Parcours de graphes

Objectifs :

- Rappeler la notion de graphe et les structures de données adaptées à leur représentation.
- Programmer des parcours en profondeur (révision de pile et de récursivité).
- Revoir l'algorithme de Dijkstra.

Notion et représentations de graphes non orientés

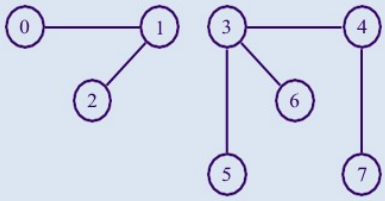
Un **graphe non orienté** est un couple (S , A) tel que :

- S est un ensemble, dont les éléments sont les sommets du graphe,
- A est un ensemble de paires de sommets, appelés arêtes du graphe.

On peut le représenter :

- Soit par une **liste d'adjacence** ou un **dictionnaire d'adjacence** : on considère les listes de voisins de chaque sommet du graphe.
  C'est intéressant lorsque le graphe est peu dense (peu d'arêtes).
- Soit par une **matrice d'adjacence** : une matrice (symétrique) M tel que Mi,j vaut 1 s'il y a une arête entre le sommet n°i et le sommet n°j et 0 sinon.
  C'est intéressant lorsque le graphe est dense (beaucoup d'arêtes).

Exemple :



$S = [\![0, 7]\!]$ et $A = \{ \{0, 1\}, \{1, 2\}, \{3, 4\}, \{3, 5\}, \{3, 6\}, \{4, 7\} \}$

Liste d'adjacence :

```
L= [[1],
    [2, 0],
    [1],
    [6, 4, 5],
    [7, 3],
    [3],
    [3],
    [4]
   ]
```

Dictionnaire d'adjacence :

```
d={ 0:[1],
    1 : [2,0],
    2 : [1],
    3 : [6, 4, 5],
    4 : [7, 3],
    5 : [3],
    6 : [3],
    7 : [4]
   }
```

Matrice d'adjacence :

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

(Indices entre 0 et 7)

Définir une fonction `genereDico` qui prend en paramètre d'une liste d'adjacence d'un graphe et la tranforme en dictionnaire d'adjacence.
Définir une fonction `genereMatrice` qui prend en paramètre d'une liste d'adjacence d'un graphe et la tranforme en liste de liste représentant la matrice d'adjacence.

Entrée[17]:

```
def genereDico(L):
    """ genereDico(L: list) -> dict
        entrée  : L, liste correspondant la liste d'adjacence du graphe
        sortie : dico, dictionnaire  correspondant au dictionnaire d'adjacence d
    """
    dico={}
    for i in range(len(L)) :
        dico[i] = []
        for j in range(len(L[i])):
            dico[i].append(L[i][j])
    return dico

def genereMatrice(L):
    """ genereDico(L: list) -> list
        entrée  : L, liste correspondant la liste d'adjacence du graphe
        sortie : mat, liste de listes, correspondant a la matrice d'adjacence du
    """
    n = len(L)
    # mat=[[0]*n]*n
    mat = [ [0 for j in range(n)] for i in range( n )  ]
    print(mat)
    for i in range(len(L)) :
        for val in L[i]:
            mat[i][val] = 1
    return mat

# L: liste d'adjacence du graphe d'exemple
L = [[1], [2, 0], [1], [6, 4, 5], [7, 3], [3], [3], [4]]
print( genereDico(L) )
# {0: [1], 1: [2, 0], 2: [1], 3: [6, 4, 5], 4: [7, 3], 5: [3], 6: [3], 7: [4]}
print( genereMatrice(L) )
# [[0, 1, 0, 0, 0, 0, 0, 0], [1, 0, 1, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0, 0, 0],
```

```
{0: [1], 1: [2, 0], 2: [1], 3: [6, 4, 5], 4: [7, 3], 5: [3], 6: [3], 7: [4]}
[[0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0], [0,
0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0,
0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0]]
[[0, 1, 0, 0, 0, 0, 0, 0], [1, 0, 1, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0, 0, 0], [0,
0, 0, 0, 1, 1, 1, 0], [0, 0, 0, 1, 0, 0, 0, 1], [0, 0, 0, 1, 0, 0, 0, 0], [0, 0, 0,
1, 0, 0, 0, 0], [0, 0, 0, 0, 1, 0, 0, 0]]
```

Parcours en profondeur et en largeur

Parcours en profondeur

Le principe du **parcours en profondeur** d'un graphe est celui du **backtracking** :
Pour parcourir tous les sommets d'un graphe, on part d'un sommet que l'on marque comme visité, puis on visite l'un de ses voisins que l'on marque, et ainsi de suite. Lorsque l'on ne trouve plus de voisin, on revient en arrière et on passe au voisin suivant du sommet précédent.
On peut implémenter un parcours en profondeur :

- soit avec une fonction récursive,
- soit en utilisant une pile.

La fonction globale `parcourir` fait appel à une fonction `parcourir_voisins` permettant de parcourir les voisins d'un sommet au sens large : c'est-à-dire tous les sommets que l'on peut atteindre à partir du sommet de départ. C'est la fonction `parcourir_voisins` qui sera écrite récursivement ou à l'aide d'une pile.

Avec le graphe codé par une liste d'adjacence L, on a donc en pseudo-code suivant :

```
parcourir(L)
    Pour chaque sommet faire
        Si le sommet n'est pas marqué comme visité alors
            parcourir_voisins(L,sommet)
        finsi
    finpour

avec parcourir_voisins pouvant être définie de 2 manières : récursive ou itérative
##### Version récursive :
parcourir_voisins(L, sommet)
    Marquer le sommet comme visité
    Traiter le sommet
    Pour chaque voisin du sommet faire
        Si le voisin n'est pas marqué comme visité alors
            parcourir_voisins(L,voisin)
        finsi
    finpour

#### Version avec une pile :
parcourir_voisins(L,sommet)
    Créer une pile
    Marquer le sommet comme visité
    Empiler le sommet
    Tant que la pile n'est pas vide faire
        Dépiler un sommet
        Traiter  le  sommet
        Pour chaque voisin du sommet faire
            Si le voisin n'est pas marqué comme visité alors
                Marquer le voisin comme visité
                Empiler le voisin
            finsi
        finpour
    fintantque
```

1. On va effectuer un parcours en profondeur (avec les 2 approches) dans lequel on gardera en mémoire l'ordre dans lequel les sommets ont été parcourus.

1.1-Définir 2 **fonctions `parcours_voisins_rec(L, sommet, visite, parcours)` et `parcours_voisins_pile(L, sommet, visite, parcours)`** prenant pour paramètres :

- `L` , la liste d'adjacence,
- `sommet` , le numero du sommet à partir duquel on commence,
- `visite` , une liste de booléens telle que `visite[sommet]` vaut `True` lorsque le sommet a été visité, modifiée par effet de bord,
- `parcours` , une liste initialement vide, dans laquelle on empilera les numéros des sommets au fur et à mesure du parcours, modifiée par effet de bord

1.2- Definir la fonction `parcourir(L, fonctionParcours = parcours_voisins_rec)` qui a pour paramètres :

- la liste d'adjacence `L` :
- le nom de la fonction qui sera appelée pour effectuer le parcours en profondeur `fonctionParcours` (ici ce paramètre pourra prendre pour valeur `parcours_voisins_rec` ou `parcours_voisins_pile` lors de l'appel)

Cette fonction parcourt l'ensemble des sommets pas encore visités grâce à la fontion `fonctionParcours` et retourne la liste `parcours` des sommets visités, dans l'ordre de la visite.
Tester avec le graphe donné au début du TP.

Entrée[19]:

```
# TP graphes

## Parcours en profondeur
def parcours_voisins_rec(L, sommet, visite, parcours):
        """parcours récursif des voisins non visités de sommet"""
        """ parcours_voisins_rec(L : list, sommet: int, visite: list, parcours:
           entrees :L : liste, liste d'adjacence du graphe
                   sommet : entier, Numero du sommet dont on part
                   visite : liste de booléens qui indique pour chaque sommet(in
                      s'il a déjà été visité (True ou False), modifiée par effe
                   parcours : liste des sommets visités (dans l'ordre de la vis
        """
        visite[sommet] = True
        parcours.append(sommet)
        lesVoisins = L[sommet]
        for voisin in lesVoisins :
            if not visite[voisin]:
                parcours_voisins_rec(L,voisin, visite, parcours)

"""Parcours en profondeur itératif"""
from collections import deque
def parcours_voisins_pile(L, sommet,  visite, parcours):
        """parcours des voisins non visités de sommet à l'aide d'une pile"""
        pile = [sommet]
        visite[sommet] = True
        while len(pile) != 0:
            sommet1 = pile.pop()
            parcours.append(sommet1)
            for voisin in L[sommet1]:
                if not visite[voisin]:
                    visite[voisin] = True
                    pile.append(voisin)

# version iterative en utilisant les deque en python
from collections import deque
def parcours_voisins_pile(L, sommet,  visite, parcours):
        """parcours des voisins non visités de sommet à l'aide d'une pile"""
        pile = deque([sommet])
        visite[sommet] = True
        while len(pile) != 0:
            sommet1 = pile.pop()
            parcours.append(sommet1)
            lesVoisins = L[sommet1]
            for voisin in lesVoisins:
                if not visite[voisin]:
                    visite[voisin] = True
                    pile.append(voisin)

def parcourir(L, fonctionParcours = parcours_voisins_rec):
    """Parcours en profondeur des sommets du graphe"""
    """ parcourir(L : list)-> list
            entrees : L, liste, liste d'adjacence du graphe
                    : fonctionParcours, nom de la fonction à appeler pour le par
            sortie : parcours, liste des sommets visités (dans l'ordre de la vis
    """
    n = len(L)
    visite = [False for _ in range(n)] # permet de savoir si un sommet a été vis
    parcours = [] # permet de garder les sommets dans l'ordre de parcours
    for sommet in range(n):
        if not visite[sommet]:
            fonctionParcours(L, sommet, visite, parcours)
    return parcours


L = [[1], [2, 0], [1], [6, 4, 5], [7, 3], [3], [3], [4]]
```

```
66
67 print(parcourir(L))
68 # ou
69 # print(parcourir(L, parcours_voisins_rec))
70 # [0, 1, 2, 3, 6, 4, 7, 5]
71
72 print(parcourir(L, parcours_voisins_pile))
73 # [0, 1, 2, 3, 5, 4, 7, 6]
```

```
[0, 1, 2, 3, 6, 4, 7, 5]
[0, 1, 2, 3, 5, 4, 7, 6]
```

Parcours en largeur

Il n'est pas facile de programmer un parcours en largeur récursivement car il ne s'agit plus de parcourir chaque voisin récursivement. Par contre, c'est très facile itérativement : il suffit de reprendre le parcours en profondeur et de remplacer la pile par une file.

1.3- Créer la fonction `parcours_voisins_file` implémentant le parcours en largeur et l'appeler avec `parcourir`

Entrée[20]:

```
1  ## Parcours en largeur
2
3  from collections import deque
4  def parcours_voisins_file(L,sommet,  visite, parcours):
5          """parcours des voisins non visités de sommet à l'aide d'une pile"""
6          file = deque([sommet])
7          while len(file) != 0:
8              sommet = file.pop()
9              if not visite[sommet] :
10                 visite[sommet] = True
11                 parcours.append(sommet)
12                 for voisin in L[sommet]:
13                     if not visite[voisin]:
14                         visite[sommet] = True
15                         file.appendleft(voisin)
16
17 def parcourir(L, fonctionParcours = parcours_voisins_rec):
18     """Parcours en profondeur des sommets du graphe"""
19     """ parcourir(L : list)-> list
20             entrees : L, liste, liste d'adjacence du graphe
21                     : fonctionParcours, nom de la fonction à appeler pour le par
22             sortie : parcours, liste des sommets visités (dans l'ordre de la vis
23         """
24     n = len(L)
25     visite = [False for _ in range(n)] # permet de savoir si un sommet a été vis
26     parcours = [] # permet de garder les sommets dans l'ordre de parcours
27
28     for sommet in range(n):
29         if not visite[sommet]:
30             fonctionParcours(L, sommet, visite, parcours)
31     return parcours
32
33 L = [[1], [2, 0], [1], [6, 4, 5], [7, 3], [3], [3], [4]]
34 print(parcourir(L, parcours_voisins_file))
35 # [0, 1, 2, 3, 6, 4, 5, 7]
```

```
[0, 1, 2, 3, 6, 4, 5, 7]
```

Composantes connexes d'un graphe

2. On souhaite maintenant déterminer les **composantes connexes du graphe** : ce sont les classes d'équivalence de la relation d'équivalence « il existe un chemin (suite d'arêtes successives) reliant

les deux sommets » sur l'ensemble des sommets.Pour ce faire faire, nous allons adapter les fonctions précédentes

Le graphe d'exemple comporte 2 composantes connexes.

D'abord, nous allons adapter le parcours en profondeur (l'une ou l'autre version) en créant une fonction `parcours_voisins` ayant pour paramètres :

- La liste `L` d'adjacence du graphe,
- une liste `CC` , initialisée avec des valeurs -1, telle que `CC[sommet]` est le numéro de la composante connexe à laquelle appartient le sommet. Cette liste sera modifiée par effet de bord au fur et à mesure du parcours
  Dans cette liste, la valeur -1 correspond donc à un sommet non visité.
- un entier `numero_CC` , qui correspond au numéro de la composante connexe en cours de parcours.

2.1- Définir la fonction `parcours_voisins(L, sommet, CC, numero_CC)`
2.2- Définir la fonction `listeComposantesConnexes` qui est une adaptation de la fonction `parcourir` (1.2). Cette fonction fait appel à la fonction `parcours_voisins` . Cette fonction prend en paramètre la liste d'adjacence `L` . Elle renvoie la liste `CC` ainsi que le nombre de composantes connexes différentes du graphe.

Entrée[22]:

```
## Composantes connexes

# Liste des composantes connexes récursif
def parcours_voisins(L, sommet, CC, numero_CC):
        """ parcours_voisins(L : list, sommet: int, CC: list, numero_CC:int):
               entrees :L : liste, liste d'adjacence du graphe
                        sommet : entier, Numero du sommet dont on part
                        CC : liste initialisée avec des valeurs -1,
                             telle que CC[sommet] est le numéro de la composante
                             modifiée par effet de bord
                        numero_CC : int, numéro de la composante connexe
        """
        CC[sommet] = numero_CC
        for voisin in L[sommet]:
            if CC[voisin] < 0:
                parcours_voisins(voisin, CC, numero_CC)
def listeComposantesConnexes(L):
    """ listeComposantesConnexes(L : list) -> list, int:
            entrees :L : liste, liste d'adjacence du graphe
            sorties :CC : liste initialisée avec des valeurs -1,
                             telle que CC[sommet] est le numéro de la composante
                     int, nombre de composantes connexes du graphe
    """
    n = len(L)
    CC = [-1 for _ in range(n)]
    numero_CC = -1
    for sommet in range(n):
        if CC[sommet] < 0:
            numero_CC += 1
            parcours_voisins(L,sommet, CC, numero_CC)
    return CC, numero_CC + 1

# Liste des composantes connexes avec pile
def parcours_voisins(L,sommet, CC, numero_CC):
        pile = [sommet]
        while pile != []:
            sommet = pile.pop()
            CC[sommet] = numero_CC
            for voisin in L[sommet]:
                if CC[voisin] < 0:
                    pile.append(voisin)
def listeComposantesConnexesv2(L):
    n = len(L)
    CC = [-1 for _ in range(n)]
    numero_CC = -1
    for sommet in range(n):
        if CC[sommet] < 0:
            numero_CC += 1
            parcours_voisins(L,sommet, CC, numero_CC)
    return CC, numero_CC + 1

#Tests sur le graphe du début de TP
CC = [-1 for _ in range(len(L))]
parcours_voisins(L, 1, CC, 0)
print(CC)     # [0, 0, 0, -1, -1, -1, -1, -1]

parcours_voisins(L, 2, CC, 1)
print(CC)     # [0, 0, 0, 1, 1, 1, 1, 1]

print(listeComposantesConnexesv2(L))
# ([0, 0, 0, 1, 1, 1, 1, 1], 2)

print(listeComposantesConnexes(L))
# ([0, 0, 0, 1, 1, 1, 1, 1], 2)

```

```
66
67
```

```
[0, 0, 0, -1, -1, -1, -1, -1]
[0, 0, 1, -1, -1, -1, -1, -1]
([0, 0, 0, 1, 1, 1, 1, 1], 2)
([0, 0, 0, 1, 1, 1, 1, 1], 2)
```

3. On peut démontrer que si l'on établit au hasard un graphe G ayant n sommets et m arêtes, avec m proche de n, il apparaîtra presque sûrement une composante géante alors que les autres composantes seront soit très petites, soit des sommets isolés. On propose ici d'observer ce phénomène.
   3.1- Écrire une fonction `compte_ComposantesConnexes(CC)` qui prend pour paramètres une liste `CC` comme ci-dessus et qui, en temps linéaire, réalise un affichage au format suivant :
   Il y a 1301 sommets isolés.
   Il y a 185 composantes connexes de taille 2.
   Il y a 60 composantes connexes de taille 3.
   Il y a 24 composantes connexes de taille 4.
   Il y a 4 composantes connexes de taille 5.
   Il y a 4 composantes connexes de taille 6.
   Il y a 5 composantes connexes de taille 7.
   Il y a 1 composante connexe de taille 8.
   Il y a 3 composantes connexes de taille 9.
   Il y a 1 composante connexe de taille 12.
   Il y a 1 composante connexe de taille 7927.

Entrée[ ]:

```python
def compte_ComposantesConnexes(L):
    """ compte_ComposantesConnexes(L : list):
            entrees :L : liste, liste d'adjacence du graphe
        """
    CC, nb_CC = listeComposantesConnexes(L)
    tailleCC = [0 for _ in range(nb_CC)]
    for i in CC:
        tailleCC[i] += 1
    compte = [0 for _ in range(max(tailleCC) + 1)]

    for n in tailleCC:
        compte[n] += 1

    if compte[1] != 0:
        s = 's' if compte[1] != 1 else ''
        print('{0} sommet{1} isolé{1}.'.format(compte[1], s))

    for i in range(2, len(compte)):
        if compte[i] != 0:
            s = 's' if compte[i] > 1 else ''
            print('{0} composante{1} connexe{1} de taille {2}.'.format(compte[i]
```

```python
compte_ComposantesConnexes(L)
# 1 composante connexe de taille 3.
# 1 composante connexe de taille 5.
```

3.2- Vérifier que le phénomène se produit bien en utilisant la fonction `graphe_aleatoire(n, m)` qui constitue un générateur de graphes aléatoires

Entrée[23]:

```
# graphe aléatoire

from random import randint, seed

def graphe_aleatoire(n, m):
    """renvoie la liste d'adjacence d'un graphe aléatoire à
n sommets et m arêtes."""
    seed()
    L = [[] for _ in range(n)]
    nb_aretes = 0
    while nb_aretes < m:
        s1 = randint(0, n - 1)
        s2 = randint(0, n - 1)
        if s1 != s2 and s2 not in L[s1]:
            nb_aretes += 1
            L[s1].append(s2)
            L[s2].append(s1)
    return L
d = 10000
compte_ComposantesConnexes(graphe_aleatoire(d, d+10))
"""
1357 sommets isolés.
179 composantes connexes de taille 2.
50 composantes connexes de taille 3.
18 composantes connexes de taille 4.
10 composantes connexes de taille 5.
4 composantes connexes de taille 6.
3 composantes connexes de taille 8.
1 composante connexe de taille 9.
1 composante connexe de taille 7956.
"""
```

```
Traceback (most recent call last):
  File "<input>", line 20, in <module>
NameError: name 'compte_ComposantesConnexes' is not defined
```

Coloration de graphes

> **Colorier un graphe**
>
> Colorier un graphe consiste à associer une couleur à chacun de ses sommets, de telle manière que deux sommets adjacents (voisins) ne soient jamais de la même couleur.
> Le **nombre chromatique** d'un graphe correspond au nombre minimal de couleurs que l'on peut utiliser pour le colorier. Déterminer ce nombre est un problème informatique difficile : on ne connaît pas d'algorithme général efficace.
> Les problèmes de coloriage de graphes ont de très nombreuses applications : placement d'antennes réseau, ordonnancement de tâches, plan de vols d'une compagnie aérienne, conception d'emploi du temps.

Une couleur sera simplement caractérisée par un entier positif ou nul.

4. Écrire une fonction `degres(M)` qui à partir d'une matrice d'adjacence M, renvoie la liste des degrés de chaque sommet, c'est-à-dire la liste du nombre de voisins de chaque sommet. On pourra utiliser la fonction sum.

Entrée[26]:

```
def degres(M):
    """ degres(M:list)->list
         renvoie la liste des degrés des sommets du graphe dont M est liste d'ad
        entree : M, liste de listes, correspondant a la matrice d'adjacence du g
        sortie : liste correspondant aux degrés de chaque sommet du graphe
    """
    n = len(M)
    return [sum(M[i][j] for j in range(n)) for i in range(n)]

# Tests
M = [[0,1,0,0,0,0,0,0],
     [1,0,1,0,0,0,0,0],
     [0,1,0,0,0,0,0,0],
     [0,0,0,0,1,1,1,0],
     [0,0,0,1,0,0,0,1],
     [0,0,0,1,0,0,0,0],
     [0,0,0,1,0,0,0,0],
     [0,0,0,0,1,0,0,0]]

print(degres(M))
# [1, 2, 1, 3, 2, 1, 1, 1]
```

```
[1, 2, 1, 3, 2, 1, 1, 1]
```

5. Écrire une fonction `ordre_sommets(M)` renvoyant la liste des sommets triés par ordre décroissant de degré.
   Ici, on n'impose pas la méthode de tri : utiliser son tri de prédilection, éventuellement la fonction sorted.

Entrée[28]:

```python
def ordre_sommets(M):
    """ ordre_sommets(M:list)->list
        renvoie la liste des sommets triée par ordre croissant de degré.
        entree : M, liste de listes, correspondant a la matrice d'adjacence du g
        sortie : liste des sommets du graphe triée par ordre décroissant de degr
    """    deg = degres(M)
    n = len(M)

    # Tri rapide en place : on n'a pas choisi la facilité !
    def partition(T, debut, fin):
        "partitionnement pour tri rapide"""
        pivot = T[debut]
        finpp = debut
        for i in range(debut + 1, fin):
            if deg[T[i]] > deg[pivot]:
                finpp += 1
                T[finpp], T[i] = T[i], T[finpp]
        T[debut], T[finpp] = T[finpp], T[debut]
        return finpp

    def tri_rapide(T, debut, fin):
        "Tri en place de T[debut:fin] par ordre décroissant de degrés"
        if debut < fin:
            finpp = partition(T, debut, fin)
            tri_rapide(T, debut, finpp)
            tri_rapide(T, finpp + 1, fin)

    sommets = [i for i in range(n)]
    tri_rapide(sommets, 0, len(sommets))
    return sommets

# Alternative avec sorted :
def ordre_sommets(M):
    "renvoie la liste des sommets triée par ordre croissant de degré."
    n, deg = len(M), degres(M)
    return sorted([i for i in range(n)], key=lambda x: deg[x], reverse=True)
# Tests
print(ordre_sommets(M))
# [3, 1, 4, 0, 2, 5, 6, 7]
```

```
  File "<input>", line 6
    """ ordre_sommets(M:list)->list
        renvoie la liste des sommets triée par ordre croissant de degré.
        entree : M, liste de listes, correspondant a la matrice d'adjacence du grap
he
        sortie : liste des sommets du graphe triée par ordre décroissant de degré
    """    deg = degres(M)
           ^^^
SyntaxError: invalid syntax
```

6. On suppose avoir partiellement colorié le graphe de matrice M et posséder une liste `couleur` contenant, pour chaque sommet d'indice i, -1 s'il n'a pas été colorié et, sinon, sa couleur (qui est un entier entre 0 et n-1 où n correspond au nombre de sommets). `couleur` est donc une liste de longueur n.

Écrire une fonction `couleur_voisins(M, sommet, couleur)` renvoyant une liste de n booléens telle que l'élément d'indice i est à True si au moins un des voisins de `sommet` a la couleur i, et à False dans le cas contraire.

Entrée[37]:

```python
def couleur_voisins(M, sommet, couleur):
    """ couleur_voisins(M: list, sommet:int, couleur:list)->list
    renvoie une liste de n booléens telle que l'élément
    d'indice i est à True si au moins un des voisins de sommet a la
    couleur i, et à False dans le cas contraire.
        entrees : M, liste de listes, correspondant a la matrice d'adjacence du
                : sommet, entier, numero du sommet
                : couleur, liste des couleurs des sommets. La couleur est a -1 s
        sortie : liste_couleurs, liste de booleens
    """
    n = len(M)
    # booléen pour savoir si une couleur est utilisée par un voisin
    liste_couleur = [False for _ in range(n)]
    for voisin in range(n):
        if M[sommet][voisin] == 1 and couleur[voisin] != -1:
            liste_couleur[couleur[voisin]] = True
    return liste_couleur

# Tests
couleur = [-1]*len(M)
print( couleur_voisins(M, 0, couleur))
#[False, False, False, False, False, False, False, False]
couleur[1]=2
print( couleur_voisins(M, 0, couleur))
#[False, False, False, False, False, False, False, False]
print( couleur_voisins(M, 5, couleur))
#[False, True, False, False, False, False, False, False]

```

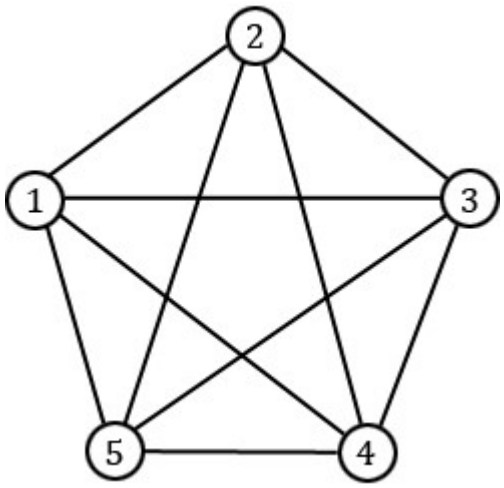
```
[False, False, False, False, False, False, False, False]
[False, False, True, False, False, False, False, False]
[False, False, False, False, False, False, False, False]
```

7. Écrire un **algorithme glouton de coloration** :

Le principe est de colorier un sommet en prenant la couleur de numéro minimal qui ne soit pas déjà utilisée par l'un des sommets, en procédant par ordre décroissant de degrés. On renverra la liste des couleurs de chaque sommet, qui pourra avoir été initialisée avec des valeurs -1 au début de la fonction.

Tester par exemple avec le graphe du départ et avec des graphes complets (dans lesquels tous les sommets sont adjacents).

Exemple de graphe complet :



Entrée[ ]:

```python
def color_glouton(M):
    n = len(M)
    couleur = [-1 for _ in range(n)]
    sommets = ordre_sommets(M) #tri des sommets par ordre décroissant

    for s in sommets:
        liste_couleur = couleur_voisins(M, s, couleur)

        # Détermination de la couleur minimale non utilisée
        mini = 0
        while liste_couleur[mini]:
            mini +=1
        couleur[s] = mini

    return couleur

print(color_glouton(M))
# [1, 0, 1, 0, 1, 1, 1, 0]


def K(n):
    """Renvoie le graphe complet K_n"""
    M = [[1 for i in range(n)] for j in range(n)]
    for i in range(n):
        M[i][i] = 0
    return M

print(color_glouton(K(10)))
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

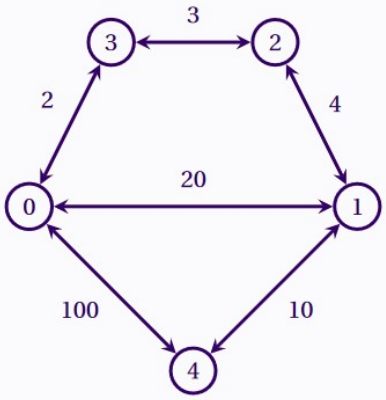
Algorithme de Dijkstra

Graphe orienté pondéré

Un graphe orienté pondéré est un triplet (S , A, p) tel que :

- S est un ensemble dont les éléments sont les sommets du graphe,
- A est un ensemble de couples de sommets, appelés arêtes orientées ou arcs du graphe (cette fois l'ordre est important).
- p est une application définie sur A à valeur dans R : chaque arc (a , b) est associé à un poids p(a , b). On le représente par une matrice d'adjacence : une matrice M tel que Mi, j vaut p(si , sj) s'il y a une arête entre le sommet n°i et le sommet n°j et ∞ sinon.
  On appelle poids d'un chemin dans le graphe la somme des poids des arêtes qui le compose, ce que l'on note (abusivement) p(s0 , . . . , sn) = ∑ p(si ,si+1) pour i de 0 à n.

Exemple



Matrice d'adjacence :

$$M = \begin{pmatrix} \infty & 20 & \infty & 2 & 100 \\ 20 & \infty & 4 & \infty & 10 \\ \infty & 4 & \infty & 3 & \infty \\ 2 & \infty & 3 & \infty & \infty \\ 100 & 10 & \infty & \infty & \infty \end{pmatrix}$$

(Dans l'implémentation, pour ∞, il suffit de prendre une valeur strictement plus grande que la somme

des poids).
On cherche, étant donnés 2 sommets i et j , à déterminer un chemin de poids minimal reliant i à j , avec l'algorithme de Dijkstra, valable seulement lorsque tous les poids sont positifs ou nuls.
Il repose sur le constat que si (s0, . . . , sN) est un plus court chemin, alors pour tout 0 ⩽ p ⩽ q ⩽ N , (sp , . . . , sq) en est un aussi.
Le principe est alors de réaliser une partition des sommets S = Svisites U Sa_visiter avec Svisites initialisé à {s0}, qui grossit, et tel que :

- on connaît la distance minimale de s0 à chaque sommet de Svisites ,
- pour chaque sommet s de Sa_visiter , on connaît la distance minimale (éventuellement infinie) de s0 à s en ne passant que par des sommets de Svisites.

À chaque étape,

- on choisit un sommet smini de Sa_visiter dont la distance à s0 est minimale,
- on le bascule dans Svisites (donc on le supprime de Sa_visiter),
- Pour chaque arête (smini , s) avec s ∈ Sa_visiter , on met à jour la distance de s0 à s via Svisites : d (s) ← min(d (s), d (smini) + p (smini , s))

Dans l'implémentation, il nous suffira de gérer une liste correspondant à Sa_visiter et un tableau de distance d à s0 via Svisites .
À la fin de l'algorithme, d contiendra les valeurs des plus courts chemins de s0 à n'importe quel sommet du graphe. Pour pouvoir retrouver les chemin de longueur minimale permettant d'aller de s0 à un sommet s donner, il faut de plus garder en mémoire dans un tableau predecesseur le dernier sommet ayant permis une mise à jour de d(s) : on calcule alors le chemin à l'envers en partant de s et en remontant tous les prédécesseurs jusqu'au revenir à s0.

8- Décrire l'algorithme pour le graphe donné en exemple :

$S_{visites}$	$d(0)$	$d(1)$	$d(2)$	$d(3)$	$d(4)$
$\{\}$	0	∞	∞	∞	∞
$\{0\}$		20	∞	2	100
$\{0,3\}$		20	5		100

et retrouver les chemins de longueur minimale du sommet 0 à n'importe quel sommet.

9- Écrire une fonction `sommet_mini(S_a_visiter, d)` renvoyant un sommet de `S_a_visiter` correspondant à un élément minimal de la liste de distances `d` .

Entrée[4]:

```
import numpy as np
infini = np.inf

M2 = [[infini, 20, infini, 2, 100],
      [20, infini, 4, infini, 10],
      [infini, 4, infini, 3, infini],
      [2, infini, 3, infini, infini],
      [100, 10, infini, infini, infini]]

def sommet_mini(S2, d):
```

```
  File "<input>", line 10
    def sommet_mini(S2, d):
                            ^
SyntaxError: incomplete input
```

Entrée[14]:

```
## Dijkstra

infini = float('inf')

M2 = [[infini, 20, infini, 2, 100],
      [20, infini, 4, infini, 10],
      [infini, 4, infini, 3, infini],
      [2, infini, 3, infini, infini],
      [100, 10, infini, infini, infini]]

def sommet_mini(S2, d):
    """ sommet_mini(S2, d) -> int
        entree : S2, liste de numeros des sommets à visiter
               : d, liste de distances
        sortie : mini, numero du sommet pour lequel la distane est minimale
    """
    mini = S2[0]
    for s in S2:
        if d[s] < d[mini]:
            mini = s
    return mini

import numpy as np
S_a_visiter=[1, 2, 4]
d=[0, 20, 5, 2, 100]

print( sommet_mini(S_a_visiter, d)) #2
```

```
2
```

10- Écrire une fonction `Dijkstra(M, depart)` renvoyant le tableau des plus courts chemins à partir du sommet `depart` .
Compléter cette fonction pour qu'elle retourne en plusune liste contenant un chemin réalisant ce minimum jusqu'à chaque sommet du graphe (on pourra laisser le chemin dans l'ordre inverse).

Entrée[6]:

```
def Dijkstra(M, debut):
```

```
  File "<input>", line 1
    def Dijkstra(M, debut):
                            ^
SyntaxError: incomplete input
```

Entrée[16]:

```
def Dijkstra(M, debut):
    """ Dijkstra(M, debut)-> list
        entrees  : M, liste de listes, matrice d'adjacence du graphe
                 : debut, entier, numero du sommet de depart
        sortie : d, liste des distances les plus courtes entre debut et tous les
    """
    n = len(M)
    S_a_visiter = [i for i in range(n)] # contient les sommets encore à visiter

    d = [infini for sommet in range(n)]
    d[debut] = 0
    while len(S_a_visiter) != 0: # Calcul des distances minimales

        s_mini = sommet_mini(S_a_visiter, d)

        S_a_visiter = [i for i in S_a_visiter if i != s_mini]
        for s2 in S_a_visiter:
            d[s2] = min(d[s2], d[s_mini] + M[s_mini][s2])
    return d
print(Dijkstra(M2, 0))
#[0, 9, 5, 2, 19]
print(Dijkstra(M2, 1))
#[9, 0, 4, 7, 10]

# Pour renvoyer un chemin réalisant la distance minimale entre le sommet debut e
# sommet donné, il faudrait à chaque étape retenir le prédécesseur d'un sommet,
# le dernier sommet à partir duquel la mise à jour de d a été faite.
# On reconstruit alors le chemin à partir du sommet final en prenant à chaque ét
# prédécesseur.

def Dijkstra_avec_chemin(M, debut):
    """ Dijkstra(M, debut)-> list
        entrees  : M, liste de listes, matrice d'adjacence du graphe
                 : debut, entier, numero du sommet de depart
        sorties : d, liste des distances les plus courtes entre debut et tous le
                 : chemin, liste de liste de chemins
    """
    n = len(M)
    S_a_visiter = [i for i in range(n)] # contient les sommets encore à visiter
    d = [infini for sommet in range(n)]
    predecesseur = [debut for _ in range(n)]
    d[debut] = 0

    while S_a_visiter != []: # Calcul des distances minimales avec prédécesseurs
        s_mini = sommet_mini(S_a_visiter, d) #s1 dans l'énoncé
        S_a_visiter = [i for i in S_a_visiter if i != s_mini]
        for s2 in S_a_visiter:
            distance = d[s_mini] + M[s_mini][s2]
            if distance < d[s2]:
                d[s2] = distance
                predecesseur[s2] = s_mini

    chemin = [[] for _ in range(n)] # Construction des chemins minimaux à rebour
    for sommet in range(n):
        if d[sommet] != infini: # S'il y a un chemin !
            precedent = sommet
            chemin[sommet].append(precedent)
            while precedent != debut:
                precedent = predecesseur[precedent]
                chemin[sommet].append(precedent)
    return d, chemin # les chemins sont à l'envers : de la fin au début.
print(Dijkstra_avec_chemin(M2, 0))
#([0, 9, 5, 2, 19], [[0], [1, 2, 3, 0], [2, 3, 0], [3, 0], [4, 1, 2, 3, 0]])
```

```
[0, 9, 5, 2, 19]
[9, 0, 4, 7, 10]
([0, 9, 5, 2, 19], [[0], [1, 2, 3, 0], [2, 3, 0], [3, 0], [4, 1, 2, 3, 0]])
```