TP matrices d'adjacences

Dans ce TP, les graphes sont représentés par *matrices d'adjacences*. Par exemple :

[1]:
```
m = [[0,1,0,0,0],\
     [1,0,1,0,0],\
     [0,0,0,1,0],\
     [0,0,1,0,0],\
     [1,0,0,0,0]]
```

- Q0. Représenter le graphe dont la matrice d'adjacence est donnée ci-dessus.

1 Divers

- Q1. Ecrire la fonction `degplus(g,i)` qui donne le degré sortant du sommet i du graphe g.
- Q2. Ecrire la fonction `degmoins(g,i)` qui donne le degré entrant du sommet i du graphe g.
- Q3. Un *puits* est un sommet vers lequel tous les autres pointent et qui ne pointe sur personne ou alors juste sur lui-même. Donner la fonction `puits(g)` qui indique si graphe g possède un puits.
- Q4.Donner la fonction `transpose(g)` qui retourne le graphe transposé du graphe g (toutes les flèches sont inversées).
- Q5. Le graphe g est dans cette question donné sous forme de listes d'arcs. Ecrire les fonctions `entrant(g,i)` et `sortant(g,i)` qui renvoient respectivement les degrés entrant et sortant d'un sommet i.
- Q6. Ecrire la fonction `voisins(g,i)` qui renvoie la liste des voisins de i dans le graphe g.

[3]:
```
m = [[0,1,0,0,0],\
     [1,0,1,0,0],\
     [0,0,0,1,0],\
     [0,0,1,0,0],\
     [1,0,0,0,0]]
degplus(m,1)
```

[3]:
```
2
```

[4]:
```
degmoins(m,0)
```

[4]:
```
2
```

[6]:
```
m = [[0,1,0,0,0],\
     [1,0,1,0,0],\
     [0,0,0,1,0],\
     [0,0,1,0,0],\
     [1,0,0,0,0]]
puits(m)
```

[6]:
```
False
```

[7]:
```
m1 = [[0,1,1,0,0],\
     [1,0,1,0,0],\
     [0,0,1,0,0],\
     [0,0,1,0,0],\
     [1,0,1,0,0]]
puits(m1)
```

[7]:
```
True
```

[9]:
```
m
```

[9]:
```
[[0, 1, 0, 0, 0],
 [1, 0, 1, 0, 0],
 [0, 0, 0, 1, 0],
 [0, 0, 1, 0, 0],
 [1, 0, 0, 0, 0]]
```

[10]:
```
tm = transpose(m)
tm
```

[10]:
```
[[0, 1, 0, 0, 1],
 [1, 0, 0, 0, 0],
 [0, 1, 0, 1, 0],
 [0, 0, 1, 0, 0],
 [0, 0, 0, 0, 0]]
```

[11]:
```
transpose(tm)
```

[11]:
```
[[0, 1, 0, 0, 0],
 [1, 0, 1, 0, 0],
 [0, 0, 0, 1, 0],
 [0, 0, 1, 0, 0],
 [1, 0, 0, 0, 0]]
```

[12]:
```
m
```

[12]:
```
[[0, 1, 0, 0, 0],
 [1, 0, 1, 0, 0],
 [0, 0, 0, 1, 0],
 [0, 0, 1, 0, 0],
 [1, 0, 0, 0, 0]]
```

[14]:
```
voisins(m,1)
```

[14]:
```
[0, 2]
```

[15]:
```
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
```

2 Conversion

- Q1. Etant donné un graphe donné par matrice d'adjacence m, écrire la fonction `convertl(m)` qui renvoie une représentation de g comme liste d'adjacence.
- Q2. Ecrire une fonction `convertm(g)` qui renvoie, sous forme de matrice d'adjacence, un graphe donné comme liste de listes de voisins.
- Q3. Ecrire la fonction `arcs(g)` qui renvoie la liste des arêtes du graphe g (donc une liste de tuples de la forme (i, j) où i, j sont deux sommets).
- Q4. Ecrire la fonction `arcs2mat(l)` qui prend en paramètres une liste d'arcs l et renvoie le graphe g donné sous forme de matrice tel que :
  - Les sommets de g sont des entiers; le premier sommet vaut 0.
  - le dernier sommet du graphe g est le sommet de plus grand numéro dans la liste des arcs.
  - Les arcs de g sont ceux de la liste d'arcs l.

[17]:
```
m = [[0,1,0,0,0],\
     [1,0,1,0,0],\
     [0,0,0,1,0],\
     [0,0,1,0,0],\
     [1,0,0,0,0]]

g=convertl(m)
g
```

[17]:
```
[[1], [0, 2], [3], [2], [0]]
```

[19]:
```
convertm(convertl(m)) == m
```

[19]:
```
True
```

[20]:
```
convertm(g)
```

[20]:
```
[[0, 1, 0, 0, 0],
 [1, 0, 1, 0, 0],
 [0, 0, 0, 1, 0],
 [0, 0, 1, 0, 0],
 [1, 0, 0, 0, 0]]
```

[22]:
```
m
```

[22]:
```
[[0, 1, 0, 0, 0],
 [1, 0, 1, 0, 0],
 [0, 0, 0, 1, 0],
 [0, 0, 1, 0, 0],
 [1, 0, 0, 0, 0]]
```

[23]:
```
am = arcs(m)
am
```

[23]:
```
[(0, 1), (1, 0), (1, 2), (2, 3), (3, 2), (4, 0)]
```

[25]:
```
arcs2mat(am)
```

[25]:
```
[[0, 1, 0, 0, 0],
 [1, 0, 1, 0, 0],
 [0, 0, 0, 1, 0],
 [0, 0, 1, 0, 0],
 [1, 0, 0, 0, 0]]
```

3 Chemins

- Q1. Ecrire la fonction `is_path(g,c)` qui indique si la liste de sommets c du graphe g est un *chemin* : pour tout i convenable, le sommet $i+1$ du chemin c doit être un voisin du sommet i.
- Q2. Ecrire la fonction `is_cycle(g,c)` qui indique si la liste de sommets c du graphe g est un *cycle* : c'est à dire un chemin dont les deux extrémités sont confondues.

[27]:
```
m3 = [[0, 1, 0, 0, 0],\
      [1, 0, 1, 0, 0],\
      [0, 0, 0, 1, 0],\
      [0, 0, 1, 0, 1],\
      [1, 0, 0, 0, 1]]
```

[28]:
```
l1 = [0,1,2,3,4,0]
is_path(m3,l1)
```

[28]:
```
True
```

[29]:
```
l2 = [0,1,2,3,1,0]
is_path(m3,l2)
```

[29]:
```
False
```

[31]:
```
is_cycle(m3,l1),is_cycle(m3,l1[:-1]), is_cycle(m3,l2)
```

[31]:
```
(True, False, False)
```

4 Parcours en profondeur

On implante le parcours en profondeur à trois couleurs vu en cours :

- Un sommet est bleu s'il n'a pas encore été visité;
- Un sommet est vert s'il a été abordé par l'exploration et si son traitement n'est pas terminé;
- Un sommet est rouge lorsqu'il a été abordé par l'exploration et si son traitement est terminé.

[32]:
```
R,V,B=["R","V","B"] # rouge, vert, bleu : les trois couleurs
```

- Q1. Ecrire la procédure `profondeur(m,s,col)` qui fait un parcours en profondeur à partir du sommet s de m (une matrice d'adjacence) et met à jour le tableau des couleurs `col` à cette occasion.
- Q2. Ecrire la fonction `accessible(m,s)` qui renvoie la liste des accessibles depuis s dans le graphe m.
- Q3. Ecrire la fonction `cycle(m)` qui renvoie un bouléen indiquant si le graphe orienté m possède un cycle. On rappelle qu'un graphe possède un cycle, si lors d'un parcours en profondeur, le sommet d ela pile pointe sur un sommet vert.

[34]:
```
g = [[1,3],[3],[4],[1],[1,3,4],[3,4,2],[0,1,2,4,5]]
m = convertm(g)
m
```

[34]:
```
[[0, 1, 0, 1, 0, 0, 0],
 [0, 0, 0, 1, 0, 0, 0],
 [0, 0, 0, 0, 1, 0, 0],
 [0, 1, 0, 0, 0, 0, 0],
 [0, 1, 0, 1, 1, 0, 0],
 [0, 0, 1, 1, 1, 0, 0],
 [1, 1, 1, 0, 1, 1, 0]]
```

[35]:
```
col =[B] * len(m)
profondeur(m,0,col)
col
```

[35]:
```
['R', 'R', 'B', 'R', 'B', 'B', 'B']
```

[37]:
```
accessibles(m,0), accessibles(m,5)
```

[37]:
```
([0, 1, 3], [1, 2, 3, 4, 5])
```

[39]:
```
g = [[1],[2],[3],[],[1,2]]
m = convertm(g)
cycle(m)
```

[39]:
```
False
```

[40]:
```
g = [[1],[2,4],[3],[],[1]]
m = convertm(g)
cycle(m)
```

[40]:
```
True
```

5 Coloration

Une *coloration* de graphe est une application c qui va des sommets du graphe vers un ensemble de valeurs (appelées *couleurs*) et vérifie la propriété suivante : *Deux sommets voisins n'ont pas la même couleur.*

Les colorations d'un graphe sont représentés par des listes de nombres ayant autant d'items que le graphe a de sommet.

- Q1. Ecrire la fonction `est_coloration(g,c)`

[42]:
```
g = [[1],[3],[0],[0,2],[1,2,3]]
m = convertm(g)
print(m)
c1 = [0,1,2,3,0]
est_coloration(g,c1)
```

```
[[0, 1, 0, 0, 0], [0, 0, 0, 1, 0], [1, 0, 0, 0, 0], [1, 0, 1, 0, 0], [0, 1, 1,
1, 0]]
```

[42]:
```
True
```

[43]:
```
c2 = [0,1,2,3,2]
est_coloration(g,c2)
```

[43]:
```
False
```

On se rend compte que tout graphe a une coloration triviale : il suffit d'affecter à chaque sommet son numéro : le sommet i porte la couleur i. Une telle coloration utilise n couleurs. Mais on peut faire mieux. Par exemple, la coloration `c1` ci-dessus n'utilise que 4 couleurs alors qu'il y a 5 sommets.

Pour tout graphe G de n sommets, il existe une coloration *optimale* au sens du nombre de couleurs utulisées. Le nombre de couleurs utilisées par une coloration optimale est appelé *nombre chromatique* de G. Un algormithme simple de recherche de coloration optimale de G consiste à essayer toutes les permutations de n éléments dans $\{0, 1, n-1\}$. Comme il y a $n!$ permutations, la complexité de cet algorithme est exponentielle (songer à la formule de Stirling).

Trouver un algorithme qui calcule une coloration optimale pour tout type de graphe en un temps polynômial est un sujet majeur de l'informatique actuelle (le problème de la coloration étant *NP-complet*). Pour le moment, on ne connaît pas d'algorithme qui fasse mieux qu'un temps exponentiel dans le pire cas. On se contente donc d'heuristiques comme l'algorithme *RLF* présenté plus bas.

5.1 Algorithme RLF

En 1979, Leighton propose l'algorithme Recursive Largest First (RLF) qui construit les classes de couleur l'une après l'autre. Toute *classe de couleur*, est un *stable*, c'est à dire un sous-ensemble de sommets dont aucun n'est voisin d'un autre.

Nous donnons l'algorithme pour les graphes non orientés. Il faut l'adapter à ce TP qui concerne les graphes orientés.

L'algorithme est principalement une double boucle imbriquée: au tour k de la boucle externe, on attribue la couleur k. On considère U_1^k l'ensemble des nœuds non colorés qui peuvent recevoir la couleur k et U_2^k l'ensemble des nœuds non colorés qui ne peuvent plus recevoir la couleur k (car ils sont voisins d'au moins un nœud coloré avec la couleur k). Au début de la création de la classe de couleur k, U_1 est égal à l'ensemble des sommets qui n'ont aucune des $k-1$ premières couleurs et U_2 est vide. La boucle interne, répète les opérations suivantes tant que $U_1^k \neq \emptyset$:

1. Assigner la couleur k au nœud v de U_1^k qui a le plus de voisins dans U_2^k. En cas d'égalité, choisir un sommet de U_1^k qui a le moins de voisins dans U_1^k. Ôter v de U_1.
2. Transférer tous les voisins de v présents dans U_1^k vers U_2^k

On sort de la boucle interne lorsque U_1^k est vide et on pose alors $U_1^{k+1} = U_2^k$. Si U_1^{k+1} est vide, l'algorithme termine sinon on recommence.

- Q1. Ecrire la fonction `est_voisin(g,u,v)` qui indique si l'un des sommets u ou v est voisin de l'autre. Il s'agit donc d'un voisinage au sens non orienté.
- Q2. Ecrire le fonction `nb_de_voisins_dans(g,U,v)` qui donne le nombre de sommets de U qui sont voisins (au sens non orienté) de v. U est une liste de sommets.
- Q3. Ecrire la fonction `choisir_sommet_a_colorier(g,U1,U2)` qui choisit le sommet de U_1 à colorier comme indiqué dans l'étape 1. de la boucle interne du RLF. La fonction supprime le sommet à colorier de la liste U_1.

[136]:
```
m = [[0,1,0,0,1],\
     [1,0,1,0,0],\
     [1,0,0,1,0],\
     [0,0,1,0,1],\
     [0,1,0,0,0]]

est_voisin(m,0,2), est_voisin(m,3,0)
```

[136]:
```
(True, False)
```

[137]:
```
nb_de_voisins_dans(m,[0,1],4),nb_de_voisins_dans(m,[0,1],3)
```

[137]:
```
(2, 0)
```

[144]:
```
m = [[0,0,0,0,1,0],\
     [0,0,0,1,0,1],\
     [0,0,0,1,0,0],\
     [0,1,1,0,0,0],\
     [1,0,0,0,0,1],\
     [0,1,0,0,0,0]]

U1, U2 = [0,1,2], [3,4,5]
print(choisir_sommet_a_colorier(m,U1,U2))
U1
```

```
1
```

[144]:
```
[0, 2]
```

[143]:
```
m = [[0,0,1,0,0,1,1],\
     [0,0,1,1,1,1,0],\
     [1,1,0,1,0,0,0],\
     [0,1,0,0,0,0,1],\
     [0,1,0,0,0,0,0],\
     [1,1,0,0,0,0,1],\
     [1,0,0,0,0,1,0]]

U1, U2 = [0,1,2,3], [4,5,6]
print(choisir_sommet_a_colorier(m,U1,U2))
U1
```

```
0
```

[143]:
```
[2, 3, 1]
```

- Q4. Ecrire le fonction `single(U)` qui renvoie une copie de U sans doublon de valeur. Etudier la complexité.

[58]:
```
l = [1,0,2,3,4,1,3,1]
single(l)
```

[58]:
```
[1, 0, 2, 3, 4]
```

- Q5. Ecrire la procédure `transfert(G,U1,U2,v)` qui prend en paramètres deux listes U_1 et U_2 et tansfert tous les voisins (au sens non orienté) de v qui sont dans U_1 vers U_2. La liste U_2

est modifiée mais pas U_1. La fonction renvoie une copie de U_1 sans les sommets transférés. On suppose que $v \notin U_1$.

[146]:
```
m = [[0,1,0,0,1],\
     [1,0,1,0,0],\
     [1,0,0,1,0],\
     [0,0,1,0,1],\
     [0,1,1,0,0]]
```

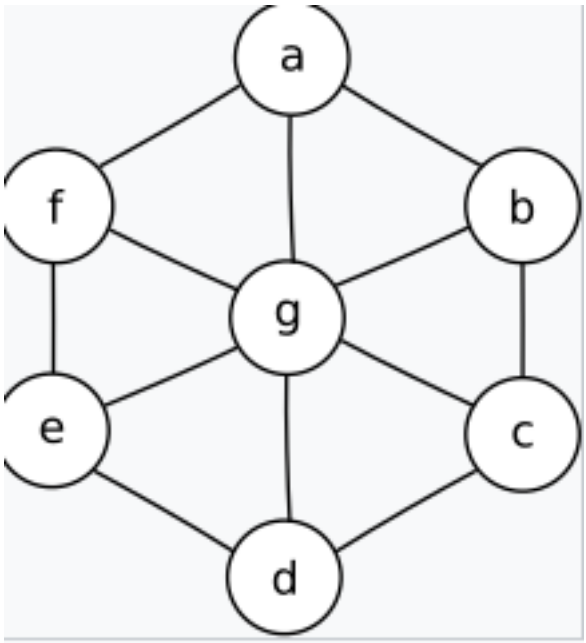
[147]:
```
U1,U2=[0,1,4],[]
print(transfert(m,U1,U2,4))
U2
```

```
[4]
```

[147]: `[0, 1]`

- Q6. Ecrire la fonction `rlf(g)` qui implémente l'algorithme RLF et renvoie une coloration de g sous forme de tableau de couleur.

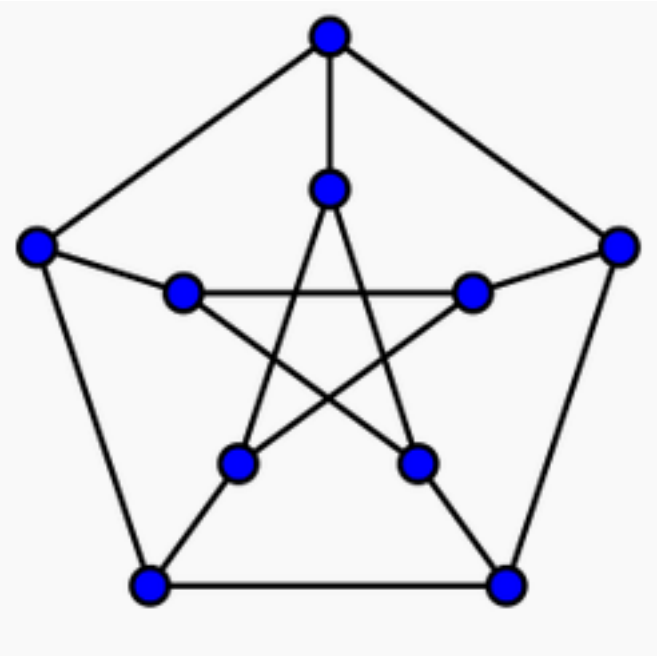
Un *wheel graph* à 7 sommets (d'après Wikipedia).



[157]:
```
m = matrice_nulle(7,7)

#juste pratique
def add_edges(m,u,v):
    m[u][v] = m[v][u] = 1

add_edges(m,0,1)
```

```
add_edges(m,0,5)

add_edges(m,2,1)
add_edges(m,2,3)

add_edges(m,4,3)
add_edges(m,4,5)

# wheel graph à 7 sommets
for i in range(len(m)):
    add_edges(m,i,6)

m
```

[157]:
```
[[0, 1, 0, 0, 0, 1, 1],
 [1, 0, 1, 0, 0, 0, 1],
 [0, 1, 0, 1, 0, 0, 1],
 [0, 0, 1, 0, 1, 0, 1],
 [0, 0, 0, 1, 0, 1, 1],
 [1, 0, 0, 0, 1, 0, 1],
 [1, 1, 1, 1, 1, 1, 1]]
```



Un graphe de Pertersen (d'après wikipedia)

[155]:
```
rlf(m)
```

[155]:
```
[1, 0, 1, 0, 1, 0, 2]
```

- Q7. Implanter un graphe de Petersen et le colorer avec RLF.

[162]:
```
rlf(m)
```

[162]:
```
[1, 0, 1, 0, 2, 0, 1, 2, 2, 0]
```