École Nationale Supérieure
d'Informatique d'Alger

2CS
OPTIMISATION
Rapport

Branch and bound appliqué au problème de coloration des graphes

Étudiants :
Wilem LAMDANI
Linda BELKESSA
Oussama DJILI
Abdelaziz TAKOUCHE

Enseignant :
Mr. Amine KECHID

30 mars 2022

Table des matières

1 Introduction

Dans ce rapport, nous utilisons l'algorithme de branch and bound pour minimiser la valeur k dans la coloration des graphes, ainsi que l'implémentation du code et la discussion des résultats obtenus.

1.1 Problème de coloration des graphes PCG

La coloration de graphe est un problème qui consiste à assigner des couleurs aux sommets d'un graphe de telle sorte que les voisins soient assignés avec des couleurs différentes. Le problème de la coloration d'un graphe avec le nombre minimum de couleurs possible est NP-difficile avec un certain nombre d'applications d'intérêt tels que l'ordonnancement des horaires, l'optimisation du code et l'établissement d'un plan d'affectation.

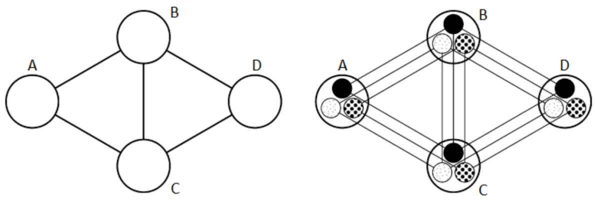


FIGURE 1 – Problème de coloration

1.2 Technique du branch and bound

Un algorithme de branche and bound génère de manière récursive des instances de problème ("branches"), dont certaines sont éliminées sur la base d'une métrique ("limite") afin de minimiser ou maximiser la valeur d'intérêt. Avec la coloration des graphes, l'algorithme peut être utilisé afin de minimiser le nombre de couleurs. A cet effet, les mesures suivantes et les définitions sont utilisées :

• nœud : soit le nœud initial, soit un nœud enfant généré récursivement. Le nœud est composé d'un graphique à colorier, une borne inférieure et une borne supérieure.

• borne inférieure : c'est la plus petite valeur de k qu'un graphe donné G puisse avoir. Le cas trivial est k = 1, car tout graphique avec V > 0 doit être coloré en utilisant au moins une couleur.

• borne supérieure : c'est la plus grande valeur de k que le graphe donné G peut avoir. Le cas trivial est k = V , car chaque sommet d'un graphe peut être coloré d'une couleur différente.

• borne supérieure globale : il s'agit de la borne supérieure représentative de la meilleure coloration minimisée à un stade donné de l'algorithme branch and bound.

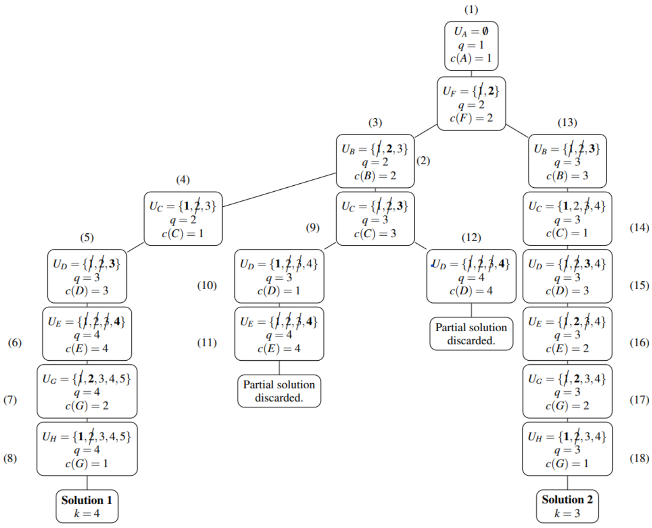


FIGURE 2 – Branch and bound

2 Solution en détails

2.1 Explication de l'algorithme dans ses grandes lignes

Nous avons structuré notre programme en une séquence de décisions telles que :

• Nous prenons nos sommets dans un ordre quelconque -en l'occurrence, leur ordre d'apparition dans la matrice d'adjacence - : à une étape i de l'algorithme, une couleur sera affectée au sommet i, et s'en suivra une série de tests.

• Les couleurs affectées au nœud i seront générées afin de n'inclure aucune couleur voisine au nœud - principe de séparation - et ce afin de ne parcourir que l'ensemble des solutions réalisables.

• à chaque affectation, une évaluation de la solution courante sera réalisée, si celle-ci se trouve supérieure ou égale à la meilleure solution rencontrée avant cela, elle sera rejetée et la branche sera élaguée : et ce car l'évaluation ne peut qu'augmenter durant le parcours

2.2 Structure des données

Les structures principales utilisées dans notre solution sont :

• class Graph : qui contient principalement la matrice d'adjacence, une matrice binaire. En plus de fonctions utiles : tel que le parsing qui permet de construire la matrice d'affichage à partir des fichiers fournis, l'affichage, et quelques getters.

Étant donné que l'algorithme de coloration implémenté est itératif, il nous faudra empiler les noeuds courants afin de pouvoir effectuer un backtracking et parcourir notre arbre :

• Pile : une liste servant à stocker les contextes afin de ne pas perdre la progression, les contextes dans notre cas sont les noeuds à parcourir Noeud : un tuple composé d'un vecteur d'entier -vecteur de coloration- , de l'indice du noeud courant ainsi que le nombre de couleurs utilisées jusqu'à présent Solution : un tuple composé du vecteur de coloration complet, ainsi que le nombre de couleurs utilisées afin de simplifier les comparaisons. Contiendra à la fin de l'exécution la solution optimale.

```python
class Graph:

    matrice = [] #Contient la matrice d'adjacence

    SUBPLOT_NUM = 211
    TYPE_COMMENT = "c"
    TYPE_PROBLEM_LINE = "p"
    TYPE_EDGE_DESCRIPTOR = "e"

    def __init__(self, file_name): #constructeur···

    def nbNoeuds(self) :···

    def get_Matrice(self) :···

    def get_Voisins(self, noeud): #renvoie la liste des voisins···

    def displayGraph(self,color) : #Affiche le graph et colorie les noeuds···

    @staticmethod
    def parse_line(line): ···

    def from_file(self, filename): #Contruit la matrice d'adjacence à partir des fichiers fournis···
```

FIGURE 3 – Définition de la classe Graph

2.3 Fonction d'évaluation utilisée

Afin d'évaluer la solution courante à un niveau i, un algorithme de complexité O(n^2) sera exécutée, son principe : Parcourir les noeuds non colorés - donc à partir de i+1 - : si l'un d'eux est adjacent à des noeuds dont les couleurs forment l'ensemble des couleurs utilisées jusqu'à présent : nous devrons impérativement utiliser une couleur de plus -> L'évaluation sera égale au nombre de couleurs de la solution courante + 1 Si en plus de celà, nous trouvons un noeuds adjacent à celui cité plus haut, en plus d'être adjacent à l'ensemble des couleurs courantes : nous devrons utiliser encore une couleur de plus dans le meilleur des cas -> L'évaluation sera égale au nombre de couleurs de la solution courante + 2

```python
def Eval(niveau, graph, colors, nb_colors, nb_noeuds):
    """
    Calcul l'évaluation du noeud courant
    """
    increment = False
    for v in range(niveau +1, nb_noeuds) :
        voisins = graph.get_Voisins(v)
        if (len(set([colors[i] for i in voisins])) >= nb_colors) :
        #toutes les couleurs avoisinent v : on retourne au moins nb_colors +1
            increment = True
            for j in range(v + 1,nb_noeuds) :
                if (j in voisins) and (len(set([colors[i] for i in graph.get_Voisins(j)])) >= nb_colors) :
        #Si un voisin non coloré de ce noeud est aussi voisin de toutes les coueulurs : on retourne nb_colors +2
                    return nb_colors + 2
    return (nb_colors + 1) if increment else nb_colors
```

FIGURE 4 – Fonction d'évaluation

2.4 Principe de Séparation

Le principe de séparation dans le problème de coloration de graphe est plutôt aisé à implémenter, et ce en générant toutes les couleurs qui ne causeront pas de conflit.

Ces couleurs sont générées en ordre croissant, leur nombre se limite au nombre de couleurs obtenu lors de l'initialisation avec algorithme glouton, de cette liste obtenue nous retirerons les couleurs de ses voisins déjà colorés.

```python
def generate_validColors(v,graph,colors,max_color) :
    """
    Génère les couleur valides d'un noeuds
    le noeud v étant le dernier noeud à colorer, nous parcourant les noeuds d'indice inférieur à lui
    Et ce en retirant les couleurs de ses voisins à l'ensemble des couleurs disponibles
    """
    Restricted_Colors = [colors[i] for i in [j for j in graph.get_Voisins(v) if j<v]]
    return [i for i in range(1,max_color+1) if i not in Restricted_Colors]
```

Figure 5 – Etape de sépération

2.5 Stratégie de parcours

Afin d'économiser le maximum d'espace mémoire, nous avons opté pour un algorithme itératif, avec l'utilisation d'une pile pour la sauvegarde du contexte courant :

• Nous parcourons notre arbre en profondeur : Avec la pile, il s'agit de dépiler le contexte de l'arbre - racine du sous-arbre - passer aux fils soit au noeud suivant : de sorte à parcourir toutes ses couleurs valides : les évaluer, et les empiler uniquement s'ils présentent une chance de comprendre une solution optimale - évaluation inférieure à la solution la plus optimale jusqu'ici si trouvée, ou inférieur ou égal au nombre de couleur de l'initialisation avec heuritique -, le fait de ne pas empiler un contexte signifie que la branche a été élaguée

```python
def GraphColoring(graph) :
    """
    La fonction chargée du traitement global
    la coloration finale se trouvera dans Solution
    Le parcours de l'arbre se fait en empilant les contexte
    """
    nbNoeuds = graph.nbNoeuds()
    colors = [0]*nbNoeuds #Liste des couleurs, 0 signifie la non coloration d'un noeuf
    pile = [] #pile de parcours
    max_color = greedy_Heurisic(graph,nbNoeuds,0)
    solution = colors.copy(),max_color+1 #Contient la meilleur solution réalisable : la plus optimale
    print("Nombre de couleurs avec Heurisitque = ", max_color)
    colors[0] = 1 #initaliser le premier noeud
    pile.append([colors.copy(),0,1]) #Empiler le contexte initial
    while (pile): #tant que la pile n'est pas vide
        colors, indice, nbcouleur = pile.pop() #On dépile
        if (indice != nbNoeuds -1 ) : #Nous n'avons pas atteint un noeud feuille
            indice += 1 #Passer au noeud suivant
            for color in generate_validColors(indice,graph,colors,max_color) :
                #Générer les couleurs possibles = solutions réalisables
                if color not in colors :
                    nbcouleur = nbcouleur + 1 #Si la couleur n'a jamais été utilisée auparavent
                    if nbcouleur >= solution[1] :
                        break
                colors[indice] = color #Colorer le noeud
                if (Eval(indice,graph, colors, nbcouleur,nbNoeuds) < solution[1] ) : #Tester la condition de non élagage
                    pile.append([colors.copy(), indice, nbcouleur]) #Empiler le contexte
        else : #Le noeud feuille a passé le test d'évaluation avant empilation, il est donc optimal
            solution = colors.copy(), nbcouleur #Remplacer la solution optimale
    end = time.time()
    print("Temps d'execution = ", end-start," seconds")
    print("Le nombre de couleurs optimal = ", solution[1])
    #Une fois la pile vide, la solution finale est contenue dans la variable solution
    graph.displayGraph(solution[0].copy())
```

Figure 6 – Problème de coloration

2.6 Résultats et temps d'exécution

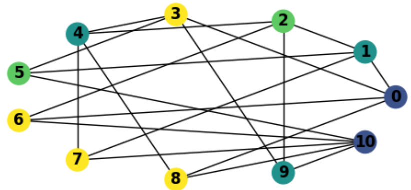
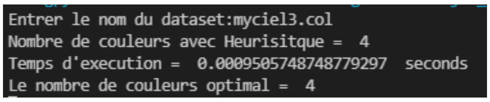


FIGURE 7 – Graphe pour myciel3.col



```
Entrer le nom du dataset:myciel3.col
Nombre de couleurs avec Heurisitque =  4
Temps d'execution =  0.0009505748748779297  seconds
Le nombre de couleurs optimal =  4
```

FIGURE 8 – Temps d'exécution myciel3.col

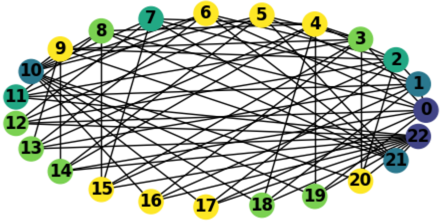
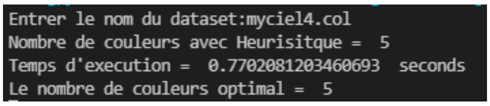


FIGURE 9 – Graphe pour myciel4.col



```
Entrer le nom du dataset:myciel4.col
Nombre de couleurs avec Heurisitque =  5
Temps d'execution =  0.7702081203460693  seconds
Le nombre de couleurs optimal =  5
```

FIGURE 10 – Temps d'exécution myciel4.col

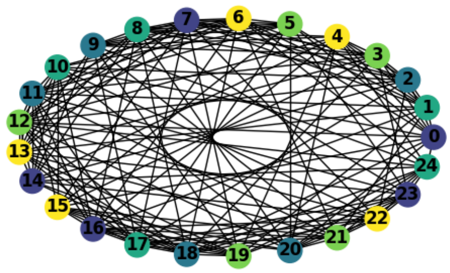
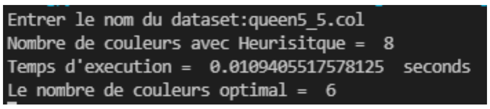


FIGURE 11 – Graphe pour graph$_{q}ueen5_{5}.col$



```
Entrer le nom du dataset:queen5_5.col
Nombre de couleurs avec Heurisitque =  8
Temps d'execution =  0.0109405517578125  seconds
Le nombre de couleurs optimal =  6
```

FIGURE 12 – Temps d'exécution graph$_{q}ueen5_{5}.col$

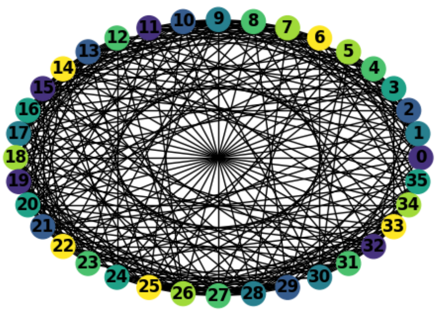
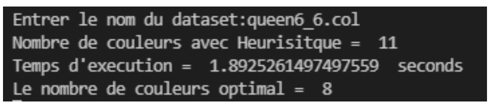


FIGURE 13 – Graphe pour graph$_{q}ueen6_{6}.col$



```
Entrer le nom du dataset:queen6_6.col
Nombre de couleurs avec Heurisitque =  11
Temps d'execution =  1.8925261497497559  seconds
Le nombre de couleurs optimal =  8
```

FIGURE 14 – Temps d'exécution graph$_{q}ueen6_{6}.col$