TD8
Algorithmes de Graphes avec Python

Soit un graphe $G = (X, U)$ d'ordre $n = |X|$ où l'on suppose, sans perte de généralité, que les éléments de X sont indicés de 0 à $n-1$. On rappelle que la matrice d'adjacence $A = (a_{ij})_{0 \leq i,j < n}$ de G est la matrice carrée d'ordre n telle que, pour tous indices i et j (pour $0 \leq i, j < n$) :

$$a_{ij} = \begin{cases} 1 & \text{si et seulement si } (i,j) \in U \\ 0 & \text{sinon} \end{cases}$$

En règle générale, lorsque le graphe est orienté, un élément de U est un *arc*. Si la matrice d'adjacence A est symétrique, alors le graphe est dit *non-orienté*. Dans ce cas, on peut considérer que U n'est constitué que d'*arêtes* en contractant chaque paire d'arcs opposés (i,j) et (j,i) en une arête $\{i,j\}$. La matrice d'adjacence est alors telle que pour toute arête $\{i,j\} \in U$, $a_{ij} = a_{ji} = 1$.

Exercice 1 : Modélisation d'un graphe

1. Créer une classe `Graphe` de façon à ce qu'un objet de type `Graphe` ait pour attributs :

   - un entier `nb_sommets` représentant le nombre d'éléments n du graphe,
   - un dictionnaire `arcs` qui, à un élément i de X associe la liste de ses successeurs dans le graphe, c'est-à-dire l'ensemble des sommets j pour lesquels il existe (i,j) dans U,
   - la matrice d'adjacence `mat_adj` du graphe.

   Dans cette classe, on crée en particulier trois méthodes :

   - la méthode "constructeur" `def __init__(self, n, L=[])` permettant d'instancier un graphe, ayant pour arguments le nombre d'éléments `n` du graphe et une liste `L` de tuples représentant ses arcs (par défaut cette liste sera vide)
   - une méthode `ajouter_arc(self, e)` mettant à jour le graphe en rajoutant le nouvel arc représenté par le tuple `e` à deux éléments
   - une méthode `enlever_arc(self, e)` mettant à jour le graphe en enlevant l'arc représenté par le tuple `e`.

   Il faudra faire en sorte que les méthodes mettent à jour à la fois le dictionnaire répertoriant les arcs et la matrice d'adjacence.

2. On veut pouvoir spécifier que le graphe est orienté et dans ce cas, rentrer directement dans les trois méthodes précédentes, des arêtes à la place des arcs. Pour cela, on va rajouter un attribut booléen `orientation`, `True` par défaut, qui dira si le graphe est orienté ou non. Si le graphe est non-orienté, on considérera que tous les tuples à deux éléments sont des arêtes. Modifier les méthodes précédentes afin d'introduire le nouvel attribut `orientation` et, dans le cas où `orientation` est `False`, considérer l'argument `L` du constructeur comme une liste d'arêtes et les arguments `e` des autres méthodes comme une arête. La matrice d'adjacence devra être construite en conséquence.

3. On veut également pouvoir modéliser un graphe valué, où des poids sont définis sur ses arcs (ou arêtes). Pour cela, lorsque l'on ajoute un arc, on peut passer au lieu d'un tuple à 2 valeurs (i, j) un tuple à 3 valeurs (i, j, p) où p est le poids de l'arc qu'on renseignera dans une matrice `poids`, avec pour éléments $p_{ij} = p$ (avec $p = 1$ par défaut). **Attention !** Dans ce cas ce n'est plus véritablement une matrice d'adjacence mais une matrice de poids. On utilise cette solution pour son aspect pratique. On remplacera donc le champ `mat_adj` de `Graphe` par le champ `poids`. Ajouter en conséquence une méthode `mat_adj(self)` renvoyant la matrice d'adjacence associée à la matrice `poids`. Ensuite, modifier les méthodes `ajouter_arc(self, e)` et `enlever_arc(self, e)` afin de tenir compte d'éventuels poids.

Exercice 2 : Parcours en profondeur d'un graphe orienté

Pour parcourir les sommets d'un graphe orienté, l'un des algorithmes classiques d'exploration est le parcours en profondeur (DFS : *Depth-First Search*).

Dans cet algorithme, chaque sommet est initialement noté non-exploré. On a pour cela une liste `etat` globale gardant en mémoire le statut (exploré ou non) de chaque sommet. L'idée est de partir d'un sommet quelconque non exploré i que l'on *ouvre* et affiche. On met alors à jour le statut de ce sommet i dans `etat` en tant qu'exploré. Puis on choisit un de ses successeurs non explorés et on répète la procédure. On *ferme* un sommet lorsque tous ses successeurs ont été explorés et fermés.

Dans un parcours DFS, la numérotation *préfixe* indique l'ordre dans lequel les sommets ont été ouverts, tandis que la numérotation *suffixe* indique l'ordre dans lequel ils ont été fermés. Les arcs qui ont été suivis dans le parcours sont appelés *arcs d'arbre*.

Avec les numéros préfixes et suffixes, on peut facilement caractériser les types d'arcs : avant, arrière ou croisé.

(i, j) est avant	préfixe(i) < préfixe(j)	suffixe(i) > suffixe(j)
(i, j) est croisé	préfixe(i) > préfixe(j)	suffixe(i) > suffixe(j)
(i, j) est arrière	préfixe(i) > préfixe(j)	suffixe(i) < suffixe(j)

1. Implémenter cet algorithme en Python et terminer en affichant les listes des arcs d'arbre, et des arcs avants, arrières et croisés.

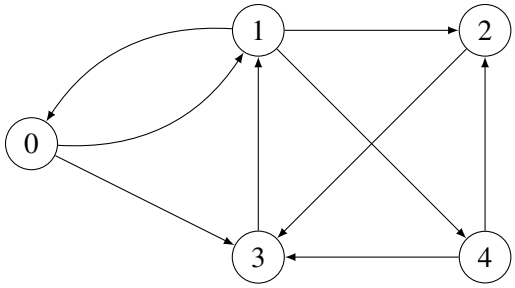
   Deux fonctions seront utilisées : la première, appelée `DFS(G)`, prend en argument un objet G de type `Graphe` tandis que l'autre fonction, `explore`, qui sera appelée par la première, prend en argument le graphe G et un sommet i à explorer (et éventuellement d'autres variables, si nécessaire).

   - La fonction `DFS(G)` définit :
     (a) la liste `etat` qui contient le statut de chaque sommet,
     (b) les listes `pref` et `suff` liées aux numérotations préfixes et suffixes, qui donnent l'ordre dans lequel chaque sommet est ouvert puis fermé (Par exemple, si le sommet 0 est ouvert en premier, et fermé en troisième, on aura `pref[0]=1`, `suff[0]=3`),
     (c) la liste vide A qui contiendra les arcs d'arbre visités,
     (d) les compteurs `ind_ouvert` et `ind_ferme` qui donnent l'ordre d'ouverture et de fermeture des sommets.

   Toutes ces variables doivent être déclarées comme globales, pour pouvoir les utiliser quand on appelle `explore`. Cette fonction devra également afficher à la fin l'ensemble des arcs d'arbre ainsi que l'ensemble des arcs avants, arrières et croisés en fonction de la numérotation préfixe et suffixe.
   Tant que tous les sommets du graphe n'ont pas été explorés, la fonction `DFS` choisit un sommet quelconque i non exploré et appelle la fonction `explore(G,i)`.

- La fonction `explore(G,i)` ouvre le sommet i, l'affiche, gère les numérotations préfixe et suffixe, et s'appelle en récursion sur les successeurs non explorés de i.

2. Tester l'algorithme de parcours en profondeur sur le graphe orienté suivant :



Exercice 3 : Matrice d'adjacence et connexité

Soit A la matrice d'adjacence d'un graphe G d'ordre n. Définissons $A^k = A * A^{k-1}$ pour $k = 2, 3, \ldots$.

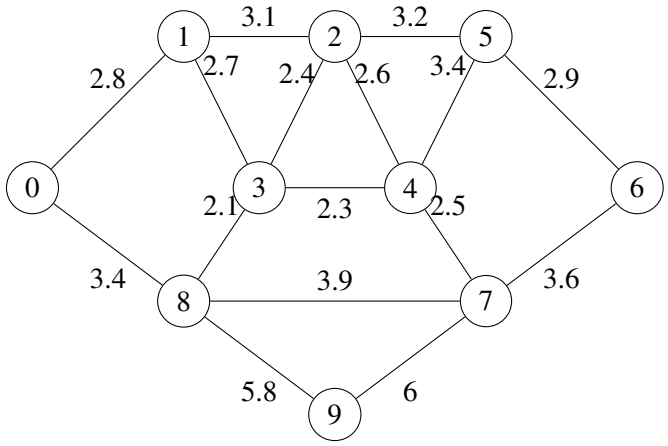
1. Écrire une fonction qui prend en paramètre un entier k et une matrice A et renvoie A^k. Faire appel au module `numpy` pour effectuer les produits de matrice. Que peut-on déduire de la valeur a_{ij}^k sur la connexité entre les sommets i et j dans le graphe G ?

2. On sait que G est fortement connexe si et seulement si la matrice R définie par $R = A + A^2 + A^3 + \cdots + A^n$ ne possède aucune entrée nulle. Écrire une fonction qui effectue ce test.

3. Appeler cette fonction sur le graphe de l'Exercice 2 pour savoir si ce graphe est fortement connexe. Et si on retire l'arc $(1, 0)$?

Exercice 4 : Arbre couvrant de poids minimal

Soit $G = (X, E)$ le graphe non-orienté valué d'ordre $n = 10$ ci-dessous.



Un *arbre couvrant* $T = (X, E_T)$ est un graphe partiel de G connexe et sans cycle. En conséquence, $|E_T| = n - 1$. Le poids d'un arbre couvrant est la somme des poids des arêtes lui appartenant. On cherche à déterminer l'arbre couvrant de poids minimal, noté T^*.

Le principe de l'algorithme de Kruskal est de sélectionner les arêtes de plus petit poids de façon à former un arbre. Après avoir ordonné les arêtes par ordre de poids croissant, on construit pas à pas

l'arbre couvrant en choisissant une arête de plus petit poids à ajouter à la solution courante si cet ajout ne provoque pas de cycle. On s'arrête lorsque l'on obtient $n-1$ arêtes dans la solution.

Pour détecter la construction d'un cycle, l'idée est d'affecter à chaque sommet un label correspondant à l'indice représentatif de la composante connexe dans laquelle il se trouve dans la solution courante (par exemple l'indice minimal parmi les éléments qui composent la composante connexe). A chaque nouvel ajout d'arête dans la solution courante, les labels sont mis à jour. Une arête est ajoutée dans la solution courante si les labels associés à ses deux extrémités sont différents.

1. Implémenter l'algorithme de Kruskal et déterminer la solution optimale du graphe donné en exemple, ainsi que son poids total.

2. Écrire une fonction permettant de générer aléatoirement un graphe non orienté valué de densité $d = 0.7$ avec des poids choisis aléatoirement entre 2 et 50. On pourra utiliser la méthode `sample` du module `random` de Python. Donner le temps d'exécution de votre programme Python pour un graphe d'ordre 100, 1000, 2000.

3. Comment pouvez-vous détecter que G généré aléatoirement n'est pas connexe et que E_T^* n'existe pas ?

Exercice 5 : Utilisation du module `networkx`

Le module `networkx` est très complet (cf. `https://networkx.github.io/` avec une documentation complète disponible): il permet de créer, représenter des graphes (orienté ou non, simple ou multiple, valué ou non) et d'appliquer des algorithmes classiques. Pour l'utiliser, il suffit d'exécuter : `import networkx` ou, `import networkx as nx` si on souhaite utiliser l'alias `nx`. Un graphe G est représenté par sa liste de voisins (ou de successeurs si le graphe est orienté), implémentée sous la forme d'un dictionnaire de dictionnaires. Pour déclarer une variable `g` de type graphe non orienté et, une variable `go` de type graphe orienté, on exécute les instructions :

```
g=nx.Graph()
go=nx.DiGraph()
```

`g` (resp. `go`) est alors un dictionnaire dont chaque clé représente un sommet i de G et, chaque valeur associée à i est un dictionnaire représentant la liste des voisins (resp. successeurs), avec comme clé le sommet j, voisin (resp. successeur) de i dans G, et comme valeur un dictionnaire représentant entre autre la longueur de l'arc (i, j) si elle existe (plus d'autres paramètres éventuels de l'arc). Pour initialiser une variable `g` ou `go` de type Graph ou DiGraph, on a plusieurs possibilités et méthodes à notre disposition :

- ajouter un sommet de nom `'s'` : `g.add_node('s')`
- ajouter une arête (i, j) : `g.add_edge(i,j)`
- ajouter une liste de sommets : `g.add_nodes_from(lst)` où `lst` est de type `list`
- ajouter une liste `lst` d'arêtes (ou d'arcs valués ou non) : `g.add_edges_from(lst)` où `lst` est de type `list` de tuples `(i,j[,val])` avec `val` la valeur de l'arête (ou l'arc) (i, j).

Suite à l'initialisation : `for n in g` permet de parcourir la liste des sommets, `for d in G[i]` permet de parcourir la liste des voisins du sommet i, `g.adj` retourne le dictionnaire de dictionnaires `g`, `g.nodes()` retourne la liste des sommets, `g.edges()` retourne la liste des arêtes.

Pour obtenir une représentation graphique de G à l'aide de `matplotlib.pyplot`, il suffit alors d'exécuter les instructions suivantes :

```
nx.draw(g)
plt.show()
```

1. Stocker le graphe de l'exercice 1 sous la forme d'une variable de type `DiGraph`.
2. Représenter graphiquement le graphe de l'exercice 1 avec `matplotlib`.
3. A l'aide du module `networkx`, tester si le graphe de l'exercice 1 est connexe. On trouve deux fonctions : `is_connected(G)` retourne vrai si `G` est connexe et faux sinon ; `number_connected_components(G)` retourne le nombre de composantes connexes de G.
4. Ecrire un programme qui génére aléatoirement un graphe orienté d'ordre `n` à l'aide de `networkx`, avec par exemple la fonction

   ```
   nx.gnp_random_graph(n, p, seed=None, directed=True)
   ```

5. A l'aide du programme de l'exercice 2, tester la connexité forte de graphes d'ordre 100, 200, 300 et 400 générés aléatoirement. Comparer les temps de calcul pour faire ce même test mais avec la fonction du module `networkx` `is_strongly_connected(G)`.

Exercice 6 : Résolution de problèmes avec `networkx`

En utilisant le module `networkx`, proposer une solution à ces différents problèmes :

- Comment faire pour aller le plus rapidement possible de Bordeaux à Grenoble sachant que :

Bordeaux → Nantes	4h
Bordeaux → Marseille	9h
Bordeaux → Lyon	12h
Nantes → Paris-Montparnasse	2h
Nantes → Lyon	7h
Paris-Montparnasse → Paris-Lyon	1h
Paris-Lyon → Grenoble	4h30
Marseille → Lyon	2h30
Marseille → Grenoble	4h30
Lyon → Grenoble	1h15

  Proposer une modélisation à l'aide d'un graphe. Le problème à résoudre dans ce graphe est un 'shortest path problem'.

- 8 groupes d'étudiants (numérotés de G1 à G8) doivent passer des examens dans différentes disciplines, chaque examen occupant une demi-journée :

chimie	G1 et G2
Electronique	G3 et G4
Informatique	G3, G5, G6 et G7
Mathématiques	G1, G5, G6 et G8
Physique	G2, G6, G7 et G8

On cherche à organiser la session d'examen la plus courte possible. Proposer une modélisation à l'aide d'un graphe. La détermination d'une session d'examen la plus courte possible est un 'coloring problem'.

- En 1941, les escadrilles anglaises se composaient d'avions biplaces, mais certains pilotes ne pouvaient pas faire équipe dans le même avion par suite de différence de langues et de capacités. Sous ces contraintes, on cherche à déterminer le nombre maximum d'avions pouvant voler simultanément.
  Considérer l'exemple numérique suivant où 8 pilotes sont disponibles pour faire voler 4 avions. Dans le tableaux ci-dessous, une croix ligne i colonne j signifie que les pilotes i et j ne peuvent pas faire équipe :

	1	2	3	4	5	6	7	8
1			×		×	×	×	
2							×	×
3	×			×	×	×	×	×
4			×			×		
5	×		×					×
6	×		×	×			×	×
7	×	×	×			×		×
8		×	×		×	×	×	

  Modéliser ce problème sous la forme d'un graphe et déterminer le nombre maximum d'avions pouvant voler simultanément. La composition d'équipes est un 'matching problem'.