



UFR des sciences et techniques - site du Madrillet, Université de Rouen
Normandie

Coloriage d'un graphe Application à la résolution du Sudoku

Rapport de projet

Membres du groupe:

- ❖ Massinissa BRAHIMI
- ❖ Mamadou Lamine DIALLO

I. Introduction

Le présent rapport détaille la conception et l'implémentation d'algorithmes de coloriage de graphes en utilisant le langage Python. Les trois principaux algorithmes à explorer sont le glouton, le Welsh-Powell, et le backtracking. Ce travail aboutira à deux applications distinctes, chacune adressant des besoins particuliers.

Nous débuterons par la présentation d'une partie culturelle qui est dédiée à la signification historique et conceptuelle des coloriages de graphes. Cette exploration dévoile pourquoi cette discipline a été profondément étudiée et comment elle trouve des applications pratiques dans divers domaines tels que la planification des emplois du temps et la résolution de casse-têtes comme le Sudoku et un tas d'autres exemples que nous allons voir un peu plus loin dans ce rapport.

Les deux applications résultantes seront ensuite exposées dans le rapport. La première se focalise sur le coloriage de graphes dynamiques, permettant à l'utilisateur d'ajuster la topologie du graphe en temps réel. La seconde donnera à l'utilisateur la liberté de fournir son propre graphe en entrée, avec la possibilité de choisir un algorithme de coloriage parmi les trois mentionnés. Enfin, parmi ces algorithmes, le backtracking sera spécifiquement retenu pour résoudre le Sudoku en tant qu'application particulière de cette discipline.

II. Histoire et Théorie de Coloriage des Graphes

Le coloriage des graphes a suscité un intérêt considérable dans le domaine des mathématiques et d'informatique en raison de leur pertinence dans la résolution de divers problèmes. Une étude approfondie dans ce domaine trouve son importance dans son application à une gamme diversifiée de problèmes pratiques, théoriques et algorithmiques. Elle constitue une méthode puissante pour modéliser et résoudre des problèmes liés à la connectivité, à l'optimisation et à la prise de décision.

Cette discipline trouve ses racines dans la théorie des graphes, où l'objectif est d'attribuer des couleurs à des entités (représentées par des nœuds dans le graphe) de telle sorte que des nœuds adjacents aient des couleurs différentes. Ce problème, connu sous le nom de problème de coloration de graphes, a des applications pratiques étendues pour résoudre des problèmes complexes. Ces applications transcendent les disciplines et incluent:

- **Planification d'Emplois du Temps**
- **Allocation de Fréquences dans les Réseaux**
- **Cartographie des Couleurs**
- **Ordonnancement de Tâches**

- **Problèmes de Jeux** (tels que le sudoku dans notre projet)

Il convient de noter que la théorie des graphes, y compris les problèmes de coloriage de graphes, a joué un rôle significatif dans le développement de l'intelligence artificielle et de la cybersécurité, deux domaines majeurs de notre ère.

1. Dans le Domaine de l'Intelligence Artificielle

Le coloriage de graphes sert de fondation pour représenter et résoudre des problèmes complexes. Les relations entre entités sont modélisées efficacement à l'aide de graphes, contribuant ainsi à des applications telles que la reconnaissance de motifs, le traitement du langage naturel, et la recherche algorithmique. Ces concepts graphiques sont essentiels pour la création et l'optimisation d'algorithmes d'apprentissage automatique.

2. Dans le Domaine de la Cybersécurité

Le coloriage de graphes est d'une importance cruciale en cybersécurité. Elle trouve des applications dans la détection d'anomalies, la gestion des accès et des autorisations, l'analyse des réseaux de menaces, la détection d'intrusions, et la sécurisation des communications. En modélisant les réseaux informatiques sous forme de graphes, les problèmes de coloriage deviennent des outils puissants pour identifier des schémas malveillants, organiser les autorisations, et renforcer les défenses contre les cyberattaques. Ces techniques graphiques contribuent significativement à la sécurité des systèmes informatiques.

III. LES ALGORITHMES FONDAMENTAUX EN LANGAGE ALGORITHMIQUE

On a opté pour la représentation par liste de successeurs en raison de sa pertinence pour les algorithmes de coloration. Cette structure offre la flexibilité nécessaire pour des algorithmes tels que le glouton, nécessitant une exploration approfondie des voisins de chaque sommet pour décider de l'attribution des couleurs. De plus, la liste de successeurs facilite les modifications locales, permettant des ajouts ou suppressions d'arêtes de manière efficace, une caractéristique utile dans un contexte dynamique. Elle simplifie également le parcours du graphe et l'itération sur les voisins, ce qui est essentiel pour les algorithmes nécessitant un accès fréquent aux voisins tels que les algorithmes de coloration envisagés.

1. Algorithme Gloutton:

Algorithme Glouton_Coloration_Graphe(graphe):

// Initialisation

vertex_list <- Liste des sommets du graphe

colored <- Dictionnaire vide

// Tant qu'il reste des sommets non colorés

Tant que (vertex_list) n'est pas vide:

 sommet <- Prendre le premier sommet dans vertex_list

 couleurs_voisins <- Couleurs des voisins déjà colorés du sommet

 // Trouver la couleur disponible la plus basse

 indice_couleur <- 0

 Tant que (indice_couleur) est dans couleurs_voisins:

 indice_couleur <- indice_couleur + 1

 // Attribuer la couleur au sommet

 colored[sommet] <- Couleur d'indice_couleur

 Retirer sommet de vertex_list

Retourner colored

Fin Algorithme

- **Informations utiles sur leur complexité en temps par rapport à la structure des données utilisées.**

L'algorithme glouton utilise un dictionnaire pour représenter le graphe, où les clés sont les sommets et les valeurs sont les listes de voisins. La coloration est également représentée par un dictionnaire, où les clés sont les sommets et les valeurs sont les couleurs attribuées.

La complexité de cet algorithme dépend du nombre de sommets et d'arêtes dans le graphe. Dans le pire cas, la complexité est $O(V^2)$ pour V sommets, car chaque sommet peut potentiellement être connecté à chaque autre sommet. Cependant, la performance peut être bien meilleure dans la pratique, en particulier pour les graphes de petite taille ou les graphes avec une structure régulière. L'algorithme glouton est rapide, mais il ne garantit pas toujours la coloration optimale.

2. Algorithme de Welsh-Powell:**Algorithme Welsh_Powell(graphe):**

// Fonction pour calculer le degré d'un sommet

Fonction Degre(graphe, sommet):

 Retourner Longueur(graphe[sommet])

// Fonction pour trier les sommets par degré décroissant

Fonction TrierSommetsParDegre(graphe):

```

sommets <- ListeDesClefs(graphe)
sommets_tries <- Trier(sommets, FonctionComparerDegreDecroissant)
Retourner sommets_tries

// Fonction pour comparer deux sommets par degré décroissant
Fonction ComparerDegreDecroissant(sommet1, sommet2):
    Retourner Degre(graphe, sommet2) - Degre(graphe, sommet1)

// Fonction principale de l'algorithme Welsh-Powell
Fonction Welsh_Powell(graphe):
    sommets_tries <- TrierSommetsParDegre(graphe)
    couleur_sommets <- {} // Dictionnaire pour stocker les couleurs attribuées

    Tant que sommets_tries non vide:
        // Prendre le sommet de plus haut degré
        sommet <- ExtrairePremierElement(sommets_tries)
        // Ensemble pour stocker les couleurs des voisins
        voisins_couleurs <- {}

        Pour chaque voisin dans graphe[sommet]:
            Si voisin dans couleur_sommets:
                Ajouter couleur_sommets[voisin] à voisins_couleurs

        // Trouver la première couleur disponible pour le sommet
        nouvelle_couleur <- PremiereCouleurNonUtilisee(voisins_couleurs,
pallete_couleurs)
        couleur_sommets[sommet] <- nouvelle_couleur

    Retourner couleur_sommets

// Fonction pour trouver la première couleur non utilisée
Fonction PremiereCouleurNonUtilisee(couleurs_utilisees, palette_couleurs):
    Pour chaque couleur dans palette_couleurs:
        Si couleur non dans couleurs_utilisees:
            Retourner couleur

// Fonction pour extraire le premier élément d'une liste
Fonction ExtrairePremierElement(liste):
    PremierElement <- liste[0]
    SupprimerElement(liste, PremierElement)
    Retourner PremierElement

```

Fin Algorithme

Complexité en temps :

Le calcul du degré d'un sommet avec **degre(graphe, sommet)** a une complexité en temps de $O(1)$.

Le tri des sommets avec **trier_sommets_par_degre(graphe)** a une complexité en temps de $O(|V| \log |V|)$, où $|V|$ est le nombre de sommets.

L'algorithme **Welsh-Powell** lui-même a une complexité en temps de $O(|V|^2)$ dans le pire des cas, où $|V|$ est le nombre de sommets. Cela résulte de la boucle principale qui parcourt tous les sommets et, dans chaque itération, examine les voisins et les couleurs des sommets voisins.

Structures de données utilisées :

graphe (dict): Dictionnaire représentant le graphe.

sommets_tries (list): Liste pour stocker les sommets triés par degré.

couleur_sommets (dict): Dictionnaire pour stocker les couleurs attribuées à chaque sommet.

3. Algorithme du backtracking (rebroussement):

Algorithme Backtrack_Coloration_Graphe(graphe, couleurs):

// Initialisation

solution <- Dictionnaire vide

// Initialiser tous les sommets à aucune couleur assignée

Pour chaque sommet dans graphe:

 solution[sommet] <- None

// Appel de la fonction récursive

Backtrack(graphe, solution, couleurs)

Fin Algorithme

Algorithme Backtrack(graphe, solution, couleurs):

// Cas de base : tous les sommets ont été assignés

Si il n'y a plus de sommets non colorés dans solution:

 Afficher la solution

 Retourner

// Choisir le prochain sommet non coloré

sommet <- Obtenir le prochain sommet non coloré dans solution

// Essayer chaque couleur possible

Pour chaque couleur dans couleurs:

Si il est possible de colorer le sommet avec cette couleur:

Colorer le sommet dans la solution

// Appel récursif pour le prochain sommet

Backtrack(graphe, solution, couleurs)

// Si la solution a été trouvée, arrêter la recherche

Si la solution est complète:

Retourner

// Retour arrière : décolorer le sommet

Décolorer le sommet dans la solution

Fin Algorithme

- **Informations utiles sur leur complexité en temps par rapport à la structure des données utilisées**

L'algorithme du backtrack utilise un dictionnaire pour représenter le graphe, où les clés sont les sommets et les valeurs sont les listes de voisins. La solution est également représentée par un dictionnaire, où les clés sont les sommets et les valeurs sont les couleurs attribuées.

La complexité de cet algorithme dépend de la structure du graphe et du nombre de couleurs. Dans le pire cas, si le graphe est complet et chaque sommet a besoin d'être vérifié pour chaque couleur, la complexité peut être exponentielle. Cependant, dans la pratique, elle peut être bien meilleure, en particulier si le graphe a des structures régulières. La performance dépend fortement du choix de l'heuristique de sélection des sommets et des couleurs.

IV. structure de notre programme

les modules sont les suivants:

- **greedy.py**: ce module implémente l'algorithme glouton, un outil d'optimisation résolvant divers problèmes en effectuant des choix locaux optimaux à chaque étape.

Fonctionnalités Principales :

Implémentation de l'Algorithme Glouton : La fonction **gloutton(graphe)** attribue des couleurs aux sommets d'un graphe pour minimiser le nombre total de couleurs utilisées.

Intégration à l'Interface Utilisateur : Le module est appelé depuis le module principal (main.py) pour la résolution gloutonne lorsque choisie par l'utilisateur.

Usage dans l'Application :

L'utilisateur peut construire ou charger un graphe via l'interface graphique fournie par la bibliothèque **tkinter** dans le module main.py. Après l'exécution de l'algorithme, les résultats sont affichés graphiquement en utilisant la fonction **draw_colored_graph(graph, colors)** de **display.py**.

- **backtrack.py:** ce module propose une mise en œuvre de l'algorithme du **backtrack**, un outil d'optimisation résolvant divers problèmes en explorant de manière récursive toutes les solutions possibles, en revenant en arrière dès qu'une impasse est atteinte.

Fonctionnalités Principales :

Implémentation de l'Algorithme de Retour Arrière : La fonction **backtrack(graphe, vertex, colors, solution)** attribue des couleurs aux sommets d'un graphe en explorant les différentes combinaisons possibles, revenant en arrière si une configuration mène à une impasse.

Intégration à l'Interface Utilisateur : Le module est appelé depuis le module principal (main.py) lorsqu'une résolution par retour arrière est choisie par l'utilisateur. L'interface utilisateur graphique est gérée par **tkinter**.

Usage dans l'Application :

L'utilisateur peut construire ou charger un graphe via l'interface graphique fournie par la bibliothèque **tkinter** dans le module main.py . Après l'exécution de l'algorithme, les résultats sont affichés graphiquement en utilisant la fonction **draw_colored_graph(graph, colors)** de **display.py**.

- **welsh_powell.py**: ce module offre une implémentation de l'algorithme de **Welsh-Powell**, un outil d'optimisation résolvant divers problèmes en attribuant des couleurs aux sommets d'un graphe, avec une stratégie basée sur le degré des sommets.

Fonctionnalités Principales :

Implémentation de l'Algorithme de Welsh-Powell : La fonction `welsh_powell(graphe)` attribue des couleurs aux sommets d'un graphe de manière à minimiser le nombre total de couleurs utilisées, en se basant sur le degré des sommets.

Intégration à l'Interface Utilisateur : Le module est appelé depuis le module principal (`main.py`) lorsqu'une résolution par l'algorithme de Welsh-Powell est choisie par l'utilisateur. L'interface utilisateur graphique est gérée par la bibliothèque **tkinter**.

Usage dans l'Application :

L'utilisateur peut construire ou charger un graphe via l'interface graphique fournie par la bibliothèque **tkinter** dans le module `main.py`. Après l'exécution de l'algorithme, les résultats sont affichés graphiquement en utilisant la fonction **`draw_colored_graph(graph, colors)`** de `display.py`.

- **display.py**: ce module est responsable de l'affichage graphique des résultats de la coloration du graphe. Il utilise la bibliothèque [NetworkX](#) pour représenter le graphe et [Matplotlib](#) pour le dessiner de manière visuelle.

Fonction Principale :

Affichage du Graphe Coloré : La fonction **`draw_colored_graph`** prend en entrée le graphe ainsi que la coloration attribuée à chaque sommet. Elle utilise [NetworkX](#) pour représenter le graphe et [Matplotlib](#) pour le dessiner de manière claire.

- **color.py**: ce module offre une gestion structurée et pré-définie des couleurs utilisées dans notre application de coloration de graphe. Il utilise une énumération (Enum) pour définir une liste de couleurs prédéfinies et fournit également une palette de couleurs à utiliser dans tout le programme.
- **main.py**: ce module sert d'interface principale pour permettre à l'utilisateur de choisir parmi trois algorithmes de coloration de graphe, à savoir l'algorithme de Backtrack, l'algorithme de Welsh-Powell, et l'algorithme Glouton (Greedy).

Fonctionnalités Principales :

Sélection d'Algorithmes : Trois boutons sont proposés à l'utilisateur, chacun représentant l'un des algorithmes disponibles. Chaque bouton,

une fois pressé, lance l'algorithme associé pour la coloration du graphe.

Exécution de l'Algorithme : La fonction **run_algorithm** permet de déclencher l'algorithme sélectionné. Elle prend en paramètre la fonction correspondant à l'algorithme choisi.

Interface Intuitive : L'interface utilisateur est conçue de manière à être simple et intuitive. Les boutons sont clairement libellés avec le nom de l'algorithme qu'ils déclenchent.

Sortie visuelle : Les résultats de l'algorithme sont visualisés graphiquement grâce aux fonctions de dessin du module **display.py**. Une fenêtre graphique s'ouvre pour montrer le graphe coloré et un bouton "**Quit**" permet de fermer l'application de manière propre.

Voici un exemple d'exécution :

(remarque : dans l'exemple j'ai choisi l'algorithme de welsh-powell et 2 sommet)

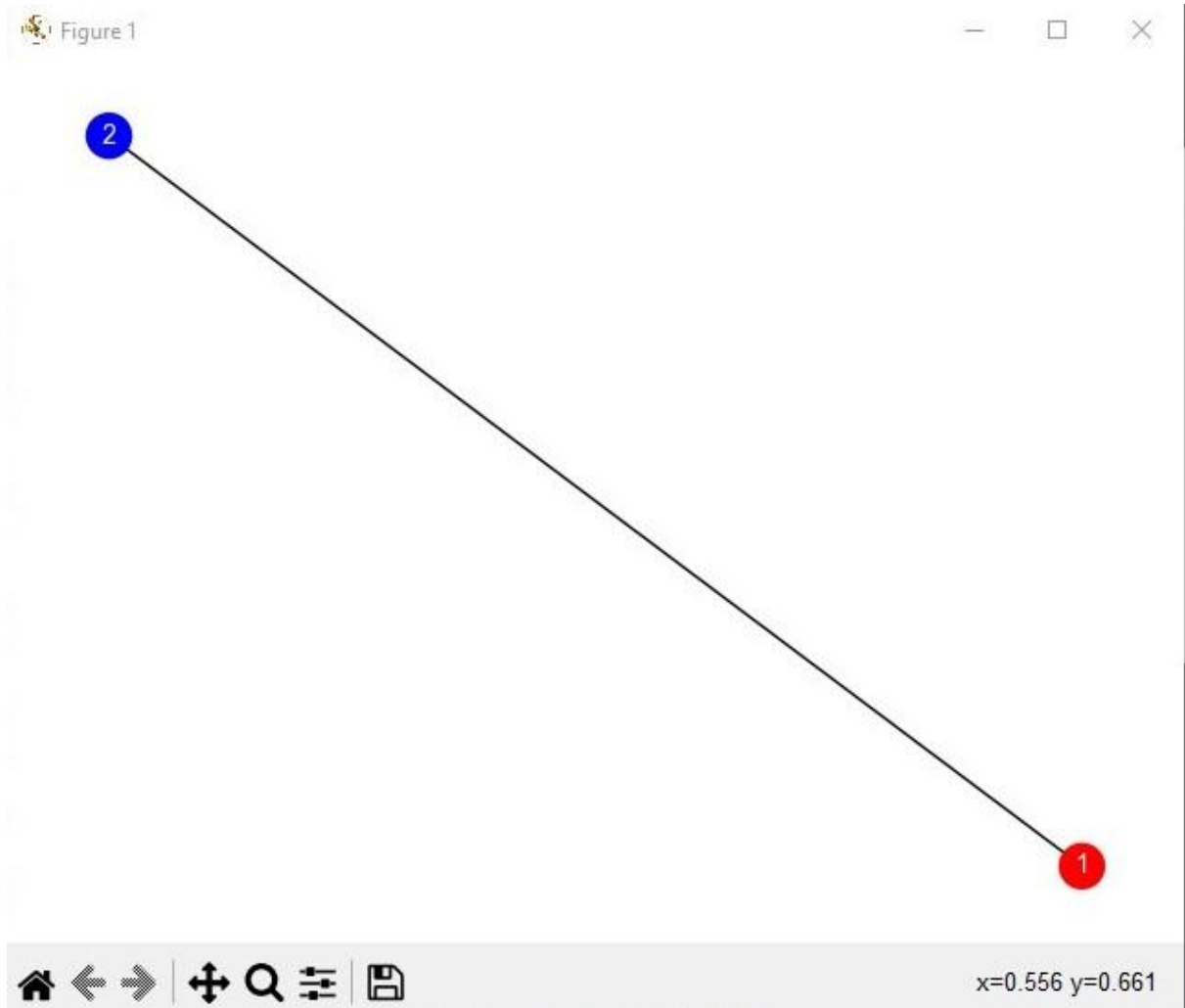


Ajouter un sommet

Sommet:

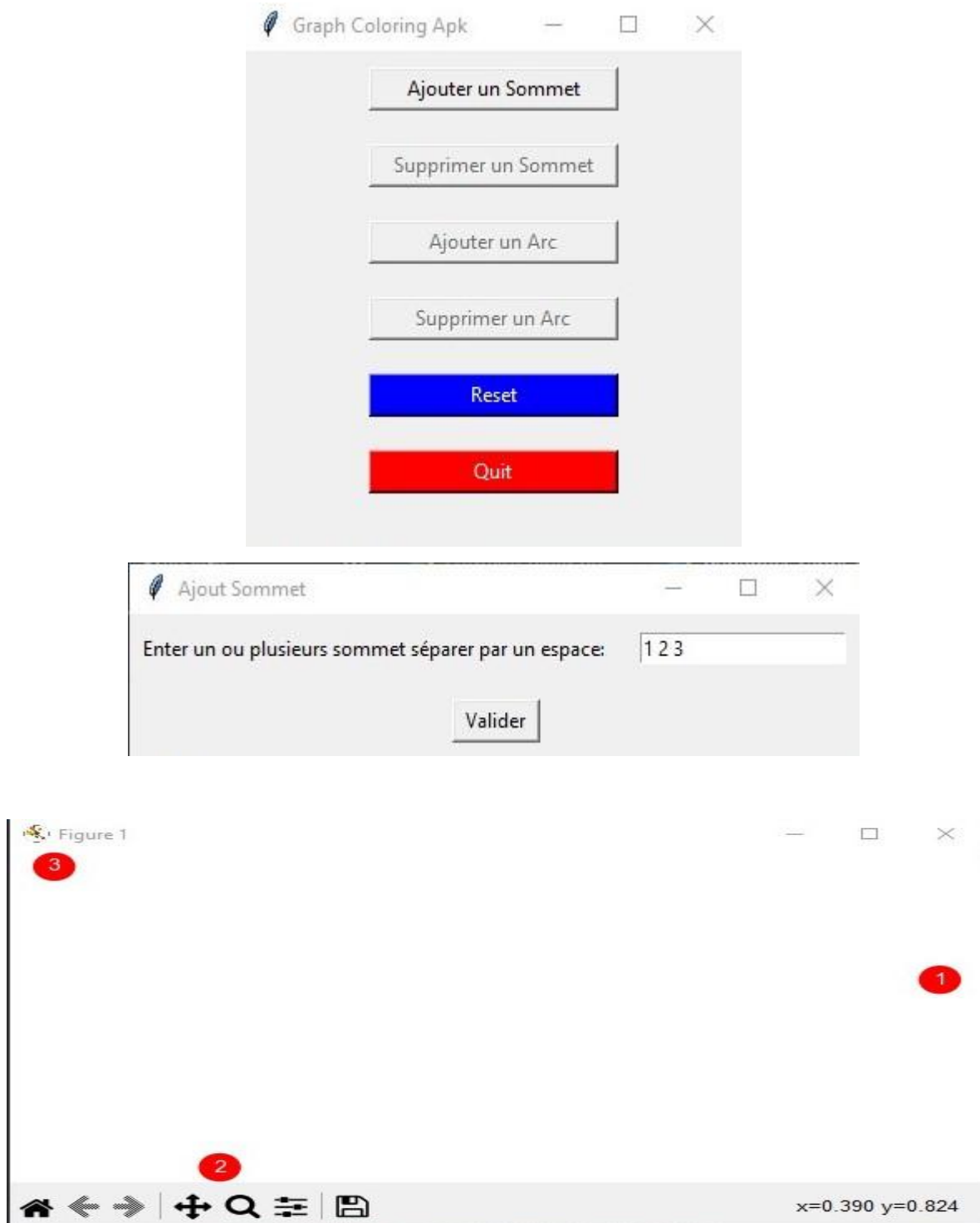
Voisins (séparés par des espaces):

Valider



- **dynamicGraphColoring.py:** module qui utilise l'algorithme de welsh-powell afin de réaliser une application qui permet de faire la coloration d'un graphe dynamique ou le nombre des sommets et d'arêtes évoluent avec le temps.

Exemple d'exécution:




Suppression So...

Enter un sommet:

Valider

Info

 le sommet '1' a été supprimé avec success

OK

Figure 1

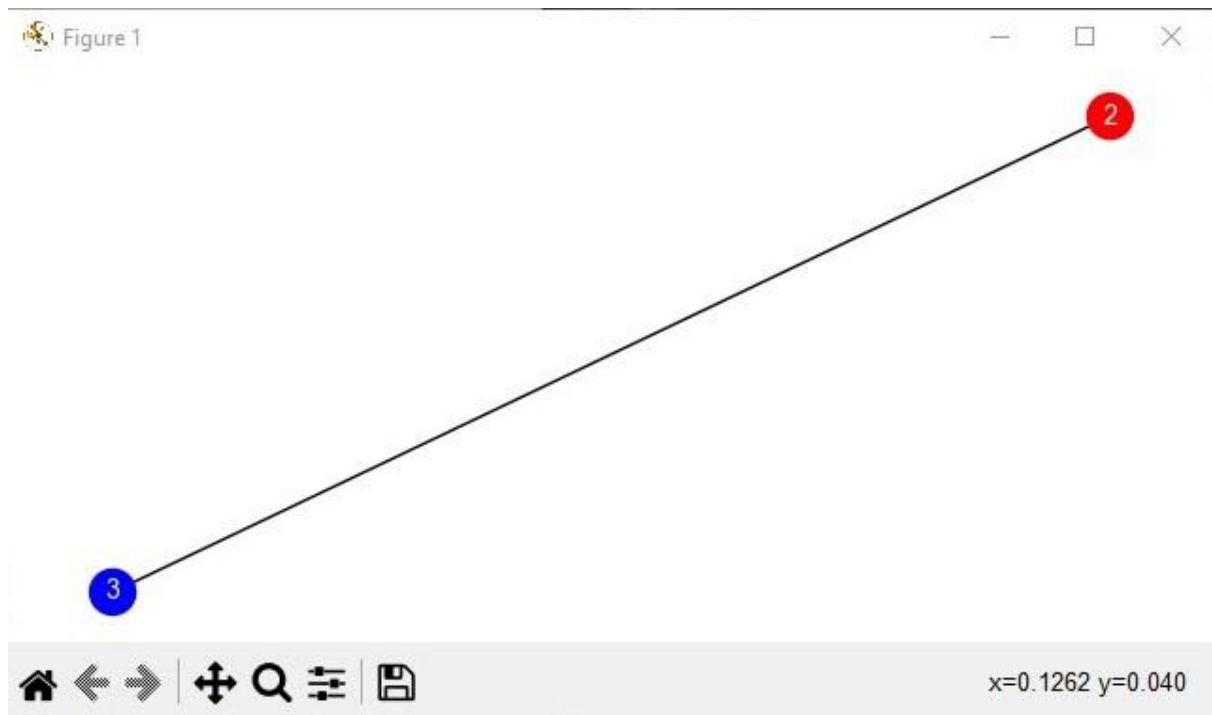


Ajout arc

Enter le premiere extrémite :

Enter la deuxieme extrémite :

Valider



Suppression arc

Entrer le première extrémité :

Entrer la deuxième extrémité :





- **sudokuGUI.py:** utilise la bibliothèque **Tkinter** pour créer une interface utilisateur graphique (GUI) permettant à l'utilisateur de visualiser la grille de Sudoku initiale, la grille de couleurs et la solution du Sudoku.

Fonctionnalités Principales :

Attribution des Indices :

La fonction **assign_indices** attribue des indices uniques à chaque position (x, y) dans une grille 9x9. Ces indices sont utilisés pour représenter chaque cellule de manière linéaire.

Classe SudokuGUI :

La classe SudokuGUI représente l'interface graphique de l'application. Elle prend en paramètre la grille de Sudoku initiale, la grille de couleurs et la solution du Sudoku.

La méthode **create_grid** crée trois cadres distincts pour afficher la grille initiale, la grille de couleurs et la solution.

Chaque cadre contient un titre, suivi d'un canevas pour afficher la grille respective.

Méthodes de Dessin :

La méthode **draw_grid** est utilisée pour dessiner la grille de Sudoku initiale et la solution. Elle utilise des rectangles pour représenter chaque cellule et affiche les chiffres correspondants dans les cellules non vides.

La méthode **draw_color_grid** est utilisée pour dessiner la grille de couleurs. Elle remplit les cellules avec des couleurs en fonction de la solution colorée. Affichage de l'Interface Graphique :

La fonction **afficher_interface_graphique** crée une instance de la classe **SudokuGUI** avec les paramètres nécessaires et affiche l'interface graphique en utilisant Tkinter.

- **sudoku.py:**

Initialisation du Sudoku

La section de code initiale définit une grille vide et un dictionnaire de couleurs. La fonction **est_valide** vérifie si une valeur peut être légalement placée dans une cellule en fonction des règles du Sudoku. La fonction **initGrille** permet à l'utilisateur de remplir la grille en entrant des valeurs, en s'assurant que les entrées sont valides.

Fonction Utilitaire

La fonction **inverse_dict** est une fonction utilitaire qui inverse les clés et les valeurs d'un dictionnaire. Dans le contexte du programme, elle est utilisée pour inverser le dictionnaire **colors_dict**, permettant de convertir les couleurs de la représentation du graphe à leur représentation initiale.

Création du Graphe à partir de la Grille

La fonction **graph_from_grid** crée un graphe à partir de la grille de Sudoku. Elle utilise les règles du Sudoku pour déterminer les voisins de chaque cellule dans le graphe.

Résolution du Sudoku avec Backtracking

La fonction **sol** utilise l'algorithme de backtracking pour attribuer des couleurs aux cellules du graphe de manière à résoudre le Sudoku. Elle vérifie également la validité des couleurs assignées en fonction des règles du Sudoku.

Conversion de la Solution

La fonction **sol_final** convertit la solution obtenue du backtracking en utilisant le dictionnaire inversé **colors_inv**. Elle prépare la solution finale pour l'affichage.

Affichage de la Solution dans le Terminal

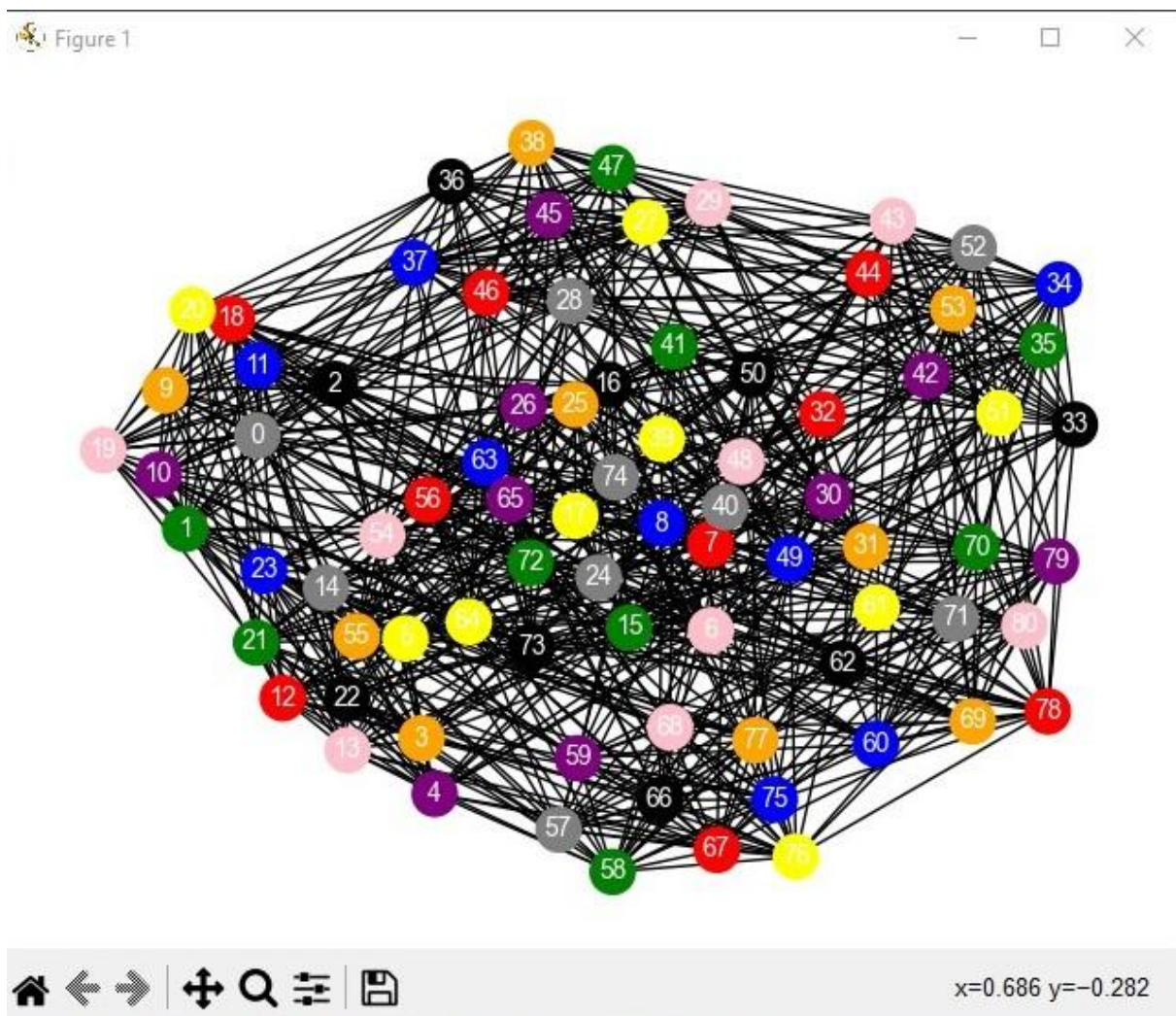
La fonction **sudokuSolver** affiche la grille initiale, la solution du Sudoku avec les couleurs attribuées, et utilise **draw_colored_graph** pour visualiser le résultat graphiquement.

Fonction Principale

La section finale exécute l'initialisation de la grille, la résolution du Sudoku, et affiche l'interface graphique avec la solution.

Exemple d'exécution:

Pour l'exécution on affiche d'abord le graphe de contraintes généré par l'algorithme du backtrack puis on affiche le résultat sous forme de 3 grilles la première pour la grille de sudoku initial entrer par l'utilisateur la deuxième pour les couleurs correspondantes puis la grille de solution finale.



Sudoku Solver

— □ ×

Sudoku Initial

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 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 |

Grille de Couleurs

| | | | | | | | | |
|--------|--------|--------|--------|--------|--------|--------|--------|--------|
| Grey | Green | Black | Orange | Purple | Yellow | Pink | Red | Blue |
| Orange | Purple | Blue | Red | Pink | Grey | Green | Black | Yellow |
| Red | Pink | Yellow | Green | Black | Blue | Grey | Orange | Purple |
| Yellow | Grey | Pink | Purple | Orange | Red | Black | Blue | Green |
| Black | Blue | Orange | Yellow | Grey | Green | Purple | Pink | Red |
| Purple | Red | Green | Pink | Blue | Black | Yellow | Grey | Orange |
| Pink | Orange | Red | Grey | Green | Purple | Blue | Yellow | Black |
| Blue | Yellow | Purple | Black | Red | Pink | Orange | Green | Grey |
| Green | Black | Grey | Blue | Yellow | Orange | Red | Purple | Pink |

Sudoku Solution

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