ALGORITHMIQUE DES GRAPHES

PROJET: COLORIAGE D'UN GRAPHE ET APPLICATION A LA RÉSOLUTION DU SUDOKU

Professeur : CARLA SELMI
Etudiants : THI DIU NGHIEM
OUMAR TALIBE BA
Groupe : L3 INFO-SD

Table des Matières

Introduction

Dans le domaine des sciences informatiques, la représentation graphique des données est un concept omniprésent, offrant une perspective visuelle puissante pour analyser et résoudre divers problèmes. Dans ce contexte, l'élaboration et l'exploration de graphes statiques et dynamiques ont captivé l'attention des chercheurs et des développeurs. Ce coloriage d'un graphe, centré sur une application dédiée à la résolution du Sudoku, se propose d'explorer les dimensions visuelles et fonctionnelles de cette représentation graphique.

La première section examinera le concept de graphe statique, mettant en lumière les propriétés structurelles qui en font un outil essentiel dans la modélisation de divers systèmes. Ensuite, la deuxième section plongera dans l'univers des graphes dynamiques, dévoilant comment ces représentations évoluent au fil du temps pour refléter des changements ou des interactions dans un système donné.

Enfin, l'application spécifique de ces concepts sera explorée dans la troisième section, où nous aborderons la résolution du Sudoku. Cette célèbre énigme numérique devient ainsi le terrain d'application d'une approche graphique, mettant en œuvre les principes des graphes statiques et dynamiques pour proposer des solutions élégantes et efficaces.

Ainsi, plongeons-nous dans ce voyage graphique captivant, où le mariage entre le coloriage d'un graphe et la résolution du Sudoku ouvre la voie à des perspectives novatrices et créatives dans le monde de l'informatique et de l'analyse algorithmique.

I. Etat De L'Art

Le coloriage de graphes est un domaine d'étude important en mathématiques discrètes et en informatique théorique. Il a de nombreuses applications pratiques, notamment dans la résolution de problèmes de planification, de conception de réseaux, de cartographie, et même dans des casse-têtes comme le Sudoku.

1. Les coloriages de graphe ont été très étudiés pour plusieurs raisons

Tout d'abord, ils sont un problème mathématique fondamental qui a des applications dans de nombreux domaines différents. En théorie des graphes, ils sont utilisés pour étudier des propriétés telles que la connectivité, la coloration et la clique. En informatique, ils sont utilisés dans des domaines tels que la conception de circuits électroniques, la planification de ressources et la résolution de problèmes. En chimie, ils sont utilisés pour étudier les structures moléculaires. En biologie, ils sont utilisés pour étudier les réseaux biologiques, tels que les réseaux de neurones et les réseaux de protéines. En physique, ils sont utilisés pour étudier les systèmes physiques, tels que les systèmes quantiques et les systèmes d'énergie.

Deuxièmement, ils sont un problème difficile qui a défié les mathématiciens pendant des siècles. Le problème des quatre couleurs, qui consiste à déterminer si un graphe planaire peut être coloré avec au plus quatre couleurs, est un problème ouvert depuis plus de 150 ans. Ce problème est considéré comme l'un des sept problèmes du prix du millénaire, qui sont des problèmes mathématiques importants qui ont offert un million de dollars de récompense pour leur résolution.

Troisièmement, ils sont un problème riche et complexe qui offre de nombreuses possibilités de recherche. Les chercheurs travaillent à développer de nouveaux algorithmes pour résoudre les problèmes de coloriage de graphe, et ils explorent de nouvelles applications pour les coloriages de graphe dans des domaines tels que l'intelligence artificielle et la bioinformatique.

De plus, le problème du coloriage de graphes a également été très étudié en raison de son lien étroit avec la théorie des jeux. Les coloriages de graphes peuvent être modélisés comme des jeux où deux joueurs s'affrontent pour colorier les sommets tout en respectant la contrainte de ne pas avoir deux sommets adjacents de la même couleur. Cela ouvre la porte à des études approfondies sur les stratégies gagnantes, les situations d'équilibre, et les structures de jeux associées.

Par ailleurs, les coloriages de graphes jouent un rôle crucial dans la compréhension de la complexité algorithmique. La recherche de l'algorithme optimal pour le coloriage de graphes, en particulier dans des cas spécifiques tels que les graphes planaires, les graphes bipartis, ou d'autres sous-classes, permet de mieux comprendre les limites de la calculabilité et de la complexité algorithmique.

Enfin, l'étude des coloriages de graphes a également des implications importantes dans le domaine de la sécurité informatique. Certains problèmes de coloration de graphes sont liés à la conception de protocoles de cryptographie et à la sécurité des réseaux, où l'utilisation de couleurs peut représenter des attributs tels que l'authenticité, la confidentialité et l'intégrité des informations circulant dans un réseau.

2. Les coloriages de graphe sont utilisés pour résoudre un certain nombre de problèmes dans de nombreux domaines différents

a. Histoire et Culture :

Les coloriages de graphe ont été introduits pour la première fois par le mathématicien suisse Leonhard Euler au XVIIIe siècle. Euler a utilisé les coloriages de graphe pour résoudre le problème des sept ponts de Königsberg, un problème classique de la théorie des graphes.

Le problème des sept ponts de Königsberg peut être formulé comme suit : Est-il possible de traverser chacun des sept ponts de la ville de Königsberg une seule fois et de revenir au point de départ ?

Euler a montré que la réponse à cette question est non. Il a utilisé un argument géométrique pour montrer qu'il est impossible de colorier les sept sommets du graphe correspondant en deux couleurs de telle manière que deux sommets adjacents n'aient pas la même couleur.

Le problème des sept ponts de Königsberg est un exemple simple de problème qui peut être résolu à l'aide des coloriages de graphe. Cependant, les coloriages de graphe peuvent également être utilisés pour résoudre des problèmes plus complexes. Cette connexion avec l'histoire des mathématiques et de la résolution de problèmes pratiques en fait un domaine d'étude fascinant.

b. Problème des cartes géographiques :

Le coloriage de graphes est utilisé pour résoudre le 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. Cette application est utilisée en cartographie pour éviter les conflits de couleurs sur une carte.

c. 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.

d. 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.

e. Allocation de registres dans la compilation

Lors de la compilation de programmes informatiques, les registres du processeur doivent être attribués aux variables. Le problème du coloriage de graphes est utilisé pour déterminer quels registres peuvent être attribués simultanément sans interférences.

f. 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.

g. Résolution de problèmes logiques

Comme mentionné, le coloriage de graphes est utilisé pour résoudre des casse-têtes logiques tels que le Sudoku, où l'objectif est d'attribuer des nombres à des cases de manière à ce que chaque ligne, chaque colonne et chaque sous-grille contient tous les chiffres de 1 à 9 sans répétition.

h. VLSI Design (Conception de circuits intégrés)

Dans la conception de circuits intégrés, le coloriage de graphes est utilisé pour résoudre le problème de placement des cellules sur une puce tout en évitant les interférences.

II. Graphes Statiques : Algorithmes

1. Structures des Données Utilisées dans les 3 Algorithmes

Type GrapheM = Enregistrement

n : Entier;

M : Tableau[1..n, 1..n] de Booléen

Fin;

2. Algorithme Glouton

L'algorithme de coloration gloutonne est un algorithme simple et intuitif qui attribue des couleurs aux sommets du graphe de manière gloutonne, c'est-à-dire en choisissant la première couleur disponible.

L'algorithme peut être résumé de manière générale comme suit :

1. Colorier le premier sommet avec la première couleur :

Cette étape est effectuée dans la boucle principale de la fonction glouton. Le premier sommet du graphe est colorié avec la couleur 1.

2. Pour les n-1 sommets restants, faire ce qui suit :

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 :

Cette étape est également implémentée dans la boucle principale de la fonction glouton. Les sommets restants sont coloriés en considérant la couleur numérotée la plus basse qui n'a pas été utilisée par les sommets adjacents déjà colorés.

b. Si toutes les couleurs déjà utilisées apparaissent sur les sommets adjacents à v, attribuer une nouvelle couleur à celui-ci :

Cela est géré par la vérification dans la boucle while à l'intérieur de la fonction glouton. Si toutes les couleurs déjà utilisées apparaissent sur les sommets adjacents à un sommet donné, une nouvelle couleur est attribuée à ce sommet.

L'algorithme est implémenté dans le code fourni comme suit:

1. Initialisation:

On commence par initialiser une liste appelée *result* qui stocke les couleurs attribuées à chaque sommet. Au début, tous les sommets sont considérés comme non colorés, ce qui est représenté par la valeur -1.

On crée une liste *colors* pour marquer les couleurs déjà utilisées par les voisins d'un sommet. Les indices de cette liste représentent les couleurs, et chaque élément est initialisé à False.

2. Attribution des couleurs:

On itère à travers chaque sommet du graphe. Pour chaque sommet, on marque les couleurs déjà utilisées par ses voisins comme indisponibles en mettant à True les éléments correspondants dans la liste *colors*.

3. Recherche de la première couleur disponible:

On commence avec la couleur 1 et on incrémente jusqu'à trouver la première couleur disponible (c'est-à-dire un indice dans la liste *colors* qui est toujours à False).

4. Attribution de la couleur:

On attribue la couleur trouvée au sommet en cours.

5. Réinitialisation des couleurs:

On réinitialise les couleurs en marquant comme non utilisées les couleurs des voisins déjà colorés, pour les prochaines itérations.

6. Affichage des résultats:

Une fois tous les sommets colorés, on affiche les résultats, indiquant le numéro de chaque sommet et la couleur qui lui a été attribuée.

L'algorithme glouton cherche à minimiser le nombre de couleurs utilisées en attribuant des couleurs de manière locale. Cependant, il ne garantit pas toujours le nombre chromatique optimal (le nombre minimum de couleurs nécessaires), mais il fournit une solution rapidement. C'est un algorithme efficace pour des graphes de petite à moyenne taille, mais il peut ne pas être optimal pour des graphes plus grands ou complexes.

a. Fonction versionNo

Cette fonction prend une matrice d'adjacence M représentant un graphe orienté en paramètre et renvoie la matrice M1 représentant la version non orientée du graphe

```
Fonction versionNo(M : GrapheM) → GrapheM:
Var i, j: Entier, M1: GrapheM
Début
    n ← G.n
    M1 ← G.M
    Pour i de 1 à G.n faire
        Pour j de 1 à G.n faire
            Si G.M[i, j] = 1 alors
                M1[i, j] ← 1
                M1[j, i] ←1
    Retourner M1
FinFonction
```

Complexité en temps : $O(n^2)$ où n est le nombre de sommets dans le graphe. Les boucles imbriquées traversent tous les sommets du graphe.

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 ← -1
    colors ← 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] ≠ -1 alors
                colors[result[i]] ← True   {Marquer la couleur de i comme indisponible}
        {Trouver la première couleur disponible}
        cd ← 1
        Tant que cd ⩽ G.n et colors[cd] = True faire
            cd ← cd + 1
        result[u] ← 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] ≠ -1 alors
                colors[result[i]] ← 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
```

Complexité en temps : La complexité en temps dépend de la taille de la matrice d'adjacence, notée n (le nombre de sommets dans le graphe). Les boucles imbriquées traversent tous les sommets du graphe, donc la complexité est généralement de l'ordre de $O(n^2)$.

c. Fonction chromaticNumber

Cette fonction renvoie le nombre chromatique du graphe

```
Fonction chromaticNumber
{Entrée : colorArray: Tableau[1..n] d'Entiers représentant les couleurs attribuées à chaque sommet}
{Sortie : Nombre chromatique du graphe}
Début
    Retourner Longueur (len) de l'ensemble (set) des valeurs uniques présentes dans colorArray
FinFonction
```

Complexité en temps : La fonction set(*colorArray*) crée un ensemble contenant les valeurs uniques de *colorArray*. La création d'un ensemble a généralement une complexité linéaire par rapport à la taille de *colorArray*. Par conséquent, la complexité de cette fonction est généralement linéaire O(n).

d. Fonction plotGraph

Cette fonction est responsable de visualiser un graphe colorié

```
Fonction plotGraph
{Entrée : G: GrapheM Matrice d'adjacence, colors: Tableau[1…n] représentant les couleurs attribuées}
{Sortie : Affiche le graphe colorié}
Var V, i, j : Entier; x, y : Tableaux[1…n] de Réels; xi, yi: Réels; N : Tableau[1..n, 1..n] d'Entiers
Début
    V ← G.n
    x ← Cosinus(2 * π * Indice de sommet / V)
```

```
    y ← Sinus(2 * π * Indice de sommet / V)
    Créer une nouvelle figure de tracé de taille (8, 8) pouces
    Pour i de 0 à V - 1 faire
        Ajouter un point rouge sans ligne de connexion à la position (x[i], y[i])
    Pour chaque sommet i et sa position (xi, yi) faire
        Ajouter un texte à côté du sommet avec les informations "Sommet i\nCouleur: colors[i]"
    N ← versionNo(G)
    Pour i de 0 à V - 1 faire
        Pour j de i + 1 à V - 1 faire
            Si N[i][j] = 1 alors
                Tracer une ligne bleue connectant les sommets i et j
    Afficher le tracé final
FinFonction
```

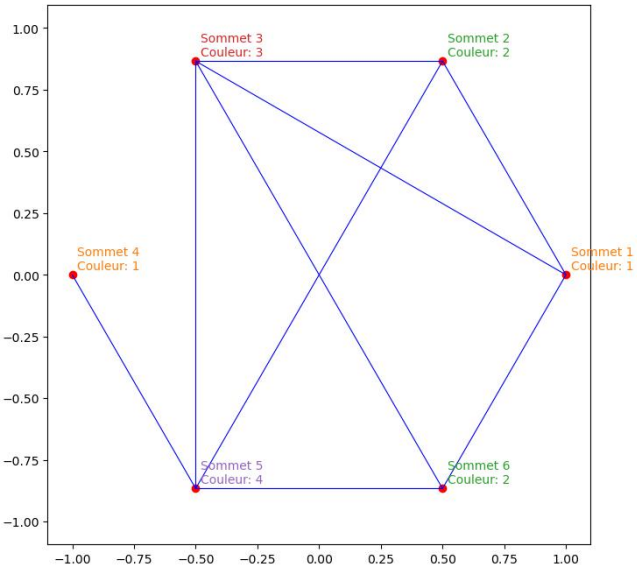
Complexité en temps : La complexité de cette fonction dépend du nombre de sommets (V). Les opérations liées à la création du tracé des sommets et des arêtes ont une complexité linéaire O(V).

e. 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
```




```
g4 = np.array([
    [0, 1, 0, 1, 0, 0, 1],
    [1, 0, 1, 1, 0, 0, 0],
    [0, 1, 0, 1, 0, 0, 0],
    [1, 1, 1, 0, 1, 0, 0],
    [0, 0, 0, 1, 0, 1, 1],
    [0, 0, 0, 0, 1, 0, 1],
    [1, 0, 0, 0, 1, 1, 0]
])
print("\nColoriage du graphe 4")
colors_g4 = glouton(versionNo(g4))
chromatic_number_g4 = chromaticNumber(colors_g4)
print("Nombre chromatique du graphe 4:", chromatic_number_g4)
plotGraph(versionNo(g4), colors_g4)
```

Résultat:

Coloriage du graphe 4

Sommet 1 ---> Couleur 1

Sommet 2 ---> Couleur 2

Sommet 3 ---> Couleur 1

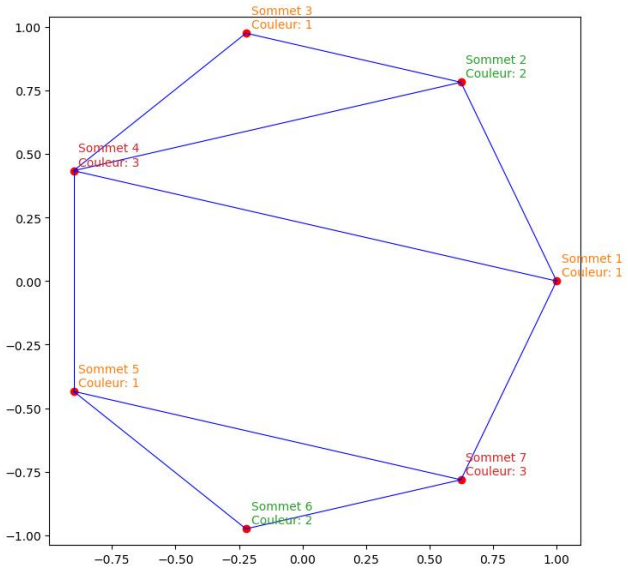
Sommet 4 ---> Couleur 3

Sommet 5 ---> Couleur 1

Sommet 6 ---> Couleur 2

Sommet 7 ---> Couleur 3

Nombre chromatique du graphe 4: 3



3. Algorithme Par Backtracking

L'algorithme de coloration de graphe avec backtracking explore de manière exhaustive les différentes combinaisons de couleurs, 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.

L'algorithme peut être décrit de manière générale comme suit :

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).

L'algorithme est implémenté dans le code fourni comme suit:

1. Initialisation :

L'algorithme commence par initialiser un tableau *colorArray* représentant les couleurs attribuées aux sommets du graphe. Au départ, toutes les couleurs sont non attribuées (0).

Ensuite, on crée une version non orientée du graphe, car l'algorithme de coloration est souvent appliqué à des graphes non orientés.

2. Exploration récursive :

L'algorithme utilise la récursivité pour explorer les différentes combinaisons de couleurs pour chaque sommet du graphe. On commence par le premier sommet et tente d'attribuer chaque couleur possible.

Pour chaque couleur, on vérifie si elle peut être attribuée de manière sûre en appelant la fonction *isSafe*.

Si une couleur est attribuée de manière sûre, on passe au sommet suivant et répète le processus récursivement.

Si à un moment donné, aucune couleur n'est possible pour un sommet particulier, l'algorithme fait marche arrière (backtracking) pour essayer une couleur différente pour le sommet précédent.

3. Critère d'arrêt :

L'algorithme continue à explorer les combinaisons de couleurs jusqu'à ce que tous les sommets aient été traités. Si tous les sommets ont été colorés avec succès, l'algorithme retourne True pour indiquer que la coloration est possible.

4. Affichage des résultats :

Une fois la coloration réussie, l'algorithme affiche des informations telles que la possibilité de coloration, le nombre chromatique du graphe, et les couleurs attribuées à chaque sommet.

On visualise le graphe colorié, où les sommets sont représentés par des cercles rouges, les arêtes par des lignes bleues, et les étiquettes indiquent le numéro du sommet et la couleur attribuée.

a. Fonction printConfiguration

Cette fonction prend un tableau *'colorArray'* qui est le tableau des couleurs attribuées aux sommets en

paramètre et affiche les couleurs attribuées à chaque sommet

Fonction printConfiguration

{Entrée : colorArray Tableau[1…n] d'Entiers représentant des couleurs attribuées aux sommets}

{Sortie : Affiche les couleurs attribuées à chaque sommet}

Var i: Entier

Début

    Afficher "Les couleurs attribuées sont les suivantes :"

    Pour i de 0 à G.n - 1 faire

        Afficher "Sommet ", i + 1, " - Couleur : ", colorArray[i]

FinFonction

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

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).

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.

d. Fonction versionNo

Cette fonction prend une matrice d'adjacence M représentant un graphe orienté en paramètre et renvoie la matrice M1 représentant la version non orientée du graphe

```
Fonction versionNo(M : GrapheM) → GrapheM:
Var i, j: Entier, M1: GrapheM
Début
    n ← G.n
    M1 ← G.M
    Pour i de 1 à G.n faire
        Pour j de 1 à G.n faire
            Si G.M[i, j] = 1 alors
                M1[i, j] ← 1
                M1[j, i] ←1
    Retourner M1
FinFonction
```

Complexité en temps : $O(n^2)$ où n est le nombre de sommets dans le graphe. Les boucles imbriquées traversent tous les sommets du graphe.

e. Fonction plotGraph

Cette fonction est responsable de visualiser un graphe colorié

Fonction plotGraph

{Entrée : G: GrapheM, m: Entier représentant le nombre maximum de couleurs}

{Sortie : Affiche le graphe colorié s'il est possible de le colorier}

```
Var V, i, j : Entier; x, y : Tableaux[1…n] de Réels; xi, yi: Réels; N : Tableau[1..n, 1..n] d'Entiers; colorArray: Tableau[1…n] d'Entiers représentant les couleurs attribuées
Début
    V ← G.n
    colorArray ← 0
    N ← versionNo(G)
    Si graphColoringBacktracking(m, colorArray, 0, N) alors
    Début
        Afficher "La coloration est possible !"
        Afficher "Le nombre chromatique du graphe est :", m
        printConfiguration(colorArray)
        Créer une nouvelle figure de tracé de taille (8, 8) pouces
        Ajuster les axes pour maintenir une échelle égale
        x ← Cosinus(2 * π * Indice de sommet / V)
        y ← Sinus(2 * π * Indice de sommet / V)
        Pour i de 0 à V - 1 faire
            Afficher un cercle rouge pour représenter le sommet i à la position (x[i], y[i])
        Pour chaque sommet i et sa position (xi, yi) faire
            Afficher un texte à côté du sommet avec les info "Sommet i\nCouleur: colorArray[i]"
        Pour i de 0 à V - 1 faire
            Pour j de i + 1 à V - 1 faire
                Si N[i][j] = 1 alors
                    Tracer une ligne bleue connectant les sommets i et j
        Afficher le graphe colorié
    Sinon
        Afficher "La coloration n'est pas possible !"
    FinSi
FinFonction
```

Complexité en temps : La complexité de cette fonction dépend du nombre de sommets (V). Les opérations liées à la création du tracé des sommets et des arêtes ont une complexité linéaire O(V).

f. 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("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

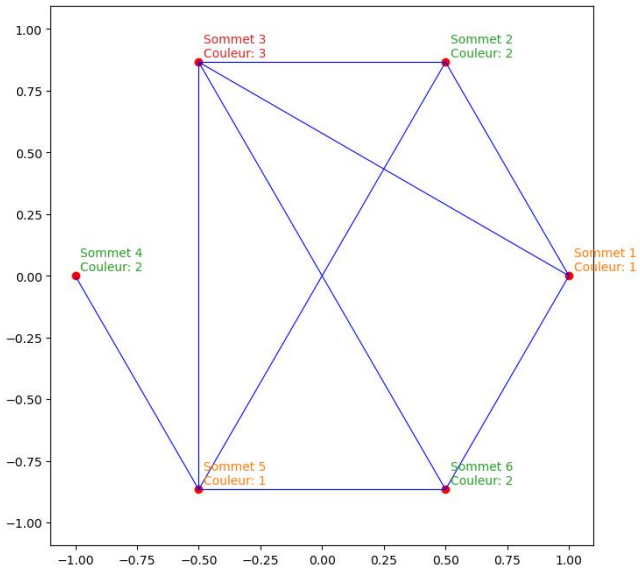
Sommet 2 - Couleur : 2

Sommet 3 - Couleur : 3

Sommet 4 - Couleur : 2

Sommet 5 - Couleur : 1

Sommet 6 - Couleur : 2




```
g4 = np.array([
    [0, 1, 0, 1, 0, 0, 1],
    [1, 0, 1, 1, 0, 0, 0],
    [0, 1, 0, 1, 0, 0, 0],
    [1, 1, 1, 0, 1, 0, 0],
    [0, 0, 0, 1, 0, 1, 1],
    [0, 0, 0, 0, 1, 0, 1],
    [1, 0, 0, 0, 1, 1, 0]
])
print("Coloriage du graphe 4")
plotGraph(versionNo(g4), m)
```

Résultat:

Coloriage du graphe 4

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

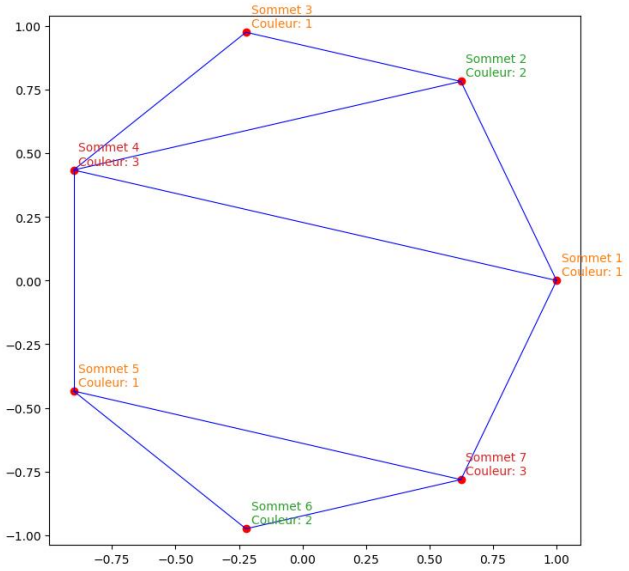
Sommet 3 - Couleur : 1

Sommet 4 - Couleur : 3

Sommet 5 - Couleur : 1

Sommet 6 - Couleur : 2

Sommet 7 - Couleur : 3



4. Algorithme De Welsh-Powell

L'objectif de l'algorithme de coloration de graphes de Welsh-Powell est de minimiser le nombre total de couleurs utilisées pour colorer le graphe de manière à ce que deux sommets adjacents n'aient jamais la même couleur. La stratégie clé est de choisir le sommet non coloré avec le maximum d'arêtes adjacentes non colorées, puis d'attribuer la couleur minimale qui n'est pas déjà utilisée par ses voisins déjà colorés. Cette approche vise à optimiser la coloration du graphe en favorisant l'utilisation de couleurs inférieures.

L'algorithme peut être résumé de manière générale comme suit :

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 la liste 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 coloré de la liste.
8. Répéter les opérations 4 à 7.
9. Continuer jusqu'à avoir coloré tous les sommets.

L'algorithme est implémenté dans le code fourni comme suit:

1. Initialisation :

On initialise une liste de couleurs attribuées à chaque sommet. Initialement, tous les sommets sont marqués comme non colorés.

2. Sélection du sommet :

On choisit le sommet non coloré avec le plus grand nombre d'arêtes adjacentes non colorées. Cela maximise la chance de trouver un sommet qui pourrait nécessiter une couleur distincte.

3. Attribution de couleurs :

On attribue au sommet choisi la couleur minimale qui n'est pas déjà utilisée par ses voisins. Cela aide à minimiser le nombre total de couleurs nécessaires.

4. Répétition :

On répète les étapes 2-3 jusqu'à ce que tous les sommets soient colorés.

5. Visualisation :

On affiche le graphe coloré où les sommets sont représentés par des cercles et les arêtes par des lignes. Les couleurs attribuées à chaque sommet sont indiquées visuellement.

6. Résultat :

Le résultat est une coloration du graphe où deux sommets adjacents n'ont jamais la même couleur, et le nombre total de couleurs utilisées est minimisé.

a. Fonction versionNo

Cette fonction prend une matrice d'adjacence M représentant un graphe orienté en paramètre et renvoie la matrice M1 représentant la version non orientée du graphe

```
Fonction versionNo(M : GrapheM) → GrapheM:
Var i, j: Entier, M1: GrapheM
Début
        n ← G.n
        M1 ← G.M
        Pour i de 1 à G.n faire
                Pour j de 1 à G.n faire
                        Si G.M[i, j] = 1 alors
                                M1[i, j] ← 1
                                M1[j, i] ←1
        Retourner M1
FinFonction
```

Complexité en temps : $O(n^2)$ où n est le nombre de sommets dans le graphe. Les boucles imbriquées traversent tous les sommets du graphe.

b. Fonction chromaticNumber

Cette fonction renvoie le nombre chromatique du graphe

```
Fonction chromaticNumber
{Entrée : colorArray: Tableau[1..n] d'Entiers représentant les couleurs attribuées à chaque sommet}
{Sortie : Nombre chromatique du graphe}
Début
        Retourner Longueur (len) de l'ensemble (set) des valeurs uniques présentes dans colorArray
FinFonction
```

Complexité en temps : La fonction set(colorArray) crée un ensemble contenant les valeurs uniques de colorArray. La création d'un ensemble a généralement une complexité linéaire par rapport à la taille de colorArray. Par conséquent, la complexité de cette fonction est généralement linéaire O(n).

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 ← G.n
color_list ← order + 1
colored_vertex ← ∅
K ← ∅
z ← ∅
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 ∉ colored_vertex alors
                    z.append(i+1)
    w ← 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])
    // 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

Complexité en temps : La complexité de cette fonction dépend du nombre de sommets (n). Les deux boucles principales de l'algorithme ont une complexité quadratique, et la fonction max utilisée pour trouver le sommet w avec le plus grand nombre d'arêtes non colorées a une complexité linéaire par rapport à order. Dans l'ensemble, la complexité est dominée par la première boucle, ce qui donne une complexité quadratique $O(n^2)$.

d. Fonction plotGraph

Cette fonction *plotGraph* prend une matrice d'adjacence G représentant un graphe orienté et un

dictionnaire colors associant à chaque sommet une couleur. Elle visualise ensuite le graphe non orienté

avec les couleurs attribuées aux sommets.

Fonction plotGraph

{Entrées : G: GrapheM Matrice d'adjacence, colors: Dictionnaire associant chaque sommet à une couleur de longueur 1 à G.n}

{Sortie : Visualisation du graphe non orienté avec les couleurs attribuées}

Var V, i, j : Entier; x, y : Tableaux[1…n] de Réels; N : Tableau[1..n, 1..n] d'Entiers, chromatic_number : Entier

Début

V ← G.n

N ← versionNo(adjacency_matrix)

x ← Cosinus(2 * π * Indice de sommet / V)

y ← Sinus(2 * π * Indice de sommet / V)

Créer une nouvelle figure pour le tracé

Ajuster les axes pour maintenir une échelle égale

Pour i de 0 à V-1 faire

Pour j de i+1 à V-1 faire

Si N[i][j] = 1 alors

Afficher une ligne reliant les sommets x[i], y[i] à x[j], y[j] en bleu

Tracer des points aux coordonnées (x, y) avec la couleur rouge

Pour i de 0 à V-1 faire

Afficher le texte "Sommet i+1 - Couleur: colors[i+1]" à la position x[i]+0.02, y[i]+0.02

Calculer le nombre chromatique et l'afficher

Afficher les couleurs attribuées à chaque sommet

Pour chaque nœud, couleur ∈ colors faire

Afficher "Sommet nœud : Couleur couleur"

Afficher le graphe colorié

FinFonction

Complexité en temps : En termes de structures de données et de primitives, la fonction utilise des tableaux numpy pour la représentation du graphe (G et N). La complexité temporelle de cette fonction dépend des opérations utilisées par matplotlib, mais elle est généralement linéaire O(n).

e. 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("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

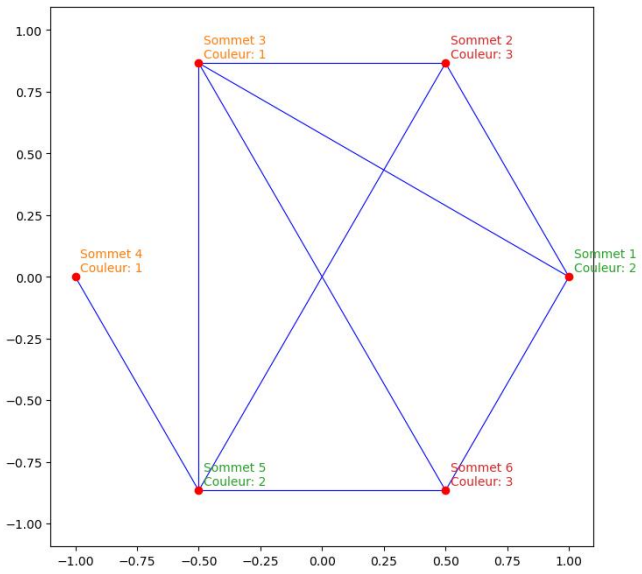
Sommet 2 : Couleur 3

Sommet 3 : Couleur 1

Sommet 4 : Couleur 1

Sommet 5 : Couleur 2

Sommet 6 : Couleur 3




```
g4 = np.array([
   [0, 1, 0, 1, 0, 0, 1],
   [1, 0, 1, 1, 0, 0, 0],
   [0, 1, 0, 1, 0, 0, 0],
   [1, 1, 1, 0, 1, 0, 0],
```

```
    [0, 0, 0, 1, 0, 1, 1],
    [0, 0, 0, 0, 1, 0, 1],
    [1, 0, 0, 0, 1, 1, 0]
])
print("Coloriage du graphe 4")
coloring4 = welsh_powell_algo(versionNo(g4))
plotGraph(versionNo(g4), coloring4)
```

Résultat:

Coloriage du graphe 4

Le nombre chromatique du graphe est : 3

Couleurs assignées à chaque sommet :

Sommet 1 : Couleur 2

Sommet 2 : Couleur 3

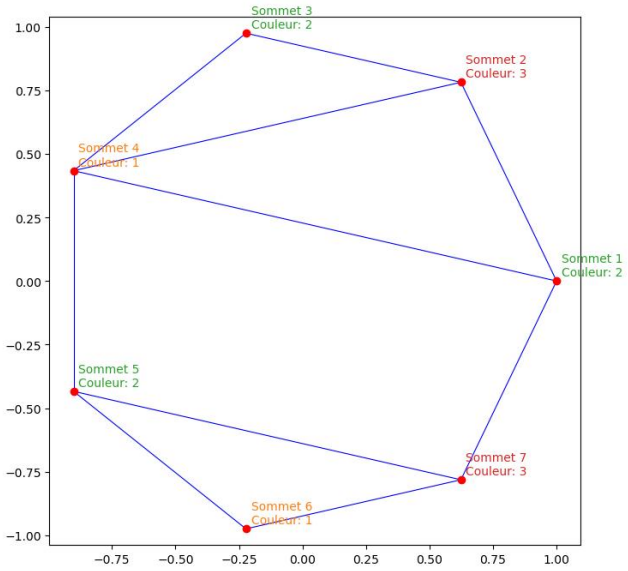
Sommet 3 : Couleur 2

Sommet 4 : Couleur 1

Sommet 5 : Couleur 2

Sommet 6 : Couleur 1

Sommet 7 : Couleur 3



III. Graphes Dynamiques: Algorithmes

1. Algorithme BFS (Breadth-First Search - Approche de Recherche en Largeur)

L'algorithme utilisé implémente une méthode de coloration de graphes dynamiques basée sur une approche de recherche en largeur (BFS). Il explore les composantes connectées du graphe, attribuant des couleurs de manière dynamique de manière itérative tout en évitant les conflits de couleurs entre les voisins. C'est un processus itératif où les sommets du graphe sont visités et coloriés au fur et à mesure de leur découverte.

L'algorithme BFS peut être généralement décrit comme suit.

1. Initialisation :

Création d'une file d'attente (queue) et ajout du sommet initial.

Marquage du sommet initial comme visité.

2. Boucle Principale :

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

3. Traitement du Sommet Actuel :

Retrait du premier sommet de la file d'attente.

Traitement du sommet (coloriage du sommet).

4. Exploration des Voisins :

Pour chaque voisin du sommet actuel qui n'a pas encore été visité, marquage comme visité et ajout à la file d'attente.

5. Terminaison :

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

L'algorithme est implémenté dans le code fourni comme suit:

1. Initialisation :

Création d'une file d'attente (queue) avec le sommet initial à explorer.

Création d'un ensemble vide (used_colors) pour suivre les couleurs déjà utilisées dans la composante connectée.

2. Parcours de la File d'Attente (BFS) :

Utilisation d'une boucle qui continue tant que la file d'attente n'est pas vide.

À chaque itération, extraction du premier sommet de la file pour l'examiner.

3. Identification des Couleurs Disponibles :

Création d'un ensemble (available_colors) contenant toutes les couleurs de 1 à N, où N est le nombre total de sommets dans le graphe.

4. Exclusion des Couleurs Utilisées par les Voisins :

Parcours des arêtes associées au sommet actuel.

Pour chaque arête, identification du sommet adjacent et exclusion de sa couleur de l'ensemble des couleurs disponibles.

5. Attribution d'une Nouvelle Couleur ou Utilisation d'une Couleur Disponible :

Si des couleurs sont disponibles, attribution de la plus petite couleur disponible au sommet actuel.

Si aucune couleur n'est disponible, attribution d'une nouvelle couleur.

6. Ajout des Voisins Non Colorés à la File d'Attente :

Parcours des arêtes associées au sommet actuel.

Ajout des sommets adjacents sans couleur attribuée à la file d'attente.

7. Gestion des Nouvelles Arêtes Ajoutées :

Si de nouvelles arêtes ont été ajoutées, s'assurer que la couleur du sommet associée à la nouvelle arête est différente.

8. Réinitialisation et Mise à Jour :

Réinitialisation de l'ensemble des arêtes ajoutées.

Mise à jour du nombre chromatique du graphe après la coloration de la composante connectée.

9. Mise à jour après modification :

Après toute modification du graphe, les couleurs sont mises à jour en appelant une fonction dédiée. Cette fonction prend en charge la mise à jour des couleurs des composantes connectées affectées par la modification.

10. Affichage du graphe coloré :

Les informations sur le graphe coloré, y compris les sommets, les arêtes et les couleurs, peuvent être affichées pour visualiser le résultat.

11. Calcul du nombre chromatique :

Le nombre chromatique du graphe est calculé, représentant le nombre minimal de couleurs nécessaires pour colorer le graphe de manière cohérente.

a. Structures des Données Utilisées de la Classe *DynamicGraph*

Ce graphe a des attributs suivants pour stocker les informations sur le graphe:

{Attributs :

vertices: Dictionnaire{Entier, Ensemble[Entier]} (sommet → ensemble d'arêtes associées);

edges: Ensemble{Frozenset[Entier]} (ensemble d'arêtes sous forme de frozenset);

colors: Dictionnaire{Entier, Entier ou None} (sommet → couleur attribuée, None sinon);

edges_added: Ensemble{Frozenset[Entier]} (ensemble d'arêtes sous forme de frozenset)}

Méthode __init__()

Cette méthode initialise des attributs

Début

self.vertices ← ∅ # dictionnaire

self.edges ← ∅ # ensemble

self.colors ← ∅ # dictionnaire

self.edges_added ← ∅

FinMéthode

Méthode add_vertex

Cette méthode ajoute un sommet au graphe

{Entrée : vertex: Entier représentant le sommet à ajouter}

{Sortie : Aucune}

Début

Si vertex ∉ self.vertices alors

self.vertices[vertex] ← ∅

self.colors[vertex] ← None

FinMéthode

Complexité en temps : La complexité temporelle de cette méthode dépend de l'opération de vérification d'appartenance (vertex not in self.vertices), qui, dans le pire des cas, est linéaire par rapport à la taille du dictionnaire vertices O(n), où n est la taille du dictionnaire vertices.

Méthode add_edge

Cette méthode ajoute une arête au graphe

{Entrée : edge: Ensemble d'Entiers représentant l'arête à ajouter}

{Sortie : Aucune}

Var vertex: Entier

Début

self.edges.add(frozenset(edge))

Pour chaque vertex ∈ edge faire

self.vertices[vertex].add(frozenset(edge))

FinMéthode

Complexité en temps: La complexité temporelle de cette méthode dépend du nombre de sommets dans l'arête et de la taille de l'ensemble edges. Si le nombre de sommets dans l'arête est k, la complexité serait généralement notée comme O(k).

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 ← len(self.vertices)

Pour chaque vertex ∈ 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).

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 ← deque([start_vertex])

used_colors ← ∅

Tant que queue ≠ ∅ faire

current_vertex ← queue.popleft()

available_colors ← Ensemble d'Entier de 1 à len(self.vertices)

Pour chaque edge ∈ self.vertices[current_vertex] faire

adjacent_vertex ← Sommet ∈ edge sauf current_vertex

Si adjacent_vertex ∈ self.colors et self.colors[adjacent_vertex] ≠ None alors

Retirer self.colors[adjacent_vertex] de available_colors

```
Si available_colors ≠ ∅  alors
    Si current_vertex ∈ self.edges_added alors
        Retirer self.colors[current_vertex] de available_colors
    color ← Minimum(available_colors)
    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).

d. Méthode update_graph_after_modification

Cette méthode met à jour les couleurs du graphe après une modification. Elle prend en charge deux

paramètres optionnels : modified_vertex et modified_edge. Ces paramètres indiquent quel sommet ou

quelle arête a été modifié.

Méthode update_graph_after_modification

```
{Entrée : modified_vertex: Entier représentant le sommet optionnellement modifié (par défaut None); modified_edge: Ensemble d'Entiers représentant les sommets optionnellement modifié (par défaut None)}
{Sortie : Aucune}
Var vertex: Entier
Début
    Si modified_vertex ≠ None alors
        color_connected_component(modified_vertex)
```

Si modified_edge ≠ None alors

Pour chaque vertex ∈ modified_edge faire

color_connected_component(vertex)

FinMéthode

Complexité en temps: La complexité temporelle dépend de la taille de la composante connectée modifiée (dans le cas de modified_vertex) ou de la taille de la composante connectée associée aux sommets modifiés (dans le cas de modified_edge). Alors, la complexité serait généralement O(n), où n est le nombre total de sommets dans le graphe.

e. Méthode display_colored_graph

Cette méthode est responsable de l'affichage des informations relatives au graphe colorié

Méthode display_colored_graph

{Entrée : Aucune}

{Sortie : Aucune}

Var Aucune

Début

Afficher "\nColored Dynamic Graph:"

Afficher "Vertices:", self.vertices # Affiche les sommets et leurs ensembles d'arêtes associés

Afficher "Edges:", self.edges # Affiche l'ensemble des arêtes du graphe

Afficher "Colors:", self.colors # Affiche les couleurs attribuées à chaque sommet du graphe

FinMéthode

Complexité en temps: La complexité temporelle de cette méthode est généralement linéaire par rapport à la taille du graphe, donc elle serait de l'ordre de O(V + E), où V est le nombre de sommets et E est le nombre d'arêtes.

f. Méthode chromatic_number

Cette méthode calcule le nombre chromatique du graphe

Méthode chromatic_number

{Entrée : Aucune}

{Sortie : Entier représentant le nombre chromatique du graphe}

Var Aucune

Début

Appeler color_graph # attribuer des couleurs aux sommets du graphe si ce n'est pas déjà fait

Retourner Longueur(EnsembleValeurs(self.colors)) # Retourne le nombre chromatique du graphe

FinMéthode

Complexité en temps: La complexité en temps totale de la méthode chromatic_number serait généralement dominée par la complexité de la méthode color_graph. Si tous les sommets nécessitent une nouvelle couleur, la complexité pourrait être proportionnelle au nombre total de sommets et d'arêtes du graphe, soit O(V + E), où V est le nombre de sommets et E est le nombre d'arêtes.

g. Méthode plotGraph

Cette méthode trace le graphe dynamique avec des couleurs attribuées aux sommets

Méthode plotGraph

{Entrée : Aucune}

{Sortie : Aucune}

Var vertices: Dictionnaire d'Entiers; edges: Ensemble d'Arêtes (Entiers); colors: Dictionnaire d'Entiers; V, i, j: Entier; pos: Dictionnaire de Réels

Début

vertices ← self.vertices

edges ← self.edges

colors ← self.colors

V ← len(vertices)

pos ← nx.circular_layout(nx.Graph(edges))

Créer une figure de taille (8, 8)

Définir l'axe en tant qu'égal

Trace les arêtes du graphe en utilisant networkx.draw

valid_nodes ← [v pour v ∈ vertices.keys() si v ∈ pos]

Dessiner uniquement les nœuds qui ont des positions calculées

Pour chaque i de 1 à V faire

Si i ∈ pos alors

Afficher le numéro du sommet et la couleur attribuée, le cas échéant

Afficher la figure

FinMéthode

Complexité en temps: La complexité temporelle dépend principalement de la taille du graphe, c'est-à-dire du nombre de sommets et d'arêtes, et peut être influencée par la complexité de la fonction nx.circular_layout. Par conséquent, la complexité serait généralement O(V + E), où V est le nombre de sommets et E est le nombre d'arêtes.

h. Exemple

```
# Exemple de Graphe 1
dynamic_graph1 = DynamicGraph()


# Ajouter des sommets en utilisant input()
num_vertices = int(input("Entrer le nombre de sommets pour le graphe 1: "))
for i in range(1, num_vertices + 1):
    dynamic_graph1.add_vertex(i)


# Ajouter des arêtes en utilisant input()
num_edges = int(input("Entrer le nombre d'arêtes pour le graphe 1: "))
for _ in range(num_edges):
    while True:
        try:
            edge_input = input("Entrer une arête (format: v1 v2): ")
            edge_vertices = list(map(int, edge_input.split()))
            if all(v in dynamic_graph1.vertices for v in edge_vertices):
```

```
            dynamic_graph1.add_edge(set(edge_vertices))
            break
        else:
            print("Sommets ou Arêtes invalides. Veuillez saisir des sommets qui existent dans le graphe.")
    except ValueError:
        print("Données invalides. Veuillez saisir des sommet en tant qu'entiers séparés par un espace.")

# Afficher le graphe coloré 1
dynamic_graph1.color_graph()
dynamic_graph1.display_colored_graph()
print("Nombre chromatique du graphe 1: ", dynamic_graph1.chromatic_number())
dynamic_graph1.plotGraph()  # Affiche le graphe coloré 1
```

Résultat:

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 4

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

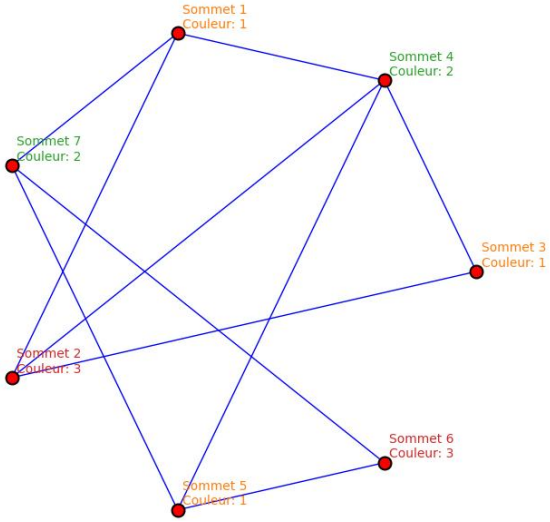
Colored Dynamic Graph:

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

Nombre chromatique du graphe 1: 3




```
# Exemple de Graphe 2
dynamic_graph2 = DynamicGraph()

# Ajouter des sommets en utilisant input()
num_vertices = int(input("Entrer le nombre de sommets pour le graphe 2: "))
for i in range(1, num_vertices + 1):
    dynamic_graph2.add_vertex(i)

# Ajouter des arêtes en utilisant input()
num_edges = int(input("Entrer le nombre d'arêtes pour le graphe 2: "))
for _ in range(num_edges):
    while True:
        try:
            edge_input = input("Entrer une arête (format: v1 v2): ")
            edge_vertices = list(map(int, edge_input.split()))
            if all(v in dynamic_graph2.vertices for v in edge_vertices):
                dynamic_graph2.add_edge(set(edge_vertices))
                break
            else:
                print("Sommets ou Arêtes invalides. Veuillez saisir des sommets qui existent dans le graphe.")
        except ValueError:
            print("Données invalides. Veuillez saisir des sommet en tant qu'entiers séparés par un espace.")

# Afficher le graphe coloré 2
dynamic_graph2.color_graph()
dynamic_graph2.display_colored_graph()
print("Nombre chromatique du graphe 1: ", dynamic_graph2.chromatic_number())
dynamic_graph2.plotGraph()  # Affiche le graphe coloré 2
```

Résultat:

Entrer le nombre de sommets pour le graphe 2: 6

Entrer le nombre d'arêtes pour le graphe 2: 10

Entrer une arête (format: v1 v2): 1 2

Entrer une arête (format: v1 v2): 1 3

Entrer une arête (format: v1 v2): 1 6

Entrer une arête (format: v1 v2): 2 3

Entrer une arête (format: v1 v2): 2 5

Entrer une arête (format: v1 v2): 3 6

Entrer une arête (format: v1 v2): 3 5

Entrer une arête (format: v1 v2): 4 5

Entrer une arête (format: v1 v2): 5 6

Entrer une arête (format: v1 v2): 6 4

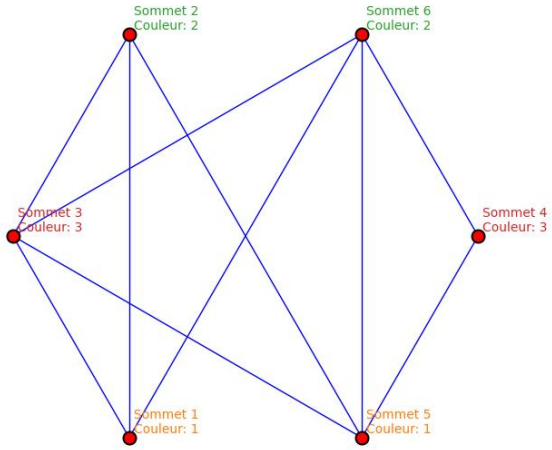
Colored Dynamic Graph:

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

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

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

Nombre chromatique du graphe 1: 3



2. Algorithme Par Backtracking

L'algorithme de coloration de graphe par rétrogradation (backtracking) utilisé attribue des couleurs aux sommets d'un graphe dynamique de manière récursive. Il explore 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.

L'algorithme peut être généralement décrit comme suit.

1. Initialisation :

L'algorithme commence par 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 :

L'algorithme attribue des couleurs aux sommets un par un, en utilisant une approche récursive. Pour chaque sommet, il explore toutes les couleurs possibles.

3. Validation de la Couleur :

Avant d'attribuer une couleur à un sommet, l'algorithme vérifie 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, l'algorithme rétrograde en revenant en arrière pour essayer une autre couleur. Cela se fait de manière récursive.

4. Réussite ou Échec :

L'algorithme répète 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.

5. Affichage du Résultat :

Une fois la coloration réussie (ou échouée), le résultat est affiché, montrant les couleurs attribuées à chaque sommet, les arêtes du graphe, et éventuellement le nombre chromatique du graphe.

a. Structures des Données Utilisées de la Classe DynamicGraph

Ce graphe a des attributs suivants pour stocker les informations sur le graphe:

{Attributs :

vertices: Dictionnaire{Entier, Ensemble[Entier]} (sommet → ensemble d'arêtes associées);

edges: Ensemble{Frozenset[Entier]} (ensemble d'arêtes sous forme de frozenset);

colors: Dictionnaire{Entier, Entier ou None} (sommet → couleur attribuée, None sinon);

deleted_vertices: Tableau d'Entier (sommet → liste de sommets supprimés)}

Méthode __init__()

Cette méthode initialise des attributs

Début

self.vertices ← ∅ # dictionnaire

self.edges ← ∅ # ensemble

self.colors ← ∅ # dictionnaire

self.deleted_vertices ← ∅ # tableau

FinMéthode

Méthode add_vertex

Cette méthode ajoute un sommet au graphe

{Entrée : vertex: Entier représentant le sommet à ajouter}

{Sortie : Aucune}

Début

Si vertex ∉ self.vertices alors

self.vertices[vertex] ← ∅

self.colors[vertex] ← None

FinMéthode

Complexité en temps : La complexité temporelle de cette méthode dépend de l'opération de vérification d'appartenance (vertex not in self.vertices), qui, dans le pire des cas, est linéaire par rapport à la taille du dictionnaire vertices O(n), où n est la taille du dictionnaire vertices.

Méthode add_edge

Cette méthode ajoute une arête au graphe

{Entrée : edge: Ensemble d'Entiers représentant l'arête à ajouter}

{Sortie : Aucune}

Var vertex: Entier

Début

self.edges.add(frozenset(edge))

Pour chaque vertex ∈ edge faire

self.vertices[vertex].add(frozenset(edge))

FinMéthode

Complexité en temps: La complexité temporelle de cette méthode dépend du nombre de sommets dans l'arête et de la taille de l'ensemble edges. Si le nombre de sommets dans l'arête est k, la complexité serait généralement notée comme O(k).

b. Méthode display_colored_graph

Cette méthode est responsable de l'affichage des informations relatives au graphe colorié

Méthode display_colored_graph

{Entrée : Aucune (utilise les attributs de l'objet DynamicGraph)}

{Sortie : Aucune (affiche les informations à la console)}

Var valid_edges: Ensemble{Frozenset[Entier]}

Début

Afficher("\nColored Dynamic Graph:")

Afficher("Vertices:", self.vertices)

valid_edges = {edge for edge in self.edges if all(v in self.vertices for v in edge)}

Afficher("Edges:", valid_edges)

Afficher("Colors:", self.colors)

Si ∀ éléments ∈ self.colors.values ≠ None alors

Afficher("Chromatic Number:", len(EnsembleUnique(self.colors.values())))

FinMéthode

Complexité en temps: La complexité en temps de cette fonction dépend des opérations utilisées pour afficher les informations, mais en général, elle est linéaire par rapport au nombre de sommets et d'arêtes dans le graphe O(V+E) où V est le nombre de sommets et E est le nombre d'arêtes.

c. Méthode chromatic_number

Cette méthode calcule le nombre chromatique du graphe

Méthode chromatic_number

{Entrée : Aucune (utilise les attributs de l'objet DynamicGraph)}

{Sortie : Entier (le nombre chromatique du graphe)}

Début

Appeler color_graph_backtracking() # attribue les couleurs aux sommets du graphe

Retourner Longueur(EnsembleUnique(self.colors.values()))

FinMéthode

Complexité en temps: La complexité de la méthode chromatic_number dépend principalement de la complexité de la méthode color_graph_backtracking qu'elle appelle. La complexité de cet algorithme de backtracking est exponentielle et est généralement $O(m^n)$, où m est le nombre maximal de couleurs disponibles et n est le nombre de sommets. Alors, la complexité de la méthode chromatic_number est dominée par la complexité de color_graph_backtracking et est exponentielle $O(m^n)$ dans le pire des cas.

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 vertex ∉ self.vertices ou vertex ∈ self.deleted_vertices alors

Retourner False # la couleur n'est pas valide

FinSi

Pour chaque edge ∈ self.vertices[vertex] faire

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

adjacent_vertex ← PremièreIntersection(Ensemble(edge), {vertex})

Si (

adjacent_vertex ∈ self.colors

et self.colors[adjacent_vertex] = color

et adjacent_vertex ∉ 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 la méthode is_valid_color dépend du nombre d'arêtes associées au sommet vertex et du nombre de sommets adjacents à vertex dans ces arêtes. Notons d le degré du sommet vertex, c'est-à-dire le nombre d'arêtes associées à ce sommet. En moyenne, la complexité serait linéaire par rapport au degré du sommet O(d). Dans le pire des cas, lorsque le graphe est dense, c'est-à-dire lorsque chaque sommet est relié à tous les autres sommets, d peut être proche du nombre total de sommets n, et la complexité peut être considérée comme O(n).

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. Supposons que le nombre de sommets dans le graphe soit n et que le nombre maximal de couleurs possibles soit m.

La fonction commence par calculer m, le nombre total de sommets dans le graphe. Ensuite, elle utilise une boucle pour chaque sommet, explorant toutes les couleurs possibles (de 1 à m).

À chaque itération de la boucle, elle appelle récursivement la fonction pour le sommet suivant (vertex+1).

La fonction s'arrête si tous les sommets sont colorés (c'est-à-dire, si vertex > m).

Pour chaque couleur, la fonction explore récursivement toutes les possibilités, en essayant chaque couleur pour le sommet actuel.

Lorsqu'une couleur est attribuée à un sommet, la fonction passe au sommet suivant. Si la coloration réussit pour tous les sommets suivants, la fonction renvoie True. Sinon, elle rétrograde (backtrack) en annulant la couleur attribuée au sommet actuel et continue à explorer d'autres couleurs.

La fonction utilise également la méthode is_valid_color pour vérifier si attribuer une couleur à un sommet est valide.

La complexité en temps de cette fonction est exponentielle dans le pire des cas, car elle explore toutes les combinaisons possibles de couleurs pour chaque sommet. 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.

f. Méthode plotGraph

Cette méthode visualise le graphe colorié

Méthode plotGraph

{Entrée : Aucune}

{Sortie : Aucune}

```
Var vertices: Dictionnaire{Entier, Ensemble[Entier]}; edges: Ensemble{Frozenset[Entier]}; colors: Dictionnaire{Entier, Entier ou None}; V, i, vertex, color, v1, v2: Entier; Edge: Frozenset[Entier]; pos: Dictionnaire{Entier, Ensemble[Réels]} (associant à chaque sommet une paire de coordonnées polaires)
Début
    vertices ← self.vertices
    edges ← self.edges
    colors ← self.colors
    V ← Longueur(vertices)
    pos = {vertex: (cos(2 * pi * i / V), sin(2 * pi * i / V)) pour chaque i, vertex ∈ Enumérer(vertices)}
    CréerFigure(figsize=(8, 8))
    Axe('equal')
    Pour chaque edge ∈ edges faire
        v1, v2 ← edge
        Tracer([pos[v1][0], pos[v2][0]], [pos[v1][1], pos[v2][1]], couleur='blue', épaisseur=0.8)
    FinPour
    Pour chaque vertex, color ∈ items(colors) faire
        Dispersion(pos[vertex][0], pos[vertex][1], couleur='red', bordures='noir', épaisseur=0.8,
        taille=100)
        Texte(pos[vertex][0] + 0.02, pos[vertex][1] + 0.02,
        "Sommet {vertex}\nCouleur: {color si color ≠ None sinon 'N/A'}",
        taillePolice=10, couleur=f'C{color si color ≠ None sinon "N/A"}')
    FinPour
    Affiche la figure
FinMéthode
```

Complexité en temps: La complexité en temps de cette fonction dépend principalement du nombre de sommets (n) dans le graphe, car le traitement des arêtes et des couleurs a une complexité linéaire par rapport au nombre de sommets O(n).

g. Exemple

```
# Exemple de graphe 1
dynamic_graph1 = DynamicGraph()


# Ajoute des sommets en utilisant input()
num_vertices = int(input("Enter the number of vertices for graph 1: "))
for i in range(1, num_vertices + 1):
    dynamic_graph1.add_vertex(i)


# Ajoutedes arêtes en utilisant input()
num_edges = int(input("Enter the number of edges for graph 1: "))
for _ in range(num_edges):
    while True:
```

```
        try:
            edge_input = input("Enter an edge (format: 'v1 v2'): ")
            edge_vertices = list(map(int, edge_input.split()))
            if all(v in dynamic_graph1.vertices for v in edge_vertices):
                dynamic_graph1.add_edge((edge_vertices[0], edge_vertices[1]))
                break
            else:
                print("Invalid edge vertices. Please enter vertices that exist in the graph.")
        except ValueError:
            print("Invalid input. Please enter vertices as space-separated integers.")

    # Colorier le graphe en utilisant backtracking
    dynamic_graph1.color_graph_backtracking()
    dynamic_graph1.display_colored_graph()

    # Visualise le graphe colorié
    dynamic_graph1.plotGraph()
```

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 3

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

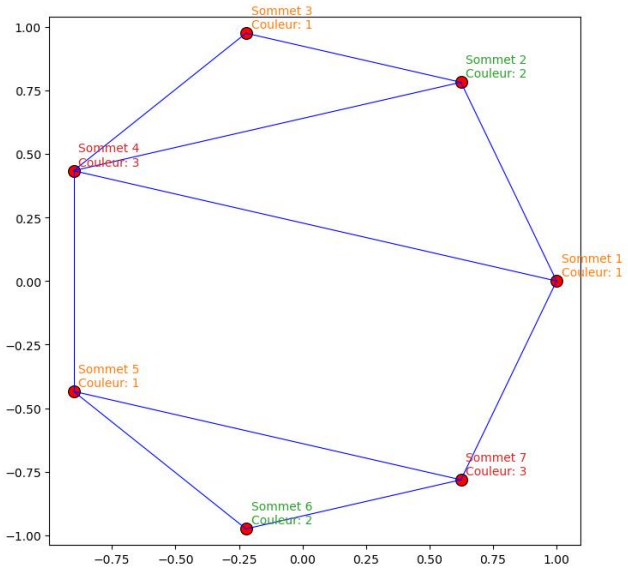
Colored Dynamic Graph:

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



IV. Application Du Coloriage A La Résolution Du Sudoku

A présent nous allons nous immerger dans le monde des applications pratiques qui intègrent les concepts de coloriage de graphe et de résolution du Sudoku. Ces deux parties apportent une dimension concrète à notre étude, démontrant comment les principes théoriques des graphes peuvent être traduits en applications fonctionnelles.

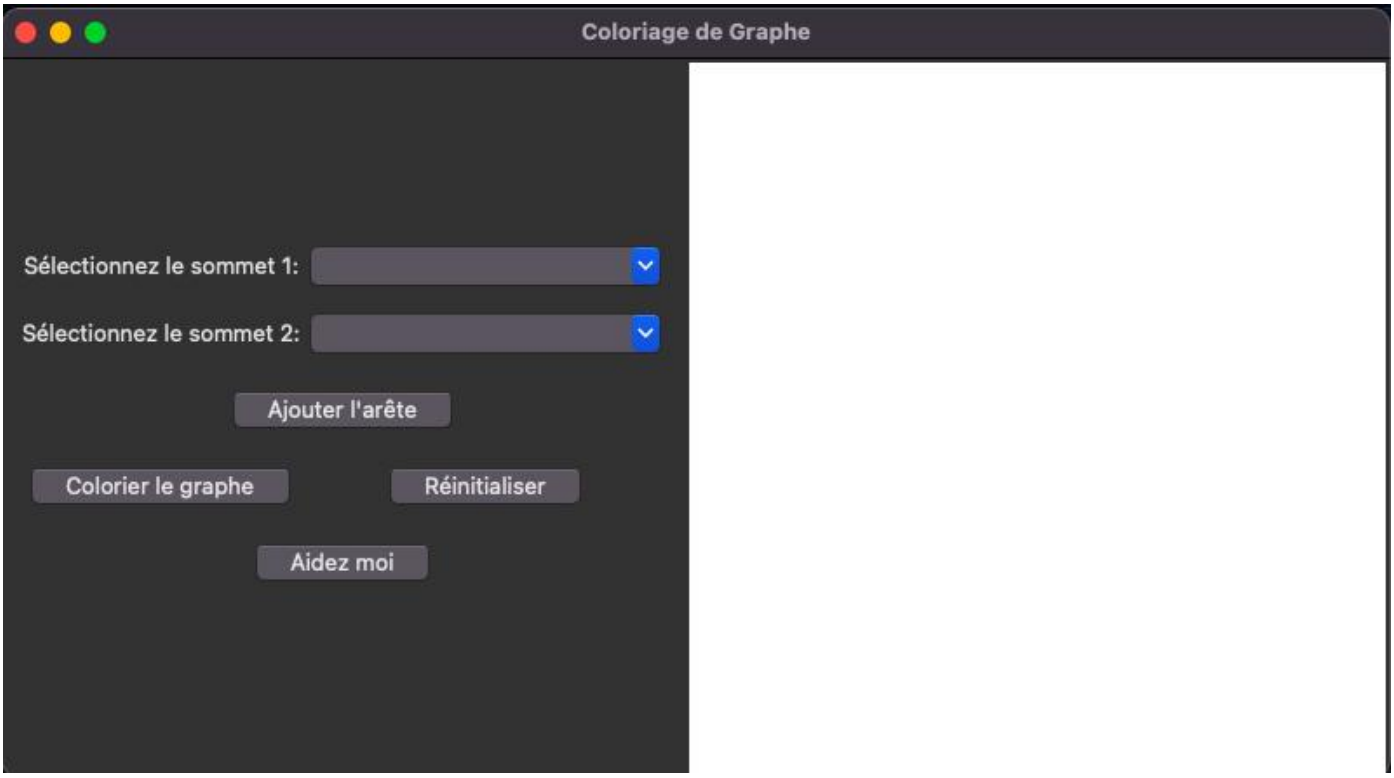
Dans un premier temps nous nous pencherons sur une application dédiée à la création et au coloriage de graphes. Nous explorerons comment cette application utilise les concepts dynamiques, ainsi que les techniques de coloriage, pour fournir une représentation visuelle et esthétiquement agréable des relations complexes au sein d'un système donné.

Et pour terminer, nous plongerons dans une application spécifique de résolution du Sudoku. En examinant les mécanismes algorithmiques utilisés, nous mettrons en lumière la manière dont les graphes sont exploités pour modéliser et résoudre cette énigme numérique emblématique : Le Sudoku.

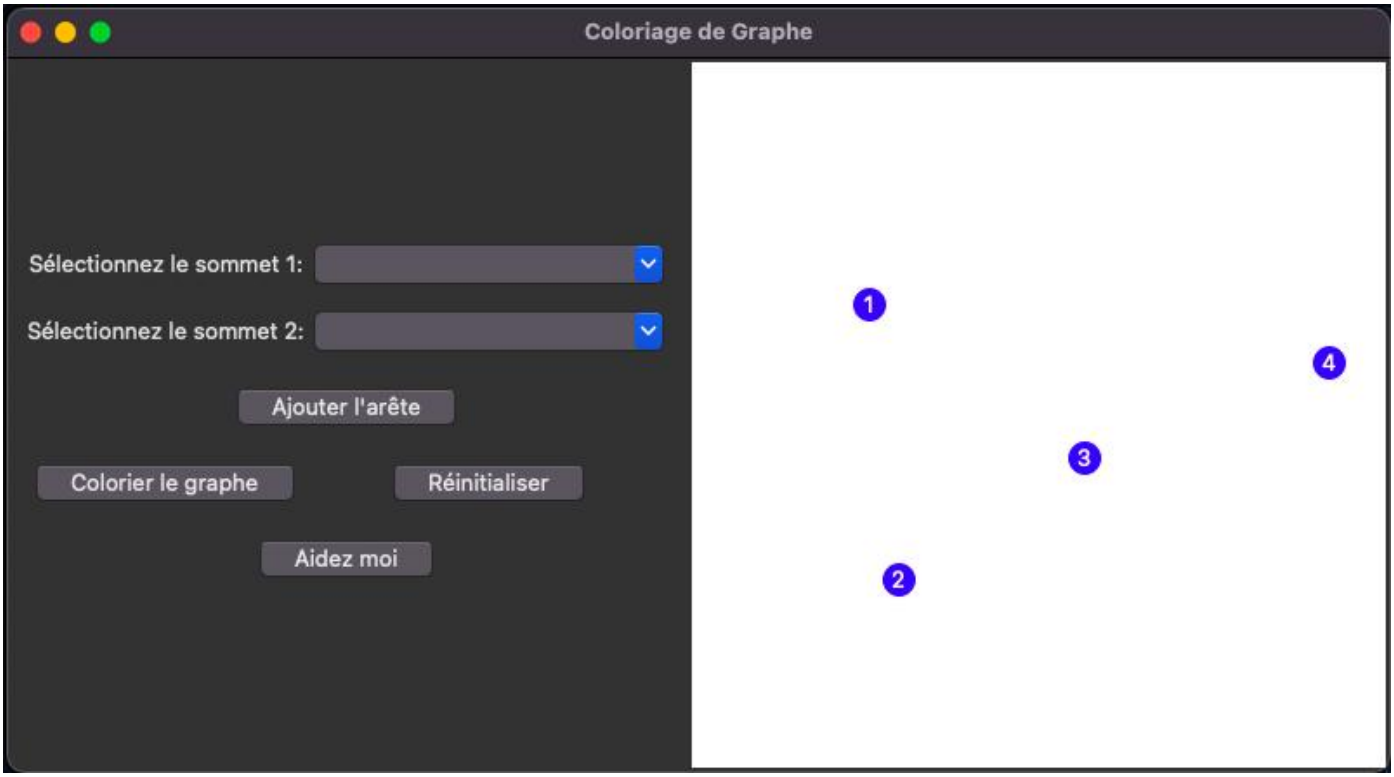
1. Application de coloriage de graphe

L'application de création et coloriage de graphe présentée dans cette section offre une immersion pratique dans l'univers des graphes dynamiques. En exploitant les caractéristiques évolutives de ces structures, l'application vise à fournir une représentation visuelle et interactive des relations complexes entre les entités d'un système.

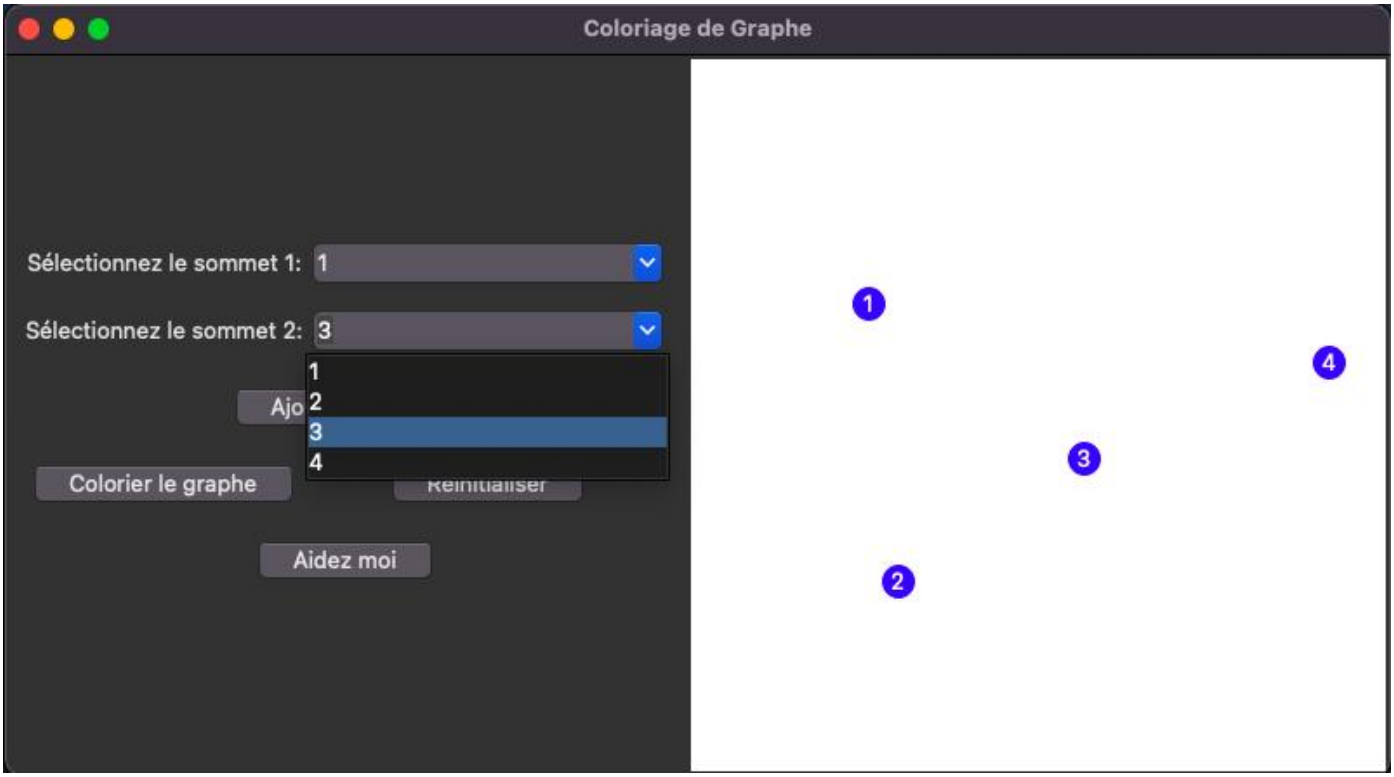
- Lors de la création de la fenêtre, une instance de ***DynamicGraph*** est créée



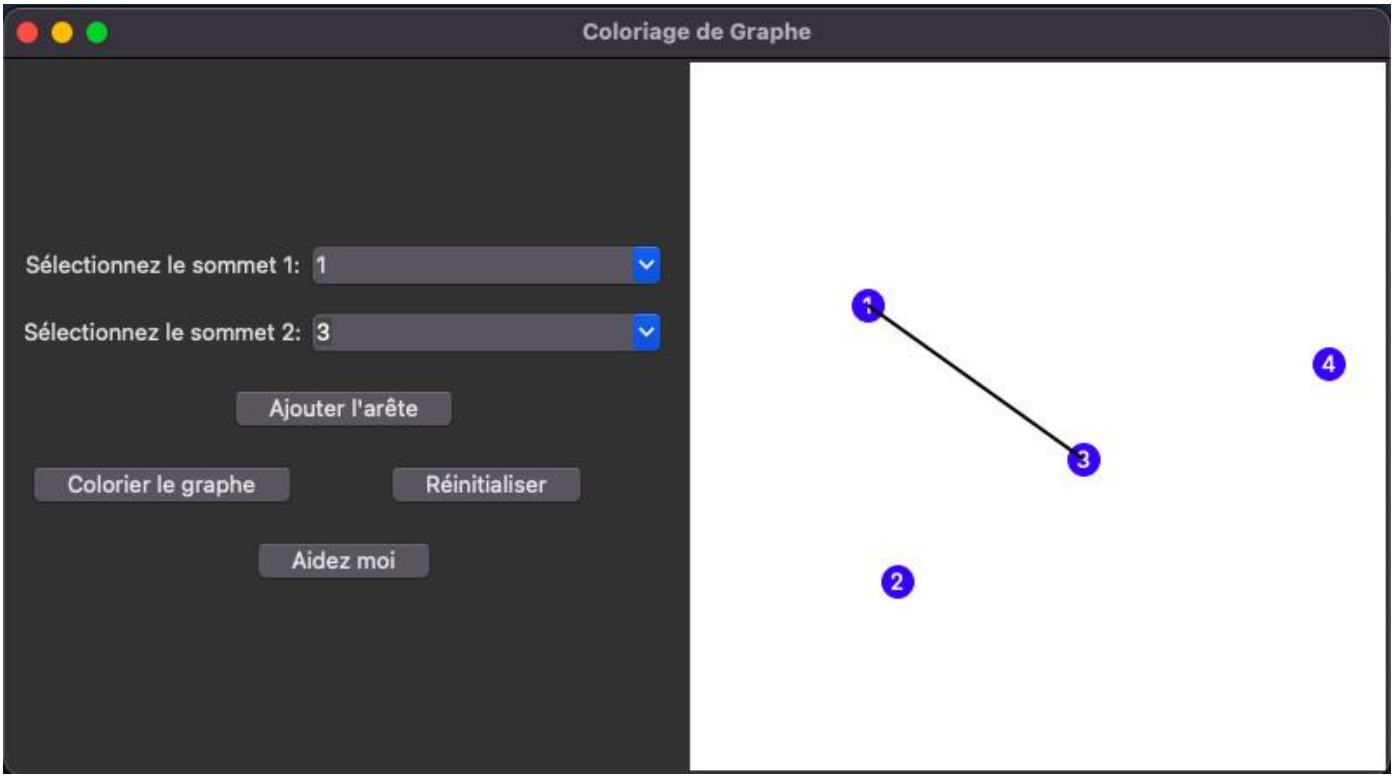
- A droite de la fenêtre il y'a un canvas pour dessiner le graphe. A chaque fois qu'on clique sur le canvas : un nouveau sommet se crée et sera numéroté automatiquement

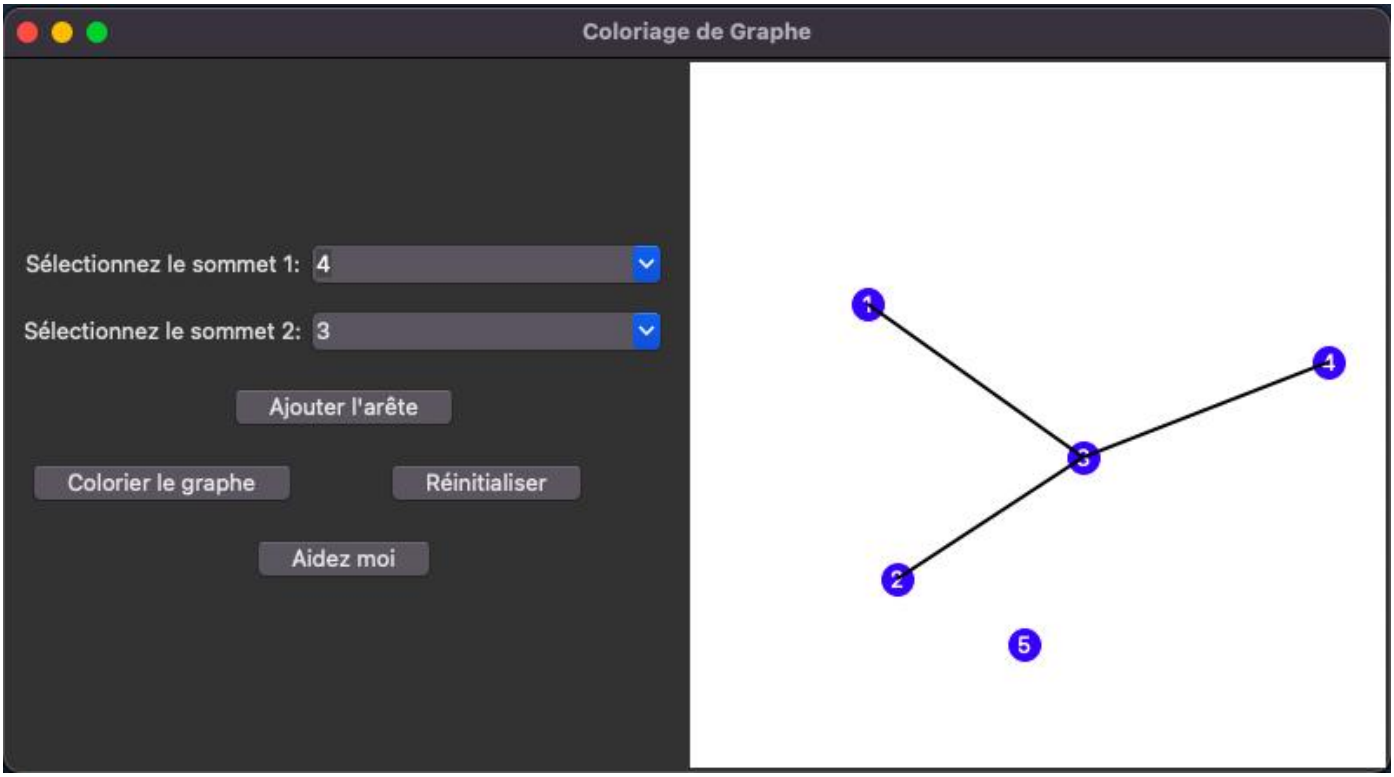


- Une fois les sommets créés, on clique sur les liste déroulantes pour les relier.

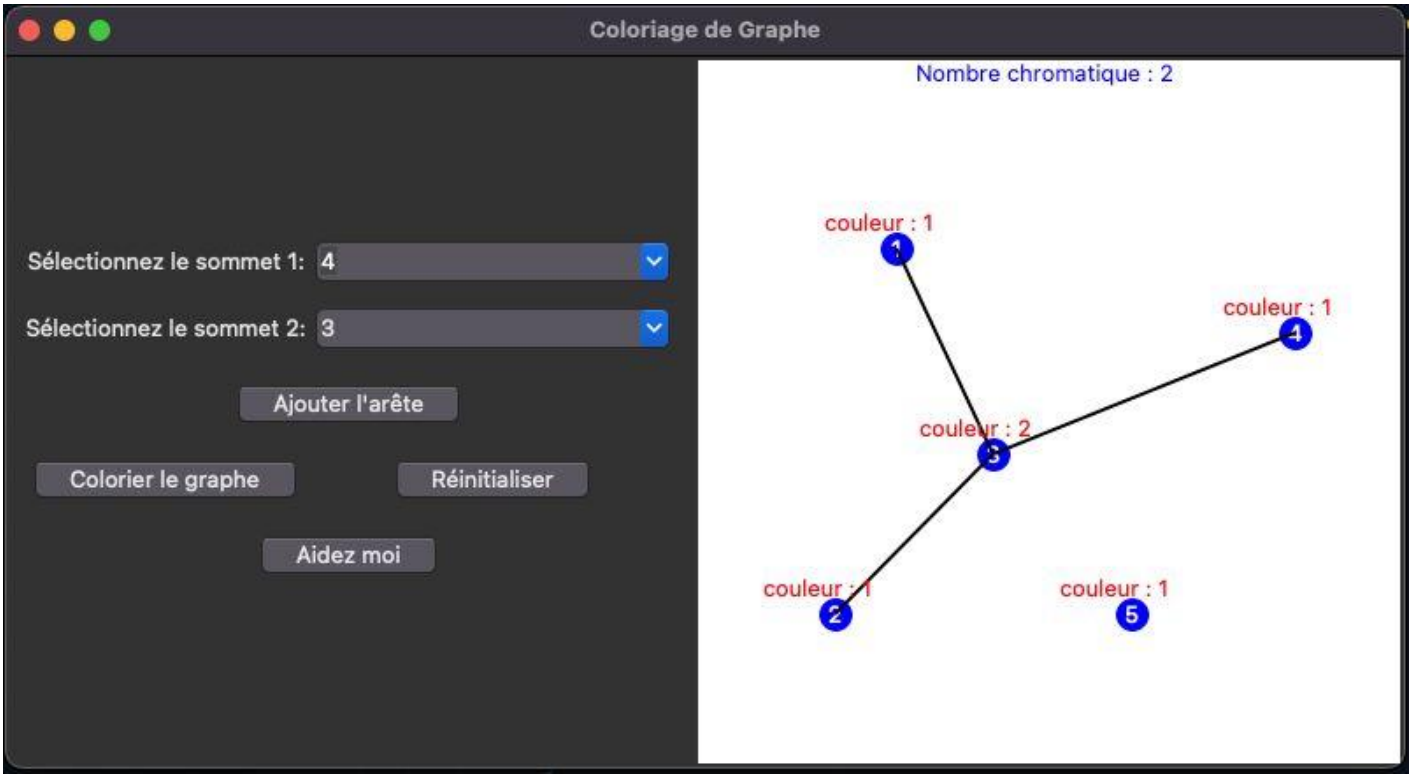


- Puis on peut créer une arête en cliquant sur ***Ajouter l'arête***

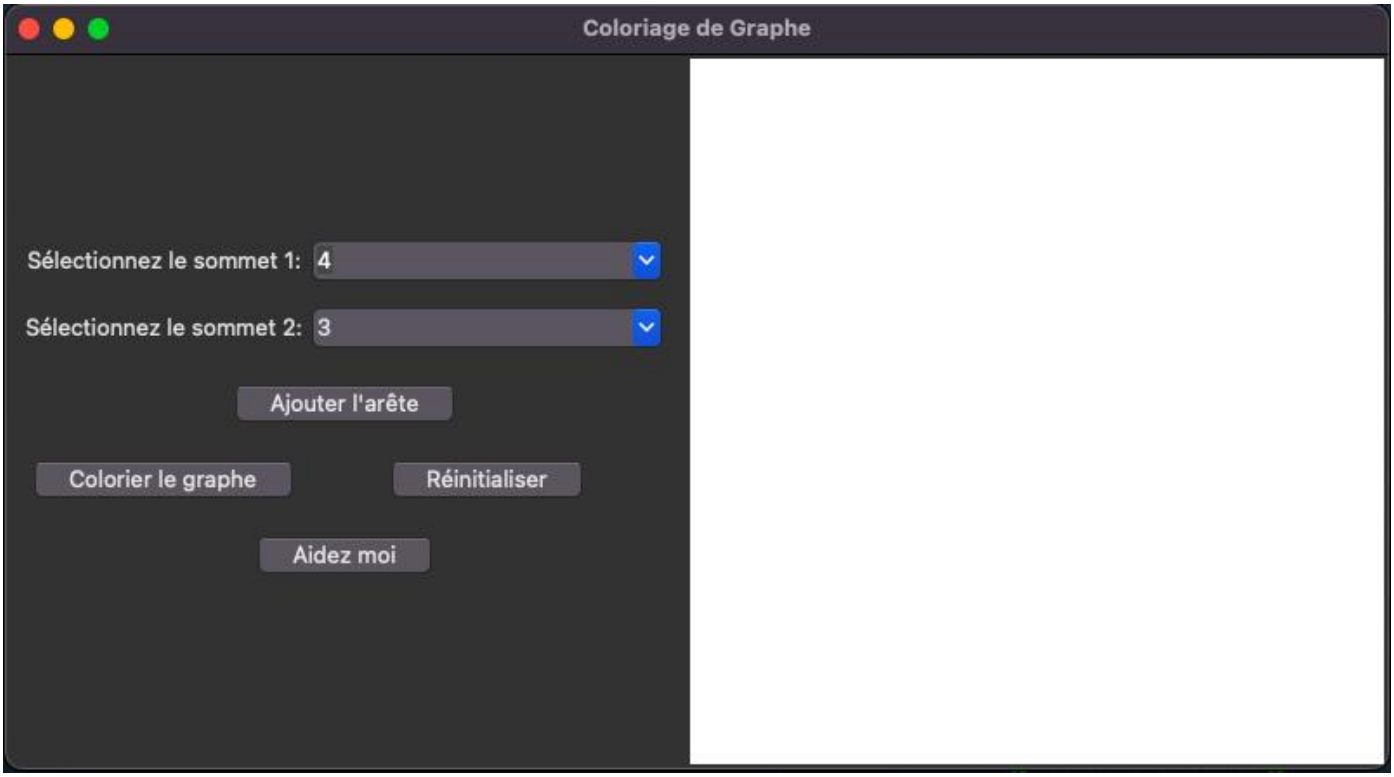




- Une fois le graphe créé, on peut le colorier en cliquant sur ***Colorier le graphe***



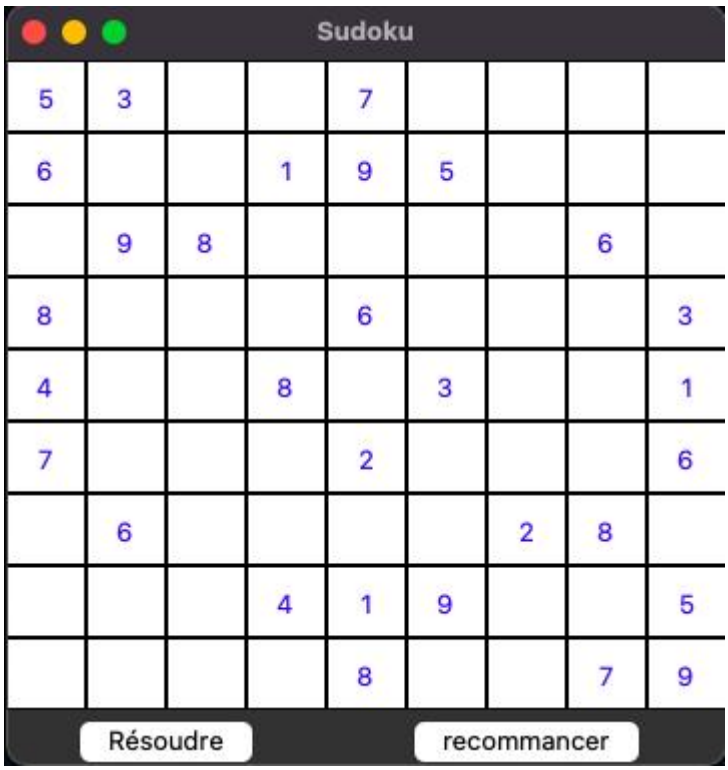
- Et puis on peut ***Réinitialiser*** l'application et puis recommencer

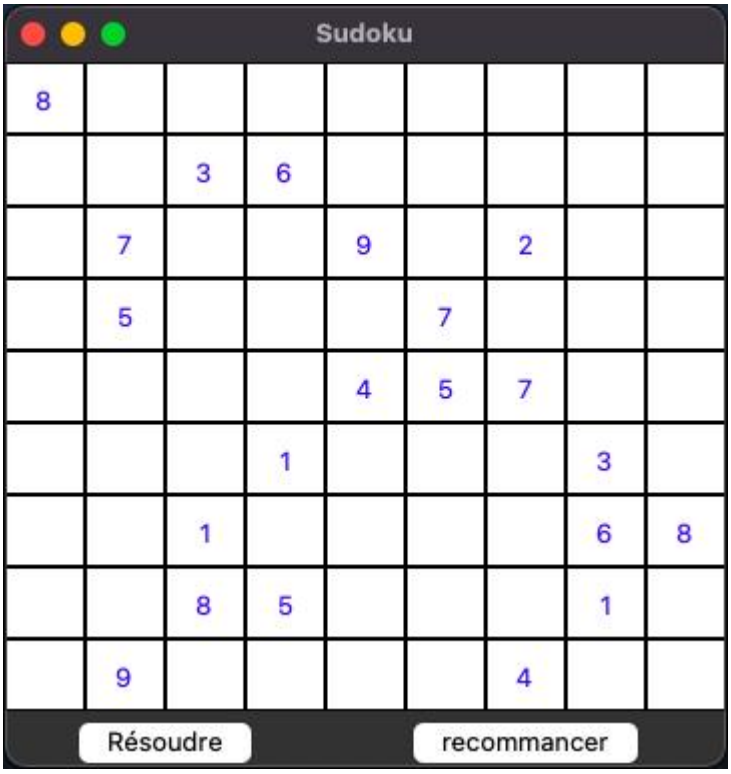


2. Application de resolution du Sudoku

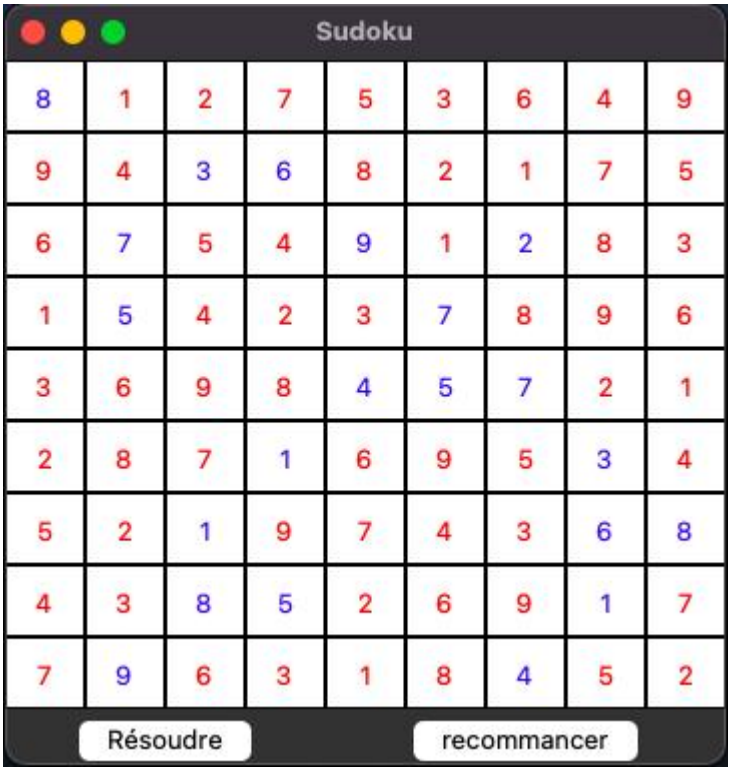
Cette section se plonge dans une application captivante qui tire parti des graphes statiques et utilise une approche de coloriage par backtracking pour résoudre le Sudoku.

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

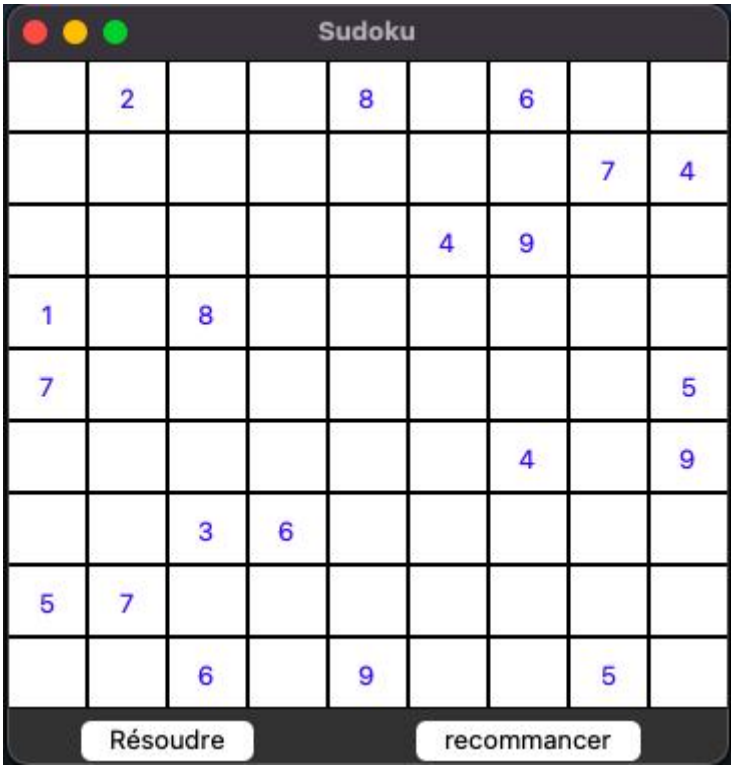




- Pour résoudre une grille : on clique sur ***Résoudre***



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



Conclusion

Notre exploration du coloriage d'un graphe dans le contexte de la résolution du Sudoku a mis en lumière la richesse des possibilités offertes par les représentations graphiques dans le domaine informatique.

À travers la première section, nous avons plongé dans les structures immuables des graphes statiques, révélant comment ces modèles statiques peuvent capturer les relations fondamentales au sein de divers systèmes.

La deuxième section a élargi notre perspective en introduisant les graphes dynamiques, offrant un aperçu des transformations temporelles et des évolutions dans des contextes complexes. Cette dynamique, essentielle pour représenter des systèmes changeants, enrichit considérablement notre capacité à modéliser et à comprendre des phénomènes réels.

Enfin, la troisième section a démontré la pertinence directe de ces concepts dans une application spécifique, à savoir la résolution du Sudoku. En combinant les principes des graphes statiques et dynamiques, nous avons pu aborder cette énigme numérique de manière innovante, offrant une méthodologie algorithmique visuelle et élaborée.

Au fil de notre exploration, il est devenu évident que le coloriage d'un graphe dépasse le simple aspect esthétique pour devenir une stratégie puissante dans la représentation et la résolution de problèmes. Les implications de cette approche sont vastes, touchant des domaines aussi variés que l'optimisation, la planification, et bien d'autres.

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.

Par conséquent, la recherche du nombre chromatique minimal reste un problème difficile, et dans de nombreux cas, des algorithmes approchés sont utilisés pour obtenir des résultats pratiques dans un temps raisonnable. Et il existe plusieurs algorithmes et heuristiques qui peuvent être utilisés pour

tenter de trouver une coloration avec un nombre chromatique proche du minimum. Certains des algorithmes couramment utilisés incluent: Algorithme Glouton, Algorithme de Coloration de Graphe par Backtracking, Algorithme de Welsh-Powell et Algorithme BFS.

En somme, notre étude nous invite à repenser la manière dont nous abordons les défis algorithmiques, en embrassant la puissance des représentations graphiques pour transformer des concepts abstraits en solutions tangibles. Ainsi, 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.