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 loins 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 des Coloriages de Graphes

Le coloriage de 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, à 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 coloriages 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 coloriages 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

//TODO