

TP : Graphes

Les graphes sont notés sous la forme $G = (V, E)$ où V désigne l'ensemble des sommets et E celui des arcs (cas orienté) ou des arêtes (cas non orienté). Un ensemble d'arcs est constitué de couples de $V \times V$, tandis qu'un ensemble d'arête est constitué de singletons (boucles) ou de paires (arêtes à deux sommets différents).

Un graphe est dit *simple* s'il ne possède pas de boucle ni de *multi-arête* (un arc/arête est parfaitement défini.e par la donnée de ses extrémités).

1 A propos des degrés

Question 1.

Montrer qu'un graphe simple non orienté a un nombre pair de sommets de degré impair.

Question 2. Nombre d'amis

Montrer que dans une assemblée de n personnes ($n > 2$), il y a toujours au moins deux personnes qui ont le même nombre d'amis présents (on considère que la relation d'amitié est symétrique mais pas réflexive).

Question 3. Réseau

Est-il possible de créer un réseau de 15 ordinateurs de sorte que chaque machine soit reliée avec exactement trois autres ?

Chemins

Un chemin est dit *élémentaire* si il ne comporte pas de doublon de sommets. Il est dit *simple* si il ne comporte pas de doublon d'arc/arête.

Question 4. Propriété de König

Montrer que dans un graphe $G = (V, E)$, s'il existe un chemin d'un sommet u vers v alors il existe un chemin élémentaire de u vers v .

2 Implémentation

Dans ce TP, les graphes sont représentés par matrices d'adjacence.
On donne :

Listing 1 – Matrice d'adjacence

```
1 m = [[0,1,0,0,0],\
2      [1,0,1,0,0],\
3      [0,0,0,1,0],\
4      [0,0,1,0,0],\
5      [1,0,0,0,0]]
```

Question 5.

Représenter le graphe 1.

2.1 Généralités

Question 6.

Écrire la fonction `degplus(g,i)` qui donne le degré sortant du sommet i du graphe g .

Question 7.

Écrire la fonction `degmoins(g,i)` qui donne le degré entrant du sommet i du graphe g .

Question 8.

Donner la fonction `transpose(g)` qui retourne le graphe transposé du graphe g (toutes les flèches sont inversées).

Question 9.

Écrire la fonction `voisins_de(g,i)` qui renvoie la liste des voisins de i dans le graphe g .

Question 10.

Écrire la fonction `est_voisins_de(g,i)` qui renvoie la liste des sommets dont i est voisin dans le graphe g .

Question 11.

Écrire la fonction `graphe_sans_arc(n)` qui renvoie la matrice d'adjacence d'un graphe à n sommet sans arc/arête.

Coloration

Une *coloration* d'un graphe est une application c qui associe à chaque sommet du graphe une valeur, appelée *couleur*, et qui vérifie la propriété suivante : deux sommets voisins n'ont pas la même couleur. Les couleurs sont ici représentées par des entiers positifs.

Les colorations d'un graphe sont représentées par des listes de nombres comportant autant d'éléments que le graphe possède de sommets. Par exemple, pour un graphe à cinq sommets, une coloration peut être représentée par

```
1 c = [0,1,0,2,1]
```

Question 12.

La liste ci-dessus ci-dessus est-elle une coloration du graphe 1 ?

Question 13.

Écrire la fonction `est_coloration(g,c)` qui indique par un booléen si c est une coloration du graphe g .

Listing 2 – checking de coloration

```
1 m = [[0, 1, 0, 0, 0],
2       [1, 0, 1, 0, 0],
3       [0, 0, 0, 1, 0],
4       [0, 0, 1, 0, 0],
5       [1, 0, 0, 0, 0]]
6 c1 = [0,1,2,3,0]
7 est_coloration(g,c1) # renvoie False
```

On remarque que tout graphe possède une coloration triviale : il suffit d'affecter à chaque sommet son numéro. Ainsi, le sommet i reçoit la couleur i . Pour un graphe à n sommets, cette coloration utilise donc n couleurs. Il est cependant souvent possible d'en utiliser beaucoup moins. Par exemple, la coloration `c1` ci-dessus n'utilise que quatre couleurs alors que le graphe possède cinq sommets.

Parmi toutes les colorations d'un graphe G , il en existe au moins une utilisant un nombre minimal de couleurs. Ce nombre minimal est appelé le *nombre chromatique* de G .

Un algorithme naïf permettant de déterminer une coloration optimale consiste à essayer toutes les affectations possibles des n sommets avec des couleurs prises dans

$$\{0, \dots, n-1\}.$$

Il y en a n^n . On teste alors lesquelles sont des colorations et on conserve celle qui utilise le moins de couleurs. La complexité de cette méthode est donc exponentielle.

La recherche d'une coloration optimale est un problème difficile. Plus précisément, pour tout entier fixé $k \geq 3$, décider si un graphe peut être colorié avec au plus k couleurs est un problème *NP-complet*. On ne connaît actuellement aucun algorithme polynomial permettant de résoudre ce problème dans le cas général.

En pratique, pour les grands graphes, on utilise donc souvent des heuristiques qui produisent rapidement de bonnes colorations, sans garantir qu'elles soient optimales. C'est le cas de l'algorithme *RLF* présenté plus bas.

2.2 Algorithme RLF

En 1979, Leighton propose l'algorithme *Recursive Largest First* (RLF), qui construit les classes de couleur les unes après les autres. Une *classe de couleur* est un *stable*, c'est-à-dire un ensemble de sommets deux à deux non voisins.

Nous présentons ici l'algorithme dans le cas des graphes non orientés. Dans un tel graphe, si x est voisin de y , alors y est également voisin de x . Il faudra donc adapter la notion de voisinage au cadre de ce TP, qui porte sur les graphes orientés.

L'algorithme repose sur deux boucles imbriquées. À l'étape k de la boucle externe, on construit la classe des sommets auxquels on attribue la couleur k .

On note :

- U_1^k l'ensemble des sommets non encore colorés qui peuvent encore recevoir la couleur k ;
- U_2^k l'ensemble des sommets non encore colorés qui ne peuvent plus recevoir la couleur k , car ils sont voisins d'au moins un sommet auquel la couleur k a déjà été attribuée.

Au début de la construction de la classe de couleur k , l'ensemble U_1^k contient tous les sommets qui n'ont encore reçu aucune couleur, tandis que U_2^k est vide. En particulier, au début de l'algorithme,

$$U_1^0 = V \quad \text{et} \quad U_2^0 = \emptyset,$$

où V désigne l'ensemble des sommets du graphe.

La boucle interne répète alors les opérations suivantes tant que $U_1^k \neq \emptyset$:

1. Choisir dans U_1^k un sommet v ayant le plus grand nombre de voisins dans U_2^k . En cas d'égalité, choisir parmi ces sommets celui qui possède le moins de voisins dans U_1^k . Attribuer à v la couleur k , puis retirer v de U_1^k .
2. Transférer de U_1^k vers U_2^k tous les voisins de v encore présents dans U_1^k .

Lorsque U_1^k devient vide, la classe de couleur k est entièrement construite. Les sommets qui restent non colorés sont précisément ceux de U_2^k . On pose donc

$$U_1^{k+1} = U_2^k.$$

Si U_1^{k+1} est vide, tous les sommets ont été colorés et l'algorithme termine. Sinon, on commence la construction de la classe de couleur suivante.

Question 14.

Écrire la fonction `est_voisin_NO(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.

Listing 3 – Voisins au sens non orienté (NO)

```

1 m = [[0,1,0,0,1],\
2      [1,0,1,0,0],\
3      [1,0,0,1,0],\
4      [0,0,1,0,1],\
5      [0,1,0,0,0]]
6
7 est_voisin_NO(m,0,2), est_voisin_NO(m,3,0) #True, False
```

Question 15.

Écrire la fonction `nb_de_voisins_dans(g,U,v)` qui renvoie le nombre de sommets de U qui sont voisins de v au sens non orienté. La variable U est une liste de sommets.

Listing 4 – Nombres de voisins dans un ensemble

```

1 m = [[0,1,0,0,1],\
2      [1,0,1,0,0],\
3      [1,0,0,1,0],\
4      [0,0,1,0,1],\
5      [0,1,0,0,0]]
6
7 nb_de_voisins_dans(m,[0,1],4),nb_de_voisins_dans(m,[0,1],3)#2,0

```

Question 16.

Écrire la fonction `choisir_sommet_a_colorier(g,U1,U2)` qui choisit le sommet de U_1 à colorier conformément à l'étape 1 de la boucle interne de l'algorithme RLF. La fonction supprime ce sommet de la liste U_1 ; cette liste est donc modifiée par l'appel à la fonction.

Listing 5 – Choix du sommet

```

1 m = [[0,0,1,0,0,1,1],\
2      [0,0,1,1,1,1,0],\
3      [1,1,0,1,0,0,0],\
4      [0,1,0,0,0,0,1],\
5      [0,1,0,0,0,0,0],\
6      [1,1,0,0,0,0,1],\
7      [1,0,0,0,0,1,0]]
8
9 U1, U2 = [0,1,2,3], [4,5,6]
10 print(choisir_sommet_a_colorier(m,U1,U2))# affiche 0
11 U1 # [2, 3, 1]

```

Question 17.

Écrire la fonction `single(U)` qui renvoie une copie de U sans doublon de valeur. Étudier la complexité.

Listing 6 – copie sans doublon

```

1 l = [1,0,2,3,4,1,3,1]
2 single(l)# [1, 0, 2, 3, 4]

```

Question 18.

Écrire la procédure `transfert(G,U1,U2,v)` qui prend en paramètres deux listes U_1 et U_2 et transfère 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 $v \notin U_1$.

Listing 7 – Transfert

```

1 m = [[0,1,0,0,1],\
2      [1,0,1,0,0],\
3      [1,0,0,1,0],\
4      [0,0,1,0,1],\
5      [0,1,1,0,0]]
6 U1,U2=[0,1,4],[ ]
7 print(transfert(m,U1,U2,4)) # [4]
8 U2 # [0, 1]

```

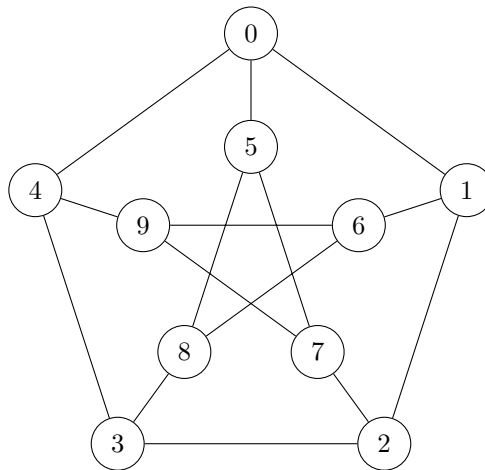
Question 19.

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

Listing 8 – RLF

```
1 m = [0, 1, 0, 0, 0, 1, 1],
2   [1, 0, 1, 0, 0, 0, 1],
3   [0, 1, 0, 1, 0, 0, 1],
4   [0, 0, 1, 0, 1, 0, 1],
5   [0, 0, 0, 1, 0, 1, 1],
6   [1, 0, 0, 0, 1, 0, 1],
7   [1, 1, 1, 1, 1, 1, 1]] # wheel graphe à 7 sommets
8
9 rlf(m) # [1, 0, 1, 0, 1, 0, 2]
```

On donne le graphe suivant, dit *graphe de Petersen* :



Question 20.

Implanter un graphe de Petersen et lancer la fonction de coloration. La réponse fournie est-elle optimale ?

Appendice

Le but de ce TP est de manipuler des graphes et non d'implémenter un algorithme de tri. L'utilisation de la méthode `sort` des listes Python est donc autorisée, ainsi que celle de son paramètre optionnel `key`.

Le paramètre `key` permet d'indiquer selon quel critère les éléments doivent être triés. Par exemple,

```
1 L.sort(key=lambda x: x[1])
```

trie une liste de couples selon leur deuxième composante.

On peut également imposer plusieurs critères en retournant un tuple. Par exemple,

```
1 L.sort(key=lambda x: (-x[0], x[1]))
```

trie d'abord selon la première composante par ordre décroissant, puis, en cas d'égalité, selon la deuxième composante par ordre croissant.