TP N°05 : COLORATION DE GRAPHES

OBJECTIFS DU TP

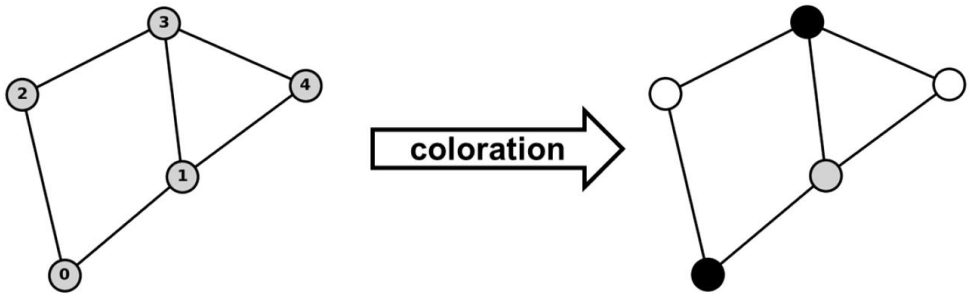
- Modéliser un problème à l'aide d'un graphe ;
- Utiliser la coloration des graphes pour résoudre un problème.

Considérons un lycée où les étudiants peuvent suivre différents enseignements disciplinaires, indépendamment de la classe à laquelle ils appartiennent. L'établissement souhaite organiser une semaine d'examens blancs. Cependant, il faut veiller à ce qu'aucun étudiant n'ait deux examens sur un même créneau horaire. Nous allons modéliser ce problème par une coloration de graphe.

Construisons un graphe dont les sommets sont les différents enseignements. Une arête relie deux sommets si les enseignements correspondants possèdent des étudiants en commun.

On va alors chercher à colorier les sommets de telle sorte que deux sommets adjacents soient de couleurs différentes. C'est ce qu'on appelle une **coloration**. On pourra alors organiser les examens correspondants aux enseignements de sommets de même couleur sur un même créneau horaire car ils n'ont pas d'étudiants en commun.

Voici un **exemple** de 3-coloration du graphe suivant :



Considérons le tableau de répartition ci-dessous. Lorsque deux enseignements possèdent des élèves en communs, la cellule correspondante contient un 1, sinon un 0. On note G(S,A) le graphe associé à ce problème.

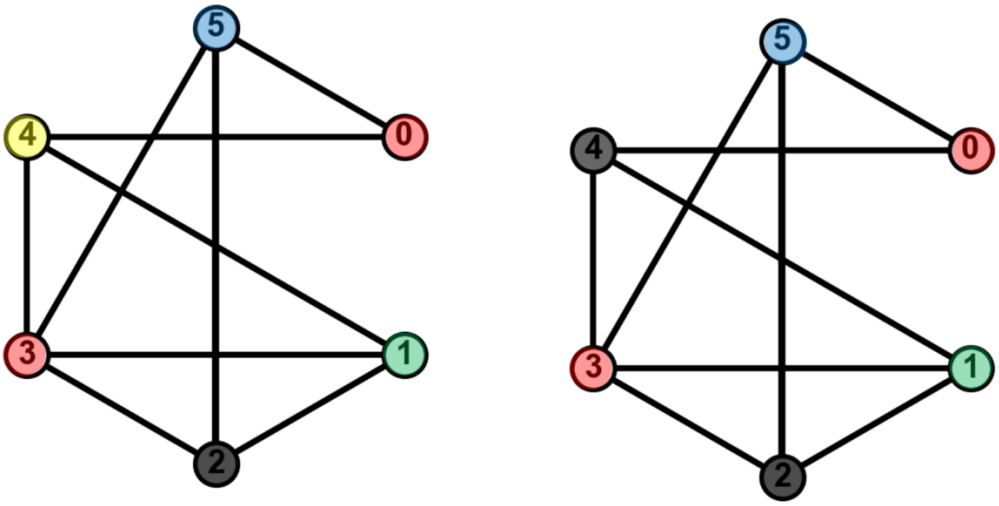
Spécialités		**LLCE**	**PC**	**SVT**	**SI**	**NSI**	**Maths**
	Sommet associé	0	1	2	3	4	5
LLCE	0	0	0	0	0	1	1
PC	1	0	0	1	1	1	0
SVT	2	0	1	0	1	0	1
SI	3	0	1	1	0	1	1
NSI	4	1	1	0	1	0	0
Maths	5	1	0	1	1	0	0

Q1. Représenter le graphe G(S,A). Proposer une 5-coloration et une 4-coloration de ce graphe.

La matrice d'adjacence de G est :

$$G = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$

5- coloration **4-coloration**



En programmation, réaliser une coloration consiste donc à construire une liste C où chaque élément C[i] correspond à la couleur du sommet s_i. En associant une leur à chaque couleur (par exemple : 0 pour blanc, 1 pour gris, 2 pour noir), la 3-coloration du graphe donné en exemple peut donc s'écrire :

C = [2,1,0,2,0]

Q2. Proposer deux listes C_5 et C_4 correspondant respectivement à une 5-coloration et une 4-coloration du graphe G(S,A). Est-il possible de proposer une 2-coloration ?

En notant les couleurs : rouge : 0, vert : 1, noir : 2, bleu : 3 et jaune : 4, on obtient les listes :

C_5 = [0,1,2,0,4,3] et C_4 = [0,1,2,0,1,3]

Le **nombre chromatique** χ associé au graphe G est le **nombre minimal de couleurs** nécessaires pour réaliser une coloration du graphe G.

On souhaite souvent utiliser un minimum de couleur pour réaliser une coloration. Dans ce problème, minimiser le nombre de couleurs revient à minimiser le nombre créneaux d'examens et ainsi minimiser le temps consacré aux examens blancs.

Q3. Justifier que quel que soit le graphe G, $1 \leq \chi(G) \leq n$, avec n l'ordre du graphe.

Un graphe non-vide G nécessite au minimum 1 couleur pour réaliser sa coloration, soit : $1 \leq \chi(G)$.

Un graphe dont tous les sommets sont de couleurs différentes vérifie nécessairement la condition $\forall$ {s,t}$\in A, c(s) \neq c(t)$ avec c(i) la couler du sommet i. Or le nombre de sommets correspond à Card(s) = n.

Document 1 : Théorème de Brooks

Il est possible d'affiner la borne supérieure de cet encadrement au moyen du théorème de Brooks :

$$\chi(G) \leq \Delta(G) + 1.$$

Avec $\Delta(G)$ le degré maximal du graphe G(S,A) :

$$\Delta(G) = \max\{ d(s)\} \, avec \, s \in S$$

Q4. Écrire une fonction `delta_max` prenant comme argument la matrice d'adjacence G associée au graphe G, et retournant le degré maximal Δ(G) de ce graphe.

```
def delta_max(G:[[int]])->int:
    """
    Retourne le degré maximal du graphe de matrice d'adjacence G
    entrée : matrice d'adjacence d'un graphe
    sortie : entier correspondant au degré maximal du graphe
    ===
    jeux de tests
    ===
    delta_max([[0,0,1],[0,0,0],[1,0,0]])
    >>> 1

    delta_max([[0,1,1],[1,0,0],[1,0,0]])
    >>> 2
    """
    n = len(G)
    d_max = 0
    for i in range(n):
        d = 0
    # On somme les "1" de la ligne i pour déterminer le nombre de voisins du
sommet s_i
        for j in range(n):
            d = d + G[i][j]
        if d > d_max:
            d_max = d
    return d_max
```

Il est possible de réaliser une coloration d'un graphe à l'aide de l'algorithme de coloration gloutonne :

- On choisit un sommet s non-colorié de G ;
- On détermine l'ensemble des voisins de ce sommet s_i ;
- En considérant les voisins déjà coloriés, on établit une liste de couleurs déjà utilisées par ce sommet ;
- On colorie ce sommet avec la première couleur (par rang croissant) dans la liste de couleurs disponibles ;
- On passe au sommet suivant, puis on répète l'opération jusqu'à obtenir un graphe totalement colorié.

Q5. Écrire une fonction **voisins** prenant comme arguments la matrice d'adjacence G associée au graphe G et le rang i du sommet s_i considéré, et retournant la liste des indices j des voisins s_j du sommet s_i.

```
def voisins(G,i):
    """
    détermine de nombre de sommets adjacents à un sommet si dans un graphe de
matrice d'adjacence G
    entrée : une matrice d'adjacence d'un graphe G et un entier i correspondant
à un sommet du graphe
    sortie : liste L contenant les numéros des sommets adjacents au sommet si

    ===
    jeux de tests
    ===
    G=[[0,1,1],[1,0,0],[1,0,0]]
    voisins(G,0)
    >>> [1,2]

    voisins(G,2)
    >>> [0]
    """
    n = len(G)
    L = []
    # Lecture de la ligne i de la matrice d'adjacence de G
    for j in range(n):
        if G[i][j] == 1:
            L = L + [j]
    return L
```

Q6. Écrire une fonction **glouton** prenant comme argument la matrice d'adjacence G associée au graphe G, et retournant une $(\Delta(G)+1)$-coloration du graphe G sous la forme d'une liste de couleurs C (chaque couleur étant repérée par un indice allant de 0 à $\Delta(G)$). On pourra utiliser la fonction **voisins**.

Si besoin, des aides pour cette question sont disponibles ci-dessous :

Aide Q6 : Vous aurez au minimum besoin de créer 4 listes différentes, bien identifiées :

- *une liste contenant la liste de toutes les couleurs, (couleur)*
- *une liste contenant des booléens pour savoir si le sommet i est coloré ou non, (colorie)*
- *une liste contenant les couleurs disponibles pour coloré un sommet i (couleur_dispo)*
- *une liste correspond à la coloration qu'il faut renvoyer. (coloration)*

```
def glouton (G:[[int]])->[int]:
    """
    renvoie une liste correspondant à une (delta_max(G)+1)-coloration du graphe
G
    entrée : la matrice d'adjacence d'un graphe G d'ordre maximal delta_max(G)
    sortie : une liste d'entiers correspondant à une coloration du graphe G
    """

    couleurs = [k for k in range(delta_max(G)+1)]
    n = len(G)
```

```
    colorie = [False for k in range(n)]
    coloration = [None for k in range(n)]
    for i in range(n):
    # Détection des couleurs déjà utilisées pour les voisins du sommets_i
        V = voisins(G, i)
        couleurs_dispo = [True for k in range(n)]
        for v in V:
            if colorie[v]:
                couleurs_dispo[coloration[v]] = False
        # Attribution d'une couleur au sommet s_i
        c = 0
        while couleurs_dispo[c] == False:
            c = c + 1
        coloration[i] = couleurs[c]
        colorie[i] = True
    return coloration
```

Q7. Vérifier le bon fonctionnement de la fonction **glouton** en utilisant l'argument optionnel **node_color** de la méthode **draw** de la bibliothèque **networkx**. Par exemple, en supposant le graphe G d'ordre 4 déjà déclaré, la suite d'instructions :

```
>>> import networkx as nx
>>> import matplotlib.pyplot as plt
>>> nx.draw(G, node_color = ["r", "g", "r", "g"])
>>> plt.show()
```

conduit à l'affichage d'une coloration du graphe G où les sommets d'indice pair sont affichés en rouge et les sommets d'indice impair sont affichés en vert.

Sous-questions (optionnelles) pour vous aider :

a. *Ecrire une liste d'instruction permettant de générer aléatoirement la matrice d'adjacence d'un graphe non-orienté G de d'ordre n. On pourra utiliser la fonction* **randint** *de la bibliothèque* **random**.

```
# Choix de l'ordre du graphe
n = 7
# Génération d'un graphe non-orienté
import random
G = [[random.randint(0, 1) if i != j else 0 for j in range(n)] for i in range(n)]

for i in range(n):
    for j in range(n):
        G[i][j] = G[j][i]
```

b. *Effectuer la coloration du graphe.*

```
C = glouton(G)
```

c. *Ecrire une liste d'instruction permettant de générer une liste de couleurs utilisable en argument* **node_color** *de la méthode* **draw** *quel que soit l'ordre du graphe G.*

D'après la documentation de la méthode draw :

```
node_color : color or array of colors (default='#1f78b4')
Node color. Can be a single color or a sequence of colors with the same
length as nodelist.
```

```
Color can be string or rgb (or rgba) tuple of floats from 0-1. If numeric values
are specified they will be mapped to colors using the cmap and vmin,vmax
parameters. See matplotlib.scatter for more details.
```

Il devient difficile de créer une liste de couleur au format chaine de caractère lorsqu'on ne connait pas le nombre de couleurs en avance. Il est alors plus pertinent d'utiliser la notation des couleurs sous forme de tuples de flottants compris entre 0 et 1 correspondant au code RGB :

```
base = [(1, 0, 0), (0, 1, 0), (0, 0, 1), (1, 1, 0), (1, 0, 1), (0, 1, 1)]
couleurs = base[:]
k = 2
while len(couleurs) < n:
    couleurs = couleurs + [(c[0]/k, c[1]/k, c[2]/k) for c in base]
    k = k + 1
coloration = [couleurs[c] for c in C]
```

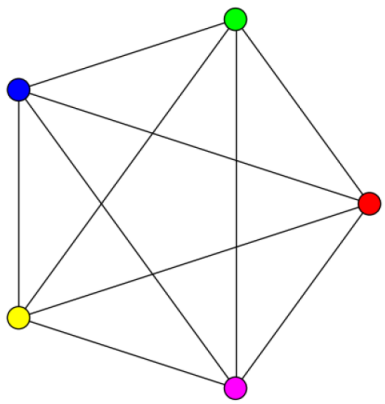
d. *Ecrire une liste d'instruction permettant d'afficher la coloration du graphe en utilisant l'argument optionnel* ***node_color*** *de la méthode* ***draw*** *de la bibliothèque* ***networkx***.

```
# Conversion en graphe pour networkx
import networkx as nx
import numpy as np

graphe = nx.from_numpy_array(np.array(G))

# Affichage
import matplotlib.pyplot as plt
nx.draw(graphe, node_color = coloration,pos=nx.layout.circular_layout(graphe))
plt.axis("equal") #facultatif
plt.show()
```

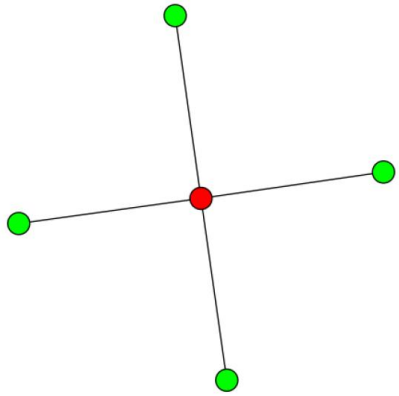
Comme indiqué par le théorème de Brooks, la (Δ(G)+1)-coloration du graphe G obtenue par la mise en œuvre d'un algorithme glouton n'est pas toujours optimale.

Q8. Proposer :

a. Un exemple de graphe G d'ordre 5 de nombre chromatique $\chi(G) = \Delta(G) + 1$;



b. Un exemple de graphe G d'ordre 5 de nombre chromatique $\chi(G) < \Delta(G) + 1$.



Q9. Proposer un algorithme permettant de vérifier, par force brute, si un graphe G admet une k-coloration, pour une valeur de k donnée ($1 \leq k \leq \Delta(G)+1$). Quelle est la complexité de cet algorithme (exprimée en notation de Landau) dans le pire des cas ?

Remarque : L'attaque par force brute est une méthode utilisée en cryptanalyse pour trouver un mot de passe ou une clé. Il s'agit de tester, une à une, toutes les combinaisons possibles jusqu'à en trouver une qui fonctionne.

Un algorithme naïf permettant de vérifier si une k-coloration est légale consiste à :

- Choisir une couleur parmi k couleurs pour chaque sommet du graphe (sans se soucier des couleurs des autres sommets) ;
- Vérifier si cette coloration est légale en testant la condition $c(s) \neq c(t)$ pour chaque paire de sommets $(s,t) \in S^2$ définissant une arête $\{s,t\} \in A$;
- si la coloration proposée n'est pas légale, on recommence en choisissant un autre jeu de couleurs pour les différents sommets.

L'algorithme s'arrête soit quand on a trouvé une coloration légale, soit quand on a testé toutes les colorations possibles.

Dans le pire des cas, il n'existe pas de k-coloration légale, il faut alors toutes les tester :
Pour un graphe d'ordre n, il y a **k^n colorations à tester**. Or pour chaque coloration, il faut tester tous les sommets 2 à 2, pour vérifier $c(s) \neq c(t)$. Un graphe d'ordre n comporte au maximum une arête entre chaque sommet soit $\binom{n}{2} = \frac{n!}{2!(n-2)!} = \frac{n(n+1)}{2}$ **arêtes à tester.**

La **complexité globale de cet algorithme est $\mathcal{O}(n^2.k^n)$**

Q10. Écrire une fonction `force_brute` prenant comme argument la matrice d'adjacence G associée au graphe G et un entier k, et retournant une k-coloration du graphe G sous la forme d'une liste de couleurs C (chaque couleur étant repérée par un indice allant de 0 à k -1). Si le graphe G n'admet pas de k-coloration, la fonction coloration retourne une liste vide. On pourra utiliser la fonction `voisins`.

```
def force_brute (G:[[int]],k:int)-> [int]:
    """
    retourne une k-coloration du graphe G sous forme d'une liste d'entier représentant les couleurs des sommets si le graphe admet une k-coloration et une liste vide sinon

```

```
    entrée : G matrice d'adjacence d'un graphe non-orienté G et k un entier correspondant au nombre de couleurs à disposition

    sortie : liste d'entiers correspondant à une k-coloration du graphe s'il en admet une, une liste vide, sinon
    ===
    Jeux de tests
    ===
    G = [[0,1,0,1],[1,0,1,0],[0,1,0,1],[1,0,1,0]]
    force_brute(G,1)
    >>>[]
    force_brute(G,2)
    >>>[1,0,1,0]

    """
    n = len(G)
    coloration = [0 for _ in range(n)]
    trouve = False


    while not trouve and coloration !=[] :
        # au départ, par défaut, il n'y a pas de conflit entre la couleur d'un sommet et l'un de ses voisins :
        coloration_OK = True

        # Vérification que chaque sommet est de couleur différente de ses voisins
        for i in range (n):
            V = voisins(G,i)

            for v in V :
                if coloration[i]==coloration[v]:
                    # la coloration proposée n'est pas valable
                    coloration_OK = False

        # La coloration trouvée est valable
        if coloration_OK == True :
            trouve = True

        # si la coloration n'est pas valable, on teste la suivante :
        if not trouve :
            r = 0
            # on incrémente le premier sommet avec la valeur suivante
            coloration[0]= coloration[0]+1

        # si un sommet dépasse le numéro des couleurs possibles :
        while coloration !=[] and coloration[r]==k :

            # on lui attribue la couleur n°0
            coloration[r]=0
            if r+1 != n :
                # et on incrémente le sommet suivant
```

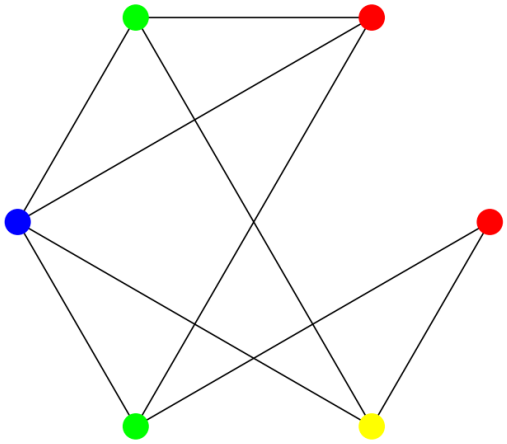
```
                coloration[r+1] = coloration [r+1]+1
                r = r+1

            # si le dernier sommet dépasse le nombre de couleurs autorisées,
alors on a testé toutes les colorations possibles, il n'existe pas de k-
coloration
            else :
                coloration = []

    return(coloration)
```

Q11. Ecrire une fonction `coloration_optimale` prenant comme argument la matrice d'adjacence G associée au graphe G, et retournant une Δ(G)-coloration du graphe G sous la forme d'une liste de couleurs C (chaque couleur étant repérée par un indice allant de 0 à Δ(G) - 1). On utilisera la fonction `force_brute`.

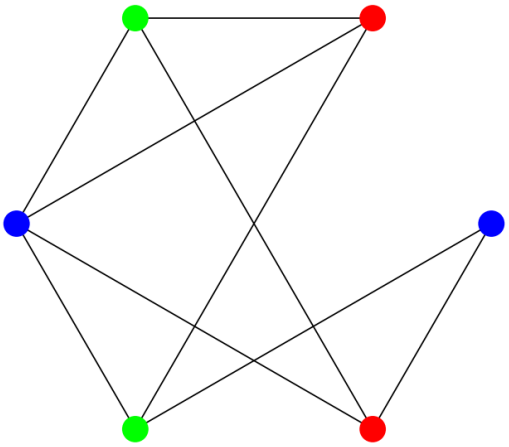
```
def coloration_optimale(G: [[int]]) -> [int]:
    """
    Déterminer le nombre chromatique chi(G) associé au graphe G et
    fournit une chi(G)-coloration du graphe G
    ===
    Entrée :
    --------
    G : matrice d'adjacence associée au graphe ([[int]])
    Sortie :
    --------
    liste des couleurs c_i (i allant de 0 à chi(G)-1) associées à chaque
    sommet s_i dans la chi(G)-coloration du graphe ([int])
    ===
    Jeux de tests :
    ---------------
    >>> G = [[0, 1, 0, 1], [1, 0, 1, 0], [0, 1, 0, 1], [1, 0, 1, 0]]
    >>> coloration_optimale(G)
    [1, 0, 1, 0]
    """
    k = 1
    coloration = []
    while coloration == []:
        coloration = force_brute(G, k)
        k = k + 1
    return coloration
```

Q12. Comparer les résultats obtenus pour les fonctions `glouton` et `coloration_optimale` sur le problème posé. Conclure.

Avec la méthode « glouton »



Avec la fonction coloration optimale



On constate que la méthode gloutonne est juste mais pas toujours optimale. Il faudra donc au minimum 3 créneaux horaires différents pour organiser les examens blancs dans le lycée.