Problème de coloration d'un graphe

MU5MAI04 : Ingénierie II, Approfondissement C++

GOPINATHAN Prédive
KRSTEVSKA Jovana

Master 2 : Ingénierie Mathématique
Ingénierie mathématique pour l'entreprise

19 janvier 2021

Résumé

Dans ce projet on nous a été demandé de créer un algorithme de *backtracking* pour le problème de coloration d'un graphe *non-orienté*, *connexe*, et puis de l'appliquer à la résolution des grilles de *Sudoku*.

Nous avons procédé par d'abord implémenter une première solution, naïve, mais assez intuitive et simple à implémenter. Nous avons essayé de l'optimiser en trouvant une meilleure façon de stocker les sommets adjacents de chaque sommet du graphe.

Ensuite, nous avons implémenté un deuxième algorithme de *backtracking*, pour lequel l'idée est similaire mais il consiste d'une amélioration importante : choisir les sommets suivants en fonction du nombre de couleurs disponibles pour sa coloration : on commence par celui qui a le moins de couleurs disponibles.

Finalement, nous avons comparé les résultats de nos deux algorithmes sur des grilles de *Sudoku* de difficulté différente, croissante, en fonction du nombre de *clues* (entrées initiales dans la grille).

Table des matières

1 Problématique et approche

1.1 Coloration d'un graphe

En théorie des graphes, la coloration d'un graphe *non-orienté*, *connexe* consiste à attribuer une couleur à chacun de ses sommets de manière que deux sommets reliés par une arête soient de couleur différente. On cherche souvent à utiliser le nombre minimal de couleurs, appelé *nombre chromatique*.

Soit $\mathcal{G}$ un graphe à N sommets et K un entier. Une coloration de $\mathcal{G}$ peut être vu comme une application :

$$f : \{1, 2, ..., N\} \mapsto \{1, 2..., K\}$$

que l'on peut considérer unique à permutation des indices de couleurs près.

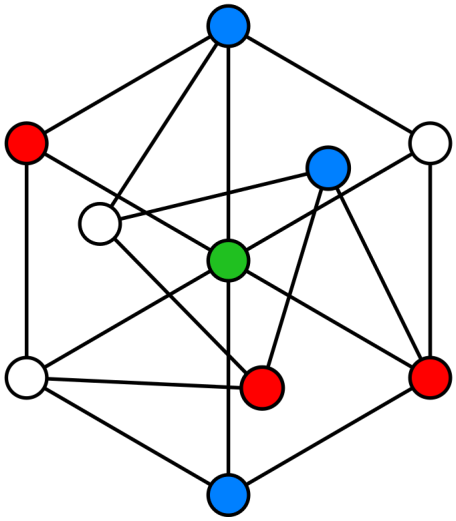


FIGURE 1 – Exemple : Graphe de Colomb

Deux problèmes algorithmiques de coloration sont les suivants :

- **problème de décision** : étant donné $\mathcal{G}$, un graphe, et K un entier, existe-t-il une *coloration valide* f de $\mathcal{G}$ utilisant K couleurs ?
- **problème d'optimisation** : étant donné $\mathcal{G}$, quel est son *nombre chromatique* ?

Dans ce projet on ce concentre sur le problème de décision et on se donnera un entier K, nombre de couleurs à utiliser.

1.2 Algorithme de backtracking

Le principe de l'algorithme de *backtracking* est assez intuitif.

Une façon simple pour comprendre la démarche de cet algorithme est de penser à comment on trouverait le chemin de sortie dans un *labyrinthe*. Cet algorithme est en fait un **parcours en profondeur** du graphe. L'algorithme commence par un sommet S. Il poursuit alors un chemin dans le graphe jusqu'à un cul-de-sac ou alors jusqu'à atteindre un sommet déjà visité. Il revient alors sur le dernier sommet où on pouvait suivre un autre chemin puis explore un autre chemin. L'exploration s'arrête quand tous les sommets depuis S ont été visités. L'exploration progresse donc, à partir d'un sommet

S en s'appelant *récursivement* pour chaque sommet voisin de S.

Dans notre cas la démarche est similaire. On prend un sommet du graphe et on le colore d'une couleur. Ensuite on essaye de continuer de colorer les autres sommets en attribuant une couleur au sommet suivant (récursivement), jusqu'à ce qu'on finisse la coloration ou bien qu'on se rende compte qu'on ne peut plus colorer le graphe à cause d'un mauvais choix de couleur précèdent. Dans le dernier cas, il faudrait revenir en arrière pour choisir une couleur différente au dernier choix qu'on a fait. Et si ceci ne permet toujours pas d'obtenir une coloration valide, on revient encore un appel en arrière. A la fin, soit on arrive à colorer tout le graphe avec les K couleurs données, soit on conclut qu'il n'y ait aucune coloration possible à K couleurs.

La complexité dépendra de la façon dont on choisit les sommets pour la récursion, mais elle est **exponentielle** dans tous les cas. La première approche sera de parcourir tous les sommets de 1 à N (l'algorithme naïf), et dans la deuxième on choisira le sommet suivant et les couleurs à essayer pour ce sommet de manière plus optimale.

2 Implémentation en C++

Pour implémenter cet algorithme on dispose des classes ***Arc***, pour construire les arcs d'un graphe et ***Graphe***, pour construire les graphes.
Dans tout ce qui suit, on gardera les notations suivantes :

- *N* — nombre de sommets dans le graphe
- *vois* — un vecteur de **N** vecteurs, dans lesquels sont stockés les voisins pour chaque sommet, donc vois[0] est un vecteur qui contient les voisins du sommet numéroté 0
- *assigned_colors* — vecteur de taille **N**, contenant les couleurs assignées aux sommets

2.1 Algorithme de backtracking simple : GC

Dans cette première approche, on parcourt tous les sommets, sans aucune préférence de choix. On a les méthodes suivantes.

• **good_to_take** : méthode qui prend le noeud *node*, la couleur *color* et le vecteur *assigned_colors* en arguments d'entrée. Elle teste si *node* a des sommets voisins qui sont colorés en *color* et s'il n'y en a pas elle renvoie **true**, c'est-à-dire on peut attribuer la couleur *color* au noeud *node*. Sinon elle renvoie **false**.

```
bool good_to_take(int node, int color, int K, std::vector<int> &assigned_colors){
    assert(node < N && node >= 0);
    assert(color <= K && color > 0);
    for (int neighbour : vois[node]) {
        // si la couleur est prise par les voisins
        if (assigned_colors[neighbour] == color) return false;
    }
    //sinon, color est une bonne couleur à prendre\n";
    return true;
}
```

• **try_graph_coloring** et **graph_coloring** : Pour mieux les expliquer, supposons qu'on appelle la méthode **graph_coloring** pour un graphe connexe quelconque.

1. On entre directement dans la méthode **try_graph_coloring** dans laquelle on essaye de colorer le graphe en itérant les noeuds, sans préférence de choix, en commençant par 0.

2. En itérant sur toutes les couleurs possibles, pour chacune on vérifie si elle peut être prise par le noeud *node* grace à la méthode **good_to_take**. Si c'est le cas, on met à jour le vecteur *assigned_colors*. Sinon, on continue en prenant la couleur suivante.
   Si aucune couleur ne fonctionne on renvoie **false** dans le premier appel de **try_graph_coloring**, donc on affiche que la solution n'existe pas dans **graph_coloring** et on sort complètement.

3. Après avoir choisi une couleur et avoir mis à jour le vecteur *assigned_colors* on commence les appels récursifs de **try_graph_coloring** avec le noeud 1, 2 ... et ainsi de suite.

   – Si cet appel récursif renvoie **true** alors on a réussi à colorer le graphe et on renvoie **true** dans le tout premier appel. Ainsi on sort de la méthode **try_graph_coloring** et on affiche que la solution existe et est trouvée.

– Si cet appel récursif renvoie **false**, on remet la couleur du noeud *node* à 0 pour revenir à l'état de *assigned_colors* avant la dernière modification et on essaye une autre couleur. Si aucune couleur ne fonctionne, il n'existe pas de coloration valide et on renvoie **false** et on sort de tous les appels, en affichant que la solution n'existe pas.

```
bool try_graph_coloring(int node, int K, std::vector<int> &assigned_colors){
    // si tout est déjà attribué c'est bon, on sort de la récursion
    if (node == N) return true;

    for (int color = 1; color <= K; color++){
        if (good_to_take(node, color, K, assigned_colors)){
            assigned_colors[node] = color;
            // backtracking:
            if (try_graph_coloring(node + 1, K, assigned_colors) == true){
                return true;
            }
            // mais si jamais on ne renvoie pas true il ne faut PAS attribuer
            //   cette couleur à ce sommet, et on remet comme c'était avant:
            assigned_colors[node] = 0;
        }
    }
    // si on ne peut attribuer aucune couleur à un sommet on renvoie false
    return false;
}

bool graph_coloring(int K, std::vector<int> &assigned_colours){
    if (try_graph_coloring(0, K, assigned_colours)){
        std::cout << "Solution exists and it's found.\n";
        return true;
    }
    std::cout << "Solution doesn't exist.\n";
    return false;
}
```

Si l'appel à **graph_coloring** renvoie **true**, on peut afficher les couleurs obtenues en affichant le vecteur *assigned_colors*.

2.1.1 Application au graphe de Petersen

Voici le **graphe de Petersen** où nous avons numéroté les sommets de 0 à 9.

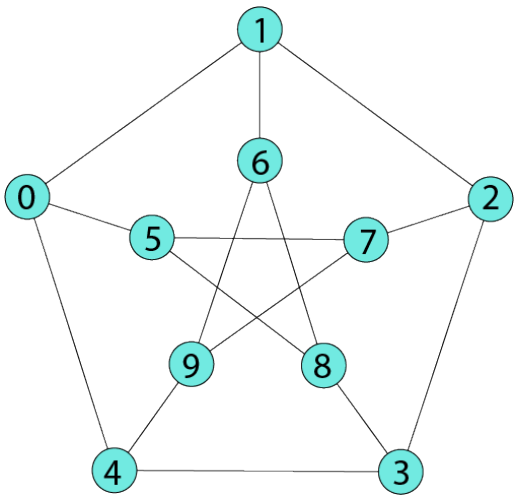


FIGURE 2 – Graphe de Petersen

Voici un appel de cette méthode pour le graphe de Petersen, en itlisant $K = 3$:

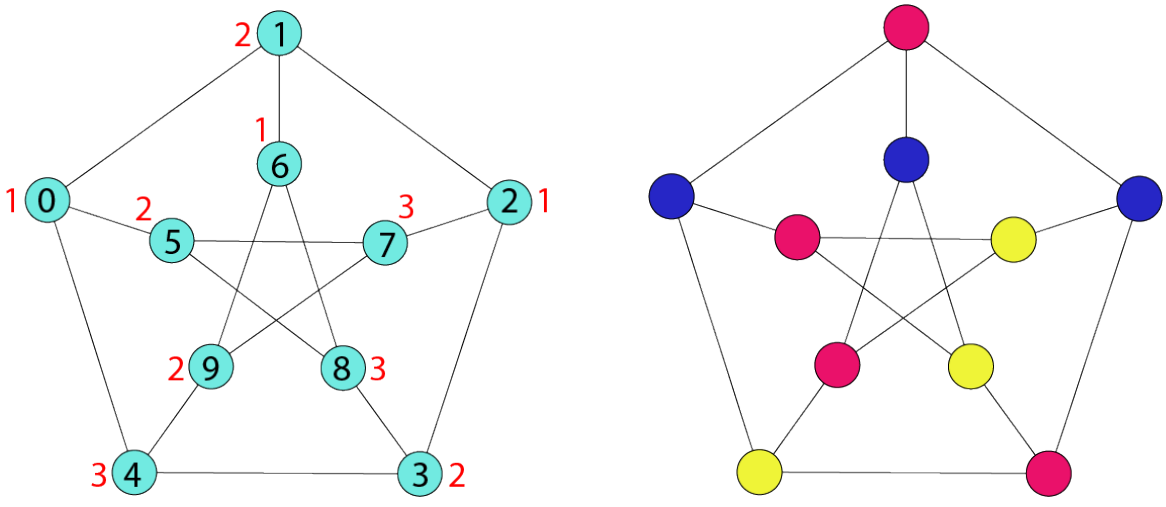
```
---------------------------- ALGORITHM STEPS -------------------------------

node = 0
        color = 1
assigned_colours = 1    0       0       0       0       0       0       0       0       0
node = 1
        color = 1
        color = 2
assigned_colours = 1    2       0       0       0       0       0       0       0       0
node = 2
        color = 1
assigned_colours = 1    2       1       0       0       0       0       0       0       0
node = 3
        color = 1
        color = 2
assigned_colours = 1    2       1       2       0       0       0       0       0       0
node = 4
        color = 1
        color = 2
        color = 3
assigned_colours = 1    2       1       2       3       0       0       0       0       0
node = 5
        color = 1
        color = 2
assigned_colours = 1    2       1       2       3       2       0       0       0       0
node = 6
        color = 1
assigned_colours = 1    2       1       2       3       2       1       0       0       0
node = 7
        color = 1
        color = 2
        color = 3
assigned_colours = 1    2       1       2       3       2       1       3       0       0
node = 8
        color = 1
        color = 2
        color = 3
assigned_colours = 1    2       1       2       3       2       1       3       3       0
node = 9
        color = 1
        color = 2
assigned_colours = 1    2       1       2       3       2       1       3       3       2
Solution exists and it's found.
1       2       1       2       3       2       1       3       3       2

---------------------------- ALGORITHM END ---------------------------------
```

FIGURE 3 – Algorithme de backtracking GC sur le graphe de Petersen avec K = 3

L'algorithme se termine en **0.000483 secondes** et attribue les "couleurs" 1, 2 et 3 à ce graphe. Si on suppose que 1 correspond à la couleur rouge, 2 à bleu, et 3 à jaune, nous obtenons :



(a) Graphe énuméré et "coloré" (b) Graphe coloré

FIGURE 4 – Coloration du graphe de Petersen

Pour ce graphe, la coloration avec seulement 2 couleurs n'est pas possible. En effet, le *nombre chromatique* du graphe de Petersen est 3. Voici comment notre algorithme se comporte lorsqu'on l'appelle avec $K = 2$:

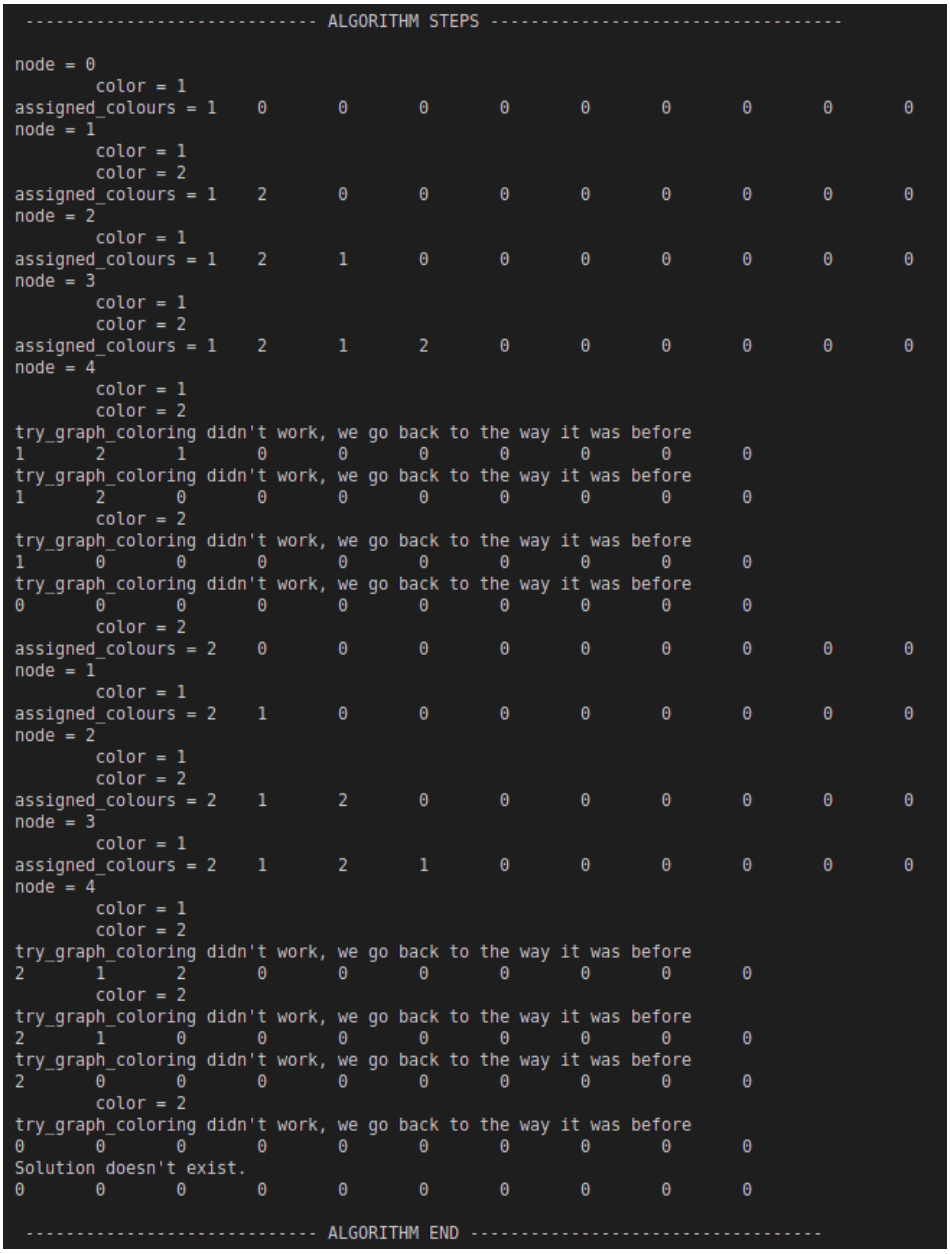


FIGURE 5 – Algorithme de backtracking GC sur le graphe de Petersen avec K = 2

L'algorithme s'est exécuté en **0.000168 secondes**. On voit bien comment l'appel récursif fonctionne, et qu'on est obligé de revenir tout au début pour pouvoir arrêter les appels et déclarer que la solution n'existe pas.

2.2 Algorithme de backtracking amélioré : GC_BIS

Pour essayer d'améliorer cet algorithme de backtracking, on introduit un **vecteur de paires (nombre de couleur disponibles, vecteur des couleurs disponibles)** de taille N que l'on nomme *col*.
On introduit également plusieurs méthodes pour l'initialisation de *col* et pour le choix du noeud suivant pour les appels récursifs. Les-voici :

- *cols* — méthode qui prend un noeud *node* en argument et place le nombre de couleurs disponibles et ces couleurs disponibles dans *col*[node]
- *Init* — méthode qui initialise le vecteur *col* pour touse les sommets
- *sort_by_available_colours* — méthode qui prend *col* et un vecteur vide *nodes* en argument (par référence) et remplit ce vecteur *nodes* avec les sommets suivants à prendre, classés par ordre de couleurs disponibles croissant, c'est-à-dire, d'abord on traite les sommets avec le moins de couleurs disponibles.

• **try_graph_coloring_BIS** et **graph_coloring_BIS** : Pour mieux les expliquer, supposons qu'on appelle la méthode **graph_coloring_BIS** pour un graphe connexe quelconque.

1. On initialise le vecteur *nodes* qui contiendra les noeuds suivants qu'on essayera de colorer, et le vecteur *col* également.

2. On entre dans la méthode **try_graph_coloring_BIS** dans laquelle on essaye de colorer le graphe en commençant par *nodes*[0]. L'argument *colored_node* contient le nombre de sommets déjà colorés. On l'initialise à N - *nodes.size()* pour pouvoir appliquer cette fonction à des graphes partiellement pré-colorés (utile pour l'application au sudoku qui suit). Pour un graphe vide, cette valeur sera évidemment 0.

3. On utilise les couleurs disponibles pour *node*. On trie les noeuds encore une fois, puisqu'on a mis à jour le vecteur *assigned_colors*.

4. On commence les appels récursifs de **try_graph_coloring_BIS** avec les nouveaux noeuds obtenus par le tri précédent (l'appel à *sort_by_available_colours*) et *colored_node* augmenté de 1, pour noter qu'on vient de colorer un sommet.

   – Si cet appel récursif renvoie **true** alors on a réussi à colorer le graphe et on renvoie **true** dans le tout premier appel. Ainsi on sort de la méthode **try_graph_coloring_BIS** et on affiche que la solution existe et est trouvée.

   – Si cet appel récursif renvoie **false**, on remet la couleur du noeud *node* à 0 pour revenir à l'état de *assigned_colors* avant la dernière modification et on essaye une autre couleur parmi celles disponible. Si aucune couleur ne fonctionne, il n'existe pas de coloration valide et on renvoie **false** et on sort de tous les appels, en affichant que la solution n'existe pas.

Comme dans l'algorithme naïf, la condition d'arrêt c'est qu'on ait réussi à colorer tous les N sommets du graphe. Voici l'implémentation en C++ :

```cpp
bool try_graph_coloring_BIS(int node, int K, std::vector<int> &nodes,
                            std::vector<std::pair<int, std::vector<int>>> &col,
                            std::vector<int> &assigned_colors, int colored_node){
    // si le nombre de noeuds colorés est N ça veut dire qu'on a tout coloré
    if (colored_node == N) return true;
    // col[node].second contient les couleurs disponibles pour node
    for (int color : col[node].second){
        assigned_colors[node] = color;
        // on trie et choisit les noeuds en fonction des couleurs disponibles
        sort_by_available_colours(K, nodes, col, assigned_colors);
        // s'il n'y a plus de noeuds disponibles on a réussi la coloration
        if (nodes.size() == 0) return true;

        // backtracking
        if (try_graph_coloring_BIS(nodes[0], K, nodes, col, assigned_colors,
                                   colored_node+1) == true){
            return true;
        }
        // mais si jamais on ne renvoie pas true il ne faut PAS attribuer
        // cette couleur à ce sommet, et on remet sa couleur à 0
        assigned_colors[node] = 0;
    }
    // si on ne peut attribuer aucune couleur à un sommet on renvoie false
    return false;
}

bool graph_coloring_BIS(int K, std::vector<int> &assigned_colours){
    // initialisation de nodes et col
    std::vector<int> nodes(N);
    std::vector<std::pair<int, std::vector<int>>> col (N);
    sort_by_available_colours(K, nodes, col, assigned_colours);

    if (try_graph_coloring_BIS(nodes[0], K, nodes, col, assigned_colours,
                               N-nodes.size())){
        std::cout << "Solution exists and it's found.\n";
        return true;
    }
    std::cout << "Solution doesn't exist.\n";
    return false;
}
```

De même que pour l'algorithme naïf, si l'appel à **graph_coloring_BIS** renvoie **true**, on peut afficher la coloration obtenue en affichant le vecteur *assigned_colors*.

L'avantage de cet algorithme se verrait plus clairement avec un grand graphe, pour lequel on devrait utiliser un K assez grand pour que la coloration soit possible. Au lieu d'itérer sur toutes les couleurs et tous les sommets, qui serait sans doute très coûteux, on colore d'abord les sommets les plus faciles à colorer. Si jamais on doit colorer un sommet avec la dernière couleur disponible, avec ce deuxième algorithme cela se ferait en une étape, alors qu'avec l'algorithme naïf on serait obligé d'itérer sur toutes les couleurs et partir dans de nombreux appels récursifs avant de trouver la solution optimale.

2.2.1 Application au graphe de Petersen

Voici l'appel de ce nouvel algorithme pour le graphe de Petersen. Nous remarquons que le nombre de vérifications est légèrement différent, compte tenu du fait que nous choisissons les couleurs sur lesquelles on itère.

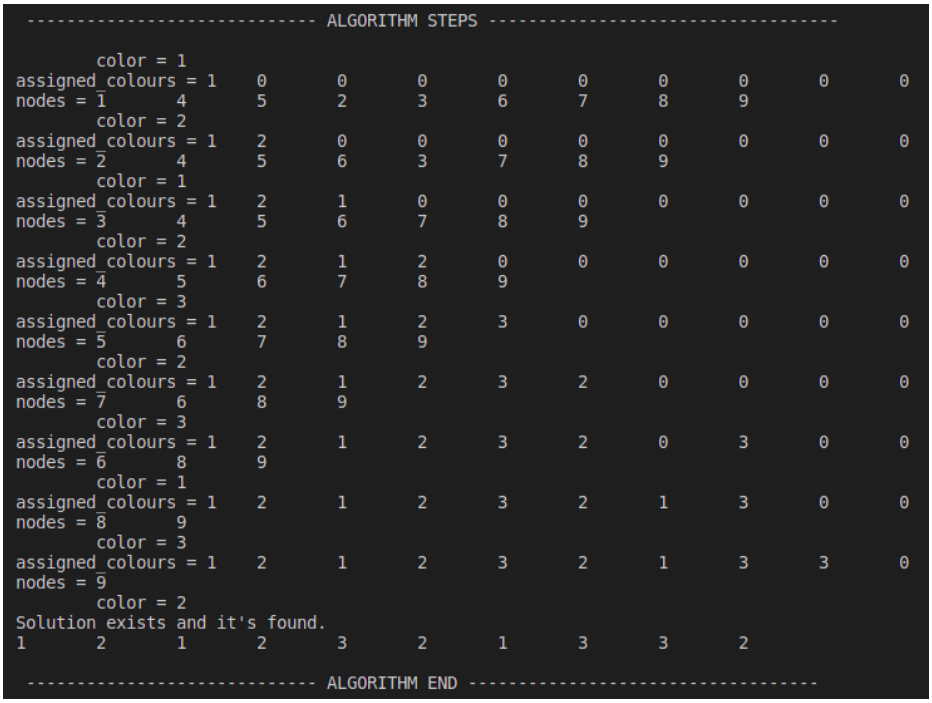


FIGURE 6 – Algorithme de backtracking GC_BIS sur le graphe de Petersen avec K = 3

L'algorithme se termine en **0.000765 secondes** et attribue les "couleurs" 1, 2 et 3 à ce graphe. Nous obtenons le même résultat qu'avec l'algorithme naïf, avec une performance similaire.

3 Application : *Sudoku*

Le *Sudoku* est un jeu de grille. Le but du jeu est de remplir la grille avec des chiffres (ou de lettres ou de symboles), qui ne se trouvent jamais plus d'une fois sur une même ligne, dans une même colonne ou dans une même région (également appelée *bloc* ou *sous-grille*).

3.1 Sudoku vu comme un graphe

Un jeu de *Sudoku* peut être vu comme un graphe de la façon suivante. Chaque case d'une grille de *Sudoku* peut être vue comme un sommet d'un graphe, où les arcs relient tous les sommets correspondants aux cases se trouvant dans la même ligne, tous les sommets correspondants aux cases se trouvant dans la même colonne et tous les sommets correspondants aux cases se trouvant dans le même bloc.

La figure suivante montre comment dénombrer le nombre de voisins d'un sommet dans une grille de *Sudoku* de taille 9×9, et comment généraliser ce dénombrement pour un *Sudoku* de taille $n^2 \times n^2$.

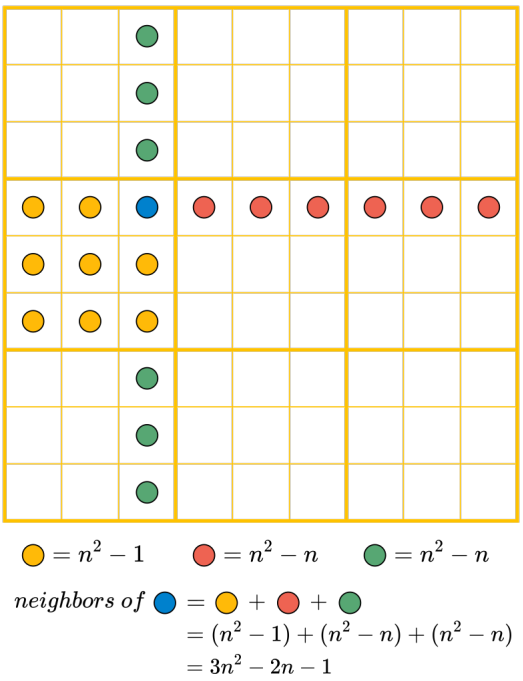


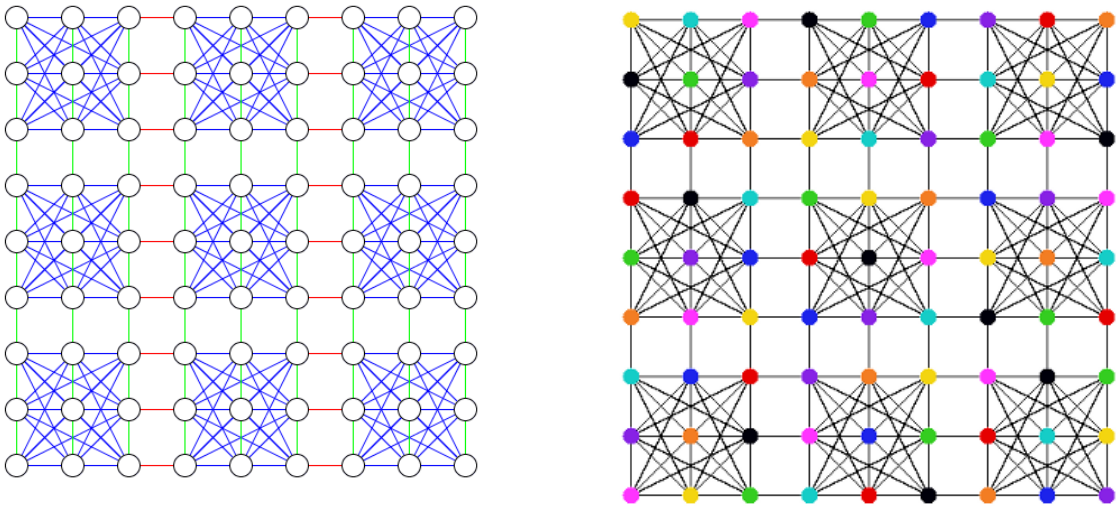
FIGURE 7 – Voisins d'un sommet d'une grille de *Sudoku*

Donc, pour une grille de taille $n^2 \times n^2$, le graphe correspondant possède n^4 sommets, chacun avec exactement $3n^2 - 2n - 1$ voisins. Ainsi, il s'agit d'un graphe *régulier* (graphe où chaque sommet a le même nombre de sommets adjacents).
Le nombre d'arcs pour un *Sudoku* de taille $n^2 \times n^2$ est donc égal à :

$$m = n^4(3n^2 - 2n - 1)/2$$

Par exemple, pour une grille de *Sudoku* de taille 4×4, le graphe correspondant a 16 sommets et 56 arcs, et est 7-régulier (chaque sommet a 7 voisins). Pour une grille de *Sudoku* de taille 9×9, le graphe correspondant a 81 sommets, 810 arcs, et est 20-régulier.

Sur la figure 8 il y a une représentation du graphe correspondant à la grille d'un jeu de *Sudoku* de taille 9×9, où pour ne pas surcharger le dessin, les arcs reliant les sommets qui se trouvent sur la même ligne et ceux reliant les sommets qui se trouvent sur la même colonne, sont omis.



(a) Graphe correspondant au *Sudoku* 9×9 (b) Une coloration possible

FIGURE 8 – Sudoku : Représentation des graphes correspondants aux grilles

Donc, **le problème de résolution d'une grille de *Sudoku*** peut être vu comme un **problème de coloration d'un graphe partiellement pré-coloré**.

Remarque : Les nombres déjà écrits dans la grille de *Sudoku*, on les appellera ***clues***.

3.2 Adaptation des algorithmes GC et GC_BIS

Pour pouvoir appliquer l'algorithme **GC** au problème de Sudoku il faudrait faire une petite modification à la fonction **try_graph_coloring**. Il s'agit d'appeler cette fonction récursive directement si le sommet courant a déjà une couleur qui lui est attribuée. Cette modification ne change rien par rapport à la coloration des graphes qui ne sont pas pré-colorés. Voici la nouvelle version :

```
bool Graphe:: try_graph_coloring(int node, int K, std::vector<int> &assigned_colors){
    // si tout est déjà attribué c'est bon, on sort de la récursion
    if (node == N) return true;
    // si une couleur est déjà atttribuée on fait le backtracking directement
    if (assigned_colors[node] != 0){
        return try_graph_coloring(node + 1, K, assigned_colors);
    }
    for (int color = 1; color <= K; color++){
        if (good_to_take(node, color, K, assigned_colors)){
            assigned_colors[node] = color;
            // backtracking:
            if (try_graph_coloring(node + 1, K, assigned_colors) == true){
                return true;
            }
            // mais si jamais on ne renvoie pas true il ne faut PAS attribuer
            //   cette couleur à ce sommet, et on remet comme c'était avant:
            assigned_colors[node] = 0;
        }
    }
    // si on ne peut attribuer aucune couleur à un sommet on renvoie false
    return false;
}
```

L'algorithme **GC_BIS** ne nécessite pas de modifications pour être appliqué au problème de résolution d'une grille de *Sudoku*.

3.3 Test préliminaire de GC et GC_BIS pour la résolution d'une grille de *Sudoku*

Voici un test simple, pour une grille de *Sudoku* de taille 9×9, qui n'est pas considérée difficile, et qui contient 24 *clues*.

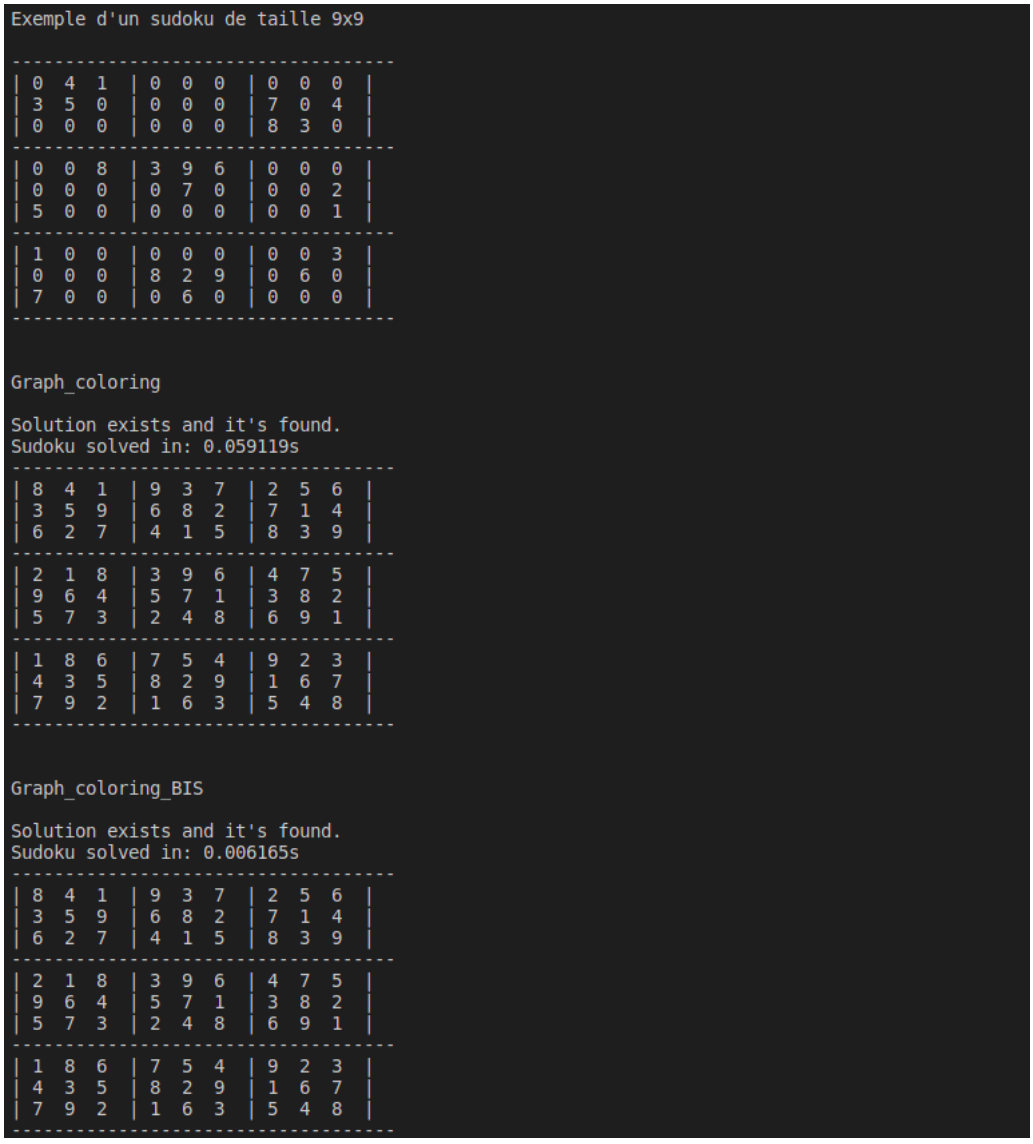


FIGURE 9 – **GC** et **GC_BIS** sur une grille de *Sudoku* de taille 9×9

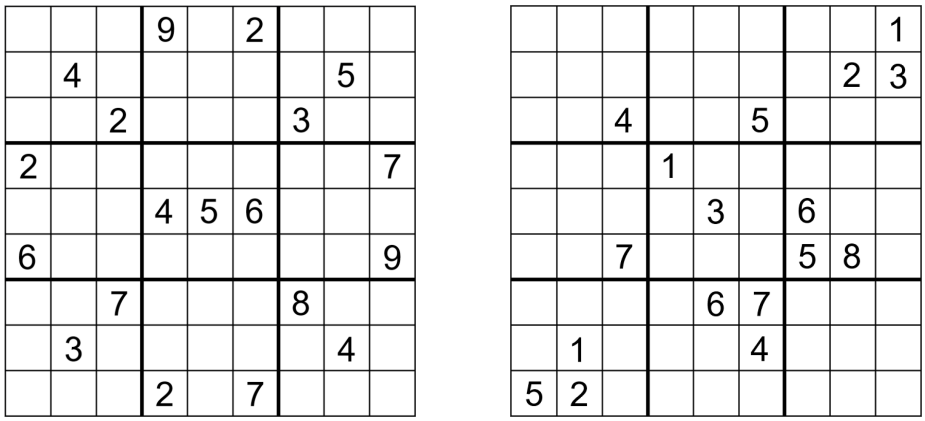
Nous pouvons remarquer que l'algorithme amélioré est environ 10 fois plus performant sur ce problème en particulier. Mais cela ne nous permet pas de conclure qu'il est globalement meilleur. Pour des grilles de *Sudoku* de taille 4×4, l'algorithme naïf était 10 fois plus performant.

Remarque : La difficulté des *Sudoku* donnée dans les magazines de jeu, ou sur des divers sites internet, n'a pas de grande importance pour la résolution du problème de coloration du graphe d'une grille de *Sudoku*. Cependant, en choisissant bien les grilles et le nombre de *clues*, on peut créer des exemples de difficultés différentes, par exemple dans un ordre croissant, pour des algorithmes de backtracking.

3.4 Comparaison des performances des deux algorithmes

Pour comparer la performance des 2 algorithmes, nous avons cherché des grilles de Sudoku de difficulté différente, et le nombre de *clues* différent. Nous avons choisi 7 grilles :

difficulté	**n°clues**	**particularité**
easy	38	/
medium	30	/
hard	25	/
expert	22	/
super expert	19	double symétrie orthogonale
hyper expert	17	symétrie diagonale
ultra expert	17	construit pour se "rebeller" contre un algorithme de backtracking naïve



(a) Double symétrie orthogonale (b) Symétrie diagonale

FIGURE 10 – Sudoku : *super expert* et *hyper expert*

La dernière grille est spécifiquement construite pour aller contre l'algorithme de backtracking naïve. Si on suppose que l'algorithme résout du haut vers le bas (comme notre algorithme naïf), ce puzzle avec peu de clues (17), pas de clues dans la première ligne et aussi et avec solution de la première ligne : (9,8,7,6,5,4,3,2,1), va en effet contre l'algorithme naïf. Le programme mettra un temps important pour remonter dans les appels récursifs à chaque fois qu'il se trompe de premier choix, et puisqu'on commence par colorer le premier noeud avec la plus petite couleur disponible, ceci sera effectivement le cas.

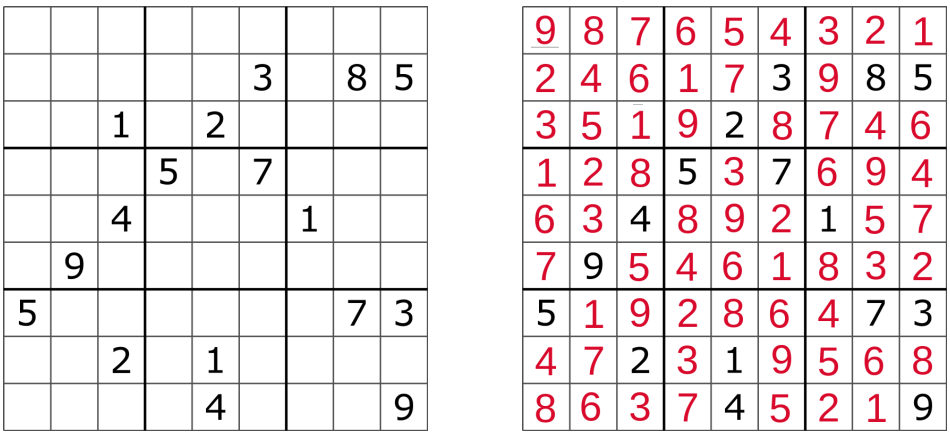


FIGURE 11 – Sudoku : *ultra expert*

Voici les performances des deux algorithmes, comparées par rapport au temps d'exécution. Pour rappel, **GC** est l'algorithme naïf et **GC_BIS** est l'algorithme amélioré.

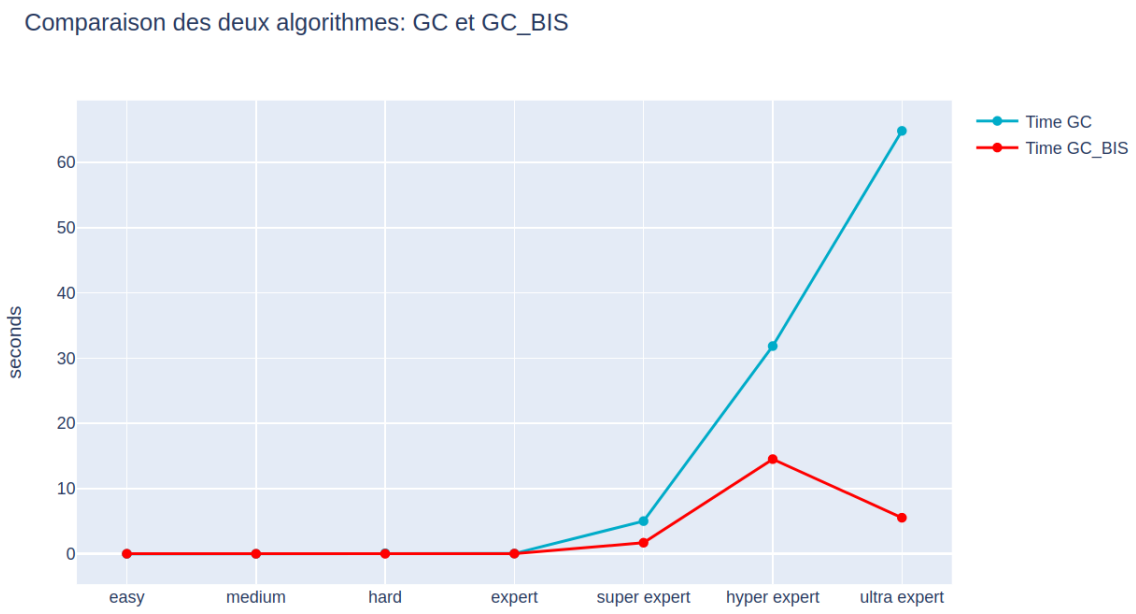


FIGURE 12 – Comparaison : GC vs. GC_BIS

Avec l'algorithme amélioré on résout la grille *ultra expert* en seulement **5.46082 secondes**, alors que l'algorithme naïf met **64.2307 secondes** !

Ceci nous montre qu'il est important de bien choisir les noeuds et les couleurs sur lesquels on fait les appels récursifs dans l'algorithme de backtracking. On voit bien que plus la difficulté du problème augmente, plus l'algorithme amélioré est performant.

4 Conclusion

Ce projet nous a permis de manipuler beaucoup de structures et concepts différents liés au langage **C++**, et à la programmation en général, comme par exemple :

- std : :pair<.,.>
- std : :vector<std : :vector<.> >
- std : :unordered_map<.,.>
- implémenter des fonctions récursives
- optimiser les algorithmes au fur et à mesure

Dans un premier temps, au lieu de stocker les voisins dans un vecteur de vecteurs, on les cherchait à chaque étape de l'algorithme, ce qui s'est traduit par un temps d'exécution d'environ **7 minutes** pour l'algorithme naïf pour la grille de *Sudoku super expert* et environ **38 minutes** pour la grille *hyper expert*. L'amélioration est d'un ordre non-négligeable. On conclut qu'il faut toujours bien lire la documentation pour les structures avec lesquelles on pourrait poser le problème, chercher de nouvelles structures et de façons de stocker les informations essentielles au fonctionnement des algorithmes, afin d'obtenir une solution correcte et un temps d'exécution optimal.