? À rendre LUNDI 2 février 2026 (au plus tard)
Devoir Maison n°6 – Graphes

Travail à faire :

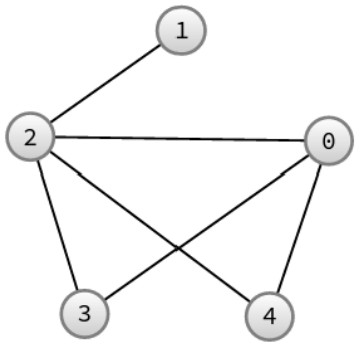
- Retravailler les cours et les TP de MPSI/PCSI sur les graphes.
- Vous aider de la fiche « Révisions sur les graphes. »
- Si vous n'êtes pas à l'aise sur les graphes : exercice n°1
- Si vous êtes plutôt à l'aise sur les graphes : exercice n°2.

En cas de difficulté, me poser des question via cahier de prépa ou par mail.

Exercice n°1 La base des graphes

Partie I Représentation des graphes

Q1. On considère le graphe :



(a) Écrire la liste d'adjacence.

(b) Écrire le dictionnaire en python pour représenter ce graphe.

(c) Écrire la matrice d'adjacence.

Q2. Tracer le graphe de la matrice $M_1 = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}$

Q3. Tracer le graphe de la matrice $M_3 = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$

Partie II Degré d'un sommet

On considère dans cette partie un **graphe non orienté**, qui peut être pondéré.

Q4. Écrire une fonction `degre_dico(d:dict,s:str)->int` qui prend en argument un graphe dont la liste d'adjacence est implémentée par un dictionnaire `d` et un sommet `s`, et renvoie le degré du sommet `s`.

Q5. Écrire une fonction `degres(d:dict)->dict` qui prend en argument un graphe dont la liste d'adjacence est implémentée par un dictionnaire `d` et renvoie un dictionnaire dont les clés sont les sommets et les valeurs les degrés de chaque sommet.

Q6. Écrire une fonction `degre_mat(M:list,s:int)->int` qui prend en argument un graphe représenté par sa matrice d'adjacence `M` et un sommet `s`, et renvoie le degré du sommet `s`.

Partie III Liste des voisins d'un sommet

On considère dans cette partie un graphe non orienté, pouvant être pondéré.

Q7. Écrire une fonction `voisin_dico(d,s)` qui prend en entrée une liste d'adjacence représentée par un dictionnaire et un sommet `s`, et renvoie la liste des voisins du sommet `s`.

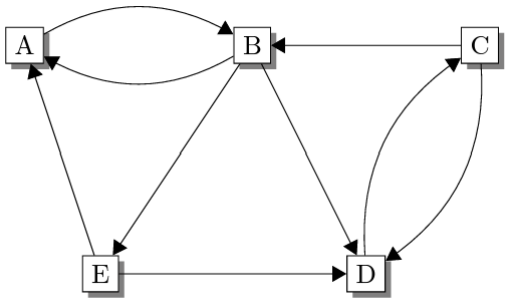
Q8. Écrire une fonction `voisin_mat(M:list,s:int)->list` qui prend en entrée une matrice d'adjacence et un sommet `s`, et renvoie la liste des voisins du sommet `s`.

Q9. Écrire une fonction `est_voisin_mat(M:list,s1:int:int,s2:int)->list` qui prend en entrée une matrice d'adjacence et deux sommets `s1` et `s2`, et renvoie `True` s'il y a une arête entre `s1` et `s2`, `False` sinon.

Partie IV Graphe orienté

On considère la liste des successeurs d'un graphe orienté représenté à l'aide d'un dictionnaire `G`.

Par exemple



Q10. Écrire le dictionnaire des successeurs du graphe de ci-dessus.

Q11. On étudie le code ci-dessous :

```
1 def mystere(G:dict)->dict:
2     G2 = {s:[] for s in G}
3     for s in G:
4         for v in G[s] :
5             G2[v].append(s)
6     return G2
```

(a) ligne 2 : que fait cette ligne ?
Donner `G2` à cette étape, si on étudie le graphe de Q10.

(b) ligne 3 : que parcourt cette boucle `for` ? quelles seront les « valeurs » successives prises par `s` ?

(c) ligne 4 : que représente `G[s]` ? que parcourt cette boucle `for` ?

(d) ligne 5 : que représente `G[v]` ? que fait cette ligne ?

(e) Recopier et remplir le tableau ci-dessous aux différentes étapes si on considère le graphe de Q10.

étape n°	s	v	G2
0 (ligne 2)	rien	rien	*à compléter*
1	A	B	`G2={A:[],B:[A],C:[],D:[],E:[]}`
2	B	A	*à compléter*
3	B	D	*à compléter*
4	B	E	*à compléter*
5	C	*à compléter*	*à compléter*
à poursuivre			

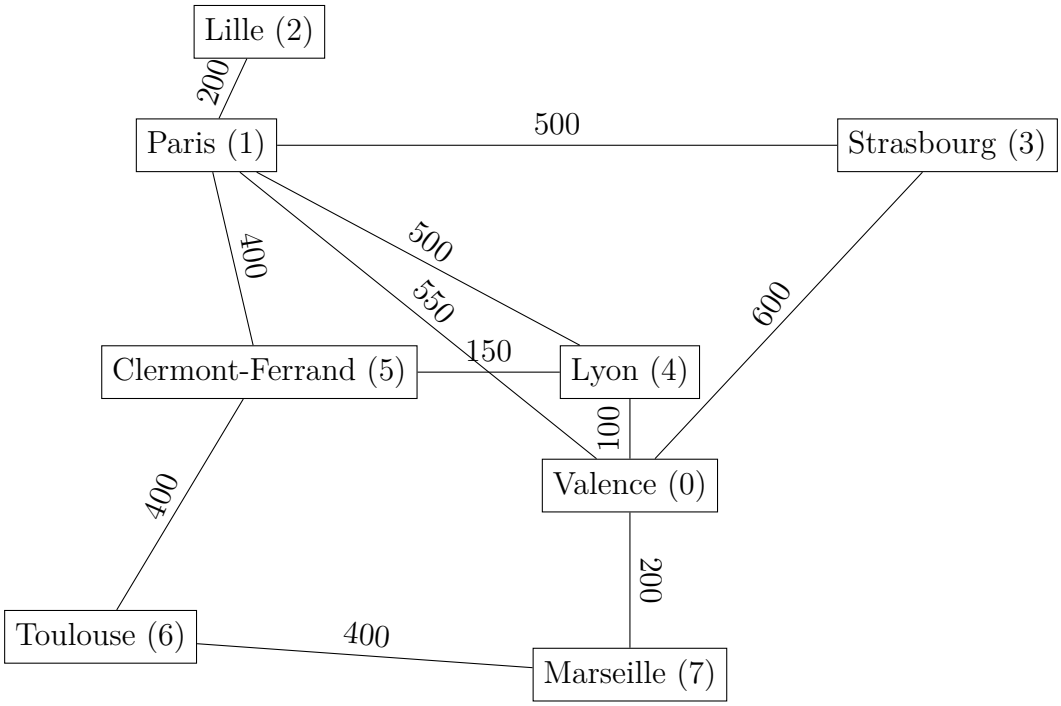
(f) Représenter le graphe `G2` ainsi obtenu.

(g) Conclure sur ce que renvoie cette fonction.

Exercice n°2 Tournée d'un orchestre

La cheffe d'orchestre d'un grand orchestre symphonique Valentinois est en train d'organiser sa future tournée. L'orchestre doit se rendre dans un certain nombre de villes. Elle réfléchit à organiser l'ordre des concerts afin de réduire les temps de trajet et les coûts liés au transport.

Les villes sont les sommets du graphe, et les distances entre les villes sont les poids des arêtes. Le graphe n'est pas orienté. Les villes sont numérotées de 0 à 7 afin de renseigner les sommets des graphes avec un entier plutôt qu'une chaîne de caractère.



Partie I Représentations d'un graphe

Q1. Écrire la dictionnaire d'adjacence en Python qui représente le graphe précédent : la clé est le sommet considéré et la valeur une liste de listes du même format que les sous-liste de `liste_adj`.

Q2. Écrire la matrice de pondération (pas en python) du graphe précédent. Pourquoi la matrice est-elle symétrique ?

Comment peut-on la représenter en python ? (on ne demande pas le code)

Q3. Écrire une fonction `degres(G:dict)->dict` qui prend en argument un graphe représenté par son dictionnaire d'adjacence et qui renvoie le dictionnaire dont les clés sont les sommets et les valeurs le degré de chaque sommet.

Q4. Écrire une fonction `voisins(G:list)->dict)` qui prend en argument un graphe représenté par sa matrice d'adjacence et qui renvoie le dictionnaire des voisins de chaque sommet, les clés sont les sommets, les valeurs les listes des voisins de chaque sommet.

Partie II Distance sur un graphe

Q5. Écrire la fonction `distance_parcourue_mat(G:list[list],tournee:list)->float` qui prend en argument le graphe `G` représenté par une matrice d'adjacence et la liste `tournee` qui est la liste des villes de la tournée, dans l'ordre de la tournée, et qui renvoie la distance parcourue.

Cette fonction renverra -1 si le parcours proposé n'est pas possible (deux villes successives ne sont pas reliées par une arête).

Q6. Écrire la fonction `distance_parcourue_dict(G:dict,tournee:list)->float` qui prend en argument le graphe `G` représenté par un dictionnaire d'adjacence et la liste `tournee` qui est la liste des villes de la tournée, dans l'ordre de la tournée, et qui renvoie la distance parcourue.

Cette fonction renverra -1 si le parcours proposé n'est pas possible (deux villes successives ne sont pas reliées par une arête).

Partie III Parcours d'un graphe

Les graphes peuvent être parcourus en largeur ou en profondeur.

Q7. Mettre en œuvre à la main le parcours du graphe, à partir de Valence en largeur, et puis en profondeur.

Le parcours en largeur passe par tous les voisins d'un sommet avant de parcourir les descendants de ces voisins. Les sommets passent dans une file d'attente, c'est-à-dire structure de données de type First In First Out.

Pour cela, on opère en marquant les sommets du graphe à visiter et les sommets du graphes déjà découverts.

Lorsqu'un sommet est découvert, il intègre l'ensemble des éléments à visiter, c'est-à-dire la file d'attente `attente` : `attente.append()`. Lorsque le sommet a été traité, il quitte la file : `attente.popleft()`.

Il est donc également nécessaire de garder la trace du passage sur un sommet afin de ne pas traiter plusieurs fois un même sommet : si un sommet a été visité alors il intègre l'ensemble des éléments découverts `marque`. On utilise pour cela un dictionnaire, car c'est plus efficace pour vérifier si un sommet a déjà été visité (accès en temps constant, alors qu'avec une liste, l'accès est de complexité linéaire).

Q8. Recopier et compléter la fonction suivante qui effectue de façon itérative le parcours en largeur d'un graphe représenté par un dictionnaire l'adjacence.

```
def parcours_largeur(G,s0):
    visite=[] # liste des voisins visités
    marque={} # dictionnaire
    attente=deque() # file qui garde les sommets en attente
    ... # on ajoute au sommet de la file le sommet de départ
    while ... : # tant que la file n'est pas vide
        s=... # on visite le sommet au début de la file
        if ... : # si le sommet n'a pas déjà été visité
            .... # on l'ajoute à la liste des sommets visités
            .... # on l'ajoute au dictionnaire marque
            ... : # il faut visiter les voisins du sommet
                if v not in  : # le voisin v de s n'a pas déjà été visité
                    ... # on le place dans la file des sommets en attente
    d'être visité
    return visite
```

Le parcours en profondeur s'exprime de façon presque identique avec le parcours en utilisant une pile (à la place de la file) afin gérer la découverte des voisins dans l'ordre de la profondeur du graphe.

Q9. Recopier et compléter la fonction suivante qui effectue de façon itérative le parcours en profondeur d'un graphe représenté par un dictionnaire d'adjacence.

```
def parcours_profondeur(G,s0):
    visite=[] # liste des voisins visités
    marque={} # dictionnaire des voisins visités
    attente=[] # pile qui garde les sommets en attente
    .... # on ajoute au sommet de la pile le sommet de départ
    while ... : # tant que la pile n'est pas vide
        s=... # on visite le sommet au fin de la pile
        if ... : # si le sommet n'a pas déjà été visité
            .... # on l'ajoute à la liste des sommets visités
            ... # on l'ajoute au dictionnaire
            for v in G[s] : # pour les voisins de sommet
                if v not in ... : # le voisin v de n'a pas déjà été
   visité
                    ... # on le place dans la pile des sommets en attente
    d'être visité
    return visite
```

Partie IV Plus petite distance sur un graphe

L'orchestre étant Valentinois, le départ de la tournée se fait depuis Valence.

On rappelle ici l'algorithme de Dijkstra qui permet de déterminer, depuis un sommet (Valence ici), la distance minimale qui le sépare des autres sommets.

On commence par affecter une valeur très grande à chacun des autres sommets, disons infinie, et la valeur 0 au sommet de départ (Valence dans notre exemple).

À chaque étape, on effectue le meilleur choix possible : c'est un algorithme glouton. Tout au long de l'algorithme on va garder en mémoire le chemin le plus court depuis Valence pour chacune des autres points dans un tableau.

On répète toujours le même processus :

1. On choisit le sommet accessible de distance minimale comme sommet à explorer.
2. À partir de ce sommet, on explore ses voisins et on met à jour les distances pour chacun. On ne met à jour la distance que si elle est inférieure à celle que l'on avait auparavant.
3. On répète jusqu'à ce qu'on arrive au point d'arrivée ou jusqu'à ce que tous les sommets aient été explorés.

Q10. Mettre en œuvre l'algorithme précédent pour déterminer l'ordre de la tournée, et la distance parcourue par l'orchestre depuis Valence pour se rendre dans chaque ville.

Pour cela, recopier et remplir le tableau.

En déduire les chemins les plus courts depuis Valence vers chaque ville.

Étape	Valence	Paris	Lille	Strasbourg	Lyon	Clermont	Toulouse	Marseille
0	**0**	∞	∞	∞	∞	∞	∞	∞
1	×							
2	×							
3	×							
4	×							
5	×							
6	×							
7	×							
8	×							

Pour écrire le programme, nous allons utiliser un dictionnaire dont les clés sont les sommets et les valeurs sont les distances respectives entre chaque sommet et le sommet de départ, comme dans le tableau complété précédemment.

Q11. Nous avons pour cela besoin d'une fonction qui calcule le minimum des valeurs contenues dans un dictionnaire et renvoie la clé correspondante.

Écrire la fonction `minimum(D:dict)`.

Pour initialiser la valeur du minimum, on pourra utiliser `float('inf')`.

Q12. Recopier et compléter le programme `dijkstra` ci-dessous.

```
def dijkstra(G,s0):
    """
    arguments :
    G : dictionnaire qui représente le graphe
    s0 : sommet de départ
    retour :
    dictionnaire des distances minimales entre s0 et les différents sommets
    """
    distances={} # dictionnaire des sommets visités et de leurs distances minimales depuis s
    d={k:float('inf') for k in dico} # initialisation des distances à l'infini des sommets restants à visiter
    d[s0] = ... # distance 0 au  sommet s de départ
    while ... # fini quand d est vide
        k= ...      # choix du sommet restant à visiter situé à la distance minimale
        for j ... # visite tous les voisins de k
            # v un voisin de k, et c la distance entre k et v :
            v,c = ...
            if ... # si v n'a pas déjà été atteint
                # distance minimale entre la précédente, et celle pour arriver à v depuis k (utiliser la fonction min(a,b)) :
                d[v] = ...
        distances[k]= ... # copie le sommet et la distance dans distances
        del d[k]                    # supprime le sommet de d
    return distances
```