TD Informatique TSI2

GRAPHS : Traversal algorithms

SOLUTION

EXERCICE 01

```
In [1]:  def parcours_en_largeur(graphe,sommet):
   ...:      file = [sommet]
   ...:      sommets_visites = []
   ...:      while len(file) != 0 :
   ...:          S = file[0] ; print("sommet a explorer:",S)
   ...:          for voisin in graphe[S]:
   ...:              if not(voisin in sommets_visites) and not(voisin in file):
   ...:                  file.append(voisin)
   ...:                  print("Construction file voisins non visited:",file)
   ...:          sommets_visites.append(file.pop(0))
   ...:          print("Sommets visited:",sommets_visites)
   ...:      return sommets_visites
In [2]: graphe = {'A' : ['B','D','E'], 'B' : ['A','C','E'], 'C' : ['B', 'D'],
'D' : ['A','C','G'], 'E' : ['A', 'B', 'F'], 'F' : ['E', 'G'],
'G' : [ 'D', 'F', 'H'], 'H' : ['G'] }
In [3]: graphe
Out[3]:
{'A': ['B', 'D', 'E'],
 'B': ['A', 'C', 'E'],
 'C': ['B', 'D'],
 'D': ['A', 'C', 'G'],
 'E': ['A', 'B', 'F'],
 'F': ['E', 'G'],
 'G': ['D', 'F', 'H'],
 'H': ['G']}
In [4]:  parcours_en_largeur(graphe,'A')
sommet a explorer: A
 Construction file voisins non visited: ['A', 'B']
 Construction file voisins non visited: ['A', 'B', 'D']
 Construction file voisins non visited: ['A', 'B', 'D', 'E']
Sommets visited: ['A']
sommet a explorer: B
 Construction file voisins non visited: ['B', 'D', 'E', 'C']
Sommets visited: ['A', 'B']
sommet a explorer: D
 Construction file voisins non visited: ['D', 'E', 'C', 'G']
Sommets visited: ['A', 'B', 'D']
sommet a explorer: E
 Construction file voisins non visited: ['E', 'C', 'G', 'F']
Sommets visited: ['A', 'B', 'D', 'E']
sommet a explorer: C
Sommets visited: ['A', 'B', 'D', 'E', 'C']
sommet a explorer: G
 Construction file voisins non visited: ['G', 'F', 'H']
Sommets visited: ['A', 'B', 'D', 'E', 'C', 'G']
sommet a explorer: F
Sommets visited: ['A', 'B', 'D', 'E', 'C', 'G', 'F']
sommet a explorer: H
Sommets visited: ['A', 'B', 'D', 'E', 'C', 'G', 'F', 'H']
Out[4]: ['A', 'B', 'D', 'E', 'C', 'G', 'F', 'H']
```

EXERCICE 02

```
In [5]: def parcours_en_profondeur(graphe,sommet):
   ...:     pile = [sommet]
   ...:     sommets_visites = []
   ...:     while len(pile) != 0 :
   ...:         S = pile.pop(); print("sommet a explorer:",S)
   ...:         sommets_visites.append(S)
   ...:         print("sommets visited:",sommets_visites)
   ...:         for voisin in graphe[S]:
   ...:             if not(voisin in sommets_visites) and not(voisin in pile):
   ...:                 pile.append(voisin)
   ...:                 print("construction de la pile des non visited :",pile)
   ...:     return sommets_visites
   ...:


In [6]: parcours_en_profondeur(graphe,'A')
sommet a explorer: A
sommets visited: ['A']
construction de la pile des non visited : ['B']
construction de la pile des non visited : ['B', 'D']
construction de la pile des non visited : ['B', 'D', 'E']
sommet a explorer: E
sommets visited: ['A', 'E']
construction de la pile des non visited : ['B', 'D', 'F']
sommet a explorer: F
sommets visited: ['A', 'E', 'F']
construction de la pile des non visited : ['B', 'D', 'G']
sommet a explorer: G
sommets visited: ['A', 'E', 'F', 'G']
construction de la pile des non visited : ['B', 'D', 'H']
sommet a explorer: H
sommets visited: ['A', 'E', 'F', 'G', 'H']
sommet a explorer: D
sommets visited: ['A', 'E', 'F', 'G', 'H', 'D']
construction de la pile des non visited : ['B', 'C']
sommet a explorer: C
sommets visited: ['A', 'E', 'F', 'G', 'H', 'D', 'C']
sommet a explorer: B
sommets visited: ['A', 'E', 'F', 'G', 'H', 'D', 'C', 'B']

Out[6]: ['A', 'E', 'F', 'G', 'H', 'D', 'C', 'B']
```

EXERCICE 03

Parmi les assertions suivantes, lesquelles sont vraies ?

A. On peut utiliser une file pour programmer avec Python un parcours en largeur ?

C'est vrai, on utilise une file. **Et ainsi l'assertion A) est True**.

B. Dans un parcours en profondeur, on commence par visiter tous les sommets adjacents au sommet de départ ?

On commence par visiter le sommet de départ puis un de ses voisins etc. pas tous les voisins directement. **Et ainsi l'assertion B) est False**.

C. On peut détecter la présence d'un cycle dans un graphe non orienté avec un parcours en largeur ?

L'idée est d'utiliser un tableau pour mémoriser le parent de chaque sommet (De quel sommet nous avons découvert chaque sommet) En découvrant les sommets, on vérifie si l'on retourne au sommet déjà visité et que ce sommet n'est pas le parent du sommet courant, si c'est le cas, alors il existe un cycle. Par exemple, le graphe qui a servi d'exemple dans les exercives 01 et 02 a des cycles.
Et l'exercice 04 est justement le développement de cette question. **Ainsi l'assertion C) est True**.

D. On ne peut pas détecter la connexité d'un graphe non orienté avec un parcours en profondeur ?

Quel que soit le type de parcours, on passe par tous les sommets appartenant à la classe de connexité du sommet de départ. **Et ainsi l'assertion D) est False**.

E. La complexité d'un parcours en profondeur dans un graphe d'ordre n est linéaire en n ?

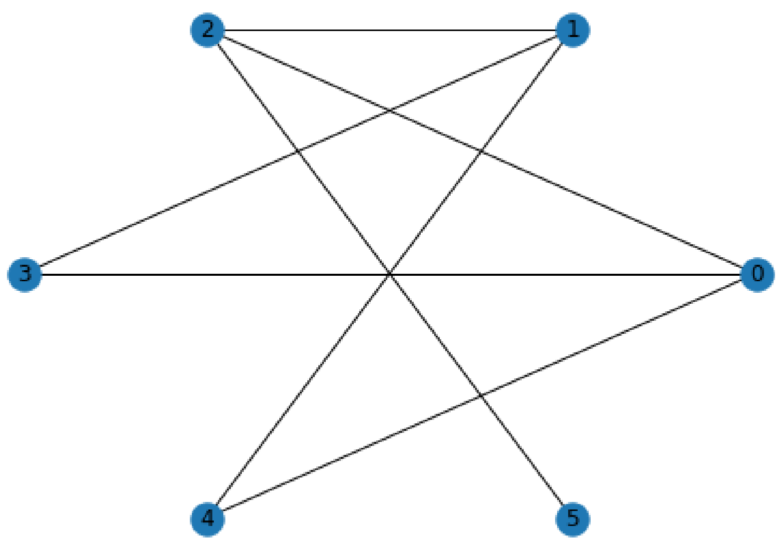
Dans un parcours en profondeur, on part d'un sommet, on passe à un de ses voisins puis ¨ a un voisin de ce voisin et ainsi de suite. S'il n'y a pas de voisin, on revient au sommet précédent et on passe à un autre de ses voisins. Au final on fait les n voisins. La complexité est bien d'ordre n.
Et ainsi l'assertion E) est True.

EXERCICE 04

1.

```
In [7]: import networkx as nx
In [8]: import matplotlib.pyplot as plt
In [9]: import numpy as np
In [10]: M1 = np.array([[0,0,1,1,1,0],[0,0,1,1,1,0],[1,1,0,0,0,1],
[1,1,0,0,0,0 ],[1,1,0,0,0,0],[0,0,1,0,0,0]])
In [11]: graphe1=nx.Graph(M1)
In [12]: position1= nx.circular_layout(graphe1)
In [13]: nx.draw(graphe1, with_labels=True, pos=position1)
```

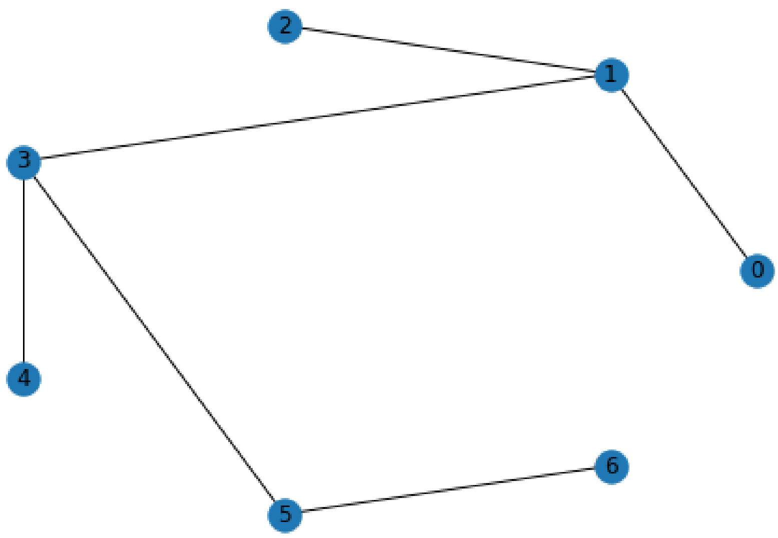
On obtient le graphe suivant :



2. On tape :

```
In [7]: import networkx as nx
In [8]: import matplotlib.pyplot as plt
In [9]: import numpy as np
In [25]: M2=np.array
    ([[0,1,0,0,0,0,0],[1,0,1,1,0,0,0],[0,1,0,0,0,0,0],
[0,1,0,0,1,1,0],[0,0,0,1,0,0,0],[0,0,0,1,0,0,1],[0,0,0,0,0,1,0]])
In [26]: M2
Out[26]:
array([[0, 1, 0, 0, 0, 0, 0],
       [1, 0, 1, 1, 0, 0, 0],
       [0, 1, 0, 0, 0, 0, 0],
       [0, 1, 0, 0, 1, 1, 0],
       [0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 1],
       [0, 0, 0, 0, 0, 1, 0]])
In [30]:  graphe2=nx.Graph(M2)
In [31]:  position2= nx.circular_layout(graphe2)
In [32]:  nx.draw(graphe2, with_labels=True, pos=position2)
```

On obtient le graphe suivant :



3. On remarque que l'on peut construire des cycles avec des sommets de `graphe1` du type $(s_0s_1...s_ps_0)$ avec les sommets intérieurs distincts et s_0 est un des sommets 0 à 4, il n'y a pas de cycle partant et revenant au sommet 5. On peut même dire qu'il n'y a pas de cycle contenant le sommet 5. Voici quelques cycles de `graphe1` :

$$(02130),\ (14021),\ (21402),\ (31403),\ (40314)$$

Par contre le graphe `graphe2` n'a aucun cycle, il s'appelle un **arbre**

Définition : Un arbre est un graphe non orienté G qui vérifie une des conditions équivalentes suivantes :
1) G est connexe et acyclique (ie sans cycle)
2) G est sans cycle et possède n -1 arêtes
3) G est connexe et admet n-1 arêtes
4) G est sans cycle, et en ajoutant une arête, on crée un et un seul cycle élémentaire (ie un cycle dont tous les sommets intérieurs sont distincts)
5) G est connexe, et en supprimant une arête quelconque, il n'est plus connexe.
6) Il existe une chaîne et une seule entre 2 sommets quelconques de G.

4-a On tape la procédure. J'ai rajouté quelques `print` et des explications pour mieux comprendre la bestiole.

```
In [28]: def cycle_som(M,debut):
    ...:     from collections import deque
    ...:     # nombre de sommets du graphe = n
    ...:     n=len(M)
    ...:     # dans FATHER on initialise avec des -1
    ...:     FATHER=[-1 for i in range(n)]
    ...:     # les sommets no traited sont blancs
    ...:     couleur = ["blanc" for i in range(n)]
    ...:     # creation deque vide
    ...:     D = deque()
    ...:     # on ajoute debut a extremite droite de D
    ...:     D.append(debut)
    ...:     # sommet debut en cours de traitement
    ...:     couleur[debut] = "gris"

    ...:     while len(D) != 0 :
    ...:         print("le deque D est:", D)
    ...:         print("PERE est :", FATHER)
    ...:         # on supprime le sommet x a extremite gauche de D
    ...:         x = D.popleft() ;print("le deque D vaut:",D)
    ...:         # le sommet x is treated
    ...:         couleur[x] = "noir"; print("couleur:",couleur)

    ...:         for i in range(n):
    ...:             if M[x][i]> 0 and couleur[i] != "blanc" and FATHER[x] != i:
    ...:                 # on a fund un chemin de x vers i
    ...:                 # i a ete visited
    ...:                 # on teste que le pere de x n'est pas i
    ...:                 return True
    ...:             elif M[x][i]>0 and couleur[i] == "blanc":
    ...:                 # on a fund un chemin de x vers i
    ...:                 # i n'est pas encore visited
    ...:                 D.append(i) ;  print("le deque D est:", D)
    ...:                 # on a added i a extremite droite de D
    ...:                 FATHER[i] = x
    ...:                 # le pere de i est x
    ...:                 couleur[i]="gris"
    ...:                 # le sommet i doit etre treated

    ...:     return False

    ...:     # pas de cycle en partant du sommet debut
    ...:
```

Tapons à la main `cycle_som(M1,0)`

Au départ, `debut=0` puis `n=6` et `FATHER=[-1,-1,-1,-1,-1,-1]`
Puis `couleur=[blanc,blanc,blanc, blanc,blanc,blanc]` et `D=()` puis `D=([0])`
Alors `couleur=[gris,blanc,blanc, blanc,blanc,blanc]` et `len(D)=1` On est dans le `while`
Ensuite `x=0` et `D=()` et `couleur=[noir,blanc,blanc, blanc,blanc,blanc]`

- Première boucle `for` avec `i in range(6)`

L'assertion `if M[x][i]> 0 and couleur[i] != "blanc"and FATHER[x] != i:` est False pour tous les `i` et `elif` est vraie :
pour `i=2` et alors `D=([2])` puis `FATHER=[-1,-1,0,-1,-1,-1]` et `couleur=[noir,blanc,gris, blanc,blanc,blanc]`
pour `i=3` et alors `D=([2,3])` puis `FATHER=[-1,-1,0,0,-1,-1]` et `couleur=[noir,blanc,gris, gris ,blanc,blanc]`
pour `i=4` et alors `D=([2,3,4])` puis `FATHER=[-1,-1,0,0,0,-1]` et `couleur=[noir,blanc,gris, gris, gris,blanc]`
On sort du premier `for`
Qu'avons-nous fait à ce stade ? On est parti du sommet `0` et on sait que les sommets `2,3,4` sont adjacents à `0`, on a mis `0` en noir, `2,3,4` en gris et les deux sommets `1,5` restent blancs et `0` est le père des trois sommets `2,3,4`. On continue.
On a alors `len(D)=3` $\neq 0$ et `PERE=[-1,-1,0,0,0,-1]` puis `x=2` et `D=([3,4])` et enfin `couleur=[noir,blanc,noir, gris, gris,blanc]`

- On se lance dans la seconde boucle `for` avec `i in range(6)`

L'assertion `if M[x][i]> 0 and couleur[i] != "blanc"and FATHER[x] != i:` est False pour tous les `i` et `elif` est vraie :
pour `i=1` et alors `D=([3,4,1])` puis `FATHER=[-1,2,0,0,0,-1]` et `couleur=[noir,gris,noir, gris,gris,blanc]`
pour `i=5` et alors `D=([3,4,1,5])` puis `FATHER=[-1,2,0,0,0,2]` et `couleur=[noir,gris,noir, gris,gris,gris]`
On sort du second `for`
Qu'avons-nous fait à ce stade ? `2` qui est à gauche dans `D` est sorti de `D` et est devenu noir et tous les sommets adjacents (non visités) de `2` soit `1,5` sont ajoutés à `D`. Et ces sommets `1,5` sont devenus gris. Et `2` est le père de `1` et `5`
Puis `len(D)=4` $\neq 0$ et `x=3` et `couleur=[noir,gris,noir, noir,gris,gris]`

- On attaque la troisième boucle `for` avec `i in range(6)`

L'assertion `if M[x][i]> 0 and couleur[i] != "blanc"and FATHER[x] != i:` est `True` pour `i=1` et donc on retourne `True` et c'est the end !

```
In [12]: cycle_som(M1,0)
le deque D est: deque([0])
PERE est : [-1, -1, -1, -1, -1, -1]
x vaut : 0
le deque D vaut: deque([])
couleur: ['noir', 'blanc', 'blanc', 'blanc', 'blanc', 'blanc']
le deque D est: deque([2])
le deque D est: deque([2, 3])
le deque D est: deque([2, 3, 4])
le deque D est: deque([2, 3, 4])
PERE est : [-1, -1, 0, 0, 0, -1]
x vaut : 2
le deque D vaut: deque([3, 4])
couleur: ['noir', 'blanc', 'noir', 'gris', 'gris', 'blanc']
le deque D est: deque([3, 4, 1])
le deque D est: deque([3, 4, 1, 5])
le deque D est: deque([3, 4, 1, 5])
PERE est : [-1, 2, 0, 0, 0, 2]
x vaut : 3
le deque D vaut: deque([4, 1, 5])
couleur: ['noir', 'gris', 'noir', 'noir', 'gris', 'gris']
Out[12]: True
```

4-b Le programme applique en fait l'algorithme du parcours en largeur. On utilise la liste `couleur` pour mémoriser la couleur des sommets. Un sommet est `"blanc"` lorsqu'il n'est pas traité. Lorsqu'on commence à traiter un sommet (quand on est dans `elif`), il est `"gris"`. Après avoir traité en largeur tous les sommets adjacents au sommet `i`, le sommet `i` est `"noir"` Ainsi, au départ, tous les sommets sont `"blanc"` sauf le sommet `debut` qui est `"gris"` car on commence par lui, évidemment.

On utilise aussi la liste `FATHER` dans la procèdure, ainsi `FATHER[i]` désigne le père du sommet `i` en traitement lors de la boucle intérieure. Au départ, `FATHER` est une liste de `-1` par convention et à chaque fois que l'on parcourt `elif` la valeur `FATHER[i]` est changée.

On utilise aussi une `deque` nommée `D` pour gérer la file d'attente FIFO. Dans la procèdure, en sortie de chaque boucle `for`, on supprime le sommet grisé à l'extremité gauche de `D` qui devient `"noir"`. Tous les sommets adjacents à ce sommet sont ajoutés à l'extremité droite de `D` et deviennent grisés.

Process de l'algorithme

Initialisation : On part de `debut` qui est le premier élément de `D` et `couleur` est une liste de `"blanc"` sauf à l'indice `debut` où c'est `"gris"` et enfin comme prévu `FATHER` est une liste de longueur `len(M)` avec que des `-1`

On rentre dans la boucle while qui fonctionne tant que `len(D) > 0` et dans la pratique, la condition `len(D)==0` n'arrivera que si l'algorithme ne trouve pas de cycle et il renverra donc `False`. Si par contre le programme trouve un cycle, ce sera dans le `while` et un `return True` sera actionné.
Plus concrétement chaque fois que le programme débute la boucle `while`, il enlève de `D` le sommet à son extremité gauche (qui s'appele `x`), le premier sommet qui partira est `debut` et sa couleur devient `"noir"`

Puis on rentre dans une boucle `for i in range(n):`
On teste l'assertion `if M[x][i]> 0 and couleur[i] != "blanc"and FATHER[x] != i:` Cette assertion est vraie s'il existe une arête de `x` à `i`, que le sommet `i` doit avoir été déjà visité et le père de `x` ne doit pas être `i` (ce qui permet d'éviter de considèrer $(i \to x \to i)$ comme cycle). Dans ce cas, on retourne `True` et le graphe possède au moins un cycle.

Si l'assertion `if M[x][i]> 0 and couleur[i] != "blanc"and FATHER[x] != i:` est `False`, on passe à `elif M[x][i]>0 and couleur[i] == "blanc":` qui signifie qu'il existe une arête de `x` à `i` mais que le sommet `i` n'est pas encore traité. On ajoute `i` en fin de `D` et on rentre `x` dans `FATHER[i]` car le père de `i` est `x` puis on met le sommet `i` en `"gris"`.
Enfin `else` qui ne contient que `L[x][i] ==0` laisse toutes les listes identiques.

```
In [13]: cycle_som(M1,1)
le deque D est: deque([1])
PERE est : [-1, -1, -1, -1, -1, -1]
x vaut : 1
le deque D vaut: deque([])
couleur: ['blanc', 'noir', 'blanc', 'blanc', 'blanc', 'blanc']
le deque D est: deque([2])
le deque D est: deque([2, 3])
le deque D est: deque([2, 3, 4])
le deque D est: deque([2, 3, 4])
PERE est : [-1, -1, 1, 1, 1, -1]
x vaut : 2
le deque D vaut: deque([3, 4])
couleur: ['blanc', 'noir', 'noir', 'gris', 'gris', 'blanc']
le deque D est: deque([3, 4, 0])
le deque D est: deque([3, 4, 0, 5])
le deque D est: deque([3, 4, 0, 5])
PERE est : [2, -1, 1, 1, 1, 2]
x vaut : 3
le deque D vaut: deque([4, 0, 5])
couleur: ['gris', 'noir', 'noir', 'noir', 'gris', 'gris']
Out[13]: True
```

```
In [14]: cycle_som(M1,2)
le deque D est: deque([2])
PERE est : [-1, -1, -1, -1, -1, -1]
x vaut : 2
le deque D vaut: deque([])
couleur: ['blanc', 'blanc', 'noir', 'blanc', 'blanc', 'blanc']
le deque D est: deque([0])
le deque D est: deque([0, 1])
le deque D est: deque([0, 1, 5])
le deque D est: deque([0, 1, 5])
PERE est : [2, 2, -1, -1, -1, 2]
x vaut : 0
le deque D vaut: deque([1, 5])
couleur: ['noir', 'gris', 'noir', 'blanc', 'blanc', 'gris']
le deque D est: deque([1, 5, 3])
le deque D est: deque([1, 5, 3, 4])
le deque D est: deque([1, 5, 3, 4])
PERE est : [2, 2, -1, 0, 0, 2]
x vaut : 1
le deque D vaut: deque([5, 3, 4])
couleur: ['noir', 'noir', 'noir', 'gris', 'gris', 'gris']
Out[14]: True
```

```
In [17]: cycle_som(M1,5)
le deque D est: deque([5])
PERE est : [-1, -1, -1, -1, -1, -1]
x vaut : 5
le deque D vaut: deque([])
couleur: ['blanc', 'blanc', 'blanc', 'blanc', 'blanc', 'noir']
le deque D est: deque([2])
le deque D est: deque([2])
PERE est : [-1, -1, 5, -1, -1, -1]
x vaut : 2
le deque D vaut: deque([])
couleur: ['blanc', 'blanc', 'noir', 'blanc', 'blanc', 'noir']
le deque D est: deque([0])
le deque D est: deque([0, 1])
le deque D est: deque([0, 1])
PERE est : [2, 2, 5, -1, -1, -1]
x vaut : 0
le deque D vaut: deque([1])
couleur: ['noir', 'gris', 'noir', 'blanc', 'blanc', 'noir']
le deque D est: deque([1, 3])
le deque D est: deque([1, 3, 4])
le deque D est: deque([1, 3, 4])
PERE est : [2, 2, 5, 0, 0, -1]
x vaut : 1
le deque D vaut: deque([3, 4])
couleur: ['noir', 'noir', 'noir', 'gris', 'gris', 'noir']
Out[17]: True
```

4-c On suppose que la ligne `M[0:]` de la matrice d'adjacence ne possède que des `0`.
Alors `D=([0])` et comme `len(D) != 0`, on commence. On enlève `x=0` de `D` qui devient `D=()` puis on rentre dans la boucle `for` mais `M[0][i] > 0` est toujours `False` et donc on sort de `for` de suite et alors comme `len(D)=0`, on sort aussi de `while`. Et on retourne `False`. Pourtant le graphe peut avoir des cycles.

En conclusion, si le programme `cycle_som(M,debut)` retourne `False`, cela ne signifie pas qu'il n'y a pas de cycle, cela signifie que l'algorithme n'en trouve pas en partant du sommet `debut`. Par contre, si l'algorithme retourne `False` en partant de tous les sommets, il n'y a pas de cycle.

4-d Tapons à la main `cycle_som(M2,0)`

Je l'ai fait avec Python, cela suffira.

```
In [20]: cycle_som(M2,0)
le deque D est: deque([0])
PERE est : [-1, -1, -1, -1, -1, -1, -1]
x vaut : 0
le deque D vaut: deque([])
couleur: ['noir', 'blanc', 'blanc', 'blanc', 'blanc', 'blanc','blanc']
le deque D est: deque([1])
le deque D est: deque([1])
PERE est : [-1, 0, -1, -1, -1, -1, -1]
x vaut : 1
le deque D vaut: deque([])
couleur: ['noir', 'noir', 'blanc', 'blanc', 'blanc', 'blanc','blanc']
le deque D est: deque([2])
le deque D est: deque([2, 3])
le deque D est: deque([2, 3])
PERE est : [-1, 0, 1, 1, -1, -1, -1]
x vaut : 2
le deque D vaut: deque([3])
couleur: ['noir', 'noir', 'noir', 'gris', 'blanc', 'blanc','blanc']
le deque D est: deque([3])
PERE est : [-1, 0, 1, 1, -1, -1, -1]
x vaut : 3
le deque D vaut: deque([])
couleur: ['noir', 'noir', 'noir', 'noir', 'blanc', 'blanc','blanc']
le deque D est: deque([4])
le deque D est: deque([4, 5])
le deque D est: deque([4, 5])
PERE est : [-1, 0, 1, 1, 3, 3, -1]
x vaut : 4
le deque D vaut: deque([5])
couleur: ['noir', 'noir', 'noir', 'noir', 'noir', 'gris','blanc']
le deque D est: deque([5])
PERE est : [-1, 0, 1, 1, 3, 3, -1]
x vaut : 5
le deque D vaut: deque([])
couleur: ['noir', 'noir', 'noir', 'noir', 'noir', 'noir','blanc']
le deque D est: deque([6])
le deque D est: deque([6])
PERE est : [-1, 0, 1, 1, 3, 3, 5]
x vaut : 6
le deque D vaut: deque([])
couleur: ['noir', 'noir', 'noir', 'noir', 'noir', 'noir','noir']
Out[20]: False
```

On peut faire les autres, on retourne `False` et voici le dernier :

```
In [26]:  cycle_som(M2,6)
le deque D est: deque([6])
PERE est : [-1, -1, -1, -1, -1, -1, -1]
x vaut : 6
le deque D vaut: deque([])
couleur: ['blanc', 'blanc', 'blanc', 'blanc','blanc','blanc','noir']
le deque D est: deque([5])
le deque D est: deque([5])
PERE est : [-1, -1, -1, -1, -1, 6, -1]
x vaut : 5
le deque D vaut: deque([])
couleur: ['blanc', 'blanc', 'blanc', 'blanc', 'blanc', 'noir','noir']
le deque D est: deque([3])
le deque D est: deque([3])
PERE est : [-1, -1, -1, 5, -1, 6, -1]
x vaut : 3
le deque D vaut: deque([])
couleur: ['blanc', 'blanc', 'blanc', 'noir', 'blanc', 'noir','noir']
le deque D est: deque([1])
le deque D est: deque([1, 4])
le deque D est: deque([1, 4])
PERE est : [-1, 3, -1, 5, 3, 6, -1]
x vaut : 1
le deque D vaut: deque([4])
couleur: ['blanc', 'noir', 'blanc', 'noir', 'gris', 'noir','noir']
le deque D est: deque([4, 0])
le deque D est: deque([4, 0, 2])
le deque D est: deque([4, 0, 2])
PERE est : [1, 3, 1, 5, 3, 6, -1]
x vaut : 4
le deque D vaut: deque([0, 2])
couleur: ['gris', 'noir', 'gris', 'noir', 'noir', 'noir','noir']
le deque D est: deque([0, 2])
PERE est : [1, 3, 1, 5, 3, 6, -1]
x vaut : 0
le deque D vaut: deque([2])
couleur: ['noir', 'noir', 'gris', 'noir', 'noir','noir','noir']
le deque D est: deque([2])
PERE est : [1, 3, 1, 5, 3, 6, -1]
x vaut : 2
le deque D vaut: deque([])
couleur: ['noir', 'noir', 'noir', 'noir', 'noir','noir','noir']
Out[26]: False
```

On peut donc voir qu'en partant de tous les sommets, on retourne toujours `False` et donc le graphe associé à `M2` n'a pas de cycle.

4-e Écrivons un programme `research_cycle(M)` qui affiche `True` si `M` possède au moins un cycle et `False` sinon. Par exemple, on tape :

```
In [3]: In [1]: def research_cycle(M):
   ...:     # fonction retourne False if no cycle
   ...:     n = len(M)
   ...:     for k in range(n):
   ...:         if cycle_som(M,k) == True :
   ...:             print("ce graphe a au moins un cycle")
   ...:             return True
   ...:             # permet de sortir if cycle found
   ...:     print("Ce graphe est sans cycle")
   ...:     return False
   ...:     # if cycle not found, on retourne False
   ...:
```