

Fiche 9- Proposition de solutions

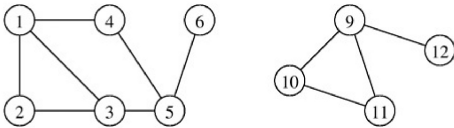
Solution 1

Considérant G_3 :

- L'ordre est : 11
- Une chaîne de longueur 5 entre s_8 et s_9 est

$$((s_8, s_6), (s_6, s_2), (s_2, s_1), (s_1, s_{10}), (s_{10}, s_9))$$

- Un cycle est $((s_2, s_7), (s_7, s_8), (s_8, s_6), (s_6, s_2))$
- Des degrés : $d(s_8) = 2$ $d(s_9) = 3$ et $d(s_1) = 4$
- G_3 est connexe ; le graphe suivant n'est pas connexe :



Solution 2

Considérant G_4 :

- Le chemin $((s_1, s_3), (s_3, s_4), (s_4, s_3), (s_3, s_2))$ est de longueur 4.
- Un circuit : $((s_3, s_4), (s_4, s_3))$
- Des degrés : $d_-(s_2) = 2$ $d_+(s_2) = 0$ $d_-(s_3) = d_+(s_3) = 2$

Solution 3

- Cas d'un graphe non orienté :

$$\sum_{s \in S} d(s) = \text{Card}(A)$$

Par disjonction de cas,

- une boucle compte pour 1 des deux côtés
- une arête comptent pour 2 des deux côtés : 2 liens, 2 voisins

- Cas d'un graphe non orienté :

$$\sum_{s \in S} d_+(s) = \sum_{s \in S} d_-(s) = \text{Card}(A)$$

Solution 4

Sur le graphe G_5 , la chaîne de poids minimal entre s_2 et s_6 est :

$((s_2, s_1), (s_1, s_3), (s_3, s_4), (s_4, s_6))$ de poids 8.

Solution 5 $M_1 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$ et $M_6 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$

Un graphe dont la matrice d'adjacente est symétrique est non orienté.

Solution 6

Procéder par récurrence :

- Initialisation : Pour $k = 1$, c'est la définition de la matrice d'adjacence.
- Hérédité : Soit $k \geq 1$. On suppose la propriété vraie au rang n .

Soit $i, j \in \llbracket 1, n \rrbracket$.

Effectuons une disjonction de cas sur le premier saut, (s_i, s_k) avec $\ell \in \llbracket 1, n \rrbracket$: le nombre de chemins de longueur $k+1$ allant de s_i à s_j avec pour premier arc (s_i, s_ℓ) revient à dénombrer les chemins de longueur k allant de s_ℓ à s_j :

$$\mathcal{C}_{i,\ell}(M) \times \mathcal{C}_{\ell,j}(M^k)$$

Au total cela donne : $\sum_{\ell=1}^n \mathcal{C}_{i,\ell}(M) \times \mathcal{C}_{\ell,j}(M^k)$.

Par définition du produit matriciel, cela donne : $\mathcal{C}_{i,j}(M^{k+1})$

- Conclusion : La propriété est vrai pour tout $k \in \mathbb{N}$.

Solution 7

Chemins de longueurs n

```
def nb_chemins(n,i,j,M):
    A=1*M
    for k in range(2,n+1):
        A=np.dot(A,M)
    return A[i,j]
```

```
>>> M=np.array([[1,1,0],[0,1,1],[0,0,1]])
>>> nb_chemins(50,0,2,M)
1225
```

Un chemin de longueur 50 entre s_0 et s_2 contient 48 boucles et 2 arc. Dénombrer les chemins revient à positionner les 2 arc parmi les 50 liens : $\binom{50}{2} = 1225$.

Solution 8

Graphe de distance

1. La matrice du graphe :

```
import numpy as np
M=np.array([[0,9,3,-1,7],[9,0,1,8,-1],
            [3,1,0,4,2],[-1,8,4,0,-1],[7,-1,2,-1,0]])
```

```
array([[ 0,  9,  3, -1,  7],
       [ 9,  0,  1,  8, -1],
       [ 3,  1,  0,  4,  2],
       [-1,  8,  4,  0, -1],
       [ 7, -1,  2, -1,  0]])
```

2. Les voisins de 4 :

```
L=[]
for i in range(5):
    if M[i,4]>0: L.append(i)
print(L)
```

On trouve :

```
[0, 2]
```

3. La fonction voisins :

```
def voisins(i):
    L=[]
    for k in range(5):
        if M[i,k]>0: L.append(k)
    return L
```

4. La fonction degre :

```
def degre(i):
    return len(voisins(i))
```

5. La fonction longueur :

```
def longueur(L):
    s=0
    for i in range(len(L)-1):
        if M[L[i],L[i+1]]!=-1:
            return -1
        else:
            s+=M[L[i],L[i+1]]
    return s
```

Ce qui donne :

```
>>> longueur([0,2,4,0,1])
21
>>> longueur([0,2,4,0,3])
-1
```

Solution 9 Passage d'une liste d'adjacences à une matrice d'adjacence :

```
def matrice(L):
    M=np.zeros((len(L),len(L)))
    for i in range(len(L)):
        for j in L[i]:
            M[i,j]=1
    return M
```

```
def degre(s,L):
    M=matrice(L)
    return (sum(M[:,s]),sum(M[s,:]))

def degre2(s,L):
    d=0
    for l in L:
        for e in l:
            if e==s:
                d=d+1
    return (d,len(L[s]))
```

Solution 10 Dictionnaire de comptage

```
def comptage(L):
    D={}
    for e in L:
        if e in D: D[e]=D[e]+1
        else: D[e]=1
    return D
```

Solution 11 • Matrices d'adjacence :

$$M_7 = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \text{ et } M_8 = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Pour les repérer, il convient de créer la liste des sommets :

```
S7=[1,2,3,4]
S8=['A','B','C','D','E','F']
```

La taille de la matrice est la même que la graphe soit dense ou creux !

- Liste des listes d'adjacence : ici aussi, il convient au préalable de définir la liste des sommets (S7 et S8),

```
L7=[[2,3,4],[3,4],[4],[]]
L8=[['B','C','D'],['D'],[],['E'],['B','F'],['C']]
```

La taille des listes d'adjacence est égale aux nombres des arcs, ce qui est efficace pour les graphe creux.

- Dictionnaire des listes d'adjacence : ici, il est inutile de définir la liste des sommets car ces derniers sont les clés du dictionnaire,

```
D7={1:[2,3,4],2:[3,4],3:[4],4:[]}
D8={'A':['B','C','D'],'B':['D'],'C':[],'D':['E'],
    'E':['B','F'],'F':['C']}
```

Cette solution est la plus pratique !

Solution 12 Modèle d'une pile bornée

Pile bornée

```
def creer_pile(c):
    """creer un pile de capacite c"""
    return (c+1)*[0]

def depiler(p):
    n=p[0]
    assert n>0,'pile vide'
    p[0]=n-1
    return p[n]

def empiler(p,v):
    n=p[0]
    assert n<len(p)-1,'pile pleine'
    n=n+1
    p[0],p[n]=n,v
```

mais aussi :

```
def taille(p):  
    return p[0]  
  
def est_vide(p):  
    return p[0]==0  
  
def sommet(p):  
    assert p[0]>0, 'pile vide'  
    return p[p[0]]
```

Solution 13 Modèle d'une pile non bornée

```
def creer_pile(c):  
    return []  
  
def depiler(p):  
    assert len(p)>0, 'pile vide'  
    return p.pop()  
  
def empiler(p,v):  
    p.append(v)  
  
def taille(p):  
    return len(p)  
  
def est_vide(p):  
    return len(p)==0  
  
def sommet(p):  
    assert len(p)>0, 'pile vide'  
    return p[-1]
```

Solution 14 Il suffit de calculer les puissances successives de la matrice d'adjacence jusqu'à obtenir un coefficient diagonal non nul. La longueur maximale du plus petit cycle est inférieure à l'ordre du graphe (le nombre de sommets).

```
def detection_cycle(M): # graphe oriente  
    n=len(M)  
    A=1*M  
    for i in range(2,n+1):  
        A=np.dot(A,M)  
        if sum([A[j,j] for j in range(n)])>0:  
            return True  
    return False
```

Solution 15 Détection de cycle dans un graphe orienté

1. Script du programme associé à l'algorithme :

```
def detection_cycle_GO_S(s,M):  
    ## graphe oriente  
    n=len(M) # nombre de sommets  
    file=[s]  
    sommets=n*[False]  
    while len(file)>0:  
        p=file.pop(0)  
        for i in range(n):  
            if M[p][i]>0:  
                if i==s:  
                    return True  
                elif sommets[i]==False:  
                    file.append(i)  
                    sommets[i]=True  
    return False
```

2. Pour répondre à l'objectif, il convient maintenant de tester chaque sommet.

```
def detection_cycle_GO(M):  
    for s in range(len(M)):  
        if detection_cycle_GO_S(s,M):  
            return True  
    return False
```

3. Cette approche effectue un parcours en largeur.

4. Si l'on remplace la *file* en *pile*, le parcours devient en profondeur.

5. Pour retourner un cycle, il suffit de remplacer la liste sommets des sommets atteignables par une liste parent qui contient soit False pour un sommet non atteint, soit un sommet entrant du sommet considéré. Ainsi, dès qu'un cycle est détecté, il convient de remonter la parentalité de *s* à *s* !

```
def recherche_cycle_GO_S(s,M): ## graphe oriente  
    n=len(M) # nombre de sommets  
    file=[s]  
    parent=n*[False]  
    while len(file)>0:  
        p=file.pop(0)  
        for i in range(n):  
            if M[p][i]>0:  
                if i==s:  
                    pp=p  
                    C=[pp,s]  
                    while pp!=s:  
                        pp=parent[pp]  
                        C=[pp]+C  
                    return True,C  
                elif parent[i]==False:  
                    file.append(i)  
                    parent[i]=p  
    return False,[]
```

Et pour conclure :

```
def recherche_cycle_GO(M):
    for s in range(len(M)):
        T,C=recherche_cycle_GO_S(s,M)
        if T:
            return C
    return False
```

Solution 16 Connexité d'un graphe non orienté : on parcourt tous les voisins du sommet initial s_0 en veillant à initialiser la liste sommets avec True en position 0.

A la fin, il suffit de compter le nombre de True dans sommets pour obtenir le nombre de sommets atteignables à partir de s_0 , s'il correspond à l'ordre du graphe alors ce dernier est connexe !

```
def connexe(M):
    n=len(M) # nombre de sommets
    file=[0]
    sommets=[True]+(n-1)*[False]
    while len(file)>0:
        p=file.pop(0)
        for i in range(n):
            if M[p][i]>0 and sommets[i]==False:
                file.append(i)
                sommets[i]=True
    return n==sum(sommets)
```