une activité à l'autre.

1. Dessiner sur feuille le graphe correspondant.
2. Écrire en Python le dictionnaire d'adjacence correspondant.

On souhaite alors représenter une journée du chat par un parcours aléatoire de ces activités. On souhaite d'abord raisonner dans un cadre très général, avec un graphe quelconque représenté par un dictionnaire d'adjacence.

3. Écrire une fonction `suivant(G, x)` qui prend en argument un graphe représenté par un dictionnaire d'adjacence, et un sommet `x`, et qui choisit un sommet au hasard parmi ceux issus de `x`.
   Pour choisir une élément au hasard dans la liste `G[x]`, on pourra utiliser la bibliothèque `random` et sa fonction `randint(a, b)` donner un nombre aléatoire entre deux bornes a et b (bornes incluses, pas comme dans `range`!!!)
4. Écrire une fonction `parcours(G, e, n)` prenant en argument un graphe G comme ci-dessus, un sommet de départ e, et un entier $n \geqslant 0$, et saute aléatoirement d'un sommet au suivant n fois de suite, en affichant à chaque fois le sommet sur lequel elle se trouve.
5. Tester la fonction pour afficher une journée typique du chat, en choisissant e et n.
6. Tester pour des grandes valeurs de n la proportion du temps que le chat passe sur le lit. Vérifier que ces proportions semblent converger et sont indépendantes de l'activité de départ.

VI Annexe : une bibliothèque pour manipuler des graphes Python

La bibliothèque `networkx` permet de travailler avec des graphes, notamment pour les dessiner. Elle s'utilise conjointement avec `matplotlib` :

```
import matplotlib.pyplot as plt
import networkx as nx
```

Un graphe orienté est créé par `nx.DiGraph()`. On peut lui passer comme argument :

1. Directement un dictionnaire d'adjacence. Par exemple pour le graphe très simple figure 3 :

   ```
   d = {0: [1, 2], 1: [0, 3], 2: [], 3: [0, 2]}
   G = nx.DiGraph(d)
   ```

2. Une matrice d'adjacence, qui doit être un tableau `numpy`, sans oublier le `import numpy as np` :

   ```
   M = np.array([[0, 1, 1, 0], [1, 0, 0, 1], [0, 0, 0, 0], [1, 0, 1, 0]])
   G = nx.DiGraph(M)
   ```

3. Une *liste* de *couples* (x, y) représentant des arêtes :

   ```
   G = nx.DiGraph([(0, 1), (0, 2), (1, 0), (1, 3), (3, 0), (3, 2)])
   ```

Pour les graphes non-orientés, c'est la fonction `nx.Graph()`.
On peut ensuite demander à tracer le graphe puis à l'afficher. La figure 3 a été obtenue ainsi :

```
d = {0: [1, 2], 1: [0, 3], 2: [], 3: [0, 2]}
G = nx.DiGraph(d)
nx.draw_networkx(G, arrowsize=25, node_size=800, font_size=12,
    node_color="white", edgecolors="black")
plt.show()
```

Le carte de France a été tracée par (et un petit ingrédient secret)

```
trains = {"Paris": ["Angers", "Le Havre", "Troyes", "Lyon"], "Angers": ["Paris"],
    "Le Havre": ["Paris"], "Troyes": ["Paris"], "Lyon": ["Paris", "Grenoble", "Marseille"],
    "Grenoble": ["Lyon"], "Marseille": ["Lyon", "Toulon"], "Toulon": ["Marseille"]}
G = nx.Graph(trains)
nx.draw_networkx(G, node_size=1500, font_size=8, node_color="white", edgecolors="black")
plt.show()
```

La bibliothèque contient de nombreuses fonctions pour travailler sur les graphes. Remarquez d'ailleurs que même la question de dessiner un graphe donné abstraitement n'est pas si évidente : si on place d'abord les sommets n'importe comment, les arêtes risquent fort de se croiser... Peut-on éviter, ou du moins minimiser, les croisements ? C'est très compliqué ! La bibliothèque contient notamment des algorithmes pour dessiner le graphe dans le plan le plus élégamment possible.
Documentation : https://networkx.org/documentation/stable/tutorial.html
Plus spécifiquement pour le dessin : https://networkx.org/documentation/stable/reference/generated/networkx.drawing.nx_pylab.draw_networkx.html