Coloration de graphes – Partie Python

Consignes

Vous enregistrez votre fichier sous le nom `coloration-NOM-PRENOM.py`.
À la fin de la séance, copier votre fichier dans le dossier `devoir`.
Vous pouvez rajouter des commentaires lorsque vous pensez que c'est nécessaire. De même, essayez de choisir des variables dont le nom est compréhensible, si c'est pertinent.
Vous pouvez aussi copier dans des commentaires les tests effectués pour vérifier vos fonctions.

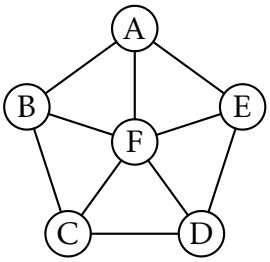
Représentation des graphes

Pour représenter les graphes, nous allons utiliser et compléter la classe ci-dessous.

```
class Graphe:
    def __init__(self):
        self.voisins = dict()  # Associe à chaque sommet la liste de ses voisins
        self.sommets = []  # Liste des sommets
        self.coloration = dict()  # Associe une couleur à chaque sommet
        self.couleurs_voisins = dict()  # Associe à chaque sommet un dictionnaire
                                        # associant à chaque couleur le nombre de
                                        # voisins qui ont cette couleur.

    def ajouter_arete(self, sommet1, sommet2):
        if sommet1 not in self.voisins:  # On ne connaît pas ce sommet
            self.voisins[sommet1] = []
            self.sommets.append(sommet1)
            self.couleurs_voisins[sommet1] = dict()
        if sommet2 not in self.voisins:
            self.voisins[sommet2] = []
            self.sommets.append(sommet2)
            self.couleurs_voisins[sommet2] = dict()
        if sommet2 not in self.voisins[sommet1]:  # On n'a pas encore cette arête
            self.voisins[sommet1].append(sommet2)
            self.voisins[sommet2].append(sommet1)
```

Le graphe suivant est obtenu à l'aide des instructions ci-contre.




```
graphe1 = Graphe()
graphe1.ajouter_arete("A", "B")
graphe1.ajouter_arete("B", "C")
graphe1.ajouter_arete("C", "D")
graphe1.ajouter_arete("D", "E")
graphe1.ajouter_arete("E", "A")
graphe1.ajouter_arete("A", "F")
graphe1.ajouter_arete("B", "F")
graphe1.ajouter_arete("C", "F")
graphe1.ajouter_arete("D", "F")
graphe1.ajouter_arete("E", "F")
```

Exercice 1 (sur papier) : Dessiner sur la feuille une représentation correspondant au graphe suivant :

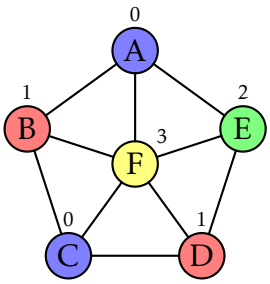
```
graphe2 = Graphe()
graphe2.ajouter_arete("A", "B")
graphe2.ajouter_arete("B", "C")
graphe2.ajouter_arete("C", "D")
graphe2.ajouter_arete("D", "E")
graphe2.ajouter_arete("E", "A")
graphe2.ajouter_arete("A", "G")
graphe2.ajouter_arete("B", "F")
graphe2.ajouter_arete("D", "G")
graphe2.ajouter_arete("D", "F")
graphe2.ajouter_arete("E", "G")
graphe2.ajouter_arete("C", "F")
```

Exercice 2 : Rajouter dans la classe `Graphe` la méthode `degre` qui prend comme paramètre, en plus de `self`, un sommet `sommet` et renvoie le degré de ce sommet dans le graphe. On rappel que le degré d'un sommet est le nombre de ses voisins.

```
>>> graphe1.degre("A")
3
>>> graphe1.degre("F")
5
>>> graphe2.degre("D")
4
```

Coloration des sommets

Une **coloration d'un graphe** consiste à associer une "couleur" à chaque sommet de telle sorte qu'il n'y ait pas deux sommets voisins de même couleur. Pour simplifier, nous utiliserons les entiers naturels comme "couleurs". Ainsi la première couleur sera 0, la deuxième 1, et ainsi de suite. On appelle **coloration** une association d'une couleur à chaque sommet. Lorsqu'il existe une coloration du graphe avec k couleurs, on dit qu'il est k-**colorable**. L'exemple ci-contre est une 3-coloration de `graphe1`.



Dans la classe `Graphe`, l'attribut `coloration` est un dictionnaire qui associe le nom des sommets colorés à leur couleur. L'attribut `couleurs_voisins` est aussi un dictionnaire qui associe chaque sommet à un dictionnaire reliant les couleurs au nombre de voisins ayant cette couleur. Si aucun voisin d'un sommet `s` n'a la couleur `coul` dans un graphe `graphe`, alors `graphe.couleurs_voisins[s]` n'a pas de clef `coul`.
La coloration suivante correspond à l'exemple ci-dessus.

```
>>> graphe1.coloration
{'A': 0, 'B': 1, 'C': 0, 'D': 1, 'E': 2, 'F': 3}
>>> graphe1.couleurs_voisins
{'A': {1: 1, 2: 1, 3: 1},
 'B': {0: 2, 3: 1},
 'C': {1: 2, 3: 1},
 'D': {0: 1, 2: 1, 3: 1},
 'E': {0: 1, 1: 1, 3: 1},
 'F': {0: 2, 1: 2, 2: 1}}
```

On peut, par exemple, voir que F a deux voisins qui ont la couleur 0, deux qui ont la couleur 1 et un qui a la couleur 2.

Exercice 3 (sur papier) : Faire l'exercice sur la feuille.

Exercice 4 : Rajouter dans votre fichier les commandes permettant de créer le `graphe3` de l'exercice 3.

Exercice 5 : Copier et compléter le code de la méthode `colore` qui prend, en plus de **`self`**, un nom de sommet `s` et une couleur `coul` et qui :

- ne fait rien si ce sommet a déjà une couleur ;
- associe cette couleur à ce sommet dans `coloration` s'il n'a pas encore de couleur et parcours tous les voisins `v` de `s` pour :
  - rajouter 1 au compteur de `coul` dans **`self`**`.couleurs_voisins[v]` si cette couleur y est déjà ;
  - mettre **`self`**`.couleurs_voisins[v][coul]` à 1 sinon.

```
    def colore(self, s, coul):
        if ...:
            #print(f"On colore {s} avec {coul}")  # Pour débuguer
            self.coloration[...] = ...
            for v in self.voisins[s]:
                if coul in self.couleurs_voisins[...]:
                    ...
                else:
                    ...
```

La coloration ci-dessus est obtenue ainsi :

```
>>> graphe1.colore("A", 0)
>>> graphe1.colore("B", 1)
>>> graphe1.colore("C", 0)
>>> graphe1.colore("D", 1)
>>> graphe1.colore("E", 2)
>>> graphe1.colore("F", 3)
```

Afin de pouvoir tester plusieurs colorations, il faut pouvoir réinitialier la coloration. Pour cela, vous pouvez utiliser la méthode ci-dessous à la classe `Graphe` :

```
    def reinitialiser_coloration(self):
        self.coloration = dict()
        self.couleurs_voisins = {s: dict() for s in self.sommets}
```

Exercice 6 : Rajouter à la classe une méthode `coloration_valide` à la classe `Graphe`, qui ne prend pas d'autre paramètre que **`self`** et qui renvoie un booléen indiquant si l'attribut `coloration` correspond à une coloration valide. Une coloration valide doit vérifier les propriétés suivantes :

- Chaque sommet de l'attribut `sommets` est une clef de l'attribut `coloration` ;
- Pour chaque sommet `s1` et chacun de ses voisins `s2`, il faut que la couleur associée à `s1` soit différente à celle associée à `s2`.

```
>>> graphe1.colore("A", 0)
>>> graphe1.colore("B", 1)
>>> graphe1.colore("C", 0)
```

```
>>> graphe1.colore("D", 1)
>>> graphe1.colore("E", 2)
>>> graphe1.coloration_valide()   # Il manque un sommet
False
>>> graphe1.colore("F", 3)
>>> graphe1.coloration_valide()
True
>>> graphe1.reinitialiser_coloration()   # On fait une autre coloration
>>> graphe1.colore("A", 0)
>>> graphe1.colore("B", 1)
>>> graphe1.colore("C", 0)
>>> graphe1.colore("D", 1)
>>> graphe1.colore("E", 2)
>>> graphe1.colore("F", 0)   # Cette couleur n'est pas valide
>>> graphe1.coloration_valide()
False
```

Algorithme Welsh-Powell

Il n'existe pas d'algorithmes capable de colorer n'importe quel graphe avec le moins de couleurs possible avec une complexité qui n'est pas exponentielle. Par contre, il existe plusieurs algorithmes gloutons qui donnent un résultat rapidement mais sans garantie d'optimalité. L'algorithme de Welsh-Powell est un des algorithmes gloutons de coloration les plus connus. Il fonctionne ainsi :

- On trie les sommets dans l'ordre décroissant de leur degré. Ceux avec le plus grand degré sont en premier et ceux avec le plus petit degré sont en dernier. Les sommets seront toujours parcourus dans cet ordre.
- On prend le premier sommet et on lui donne la plus petite couleur c disponible.
- On parcourt ensuite tous les sommets suivants. À chaque fois qu'on voit un sommet non coloré et qui n'a pas de voisin coloré avec c, on le colore avec c.
- On passe au premier sommet non encore coloré et on recommence.

Pour simplifier les exemples, on supposera que les sommets de même degré seront parcourus dans l'ordre alphabétique.

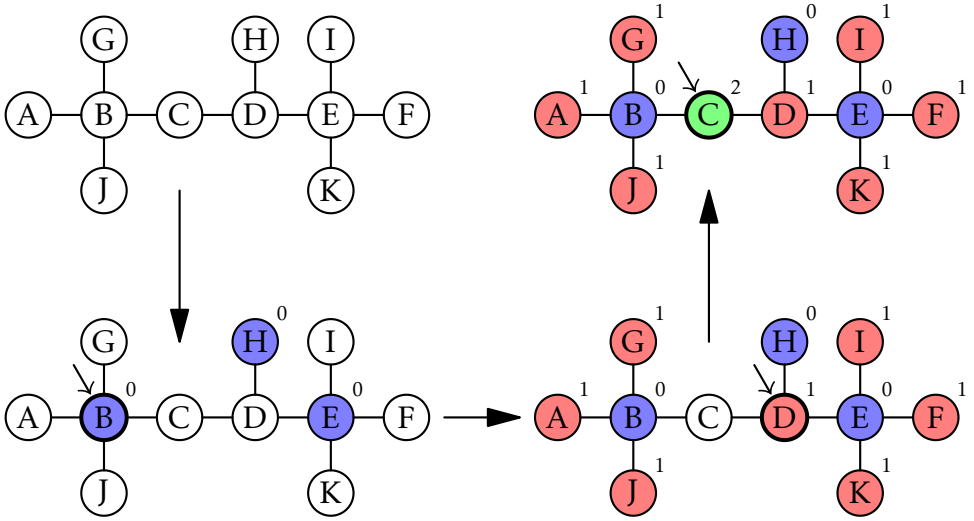
Le tableau ci-contre montre les différentes étapes pour la coloration de `graphe3`.

- Tout d'abord on ordonne les sommets par degré décroissant et on obtient l'ordre du tableau.

Sommets	B	E	D	C	A	F	G	H	I	J	K
Degrés	4		3	2	1						
Étape 1	0	0	-	-	-	-	-	0	-	-	-
Étape 2			1	-	1	1	1		1	1	1
Étape 3				2							
Coloration	0	0	1	2	1	1	1	0	1	1	1

- On commence par B à qui on donne la couleur 0. Le sommet suivant, E, n'est pas voisin avec B, donc on peut aussi le colorer avec 0. Tous les sommets suivants sont voisins de B ou de E, à l'exception de H, qu'on colore aussi en 0.
- On recommence avec le premier sommet non coloré, qui est D et on colore tous les autres sommets, à l'exception de C, qui n'ont pas d'arêtes les reliant aux autres sommets colorés en 1.
- On recommence avec C, qui est le dernier, donc on a fini.

La figure suivante illustre ces étapes.

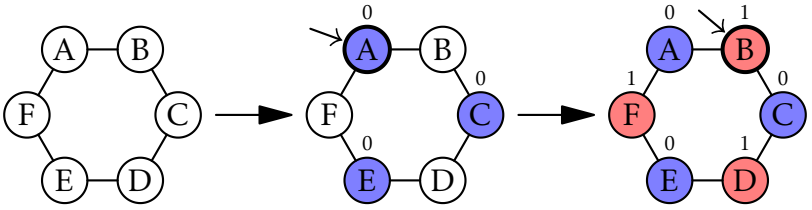


Exercice 7 (sur papier) : Faire l'exercice sur la feuille.

Comme tous les algorithmes gloutons, l'algorithme de Welsh-Powell ne garantit pas de trouver une coloration optimale. C'est le cas de `graphe3` qui est 2-colorable. Dans certains cas, la qualité de la coloration dépend de l'ordre dans lequel sont nommés les sommets.

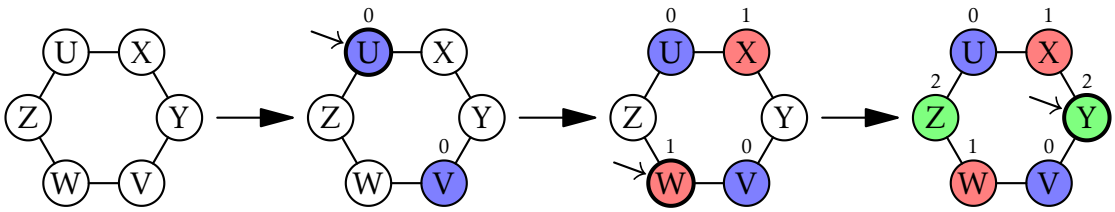
Dans l'exemple ci-desous, tous les sommets ont le même degré. L'ordre de traitement des sommets dépend intégralement de l'ordre de nommage des sommets. On commence par le sommet A avec la couleur 0. On ne peut pas colorer B qui est son voisin mais on colore C puis E. On passe à l'étape suivante en colorant successivement B, D et F. On obtient donc une 2-coloration.

Sommets	A	B	C	D	E	F
Étape 1	0	-	0	-	0	-
Étape 2		1		1		1
Coloration	0	1	0	1	0	1



Par contre avec l'ordre ci-dessous, on peut colorer U puis V avec la couleur 0. Ensuite, on colore W et X avec la couleur 1. Et on finit avec la couleur 2 pour Y et Z. On a donc uee 3-coloration.

Sommets	U	V	W	X	Y	Z
Étape 1	0	0	-	-	-	-
Étape 2			1	1	-	-
Étape 3					2	2
Coloration	0	0	1	1	2	2



Exercice 8 (sur papier) : Faire l'exercice sur la feuille.

Exercice 9 : Recopier et compléter dans la classe `Graphe` la méthode `plus_petite_couleur_dispo` qui prend comme paramètre, en plus de **`self`**, un sommet `sommet` et renvoie la plus petite couleur qui n'apparaît pas encore dans ses voisins. Pour rappel, **`self`**`.couleurs_voisins[sommet]` est un dictionnaire qui associe à des couleurs le nombre de voisins de `sommet` qui ont cette couleur. On ne testera pas si le sommet a déjà une couleur.

```
    def plus_petite_couleur_dispo(self, sommet):
        coul = 0
        while ...:
            coul += 1
        return coul
```

```
>>> graphe1.colore("A", 0)
>>> graphe1.colore("B", 1)
>>> graphe1.plus_petite_couleur_dispo("C")
0
>>> graphe1.plus_petite_couleur_dispo("E")
1
>>> graphe1.plus_petite_couleur_dispo("F")
2
```

Exercice 10 : Compléter la méthode `welsh_powell` qui utilise l'algorithme de Welsh-Powell pour déterminer une coloration du graphe. Pour ne pas alourdir la fonction, on ne cherchera pas à déterminer l'ordre de parcours des sommets de même degré. À cause de cela, selon la façon dont est défini le graphe, le résultat peut varier. Vous pouvez donc avoir des résultats différents que ceux des exemples.

```
    def welsh_powell(self):
        self.reinitialiser_coloration()   # On efface toutes les couleurs
        # On ordonne les sommets par degré
        liste_sommets = sorted(self.sommets, key=self.degre, reverse=True)
        for i in range(len(liste_sommets)):
            sommet = liste_sommets[i]
            if ...:   # Le sommet n'a pas de couleur
                coul = ...   # On prend la plus petite couleur
                ...   # On colore le sommet
                for j in range(i+1, len(liste_sommets)):
                    sommet2 = liste_sommets[j]
                    if ...:   # On peut colorer sommet2 avec coul
                        ...   # On le colore
```

```
>>> graphe1.welsh_powell()
>>> graphe1.coloration
{'F': 0, 'A': 1, 'C': 1, 'B': 2, 'D': 2, 'E': 3}
>>> graphe2.welsh_powell()
>>> graphe2.coloration
{'D': 0, 'A': 0, 'B': 1, 'E': 1, 'C': 2, 'G': 2, 'F': 3}
>>> graphe3.welsh_powell()
>>> graphe3.coloration
{'B': 0, 'E': 0, 'H': 0, 'D': 1, 'A': 1, 'F': 1, 'G': 1, 'I': 1, 'J': 1, 'K': 1, 'C': 2}
```

Algorithme D-SATUR

Le résultat de l'algorithme de Welsh-Powell peut dépendre de l'ordre dans lequel sont parcourus les sommets de même degré. Au lieu de fixer l'ordre de parcours à l'avance, l'algorithme D-SATUR va déterminer à chaque étape quel est le sommet prioritaire à colorer. À chaque fois, c'est le sommet qui a le plus de couleurs apparaissant parmi les voisins qui est sélectionné. En cas d'égalité, on prend le sommet de degré maximal. On assigne la plus petite couleur disponible pour le sommet traité. On recommence jusqu'à avoir coloré tous les sommets.
Voici les étapes de la coloration de `graphe1`. On place F en premier, puisqu'il est de plus haut degré.

	Couleurs des voisins						
Sommets	F	A	B	C	D	E	Action
Étape 1							F ← 0
Étape 2		{0}	{0}	{0}	{0}	{0}	A ← 1
Étape 3			{0,1}	{0}	{0}	{0,1}	B ← 2
Étape 4				{0, 2}	{0}	{0,1}	C ← 1
Étape 5					{0,1}	{0,1}	D ← 2
Étape 6						{0,1,2}	E ← 3

Les sommets ne sont pas forcément parcourus dans le même ordre qu'avec l'algorithme de Welsh-Powell. C'est le cas lors de la coloration de `graphe3`. Les sommets sont déjà classés dans l'ordre de parcours de Welsh-Powell. En cas d'égalité de deux sommets sur le nombre de couleurs utilisés par les voisins, on prendra le sommet le plus à gauche.

	Couleurs des voisins											
Sommets	B	E	D	C	A	F	G	H	I	J	K	Action
Étape 1												B ← 0
Étape 2				{0}	{0}		{0}			{0}		C ← 1
Étape 3			{1}		{0}		{0}			{0}		D ← 0
Étape 4		{0}			{0}		{0}	{0}		{0}		E ← 1
Étape 5					{0}	{1}	{0}	{0}	{1}	{0}	{1}	A ← 1
Étape 6						{1}	{0}	{0}	{1}	{0}	{1}	F ← 0
Étape 7							{0}	{0}	{1}	{0}	{1}	G ← 1
Étape 8								{0}	{1}	{0}	{1}	H ← 1
Étape 9									{1}	{0}	{1}	I ← 0
Étape 10										{0}	{1}	J ← 1
Étape 11											{1}	K ← 0

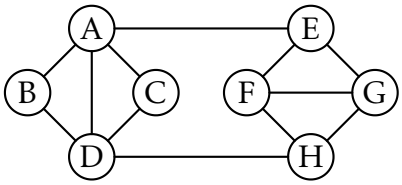
On obtient bien une 2-coloration, comme attendu et pas une 3-coloration comme avec l'algorithme de Welsh-Powell.

Exercice 11 : Compléter le code de la méthode `dsatur`. On commence par classer les sommets par ordre décroissant de degré. Ainsi, en cas d'égalité sur le nombre maimal de couleurs utilisées dans les voisins, c'est le premier sommet qui est gardé, et qui a donc un degré supérieur ou égal aux suivants. On rappelle que `liste.remove(x)` supprime la première occurrence de `x` dans `liste` ou lève une erreur si `x` n'y est pas.

```
    def dsatur(self):
        # On trie les sommets par ordre décroissant de degré
        sommets_restants = sorted(self.sommets, key=self.degre, reverse=True)
        while len(sommets_restants) > 0:
            # On cherche le sommet qui a le plus de couleurs utilisées dans les voisins
            meilleur = 0
            for s in sommets_restants:
                nb_couleurs_voisins = len(self.couleurs_voisins[...])
                if ...:
                    meilleur = ...
                    prochain_a_traiter = ...
            self.colore(..., ...)
            sommets_restants.remove(...)   # On enlève le sommet traité
```

```
>>> graphe1.dsatur()
>>> graphe1.coloration
{'F': 0, 'A': 1, 'B': 2, 'C': 1, 'D': 2, 'E': 3}
>>> graphe2.dsatur()
>>> graphe2.coloration
{'D': 0, 'C': 1, 'F': 2, 'B': 0, 'A': 1, 'E': 2, 'G': 3}
>>> graphe3.dsatur()
>>> graphe3.coloration
{'B': 0, 'C': 1, 'D': 0, 'E': 1, 'A': 1, 'F': 0, 'G': 1, 'H': 1, 'I': 0, 'J': 1, 'K': 0}
```

Cet algorithme, étant un algorithme glouton, ne donne pas systématiquement la meilleure coloration. C'est le cas avec le graphe ci-contre, qu'on appellera `graphe4`.



Exercice 12 (sur papier) : Faire l'exercice sur la feuille.

La définition de `graphe4` est la suivante :

```
graphe4 = Graphe()
graphe4.ajouter_arete("A", "B")
graphe4.ajouter_arete("A", "C")
graphe4.ajouter_arete("A", "D")
graphe4.ajouter_arete("B", "D")
graphe4.ajouter_arete("C", "D")
graphe4.ajouter_arete("A", "E")
graphe4.ajouter_arete("E", "F")
graphe4.ajouter_arete("E", "G")
graphe4.ajouter_arete("F", "G")
graphe4.ajouter_arete("F", "H")
graphe4.ajouter_arete("G", "H")
graphe4.ajouter_arete("D", "H")
```

Algorithme avec backtracking

Lorsque le graphe est de taille suffisamment petite, il est raisonnable d'essayer de chercher une coloration de taille minimale à l'aide d'un algorithme exhaustif, par exemple par backtracking. On va utiliser cela pour tester si le graphe est k-colorable, avec un k donné.
Le problème de la plupart des algorithmes gloutons, comme D-SATUR vient du fait qu'ils attribuent la plus petite couleur disponible, donc potentiellement déjà donnée, alors que parfois, il faut attribuer une nouvelle couleur à un sommet pour éviter des problèmes plus tard. Nous allons utiliser une variation de D-SATUR. On part d'une coloration partielle.

- Si tous les sommets sont colorés, on a trouvé une k-coloration.
- Sinon, on cherche le sommet qui a le plus de couleurs utilisées parmi les voisins et éventuellement de degré maximal.
- On colore successivement le sommet avec toutes les couleurs déjà utilisées encore disponibles et on regarde si le reste du graphe est k-colorable.
  - Si c'est le cas, on a finit et on renvoie vrai.
  - Sinon on efface la couleur du sommet et on teste la couleur suivante.
- Si aucune des couleurs déjà utilisées ne permet de colorer le reste du graphe et si on en utilise moins de k, on essaie une nouvelle couleur.

Exercice 13 : Compléter la méthode `decolore` qui prend **`self`** et un sommet `s` et qui enlève la couleur associée à `s`, s'il en a une, et qui diminue le compteur associé à cette couleur de chacun de ces voisins.

```
    def decolore(self, s):
        if s in self.coloration:   # On vérifie que le sommet est coloré
            coul = self.coloration.pop(s)   # On enlève sa couleur
            for v in self.voisins[s]:
                # On diminue le compteur de la couleur de chaque voisin
                self.couleurs_voisins[...][...] -= ...
                if self.couleurs_voisins[...][...] == ...:
                    del self.couleurs_voisins[...][...]   # On efface la couleur
```

```
>>> graphe1.colore("A", 0)
>>> graphe1.colore("B", 1)
>>> graphe1.colore("C", 0)
>>> graphe1.colore("E", 1)
>>> graphe1.couleurs_voisins
{'A': {1: 2}, 'B': {0: 2}, 'C': {1: 1}, 'D': {0: 1, 1: 1},
 'E': {0: 1}, 'F': {0: 2, 1: 2}}
>>> graphe1.decolore("A")
>>> graphe1.couleurs_voisins
{'A': {1: 2}, 'B': {0: 1}, 'C': {1: 1},
 'D': {0: 1, 1: 1}, 'E': {}, 'F': {0: 1, 1: 2}}
>>> graphe1.coloration
{'B': 1, 'C': 0, 'E': 1}
```

Exercice 14 : Compléter le code de la méthode récursive `k_colorable_rec` qui prend en argument, en plus de **`self`**, un entier `k` et un entier `couleur_max_utilisable`. Cette méthode renvoie un booléen indiquant si on a trouvé une `k`-coloration ou non. Le prochain sommet à colorer pourra avoir au maximum la couleur `couleur_max_utilisable`.

```
    def k_colorable_rec(self, k, couleur_max_utilisable):
        if len(self.coloration) == len(self.sommets):
            return ...   # On a trouvé une k-coloration
        # On cherche le sommet qui a le plus de couleurs utilisées dans les voisins
        meilleur = (-1, 0)   # (Nb couleurs des voisins, degré)
        for s in self.sommets:
            if ...:   # On ne regarde plus les sommets colorés
                valeurs = (len(self.couleurs_voisins[...]), self.degre(...))
                if ...:
                    meilleur = ...
                    prochain_a_traiter = ...
        # On va tester toutes les couleurs encore possibles
        for coul in range(couleur_max_utilisable+1):
            if coul not in ...: # La couleur n'est pas dans les voisins
                self.colore(..., ...)
                # On sauvegarde la couleur max
                ancienne_couleur_max = couleur_max_utilisable
                if (coul == couleur_max_utilisable
                    and couleur_max_utilisable < k-1):
                    ...   # Il faut donc rajouter une nouvelle couleur
                if self.k_colorable_rec(k, couleur_max_utilisable):
                    return ...   # On a trouvé une k-coloration
                self.decolore(...)   # On enlève la couleur
                couleur_max_utilisable = ...   # On rétablit la couleur max
        return ...   # On a tout testé et rien trouvé

    def k_colorable(self, k):
        self.reinitialiser_coloration()
        return self.k_colorable_rec(k, 0)
```

```
>>> graphe1.k_colorable(2)
False
>>> graphe1.k_colorable(3)
False
>>> graphe1.k_colorable(4)
True
>>> graphe1.coloration
{'F': 0, 'A': 1, 'B': 2, 'C': 1, 'D': 2, 'E': 3}
```

Exercice 15 (sur papier) : Faire l'exercice sur la feuille.

On appelle **nombre chromatique** d'un graphe le nombre minimal de couleurs qu'il faut pour le colorer.

Exercice 16 : Écrire une méthode `nombre_chromatique` qui ne prend pas de paramètre autre que `self` et qui renvoie le nombre chromatique du graphe considéré. On utilisera la méthode `k_colorable`.

```
>>> graphe2.nombre_chromatique()
4
>>> graphe3.nombre_chromatique()
2
```

Organisation du bac blanc

Au lycée, si on veut faire un bac blanc des spécialités où il n'y a qu'un seul sujet par spécialité, il faut que tous les élèves faisant la même spécialité passent l'épreuve en même temps. Cela a pour conséquence que deux épreuves de spécialités ne peuvent pas avoir lieu en même temps si un élève fait ces deux spécialités.
Mais il faut aussi essayer de limiter le nombre de demi-journées bloquées. Au lycée, il y a 8 spécialités. On peut donc faire le bac blanc en bloquant 8 demi-journées. Sauf que cela veut dire que chaque élève de Tle a 6 demi-journées pendant lesquelles il n'a pas cours. Il faut donc essayer de mettre le plus possible d'épreuves en même temps.
Voici la liste des couples de spécialités pris cette année par les élèves de Tle.

```
liste_couples = [
    ('AMC', 'HGGSP'), ('AMC', 'HLPHI'), ('AMC', 'MATHS'), ('AMC', 'NSI'), ('AMC', 'SES'),
    ('AMC', 'SVT'), ('HGGSP', 'HLPHI'), ('HGGSP', 'MATHS'), ('HGGSP', 'NSI'),
    ('HGGSP', 'SES'), ('HLPHI', 'SES'), ('MATHS', 'NSI'), ('MATHS', 'PH-CH'),
    ('MATHS', 'SES'), ('MATHS', 'SVT'), ('NSI', 'PH-CH'), ('NSI', 'SES'), ('NSI', 'SVT'),
    ('PH-CH', 'SVT'), ('SES', 'SVT')]
```

Exercice 17 (sur papier) : Faire l'exercice sur la feuille.

Barrettes de spécialités en première

Pour pouvoir faire les emplois du temps en première et terminale, il faut mettre les spécialités en barrettes. On dit que des cours sont en barrette lorsqu'ils ont lieu en même temps et que les élèves d'un certains nombre de classes sont répartis dans ces différents cours. En première, comme il y a 3 spécialités, il faut faire 3 barrettes.
Un logiciel a réparti les élèves dans les différents groupes de spécialités.

```
liste_triplets = [
('AMC1', 'HGGSP2', 'HLPHI1'), ('AMC1', 'HGGSP2', 'SES2'), ('AMC1', 'HLPHI1', 'MATHS3'),
('AMC1', 'HLPHI1', 'SES1'), ('AMC1', 'MATHS2', 'PH-CH2'), ('AMC1', 'MATHS2', 'SES2'),
('AMC1', 'MATHS3', 'SES2'), ('AMC1', 'NSI2', 'SES1'), ('AMC1', 'SES1', 'SVT2'),
('AMC2', 'HGGSP1', 'HLPHI1'), ('AMC2', 'HGGSP1', 'SES2'), ('AMC2', 'HLPHI1', 'SVT1'),
('AMC2', 'MATHS1', 'NSI2'), ('AMC2', 'MATHS1', 'PH-CH2'), ('AMC2', 'MATHS1', 'PH-CH3'),
('AMC2', 'MATHS1', 'SES2'), ('AMC2', 'MATHS1', 'SVT2'), ('HGGSP1', 'HLPHI1', 'SES1'),
('HGGSP1', 'NSI2', 'SES1'), ('HGGSP1', 'SES1', 'SVT2'), ('HGGSP2', 'HLPHI1', 'MATHS1'),
('HGGSP2', 'MATHS1', 'NSI2'), ('HGGSP2', 'MATHS1', 'PH-CH2'),
('HGGSP2', 'MATHS1', 'PH-CH3'), ('HGGSP2', 'MATHS1', 'SES2'),
('HGGSP2', 'MATHS1', 'SVT2'), ('HGGSP2', 'PH-CH3', 'SVT1'),
('HLPHI1', 'MATHS3', 'PH-CH1'), ('HLPHI1', 'NSI1', 'SES1'), ('MATHS1', 'NSI2', 'SES1'),
('MATHS1', 'PH-CH2', 'SES1'), ('MATHS1', 'PH-CH3', 'SES1'), ('MATHS1', 'SES1', 'SVT2'),
('MATHS2', 'NSI2', 'PH-CH1'), ('MATHS2', 'PH-CH1', 'SVT2'),
('MATHS2', 'PH-CH2', 'SVT1'), ('MATHS3', 'NSI1', 'PH-CH2'),
('MATHS3', 'NSI1', 'PH-CH3'), ('MATHS3', 'NSI2', 'SVT1'), ('MATHS3', 'PH-CH1', 'SVT2'),
('MATHS3', 'PH-CH2', 'SVT1'), ('MATHS3', 'PH-CH3', 'SVT1')]
```

Il faut vérifier que cette répartition peut être organisée en 3 barrettes.

Exercice 18 (sur papier) : Faire l'exercice sur la feuille.