



ALGORITHMIQUE DES GRAPHS

PROJET: COLORIAGE D'UN GRAPHE

UNE APPLICATION A LA RESOLUTION DU SUDOKU

Professeure:	CARLA SELMI
Etudiant(e)s:	THI DIU NGHIEM OUMAR BA
Groupe:	L3 INFOSD

Introduction

- Ce projet explore les concepts de coloriage de graphes et leur application pratique dans la résolution du Sudoku.
- Nous allons examiner les différents algorithmes utilisés pour le coloriage de graphes statiques et dynamiques.
- Et nous allons démontrer l'application pratique du coloriage de graphes dans la résolution du Sudoku.



Les coloriage de graphe ont été très étudiés pour plusieurs raisons

- **Ils ont des applications dans de nombreux domaines différents.**
 - En théorie des graphes : la connectivité, la coloration et la clique.
 - En informatique : la conception de circuits électroniques, la planification de ressources et la résolution de problèmes logiques (le Sudoku).
 - En chimie : les structures moléculaires.
 - En biologie : les réseaux biologiques, tels que les réseaux de neurones et les réseaux de protéines.
 - En physique : les systèmes physiques, tels que les systèmes quantiques et les systèmes d'énergie.



Les coloriage de graphe sont utilisés dans de nombreux problèmes

- **Problème de coloration des cartes géographiques :**
 - L'objectif est de colorier les régions d'une carte de telle sorte que des régions adjacentes (partageant une frontière) aient des couleurs différentes.
- **Allocation des fréquences**
 - En télécommunications, le coloriage de graphes est utilisé pour allouer des fréquences aux antennes dans un réseau de téléphonie mobile, en veillant à ce que les antennes voisines n'utilisent pas la même fréquence.



Les coloriage de graphe sont utilisés dans de nombreux problèmes

- **Ordonnancement des examens**
 - Dans le domaine de la planification des examens, le coloriage de graphes est utilisé pour attribuer des plages horaires aux examens de manière à ce que deux examens ne se déroulent pas en même temps si des étudiants doivent passer les deux.
- **Problème des sports d'assignation**
 - Dans le cadre de la planification de tournois ou de ligues sportives, le problème du coloriage de graphes est utilisé pour attribuer des créneaux horaires aux différents matchs, en évitant les chevauchements de temps pour les équipes qui partagent des joueurs.



Implémentation

A. Graphes Statiques : Algorithmes

1. Algorithme Glouton
2. Algorithme Par Backtracking
3. Algorithme De Welsh-Powell

B. Graphes Dynamiques : Algorithmes

1. Algorithme BFS (Breadth-First Search)
2. Algorithme Par Backtracking

C. Application du coloriage à la résolution du Sudoku

1. Application de coloriage de graphe
2. Application de resolution du Sudoku



Graphes Statiques : Algorithmes

Structure des Données

Type GrapheM = Enregistrement

n : Entier;

M : Tableau[1..n, 1..n] de Booléen

Fin;



Algorithme Glouton

Attribuer des couleurs aux sommets du graphe de manière gloutonne, c'est-à-dire en choisissant la première couleur disponible.



Algorithme Glouton

1. Colorier le premier sommet (sommet 1) avec la première couleur (couleur 1)
2. Pour les $n-1$ sommets restants :
 - a. Considérer le sommet actuellement choisi et le colorier avec la couleur numérotée la plus basse qui n'a pas été utilisée sur les sommets déjà colorés adjacents à celui-ci
 - b. Si toutes les couleurs déjà utilisées apparaissent sur les sommets adjacents à ce sommet, attribuer une nouvelle couleur à celui-ci



Algorithme Glouton

Complexité en temps :
 $O(n^2)$

b. Fonction glouton

Cette fonction implémente l'algorithme de coloration gloutonne pour colorier un graphe non orienté

Fonction glouton

{Entrée : G: GrapheM du graphe non orienté}

{Sortie : result Tableau[1...n] d'Entier représentant les couleurs attribuées à chaque sommet}

Var result : Tableau[1...n] d'Entiers; colors : Tableau[1...n] de Booléens; u, i, cd : Entiers

Début

result \leftarrow -1

colors \leftarrow False

Pour u de 0 à G.n - 1 faire

 Pour i de 0 à G.n - 1 faire

 Si G[u, i] = 1 et result[i] \neq -1 alors

 colors[result[i]] \leftarrow True {Marquer la couleur de i comme indisponible}

 {Trouver la première couleur disponible}

 cd \leftarrow 1

 Tant que cd \leq G.n et colors[cd] = True faire

 cd \leftarrow cd + 1

 result[u] \leftarrow cd {Attribuer la couleur cd au sommet u}

 {Réinitialiser les couleurs pour la prochaine itération}

 Pour i de 0 à G.n - 1 faire

 Si G[u, i] = 1 et result[i] \neq -1 alors

 colors[result[i]] \leftarrow False

 {Afficher les résultats}

 Pour u de 0 à G.n - 1 faire

 Afficher "Sommet ", u + 1, " ---> Couleur ", result[u]

Retourner result {Liste des couleurs attribuées à chaque sommet}

FinFonction

Cette fonction implémente l'algorithme de coloration gloutonne pour colorier un graphe non orienté

def glouton(G):

Initialise la liste result avec -1. Cette liste stocke les couleurs attribuées à chaque sommet. Un sommet non coloré est représenté par -1
result = [-1] * len(G)

Initialise liste 'colors' avec False. Elle marque les couleurs déjà utilisées par les voisins d'un sommet. Les indices représentent les couleurs
colors = [False] * (len(G) + 1) *# Le tableau des couleurs commence à partir de l'index 1*

Attribue des couleurs aux sommets

for u **in** range(len(G)): *# Boucle à travers chaque sommet du graphe*

Marque toutes les couleurs adjacentes comme indisponibles

for i **in** range(len(G)): *# Boucle à travers les sommets voisins du sommet u*

if G[u, i] == 1 **and** result[i] != -1: *# Vérifie si u est connecté à i et si i est déjà coloré*
 colors[result[i]] = True *# Marque la couleur de i comme indisponible*

Trouve la première couleur disponible

cd = 1 *# Initialise cd à 1. cd représente la couleur actuelle en cours d'évaluation*

while cd <= len(G): *# Boucle pour trouver la première couleur disponible*

if not colors[cd]: *# Vérifie si la couleur cd n'est pas utilisée*
 break *# Sort de la boucle 'while' si la couleur est disponible*

cd += 1 *# Passe à la couleur suivante*

result[u] = cd *# Attribue la couleur cd au sommet u*

Réinitialise les couleurs pour la prochaine itération

for i **in** range(len(G)): *# Réinitialise les couleurs en marquant comme non utilisées les couleurs des voisins déjà colorés*

if G[u, i] == 1 **and** result[i] != -1: *# Vérifie si u est connecté à i et si i est déjà coloré*
 colors[result[i]] = False *# Marque la couleur de i comme disponible pour la prochaine itération*

Affiche les résultats

for u **in** range(len(G)): *# Boucle pour afficher les résultats*

print("Sommet", u + 1, "---> Couleur", result[u]) *# Affiche le numéro du sommet et la couleur qui lui a été attribuée*

return result *# Retourne le résultat de la fonction, qui est une liste contenant la couleur attribuée à chaque sommet*

Algorithme Glouton: Exemple

```
g3 = np.array([[0, 1, 1, 0, 0, 1],
               [1, 0, 1, 0, 1, 0],
               [1, 1, 0, 0, 1, 0],
               [0, 0, 0, 0, 1, 0],
               [0, 1, 1, 1, 0, 1],
               [1, 0, 1, 0, 1, 0]])

print("\nColoriage du graphe 3")

colors_g3 = glouton(versionNo(g3))

chromatic_number_g3 = chromaticNumber(colors_g3)

print("Nombre chromatique du graphe 3:",
      chromatic_number_g3)

plotGraph(versionNo(g3), colors_g3)
```

Résultat:

Coloriage du graphe 3

Sommet 1 ---> Couleur 1

Sommet 2 ---> Couleur 2

Sommet 3 ---> Couleur 3

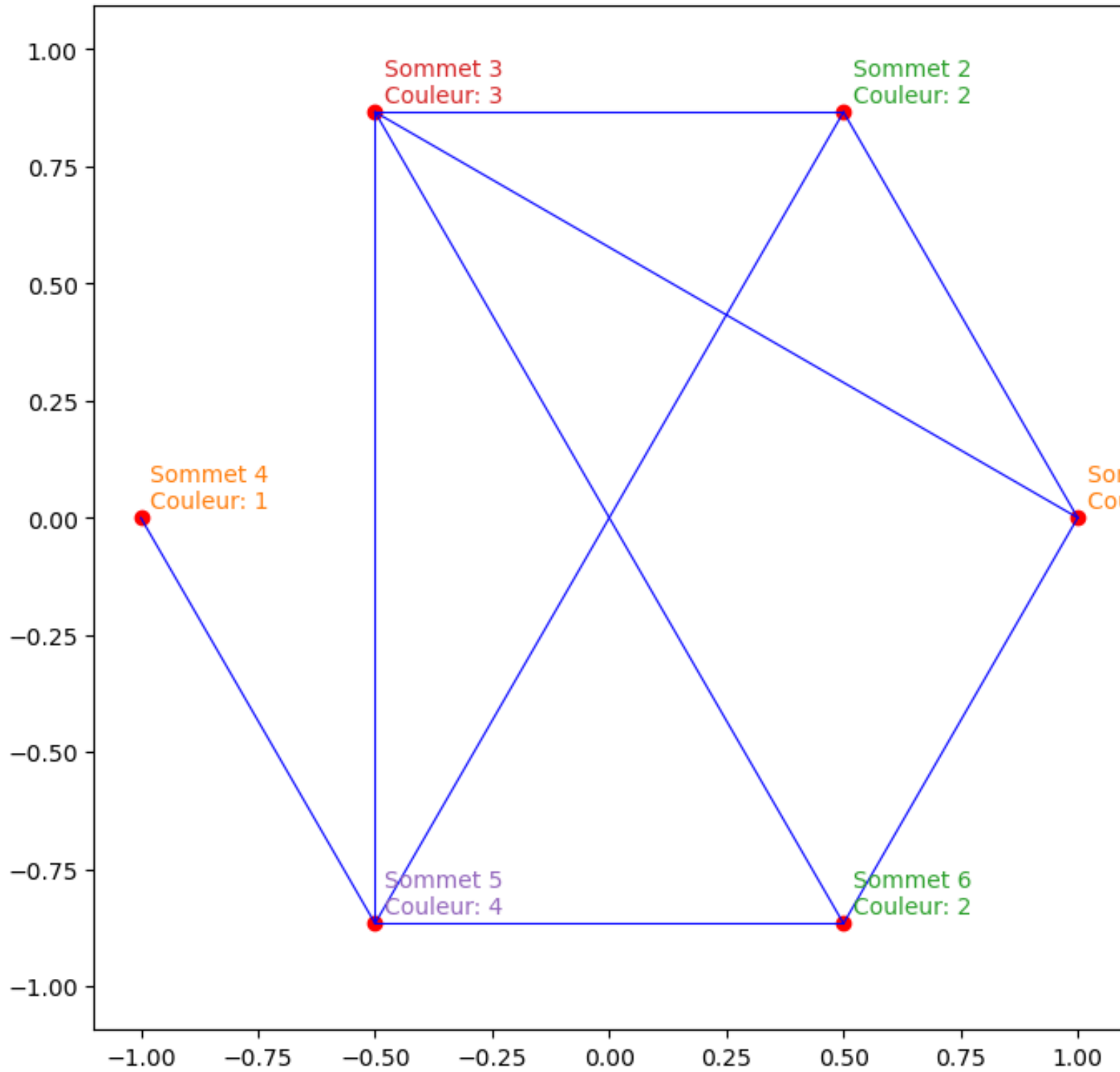
Sommet 4 ---> Couleur 1

Sommet 5 ---> Couleur 4

Sommet 6 ---> Couleur 2

Nombre chromatique du graphe 3: 4





Algorithme Par Backtracking

Explorer de manière exhaustive les différentes combinaisons de couleurs (le nombre maximal de couleurs disponibles est fixé à m), faisant preuve de backtracking lorsqu'il rencontre des configurations invalides, jusqu'à ce qu'une coloration valide soit trouvée ou que toutes les combinaisons aient été épuisées.



Algorithme Par Backtracking

- Attribuer des couleurs une par une à différents sommets, à partir du sommet 1.
- Avant d'attribuer une couleur, vérifier la sécurité en considérant les couleurs déjà affectées aux sommets adjacents, c'est-à-dire vérifier si les sommets adjacents ont la même couleur ou non.
- S'il y a une affectation de couleur qui ne viole pas les conditions, marquer l'affectation des couleurs dans le cadre de la solution.
- Si aucune affectation de couleur n'est possible, revenir en arrière (backtrack).



Algorithme Par Backtracking

c. Fonction graphColoringBacktracking

Cette fonction 'graphColoringBacktracking' utilise la récursivité pour explorer toutes les combinaisons
possibles de couleurs pour chaque sommet du graphe.

Fonction graphColoringBacktracking

{Entrée : m: Entier représentant le nombre maximum de couleurs disponibles; colorArray: Tableau[1...n]
d'Entiers représentant les couleurs attribuées aux sommets; currentVertex: Entier représentant l'indice du
sommet en cours de traitement, G: GrapheM matrice d'adjacence}

{Sortie : True si la coloration est réussie, sinon False}

Var i: Entier

Début

 Si currentVertex = G.n alors

 Retourner True

 Pour i de 1 à m faire

 Si isSafe(currentVertex, colorArray, i, G) alors

 colorArray[currentVertex] = i

 Si graphColoringBacktracking(m, colorArray, currentVertex + 1, G) alors

 Retourner True

 colorArray[currentVertex] = 0 # backtracking

 Retourner False

FinFonction

Complexité en temps : La complexité de cette fonction dépend du nombre maximal de couleurs (m) et du nombre de sommets (n). Dans le pire cas, la complexité est exponentielle, $O(m^n)$, car la fonction explore toutes les combinaisons possibles de couleurs pour chaque sommet.



"""

Cette fonction 'graphColoringBacktracking' utilise la récursivité pour explorer toutes les combinaisons possibles de couleurs pour chaque sommet du graphe.

La fonction commence par le premier sommet et tente toutes les couleurs possibles. Si une couleur est attribuée de manière sûre (vérifiée par la fonction isSafe), elle passe au sommet suivant. Si à un moment donné, aucune couleur n'est possible pour un sommet particulier, la fonction fait marche arrière (backtracking) et réessaie une couleur différente pour le sommet précédent.

Elle prend quatre paramètres :

m : le nombre maximum de couleurs disponibles

colorArray : un tableau représentant les couleurs attribuées aux sommets

currentVertex : l'indice du sommet en cours de traitement

G : la matrice d'adjacence représentant le graphe

"""

```
def graphColoringBacktracking(m, colorArray, currentVertex, G):
    if currentVertex == len(colorArray): # Vérifie si tous les sommets ont été traités
        return True # la fonction retourne True, indiquant que la coloration a été réussie

    for i in range(1, m + 1): # itère sur les couleurs de 1 à m
        if isSafe(currentVertex, colorArray, i, G): # Vérifie si attribuer la couleur i au sommet 'currentVertex' est sûr
            colorArray[currentVertex] = i # Si la couleur est sûre, attribue la couleur i au sommet 'currentVertex'
            # Appelle récursivement la fonction graphColoringBacktracking pour traiter le sommet suivant.
            if graphColoringBacktracking(m, colorArray, currentVertex + 1, G):
                # Si cette appel récursif renvoie True, càd que la coloration a été réussie pour les sommets suivants, et la fonction retourne True
                return True
            # Si l'appel récursif ne réussit pas, la couleur attribuée est réinitialisée à 0 (non attribuée) pour essayer une autre couleur (backtrack)
            colorArray[currentVertex] = 0

    # Si boucle 'for' se termine sans trouver une combinaison valide, retourne False. Impossible de colorer le graphe avec les contraintes données
    return False
```

Algorithme Par Backtracking

b. Fonction isSafe

Cette fonction détermine si attribuer une certaine couleur à un sommet particulier est sûr. Elle retourne

True si la couleur est sûre, sinon False

Fonction isSafe

{Entrée : v: Entier représentant l'indice du sommet; colorArray: Tableau[1...n] d'Entiers représentant des couleurs; color: Entier représentant la couleur que l'on veut attribuer au sommet v, G: GrapheM matrice d'adjacence}

{Sortie : True si la couleur est sûre, sinon False}

Var i: Entier

Début

 Pour i de 0 à G.n - 1 faire

 Si $G[v][i] = 1$ et $colorArray[i] = color$ alors

 Retourner False

Retourner True

FinFonction

Complexité en temps : La complexité de cette fonction est linéaire, $O(n)$, où n est la longueur du tableau des couleurs (colorArray).



```
"""
La fonction 'isSafe' prend quatre paramètres :
v : l'indice du sommet pour lequel on veut vérifier la sécurité de la couleur
colorArray : un tableau contenant les couleurs déjà attribuées aux sommets
color : la couleur que l'on veut attribuer au sommet v
G : la matrice d'adjacence représentant le graphe
Cette fonction détermine si attribuer une certaine couleur à un sommet particulier est sûr. Elle retourne True si la couleur est sûre, sinon False
"""
def isSafe(v, colorArray, color, G):
    for i in range(len(colorArray)): # itère à travers les indices des couleurs déjà attribuées aux sommets
        # Vérifie si v est connecté à i et si la couleur de i est égale à la couleur que l'on veut attribuer à v
        if G[v][i] == 1 and colorArray[i] == color:
            return False # Si les deux conditions sont satisfaites, cela signifie que la couleur n'est pas sûre, donc la fonction retourne False
    return True # Si aucun conflit de couleur n'a été trouvé, et la fonction retourne True, indiquant que la couleur est sûre pour le sommet v
```



Algorithme Par Backtracking

```
g3 = np.array([[0, 1, 1, 0, 0, 1],  
               [1, 0, 1, 0, 1, 0],  
               [1, 1, 0, 0, 1, 0],  
               [0, 0, 0, 0, 1, 0],  
               [0, 1, 1, 1, 0, 1],  
               [1, 0, 1, 0, 1, 0]])  
  
print("Coloriage du graphe 3")  
plotGraph(versionNo(g3), m)
```

Résultat:

Coloriage du graphe 3

La coloration est possible !

Le nombre chromatique du graphe est : 3

Les couleurs attribuées sont les suivantes :

Sommet 1 - Couleur : 1

Sommet 2 - Couleur : 2

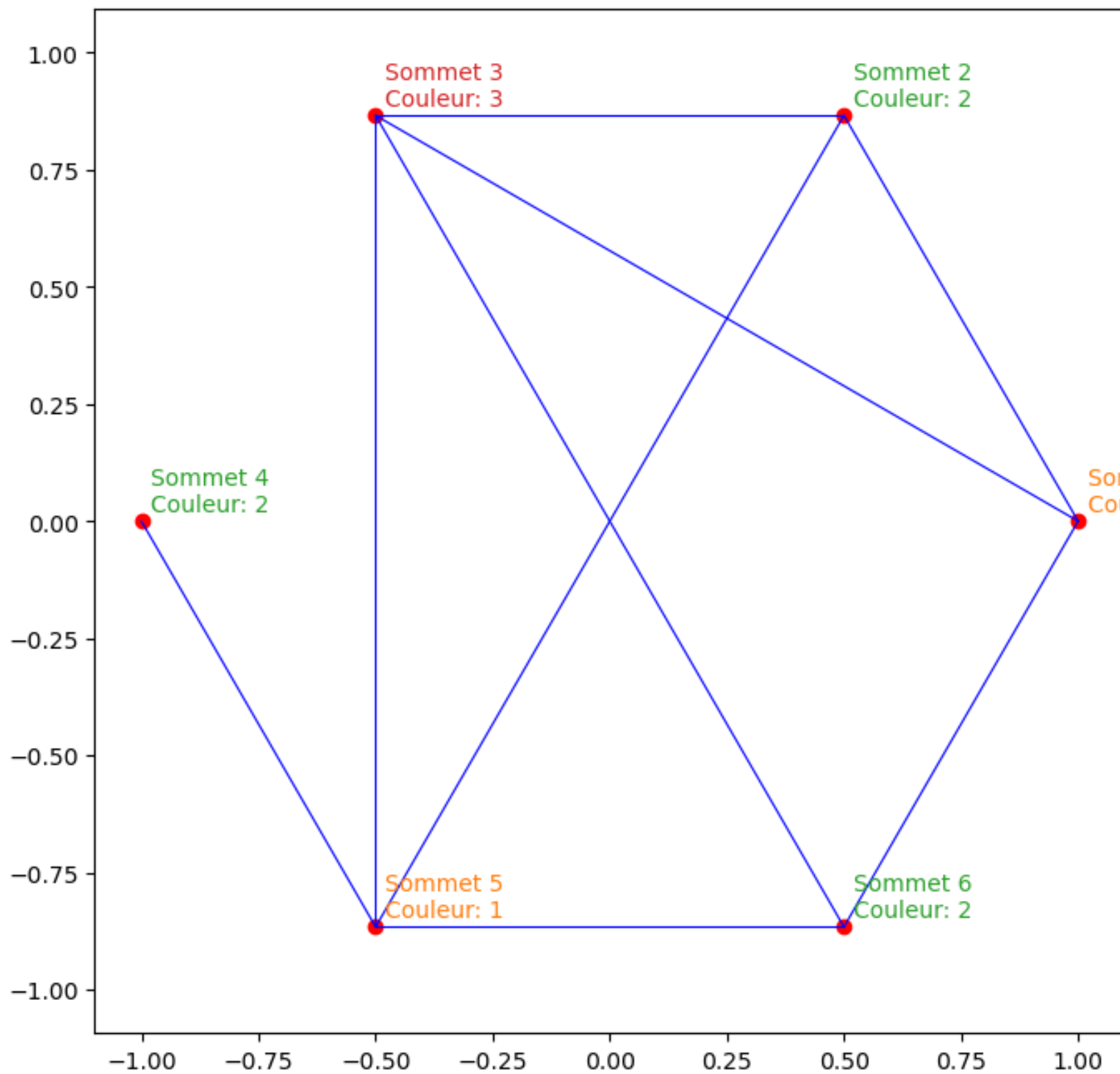
Sommet 3 - Couleur : 3

Sommet 4 - Couleur : 2

Sommet 5 - Couleur : 1

Sommet 6 - Couleur : 2





Algorithme De Welsh-Powell

- Choisir le sommet non colorié avec le maximum d'arêtes adjacentes non colorées, puis attribuer la couleur minimale qui n'est pas déjà utilisée par ses voisins déjà coloriés.
- Cette approche vise à optimiser la coloration du graphe en favorisant l'utilisation de couleurs inférieures.



Algorithme De Welsh-Powell

1. Repérer le degré de chaque sommet.
2. Ranger les sommets par ordre de degrés décroissants (dans certains cas plusieurs possibilités).
3. Attribuer au premier sommet (sommet 1) de cette liste ordonnée la couleur 1.
4. Suivre la liste en attribuant la même couleur au premier sommet (*) qui ne soit pas adjacent au sommet 1.
5. Suivre (si possible) la liste jusqu'au prochain sommet qui ne soit adjacent ni à 1 ni à (*).
6. Continuer jusqu'à ce que la liste soit finie.
7. Prendre une deuxième couleur (couleur 2) pour le premier sommet (**) non encore colorié de la liste.
8. Répéter les opérations 4 à 7.
9. Continuer jusqu'à avoir colorié tous les sommets.



Algorithme De Welsh- Powell

c. Fonction `welsh_powell_algo`

Cette fonction `welsh_powell_algo` implémente l'algorithme de coloration de graphes de Welsh-Powell.

Fonction `welsh_powell_algo`

{Entrée : G: GrapheM Matrice d'adjacence}

{Sortie : result: Dictionnaire associant chaque sommet à sa couleur}

Var `color_list`: Tableau[1...n] d'Entiers; `diff_list`: Tableau[1...order+1] d'Entiers; `colored_vertex`: Tableau[1...n] d'Entiers; `K`, `z` : Tableaux d'Entiers; `order`, `c`, `i`, `j`, `w`: Entier; `result` : Dictionnaire des clés et des valeurs Entières

Début

`order` \leftarrow G.n

`color_list` \leftarrow `order` + 1

`colored_vertex` \leftarrow \emptyset

`K` \leftarrow \emptyset

`z` \leftarrow \emptyset

Pour `c` de 0 à `order` - 1 faire

// Trouver le sommet non coloré avec le plus grand nombre d'arêtes non colorées

 Pour `i` de 0 à `order` - 1 faire

 Pour `j` de 0 à `order` - 1 faire

 Si `G[i][j]` = 1 alors

 Si `i+1` \notin `colored_vertex` alors

`z.append(i+1)`

`w` \leftarrow Maximum de `z` en fonction du nombre d'occurrences

 pour `j` de 0 à `order` - 1 faire:

 si `G[w-1][j]` = 1:

`K.append(color_list[j])`

Algorithme De Welsh- Powell

Complexité en temps:
 $O(n^2)$

```
// Attribuer des couleurs
diff_list ← list(set(range(1, order + 1)) - set(K))
color_list[w-1] ← Minimum de diff_list
# Nettoyer les listes temporaires K et diff_list pour la prochaine itération
K ← ∅
diff_list ← ∅
```

```
colored_vertex.append(w)
// Coloration des sommets restants
Pour j de 0 à order - 1 faire
    Si G[w-1][j] = 0 alors
        Si j+1 ∉ colored_vertex alors
            Si color_list[j] = order + 1 alors
                Pour i de 0 à order - 1 faire
                    Si G[j][i] = 1 alors
                        K.append(color_list[i])
            Si color_list[w-1] ∉ K alors
                color_list[j] ← color_list[w-1]
                colored_vertex.append(j+1)
                w ← j+1

    K ← ∅
```

```
result ← zip(range(1, order+1), color_list)
Retourner result
```

FinFonction

```

def welsh_powell_algo(G):
    order = len(G) # Calcule le nombre de sommets du graphe
    # Initialise une liste color_list avec des valeurs order + 1, représentant les couleurs attribuées à chaque sommet.
    color_list = [order + 1] * order # Le choix de order + 1 comme couleur initiale permet de détecter les sommets non colorés.
    colored_vertex = [] # Initialise une liste vide pour suivre les sommets déjà colorés
    # Initialise des listes temporaires K et z pour les calculs dans l'algorithme.
    K = []
    z = []

    for c in range(order): # Boucle principale de l'algorithme qui itère pour chaque couleur

        # Trouver le sommet non coloré avec le plus grand nombre d'arêtes non colorées :
        for i in range(order): # Boucle pour parcourir les sommets du graphe
            for j in range(order): # Boucle pour parcourir les sommets adjacents à chaque sommet
                if G[i][j] == 1: # Vérifie si les sommets i et j sont adjacents
                    if i+1 not in colored_vertex: # Vérifie si le sommet i n'est pas déjà coloré
                        z.append(i+1) # Les sommets adjacents non colorés sont ajoutés à la liste temporaire z
            w = max(set(z), key=z.count) # Trouve le sommet w avec le plus grand nombre d'arêtes adjacentes non colorées

        # Attribuer des couleurs :
        for j in range(order): # Boucle pour parcourir les sommets adjacents à w
            if G[w-1][j] == 1: # Vérifie si le sommet w est adjacent à j
                K.append(color_list[j]) # Ajoute la couleur du sommet j à la liste K (K stocke les couleurs des sommets adjacents à w)
        # Calcule la liste des couleurs disponibles pour le sommet w
        diff_list = list(set(range(1, order + 1)) - set(K)) # Elle contient toutes les couleurs qui ne sont pas encore utilisées par les voisins du w
        color_list[w-1] = min(diff_list) # Attribue au sommet w la couleur minimale disponible (la plus petite couleur disponible)
        # Nettoie les listes temporaires K et diff_list pour la prochaine itération
        K.clear()
        diff_list.clear()
        colored_vertex.append(w) # Ajoute le sommet w à la liste des sommets colorés

    # Coloration des sommets restants :
    for j in range(order): # Boucle pour parcourir les sommets adjacents à w qui ne sont pas encore colorés
        if G[w-1][j] == 0: # Vérifie si le sommet w n'est pas adjacent à j
            if j+1 not in colored_vertex: # Vérifie si le sommet j n'est pas déjà coloré
                if color_list[j] == order + 1: # Vérifie si le sommet j n'est pas encore coloré
                    for i in range(order): # Boucle pour parcourir et vérifier les sommets adjacents à j
                        if G[j][i] == 1: # Vérifie si le sommet j est adjacent à i
                            K.append(color_list[i]) # Ajoute la couleur du sommet i à la liste K
                    if color_list[w-1] not in K: # Vérifie si la couleur de w n'est pas dans K (la liste des couleurs des sommets adjacents à j)
                        color_list[j] = color_list[w-1] # Attribue au sommet j la couleur de w (j prend la couleur de w)
                        colored_vertex.append(j+1) # Ajoute le sommet j à la liste des sommets colorés
                        w = j+1 # Met à jour w avec le sommet j
                    K.clear() # Nettoie la liste temporaire K pour la prochaine itération

    # La fonction renvoie un dictionnaire où les clés sont les numéros de sommet et les valeurs sont les couleurs attribuées
    result = zip(range(1, order+1), color_list) # Crée une séquence de paires (sommet, couleur)
    return dict(result) # Convertit la séquence en un dictionnaire représentant la coloration du graphe

```


Algorithme De Welsh-Powell

```
g3 = np.array([[0, 1, 1, 0, 0, 1],  
               [1, 0, 1, 0, 1, 0],  
               [1, 1, 0, 0, 1, 0],  
               [0, 0, 0, 0, 1, 0],  
               [0, 1, 1, 1, 0, 1],  
               [1, 0, 1, 0, 1, 0]])  
  
print("Coloriage du graphe 3")  
  
coloring3 = welsh_powell_algo(versionNo(g3))  
  
plotGraph(versionNo(g3), coloring3)
```

Résultat:

Coloriage du graphe 3

Le nombre chromatique du graphe est : 3

Couleurs assignées à chaque sommet :

Sommet 1 : Couleur 2

Sommet 2 : Couleur 3

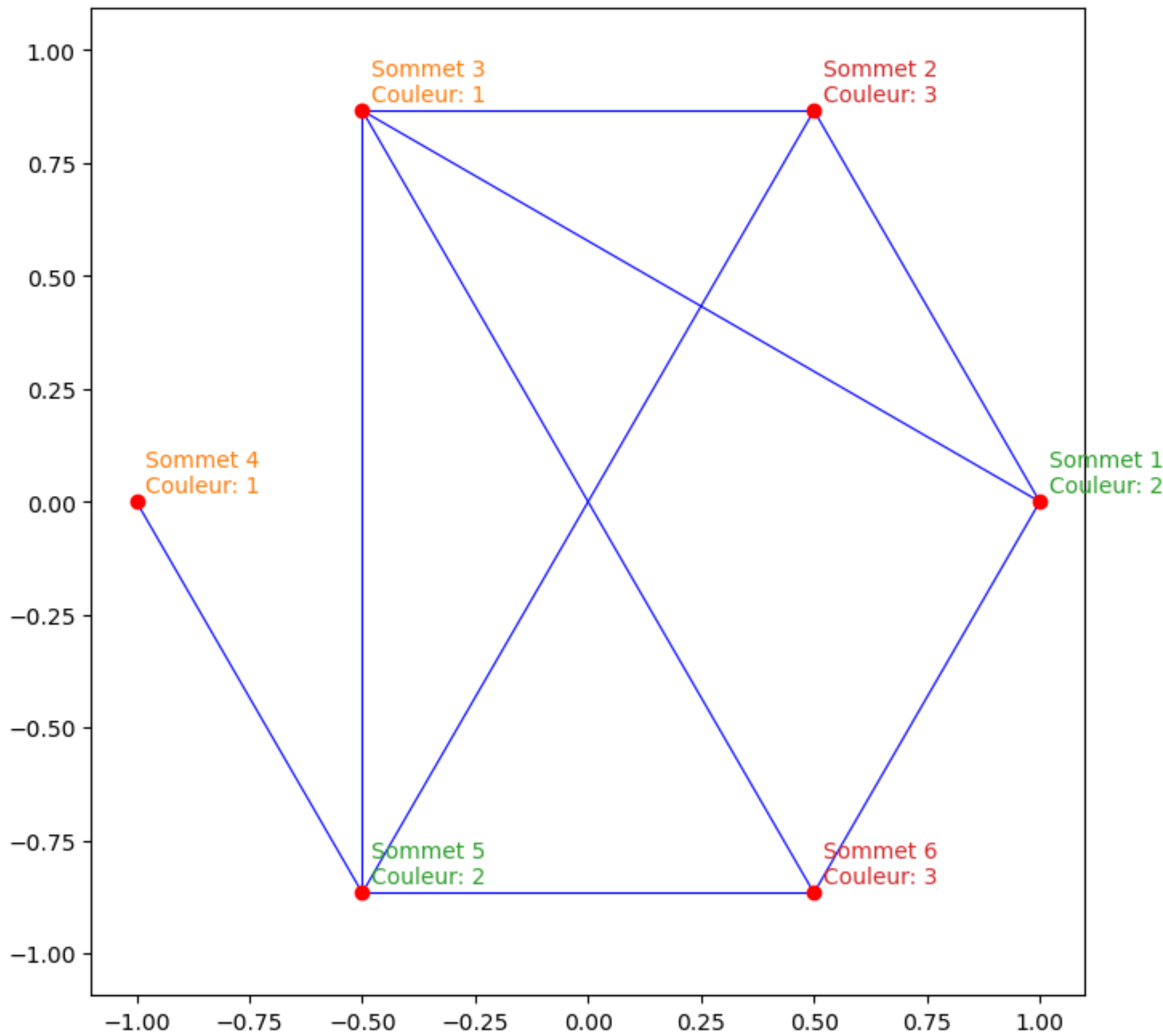
Sommet 3 : Couleur 1

Sommet 4 : Couleur 1

Sommet 5 : Couleur 2

Sommet 6 : Couleur 3





Graphes Dynamiques : Algorithmes



Algorithme BFS (Breadth-First Search)

- L'algorithme utilisé implémente une méthode de coloration de graphes dynamiques basée sur une approche de recherche en largeur (BFS).
- C'est un processus itératif où les sommets du graphe sont visités et coloriés au fur et à mesure de leur découverte.



Algorithme BFS (Breadth-First Search)

1. Initialisation :

- Créer une file d'attente (queue) et ajouter le sommet 1.
- Marquer le sommet 1 comme visité.

2. Boucle Principale :

Tant que la file d'attente n'est pas vide, répéter les étapes suivantes.

a. Traitement du Sommet Actuel :

- Retirer le sommet 1 de la file d'attente.
- Traiter le sommet (colorier le sommet).

b. Exploration des Voisins :

- Pour chaque voisin du sommet actuel qui n'a pas encore été visité, le marquer comme visité et l'ajouter à la file d'attente.

3. Terminaison :

- L'algorithme se termine lorsque la file d'attente est vide, ce qui signifie que tous les sommets accessibles ont été visités.



Algorithme BFS

b. Méthode color_graph

Cette méthode colorie l'ensemble du graphe de manière cohérente

Méthode color_graph

{Entrée : Aucune}

{Sortie : Aucune}

Var m, vertex: Entier

Début

$m \leftarrow \text{len}(\text{self.vertices})$

 Pour chaque vertex \in self.vertices faire

 Si self.colors[vertex] = None alors

 self.color_connected_component(vertex)

FinMéthode

Complexité en temps: La complexité temporelle de cette méthode dépend du nombre total de sommets dans le graphe, noté n . La complexité serait généralement $O(n)$.



Algorithme BFS

c. Méthode `color_connected_component`

Cette méthode explore la composante connectée (CC) du sommet initial, attribuant des couleurs
cohérentes aux sommets en évitant les couleurs déjà utilisées par les voisins

Méthode `color_connected_component`

{Entrée : `start_vertex`: Entier représentant le sommet initial}

{Sortie : Aucune}

Var `queue`: Deque d'Entiers représentant les sommets; `used_colors`: Ensemble d'Entier; `current_vertex`,
`adjacent_vertex`, `color`, `new_color`: Entier; `available_colors`: Ensemble d'Entier de 1 à `len(self.vertices)`;

Début

`queue` \leftarrow deque(`[start_vertex]`)

`used_colors` $\leftarrow \emptyset$

Tant que `queue` $\neq \emptyset$ faire

`current_vertex` \leftarrow `queue.popleft()`

`available_colors` \leftarrow Ensemble d'Entier de 1 à `len(self.vertices)`

Pour chaque `edge` \in `self.vertices[current_vertex]` faire

`adjacent_vertex` \leftarrow Sommet \in `edge` sauf `current_vertex`

Si `adjacent_vertex` \in `self.colors` et `self.colors[adjacent_vertex]` \neq None alors

Retirer `self.colors[adjacent_vertex]` de `available_colors`

Si `available_colors` $\neq \emptyset$ alors

Si `current_vertex` \in `self.edges_added` alors

Retirer `self.colors[current_vertex]` de `available_colors`

`color` \leftarrow Minimum(`available_colors`)

Algorithme BFS

`self.colors[current_vertex] ← color`

`used_colors.add(color)`

Pour chaque `edge ∈ self.vertices[current_vertex]` faire

`adjacent_vertex ← Entier ∈ edge` sauf `current_vertex`

Si `adjacent_vertex ∈ self.colors` et `self.colors[adjacent_vertex] = None` alors

`queue.append(adjacent_vertex)`

Si `self.edges_added ≠ ∅` alors

Pour chaque `vertex ∈ self.edges_added` faire

Si `vertex ≠ current_vertex` et `vertex ∉ queue` alors

`queue.append(vertex)`

Sinon alors

`new_color ← Maximum(used_colors) + 1`

`self.colors[current_vertex] ← new_color`

`used_colors.add(new_color)`

Pour chaque `edge ∈ self.vertices[current_vertex]` faire

`adjacent_vertex ← Entier ∈ edge` sauf `current_vertex`

Si `adjacent_vertex ∈ self.colors` et `self.colors[adjacent_vertex] = None` alors

`queue.append(adjacent_vertex)`

`self.edges_added ← ∅` # Réinitialiser l'ensemble des arêtes ajoutées

`self.chromatic_number()` # Mettre à jour le nombre chromatique

FinMéthode

Complexité en temps: La complexité temporelle de cette méthode dépend du nombre total de sommets dans le graphe, noté n . La complexité serait généralement notée comme $O(n)$.

```

def color_connected_component(self, start_vertex):
    queue = deque([start_vertex]) # Crée une deque 'queue' avec Le sommet initial comme élément unique pour parcourir les sommets de la CC
    used_colors = set() # Initialise un ensemble vide 'used_colors' pour suivre les couleurs déjà utilisées dans la CC

    while queue: # boucle qui continue tant que la file d'attente n'est pas vide (càd qu'il reste des sommets à explorer dans la CC)
        current_vertex = queue.popleft() # Retire et récupère le sommet de 'queue'. Il est actuellement examiné dans le processus de coloration
        available_colors = set(range(1, len(self.vertices) + 1)) # Initialise un ensemble de couleurs disponibles (de 1 au nombre total de sommets)

        for edge in self.vertices[current_vertex]: # Parcourt toutes les arêtes (edges) associées au sommet actuel dans le dictionnaire vertices
            adjacent_vertex = next(iter(set(edge) - {current_vertex})) # Détermine le sommet adjacent au sommet actuel via l'arête en cours d'examen

            if adjacent_vertex in self.colors and self.colors[adjacent_vertex] is not None: # Vérifie si le sommet adj a déjà 1 couleur attribuée
                available_colors.discard(self.colors[adjacent_vertex]) # cette couleur est retirée de l'ensemble des couleurs disponibles

        if available_colors: # Vérifie si des couleurs sont disponibles après avoir examiné les voisins
            # Modification pour s'assurer que la couleur du sommet associée à la nouvelle arête est différente
            if current_vertex in self.edges_added: # Vérifie si le sommet actuel est associé à une nouvelle arête ajoutée
                available_colors.discard(self.colors[current_vertex]) # Supprime la couleur du sommet actuel de l'ensemble des couleurs disponibles

            color = min(available_colors) # choisit la plus petite couleur disponible
            self.colors[current_vertex] = color # Attribue cette couleur au sommet actuel dans le dictionnaire 'colors'
            used_colors.add(color) # Ajoute la couleur utilisée à l'ensemble des couleurs déjà utilisées 'used_colors'

            for edge in self.vertices[current_vertex]: # Parcourt à nouveau les arêtes associées au sommet actuel
                adjacent_vertex = next(iter(set(edge) - {current_vertex})) # Détermine à nouveau le sommet adjacent à partir de l'arête

                # Vérifie si le sommet adj n'a pas encore de couleur attribuée
                if adjacent_vertex in self.colors and self.colors[adjacent_vertex] is None:
                    queue.append(adjacent_vertex) # Ajoute le sommet adjacent à la file d'attente pour le traiter ultérieurement

            if self.edges_added: # Vérifie si de nouvelles arêtes ont été ajoutées au graphe
                for vertex in self.edges_added: # Itère sur les sommets associés aux arêtes nouvellement ajoutées
                    # Vérifie si le sommet n'est pas le sommet actuel et n'a pas encore été ajouté à 'deque'
                    if vertex != current_vertex and vertex not in queue:
                        queue.append(vertex) # ajoute le sommet à la file pour un traitement ultérieur

        else: # Si aucune couleur n'est disponible pour le sommet actuel
            new_color = max(used_colors) + 1 # Trouve la couleur maximale utilisée et l'incrémente pour obtenir une nouvelle couleur
            self.colors[current_vertex] = new_color # Attribue la nouvelle couleur au sommet actuel
            used_colors.add(new_color) # Ajoute la nouvelle couleur à l'ensemble des couleurs utilisées

            for edge in self.vertices[current_vertex]: # Itère sur les arêtes du sommet actuel
                adjacent_vertex = next(iter(set(edge) - {current_vertex})) # Détermine le sommet adjacent au sommet actuel dans l'arête actuelle

                # Vérifie si le sommet adjacent n'a pas encore été attribué une couleur
                if adjacent_vertex in self.colors and self.colors[adjacent_vertex] is None:
                    queue.append(adjacent_vertex) # ajoute le sommet adjacent à la file pour un traitement ultérieur

    self.edges_added = set() # Réinitialise l'ensemble des arêtes ajoutées après le traitement de la composante connectée
    self.chromatic_number() # Met à jour le nombre chromatique après que la composante connectée a été colorée

```

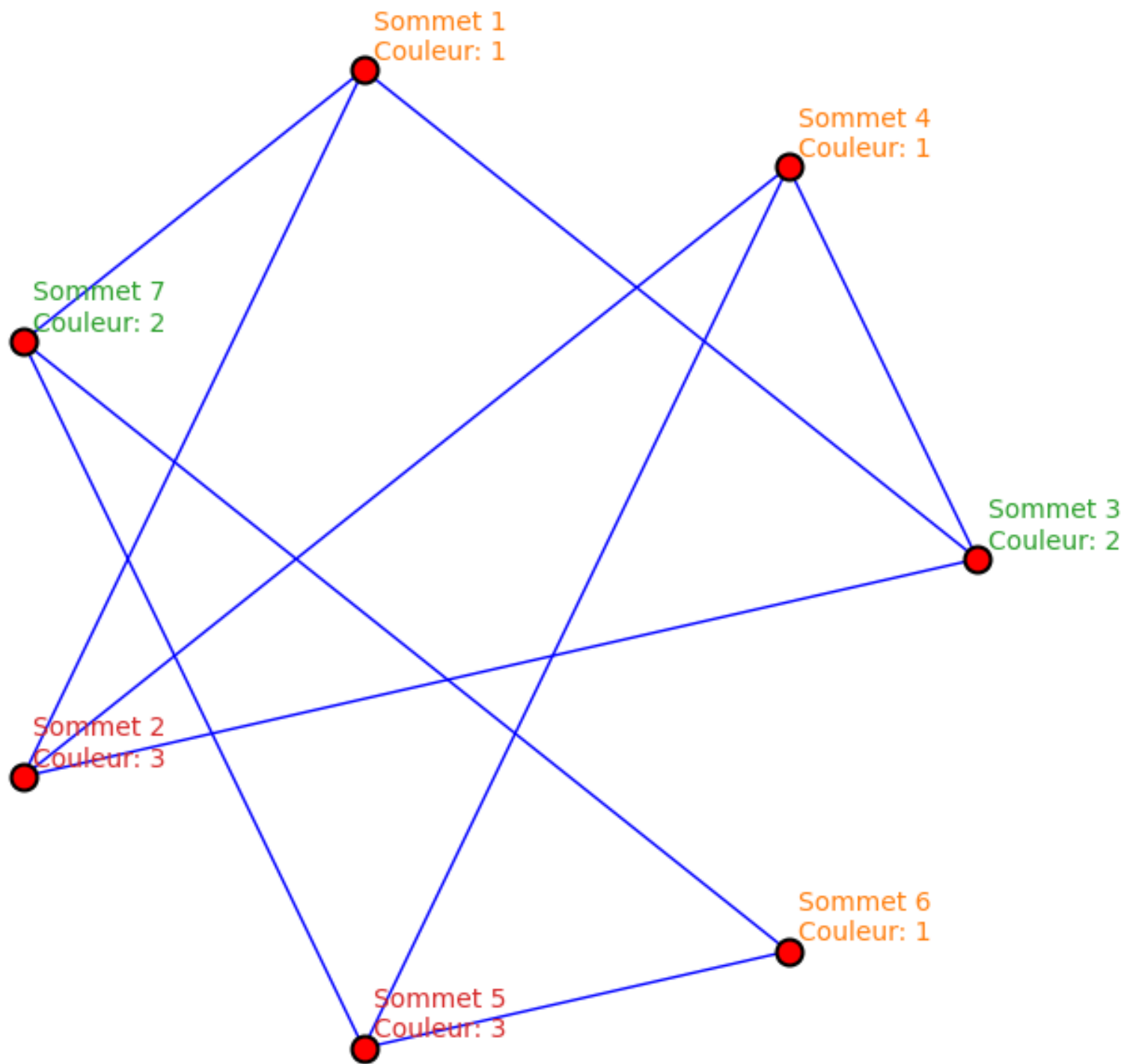
Algorithme BFS: Exemple

```
Entrer le nombre de sommets pour le graphe 1: 7
Entrer le nombre d'arêtes pour le graphe 1: 10
Entrer une arête (format: v1 v2): 1 2
Entrer une arête (format: v1 v2): 1 3
Entrer une arête (format: v1 v2): 1 7
Entrer une arête (format: v1 v2): 2 3
Entrer une arête (format: v1 v2): 2 4
Entrer une arête (format: v1 v2): 3 4
Entrer une arête (format: v1 v2): 4 5
Entrer une arête (format: v1 v2): 5 6
Entrer une arête (format: v1 v2): 5 7
Entrer une arête (format: v1 v2): 6 7
```

Colored Dynamic Graph:

```
Vertices: {1: {frozenset({1, 3}), frozenset({1, 7}), frozenset({1, 2})}, 2:
{frozenset({2, 4}), frozenset({2, 3}), frozenset({1, 2})}, 3: {frozenset({3,
4}), frozenset({1, 3}), frozenset({2, 3})}, 4: {frozenset({3, 4}), frozenset({2,
4}), frozenset({4, 5})}, 5: {frozenset({4, 5}), frozenset({5, 6}), frozenset({5,
7})}, 6: {frozenset({6, 7}), frozenset({5, 6})}, 7: {frozenset({6, 7}),
frozenset({1, 7}), frozenset({5, 7})}}
Edges: {frozenset({3, 4}), frozenset({1, 7}), frozenset({2, 3}), frozenset({1,
2}), frozenset({4, 5}), frozenset({6, 7}), frozenset({2, 4}), frozenset({5, 6}),
frozenset({1, 3}), frozenset({5, 7})}
Colors: {1: 1, 2: 3, 3: 2, 4: 1, 5: 3, 6: 1, 7: 2}
Nombre chromatique du graphe 1: 3
```





Algorithme Par Backtracking

- Attribuer des couleurs aux sommets d'un graphe dynamique de manière réursive.
- Explorer toutes les combinaisons possibles de couleurs pour chaque sommet, en vérifiant la validité de la couleur en s'assurant qu'aucun des voisins n'a la même couleur.
- En cas de conflit, l'algorithme rétrograde et explore d'autres options de couleurs jusqu'à ce que tous les sommets soient colorés de manière cohérente ou que toutes les possibilités aient été épuisées.



Algorithme Par Backtracking

- 1. Initialisation :** Initialiser le graphe avec les sommets et les arêtes, ainsi que les structures de données nécessaires pour suivre les couleurs attribuées à chaque sommet.
- 2. Attribution de Couleurs :** Attribuer des couleurs aux sommets un par un, en utilisant une approche récursive. Pour chaque sommet, explorer toutes les couleurs possibles.
- 3. Validation de la Couleur :** Avant d'attribuer une couleur à un sommet, vérifier si la couleur est valide en s'assurant qu'aucun des voisins du sommet n'a la même couleur.
- Rétrogradation (Backtracking) :** Si une couleur ne fonctionne pas pour un sommet, rétrograder en revenant en arrière pour essayer une autre couleur. Cela se fait de manière récursive.
- 4. Réussite ou Échec :** Répéter le processus d'attribution de couleurs jusqu'à ce que tous les sommets soient colorés avec succès ou qu'il n'y ait pas de couleur valide pour un sommet particulier.



Algorithme Par Backtracking

d. Méthode `is_valid_color`

Cette méthode vérifie si attribuer une certaine couleur à un sommet est valide

Méthode `is_valid_color`

{Entrée : vertex: Entier représentant le sommet du graphe; color: Entier représentant la couleur à vérifier}

{Sortie : Booléen (True si la couleur est valide, False sinon)}

Var adjacent_vertex: Entier, edge: Frozenset[Entier]

Début

Si $\text{vertex} \notin \text{self.vertices}$ ou $\text{vertex} \in \text{self.deleted_vertices}$ alors

Retourner False # la couleur n'est pas valide

FinSi

Pour chaque $\text{edge} \in \text{self.vertices}[\text{vertex}]$ faire

Trouve le sommet adjacent à 'vertex' dans l'arête en cours

$\text{adjacent_vertex} \leftarrow \text{PremièreIntersection}(\text{Ensemble}(\text{edge}), \{\text{vertex}\})$

Si (

$\text{adjacent_vertex} \in \text{self.colors}$

et $\text{self.colors}[\text{adjacent_vertex}] = \text{color}$

et $\text{adjacent_vertex} \notin \text{self.deleted_vertices}$

) alors

Retourner False # retourne False, indiquant que la couleur n'est pas valide

FinSi

FinPour

Si aucune des conditions ci-dessus n'est satisfaite, la méthode retourne True

Retourner True

FinMéthode

Complexité en temps: La complexité de cette méthode dépend du nombre d'arêtes associées au sommet vertex et du nombre de sommets adjacents à vertex. Notons d le degré du sommet vertex. En moyenne, la complexité serait linéaire par rapport au degré du sommet $O(d)$. Dans le pire des cas, lorsque le graphe est dense, d peut être proche du nombre total de sommets n, et la complexité peut être considérée comme $O(n)$.

```
def is_valid_color(self, vertex, color): # Cette méthode vérifie si attribuer une certaine couleur à un sommet est valide
    # Vérifie si le sommet 'vertex' n'est pas dans self.vertices ou s'il a été marqué comme supprimé (vertex in self.deleted_vertices)
    if vertex not in self.vertices or vertex in self.deleted_vertices:
        return False # la couleur n'est pas valide

    for edge in self.vertices[vertex]: # Parcourt chaque arête associée au sommet 'vertex'
        # Trouve le sommet adjacent à 'vertex' dans l'arête en cours
        adjacent_vertex = next(iter(set(edge) - {vertex})) # obtenir le sommet adj à un sommet donné dans 1 arête
        if ( # Vérifie si le sommet adjacent a déjà une couleur attribuée et si cette couleur est la même que celle qu'on veut attribuer à 'vertex'
            adjacent_vertex in self.colors
            and self.colors[adjacent_vertex] == color
            and adjacent_vertex not in self.deleted_vertices # Vérifie également que le sommet adjacent n'a pas été marqué comme supprimé
        ):
            return False # retourne False, indiquant que la couleur n'est pas valide
    # Si aucune des conditions ci-dessus n'est satisfaite, la méthode retourne True, indiquant que la couleur est valide pour le sommet 'vertex'
    return True
```



Algorithme Par Backtracking

e. Méthode `color_graph_backtracking`

Cette méthode met en œuvre l'algorithme de coloration de graphe par rétrogradation (backtracking)

Méthode `color_graph_backtracking`

{Entrée : vertex: Entier représentant l'indice du sommet en cours de traitement}

{Sortie : Booléen (True si la coloration est réussie, False sinon)}

Var m, color: Entier

Début

m ← Longueur(self.vertices)

Si vertex > m alors

Retourner True # Tous les sommets ont été colorés, la fonction retourne True

FinSi

Pour chaque color de 1 à m faire

Si self.is_valid_color(vertex, color) alors

self.colors[vertex] = color

Si self.color_graph_backtracking(vertex + 1) alors

Retourner True

FinSi

self.colors[vertex] = None

FinSi

FinPour

Retourner False

FinMéthode

Complexité en temps: La complexité en temps de cette fonction dépend du nombre de sommets du graphe et du nombre maximal de couleurs possibles. Dans le pire des cas, la complexité est $O(m^n)$, où n est le nombre de sommets et m est le nombre maximal de couleurs possibles.

```
def color_graph_backtracking(self, vertex=1): # Cette méthode met en œuvre l'algorithme de coloration de graphe par rétrogradation (backtracking)
    m = len(self.vertices) # Calcule le nombre total de sommets dans le graphe et l'assigne à la variable m
    if vertex > m: # Vérifie si vertex dépasse le nombre total de sommets, ce qui signifie que tous les sommets ont été colorés
        return True # La fonction retourne True

    for color in range(1, m + 1): # Boucle à travers toutes les couleurs possibles (de 1 à m inclus)
        if self.is_valid_color(vertex, color): # Vérifie si attribuer 'color' à 'vertex' est une coloration valide
            self.colors[vertex] = color # Si la couleur est valide, elle est attribuée au sommet 'vertex'

            if self.color_graph_backtracking(vertex + 1): # passe au sommet suivant (vertex + 1) et si cela retourne True
                return True # La fonction retourne True (la coloration a réussi pour les sommets suivants)
            # Si la coloration n'est pas réussie, la couleur attribuée à 'vertex' est remise à None pour essayer une autre couleur dans la boucle
            self.colors[vertex] = None

    # Si aucune couleur n'a conduit à une coloration valide pour les sommets suivants, la fonction retourne False.
    # Cela déclenche le mécanisme de rétrogradation (backtracking) pour essayer une autre couleur pour le sommet précédent dans l'appel récursif
    return False
```



Algorithme Par Backtracking: Exemple

Résultat:

Enter the number of vertices for graph 1: 7

Enter the number of edges for graph 1: 10

Enter an edge (format: 'v1 v2'): 1 2

Enter an edge (format: 'v1 v2'): 1 4

Enter an edge (format: 'v1 v2'): 1 7

Enter an edge (format: 'v1 v2'): 2 3

Enter an edge (format: 'v1 v2'): 2 4

Enter an edge (format: 'v1 v2'): 3 4

Enter an edge (format: 'v1 v2'): 4 5

Enter an edge (format: 'v1 v2'): 5 6

Enter an edge (format: 'v1 v2'): 5 7

Enter an edge (format: 'v1 v2'): 6 7

Colored Dynamic Graph:

Vertices: {1: {frozenset({1, 4}), frozenset({1, 7}), frozenset({1, 2})}, 2: {frozenset({2, 4}), frozenset({2, 3}), frozenset({1, 2})}, 3: {frozenset({3, 4}), frozenset({2, 3})}, 4: {frozenset({3, 4}), frozenset({2, 4}), frozenset({1, 4}), frozenset({4, 5})}, 5: {frozenset({4, 5}), frozenset({5, 6}), frozenset({5, 7})}, 6: {frozenset({6, 7}), frozenset({5, 6})}, 7: {frozenset({6, 7}), frozenset({1, 7}), frozenset({5, 7})}}

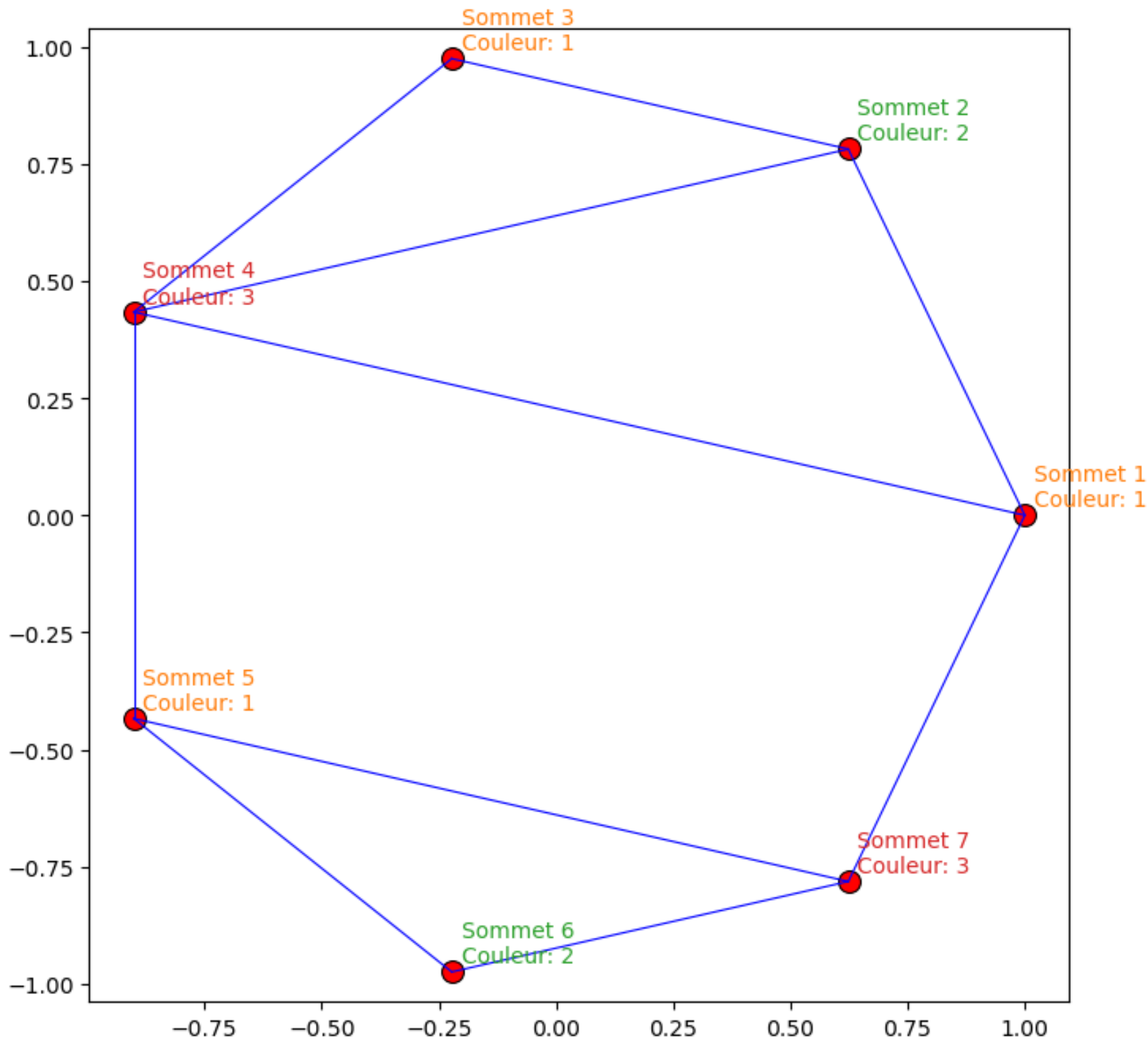
Edges: {frozenset({3, 4}), frozenset({1, 4}), frozenset({1, 7}), frozenset({2, 3}), frozenset({1, 2}), frozenset({4, 5}), frozenset({6, 7}), frozenset({2, 4}), frozenset({5, 6}), frozenset({5, 7})}

Colors: {1: 1, 2: 2, 3: 1, 4: 3, 5: 1, 6: 2, 7: 3}

Chromatic Number: 3

Chromatic Number of graph 1: 3

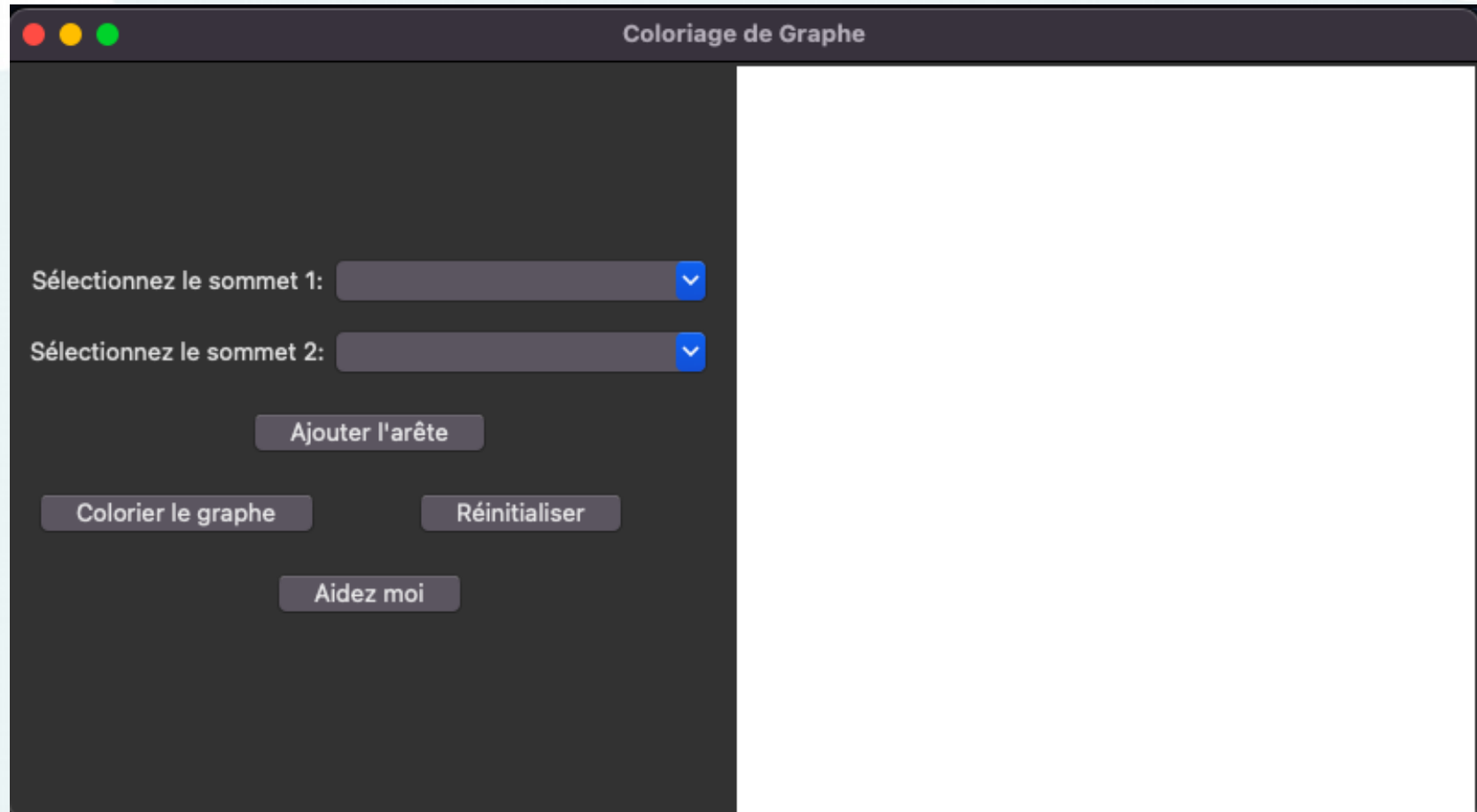




Application Du Coloriage A La Résolution Du Sudoku



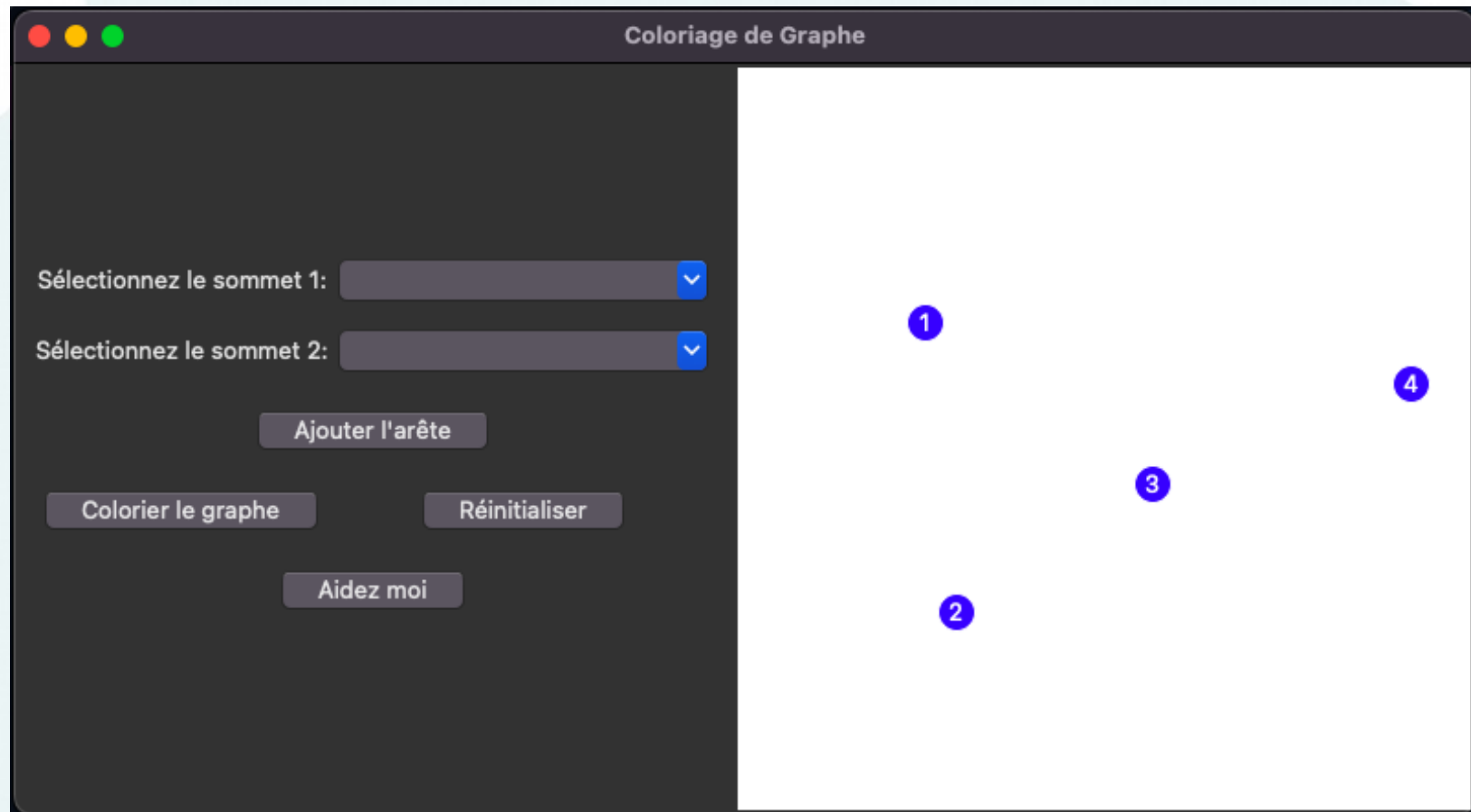
Application de coloriage de graphe



Lors de la création de la fenêtre, une instance de *DynamicGraph* est créée



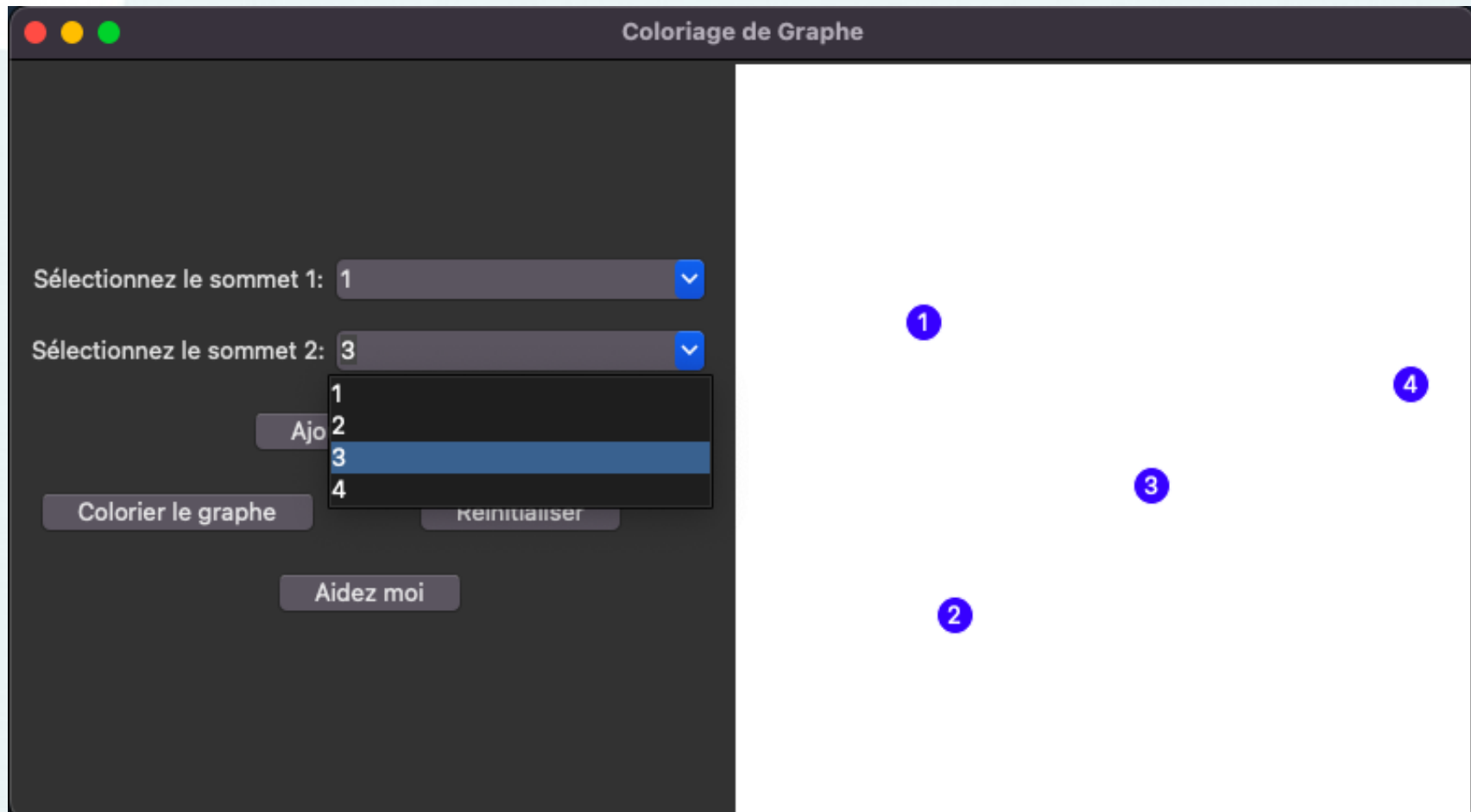
Application de coloriage de graphe



A droite de la fenêtre il y'a un canvas pour dessiner le graphe. A chaque fois qu'on clique sur le canvas : un nouveau sommet se crée et sera numéroté automatiquement



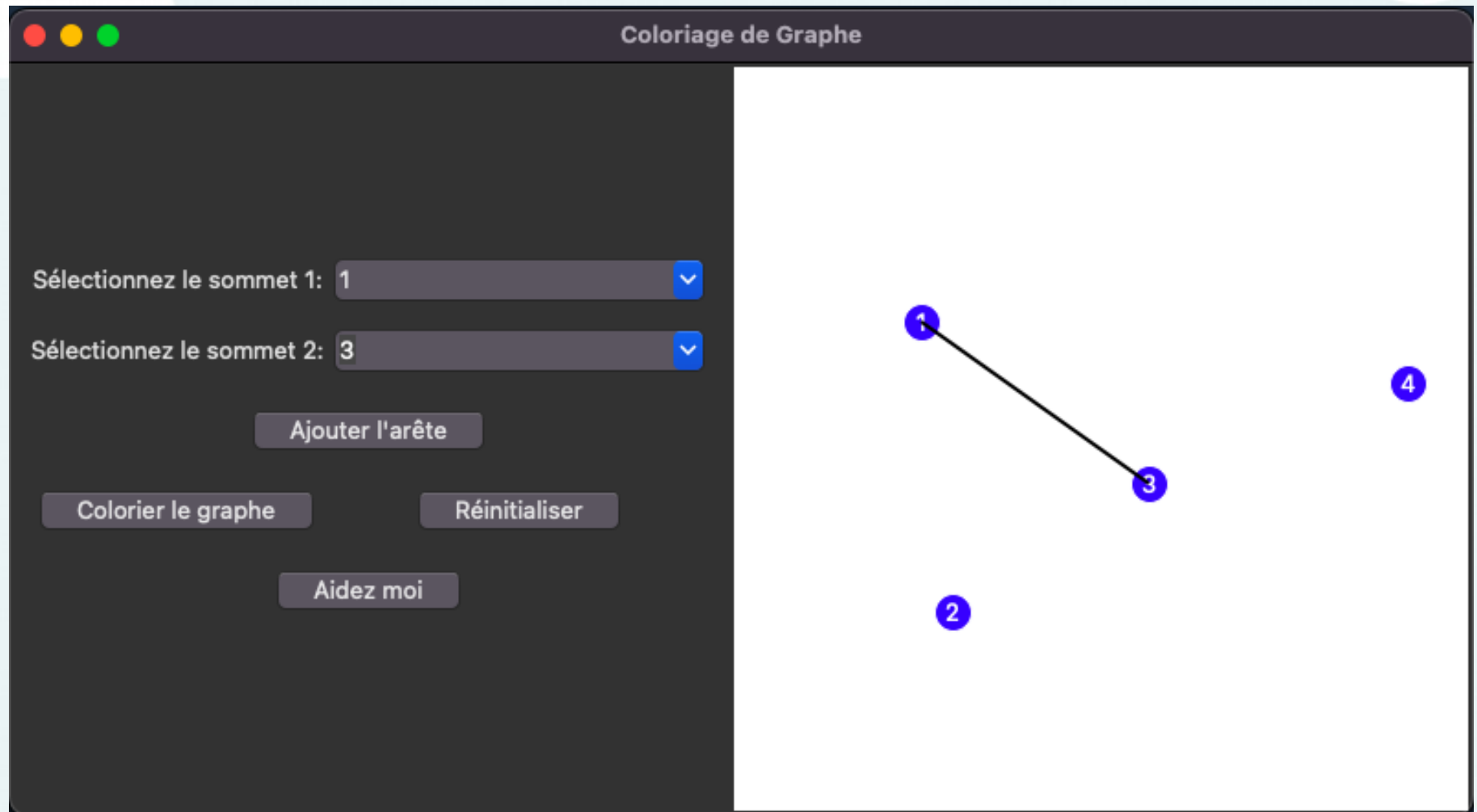
Application de coloriage de graphe



Une fois les sommets créés, on clique sur les liste déroulantes pour les relier.



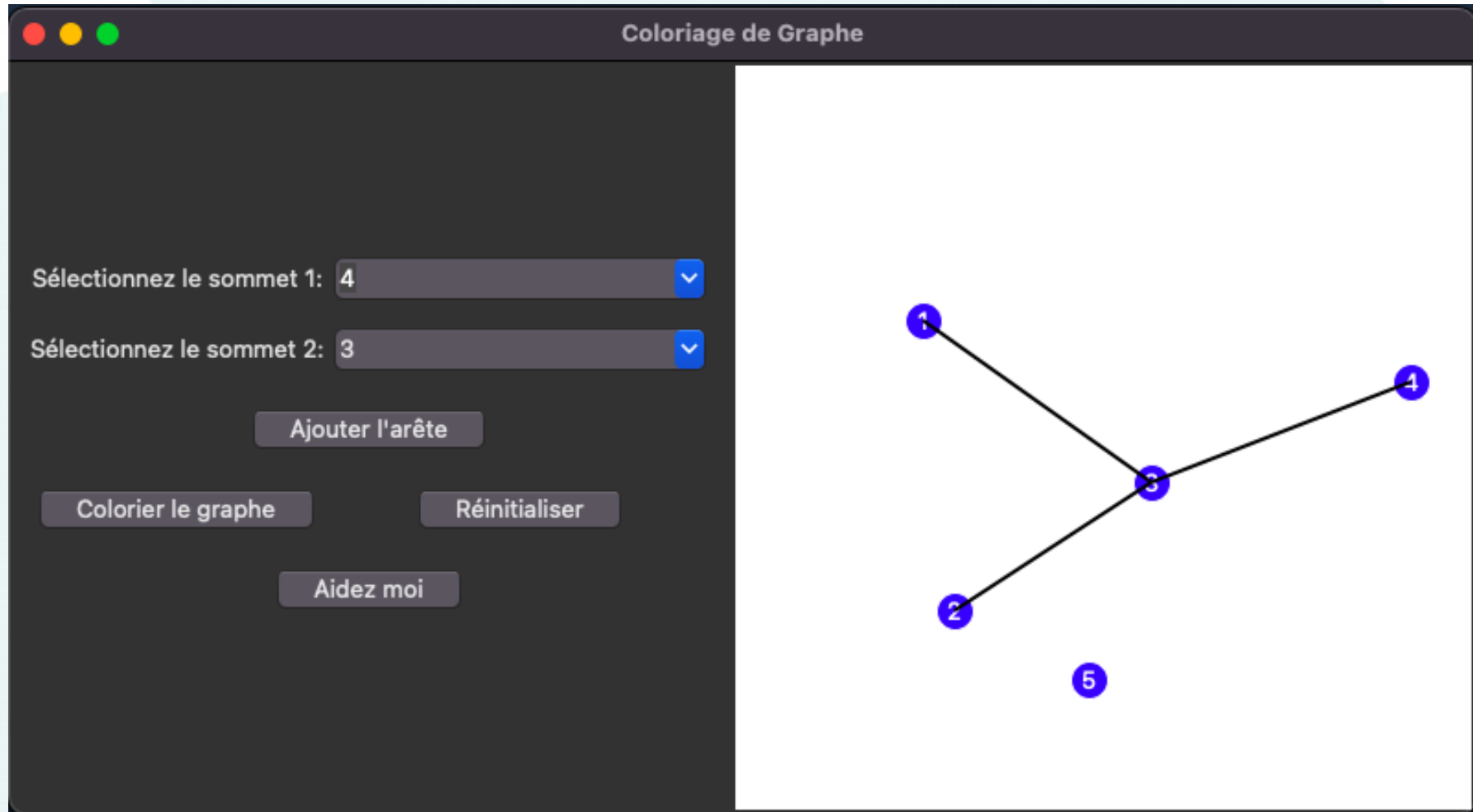
Application de coloriage de graphe



Puis on peut créer une arête en cliquant sur *Ajouter l'arête*



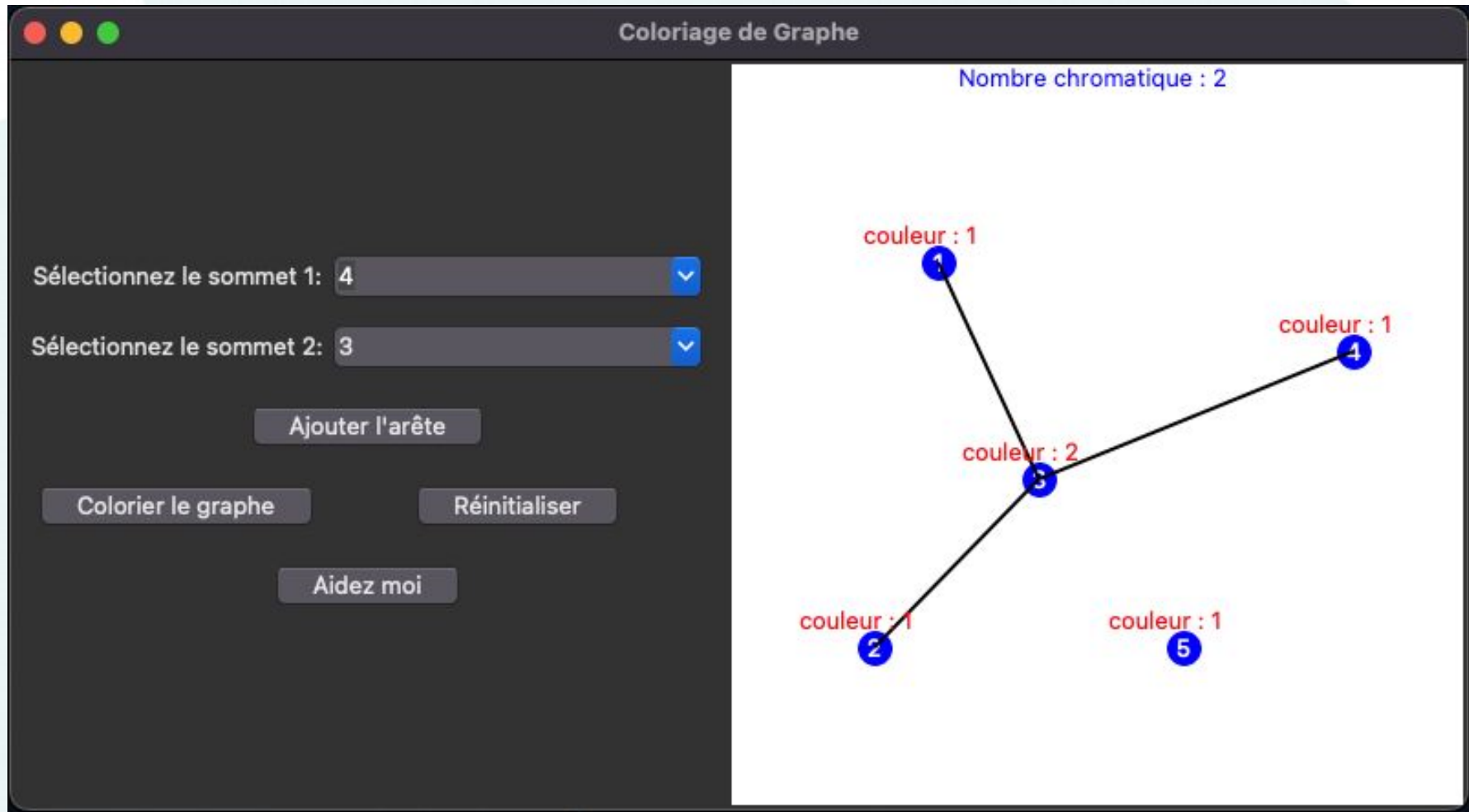
Application de coloriage de graphe



Puis on peut créer des arêtes en cliquant sur *Ajouter l'arête* successivement



Application de coloriage de graphe



Une fois le graphe créé, on peut le colorier en cliquant sur
Colorier le graphe



Application de coloriage de graphe



The screenshot shows a window titled "Coloriage de Graphe" with a dark gray sidebar on the left and a large white canvas on the right. The sidebar contains the following elements:

- Two dropdown menus: "Sélectionnez le sommet 1:" with the value "4" and "Sélectionnez le sommet 2:" with the value "3".
- A button labeled "Ajouter l'arête".
- Two buttons labeled "Colorier le graphe" and "Réinitialiser".
- A button labeled "Aidez moi".

The right-hand canvas is currently empty.

Et puis on peut ***Réinitialiser*** l'application et puis recommencer



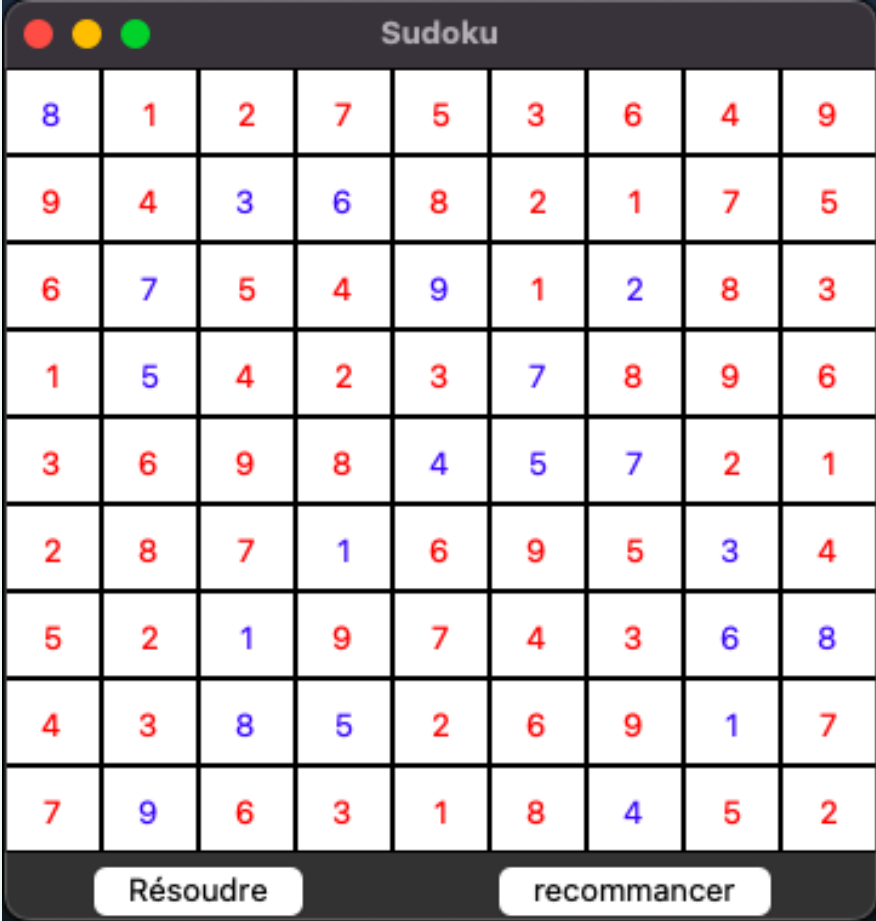
Application Du Coloriage A La Résolution Du Sudoku

Sudoku								
5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9
Résoudre			recommencer					

Sudoku								
8								
		3	6					
	7			9		2		
	5				7			
				4	5	7		
			1				3	
		1					6	8
		8	5				1	
	9					4		
Résoudre			recommencer					

L'application encapsule plusieurs **grilles de Sudoku** non résolues. A chaque fois qu'on clique sur *recommencer* : une nouvelle grille est représentée.

Application Du Coloriage A La Résolution Du Sudoku

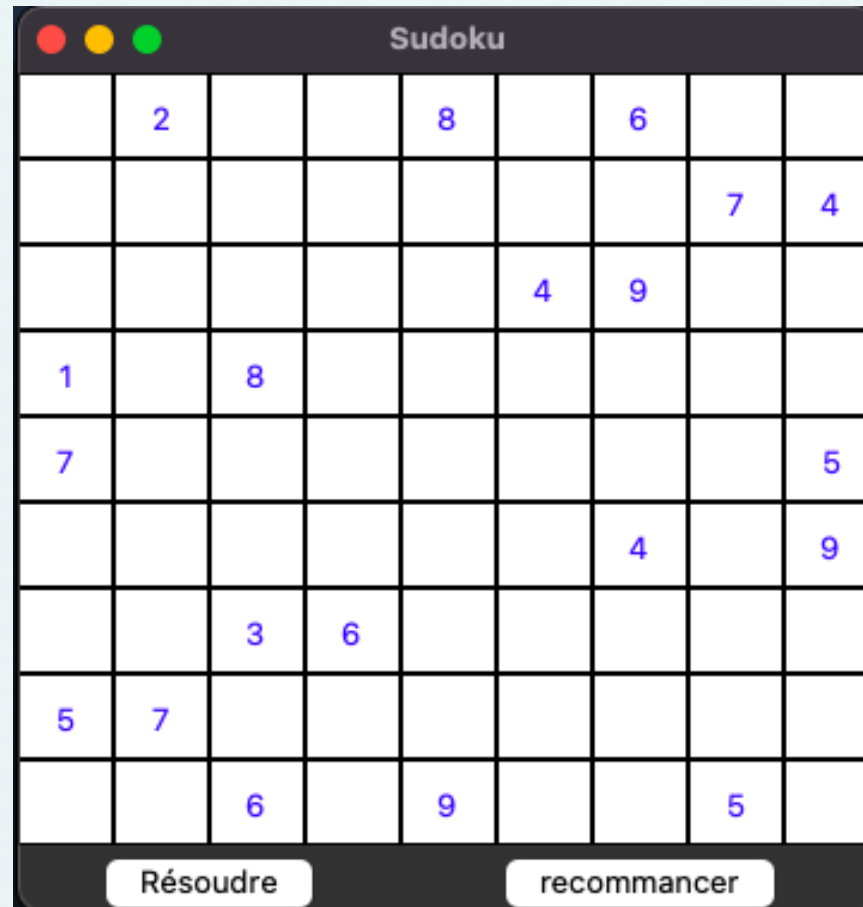


8	1	2	7	5	3	6	4	9
9	4	3	6	8	2	1	7	5
6	7	5	4	9	1	2	8	3
1	5	4	2	3	7	8	9	6
3	6	9	8	4	5	7	2	1
2	8	7	1	6	9	5	3	4
5	2	1	9	7	4	3	6	8
4	3	8	5	2	6	9	1	7
7	9	6	3	1	8	4	5	2

Pour résoudre une grille : on clique sur *Résoudre*



Application Du Coloriage A La Résolution Du Sudoku



Puis on peut à nouveau cliquer sur *recommencer* pour avoir accès à un nouvelle grille



Conclusion

- Trouver le nombre chromatique minimal d'un graphe est un problème NP-complet, ce qui signifie qu'il n'existe pas d'algorithme efficace connu qui puisse résoudre ce problème de manière optimale pour tous les graphes en un temps polynomial.
- Il existe plusieurs algorithmes et heuristiques qui peuvent être utilisés pour tenter de trouver une coloration avec un nombre chromatique proche du minimum dans un temps raisonnable : Algorithme Glouton, Algorithme de Coloration de Graphe par Backtracking, Algorithme de Welsh-Powell et Algorithme BFS.
- Le coloriage d'un graphe s'affirme comme une palette riche et polyvalente dans la boîte à outils du chercheur et du développeur informatique, ouvrant la voie à des découvertes et des innovations continues dans le monde passionnant de l'informatique graphique.

