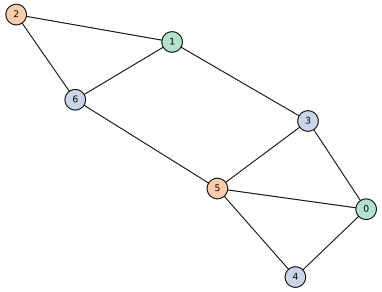
Contexte

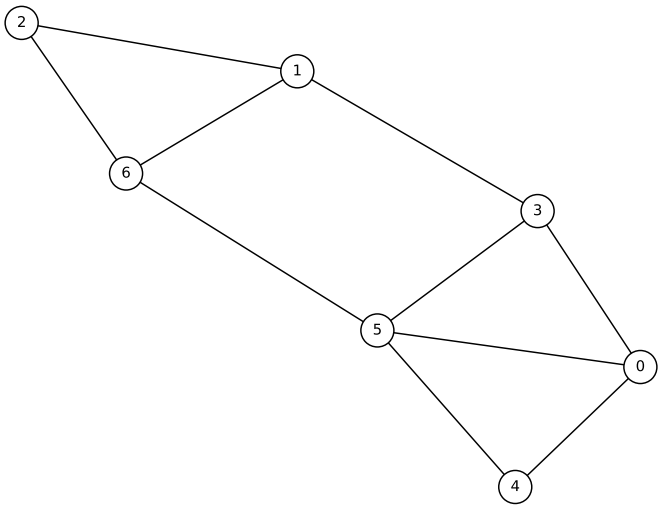


Ce problème étudie le nombre chromatique d'un graphe, c'est à dire le nombre minimum de couleurs nécessaires pour colorier les sommets d'un graphe de telle sorte que deux sommets voisins soient de couleurs différentes. On s'intéresse à des algorithmes permettant d'obtenir des majorations du nombre chromatique, en particulier les algorithmes de coloriage glouton et de DSATUR.

1. Nombres chromatiques

Dans tout le problème on considère des graphes non pondérés et non orientés dont l'ensemble des sommets est indexé par $[\![0, n-1]\!]$ et représentés par une liste d'adjacence.
Plus précisément, un graphe (ensemble des sommets S et d'ensemble des arrêtes A) $G = (S, A)$ où $S = [\![0, n-1]\!]$ sera représenté par une liste de listes `G` telle que pour tout $u \in S$, `G[u]` est la liste des sommets voisins de u.

Par exemple le graphe suivant:



est représenté par la liste d'adjacence suivante:

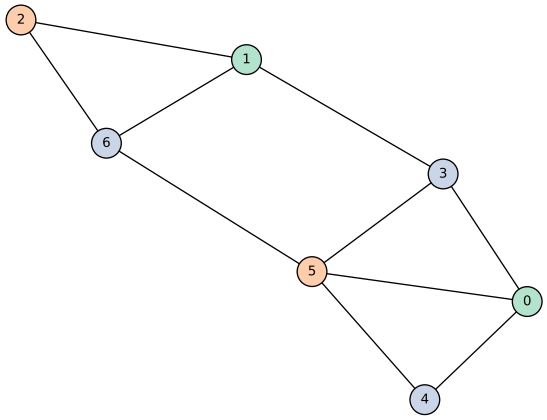
```
G0 = [[3, 5, 4], [3, 6, 2], [1, 6], [0, 1, 5], [0, 5], [0, 3, 4, 6], [1, 2, 5]]
```

Pour tout sommet $u \in S$ on appelle **degré** de u, noté $\deg(u)$ le nombre de sommets voisins de u, c'est à dire le nombre de sommets $v \in S$ tels que $(u, v) \in A$.
On appelle **degré maximal** de G, noté $\Delta(G)$, le degré maximal des sommets de G c'est à dire $\Delta(G) = \max_{u \in S} \deg(u)$.
Pour $N \in \mathbb{N}$ on appelle **coloriage** de G à N couleurs un coloriage de ses sommets en N couleurs c'est à dire une application $c : S \to [\![0, N-1]\!]$.
Un tel coloriage est dit **correct** si deux sommets voisins sont toujours coloriés de couleurs différentes c'est à dire si pour tout $(u, v) \in A$, $c(u) \neq c(v)$.
Un coloriage sera représenté par une liste `c` telle que pour tout $u \in S$, `c[u]` est la couleur de u.
Par exemple, pour le graphe précédent:

Le coloriage suivant:

```
c0_correct = [0, 0, 1, 2, 2, 1, 2]
```

correspond au schéma

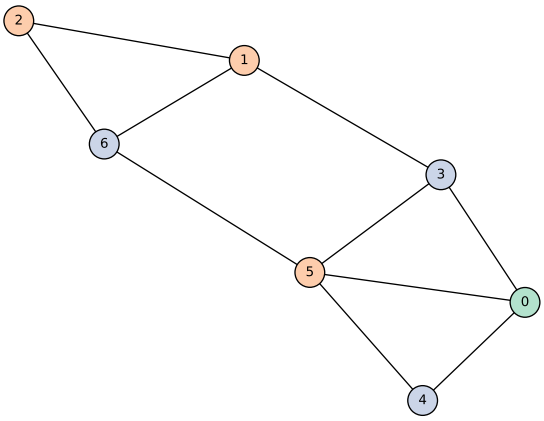


et est un coloriage correct, car deux sommets voisins ont toujours des couleurs distinctes.

Le coloriage suivant:

```
c0_incorrect = [0, 1, 1, 2, 2, 1, 2]
```

correspond au schéma



et est un coloriage incorrect, par exemple car les sommets 1 et 2 sont voisins et de même couleur.

On appelle **nombre chromatique** de G, noté $\chi(G)$ le plus petit entier N tel qu'il existe un coloriage correct de G à N couleurs.

1.1. Montrer que pour tout graphe $G = (S, A)$, $\chi(G)$ est bien défini et $\chi(G) \leq |S|$.

> Il suffit de colorier chaque sommet avec une couleur différente pour obtenir un coloriage correct à $|S|$ couleurs.

L'objectif de ce problème est de déterminer des algorithmes permettant d'obtenir des coloriages corrects de graphes et ainsi de majorer le nombre chromatique.

1.I. Quelques cas particuliers

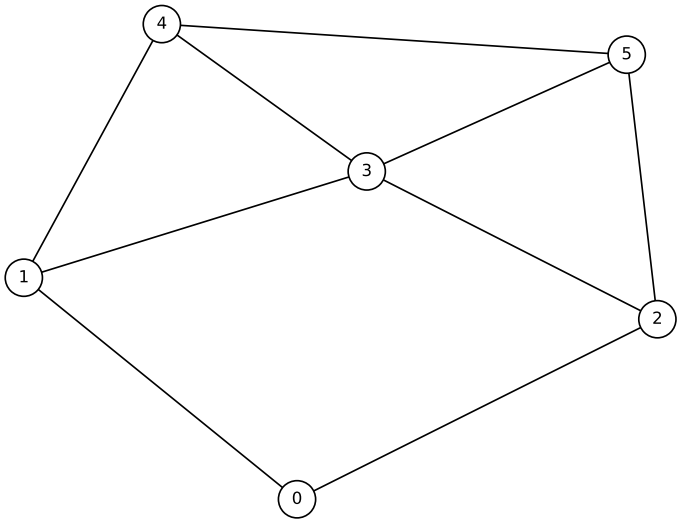
1.2.

On considère le graphe G_{ex} représenté par la liste d'adjacence suivante:

```
G_ex = [[1, 2], [0, 3, 4], [0, 3, 5], [1, 2, 4, 5], [1, 3, 5], [2, 3, 4]]
```

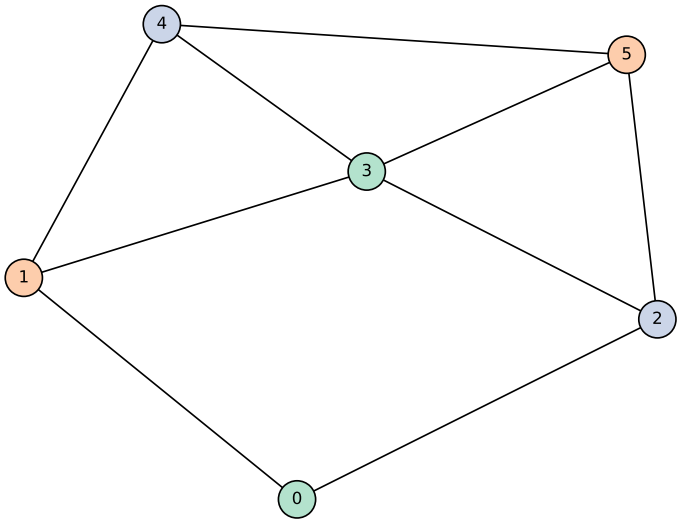
1.2.1. Représenter graphiquement G_{ex}.

On obtient le graphe suivant



1.2.2. Montrer que $\chi(G_{ex}) \leq 3$.

Pour montrer que $\chi(G_{ex}) \leq 3$ il suffit d'exhiber un coloriage correct à 3 couleurs.
Par exemple le coloriage suivant



est correct.
Donc $\boxed{\chi(G) \leq 3}$.

1.2.3. Montrer que $\chi(G_{ex}) = 3$.

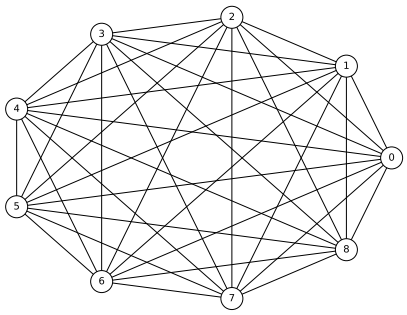
Montrons que G n'admet aucun coloriage correct à 2 couleurs.
Considérons un coloriage à deux (ou moins) couleurs.
Les sommets $1, 3, 4$ forment un triangle, par principe des tiroirs au moins deux sont de la même couleur. Donc le coloriage n'est pas correct.
Par conséquent $\chi(G) \geq 3$ et donc $\boxed{\chi(G) = 3}$.

1.3.

Pour $n \in \mathbb{N}^*$, on note K_n le graphe complet à n sommets c'est à dire le graphe dont les sommets sont les entiers de $[\![0, n-1]\!]$ et dont les arêtes sont les paires $\{(u, v) \in [\![0, n-1]\!]^2 \mid u \neq v\}$.

1.3.1. Écrire une fonction python `generer_K(n)` qui prend en argument un entier n et renvoie la liste d'adjacence du graphe K_n.

1.3.2. Déterminer pour tout $n \in \mathbb{N}^*$ la valeur de $\chi(K_n)$.



Représentation graphique de K_9.

1.3.a) On peut par exemple parcourir les sommets et pour chaque sommet et pour chaque sommet créer la liste de ses voisins (tous les autres sommets).

```python
def generer_K(n):
    """
    Génère un graphe complet K_n sous forme de liste d'adjacence.
    Prend en en argument n (int) Le nombre de sommets du graphe.
    Renvoie une liste d'adjacence représentant le graphe complet K_n.
    """
    G = []
    for i in range(n):
        l = []  # La liste des voisins du sommet i
        for j in range(n):
            if i != j:
                l.append(j)
        G.append(l)
    return G
```

On remarque qu'une version par compréhension (non exigible) est possible:

```python
def generer_K_comprehension(n):
    """
    Génère un graphe complet K_n sous forme de liste d'adjacence.
    Prend en en argument n (int) Le nombre de sommets du graphe.
    Renvoie une liste d'adjacence représentant le graphe complet K_n.
    """
    return [[j for j in range(n) if j != i] for i in range(n)]
```

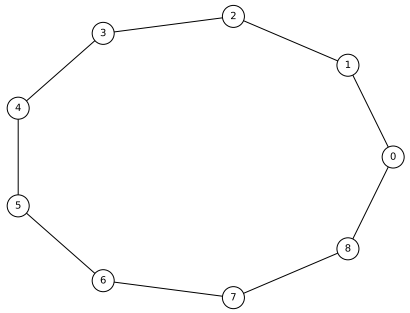
1.3.b) Soit $n \in \mathbb{N}^*$.
Comme tout sommet est voisin de tous les autres sommets il faut au moins n couleurs pour colorier correctement K_n. De plus le coloriage où chaque sommet est colorié avec une couleur différente est correct. Ainsi $\boxed{\chi(K_n) = n}$.

1.4.

Pour $n \in \mathbb{N}^*$, on note C_n le cycle à n sommets c'est à dire le graphe dont les sommets sont les entiers de $[\![0, n-1]\!]$ et dont les arêtes sont les paires $\{(u, v) \in [\![0, n-1]\!]^2 \mid |u-v| \% n = 1\}$.

1.4.1. Écrire une fonction python `generer_C(n)` qui prend en argument un entier n et renvoie la liste d'adjacence du graphe C_n.

1.4.2. Déterminer pour $n \in \mathbb{N}^*$ la valeur de $\chi(C_n)$.



Représentation graphique de C_9.

1.4.a) On peut par exemple parcourir les sommets et pour chaque sommet et pour chaque sommet créer la liste de ses voisins (le sommet précédent et le sommet suivant).

```python
def generer_C(n):
    """
    Génère un graphe cycle C_n sous forme de liste d'adjacence.
    Prend en en argument n (int) Le nombre de sommets du graphe.
    Renvoie une liste d'adjacence représentant le graphe cycle C_n.
    """
    G = []
    for i in range(n):
        l = []  # La liste des voisins du sommet i
        l.append((i + 1) % n)
        l.append((i - 1) % n)
        G.append(l)
    return G
```

On remarque qu'une version par compréhension (non exigible) est possible:

```python
def generer_C_comprehension(n):
    """
    Génère un graphe cycle C_n sous forme de liste d'adjacence.
    Prend en en argument n (int) Le nombre de sommets du graphe.
    Renvoie une liste d'adjacence représentant le graphe cycle C_n.
    """
    return [[(i + 1) % n, (i - 1) % n] for i in range(n)]
```

1.4.b) Soit $n \in \mathbb{N}^*$.
On procède par disjonction de cas.

- **Si $n = 1$**
  Alors $\chi(C_1) = 1$.
- **Si n est pair**
  Alors comme l'ensemble des arrêtes est non vide, il faut au moins deux couleurs pour colorier C_n d'où $\chi(C_n) \geq 2$. De plus le coloriage où les sommets d'indices pairs sont coloriés avec une couleur et les sommets d'indices impairs avec une autre couleur est correct. Donc $\chi(C_n) \leq 2$.
  Ainsi $\boxed{\chi(C_n) = 2}$.
- **Si $n \geq 3$ est impair**
  On observe que le coloriage où les sommets pairs entre 0 et $n-2$ sont coloriés avec une couleur et les sommets impairs avec une autre couleur et le sommet $n-1$ en une troisième couleur est correct. Donc $\chi(C_n) \leq 3$.
  Supposons par l'absurde que $\chi(G) \leq 2$. Considérons alors un coloriage correct à deux couleurs. Quitte à renuméroter, le sommet 0 est colorié de la couleur 0. Comme $(0, 1) \in A$, on en déduit que le sommet 1 est colorié de la couleur 1. De même comme $(1, 2) \in A$, le sommet 2 est colorié par la couleur 0. Par récurrence on en déduit que pour tout

$k \in [\![0, n-1]\!]$, le sommet k est colorié par la couleur $k\%2$. Comme n est impaire, les sommets 0 et $n-1$ sont donc coloriés de la couleurs 0 et voisins ce qui est absurde.
Ainsi $\boxed{\chi(C_n) = 3}$.

1.5. Écrire une fonction python `est_correct(G,c)` qui prend en arguments une liste d'adjacence d'un graphe G et un coloriage c de G et renvoie `True` si c est correct et `False` sinon.

On parcourt tous les sommets et pour chaque sommet on vérifie que tous les voisins sont de couleurs différentes.
(C'est à dire on vérifie $\neg(\exists u \in S, \exists v \in \text{voisins}(u), c(u) = c(v))$).

```
def est_correct(G, c):
    """
    Vérifie si un coloriage est correct.
    Prend en argument G (liste de liste) une liste d'adjacence du graphe et c (liste) le
    ↪ coloriage.
    Renvoie True si le coloriage est correct, False sinon.
    """
    n = len(G)
    for u in range(n):
        for v in G[u]:
            if c[u] == c[v]:
                return False
    return True
```

1.6. Écrire une fonction python `generer_tous_coloriages(n,N)` prenant en argument un entier n et un entier N et renvoyant ne liste de tous les éléments de $[\![0, N-1]\!]^n$ (c'est à dire une liste de tous les coloriages de $[\![0, n-1]\!]$ à N couleurs).
Par exemple `generer_tous_coloriages(2,3)` renverra
`[[0,0],[0,1],[0,2],[1,0],[1,1],[1,2],[2,0],[2,1],[2,2]]`
(non nécessairement dans cet ordre).

On peut par exemple procéder de manière récursive: une liste de $[\![0, N-1]\!]^n$ est une liste de $[\![0, N-1]\!]^{n-1}$ à laquelle on ajoute un élément de $[\![0, N-1]\!]$ à la fin.
On obtient le code récursif suivant:

```
def generer_tous_coloriages(n, N):
    """
    Prend en argument un n (int) et N (int) et renvoie tous les éléments de l'ensemble
    ↪ {0,1,...,N-1}^n.
    """
    if n == 0:
        return [[]]
    l = []
    lprec = generer_tous_coloriages(n - 1, N)
    for s in lprec:
        for i in range(N):
            l.append([i] + s)
    return l
```

1.7. En déduire une fonction python `nombre_chromatique_naif(G)` qui prend en argument une liste d'adjacence d'un graphe G et renvoie le nombre chromatique de G.
Pour cela on pourra tester exhaustivement les coloriages de G jusqu'à en trouver un correct.

On sait que $\chi(G) \leq n$.
On peut alors très naïvement tester tous les coloriages de G de moins de n couleurs jusqu'à en trouver un correct.

```
def nombre_chromatique_naif(G):
    """
    Prend en argument G (liste de liste) une liste d'adjacence du graphe.
    Renvoie le nombre chromatique du graphe.
    """
    n = len(G)
    for k in range(1, n + 1):
        lck = generer_tous_coloriages(n, k)  # Liste de tous les coloriages à k couleurs
        for c in lck:
            if est_correct(G, c):
```

```
                return k
```

1.8. Quelle est la complexité dans le pire des cas de la fonction `nombre_chromatique_naif(G)` en fonction de $n = |S|$?
Commenter.

La fonction `est_correct(G,c)` a une complexité en $O(|A|)$ car chaque arête est testée deux fois. Comme $|A| \leq \binom{|S|}{2}$ on en déduit que la complexité de `est_correct(G,c)` est en $O(n^2)$.
La fonction `generer_tous_coloriages(n,N)` a une complexité en $O(N^n)$ (produit).
Comme

$$\sum_{N=1}^{\chi}(G)n^2N^n = O(n^2\chi(G)^{n+1})$$

et que $\chi(G) \leq n$ (l'égalité pouvant être atteinte, par exemple par les graphes complets) on en déduit une complexité en $\boxed{O(n^{n+3})}$.
Cette complexité est beaucoup trop grande pour être utilisée en pratique, d'où la recherche d'algorithmes plus efficaces.

1.II. Degré maximal et coloriage glouton

On se propose dans cette partie de montrer que pour tout graphe G, $\chi(G) \leq \Delta(G) + 1$ en exhibant un algorithme permettant de colorier correctement G en $\Delta(G) + 1$ couleurs.

1.9. Écrire une fonction python `liste_degrés(G)` qui prend en argument une liste d'adjacence d'un graphe G et renvoie la liste des degrés des sommets de G.

Le degré d'un sommet u étant la longueur de la liste des voisins on obtient par exemple

```
def liste_degrés(G):
    """
    Prend en argument G (liste de liste) une liste d'adjacence du graphe.
    Renvoie la liste des degrés des sommets du graphe.
    """
    n = len(G)
    ld = []
    for u in range(n):
        ld.append(len(G[u]))
    return ld
```

Une version par compréhension (et par fonctions anonymes pour le même prix) est possible (non exigible):

```
liste_degrés = lambda G: [len(lv) for lv in G]
```

1.10. En déduire une fonction python `degré_maximal(G)` qui prend en argument une liste d'adjacence d'un graphe G et renvoie le degré maximal de G.

On obtient (algorithme de calcul de max)

```
def plus_petit_disponible(c):
    """
    Prend en argument c (liste) une liste d'entiers.
    Renvoie le plus petit entier qui n'est pas dans c.
    """
    n = len(c)
    # Par principe des tiroirs au moins un des entiers de 0 à n n'est pas dans c
    # est_present est une liste de booléens qui indique si l'entier i est dans c
    est_present = [False for _ in range(n + 1)]
    for x in c:
        if x < n + 1:
            est_present[x] = True
    # une fois la liste crée il suffit de parcourir les entiers de 0 à n
    for i in range(n + 1):
        if not est_present[i]:
            return i
```

On considère l'algorithme suivant pour obtenir un coloriage correct d'un graphe G.
▶ On commence avec aucun sommet colorié.

- Pour tout sommet u de G.
  - On détermine la plus petite couleur c telle que aucun voisin de u n'est colorié de cette couleur c'est à dire $\min(\{c(v) \mid v \in \text{voisins}(u)\})$.
  - On colorie u de cette couleur.
- On renvoie le coloriage obtenu.

1.11. Écrire une fonction python `plus_petit_disponible(l)` qui prend en argument une liste d'entiers ℓ et qui renvoie le plus petit entier positif qui n'est pas dans ℓ.
Par exemple `plus_petit_disponible([0,1,3,4,8])` renverra `2`.

Pour ne pas utiliser le test d'appartenance à une liste `in` on peut créer une liste `est_present` pour déterminer en un seul parcours les entiers présents.
On remarque que pour une liste de longueur n, par principe des tiroirs au moins un élément de $[\![0, n]\!]$ n'est pas dans la liste.
On obtient le code suivant:

```
def plus_petit_disponible(c):
    """
    Prend en argument c (liste) une liste d'entiers.
    Renvoie le plus petit entier qui n'est pas dans c.
    """
    n = len(c)
    # Par principe des tiroirs au moins un des entiers de 0 à n n'est pas dans c
    # est_present est une liste de booléens qui indique si l'entier i est dans c
    est_present = [False for _ in range(n + 1)]
    for x in c:
        if x < n:
            est_present[x] = True
    for i in range(n + 1):
        if not est_present[i]:
            return i
```

1.12. Compléter sur votre copie la fonction python `coloriage_glouton(G)` qui prend en argument une liste d'adjacence d'un graphe G et renvoie un coloriage correct de G obtenu par l'algorithme précédent.

```
def coloriage_glouton(G):
    """
    Colorie un graphe avec l'algorithme glouton.
    Prend en argument G (liste de liste) une liste d'adjacence du graphe.
    Renvoie c (liste) La liste des couleurs attribuées à chaque sommet.
    """
    n = len(G)  # Nombre de sommets du graphe
    # Crée une liste de couleurs, None correspondant à un sommet non colorié
    c = [None for _ in range(n)]
    for u in range(n):  # Parcourt chaque sommet du graphe
        # Détermine la première couleur disponible pour le sommet u, notée x
        ##############
        #
        # À compléter
        #
        ##############
        # Attribution de la couleur au sommet u
        c[u] = x
    return c
```

On obtient le code suivant:

```
def coloriage_glouton(G):
    """
    Colorie un graphe avec l'algorithme glouton.
    Prend en argument G (liste de liste)une liste d'adjacence du graphe.
    Renvoie c (liste) La liste des couleurs attribuées à chaque nœud.
    """
    n = len(G)
    # Crée une liste de couleurs
    c = [None for _ in range(n)]
    # Parcourt chaque nœud du graphe
```

```
    for u in range(n):
        # Détermine la première couleur disponible pour le nœud u, notée x
        lc = []  # Liste des couleurs des voisins coloriés de u
        for v in G[u]:
            if c[v] != None:
                lc.append(c[v])
        x = plus_petit_disponible(lc)
        print(u, lc, x, c)
        c[u] = x

    return c
```

1.13. Démontrer la terminaison et la correction de l'algorithme précédent.

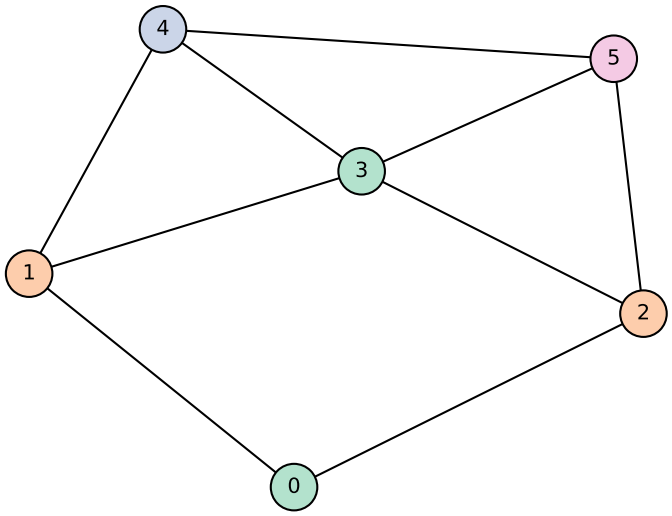
- Un **variant** de boucle est le nombre de sommets non coloriés. D'où la terminaison.
- Un **invariant** de boucle est pour toute arête $(u, v) \in A$, si u et v sont coloriés alors leurs couleurs sont distinctes. D'où la correction.

1.14. En déduire que $\chi(G) \leq \Delta(G) + 1$.

Soit $G = (S, A)$ un graphe et c le coloriage obtenu par l'algorithme `coloriage_glouton`.
Soit $u \in S$. On note $c(u)$ la couleur de u dans c.
On a $c(u) = \min(\{c(v) \mid v \in \text{voisins}(u) \text{ et } v \leq u\})$ (le minimum des couleurs des voisins de u déjà coloriés).
En particulier (principe des tiroirs) $c(u) \leq |\{v \in S \mid v \in \text{voisins}(u) \text{ et } v \leq u\}|$. Donc $c(u) \leq \deg(u) \leq \Delta(G)$.
Donc $\chi(G) \leq \Delta(G) + 1$.

1.15. Appliquer l'algorithme `coloriage_glouton` au graphe G_{ex} de la question **1.2.**.
Commenter.

On obtient (cf figure 1 pour les étapes intermédiaires):



Et on observe que le coloriage obtenu n'est pas nécessairement optimal.
En effet $4 > 3$.

1.16. **Bonus** Déterminer l'ensemble des graphes tels que $\chi(G) = \Delta(G) + 1$.
(Attention cette question est beaucoup plus difficile que le reste du sujet).

> On observe que pour tout $n \in \mathbb{N}^*$, on a K_n vérifie $\chi(K_n) = n = (n-1)+1 = \Delta(K_n)+1$.
> De plus pour tout $n \in \mathbb{N}^*$ impair, on a C_n vérifie $\chi(C_n) = 3 = 2+1 = \Delta(C_n)+1$.
> Il s'agit des seuls graphes vérifiant cette propriété (théorème de Brooks).
> [Preuve à rédiger, voir par exemple cette courte preuve].

1.III. Le cas des graphes bipartis

Un graphe G est dit biparti ssi il admet un coloriage correct en deux couleurs c'est à dire ssi $\chi(G) \leq 2$.
On rappelle que pour tout $k \geq 2$ on appelle **cycle** de longueur k de G une suite de sommets $(u_0, u_1, \dots, u_{k-1}) \in S^k$ telle que pour tout $i \in [\![0, k-2]\!]$, $(u_i, u_{i+1}) \in A$ et $(u_{k-1}, u_0) \in A$.
L'objectif de cette partie est de montrer qu'un graphe est biparti ssi il n'admet pas de cycle de longueur impaire.

1.17. Soit G un graphe biparti. Montrer que tout cycle de G est de longueur paire.

> Soit G un graphe biparti, $k \geq 2$, et $(u_0, u_1, \dots, u_{k-1})$ un cycle de G de longueur k.
> Comme G est biparti, on peut colorier les sommets de ce cycle avec deux couleurs. Quitte à renuméroter, on peut supposer que u_0 est colorié avec la couleur 0.
> Comme $(u_0, u_1) \in A$, on en déduit que u_1 est colorié avec la couleur 1. De même comme $(u_1, u_2) \in A$, on en déduit que u_2 est colorié avec la couleur 0.
> Par récurrence on en déduit que pour tout $i \in [\![0, k-1]\!]$, u_i est colorié avec la couleur $i\%2$.
> En particulier u_{k-1} est colorié avec la couleur $(k-1)\%2$ et est voisin de u_0 qui est colorié avec la couleur 0. Donc $(k-1)\%2 = 1$ c'est à dire k est pair.
> Ainsi tout cycle de G est de longueur paire.

On considère l'algorithme suivant:
- On crée une file de sommets à traiter.
- On choisit un sommet non colorié de G. On le colorie et on l'ajoute à la file.
- Tant que la file est non vide:
  - On retire le premier sommet de la file.
  - Pour tous les voisins non déjà coloriés de ce sommet, on les colorie avec la couleur opposée à celle du sommet et on les ajoute à la file.
- On renvoie le coloriage obtenu.

1.18. L'algorithme précédent implémente il un parcours en profondeur ou un parcours en largeur ?
Commenter sa terminaison et sa complexité.

> La structure pour stocker les sommets à traiter est une structure de **file**, il s'agit donc d'un parcours en largeur.
> Comme parcours en largeur il se termine (un variant étant par exemple le nombre de sommets colorié) et est de complexité est $O(|S| + |A|)$.

1.19. Compléter sur votre copie la fonction python `coloriage_biparti(G)` qui prend en argument une liste d'adjacence d'un graphe biparti G et renvoie un coloriage de G obtenu par l'algorithme précédent en partant de 0 comme sommet initial.

```
1  from collections import deque
2
3
4  def coloriage_biparti(G):
5      """
6      Colorie un graphe en deux couleurs par un parcours en ??? en supposant que le graphe
       ↪  n'admet pas de cycle impair et est connexe.
7      Prend en argument G (liste de liste) une liste d'adjacence du graphe.
8      Renvoie c (liste) La liste des couleurs attribuées à chaque sommet.
9      """
10     n = len(G)  # Nombre de sommets du graphe
11     # Crée une file de sommets à traiter, initialement vide
12     file = deque()
13     # Crée une liste de couleurs, None correspondant à un sommet non colorié
14     c = [None for _ in range(n)]
15     # On commence par colorier et par rajouter à la file le sommet 0
16     c[0] = 0
```

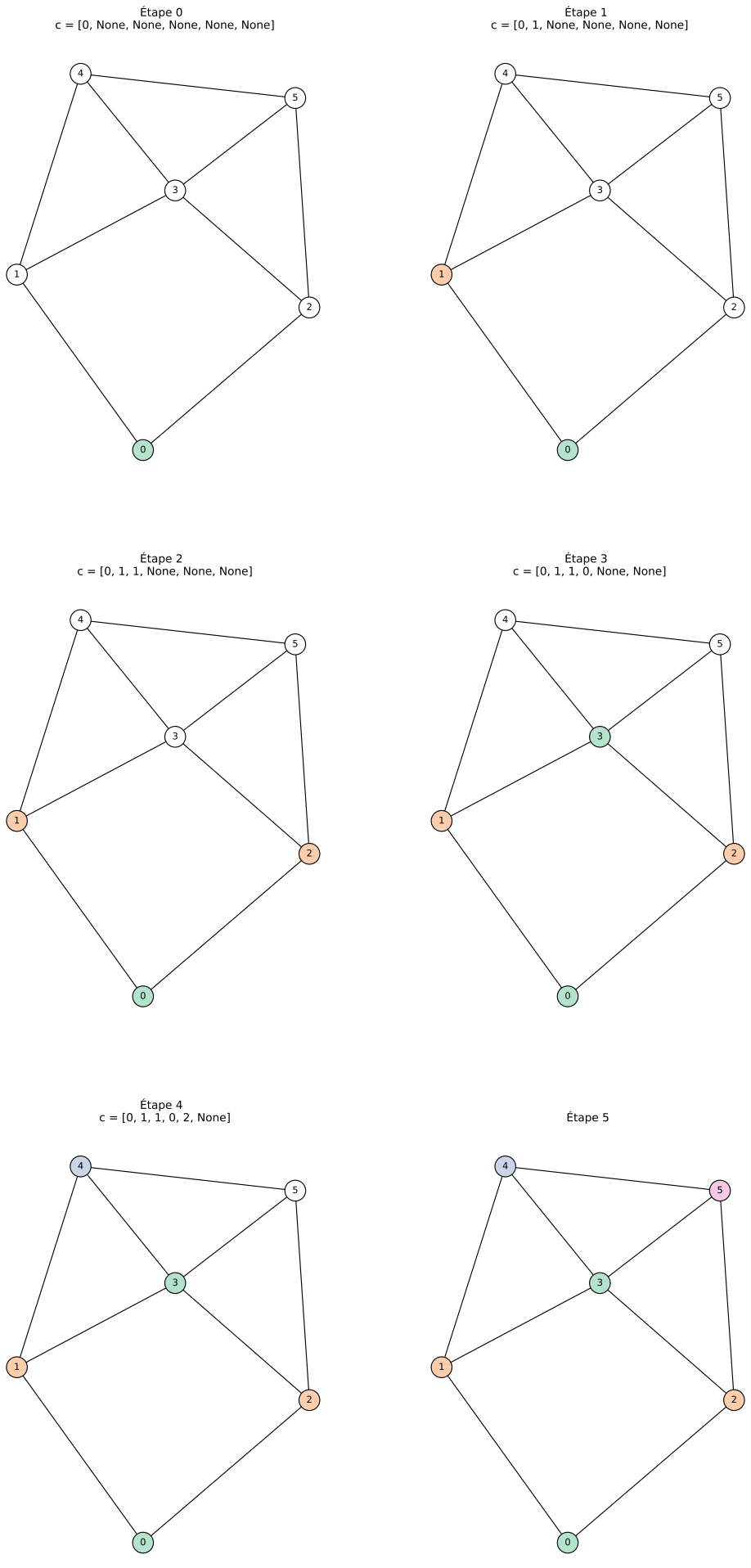


Figure 1: Détails des étapes pour le coloriage glouton

```
    file.append(0)  # On rajoute un élément à droite
    while file:  # Tant que la file est non vide
        u = file.popleft()  # On retire l'élément à gauche
        # Pour tous les voisins de u non coloriés on attribue la couleur différente de celle de u et on les ajoute à la file
        ##############
        #
        # À compléter, on rappelle que l'on peut rajouter un sommet v à la file avec l'instruction «file.append(v)».
        #
        ##############
    return c
```

On rappelle que pour la structure `deque` du module `collections` l'ajout d'éléments à droite s'effectue par la méthode `append` et le retrait d'éléments à gauche par la méthode `popleft`.

On obtient le code suivant:

```
from collections import deque


def coloriage_biparti(G):
    """
    Colorie un graphe en deux couleurs par un parcours en ??? en supposant que le graphe n'admet pas de cycle impair et est connexe.
    Prend en argument G (liste de liste) une liste d'adjacence du graphe.
    Renvoie c (liste) La liste des couleurs attribuées à chaque sommet.
    """
    n = len(G)  # Nombre de sommets du graphe
    # Crée une file de sommets à traiter, initialement vide
    file = deque()
    # Crée une liste de couleurs, None correspondant à un sommet non colorié
    c = [None for _ in range(n)]
    # On commence par colorier et par rajouter à la file le sommet 0
    c[0] = 0
    file.append(0)  # On rajoute un élément à droite
    while file:  # Tant que la file est non vide
        u = file.popleft()  # On retire l'élément à gauche
        # Pour tous les voisins de u non coloriés on attribue la couleur différente de celle de u et on les ajoute à la file
        for v in G[u]:
            # Ici pas besoin de liste des sommets déjà visités, car on utilise c
            if c[v] == None:
                c[v] = 1 - c[u]  # Envoie 0 sur 1 et 1 sur 0
                file.append(v)
    return c
```

1.20. Démontrer la correction de l'algorithme précédent dans le cas où G est connexe et n'admet pas de cycle impair.

Si G est connexe alors le parcours en largeur permet de colorier tous les sommets de G.
Notons c le coloriage obtenu par l'algorithme.
On constante (invariant de boucle) que les sommets à distance paire de 0 sont coloriés avec la couleur 0 et les sommets à distance impaire de 0 sont coloriés avec la couleur 1.
Supposons par l'absurde que le coloriage n'est pas correct.
Soient u et v deux sommets voisin de G tels que $c(u) = c(v)$.
Si la couleur commune est 0, alors u et v sont à distance paire de 0.
En particulier il existe un chemin de longueur paire entre 0 et u et un chemin de longueur paire entre v et 0.
En concaténant ces deux chemins avec (u, v) entre les deux on obtient un cycle de longueur impaire. Absurde.
D'où le coloriage obtenu est correct.
[Faire un schéma]

1.21. Proposer une fonction python `coloriage_biparti_bis(G)` qui prend en argument une liste d'adjacence d'un graphe G et renvoie un coloriage correct si G est biparti et `False` si sinon.
On prendra soin à ce que la fonction soit correcte pour des graphes non nécessairement connexes.

On modifie l'algorithme précédent:
- À chaque étape en plus de colorier les voisins non coloriés on vérifie qu'il n'existe pas de cycle de longueur impaire, c'est à dire que deux voisins ne sont pas de la même couleur.
- Une fois un parcours terminé, on reprend un sommet non colorié et on recommence.

On obtient le code suivant:

```
def coloriage_biparti_bis(G):
    """
    Colorie un graphe en deux couleurs par un parcours en largeur.
    Prend en argument G (liste de liste) une liste d'adjacence du graphe.
    Renvoie c (liste) La liste des couleurs attribuées à chaque sommet si le graphe est biparti,
    ↪  False sinon.
    """
    n = len(G)  # Nombre de sommets du graphe
    # Crée une file de sommets à traiter, initialement vide
    file = deque()
    # Crée une liste de couleurs, None correspondant à un sommet non colorié
    c = [None for _ in range(n)]
    for u0 in range(n):  # On parcourt tous les sommets
        if c[u0] == None:  # On part d'un sommet non colorié
            ## On effectue un coloriage par parcours en largeur
            c[u0] = 0
            file.append(u0)
            while file:  # Tant que la file est non vide
                u = file.popleft()  # On retire l'élément à gauche
                # Pour tous les voisins de u non coloriés on attribue la couleur différente de
                ↪  celle de u et on les ajoute à la file
                for v in G[u]:
                    if (
                        c[v] == c[u]
                    ):  # Si un voisin est de la même couleur que u, le graphe n'est pas biparti
                        return False
                    elif c[v] == None:  # On colorie les voisions non coloriés
                        c[v] = 1 - c[u]  # Envoie 0 sur 1 et 1 sur 0
                        file.append(v)
    return c
```

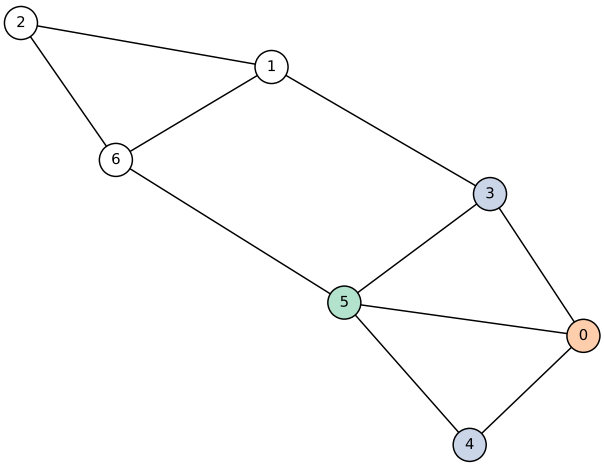
1.22. Conclure.

On a montré qu'un graphe coloriable en deux couleurs n'admet pas de cycle de longueur impaire et on a exhibé un algorithme permettant de colorier correctement un graphe n'admettant pas de cycle de longueur impaire en deux couleurs.
Par conséquent un graphe est biparti ssi il n'admet pas de cycle de longueur impaire.

1.IV. Un raffinement du coloriage glouton: DSATUR

On se propose dans cette partie de présenter un algorithme plus efficace que le coloriage glouton pour obtenir un coloriage correct d'un graphe.

Étant donné un sommet d'un graphe partiellement colorié on appelle **degré de saturation** le nombre de couleurs différentes parmi ses voisins déjà coloriés. Par exemple sur le `G0` muni du coloriage partiel `[1, None, None, 2, 2, 0, None]` (cf à droite), le degré de saturation du sommet 5 est 2 car il est voisin il y a deux couleurs différentes parmi ses voisins déjà coloriés.
On représentera la liste des degrés de saturation par une liste `sat` telle que pour tout $u \in S$, `sat[u]` est le degré de saturation de u.
Par exemple, toujours sur le même graphe partiellement coloré à droite on a: `sat = [2, 1, 0, 2, 2, 2, 1]`.



On considère alors l'algorithme suivant:
▶ On commence avec aucun sommet colorié.

- Tant qu'il reste des sommets à colorier.
  - Parmi les sommets non coloriés on choisit celui de degré de saturation maximal. Si il y a égalité on choisit celui de degré maximal. Si il y a égalité on choisit un sommet quelconque.
  - On colorie ce sommet avec la plus petite couleur disponible.
  - On met à jour la liste des degrés de saturation.
- On renvoie le coloriage obtenu.

1.23. Écrire une fonction python `calcul_saturation(G,c,v)` prenant en argument une liste d'adjacence d'un graphe G, un coloriage (partiel) c de G et un sommet v et renvoie le degré de saturation de v.
Par exemple `calcul_saturation(G0,[1, None, None, 2, 2, 0, None],5)` renverra 2.

On parcours les voisins de v et on compte le nombre de couleurs différentes.

```
def calcul_saturation(G, c, v):
    """
    Prend en argument G (liste de liste) une liste d'adjacence du graphe, c (liste) une liste de couleurs attribuées à chaque sommet et v (int) un sommet du graphe.
    Renvoie le nombre de couleurs différentes des voisins de v.
    """
    n = len(G)  # Nombre de sommets du graphe, majoration du nombre de couleurs
    # nc[i] vaut 1 si la couleur i est présente dans les voisins de v
    nc = [0 for _ in range(n)]
    for u in G[v]:
        if c[u] != None:
            nc[c[u]] = 1
    # On effectue ensuite la somme
    S = 0
    for k in range(n):
        S = S + nc[k]
    return S
```

On aurait aussi pu utiliser un dictionnaire.

1.24. Écrire une fonction python `plus_grande_saturation(c,sat,deg)` prenant en argument trois listes de même longueur et renvoie l'indice i tel que $c[i]$ vaut `None` (non colorié), $\text{sat}[i]$ est maximal, et si il y a égalité $\deg[i]$ est maximal (c'est un élément maximal pour l'ordre lexicographique sur $\text{sat} \times \deg$). Si pour tout i, $c[i]$ est différent de `None` (c'est à dire tous les sommets sont coloriés) la fonction renverra `None`.
Par exemple `plus_grande_saturation([0,None,None,None,1,0],[1,0,1,1,0,2],[2,2,3,1,2,1])` renverra 2.

On parcourt tous les sommets et on garde en mémoire le sommet de plus grande saturation et de plus grand degré, en prenant garde d'éviter les sommets non coloriés et de traiter le cas où aucun sommet n'est colorié.

```
def plus_grande_saturation(c, sat, deg):
    """
    Prend en argument c (liste) une liste de couleurs attribuées à chaque sommet, sat (liste) la saturation de chaque sommet et deg (liste) le degré de chaque sommet.
    Renvoie un sommet de plus grande saturation et si égalité de plus grand degré.
    """

    n = len(c)
    i = 0
    # Si toutes les couleurs sont attribuées, on renvoie None
    while i < n and c[i] != None:
        i += 1
    if i == n:
        return None
    # Sinon i représente le premier sommet colorié
    # On calcule ensuite alors l'indice du plus grand sommet par ordre lexicographique
    for j in range(1, n):
        if c[j] == None:
            if sat[j] > sat[i] or (sat[j] == sat[i] and deg[j] > deg[i]):
                i = j
    return i
```

1.25. Compléter sur votre copie la fonction python `coloriage_DSATUR(G)` qui prend en argument une liste d'adjacence d'un graphe G et renvoie un coloriage correct de G obtenu par l'algorithme précédent.

```
def coloriage_DSATUR(G):
    """
    Colorie un graphe avec l'algorithme DSATUR.
```

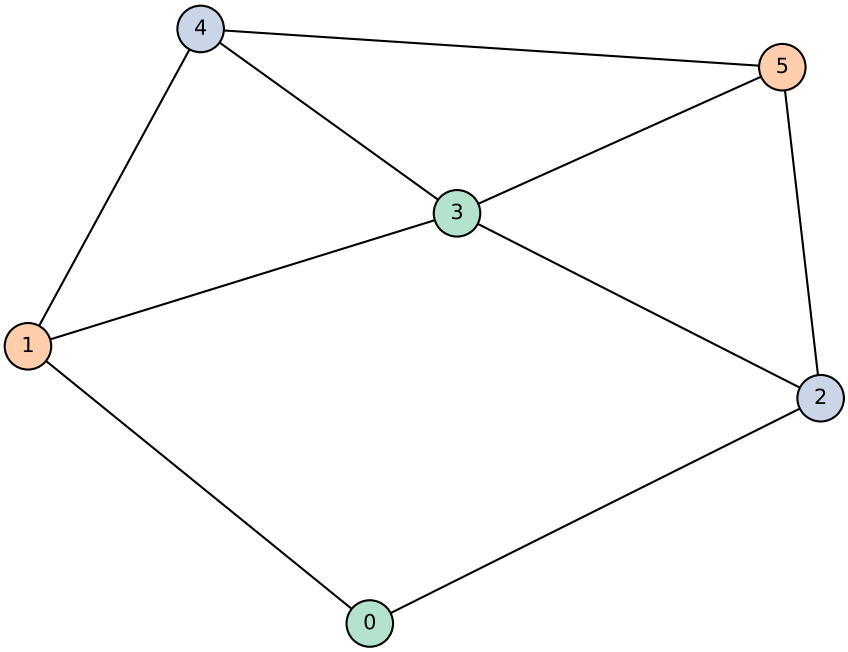
```
    Prend en argument G (liste de liste) une liste d'adjacence du graphe.
    Renvoie c (liste) La liste des couleurs attribuées à chaque nœud.
    """
    n = len(G)  # Nombre de sommets du graphe
    # Crée une liste de couleurs
    c = [None for _ in range(n)]
    # Crée une liste de saturation
    sat = [0 for _ in range(n)]
    # Crée la liste des degrés
    deg = liste_degrés(G)
    # Initialisation
    u = plus_grande_saturation(c, sat, deg)
    # Tant qu'il reste des sommets à colorier
    while u != None:
        ##############
        #
        # À compléter
        #
        ##############
        u = plus_grande_saturation(c, sat, deg)
    return c
```

On obtient

```
def coloriage_DSATUR(G):
    """
    Colorie un graphe avec l'algorithme DSATUR.
    Prend en argument G (liste de liste)une liste d'adjacence du graphe.
    Renvoie c (liste) La liste des couleurs attribuées à chaque nœud.
    """

    n = len(G)

    # Crée une liste de couleurs
    c = [None for _ in range(n)]
    # Crée une liste de saturation
    sat = [0 for _ in range(n)]
    # Crée la liste des degrés
    deg = liste_degrés(G)
    # Initialisation
    u = plus_grande_saturation(c, sat, deg)
    # Tant qu'il reste des sommets à colorier
    while u != None:
        # Détermine la première couleur disponible pour le sommet u, notée x
        x = plus_petit_disponible([c[v] for v in G[u]])
        # Attribution de la couleur au nœud u
        c[u] = x
        # Met à jour la saturation des voisins du nœud u
        for v in G[u]:
            sat[v] = calcul_saturation(G, c, v)
        # Détermination du prochain sommet
        u = plus_grande_saturation(c, sat, deg)

    return c
```

1.26. Appliquer l'algorithme `coloriage_DSATUR` au graphe G_{ex} de la question **1.2.**.
Commenter.

On obtient (cf figure **??** pour les étapes intermédiaires):



On observe que l'algorithme DSATUR est plus efficace que l'algorithme glouton, dans le sens où il utilise moins de couleurs.
On pourrait montrer qu'il est même exact pour certaines classes de graphes (par exemple les cycles ou les graphes bipartis).
Néanmoins il n'est pas optimal en général.