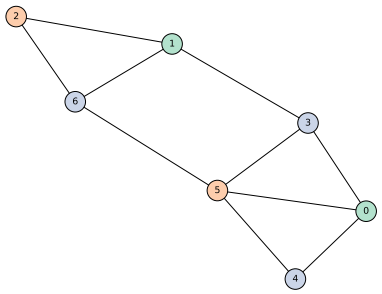
Nombre chromatique d'un graphe

Contexte

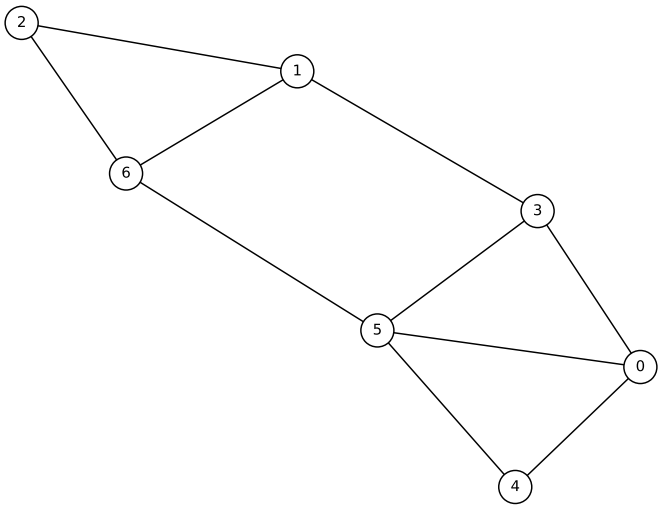


Ce problème étudie le nombre chromatique d'un graphe, c'est à dire le nombre minimum de couleurs nécessaires pour colorier les sommets d'un graphe de telle sorte que deux sommets voisins soient de couleurs différentes. On s'intéresse à des algorithmes permettant d'obtenir des majorations du nombre chromatique,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

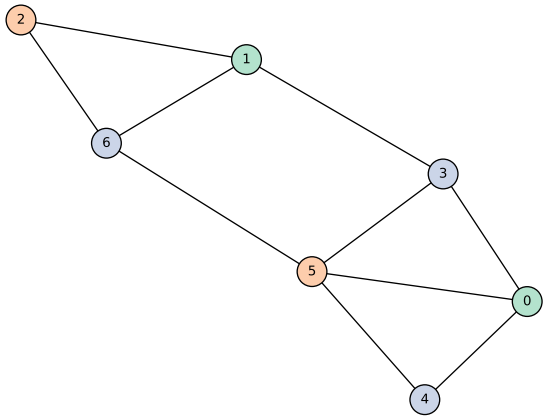
```
G0 = [[3, 5, 4], [3, 6, 2], [1, 6], [0, 1, 5], [0, 5], [0, 3, 4, 6], [1, 2, 5]]
```

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

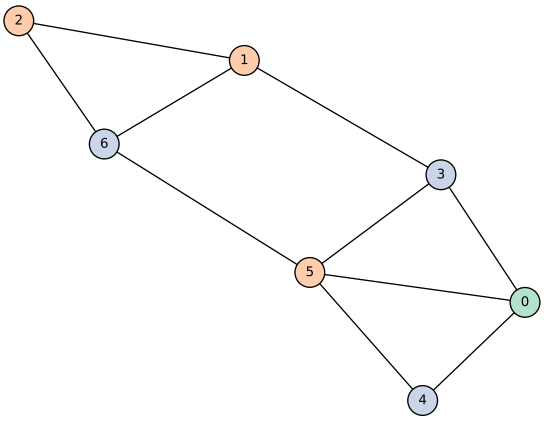


et est un coloriage correct, car deux sommets voisins ont toujours des couleurs distinctes.

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 $\chi(G) \leq |S|$.

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}.

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

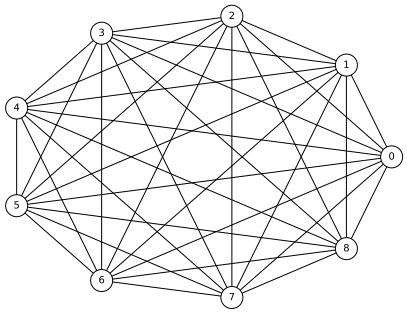
1.2.3. Montrer que $\chi(G_{ex}) = 3$.

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 $\chi(K_n)$.



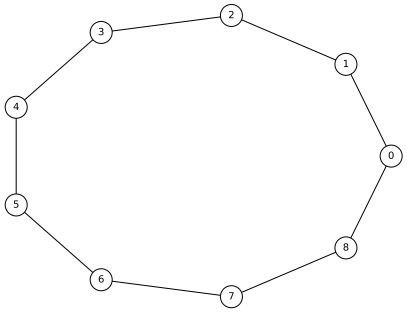
Représentation graphique de K_9.

1.4.

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.

1.5. 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.

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).

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.

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

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.

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 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`.

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.

```
1  def coloriage_glouton(G):
2      """
3      Colorie un graphe avec l'algorithme glouton.
4      Prend en argument G (liste de liste) une liste d'adjacence du graphe.
5      Renvoie c (liste) La liste des couleurs attribuées à chaque sommet.
6      """
7      n = len(G)  # Nombre de sommets du graphe
8      # Crée une liste de couleurs, None correspondant à un sommet non colorié
9      c = [None for _ in range(n)]
10     for u in range(n):  # Parcourt chaque sommet du graphe
11         # Détermine la première couleur disponible pour le sommet u, notée x
12         ##############
13         #
14         # À compléter
15         #
16         ##############
17         # Attribution de la couleur au sommet u
18         c[u] = x
19     return c
```

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

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

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

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).

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.

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.

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
17     file.append(0)  # On rajoute un élément à droite
18     while file:  # Tant que la file est non vide
19         u = file.popleft()  # On retire l'élément à gauche
20         # Pour tous les voisins de u non coloriés on attribue la couleur différente de celle
           ↪  de u et on les ajoute à la file
21         ##############
22         #
23         # À compléter, on rappelle que l'on peut rajouter un sommet v à la file avec
           ↪  l'instruction «file.append(v)».
24         #
25         ##############
26     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`.

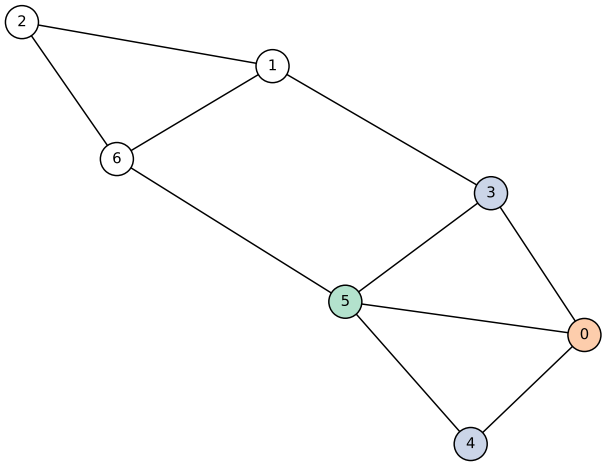
1.20. Démontrer la correction de l'algorithme précédent dans le cas où G est connexe et n'admet pas de cycle impair.

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.

1.22. Conclure.

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.

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), sat$[i]$ est maximal, et si il y a égalité deg$[i]$ est maximal (c'est un élément maximal pour l'ordre lexicographique sur 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.

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.

```
1  def coloriage_DSATUR(G):
2      """
3      Colorie un graphe avec l'algorithme DSATUR.
4      Prend en argument G (liste de liste) une liste d'adjacence du graphe.
5      Renvoie c (liste) La liste des couleurs attribuées à chaque nœud.
6      """
7      n = len(G)  # Nombre de sommets du graphe
8      # Crée une liste de couleurs
9      c = [None for _ in range(n)]
10     # Crée une liste de saturation
11     sat = [0 for _ in range(n)]
12     # Crée la liste des degrés
13     deg = liste_degrés(G)
14     # Initialisation
15     u = plus_grande_saturation(c, sat, deg)
16     # Tant qu'il reste des sommets à colorier
17     while u != None:
18         ##############
19         #
20         # À compléter
21         #
22         ##############
23         u = plus_grande_saturation(c, sat, deg)
24     return c
```

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