

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

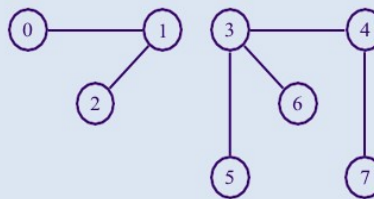
Un **graphe non orienté** est un couple (S, A) tel que :

- 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 $M_{i,j}$ vaut 1 s'il y a une arête entre le sommet $n^o i$ et le sommet $n^o j$ et 0 sinon.
C'est intéressant lorsque le graphe est dense (beaucoup d'arêtes).

Exemple :



$S = \llbracket 0, 7 \rrbracket$ 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],  
    [0,0,0,1,1,1,0,0],  
    [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 transforme en dictionnaire d'adjacence.

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

Entrée[17]: ▶

```
1 def genereDico(L):
2     """ genereDico(L: list) -> dict
3         entrée : L, liste correspondant la liste d'adjacence du graphe
4         sortie : dico, dictionnaire correspondant au dictionnaire d'adjacence d
5     """
6     dico={}
7     for i in range(len(L)) :
8         dico[i] = []
9         for j in range(len(L[i])):
10             dico[i].append(L[i][j])
11     return dico
12
13 def genereMatrice(L):
14     """ genereDico(L: list) -> list
15         entrée : L, liste correspondant la liste d'adjacence du graphe
16         sortie : mat, liste de listes, correspondant a la matrice d'adjacence d
17     """
18     n = len(L)
19     # mat=[[0]*n]*n
20     mat = [ [0 for j in range(n)] for i in range( n ) ]
21     print(mat)
22     for i in range(len(L)) :
23         for val in L[i]:
24             mat[i][val] = 1
25     return mat
26
27 # L: liste d'adjacence du graphe d'exemple
28 L = [[1], [2, 0], [1], [6, 4, 5], [7, 3], [3], [3], [4]]
29 print( genereDico(L) )
30 # {0: [1], 1: [2, 0], 2: [1], 3: [6, 4, 5], 4: [7, 3], 5: [3], 6: [3], 7: [4]}
31 print( genereMatrice(L) )
32 # [[0, 1, 0, 0, 0, 0, 0, 0], [1, 0, 1, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0, 0, 0],
33
34 {0: [1], 1: [2, 0], 2: [1], 3: [6, 4, 5], 4: [7, 3], 5: [3], 6: [3], 7: [4]}
35 [[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,
36 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,
37 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0]]
38 [[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,
39 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,
40 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 :

```

1 parcourir(L)
2   Pour chaque sommet faire
3     Si le sommet n'est pas marqué comme visité alors
4       parcourir_voisins(L,sommet)
5     finsi
6   finpour
7
8 avec parcourir_voisins pouvant être définie de 2 manières : récursive ou itérative
9 ##### Version récursive :
10 parcourir_voisins(L, sommet)
11   Marquer le sommet comme visité
12   Traiter le sommet
13   Pour chaque voisin du sommet faire
14     Si le voisin n'est pas marqué comme visité alors
15       parcourir_voisins(L,voisin)
16     finsi
17   finpour
18
19 ##### Version avec une pile :
20 parcourir_voisins(L,sommet)
21   Créer une pile
22   Marquer le sommet comme visité
23   Empiler le sommet
24   Tant que la pile n'est pas vide faire
25     Dépiler un sommet
26     Traiter le sommet
27     Pour chaque voisin du sommet faire
28       Si le voisin n'est pas marqué comme visité alors
29         Marquer le voisin comme visité
30         Empiler le voisin
31     finsi
32   finpour
33   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- Définir 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 fonction `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]: ▶

```
1 # TP graphes
2
3 ## Parcours en profondeur
4 def parcours_voisins_rec(L, sommet, visite, parcours):
5     """parcours récursif des voisins non visités de sommet"""
6     """ parcours_voisins_rec(L : list, sommet: int, visite: list, parcours:
7         entrees :L : liste, liste d'adjacence du graphe
8             sommet : entier, Numero du sommet dont on part
9             visite : liste de booléens qui indique pour chaque sommet(ir
10                 s'il a déjà été visité (True ou False), modifiée par effe
11             parcours : liste des sommets visités (dans l'ordre de la vis
12     """
13     visite[sommet] = True
14     parcours.append(sommet)
15     lesVoisins = L[sommet]
16     for voisin in lesVoisins :
17         if not visite[voisin]:
18             parcours_voisins_rec(L,voisin, visite, parcours)
19
20 """Parcours en profondeur itératif"""
21 from collections import deque
22 def parcours_voisins_pile(L, sommet, visite, parcours):
23     """parcours des voisins non visités de sommet à l'aide d'une pile"""
24     pile = [sommet]
25     visite[sommet] = True
26     while len(pile) != 0:
27         sommet1 = pile.pop()
28         parcours.append(sommet1)
29         for voisin in L[sommet1]:
30             if not visite[voisin]:
31                 visite[voisin] = True
32                 pile.append(voisin)
33
34 # version iterative en utilisant les deque en python
35 from collections import deque
36 def parcours_voisins_pile(L, sommet, visite, parcours):
37     """parcours des voisins non visités de sommet à l'aide d'une pile"""
38     pile = deque([sommet])
39     visite[sommet] = True
40     while len(pile) != 0:
41         sommet1 = pile.pop()
42         parcours.append(sommet1)
43         lesVoisins = L[sommet1]
44         for voisin in lesVoisins:
45             if not visite[voisin]:
46                 visite[voisin] = True
47                 pile.append(voisin)
48
49 def parcourir(L, fonctionParcours = parcours_voisins_rec):
50     """Parcours en profondeur des sommets du graphe"""
51     """ parcourir(L : list)-> list
52         entrees : L, liste, liste d'adjacence du graphe
53             : fonctionParcours, nom de la fonction à appeler pour le par
54         sortie : parcours, liste des sommets visités (dans l'ordre de la vis
55     """
56     n = len(L)
57     visite = [False for _ in range(n)] # permet de savoir si un sommet a été vis
58     parcours = [] # permet de garder les sommets dans l'ordre de parcours
59     for sommet in range(n):
60         if not visite[sommet]:
61             fonctionParcours(L, sommet, visite, parcours)
62     return parcours
63
64
65 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]: ▶

```
1  ## Composantes connexes
2
3  # Liste des composantes connexes récursif
4  def parcours_voisins(L, sommet, CC, numero_CC):
5      """ parcours_voisins(L : list, sommet: int, CC: list, numero_CC:int):
6          entrees :L : liste, liste d'adjacence du graphe
7              sommet : entier, Numero du sommet dont on part
8              CC : liste initialisée avec des valeurs -1,
9                  telle que CC[sommet] est le numéro de la composante
10                 modifiée par effet de bord
11                 numero_CC : int, numéro de la composante connexe
12
13             """
14             CC[sommet] = numero_CC
15             for voisin in L[sommet]:
16                 if CC[voisin] < 0:
17                     parcours_voisins(voisin, CC, numero_CC)
18
19 def listeComposantesConnexes(L):
20     """ listeComposantesConnexes(L : list) -> list, int:
21         entrees :L : liste, liste d'adjacence du graphe
22         sorties :CC : liste initialisée avec des valeurs -1,
23             telle que CC[sommet] est le numéro de la composante
24             int, nombre de composantes connexes du graphe
25
26     """
27     n = len(L)
28     CC = [-1 for _ in range(n)]
29     numero_CC = -1
30     for sommet in range(n):
31         if CC[sommet] < 0:
32             numero_CC += 1
33             parcours_voisins(L,sommet, CC, numero_CC)
34     return CC, numero_CC + 1
35
36 # Liste des composantes connexes avec pile
37 def parcours_voisins(L,sommet, CC, numero_CC):
38     pile = [sommet]
39     while pile != []:
40         sommet = pile.pop()
41         CC[sommet] = numero_CC
42         for voisin in L[sommet]:
43             if CC[voisin] < 0:
44                 pile.append(voisin)
45
46 def listeComposantesConnexesv2(L):
47     n = len(L)
48     CC = [-1 for _ in range(n)]
49     numero_CC = -1
50     for sommet in range(n):
51         if CC[sommet] < 0:
52             numero_CC += 1
53             parcours_voisins(L,sommet, CC, numero_CC)
54     return CC, numero_CC + 1
55
56 #Tests sur Le graphe du début de TP
57 CC = [-1 for _ in range(len(L))]
58 parcours_voisins(L, 1, CC, 0)
59 print(CC)      # [0, 0, 0, -1, -1, -1, -1, -1]
60
61 parcours_voisins(L, 2, CC, 1)
62 print(CC)      # [0, 0, 0, 1, 1, 1, 1, 1]
63
64 print(listeComposantesConnexesv2(L))
65 # ([0, 0, 0, 1, 1, 1, 1, 1], 2)
66
67 print(listeComposantesConnexes(L))
68 # ([0, 0, 0, 1, 1, 1, 1, 1], 2)
69
```

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[]: ▶

```
1 def compte_ComposantesConnexes(L):
2     """ compte_ComposantesConnexes(L : list):
3         entrees :L : liste, liste d'adjacence du graphe
4         """
5     CC, nb_CC = listeComposantesConnexes(L)
6     tailleCC = [0 for _ in range(nb_CC)]
7     for i in CC:
8         tailleCC[i] += 1
9     compte = [0 for _ in range(max(tailleCC) + 1)]
10
11     for n in tailleCC:
12         compte[n] += 1
13
14     if compte[1] != 0:
15         s = 's' if compte[1] != 1 else ''
16         print('{0} sommet{1} isolé{1}'.format(compte[1], s))
17
18     for i in range(2, len(compte)):
19         if compte[i] != 0:
20             s = 's' if compte[i] > 1 else ''
21             print('{0} composante{1} connexe{1} de taille {2}'.format(compte[i], s, i))
22
23 compte_ComposantesConnexes(L)
24 # 1 composante connexe de taille 3.
```

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]:

```
1 # graphe aléatoire
2
3 from random import randint, seed
4
5 def graphe_aleatoire(n, m):
6     """renvoie la liste d'adjacence d'un graphe aléatoire à
7     n sommets et m arêtes."""
8     seed()
9     L = [[] for _ in range(n)]
10    nb_aretes = 0
11    while nb_aretes < m:
12        s1 = randint(0, n - 1)
13        s2 = randint(0, n - 1)
14        if s1 != s2 and s2 not in L[s1]:
15            nb_aretes += 1
16            L[s1].append(s2)
17            L[s2].append(s1)
18    return L
19
20 d = 10000
21 compte_ComposantesConnexes(graphe_aleatoire(d, d+10))
22 """
23 1357 sommets isolés.
24 179 composantes connexes de taille 2.
25 50 composantes connexes de taille 3.
26 18 composantes connexes de taille 4.
27 10 composantes connexes de taille 5.
28 4 composantes connexes de taille 6.
29 3 composantes connexes de taille 8.
30 1 composante connexe de taille 9.
31 1 composante connexe de taille 7956.
32 """
```

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]: ▶

```
1 def degres(M):
2     """ degres(M:list)->list
3         renvoie la liste des degrés des sommets du graphe dont M est liste d'adjacence
4         entree : M, liste de listes, correspondant a la matrice d'adjacence du graphe
5         sortie : liste correspondant aux degrés de chaque sommet du graphe
6     """
7     n = len(M)
8     return [sum(M[i][j] for j in range(n)) for i in range(n)]
9
10 # Tests
11 M = [[0,1,0,0,0,0,0,0],
12      [1,0,1,0,0,0,0,0],
13      [0,1,0,0,0,0,0,0],
14      [0,0,0,0,1,1,1,0],
15      [0,0,0,1,0,0,0,1],
16      [0,0,0,1,0,0,0,0],
17      [0,0,0,1,0,0,0,0],
18      [0,0,0,0,1,0,0,0]]
19
20 print(degres(M))
21 [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]: ▶

```
1 def ordre_sommets(M):
2     """ ordre_sommets(M:list)->list
3         renvoie la liste des sommets triée par ordre croissant de degré.
4         entree : M, liste de listes, correspondant a la matrice d'adjacence du g
5         sortie : liste des sommets du graphe triée par ordre décroissant de degr
6     """
7     deg = degrees(M)
8     n = len(M)
9
10    # Tri rapide en place : on n'a pas choisi la facilité !
11    def partition(T, debut, fin):
12        "partitionnement pour tri rapide"
13        pivot = T[debut]
14        finpp = debut
15        for i in range(debut + 1, fin):
16            if deg[T[i]] > deg[pivot]:
17                finpp += 1
18                T[finpp], T[i] = T[i], T[finpp]
19        T[debut], T[finpp] = T[finpp], T[debut]
20        return finpp
21
22    def tri_rapide(T, debut, fin):
23        "Tri en place de T[debut:fin] par ordre décroissant de degrés"
24        if debut < fin:
25            finpp = partition(T, debut, fin)
26            tri_rapide(T, debut, finpp)
27            tri_rapide(T, finpp + 1, fin)
28
29    sommets = [i for i in range(n)]
30    tri_rapide(sommets, 0, len(sommets))
31    return sommets
32
33 # Alternative avec sorted :
34 def ordre_sommets(M):
35     "renvoie la liste des sommets triée par ordre croissant de degré."
36     n, deg = len(M), degrees(M)
37     return sorted([i for i in range(n)], key=lambda x: deg[x], reverse=True)
38
39 # Tests
40 print(ordre_sommets(M))
41 # [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 = degrees(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]: ▶

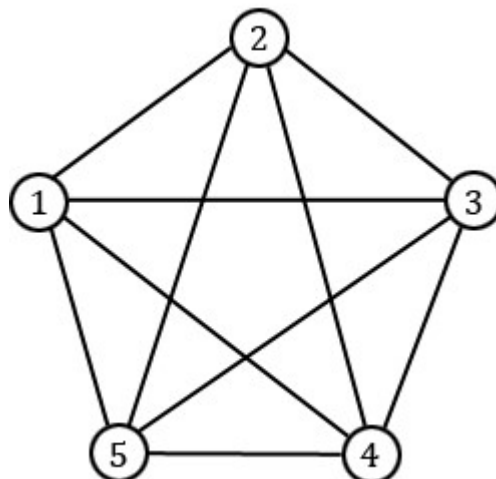
```
1 def couleur_voisins(M, sommet, couleur):
2     """ couleur_voisins(M: list, sommet:int, couleur:list)->list
3     renvoie une liste de n booléens telle que l'élément
4     d'indice i est à True si au moins un des voisins de sommet a la
5     couleur i, et à False dans le cas contraire.
6     entrees : M, liste de listes, correspondant a la matrice d'adjacence du
7               : sommet, entier, numero du sommet
8               : couleur, liste des couleurs des sommets. La couleur est a -1 s
9     sortie : liste_couleurs, liste de booléens
10    """
11    n = len(M)
12    # booléen pour savoir si une couleur est utilisée par un voisin
13    liste_couleur = [False for _ in range(n)]
14    for voisin in range(n):
15        if M[sommet][voisin] == 1 and couleur[voisin] != -1:
16            liste_couleur[couleur[voisin]] = True
17    return liste_couleur
18
19 # Tests
20 couleur = [-1]*len(M)
21 print( couleur_voisins(M, 0, couleur))
22 #[False, False, False, False, False, False, False, False]
23 couleur[1]=2
24 print( couleur_voisins(M, 0, couleur))
25 #[False, False, False, False, False, False, False, False]
26 print( couleur_voisins(M, 5, couleur))
27 #[False, True, False, False, False, False, False, False]
28
[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[]: ▶

```
1 def color_glouton(M):
2     n = len(M)
3     couleur = [-1 for _ in range(n)]
4     sommets = ordre_sommets(M) #tri des sommets par ordre décroissant
5
6     for s in sommets:
7         liste_couleur = couleur_voisins(M, s, couleur)
8
9         # Détermination de la couleur minimale non utilisée
10        mini = 0
11        while liste_couleur[mini]:
12            mini += 1
13        couleur[s] = mini
14
15    return couleur
16
17 print(color_glouton(M))
18 # [1, 0, 1, 0, 1, 1, 1, 0]
19
20
21 def K(n):
22     """"Renvoie le graphe complet K_n"""
23     M = [[1 for i in range(n)] for j in range(n)]
24     for i in range(n):
25         M[i][i] = 0
26     return M
27
28 print(color_glouton(K(10)))
```

Algorithme de Dijkstra

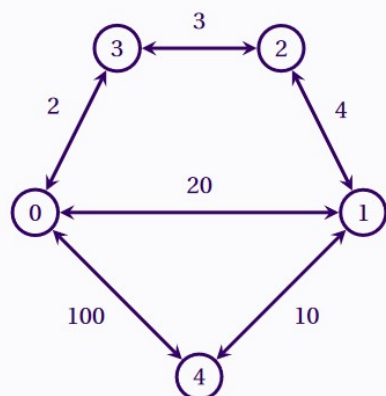
Graphe orienté pondéré

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 $M_{i,j}$ vaut $p(s_i, s_j)$ s'il y a une arête entre le sommet $n^o i$ et le sommet $n^o j$ 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) $p(s_0, \dots, s_n) = \sum p(s_i, s_{i+1})$ 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 $(s_0, \dots, s_N)$ est un plus court chemin, alors pour tout $0 \leq p \leq q \leq N$, $(s_p, \dots, s_q)$ en est un aussi.

Le principe est alors de réaliser une partition des sommets $S = S_{visites} \cup S_{a_visiter}$ avec $S_{visites}$ initialisé à $\{s_0\}$, qui grossit, et tel que :

- on connaît la distance minimale de s_0 à chaque sommet de $S_{visites}$,
- pour chaque sommet s de $S_{a_visiter}$, on connaît la distance minimale (éventuellement infinie) de s_0 à s en ne passant que par des sommets de $S_{visites}$.

À chaque étape,

- on choisit un sommet s_{mini} de $S_{a_visiter}$ dont la distance à s_0 est minimale,
- on le bascule dans $S_{visites}$ (donc on le supprime de $S_{a_visiter}$),
- Pour chaque arête (s_{mini}, s) avec $s \in S_{a_visiter}$, on met à jour la distance de s_0 à s via $S_{visites}$:
 $d(s) \leftarrow \min(d(s), d(s_{mini}) + p(s_{mini}, s))$

Dans l'implémentation, il nous suffira de gérer une liste correspondant à $S_{a_visiter}$ et un tableau de distance d à s_0 via $S_{visites}$.

À la fin de l'algorithme, d contiendra les valeurs des plus courts chemins de s_0 à n'importe quel sommet du graphe. Pour pouvoir retrouver le chemin de longueur minimale permettant d'aller de s_0 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 s_0 .

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]: ▶

```
1 import numpy as np
2 infini = np.inf
3
4 M2 = [[infini, 20, infini, 2, 100],
5       [20, infini, 4, infini, 10],
6       [infini, 4, infini, 3, infini],
7       [2, infini, 3, infini, infini],
8       [100, 10, infini, infini, infini]]
9
10 def sommet_mini(S2, d):
11
```

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

SyntaxError: incomplete input

Entrée[14]: ▶

```
1  ## Dijkstra
2
3  infini = float('inf')
4
5  M2 = [[infini, 20, infini, 2, 100],
6        [20, infini, 4, infini, 10],
7        [infini, 4, infini, 3, infini],
8        [2, infini, 3, infini, infini],
9        [100, 10, infini, infini, infini]]
10
11 def sommet_mini(S2, d):
12     """ sommet_mini(S2, d) -> int
13         entree : S2, liste de numeros des sommets à visiter
14             : d, liste de distances
15         sortie : mini, numero du sommet pour lequel la distane est minimale
16     """
17     mini = S2[0]
18     for s in S2:
19         if d[s] < d[mini]:
20             mini = s
21     return mini
22
23 import numpy as np
24 S_a_visiter=[1, 2, 4]
25 d=[0, 20, 5, 2, 100]
26
27 print( sommet_mini(S_a_visiter, d)) #2
28
```

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 plus une liste contenant un chemin réalisant ce minimum jusqu'à chaque sommet du graphe (on pourra laisser le chemin dans l'ordre inverse).

Entrée[6]: ▶

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

SyntaxError: incomplete input

Entrée[16]: ▶

```
1
2
3 def Dijkstra(M, debut):
4     """ Dijkstra(M, debut)-> list
5         entrees : M, liste de listes, matrice d'adjacence du graphe
6                 : debut, entier, numero du sommet de depart
7         sortie : d, liste des distances les plus courtes entre debut et tous les
8     """
9     n = len(M)
10    S_a_visiter = [i for i in range(n)] # contient les sommets encore à visiter
11
12    d = [infini for sommet in range(n)]
13    d[debut] = 0
14    while len(S_a_visiter) != 0: # Calcul des distances minimales
15
16        s_mini = sommet_mini(S_a_visiter, d)
17
18        S_a_visiter = [i for i in S_a_visiter if i != s_mini]
19        for s2 in S_a_visiter:
20            d[s2] = min(d[s2], d[s_mini] + M[s_mini][s2])
21    return d
22 print(Dijkstra(M2, 0))
23 #[0, 9, 5, 2, 19]
24 print(Dijkstra(M2, 1))
25 #[9, 0, 4, 7, 10]
26
27 # Pour renvoyer un chemin réalisant la distance minimale entre Le sommet debut et
28 # sommet donné, il faudrait à chaque étape retenir le prédécesseur d'un sommet,
29 # le dernier sommet à partir duquel la mise à jour de d a été faite.
30 # On reconstruit alors le chemin à partir du sommet final en prenant à chaque ét
31 # prédécesseur.
32
33 def Dijkstra_avec_chemin(M, debut):
34     """ Dijkstra(M, debut)-> list
35         entrees : M, liste de listes, matrice d'adjacence du graphe
36                 : debut, entier, numero du sommet de depart
37         sorties : d, liste des distances les plus courtes entre debut et tous les
38                 : chemin, liste de liste de chemins
39     """
40    n = len(M)
41    S_a_visiter = [i for i in range(n)] # contient les sommets encore à visiter
42    d = [infini for sommet in range(n)]
43    predecesseur = [debut for _ in range(n)]
44    d[debut] = 0
45
46    while S_a_visiter != []: # Calcul des distances minimales avec prédécesseurs
47        s_mini = sommet_mini(S_a_visiter, d) #s1 dans l'énoncé
48        S_a_visiter = [i for i in S_a_visiter if i != s_mini]
49        for s2 in S_a_visiter:
50            distance = d[s_mini] + M[s_mini][s2]
51            if distance < d[s2]:
52                d[s2] = distance
53                predecesseur[s2] = s_mini
54
55    chemin = [[] for _ in range(n)] # Construction des chemins minimaux à rebours
56    for sommet in range(n):
57        if d[sommet] != infini: # S'il y a un chemin !
58            precedent = sommet
59            chemin[sommet].append(precedent)
60            while precedent != debut:
61                precedent = predecesseur[precedent]
62                chemin[sommet].append(precedent)
63    return d, chemin # Les chemins sont à l'envers : de la fin au début.
64 print(Dijkstra_avec_chemin(M2, 0))
65 #([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]])