Rapport de projet en Algorithme des graphes

Lucas Coussement

25 Novembre 2023

Résumé

Ce rapport présente le travail réalisé dans le cadre du projet d'Algorithme des graphes. Le but de ce projet est d'implémenter des algorithmes de coloriage de graphes en *Python*. Ces algorithmes vont ensuite être utilisés pour développer une application permettant de résoudre des grilles de *Sudoku*.

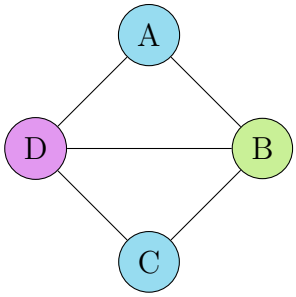
Sommaire

1 Introduction

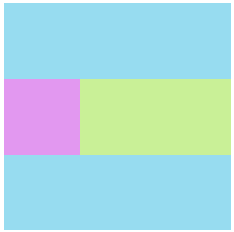
Avant de commencer à parler des algorithmes de coloriage de graphes, de leur implémentation et de Sudoku, nous allons commencer par définir ce qu'est la coloration d'un graphe et d'où vient ce problème.

1.1 Problème du coloriage des graphes

La *coloration d'un graphe* consiste à attribuer une couleur à chacun des sommets d'un graphe non orienté de manière que *deux sommets reliés par une arête soient de couleurs différentes*. On cherche souvent à minimiser le nombre de couleurs utilisées. On parle alors de *coloration minimale*.



Exemple de coloration d'un graphe



Représentation équivalente du graphe sous forme de grille

Ces exemples illustrent bien le fait que d'une carte de pays ou d'une grille, nous pouvons construire un graphe et rapporter le problème de la coloration aux graphes.

1.2 Origine

L'origine du problème de la coloration d'un graphe remonte en 1852, lorsque le mathématicien *Francis Guthrie* se demanda si *quatre couleurs suffisaient* à colorier une carte de pays de telle sorte que deux pays frontaliers soient de couleurs différentes. À cette époque, il s'interessait à la coloration de la carte d'Angleterre. Il fit part du problème à son professeur *Augustus De Morgan* qui enseignait à l'Université de Londres.

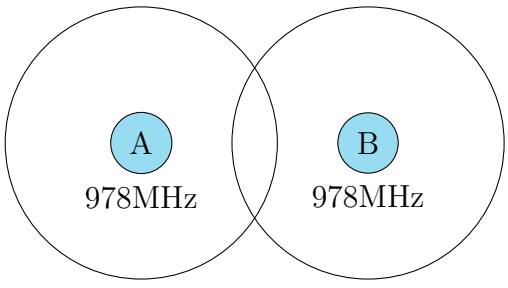
1.3 Preuve

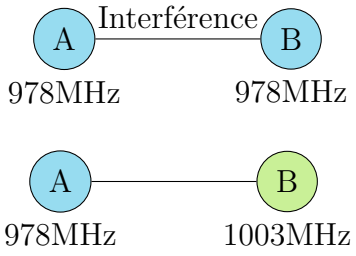
Les démonstrations connues de ce théorème décomposent le problème en un nombre de sous cas tellement important qu'il était nécessaire de faire appel à un ordinateur pour vérifier la validité de la preuve. Lorsqu'on généralise le problème à un graphe quelconque, il devient *NP-complet*. Cela signifie qu'il est *très difficile* de trouver une solution en un temps raisonnable.

Il aura fallu attendre 1976 pour que *Kenneth Appel* et *Wolfgang Haken* parviennent à démontrer le théorème des quatre couleurs. Ils ont utilisé un ordinateur pour vérifier la validité de leur preuve. L'ordinateur aura évalué *1478 cas critiques* et mis *1200 heures* pour réaliser la preuve.

1.4 Applications

La coloration des graphes est un problème complexe qui a de nombreuses applications dans la vie de tous les jours. Par exemple dans le domaine de la télécommunication, les fréquences utilisées par les antennes relais doivent être différentes pour éviter les interférences. On peut donc modéliser les antennes par des sommets, les fréquences utilisées par des couleurs et la proximité des antennes par des arêtes.





Exemple d'une représentation sous la forme d'un graphe d'une interférence entre deux antennes relais.

2 Les graphes dynamiques

Avant de commencer à implémenter et concevoir des algorithmes de coloration de graphe, il est nécessaire de définir sur quelle représentation de graphe nous travaillerons. Nous avons choisi de travailler sur des matrices d'adjacence car elles sont facile à manipuler et visuellement plus parlantes que les listes d'arêtes.

2.1 Graphe dynamiques

Les graphes dynamiques sont des graphes sur lesquelles nous pouvons effectuer certaines opérations qui changent leur structure. Ces opérations sont l'ajout et le retrait d'arrêtes entre deux sommets.

2.1.1 Ajout d'arrêtes

Soit un graphe $G = \langle S, A \rangle$ où S est un ensemble de sommets et A un ensemble d'arêtes entre les sommets de S. L'ajout d'une arrêtes entre deux sommets u de S revient à ajouter un élément à l'ensemble des arrêtes de G. On obtient alors un graphe $G' = \langle S, A' \rangle$ où A' est un ensemble d'arrêtes tel que $A \subseteq A'$ et $u \subset A'$.

Dans le cadre d'un graphe représenté par une matrice d'adjacence, l'ajout d'une arête entre deux sommets s_1 et s_2 revient à mettre à 1 la valeur à la ligne s_1 et à la colonne s_2 de la matrice d'adjacence.

2.1.2 Retrait d'arrêtes

Soit un graphe $G = \langle S, A \rangle$ où S est un ensemble de sommets et A un ensemble d'arêtes entre les sommets de S. Le retrait d'une arrêtes entre deux sommets u de S revient à retirer un élément à l'ensemble des arrêtes de G. On obtient alors un graphe $G' = \langle S, A' \rangle$ où A' est un ensemble d'arrêtes tel que $A' \subseteq A$ et $u \notin A'$.

Dans le cadre d'un graphe représenté par une matrice d'adjacence, le retrait d'une arête entre deux sommets s_1 et s_2 revient à mettre à 0 la valeur à la ligne s_1 et à la colonne s_2 de la matrice d'adjacence.

2.2 Implémentation des graphes dynamiques en python

Nous avons implémenté les graphes dynamiques en python dans une classe nommée *Matrice*. Cette classe contient toutes les opérations relatives aux graphes dynamiques. Elle contient également quelques opérations supplémentaires qui seront utiles pour la suite du projet. Nottement des fonctions permettant d'avoir des informations sur les prédécesseurs, les successeurs et les sommets non liée d'un sommet.

2.2.1 Matrices d'adjacence

Une matrice d'adjacence en python sera représentée par une matrice provenant du module *numpy*. Ce module est utile pour manipuler des matrices.

2.2.2 Prédécesseurs, successeurs et autres

Pour obtenir les prédécesseurs d'un graphe, il suffit de parcourir la matrice d'adjacence et de regarder les sommets qui ont une valeur à 1 à la colonne correspondant au sommet dont on cherche les prédécesseurs.

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

FIGURE 1 – Exemple d'une matrice d'adjacence

Dans l'exemple de la figure 1, le prédécesseurs du sommet 1 est le sommet 3 puisque la valeur à la colonne 1 et à la ligne 3 est à 1.

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

Pour obtenir les successeurs d'un graphe, il suffit de parcourir la matrice à la ligne correspondant au sommet dont on cherche les successeurs et de regarder toutes les colonnes qui ont une valeur à 1.

Toujours dans l'exemple du graphe de la figure 1, les successeurs du sommet 2 sont les sommets 2 et 3 puisque les valeurs à la ligne 2 des colonnes 2 et 3 sont à 1.

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

Pour les autres sommets non liés à un sommet, il suffit de lister tous les sommets qui ne sont ni des prédécesseurs ni des successeurs.

3 Les graphes coloré

Les graphes colorés sont des graphes dynamiques auxquels nous avons associé à tous ses sommets ou quelques uns une couleur.

3.1 Représentation d'un graphe coloré

Un graphe coloré est donc constitué d'un graphe dynamique, d'une liste associative associant à un sommet une couleur et une liste de couleurs.

3.2 Opérations sur les graphes colorés

Il est possible d'effectuer certaines opérations sur les graphes colorés. Ces opérations permettent d'altérer toutes les informations concernant les couleurs choisi pour les sommets. Un graphe coloré doit permettre de modifier les couleurs associées à un sommet, c'est pourquoi il doit contenir une opération qui affecte une couleur à un sommet et une autre qui retire la couleur d'un sommet. De plus il doit aussi être possible de modifier le nombre de couleurs disponibles, soit en ajoutant une couleur, soit en retirant une couleur.

3.2.1 Ajout et retrait de couleurs à un sommet

L'ajout d'une couleur à un sommet revient à ajouter le sommet concerné à la liste associative et de lui donner comme valeur la couleur spécifiée. Le retrait d'une couleur est l'opération inverse, c'est la suppression du sommet de la liste associative.

3.2.2 Ajout et retrait de couleurs disponibles

L'ajout d'une couleur disponible revient à ajouter la couleur à la liste de couleurs disponible. Nous partons du principe qu'il n'est pas possible d'avoir deux fois la même couleur, c'est pourquoi cette opération est invalide si la couleur existe déjà. Le retrait d'une couleur disponible consiste à supprimer de la liste de couleurs disponibles la couleur spécifiée.

3.3 Implémentation des graphes colorés en python

Les graphes coloré sont implémentés en python dans une classe nommée *Coloration*. Cette classe contient toutes les opérations décrites précédemment et quelques autres qui sont des requêtes sur les graphes colorés pour obtenir des informations plus efficacement.

Elle utilise une instance de la classe *Matrice* pour représenter le graphe coloré, un dictionnaire python pour les couleurs associées aux sommets et une liste pour les couleurs disponibles.

4 Les algorithmes de coloration

Les algorithmes de coloration dans le cadre du projet sont au nombre de trois. Il y a l'algorithme de coloration glouton, l'algorithme de coloration par backtracking et l'algorithme de welsh-powell.

4.1 Représentation des algorithmes

Les algorithmes sont rassemblés dans une même structure qui comportera un graphe coloré. Cette structure permettra l'application des algorithmes sur le graphe pour le colorer en suivant les principes de la coloration de graphe établis précédemment. Nous verrons plus tard que l'ordre d'application des algorithmes a une importance sur le résultat obtenu. Les algorithmes glouton et de welsh-powell s'organisent autour d'un ordre de parcours, c'est pourquoi la structure comportera un ordre de parcours sous la forme d'une liste de sommet.

4.2 Opérations relatives aux algorithmes

La structure de données contenant les algorithmes de coloration doit permettre d'appliquer les algorithmes sur le graphe coloré. Pour cela, elle doit contenir 3 opérations : une opération qui applique l'algorithme glouton, une opération qui applique l'algorithme de welsh-powell et une opération qui applique l'algorithme de backtracking. Nous ajouterons aussi une opération qui calculera l'ordre de parcours du graphe d'une manière spécifiée plus tard.

4.3 Implémentation de la structure

Cette structure se traduira par une classe python nommée *AlgoColo*. Cette classe comportera un graphe coloré qui sera une instance de la classe *Coloration* et un ordre de parcours qui sera une liste de sommets. L'implémentation des opérations relatives aux algorithmes sera détaillés par la suite.

5 L'algorithme glouton

Un algorithme glouton est un algorithme qui suit le principe de réaliser, étape par étape, un choix optimal local pour trouver une solution qui se rapproche le plus possible d'un résultat optimal global.

L'algorithme glouton de la coloration d'un graphe permet de résoudre le problème de la coloration minimale d'un graphe en utilisant une approche "gloutonne".

5.1 Procédé

L'algorithme est constitué d'un parcours du graphe avec un ordre donnée en paramètre. Pour chaque sommet parcouru, on cherche la couleur avec la plus petite valeur qui n'est pas déjà utilisé par un voisins [1] du sommet choisi. Si aucune couleur n'est disponible, nous ajoutons une couleur à la liste des couleurs disponibles et nous l'attribuons au sommet. A la fin du parcours du graphe, nous obtenons un graphe coloré correctement.

5.2 Inconvénients

L'algorithme glouton n'est par définition pas optimal. Il peut arriver qu'il utilise plus de couleurs que nécessaire pour colorer un graphe. Cela peut s'avérer problématique dans le cadre du Sudoku puisque nous cherchons à avoir le moins de couleurs possible pour résoudre une grille.

Un autre problème de l'algorithme glouton est qu'il est sensible à l'ordre de parcours du graphe. En effet, si l'ordre de parcours n'est pas optimal, il est possible dans le pire des cas qu'il utilise autant de couleur qu'il n'y a de sommets dans le graphe.

La selection de l'ordre optimal de parcours est un problème NP-complet, cela signifie que dans la plupart des cas, il est impossible de trouver un ordre optimal en un temps raisonnable.

5.3 Solutions

Comme il est difficile de trouver le bon ordre, nous pouvons utiliser une heuristique pour tenter de réduire au maximum le nombre de couleur.

1. Les voisins d'un sommets sont tous les sommets qui sont soit successeurs, soit prédécesseurs du sommet.

Cette heuristique consiste à trier les sommets du graphe par ordre décroissant de degré. Le degré d'un sommet est son nombre de voisins. Cette heuristique est réalisable en un temps raisonnable puisqu'il suffit de parcourir les sommets pour calculer leur nombre de voisin et de trier les sommets par ordre décroissant de leur voisins. Si l'ordre de parcours est celui des sommets triés, l'algorithme glouton utilisera dans le pire des cas $n/2$ couleurs avec n le nombre de sommet du graphe.

5.4 Implémentation

L'algorithme glouton est implémenté dans la classe *AlgoColo* dans une méthode nommée *color_with_glouton*. Cette méthode ne prend aucun parmaètre puisqu'elle profite simplement du contexte de la classe pour récupérer le graphe coloré et l'ordre de parcours. Elle ne fait que modifier les attributs de la classe pour colorer le graphe.

Algorithme 1 : Algorithme glouton

Entrées : Un graphe G de type $GrapheM$; Un ordre O de type $Tableau[1 \dots n]$; Un dictionnaire *colo_dico* ; Une liste *colo_list*
Sorties : Un dictionnaire *colo_dico*
Variables : i, j : Entier ; $succ$: $Tableau[1 \dots n]$
début
  si $est\_vide(O)$ **alors**
    $O \leftarrow ordre(G)$
  fin
  pour i *de* 1 *à* $O.n$ **faire**
    pour j *de* 1 *à* $successors(G, O[i]).n$ **faire**
      $ajouter(succ, successors(G, O[i])[j])$
    fin
    pour j *de* 1 *à* $successors(G, O[i]).n$ **faire**
      $ajouter(succ, predecessors(G, O[i])[j])$
    fin
    $colo\_dico[O[i]] \leftarrow first\_not\_in\_list(succ, colo\_list)$
  fin
fin

$first\_not\_in\_list$ est une fonction qui prend en paramètre une liste de

couleur, une liste des couleurs disponible et qui retourne la première valeur (dans la liste de couleur disponible) qui n'est pas dans la liste passée en paramètre si elle existe, sinon qui génère une couleur, qui l'ajoute à la liste de couleur et qui la retourne.

ordre est une fonction qui prend en entrée un graphe et qui retourne un ordre de parcours du graphe décroissant de degré.

Algorithme 2 : *first_not_in_list*

```
Entrées : Une liste de couleur succ ; Une liste de couleur colo_list
Sorties : Une couleur c
Variables : i, j : Entier ;
début
    pour i de 1 à colo_list.n faire
        si non est_dans(colo_list[i], succ) alors
            retourner colo_list[i]
        fin
    fin
    retourner generer_couleur(colo_list)
fin
```

generer_couleur est une fonction qui prend en paramètre une liste de couleur et qui génère une couleur qui n'est pas dans la liste de couleur et qui l'ajoute à la liste de couleur et qui la retourne.

est_dans est une fonction qui prend en paramètre une couleur et une liste de couleur et qui retourne vrai si la couleur est dans la liste de couleur, sinon faux.

5.5 Complexité

Nous avons une boucle principale qui parcours tous les sommets du graphe. Dans cette boucle, nous avons deux boucles imbriquées qui parcourent les successeurs et les prédécesseurs du sommet courrant. Ces deux boucles ont une complexité de $O(n)$ où n est le nombre de sommets du graphe. La boucle principale a une complexité de $O(n)$. La fonction *first_not_in_list* a une complexité de $O(n)$ dans le pire des cas. Nous avons donc une complexité de $O(n \times 3 \times n) = O(n^2)$ pour le corps du programme. La fonction *ordre* est une fonction qui a une complexité de $O(n^2)$ dans le pire des cas. Nous avons donc une complexité finale de $O(2 \times n^2) = O(n^2)$ dans le pire des cas.

6 L'algorithme de welsh-powell

L'algorithme de welsh-powell est une des heuristique qui ont été conçues pour faire face au problème de la coloration de graphe.

6.1 Procédé

Pour résoudre le problème, l'algorithme commence par trier les sommets du graphe par ordre décroissant de degré, comme pour l'amélioration de l'algorithme glouton. Ensuite, il parcourt les sommets dans l'ordre obtenu. Il attribut une couleur au premier sommet et il attribut cette même couleur à tous les sommets qui ne sont pas voisins du sommet courant et qui n'ont pas encore de couleur et ce jusqu'à arriver à la fin de l'ordre.

6.2 Inconvénients

Comme pour l'algorithme glouton, l'algorithme de welsh-powell ne donne pas une solution optimale. Il peut arriver qu'il utilise plus de couleurs que nécessaire pour colorer un graphe. L'algorithme de welsh-powell dans le pire des cas utilise $n/2$ couleurs pour colorer un graphe, n étant le nombre de sommets du graphe.

L'algorithme de welsh-powell est aussi incompatible avec les couleurs déjà définies avant son appel. En effet, il ne prend pas en compte les couleurs déjà attribuées aux sommets du graphe puisqu'il a besoin de les attribuer lui même pour fonctionner.

6.3 Implémentation

L'algorithme de welsh-powell est implémenté dans la classe *AlgoColo* dans une méthode nommée *color_with_welsh_powell*. Cette méthode ne prend aucun paramètre puisqu'elle profite du contexte de la classe pour récupérer le graphe coloré. Elle ne fait que modifier les attributs de la classe pour colorer le graphe.

Algorithme 3 : Algorithme welsh-powell

Entrées : Un graphe G de type $GrapheM$; Un dictionnaire $colo\_dico$; Une liste $colo\_list$
Sorties : Un dictionnaire $colo\_dico$
Variables : $i, k, color$: Entier ; $O : Tableau[1 \dots n]$
début
  $reset(colo\_dico)$
  $O \leftarrow ordre(G)$
  $color \leftarrow 1$
  pour i *de* 1 *à* $O.n$ **faire**
    si $colo\_dico[O[i]] = Nil$ **alors**
      si $colo\_list.n \leq color$ **alors**
        $generer\_color(colo\_list)$
      fin
      $colo\_dico[O[i]] \leftarrow color\_list[color]$
    fin
    pour k *de* 1 *à* $autres\_sommets(G, O[i])$ **faire**
      si $colo\_dico[autres\_sommets(G, O[i])[k]] = Nil$ **alors**
        $colo\_dico[autres\_sommets(G, O[i])[k]] \leftarrow color$
      fin
    fin
    $color \leftarrow color + 1$
  fin
fin

autres_sommets est la fonction qui prend en paramètre un graphe et un sommet et qui renvoie la liste des sommets qui ne sont pas voisins du sommet passé en paramètre. Cette fonction a une complexité de $O(n)$ où n est le nombre de sommets du graphe.

reset est une fonction qui prend en paramètre un dictionnaire et qui met à Nil toutes les valeurs du dictionnaire. Cette fonction a une complexité de $O(n)$ où n est le nombre de sommets du graphe.

6.4 Complexité

Nous avons une boucle principale qui parcours tous les sommets du graphe. Dans cette boucle, nous avons une boucle imbriquées qui parcourt les sommets qui ne sont pas voisins du sommet courrant. Cette boucle a une com-

plexité de $O(n)$ où n est le nombre de sommets du graphe. La boucle principale a une complexité de $O(n)$. Nous avons donc une complexité finale de $O(n \times n) = O(n^2)$

7 L'algorithme Backtrack

L'algorithme de backtracking est une algorithme de recherche qui consiste à explorer toutes les solutions possibles d'un problème en construisant incrémentalement des candidats à la solution. Lorsqu'il trouve une solution, il l'enregistre et continue à explorer les autres solutions possibles. Si aucune solution n'est trouvée, il revient en arrière et explore une autre solution.

L'algorithme de backtracking pour la coloration d'un graphe permet de trouver la solution optimale du problème de la coloration minimale d'un graphe.

7.1 Procédé

L'algorithme de backtracking fonctionne avec un nombre fini de couleurs. Il commence par choisir un sommet et une couleur. Il teste s'il est possible d'attribuer cette couleur à ce sommet. Si c'est le cas, il colorie le sommet avec la couleur, il teste si tous les sommets sont coloriés. Si c'est le cas, c'est la fin de l'algorithme, sinon il choisit un autre sommet et la plus petite couleur disponible et il recommence.

Dans le cas où il n'est pas possible d'attribuer la couleur au sommet, il change de couleur et recommence. Si toutes les couleurs ont été testées, il revient en arrière et change la couleur du sommet précédent et recommence. Si toutes les couleurs ont été testées pour tous les sommets, c'est la fin de l'algorithme, cela signifie qu'il n'y a pas de solution.

7.2 Inconvénients

De prime abord, l'algorithme de backtracking pourrait sembler être une solution parfaite au problème de la coloration de graphe. Cependant, il est très couteux en temps de calcul. En effet, il doit tester toutes les combinaisons possibles de couleurs pour tous les sommets.

7.3 Implémentation

L'algorithme de backtracking est implémenté dans la classe *AlgoColo* dans une méthode nommée *color_with_backtrack*.

Algorithme 4 : Algorithme backtracking

Entrées : Un graphe G de type $GrapheM$; Un dictionnaire $colo\_dico$; Une liste $colo\_list$
Sorties : Un dictionnaire $colo\_dico$
Variables : $i, k, color$: Entier ;
début
  $i \leftarrow 0$
  while *non* $backtrack(0, 0, G, colo\_dico, color\_list)$ *et* $i < G.n$ **do**
    $i \leftarrow i + 1$
    $generate\_color(colo\_list)$
  end
fin

La fonction *backtrack* étant la fonction auxiliaire de l'algorithme de backtracking prenant en paramètre un entier $actualColor$ et un entier $actualNode$ qui correspondent respectivement à la couleur et au sommet choisi pour l'itération courante.

Algorithme 5 : backtrack

Entrées : Un entier $actualColor$; Un entier $actualNode$; Un graphe G de type $GrapheM$; Un dictionnaire $colo\_dico$; Une liste $colo\_list$

Sorties : Un booléen

Variables : $i, k, color$: Entier ;

début

  si $actualColor = colo\_list.n$ **alors**

    si $actualNode = 0$ **alors**

      $reset(colo\_dico)$

      retourner $faux$

    fin

    sinon

      $color = color\_index(colo\_dico[actualNode - 1], colo\_list)$

      $colo\_dico[actualNode - 1] \leftarrow Nil$

      retourner $backtrack(color + 1, actualNode - 1, G, colo\_dico, color\_list)$

    fin

  fin

  si $actualNode = G.n$ **alors**

    retourner $vrai$

  fin

  si $colo\_dico[actualNode] \neq Nil$ **alors**

    retourner $backtrack(0, actualNode + 1, G, colo\_dico, color\_list)$

  fin

  si $is\_coloriable(actualNode, actualColor, G, colo\_dico)$ **alors**

    $colo\_dico[actualNode] \leftarrow color\_list[actualColor]$

    retourner $backtrack(0, actualNode + 1, G, colo\_dico, color\_list)$

  fin

  sinon

    retourner $backtrack(actualColor + 1, actualNode, G, colo\_dico, color\_list)$

  fin

fin

is_coloriable est une fonction qui prend en paramètre un sommet, une couleur, un graphe et un dictionnaire pour déterminer si le sommet peut être

colorié avec la couleur spécifiée. Cette fonction a une complexité de $O(n)$ où n est le nombre de sommets du graphe. *color_index* est une fonction qui prend en paramètre une couleur et une liste de couleur et qui retourne l'index de la couleur dans la liste de couleur. Cette fonction a une complexité de $O(n)$ où n est le nombre de couleur de la liste de couleur.

7.4 Complexité

Dans le pire des cas, l'algorithme de backtracking doit tester toutes les combinaisons possibles et effectue donc un nombre d'appel récursif égal au nombre de sommet multiplié par le nombre de couleur minimale qu'il faut pour colorier le graphe. Nous avons donc une complexité en nombre d'appel récursif de $O(n^2)$ où n est le nombre de sommets. La complexité d'une telle situation en temps de calcul pour tester toutes les combinaisons possible seraient de $O(n^m)$ où m est le nombre de couleur minimale qu'il faut pour colorier le graphe. Le pire des cas seraient que $m = n$ donc nous aurions une complexité en temps de calcul de $O(n^n)$.

8 Sudoku

Le Sudoku est un jeu de réflexion qui consiste à remplir une grille de 9x9 cases avec des chiffres allant de 1 à 9. La grille est divisée en 9 sous-grilles de 3x3 cases appelées régions. Le but du jeu est de remplir toutes les cases de la grille en prenant en compte les contraintes suivantes :

— Chaque ligne doit contenir tous les chiffres de 1 à 9.
— Chaque colonne doit contenir tous les chiffres de 1 à 9.
— Chaque région doit contenir tous les chiffres de 1 à 9.

Le problème de la résolution d'un sudoku est un problème de coloration de graphe où les sommets sont les cases de la grille, les couleurs sont les chiffres de 1 à 9 et les arêtes sont les contraintes.

Pour créer un sudoku à partir d'une grille, il suffit de créer un graphe coloré où il y a $9 \times 9 = 81$ sommets, 9 couleurs et des arêtes entre les sommets qui sont dans la même ligne, dans la même colonne et dans la même région.

On pourrait même aller plus loin en disant que pour avoir un sudoku valide, il faut que la dimension de la grille soit un carré parfait. En effet, si la dimension de la grille n'est pas un carré parfait, il est impossible de créer une grille de sudoku valide. Cela est dû au fait qu'une grille de sudoku de

dimension n, contient $\sqrt{n} \times \sqrt{n}$ régions. Si $\sqrt{n}$ n'est pas un entier, il sera impossible de créer la grille.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

FIGURE 2 – Exemple d'une grille de sudoku de dimension 4 avec le graphe correspondant

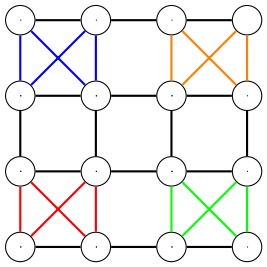


FIGURE 3 – Exemple d'un graphe correspondant à une grille de sudoku de dimension 4

8.1 Implémentation

Le sudoku est implémenté dans une classe nommée *SudokuModel*. Cette classe est constitué d'une matrice de taille $n \times n$ où n est un carré parfait, d'une instance de la classe *AlgoColo*. Cette classe contient toutes les opérations relatives au sudoku. Elle contient des opération qui permettent de générer un sudoku, de le résoudre etc... Un sudoku comporte des régions, ces régions sont des sous-grilles de taille $\sqrt{n} \times \sqrt{n}$. Il y a donc un tableau associatif qui associe à chaque région un ensemble de sommets.

8.2 Classement des sommets

Il est important de connaître les sommets qui sont dans la même région pour pouvoir construire le graphe du sudoku. Pour cela, nous avons besoin d'une fonction initialisant le tableau associatif qui associe à chaque région un ensemble de sommet.

Pour cela nous devons trouver une formule qui permet de calculer à partir d'un sommet, la région à laquelle il appartient.

Soit r l'ensemble des régions $\{x_0, \ldots, x_j, x_{j+1}, \ldots, x_{n-1}\}$, et n la dimension

$$\forall x_j \in r, x_j = \{(i \bmod \sqrt{n}) + (j \times \sqrt{n}) + (\lfloor \frac{i}{\sqrt{n}} \rfloor) \times n + (\lfloor \frac{j}{\sqrt{n}} \rfloor) \times n \mid \forall i \in \mathbb{N}, i < n\}$$

Faisons un exemple avec une grille de sudoku de dimension 4. Caclulons la région x_2. Nous illustrerons tous les $i < n$ dans x_2 sous la forme d'une matrice. Cette matrice aura pour dimension $\sqrt{n}$. Pour rappel dans notre cas précis $n = 4$ et $\sqrt{n} = 2$.

Exemple de l'ensemble x_2 si $x_2 = \{i \mid \forall i \in \mathbb{N}, i < n\}$

$$\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}$$

Maintenant si nous ajoutons $i \bmod \sqrt{n}$ (c'est à dire que nous avons $x_2 = \{i \bmod \sqrt{n} \mid \forall i \in \mathbb{N}, i < n\}$), nous obtenons :

$$\begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix}$$

Si nous additionnons à tout ça $j \times \sqrt{n}$, nous avons $x_2 = \{(i \bmod \sqrt{n}) + (2 \times \sqrt{n}) \mid \forall i \in \mathbb{N}, i < n\}$

$$\begin{pmatrix} 4 & 5 \\ 4 & 5 \end{pmatrix}$$

Nous additionnons $(\lfloor \frac{i}{\sqrt{n}} \rfloor) \times n$, ce qui nous donne
$x_2 = \{(i \bmod 2) + (2 \times 2) + (\lfloor \frac{i}{2} \rfloor) \times 4 \mid \forall i \in \mathbb{N}, i < n\}$

$$\begin{pmatrix} 4 & 5 \\ 8 & 9 \end{pmatrix}$$

Et pour finir nous ajoutons $(\lfloor \frac{j}{\sqrt{n}} \rfloor) \times n$ ce qui nous donne
$x_2 = \{(i \bmod 2) + (2 \times 2) + (\lfloor \frac{i}{2} \rfloor) \times 4 + (\lfloor \frac{2}{2} \rfloor) \times 4 \mid \forall i \in \mathbb{N}, i < n\}$

$$\begin{pmatrix} 8 & 9 \\ 12 & 13 \end{pmatrix}$$

Cela correspond bien à la région numéro 2 du graphe de dimension 4.

Maintenant que nous avons la formule pour calculer la région à partir d'un sommet, nous pouvons trouver la formule qui permet de calculer tous les sommets qui sont dans la même ligne ou dans la même colonne.

Soit L l'ensemble des arêtes de la grille de sudoku qui sont généré par les sommets qui sont sur la même ligne et n la dimension

$$\forall a \in L, a = \{\{in + j, in + k\} \mid \forall i \forall j \forall k \in \mathbb{N}^3, i < n, j < n, k < n\}$$

Soit C l'ensemble des arêtes de la grille de sudoku qui sont généré par les sommets qui sont sur la même colonne et n la dimension

$$\forall a \in C, a = \{\{jn + i, kn + i\} \mid \forall i \forall j \forall k \in \mathbb{N}^3, i < n, j < n, k < n\}$$

Soit R l'ensemble des arêtes de la grille de sudoku, r l'ensemble des régions $\{x_0, \ldots, x_j, x_{j+1}, \ldots, x_{n-1}\}$ et n la dimension

$$\forall a \in R, a = \{\{j, k\} \mid \forall i \in \mathbb{N}, i < n, j \in x_i, k \in x_i, j \neq k\}$$

L'ensemble des arêtes de la grille de sudoku est $A = L \cup C \cup R$. Il suffit donc de calculer A et de l'ajouter au graphe pour avoir un graphe correspondant à une grille de sudoku.

8.3 Résolution et changement de dimension

Dans le but de rendre le sudoku dynamique et de pouvoir faire des tests, nous avons implémenté une fonction qui permet de changer la dimension du sudoku en le réinitialisant. Cette fonction prend en paramètre une dimension qui doit être un carré parfait. Elle réinitialise la matrice du sudoku et le graphe coloré. Elle génère ensuite un nouveau sudoku de la dimension spécifiée.

Pour résoudre le sudoku, tous les algorithmes sont disponibles. Il suffit de choisir l'algorithme que l'on souhaite utiliser et de l'appliquer sur le graphe lié au sudoku.

Nous avons également rajouté une fonction de résolution partielle qui permet de résoudre aléatoirement un nombre de case spécifiée en paramètre. Cette fonction permet de créer un sudoku et de pouvoir gérer la difficulté de celui ci en fonction du nombre de case résolue. Pour cela, il suffit de résoudre entièrement le sudoku, de garder un nombre de case résolue prises au hasard et de réinitiliser la coloration du sudoku. On peut également mélanger les couleurs pour rendre chaque sudoku unique.

8.4 Interface graphique

Afin de pouvoir tester le sudoku, nous avons créé une interface graphique permettant de générer un sudoku, de le résoudre, de modifier sa dimension et de le résoudre partiellement selon un pourcentage de case résolue. Pour se faire nous avons utilisé la bibliothèque graphique Tkinter de python. Le choix de l'algorithme utilisé pour résoudre le sudoku (partiellement ou non) est l'algorithme backtrack. Nous avons choisi cet algorithme car il est le seul à pouvoir résoudre un sudoku de dimension 9 à coup sûr. Il arrive que les autres algorithmes ne puissent pas résoudre un sudoku de dimension 9 avec seulement 9 couleurs. Cela est dû au fait que les algorithmes glouton et welsh-powell ne sont pas optimaux et qu'ils peuvent utiliser plus de couleurs que nécessaire pour résoudre un sudoku. De plus l'inconvénients de l'algorithme backtrack qui est sa complexité en temps n'est pas un problème dans le cadre d'un sudoku de dimension inférieur ou égal à 9. Cela dit l'algorithme glouton est capable de résoudre un sudoku de dimension 4 avec seulement 4 couleurs. Il pourrait donc lui aussi être utilisé pour résoudre un sudoku de dimension 4. L'algorithme de welsh-powell par contre, est capable de résoudre un sudoku de dimension 4 cependant il n'est pas compatible avec les couleurs déjà définies avant son appel. Il ne peut donc pas être compatible avec la résolution partiel. En fin de compte l'algorithme backtrack est le plus polyvalent et le plus adapté dans notre situation.

L'interface graphique est implémenté dans une classe nommée *Sudoku*. Elle est constitué d'une instance de la classe *SudokuModel* et de tous les composants graphiques majeurs nécessaire à l'interface graphique (les boutons, la grille, etc...). La grille peut être d'une dimension correspondant à

tous les carré parfait entre 4 et 16.

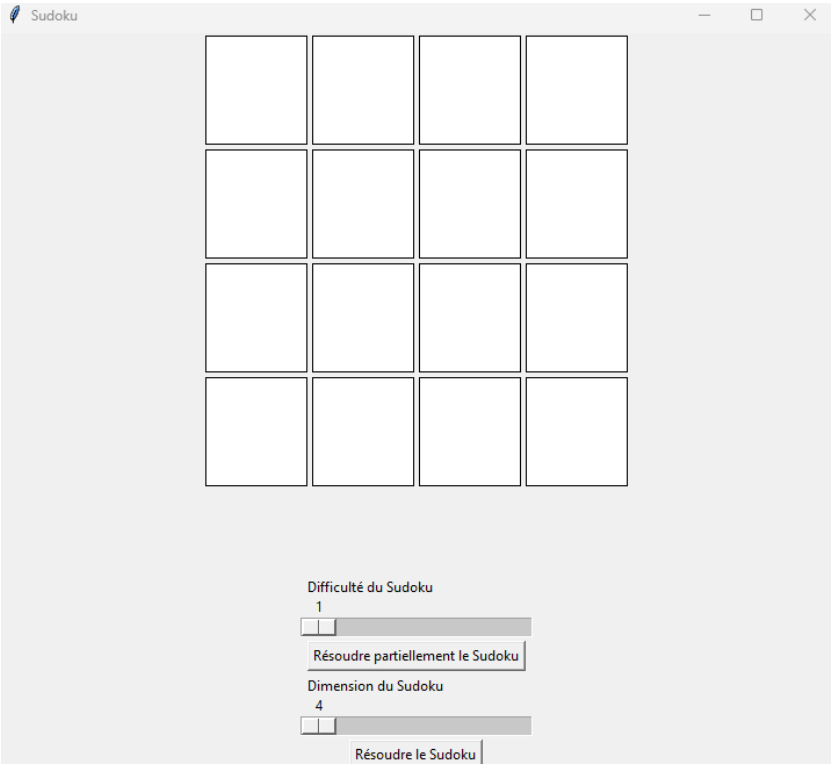


FIGURE 4 – Interface graphique pour un sudoku de dimension 4 vide

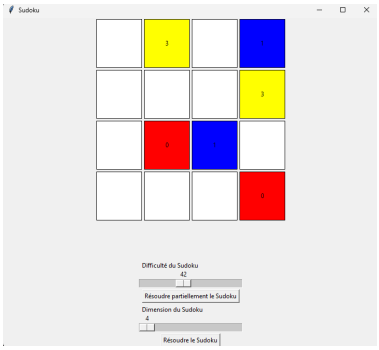


FIGURE 5 – Interface graphique pour un sudoku de dimension 4 partiellement résolue

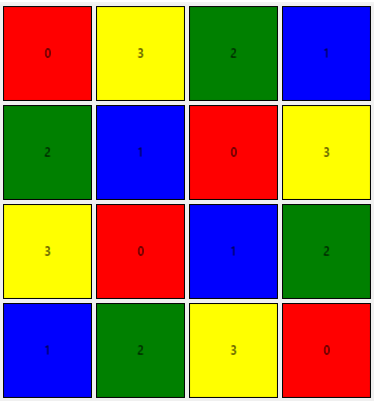


FIGURE 6 – Sudoku de dimension 4 résolue

Chaque couleur représente un chiffre du sudoku. En variant le bouton *scale*, nous pouvons altérer le pourcentage de case révélée.

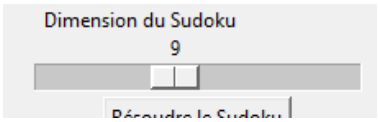


FIGURE 7 – Bouton pour changer la dimension du sudoku

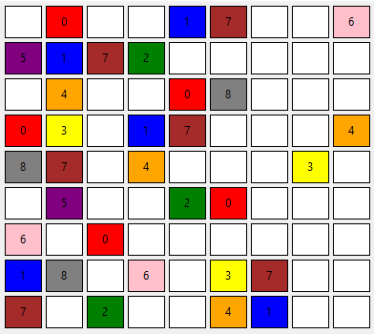


FIGURE 8 – Sudoku de dimension 8 partiellement résolue

Lorsque l'utilisateur rentre avec le bouton *scale* une dimension correcte pour le sudoku, le sudoku est réinitialisé et un nouveau sudoku de la dimension spécifiée est généré. Il est possible d'avoir un sudoku de dimension 16 au maximum. Au delà de cette dimension, le temps de calcul est trop long.

9 Difficulté

Cette partie est dédiées aux problèmes que nous avons pu rencontrer lors de la réalisation de ce projet.

Le seul problème que nous avons rencontré est le problème du débordement de pile lors de l'utilisation de l'algorithme de backtracking. En effet, à partir d'un certain nombre de sommet et de couleur, l'algorithme de backtracking récursif est tout simplement inutilisable puisqu'il fait déborder la pile. Et il ne faut pas attendre des dimensions très élevé pour que cela se produise. En effet par exemple pour un sudoku de dimension 16, il y a 256 sommets et 16 couleurs. Cela fait donc $16 \times 256 = 4096$ appels récursifs au pire des cas. La pile en python est limité à 1000 appels récursifs par défaut. Nous pouvons augmenter cette limite certes mais ce n'est pas une solution viable. A la place pour régler ce problème, nous avons dérécursifé l'algorithme de backtracking en utilisant une pile de sommet et de couleur. Cette amélioration nous permet d'avoir pour la majorité des cas comme seule limite le temps de calcul.

10 Sitographie

— Francis Guthrie
— Théorème des quatre couleurs
— Coloration de graphe
— Coloration gloutonne
— Graph Coloring Problem