Rapport de projet

Coloriage d'un graphe et résolution du Sodoku

Réalisé par :

Mazarine DAHMANI

Mouhamed Rachid MOUCI

Introduction

Ce document est le rapport de notre projet d'Algorithmique des graphes qui consiste à implémenter les trois algorithmes : Glouton, Welsh_Powell, Backtracking, qui permettent de colorier un graphe, de concevoir et d'implémenter un algorithme qui colorie un graphe dynamique et de développer une application, en utilisant les algorithmes précédents, qui permet de résoudre une grille de Sudoku donné par l'utilisateur.

Dans ce rapport, nous allons tout d'abord présenter une partie culturelle qui explique l'importance de la coloration des graphes ainsi que les différents problèmes où cette dernière est utilisée, puis présenter les différents algorithmes demandés en langages algorithmique ainsi que leur complexité, et pour finir la structure de notre application.

Coloration de graphes : Histoire et Applications

La coloration des graphes est sans doute le sujet le plus populaire de la théorie des graphes. Depuis l'émergence du fameux problème des quatre couleurs, posé par Francis Guthrie en 1852, l'intérêt pour la coloration des graphes a fortement augmenté. Le problème des quatre couleurs, questionne sur la possibilité de colorer toute une carte géographique avec seulement quatre couleurs tout en respectant la condition que deux pays partageant la même frontière ne partage pas la même couleur. Ce problème fu finalement résolu par Kenneth Appel et Wolfgang Haken en 1976.

Outre le problème des quatre couleurs, d'autres problèmes célèbres utilisent la coloration des graphes et ont contribué à l'importance de cette dernière. Parmi eux, on trouve :

- **L'organisation des examens :** Comment établir des horaires pour des examens de manières à éviter les conflits entre les examens partageant des étudiants communs.
- **Optimisation de fréquence dans les réseaux de télécommunication :** Comment minimiser le nombre de fréquences utilisées dans un réseau de télécommunication pour éviter les interférences ?
- **Résolution du sudoku :** Comment remplir une grille avec des chiffres différents de sorte qu'ils n'apparaissent jamais deux fois dans une même ligne, une même colonne ou une même zone ?

Algorithmes

1. Algorithme Glouton

1.1. Langage algorithmique :

Algorithme GreedyColoring

{**Entrée** : liste_suc : Liste des successeurs pour chaque sommet, number : Entier}
{**Sortie** : result : Tableau d'Entiers}
Var liste_succ : ListeSuccesseurs ; number, i, j, color : Entier ; result : Tableau[1..number] d'Entier ; used_color : Tableau [1..number] de Booléen ;
Debut
Pour i allant de 1 à number **faire**
    result [i] ←-1;
FinPour
result[1] ← 0;
Pour i de 1 à number **faire**
    used_colors[i] ← Faux;
Finpour
Pour i de 1 à number **faire**
    Pour chaque j dans liste_suc[i] **faire**
        Si result[j] ≠ -1 **alors**
            used_colors[result[j]] ← Vrai;
        Finsi

    Finpour
    color ← 0;
    Tantque color < number **faire**
        Si used_colors[color] = Faux **alors**
            Sortir de la boucle;
        Finsi
        color ← color + 1 ;
    Fintantque
    result[i] ← color;
    Pour chaque j dans liste_suc[i] **faire**
        Si result[j] ≠ -1 **alors**
            used_colors[result[j]] ← Faux;
        Finsi
    Finpour
Finpour
Retourner result;

2. Algorithme Welsh_Powell

2.1.Langage algorithmique :

```
Fonction  isSafe (v : Entier, graph : Dictionnaire, colors : Tableau d'Entier, c : Entier)
Var i : Entier;
Debut
Pour chaque i dans graph[v + 1] faire
        Si colors[i - 1] = c Alors
                Retourner Faux;
        Finsi
Finpour
Retourner Vrai;
FinFonction
```

```
Algorithme WelshPowell
{Entrée : graph : Graphe sous forme de liste de successeurs}
{Sortie : colors : Tableau de couleurs attribuées aux sommets}
Var dico : Dictionnaire ; sorted_list : Dictionnaire ; i, c, vertex : Entiers
Debut
dico ← deMaTL(graph);
sorted_list ← Trier;
Si colors est None alors
        colors ← Tableau [1..longueur(dico)] * -1;
Finsi
Pour i de 1 à longueur (sorted_list) faire
vertex ← sorted_list[i].premier;
        Si colors[vertex - 1] = -1 alors
                Pour c de 1 à longueur dico) faire
                        Si isSafe(vertex - 1, dico, colors, c) alors
                                colors[vertex - 1] ← c;
                                Sortir de la boucle;
                Finsi
        Finpour
Finsi
Finpour
Retourner colors;
```

3. Algorithme Backtracking

3.1.Langage algorithmique :

```
Fonction  peutColorier (graph : Tableau d'Entiers, colors : Tableau d'Entiers,
sommet : Entier, c : Entier)
Var element : Entier;
Debut
Pour element dans graph[sommet + 1] faire
        Si colors[element - 1] = c alors
                Retourner Faux;
        Finsi
Finpour
Retourner Vrai;
FinFonction
```

```
Fonction  testerBackTraching (graph : Tableau d'Entiers, color_nb : Entiers,
colors : Tableau d'Entier, sommet : Entier)
Var c : Entier;
Debut
Si sommet = graph.n+1 alors
        Retourner Vrai;
FinSi
Si colors[sommet - 1] ≠ -1 alors
        Retourner TesterBackTracking(graph, color_nbr, colors, sommet+1);
Finsi
Pour c de 0 à color_nbr - 1 faire
        Si PeutColorier(graph, colors, sommet, c) alors
                colors[sommet - 1] ← c;
                Si TesterBackTracking(graph, color_nbr, colors, sommet+1)
                alors
                        Retourner Vrai;
                Finsi
                colors[sommet - 1] ← -1;
        Finsi
Finpour
Retourner Faux;
```

```
Procédure Backtracking (graph : Tableau d'Entiers, color_nbr : Entier,
Var colors : Tableau d'Entiers)
Var sommet : Entier;
Debut
Si colors = None alors
        colors ← -1 * graph.n ;
Finsi

sommet ← 1;
Si TesterBackTracking(graph, color_nbr, colors, sommet) = Faux alors
        Afficher "Impossible de trouver une solution";
        Retourner None;
Finsi
```

Retourner colors;
FinProcédure

4. Algorithme Coloration_Dynamique

Le fichier source “coloration_dynamique.py” permet implémenter une coloration dynamique de graphe, où chaque sommet ajouté au graphe est attribué d'une couleur tout en évitant les conflits chromatiques avec ses voisins. La fonction add_sommet assure la présence des couleurs nécessaires pour chaque sommet, vérifie l'existence des voisins dans le graphe, et attribue une couleur au sommet courant en réutilisant les couleurs déjà attribuées à ses voisins. Ainsi, à chaque ajout de sommet, le code cherche à minimiser le nombre de couleurs utilisées, rendant la coloration du graphe la plus efficace possible.

Structure du code

Notre code se divise en deux :

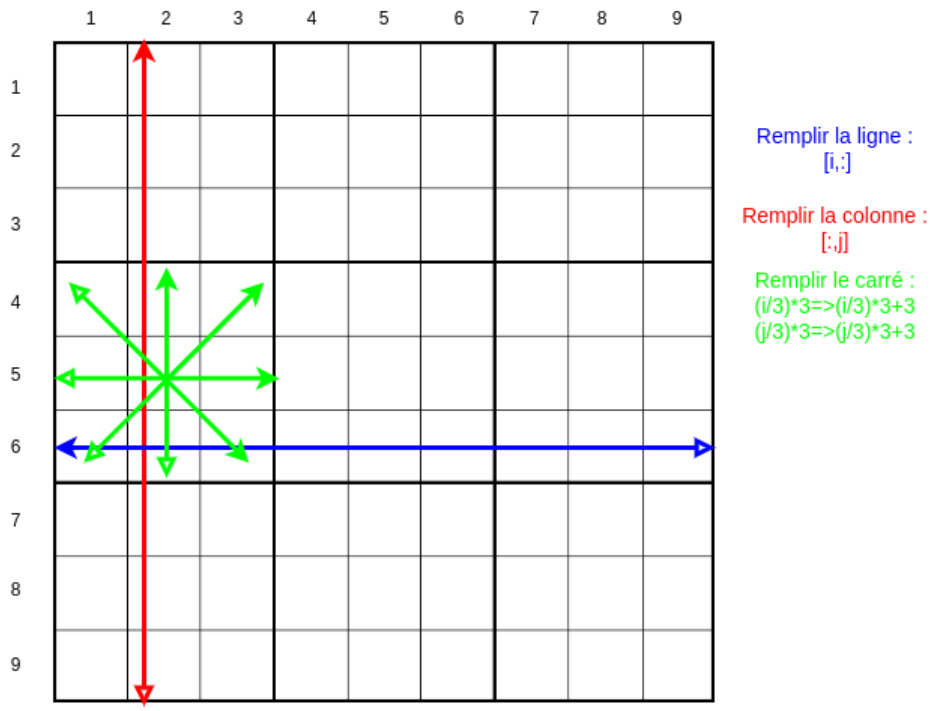
- Les fichiers se trouvant dans le dossier src :

Ce dossier comporte les implémentations des algorithmes vu précédemment tels que : Backtracking, Greedy et WelshPowell, ainsi qu'un fichier source "tools.py" qui contient différentes fonctions utilisées dans la manipulation des graphes : list_to_graph() qui transforme une liste en un graphe, liste_successeurs(graph) qui renvoie la liste des successeurs d'une matrice d'adjacence passée en paramètre ….

- Les fichiers se trouvant dans le répertoire principal :

L'un des fichiers les plus importants est le fichier source "sudoku.py", qui contient les fonctions create_sudoku_empty(taille) et solve_sudoku(graph), resp. une pour créer une grille de sudoku vide en créant le graphe correspondant (où chaque case est reliée avec sa ligne, colonne et carré) et une pour résoudre le sudoku à partir d'une grille vide ou partiellement remplie.

  - Pour la création du sudoku on doit, pour chaque case de la grille, qui représente en fait un sommet, ajouter une arête entre cette dernière et toutes les cases de la ligne, la colonne et le petit carré auquel la case appartient. Pour expliquer le fonctionnement, prenons le cas de la case [5,2] représentant le sommet 38 ⟨ (ligne-1) * taille + colonne ⟩:



La liste des successeurs de ce sommet sera équivalente à : [2, 11, 20, 28, 29, 30, 37, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 56, 65, 74]. Cet algorithme fera le même travail pour toutes les cases

afin de générer un graphe de 81 sommets qu'on pourra ensuite colorier pour trouver une solution.

- Pour la résolution, on a décidé d'utiliser l'algorithme de backtracking car les autres peuvent ne pas trouver la solution. On passe donc le graphe à la fonction solve_sudoku(graph), qui doit d'abord valider ce dernier, sinon elle renvoie une erreur. Ensuite on utilise le backtracking pour trouver la couleur (allant de 0 à $\sqrt{taille}$) des sommets sans changer les cases mises au préalable par l'utilisateur. Enfin, la liste des couleurs est transformée en un graphe pour pouvoir être affichée à l'utilisateur.

Le fichier "interface.py" contient tous les éléments nécessaires à l'affichage de la grille du sudoku ainsi que le choix de difficulté et les boutons de génération, résolution et effacement de la grille.

Conclusion

Ce projet nous a permis de découvrir comment les graphes et les algorithmes peuvent être utilisés pour résoudre des problèmes concrets comme le Sudoku. En appliquant la théorie des graphes, nous avons appris des méthodes utiles qui seront bénéfiques pour nos futurs projets informatiques.

Bibliographie

- Différents problèmes en théorie des graphes
- Coloration de graphe