

**ECOLE NATIONALE SUPERIEURE
D'INFORMATIQUE D'ALGER**

Rapport de projet

Module d'optimisation combinatoire

Etude des techniques d'optimisation combinatoire

Appliquées au problème de coloration des graphes

Réalisa par :

BELKESSA Linda

KHAMOUM Amel

LAMDANI Wilem

LAOUADI Abir

MABROUKI Mohamed Laid Malik

SEBAA Souad

Encadré par : KECHID Amine

Mai, 2022

RÉSUMÉ

Techniques d'optimisation combinatoire Coloration des graphes

Dans ce rapport, nous utilisons une variété de techniques d'optimisation combinatoire pour minimiser la valeur k dans la coloration des graphes, ainsi que l'implémentation du code et la discussion des résultats obtenus.

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.

Nous effectuons nos tests sur les datasets benchmark DIMACS, en appliquant des méthodes exactes (le branch and bound), ensuite les heuristiques spécifiques par construction notamment, Greedy coloring, DSATUR et Max stable, enfin nous passons aux méta-heuristiques, nous avons choisi de travailler avec l'algorithme génétique et la recherche tabou.

REMERCIEMENTS

Nous remercions nos enseignants pour l'encadrement et les conseils fournis durant la réalisation du projet .

PLAN DU RAPPORT

Résumé	2
Remerciements	3
Table des figures	7
1 Etat de l'art	9
1.1 Présentation du problème	9
1.2 Représentation des données	9
1.2.1 Graphe non coloré :	9
1.2.2 Graphe coloré :	10
1.3 Méthodes de résolution du PGC :	10
1.3.1 Les méthodes exactes : Branch Bound	10
1.3.2 Les heuristiques spécifiques :	11
1.3.3 Les méta-heuristiques :	11
2 Conception et implémentation des techniques d'optimisation	13
2.1 Méthodes exactes : Branch and bound (Séparation et évaluation	13
2.1.1 Explication de l'algorithme	13
2.1.2 Tests (captures du temps d'exécution) :	16
2.1.3 Code source et implémentation	21
3 Méthodes approchées : Heuristiques	30
3.1 Introduction :	30
3.2 Explication des algorithmes :	30
3.2.1 Coloration gloutonne (Greedy algorithm)	30
3.2.2 DSatur :	31
3.2.3 Max stable (avec coloration gloutonne)	34
3.3 Tests (captures du temps d'exécution) :	36
3.3.1 Instance queen5_5.col (25 noeuds) :	36
3.3.2 Instance myciel4.col (23 noeuds) :	38
3.4 Code source et implémentation	40
4 Méthodes approchées : métahéuristiques	51
4.1 Explication des algorithmes :	51

4.1.1	Recherche Tabou :	51
4.1.2	AG : Algorithme génétique :	53
5	Etude des résultats	56
5.1	Etude empirique	56
5.1.1	Paramètres des algorithmes génétiques :	56
5.1.2	Effet de la taille de la liste Tabou :	60
5.2	Etude comparative	61
	Références	63

Liste des figures

1.1	Problème de coloration des graphes	9
1.2	Matrice d'adjacence	10
2.1	Algorithme du branch and bound	14
2.2	Contenu bb du fichier config.py	16
2.3	Résultat du programme avec l'instance “queen5_5.col”	17
2.4	Résultat du programme avec l'instance “myciel3.col”	18
2.5	Graphe de l'instance “myciel3.col” après coloration	19
2.6	Résultat du programme avec l'instance “queen7_7.col”	20
2.7	Graphe de l'instance “queen7_7.col” après coloration	21
2.8	Contenu bb du fichier config.py	22
2.9	Contenu bb du fichier graph.py	25
2.10	Contenu bb du fichier color_graph.py	29
3.1	Algorithme DSatur	33
3.2	Résultat de l'algorithme “Coloration gloutonne” avec l'instance “queen5_5.col”	36
3.3	Graphe de l'instance “queen5_5.col” après coloration avec l'algorithme “Coloration gloutonne”	36
3.4	Résultat de l'algorithme “DSatur” avec l'instance “queen5_5.col”	37
3.5	Graphe de l'instance “queen5_5.col” après coloration avec l'algorithme “DSatur”	37
3.6	Résultat de l'algorithme “Max stable” avec l'instance “queen5_5.col”	37
3.7	Graphe de l'instance “queen5_5.col” après coloration avec l'algorithme “Max stable”	38
3.8	Résultat de l'algorithme “Coloration gloutonne” avec l'instance “myciel4.col”	38
3.9	Graphe de l'instance “myciel4.col” après coloration avec l'algorithme “Coloration gloutonne”	39
3.10	Résultat de l'algorithme “DSatur” avec l'instance “myciel4.col”	39
3.11	Graphe de l'instance “myciel4.col” après coloration avec l'algorithme “DSatur”	40
3.12	Résultat de l'algorithme “Max stable” avec l'instance “myciel4.col”	40

3.13 Graphe de l'instance “myciel4.col” après coloration avec l'algorithme “Max stable”	40
3.14 Contenu du fichier config.py (heuristiques)	41
3.15 Contenu du fichier graph.py (heuristiques)	44
3.16 Contenu du fichier color _{graph} .py(<i>heuristiques</i>)	50
4.1 Algorithme de la recherche Tabou	53
5.1 Paramètres AG	56
5.2 Code pour tester l'impact de la MUTATION_PROBABILITY . .	57
5.3 Effet de la MUTATION_PROBABILITY	58
5.4 Code pour tester l'impact de la POPULATION_SIZE	58
5.5 Effet de la POPULATION_SIZE	59
5.6 Code pour tester l'impact de la GENERATIONS_NUM	59
5.7 Effet de la GENERATIONS_NUM	60
5.8 Effet de la taille de la liste sur des benchmarks	60
5.9 Histogramme associé	61
5.10 Comparaison des temps d'exécution	61
5.11 Comparaison du nombre de couleurs trouvé	62

Chapitre 1

Etat de l'art

1.1 Présentation du problème

La coloration de graphe est un problème qui consiste à assigner des couleurs aux sommets d'un graphe sous certaines contraintes : celle d'assigner à chaque voisin une couleur distincte. 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.

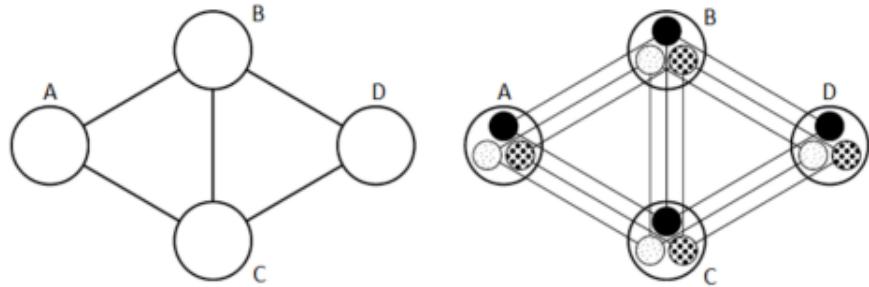


FIGURE 1 – Problème de coloration

FIGURE 1.1 – Problème de coloration des graphes

1.2 Représentation des données

1.2.1 Graphe non coloré :

La manière la plus simple de représenter un graphe non orienté est d'utiliser sa matrice d'adjacence.

Une matrice d'adjacence est une matrice carrée binaire, où chaque indice représente

un noeud, la valeur d'un élément est de 1 s'il existe une adjacence entre les noeuds d'indice ligne/colonne, et de 0 si ces noeuds ne sont pas voisins.

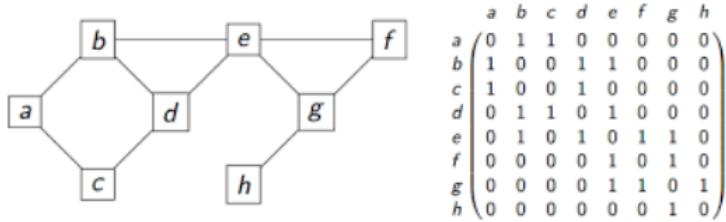


FIGURE 1.2 – Matrice d'adjacence

1.2.2 Graphe coloré :

Le graphe coloré - soit la solution- est représenté par un tableau où chaque case représente le numéro de la couleur choisie pour le nœud dont son numéro est l'index de son emplacement dans le tableau.

Les indexées sont tirés de la matrice d'adjacence citée précédemment.

1.3 Méthodes de résolution du PGC :

Pour la résolution de ce problème, plusieurs approches ont été adoptées, allant d'une résolution exacte, coûteuse en temps et en ressources, à des solutions plus légères bien qu'approchées.

1.3.1 Les méthodes exactes : Branch Bound

Les méthodes exactes - ou complètes - garantissent l'obtention d'une solution optimale à une instance de problème d'optimisation donnée.

Elles se reposent généralement sur la recherche arborescente et sur l'énumération partielle de l'espace de solutions.

Elles sont utilisées pour trouver au moins une solution optimale d'un problème. Les algorithmes exacts les plus connus sont la méthode de séparation et évaluation -BranchBound-, la programmation dynamique, et la programmation linéaire.

Et bien qu'elles offrent les meilleures solutions possibles aux problèmes, ces méthodes

souffrent de ce que l'on appelle "l'explosion combinatoire" : Le nombre de combinaisons augmentant avec la dimension du problème, le temps d'exécution s'en trouve exponentiel.

Ainsi L'efficacité de ces algorithmes n'est prometteuse que pour les instances de problèmes de petites tailles.

1.3.2 Les heuristiques spécifiques :

Une heuristique est un algorithme qui fournit rapidement (en temps polynomial) une solution réalisable, pas nécessairement optimale, pour un problème d'optimisation NP-difficile.

Une heuristique est une stratégie de bon sens pour se déplacer intelligemment dans l'espace des solutions, afin d'obtenir une solution approchée, la meilleure possible, dans un délai de temps raisonnable.

Celles-ci sont dites spécifiques : car se basant sur la structuration du problème en lui-même, il serait difficile de l'appliquer à d'autres contextes, elles sont donc dépendantes du problème à résoudre, et sont donc spécifiques à celui-ci.

La stratégie de ces méthodes appliquée à notre problème est souvent de colorer le graphe un sommet à la fois, en choisissant à chaque étape celui qui semble être le meilleur selon un critère bien défini, sans jamais se remettre en cause. ce qui font d'elles des méthodes rapides et donnent la plupart du temps une solution satisfaisante, mais la qualité du résultat dépend fortement des divers paramètres tel que l'ordre de parcours du graphe.

1.3.3 Les méta-heuristiques :

Le mot métaheuristique est composé de deux mots grecs : méta et heuristique. Le mot méta est un suffixe signifiant au-delà c'est-à-dire de niveau supérieur. Les métaheuristiques sont des méthodes généralement inspirées de la nature. Contrairement aux heuristiques, elles sont réutilisables et peuvent s'appliquer à plusieurs problèmes de nature différentes.

Dédiées principalement à la résolution des problèmes d'optimisation. Leur but est d'atteindre un optimum global tout en échappant les optima locaux.

Les métaheuristiques regroupent des méthodes qui peuvent se diviser en deux classes :

1. Métaheuristiques à solution unique : Ces méthodes traitent une seule solution à la fois, afin de trouver la solution optimale.
2. Métaheuristiques à population de solutions : Ces méthodes utilisent une population de solutions à chaque itération jusqu'à l'obtention de la solution globale.

Chapitre 2

Conception et implémentation des techniques d'optimisation

2.1 Méthodes exactes : Branch and bound (Séparation et évaluation)

2.1.1 Explication de l'algorithme

L'algorithme Branch Bound se base sur la construction d'un arbre de recherche sur l'espace de solutions possibles. Il est à noter que l'arbre n'est pas construit en entier au lancement du programme, mais qu'il est construit au fur et à mesure de l'exécution et de l'évaluation de chaque noeud. Chaque noeud de l'arbre constitue une solution partielle à laquelle une fonction d'évaluation est appliquée pour décider s'il y aura parcours des solutions basées sur ce noeud (c'est-à-dire continuer à parcourir et évaluer les fils de ce noeud) ou élagage du noeud.

Nœud racine de l'arbre

Dans notre cas, un noeud du niveau i représente le tableau des couleurs possibles du i ème sommet du graphe en entrée. Notons que les sommets dans le graphe sont numérotés aléatoirement. Intuitivement, le premier sommet numéroté devrait avoir n couleurs possibles avec n est le nombre total de sommets dans le graphe, la racine de l'arbre de recherche devrait être un tableau de n éléments. Or puisque le premier noeud peut avoir n'importe quelle couleur (car c'est le premier à tester) et qu'il est aléatoirement numéroté dans le graphe, nous pouvons attribuer à ce noeud une seule couleur aléatoire. Cette optimisation permet de

réduire considérablement la taille de l'arbre.

Construction et parcours de l'arbre de recherche :

Nous avons utilisé une heuristique qui va retourner une coloration du graphe en utilisant un certain nombre de couleurs qui n'est pas forcément le nombre de couleurs optimal. Une fois le nœud racine rempli, il s'agit de passer à la construction du nœud suivant (le niveau suivant dans l'arbre) selon l'ordre de numérotation dans le graphe. L'algorithme construit le nœud suivant en se basant sur les couleurs possibles qu'un nœud peut avoir selon ses voisins dans le graphe qui peuvent être déduits de la matrice d'adjacence. Avant de générer les nœuds fils, il faudrait évaluer le nœud courant pour savoir s'il pourrait contenir la solution optimale ou bien dans le cas contraire, qu'il ne mérite pas d'être parcouru et qu'il faudrait l'élaguer. Le principe de séparation est ici réalisé grâce à l'attribution des couleurs possibles à chaque nœud puis selon chaque valeur, il s'agit de donner une attribution différente des nœuds inférieurs. Notons que la stratégie de parcours est en DFS. Nous présentons dans ce qui suit une illustration de notre arbre de recherche sur un graphe de 5 nœuds représenté par une matrice et le graphe coloré produit.

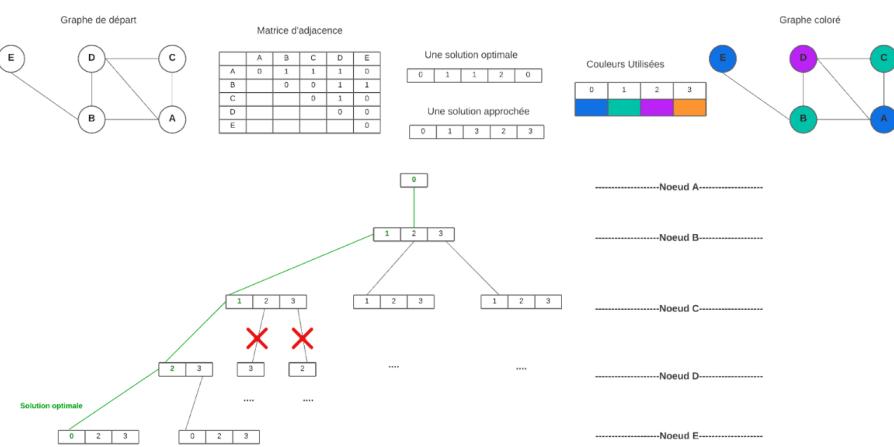


FIGURE 2.1 – Algorithme du branch and bound

Évaluation d'un nœud interne de l'arbre de recherche :

Un nœud interne de l'arbre représente une solution partielle que nous essayons d'estimer si elle peut contenir une meilleure solution. Si après estimation de ce nœud, on trouve qu'il ne contient pas une meilleure solution alors ce n'est pas la peine de parcourir les solutions de ce nœud et on passe au prochain nœud dans l'arbre pour l'explorer (c'est le principe d'élagage qui est détaillé dans le paragraphe suivant). L'évaluation d'un nœud dans notre cas consiste à calculer un minorant en termes du nombre de couleurs de toutes les solutions contenues dans ce nœud. La fonction d'évaluation fonctionne comme suite : Sachant qu'un nœud dans l'arbre représente une solution partielle, celle d'avoir x nœuds colorés dans le graphe avec k couleurs ($k \geq x$), alors pour colorer le graphe, y compris les nœuds non colorés, on doit obligatoirement utiliser au moins k couleurs, donc k est un minorant en termes du nombre de couleurs de toutes les solutions que peut contenir ce nœud. En résumé : évaluation d'un nœud = nombre de couleurs utilisées pour colorer les nœuds du graphe jusque là.

Élagage d'un nœud :

On élague un nœud si son évaluation retourne une valeur plus grande que la solution courante. On rappelle que l'évaluation d'un nœud retourne un minorant pour le nombre de couleurs à utiliser pour colorer tous les nœuds du graphe. Si on explore un nœud dont l'évaluation a retourné une valeur plus grande que la solution courante i.e. on va utiliser un nombre de couleurs supérieur ou égale au nombre de couleurs utilisées dans une solution déjà trouvée, alors que l'objectif est d'améliorer cette solution en réduisant le nombre de couleurs utilisé et non pas en l'augmentant, pour cela, ce n'est pas la peine de parcourir les solutions de ce nœud, qui ne vont en aucun cas améliorer la solution déjà trouvée et c'est ce qui nous fait un gain en temps.

Trouver la meilleure solution :

Une solution est considérée comme meilleure que celle déjà trouvée si le nombre de couleurs utilisées dans cette solution est inférieur à celui utilisé dans la solution courante. De plus, comme les feuilles de l'arbre représentent les solutions alors la considération d'une solution n'est effectuée que si on arrive à une feuille de l'arbre.

2.1.2 Tests (captures du temps d'exécution) :

Dans les tests suivants, nous allons utiliser des instances de graphes connues pour le problème de coloration de graphe.

Instance queen5_5.col (25 nœuds) : 0.00157 secondes

```

└─(chenx3n㉿shellmates)-[~/.../OPT/Projet]
└─$ python3 color_graph.py datasets/queen5_5.col
Solution obtenue avec l'algorithme glouton :
- Nombre de couleurs utilisés : 8
- Coloration : [0, 1, 2, 3, 4, 2, 3, 0, 1, 2, 3, 4, 2, 3, 0, 1, 2, 3, 4, 2, 3, 0, 1, 2, 3, 4, 2, 3, 0, 1, 2, 3, 4]
Solution obtenue avec l'algorithme Branch & Bound :
- Nombre de couleurs utilisés : 5
- Coloration : [0, 1, 2, 3, 4, 2, 3, 0, 1, 2, 3, 4, 2, 3, 0, 1, 2, 3, 4, 2, 3, 0, 1, 2, 3, 4, 2, 3, 0, 1, 2, 3, 4]
- Temps pris par l'algorithme: 0.00157

```

FIGURE 2.2 – Contenu bb du fichier config.py

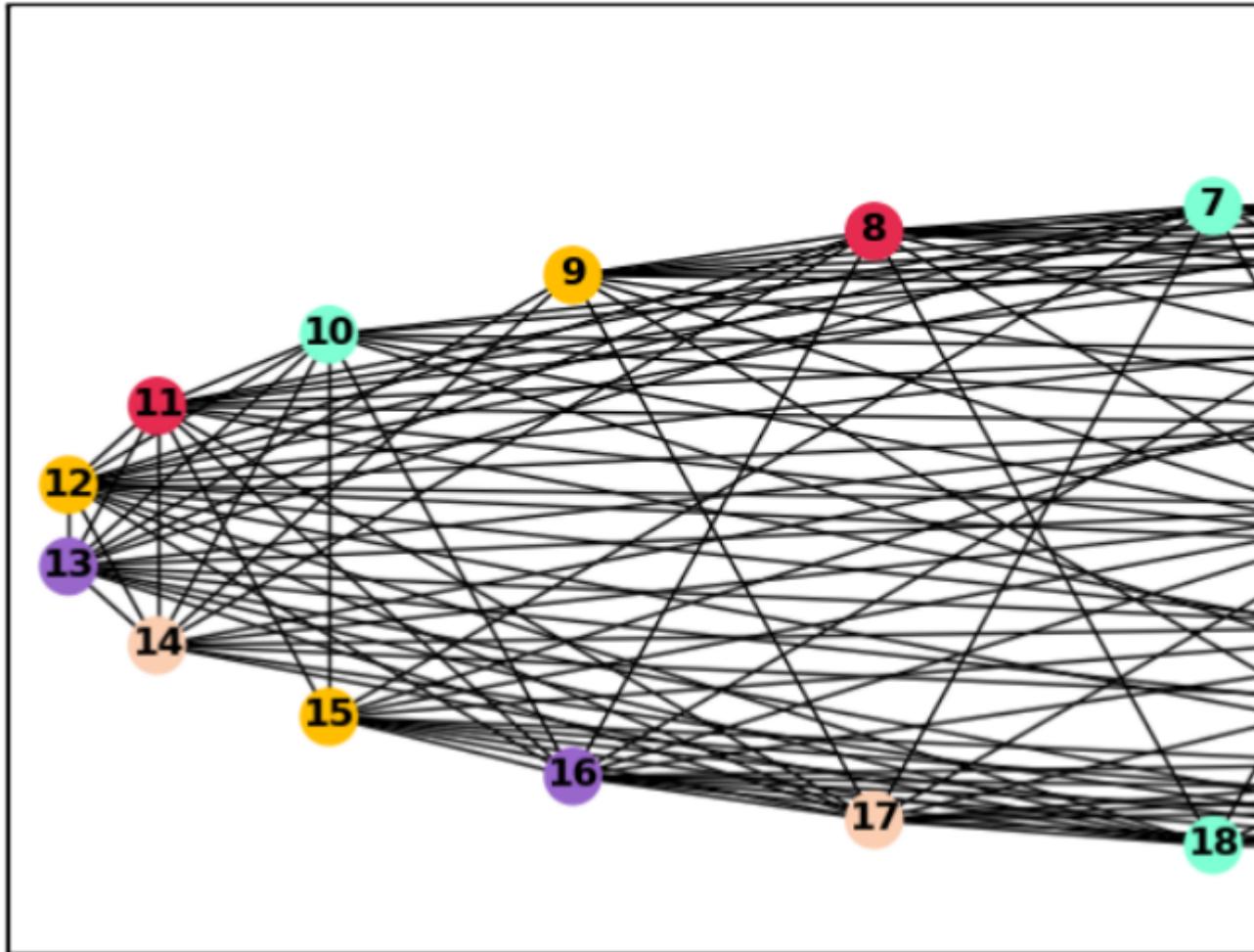


FIGURE 2.3 – Résultat du programme avec l’instance “queen5_5.col”

Instance myciel3.col (11 nœuds) : 0.00076 secondes

```
(chenx3n@shellmates) - [~]
$ python3 color_graph.py
Solution obtenue avec l'alg
- Nombre de couleurs ut
- Coloration : [0, 1, 0
Solution obtenue avec l'alg
- Nombre de couleurs ut
- Coloration : [0, 1, 0
- Temps pris par l'algo
```

FIGURE 2.4 – Résultat du programme avec l’instance “myciel3.col”

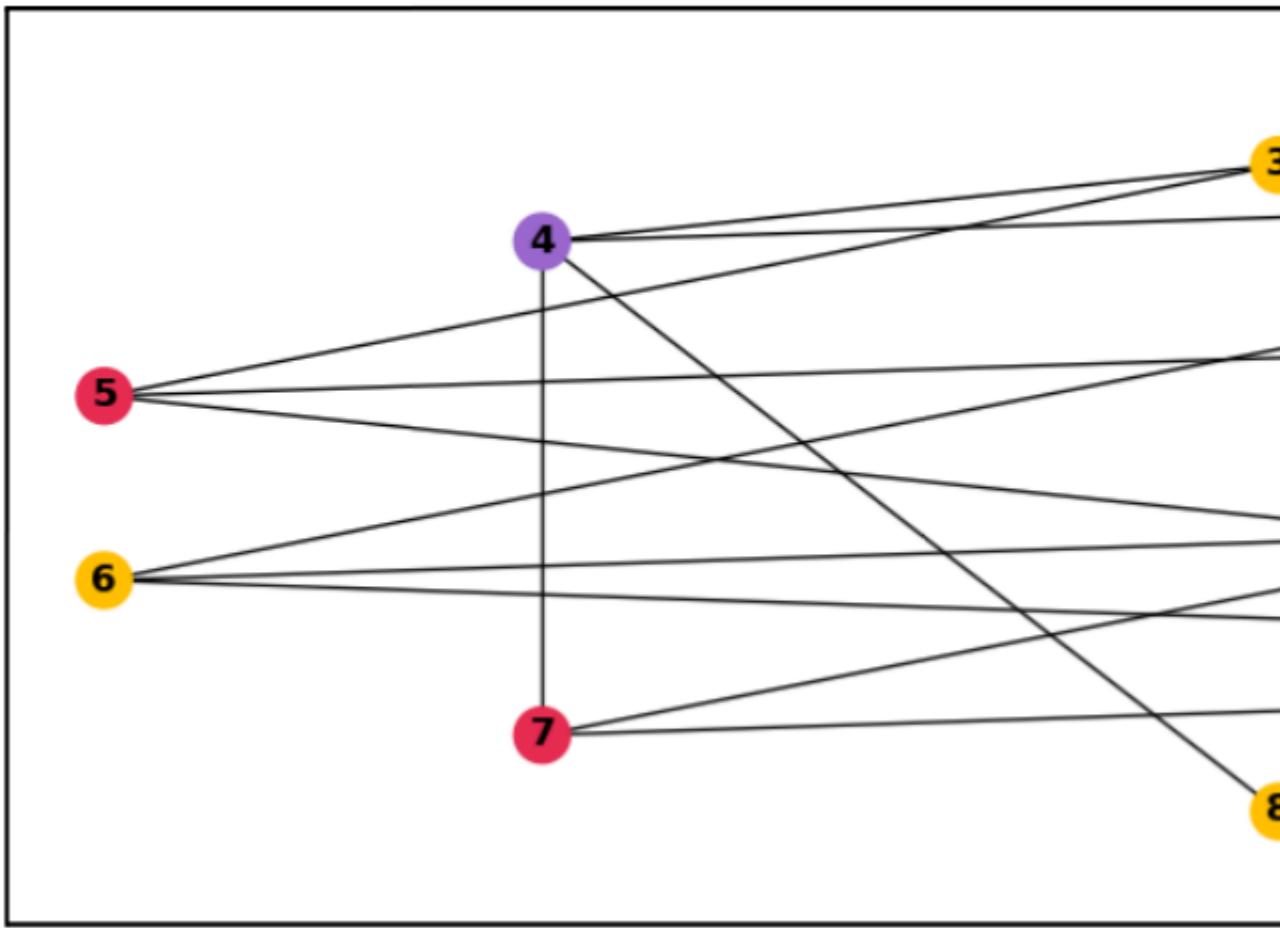


FIGURE 2.5 – Graphe de l’instance “myciel3.col” après coloration

Instance queen7_7.col (49 nœuds) : 0.7577 secondes

```
(chenx3n@shellmates) - [~/.../OPT/Pro]
$ python3 color_graph.py datasets/que
Solution obtenue avec l'algorithme gl
    - Nombre de couleurs utilisés : 10
    - Coloration : [0, 1, 2, 3, 4, 5,
2, 3, 7]
Solution obtenue avec l'algorithme Br
    - Nombre de couleurs utilisés : 7
    - Coloration : [0, 1, 2, 3, 4, 5,
2, 3, 4]
-Temps pris par l'algorithme: 0.7577
```

FIGURE 2.6 – Résultat du programme avec l’instance “queen7_7.col”

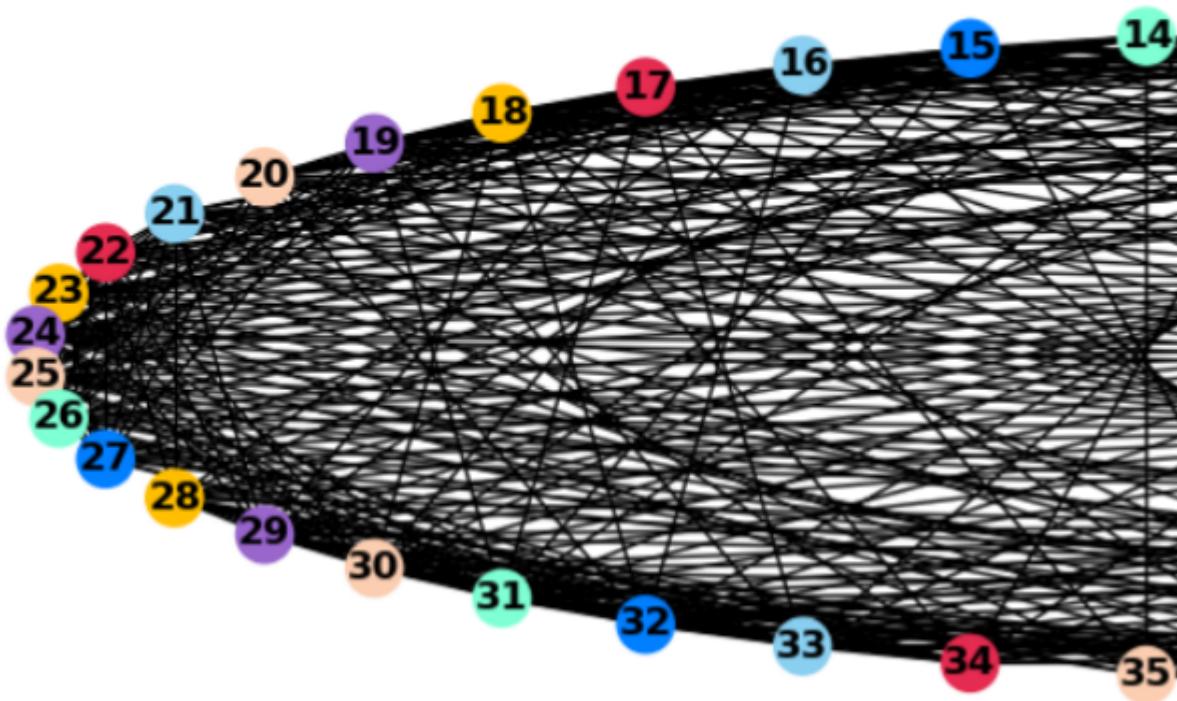


FIGURE 2.7 – Graphe de l’instance “queen7_7.col” après coloration

2.1.3 Code source et implémentation

Pour pouvoir implémenter notre solution avec Python, nous avons structuré notre projet comme ceci :

- config.py : fichier de configuration contenant des variables globales.
- graph.py : implémentation de la structure de données de graphe.
- color_graph.py : le cœur du projet, qui contient l’algorithme Branch Bound et le programme principal.
- colors.txt : fichier texte contenant des codes de couleurs hexadécimaux.
- datasets : dossier contenant plusieurs instances d’exemples de graphe (tirées de ce site web). Dans ce qui suit, nous allons présenter notre code Python commenté :

config.py :

```

from graph import Graph
import sys

# Fichier des couleurs
COLOR_FILE = "colors.txt"
# Fichier d'instance par défaut
GRAPH_FILE_DEFAULT = "datasets/queen5_5.col"
# Nombre de noeuds du graphe pour les graphes aléatoires
N = 25
# Générer un graphe aléatoire ou pas
RANDOM_GRAPH = False

# Si un argument est fourni, il est considéré comme étant le
# fichier d'instance du graphe
GRAPH_FILE = sys.argv[1] if len(sys.argv) > 1 else
GRAPH_FILE_DEFAULT

```

```

# RANDOM_GRAPH = True -> Générer un graphe aléatoire de taille N
if RANDOM_GRAPH:
    graph = Graph.rand_graph(N)
# RANDOM_GRAPH = False -> Lire le graphe à partir d'un fichier
else:
    graph = Graph.from_file(GRAPH_FILE)

```

FIGURE 2.8 – Contenu bb du fichier config.py

graph.py :

```

# RANDOM_GRAPH = True -> Générer un graphe aléatoire de taille N
if RANDOM_GRAPH:
    graph = Graph.rand_graph(N)
# RANDOM_GRAPH = False -> Lire le graphe à partir d'un fichier
else:
    graph = Graph.from_file(GRAPH_FILE)

```

```
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
import random

# Classe pour représenter un graphe
class Graph(list):
    SUBPLOT_NUM = 211
    TYPE_COMMENT = "c"
    TYPE_PROBLEM_LINE = "p"
    TYPE_EDGE_DESCRIPTOR = "e"

    # Retourne le nombre de noeuds du graphe
    def getVerticesNum(self):
        return len(self)

    # Retourne les noeuds voisins du noeud spécifié
    def getNeighbors(self, u):
        return [ v for v, j in enumerate(self[u]) if j != 0 ]

    # Dessiner le graphe avec matplotlib
    def draw(self, title=None, draw_func=nx.draw_circular,
color_key={}, default_color='lightblue', visualize=False):
        nxgraph = nx.from_numpy_matrix(np.array(self))
        node_color = [ color_key.get(v, default_color) for v in
nxgraph ]
        plt.subplot(Graph.SUBPLOT_NUM, title=title)
        Graph.SUBPLOT_NUM += 1
        draw_func(nxgraph, node_color=node_color, font_weight="bold",
with_labels=True)
        if visualize:
            plt.show()
```

```

# Lire et analyser une ligne d'un fichier d'instance de
graphe
@staticmethod
def parse_line(line):
    if line.startswith(Graph.TYPE_COMMENT):
        return Graph.TYPE_COMMENT, None
    elif line.startswith(Graph.TYPE_PROBLEM_LINE):
        _, _, num_nodes, num_edges = line.split(' ')
        return Graph.TYPE_PROBLEM_LINE, (int(num_nodes),
                                         int(num_edges))
    elif line.startswith(Graph.TYPE_EDGE_DESCRIPTOR):
        _, node1, node2 = line.split(' ')
        return Graph.TYPE_EDGE_DESCRIPTOR, (int(node1),
                                         int(node2))
    else:
        raise ValueError(f"Unable to parse '{line}'")

# Générer un graphe à partir d'un fichier d'instance de
graphe
@classmethod
def from_file(cls, filename):
    matrix = None

    with open(filename) as f:
        problem_set = False
        for line in f.readlines():
            line_type, val = Graph.parse_line(line.strip())
            if line_type == Graph.TYPE_COMMENT:
                continue
            elif line_type == Graph.TYPE_PROBLEM_LINE and not
problem_set:
                num_nodes, num_edges = val
                matrix = [ [0 for _ in range(num_nodes)] for _ in
range(num_nodes) ]
                problem_set = True
            elif line_type == Graph.TYPE_EDGE_DESCRIPTOR:
                if not problem_set:
                    raise RuntimeError("Edge descriptor found
before problem line")
                node1, node2 = val
                matrix[node1-1][node2-1] = 1

```

```

        matrix[node2-1][node1-1] = 1

    return cls(matrix)

    # Générer un graphe aléatoire de taille `n`
    @classmethod
    def rand_graph(cls, n):
        mat = [ [0 for _ in range(n)] for _ in range(n) ]
        for i in range(n):
            for j in range(i+1, n, 1):
                mat[i][j] = random.randint(0, 1)
                mat[j][i] = mat[i][j]
        return cls(mat)

```

FIGURE 2.9 – Contenu bb du fichier graph.py

color_graph.py :

```

from graph import Graph
from config import graph, COLOR_FILE
import matplotlib.pyplot as plt
import time
import random

# Retourne le nombre de couleurs utilisées par le graphe
def nbColorsUsed(coloredGraph):
    return len(set( coloredGraph ))

# Retourne une coloration du graphe avec l'algorithme glouton
def greedyColoring(graph) :
    numVertices = graph.getVerticesNum()
    # Initialisation
    coloring = [-1] * numVertices

    # Assigner la première couleur au premier noeud
    coloring[0] = 0

    available = [False] * numVertices

    # Assigner des couleurs aux noeuds restants
    for u in range(numVertices):
        # Traiter tous les noeuds adjacents et marquer leurs couleurs

```

```

comme non disponibles
    neighbors = graph.getNeighbors(u)
    for i in neighbors:
        if (coloring[i] != -1):
            available[coloring[i]] = True

    # Trouver la première couleur disponible
    cr = 0
    while cr < numVertices:
        if available[cr] == False:
            break

        cr += 1

    # Assigner la couleur trouvée
    coloring[u] = cr

    # Remettre les valeurs à faux pour la prochaine itération
    for i in neighbors:
        if coloring[i] != -1:
            available[coloring[i]] = False

    return coloring

# Fonction d'évaluation qui retourne le nombre de couleurs
utilisées jusqu'à maintenant
def evalFunc(graph, branch, nbColorsUsedSoFar) :
    return nbColorsUsedSoFar

# Retourne un tableau des couleurs possibles pour un noeud donné
non coloré en ayant déjà des noeuds colorés dans le graphe
def possibleColors(graph, branch, level, n) :
    neighbors = graph.getNeighbors(level)
    impossible_colors = [ branch[i] for i in range(level) if i in
neighbors ]
    return [ k for k in range(n) if k not in impossible_colors ]

# Mettre à jour les solutions
def updateSolutions(solutions, new) :
    cout, sol = new
    if cout not in solutions :
        solutions.clear()

```

```

        solutions[cout] = []
solutions[cout].append(sol)

# Algorithme Branch & Bound itératif
# - graph: graphe en entrée
# - heuristique_coloration: coloration avec heuristique
# - k: nombre de couleurs utilisées par l'heuristique de coloration
def BandB_it(graph, heuristique_coloration, k) :
    # Nombre de noeuds du graphe
    n = graph.getVerticesNum()
    # Solution courante (nombre de couleurs utilisées par
    l'heuristique)
    currSol = k
    # Branche représentante de la coloration de graphe
    branch = [ -1 for i in range(n) ]
    # Couleurs possibles pour le premier noeud (racine)
    # (Pour éviter de faire un traitement multiple,
    # on force seulement une couleur possible pour la racine)
    root = [ 0 ]
    # Dictionnaire des solutions
    solutions = {k: [ heuristique_coloration ]}

    # Pile de tuples structurés comme ceci :
    # 1. Niveau de l'arbre de solutions
    # 2. Indice de la couleur actuelle
    # 3. Tableau de couleurs possibles pour le noeud actuel
    nodeStack = [ (0, 0, root) ]

    # Tant que la pile n'est pas vide
    while len(nodeStack) != 0 :
        child = None

        # Dépiler de la pile
        level, indCurrColor, currNode = nodeStack.pop()
        currColor = currNode[indCurrColor]

        # Mettre à jour la branche
        branch[level] = currColor
        for i in range(n-1, level, -1):
            branch[i] = -1

        # Nombre de couleurs utilisées jusqu'à maintenant

```

```

nbColorsUsedSoFar = nbColorsUsed(branch[:level+1])

# Si c'est une feuille
if level == n-1:
    # Et le nombre de couleurs utilisées est moins que la
    solution courante
    if nbColorsUsedSoFar < currSol:
        # Mettre à jour la solution courante
        currSol = nbColorsUsedSoFar
        updateSolutions(solutions, (currSol,
branch.copy()))

    # Sinon si la fonction d'évaluation retourne une valeur
    meilleure que la solution courante,
    # continuer à explorer
elif evalFunc(graph, branch, nbColorsUsedSoFar) < currSol:
    # construire le noeud fils
    child = possibleColors(graph, branch, level+1, k)

    # Élaguer
else:
    continue

# Si l'indice de la couleur courante + 1 est inférieur à la
taille
# du tableau de couleurs possibles pour le noeud courant,
# on empile
if indCurrColor + 1 < len(currNode):
    nodeStack.append((level, indCurrColor+1, currNode))
# Si le fils n'est pas vide, on empile
if child != None:
    nodeStack.append((level+1, 0, child))

# Retourner les solutions
return solutions

# Retourne une liste de couleurs à partir d'un fichier texte de
couleurs
def parse_colors(color_file, randomize=False):
    with open(color_file) as f:
        colors = f.read().split('\n')
    randomize and random.shuffle(colors)

```

```

    return colors

if __name__ == "__main__":
    # Graphe en entrée
    g = Graph(graph)
    # Nombre de noeuds du graphe
    n = g.getVerticesNum()
    # Dessiner le graphe
    g.draw(title="Graphe en entrée")
    # Solution (coloration) retournée par l'algorithme glouton
    heuristique_coloration = greedyColoring(graph)
    # Nombre de couleurs utilisées par l'heuristique de
    coloration
    k = nbColorsUsed(heuristique_coloration)
    print("Solution obtenue avec l'algorithme glouton :")
    print(f" - Nombre de couleurs utilisés : {k}")
    print(f" - Coloration : {heuristique_coloration}")

    # Démarrer le minuteur
    t = time.time()
    # Exécuter l'algorithme Branch & Bound itératif
    solutions = BandB_it(g, heuristique_coloration, k)
    # Mesure du temps pris par l'algorithme Branch & Bound
    dt = time.time() - t

    # Affichage de la solution
    nbColors = min(solutions)
    coloring = solutions[nbColors][0]
    print("Solution obtenue avec l'algorithme Branch and Bound
itératif :")
    print(f" - Nombre de couleurs utilisés : {nbColors}")
    print(f" - Coloration : {coloring}")
    print(f" - Temps pris par l'algorithme: {dt} secondes")

    # Génération des couleurs pour afficher le résultat
    graphiquement
    colors = parse_colors(COLOR_FILE)
    color_key = {
        i: colors[color_idx]
        for i, color_idx in enumerate(coloring)
    }
    g.draw(color_key=color_key, title="Graphe coloré")

```

FIGURE 2.10 – Contenu bb du fichier color-graph.py

Chapitre 3

Méthodes approchées : Heuristiques

3.1 Introduction :

Après avoir traité la première catégorie de méthodes servant à résoudre le problème de coloration des graphes de façon optimale, on a constaté qu'avec ces méthodes exactes, le temps de calcul croît exponentiellement avec la taille -soit le nombre de sommets- du graphe. Pour contrer cela, d'autres méthodes ont été implémentées, celles-ci sont constructives et permettent de donner rapidement une approximation de la solution optimale du problème. Nous aborderons dans la suite de ce rapport trois méthodes heuristiques différentes tout en décrivant leur implémentation respective.

3.2 Explication des algorithmes :

3.2.1 Coloration gloutonne (Greedy algorithm)

Dans l'étude des problèmes de coloration de graphes, une coloration dite gloutonne - ou coloration séquentielle -, est une coloration des sommets d'un graphe obtenue par un algorithme glouton qui examine les sommets du graphe en séquence et attribue à chaque sommet la première couleur disponible. Les colorations gloutonnes peuvent être obtenues en temps linéaire, mais n'utilisent généralement pas un nombre minimum de couleurs.

Différents choix pour la séquence (l'ordre) des sommets produisent des colorations différentes du graphe. Les stratégies couramment utilisées pour fixer l'ordre des sommets impliquent de choisir les sommets de degré élevé avant les sommets de degré inférieur.

La coloration gloutonne pour un ordre de sommets donné peut être calculée par un algorithme qui s'exécute en temps linéaire. L'algorithme traite les sommets dans l'ordre donné en attribuant une couleur à chacun au fur et à mesure qu'il est traité. Les couleurs peuvent être représentées par les entiers $0, 1, 2, \dots$, et chaque sommet reçoit la couleur correspondant au plus petit entier non utilisé par l'un de ses voisins. Pour traiter la plus petite couleur disponible, on peut utiliser un tableau qui compte le nombre de voisins de chaque couleur (ou qui représente l'ensemble des couleurs des voisins), puis parcourir le tableau pour trouver l'indice de sa première valeur nulle.

3.2.2 DSatur :

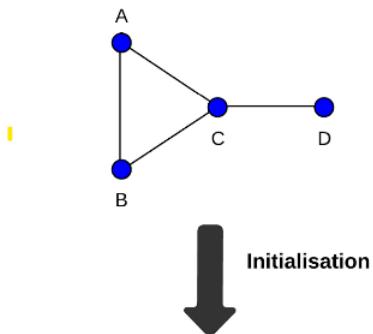
DSatur est une heuristique qui vise à colorer les noeuds selon un certain ordre, en effet, il a pour but de colorer les noeuds les plus contraignants en premier, à l'encontre de l'algorithme de coloration gloutonne qui attribue des couleurs aux noeuds avec un ordre aléatoire.

Pseudo algorithme :

- Tant qu' il y a des noeuds non colorés :
- Trouver le noeud dont la saturation est maximale (si plusieurs noeuds ont une saturation maximale, choisir le noeud avec le plus grand degré)
- Colorer le noeud avec la plus petite couleur disponible
- Mettre à jour les saturations des noeuds
- Où :

Saturation d'un noeud : est le nombre de couleurs utilisées pour colorer les voisins.

Exemple :

Graphe à colorer :**Coloration :**

Noeud	A	B	C	D
degré	0	0	0	0

Tableau de saturation :

Noeud	A	B	C	D
degré	0	0	0	0

Tableau des degrés :

Noeud	A	B	C	D
degré	2	2	3	1

C est le premier noeud à colorer**Coloration :**

Noeud	A	B	C	D
degré	0	0	1	0

Mettre à jour le tableau de saturation :

Noeud	A	B	C	D
degré	1	1	0	1

Tableau des degrés pour choisir le prochain noeud à colorer :

Noeud	A	B	C	D
degré	2	2	3	1

Le prochain noeud à colorer est A ou B, on choisit un noeud arbitrairement

Coloration :

Noeud	A	B	C	D
degré	2	0	1	0

Mettre à jour le tableau de saturation :

Noeud	A	B	C	D
degré	1	2	0	1

B est le prochain noeud à colorer

Coloration :

Noeud	A	B	C	D
degré	2	3	1	0

Mettre à jour le tableau de saturation :

Noeud	A	B	C	D
degré	1	2	0	1

D est le prochain et dernier noeud à colorer

Coloration :

Noeud	A	B	C	D
degré	2	3	1	2

Mettre à jour le tableau de saturation :

Noeud	A	B	C	D
degré	1	2	0	1

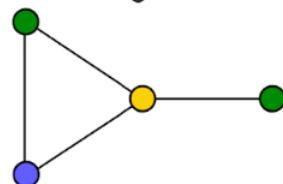


FIGURE 3.1 – Algorithme DSatur

3.2.3 Max stable (avec coloration gloutonne)

Principe général :

Il est difficile, voire impossible, de trouver un k-coloring d'un grand graphe G (par exemple —V— 1000) avec k proche de (G) en appliquant directement un algorithme donné sur G. Pour colorier les grands et très grands graphes, il existe une approche possible, en appliquant en premier un prétraitement pour extraire i grands ensembles indépendants (i stables maximales) du graphe afin d'obtenir un graphe résiduel beaucoup plus petit qui devrait être plus facile à colorier que le graphe initial. Un algorithme de coloration est ensuite invoqué pour colorier le graphe résiduel.

Puisque chacun des i ensembles forme une classe de couleur, 1 + le nombre de couleurs nécessaires pour le graphe résiduel donne une borne supérieure pour (G).

Les méthodes conventionnelles pour la phase de prétraitement opèrent de manière greedy en extrayant à chaque fois un ensemble indépendant du graphe G. Puis en supprimant de G les sommets de l'ensemble indépendant ainsi que leurs arêtes incidentes. Nous pouvons affiner le choix de l'ensemble indépendant à supprimer en prenant l'ensemble indépendant qui est connecté au plus grand nombre possible de sommets du graphe restant.

Un k-coloring légal d'un graphe donné $G = (V, E)$ correspond à une partition de V en k ensembles indépendants. Supposons que l'on veuille colorier G avec k couleurs. Supposons maintenant que nous extrayons $t \leq k$ ensembles indépendants I_1, \dots, I_t de G. Il est évident que si nous pouvions maximiser le nombre de sommets couverts par les t ensembles indépendants (c'est-à-dire que $|I_1| \dots |I_t|$ est aussi grand que possible), nous obtiendrons un graphe résiduel avec moins de sommets lorsque $I_1 \dots I_t$ sont retirés de G. Cela pourrait à son tour entraîner une diminution du nombre de sommets et en se ramenant à une borne supérieure de coloration inférieure ou égale à $k - t$ couleurs car il reste moins de sommets

dans le graphe résiduel.

Définition d'un max stable et complexité algorithmique :

Nous avons utilisé une heuristique qui se base principalement sur la relation entre le nombre chromatique et le cardinal de l'ensemble indépendant maximal. $(G) + (G) \leq n+1$ où n est le nombre de sommets du graphe.

Preuve par Induction : On remarque qu'un seul sommet a $(G) + (G) = 2$. On ajoute ensuite chaque sommet un par un. Remarquez qu'en ajoutant un seul sommet, on ne peut les augmenter que d'une unité. En faisant cela, nous convertissons G en G^* . Disons que le nouveau sommet v est connecté aux sommets 1 à r de notre G précédent et n'est pas connecté aux sommets $r+1$ à n . Pour que l'inégalité soit résolue conformément à notre hypothèse d'induction, les deux doivent être augmentés en un point.

Nous comprenons aussi que ce problème est NP-complet, mais il existe un algorithme séquentiel très efficace (en terme de temps d'exécution) permettant de calculer ce stable. La complexité est clairement $O(m)$ où m est l'ordre du graphe, or d'un point de vue technique cet algorithme peut très bien être parallélisé et donc optimisé pour les graphes très larges.

Algorithme d'extraction du stable maximum :

Étant donné un graphe $G(V,E)$, il est facile de trouver un seul MIS en utilisant l'algorithme suivant :

- 1- Initialiser I à un ensemble vide
- 2- Tant que V n'est pas vide :
 - Choisir un nœud vV ;
 - Ajouter v à l'ensemble I ;
 - Retirer de V le nœud v et tous ses voisins.
- 3- Retourner

3.3 Tests (captures du temps d'exécution) :

Dans les tests suivants, nous allons utiliser des instances de graphes connues pour le problème de coloration de graphe.

3.3.1 Instance queen5_5.col (25 nœuds) :

Coloration gloutonne : 0.0000672 secondes

```
Solution obtenue avec l'algorithme 'Coloration gloutonne' :
- Nombre de couleurs utilisées : 8
- Coloration : [0, 1, 2, 3, 4, 2, 3, 0, 1, 5, 1, 4, 5, 2, 0, 5, 0, 1, 4, 3, 3, 6, 7, 0, 1]
- Temps pris par l'algorithme : 6.723403930664062e-05 seconde(s)
```

FIGURE 3.2 – Résultat de l'algorithme “Coloration gloutonne” avec l'instance “queen5_5.col”

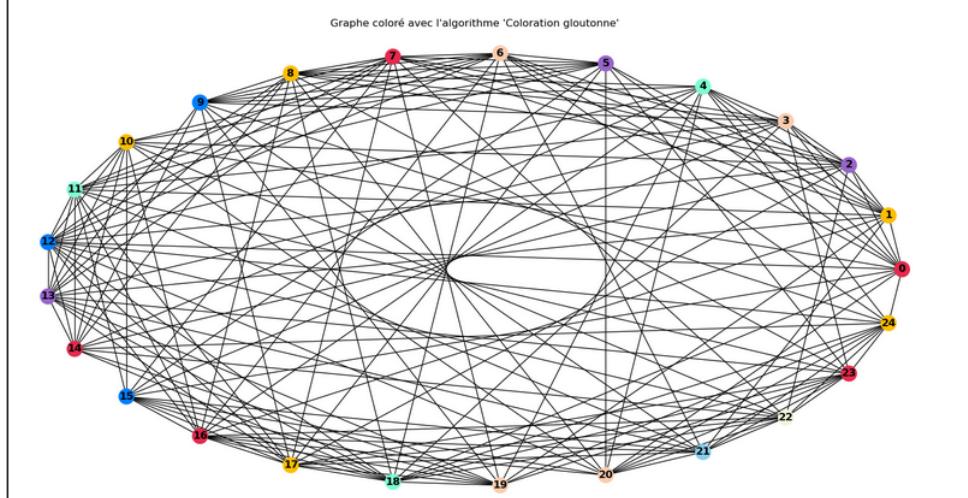


FIGURE 3.3 – Graphe de l'instance “queen5_5.col” après coloration avec l'algorithme “Coloration gloutonne”

DSatur : 0.0006904 secondes

```
Solution obtenue avec l'algorithme 'DSatur' :
- Nombre de couleurs utilisées : 5
- Coloration : [4, 1, 5, 2, 3, 5, 2, 3, 4, 1, 3, 4, 1, 2, 1, 5, 2, 1, 5, 2, 3, 4, 2, 3, 4, 1, 5]
- Temps pris par l'algorithme : 0.000690460205078125 seconde(s)
```

FIGURE 3.4 – Résultat de l'algorithme “DSatur” avec l'instance “queen5_5.col”

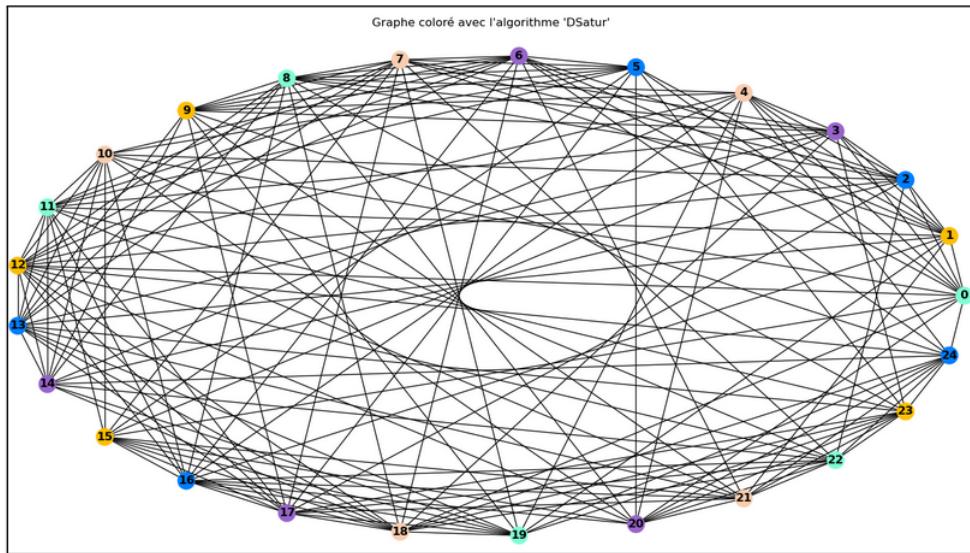


FIGURE 3.5 – Graphe de l'instance “queen5_5.col” après coloration avec l'algorithme “DSatur”

Max stable : 0.0021884 secondes

```
Solution obtenue avec l'algorithme 'Max stable' :
- Nombre de couleurs utilisées : 7
- Coloration : [0, 2, 4, 1, 3, 1, 6, 3, 0, 2, 3, 0, 2, 4, 1, 2, 1, 5, 3, 0, 4, 3, 0, 2, 5]
- Temps pris par l'algorithme : 0.002188444137573242 seconde(s)
```

FIGURE 3.6 – Résultat de l'algorithme “Max stable” avec l'instance “queen5_5.col”

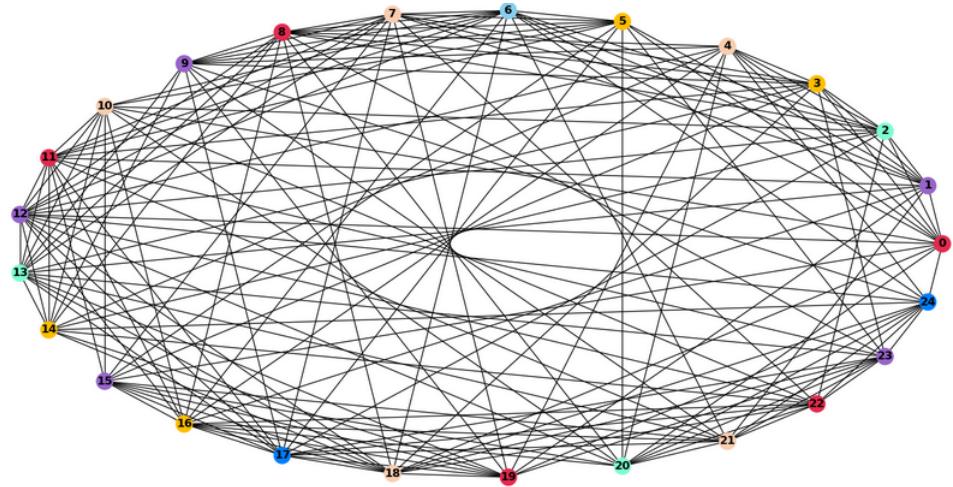


FIGURE 3.7 – Graphe de l’instance “queen5_5.col” après coloration avec l’algorithme “Max stable”

3.3.2 Instance myciel4.col (23 noeuds) :

Coloration gloutonne : 0.0000457 secondes

```
Solution obtenue avec l'algorithme 'Coloration gloutonne' :
- Nombre de couleurs utilisées : 5
- Coloration : [0, 1, 0, 1, 2, 0, 1, 0, 1, 2, 3, 0, 1, 0, 1, 2, 0, 1, 0, 1, 2, 3, 4]
- Temps pris par l'algorithme : 4.57763671875e-05 seconde(s)
```

FIGURE 3.8 – Résultat de l’algorithme “Coloration gloutonne” avec l’instance “myciel4.col”

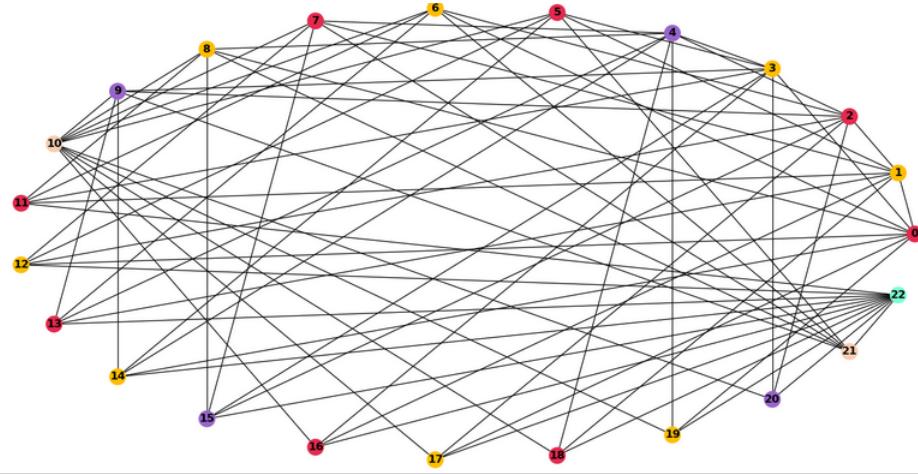


FIGURE 3.9 – Graphe de l’instance “myciel4.col” après coloration avec l’algorithme “Coloration gloutonne”

DSatur : 0.0003023 secondes

```
Solution obtenue avec l'algorithme 'DSatur' :
- Nombre de couleurs utilisées : 5
- Coloration : [3, 2, 3, 2, 4, 1, 1, 1, 1, 1, 2, 3, 2, 3, 2, 4, 3, 4, 3, 5, 4, 2, 1]
- Temps pris par l'algorithme : 0.00030231475830078125 seconde(s)
```

FIGURE 3.10 – Résultat de l’algorithme “DSatur” avec l’instance “myciel4.col”

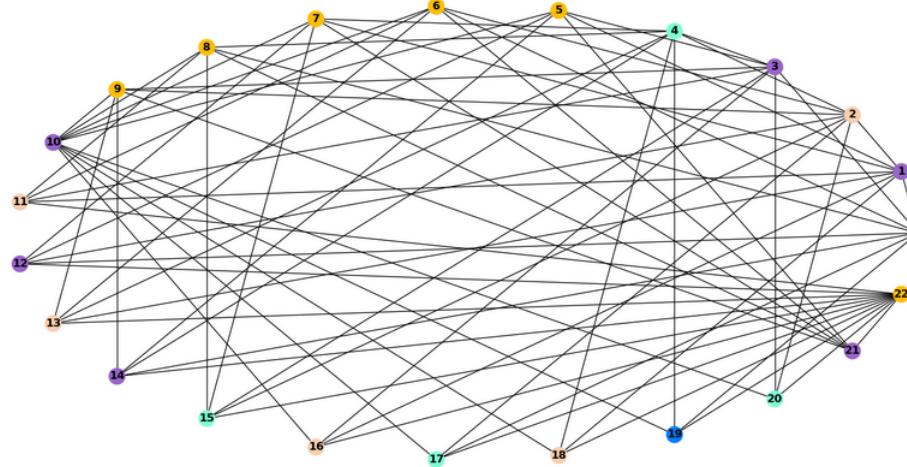


FIGURE 3.11 – Graphe de l’instance “myciel4.col” après coloration avec l’algorithme “DSatur”

Max stable : 0.0018234 secondes

```
Solution obtenue avec l'algorithme 'Max stable' :
- Nombre de couleurs utilisées : 5
- Coloration : [2, 3, 2, 3, 4, 1, 1, 1, 1, 1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
- Temps pris par l'algorithme : 0.00182342529296875 seconde(s)
```

FIGURE 3.12 – Résultat de l’algorithme “Max stable” avec l’instance “myciel4.col”

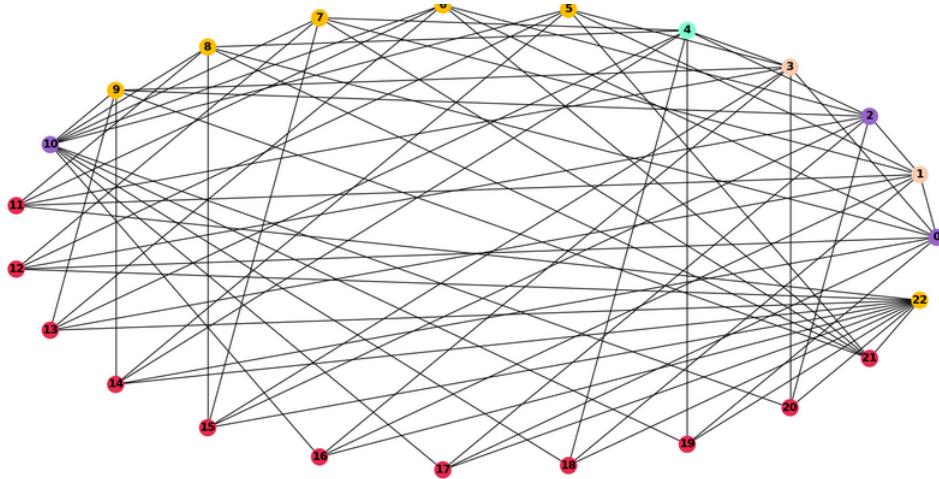


FIGURE 3.13 – Graphe de l’instance “myciel4.col” après coloration avec l’algorithme “Max stable”

3.4 Code source et implémentation

Pour pouvoir implémenter notre solution avec Python, nous avons structuré notre projet comme ceci :

- config.py : fichier de configuration contenant des variables globales.
- graph.py : implémentation de la structure de données de graphe.
- color_graph.py : le cœur du projet, qui contient les algorithmes heuristiques et le programme principal.
- colors.txt : fichier texte contenant des codes de couleurs hexadécimaux.
- datasets : dossier contenant plusieurs instances d'exemples de graphe tirée de ce site web.

Dans ce qui suit, nous allons présenter notre code Python commenté :

config.py

```
from graph import Graph
import sys

# Fichier des couleurs
COLOR_FILE = "colors.txt"
# Fichier d'instance par défaut
GRAPH_FILE_DEFAULT = "datasets/myciel4.col"
# Nombre de noeuds du graphe pour les graphes aléatoires
N = 25
# Générer un graphe aléatoire ou pas
RANDOM_GRAPH = False

# Si un argument est fourni, il est considéré le fichier
# d'instance du graphe
GRAPH_FILE = sys.argv[1] if len(sys.argv) > 1 else
GRAPH_FILE_DEFAULT

# RANDOM_GRAPH = True -> Générer un graphe aléatoire de taille N
if RANDOM_GRAPH:
    GRAPH = Graph.rand_graph(N)
# RANDOM_GRAPH = False -> Lire le graphe à partir d'un fichier
else:
    GRAPH = Graph.from_file(GRAPH_FILE)
```

FIGURE 3.14 – Contenu du fichier config.py (heuristiques)

graph.py

```

import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
import random

# Classe pour représenter un graphe
class Graph(list):
    SUBPLOT_NUM = 411
    TYPE_COMMENT = "c"
    TYPE_PROBLEM_LINE = "p"
    TYPE_EDGE_DESCRIPTOR = "e"

    # Retourne le nombre de noeuds du graphe
    def getVerticesNum(self):
        return len(self)

    # Retourne les noeuds voisins du noeud spécifié
    def getNeighbors(self, u):
        return [v for v, j in enumerate(self[u]) if j != 0]

    # Dessiner le graphe avec matplotlib
    def draw(
        self,
        title=None,

```

```

        draw_func=nx.draw_circular,
        color_key={},
        default_color="lightblue",
        visualize=False,
    ):
        nxgraph = nx.from_numpy_matrix(np.array(self))
        node_color = [color_key.get(v, default_color) for v in
nxgraph]
        plt.subplot(Graph.SUBPLOT_NUM, title=title)
        Graph.SUBPLOT_NUM += 1
        draw_func(nxgraph, node_color=node_color, font_weight="bold",
with_labels=True)
        if visualize:
            plt.show()

    # Lire et analyser une ligne d'un fichier d'instance de
graphe
    @staticmethod
    def parse_line(line):

```

```

if line.startswith(Graph.TYPE_COMMENT):
    return Graph.TYPE_COMMENT, None
elif line.startswith(Graph.TYPE_PROBLEM_LINE):
    _, _, num_nodes, num_edges = line.split(" ")
    return Graph.TYPE_PROBLEM_LINE, (int(num_nodes),
int(num_edges))
elif line.startswith(Graph.TYPE_EDGE_DESCRIPTOR):
    _, node1, node2 = line.split(" ")
    return Graph.TYPE_EDGE_DESCRIPTOR, (int(node1),
int(node2))
else:
    raise ValueError(f"Unable to parse '{line}'")

# Générer un graphe à partir d'un fichier d'instance de
graphe
@classmethod
def from_file(cls, filename):
matrix = None

with open(filename) as f:
    problem_set = False
    for line in f.readlines():
        line_type, val = Graph.parse_line(line.strip())
        if line_type == Graph.TYPE_COMMENT:
            continue
        elif line_type == Graph.TYPE_PROBLEM_LINE and not

```

```

problem_set:
    num_nodes, num_edges = val
    matrix = [[0 for _ in range(num_nodes)] for _ in
range(num_nodes)]
    problem_set = True
    elif line_type == Graph.TYPE_EDGE_DESCRIPTOR:
        if not problem_set:
            raise RuntimeError("Edge descriptor found
before problem line")
            node1, node2 = val
            matrix[node1 - 1][node2 - 1] = 1
            matrix[node2 - 1][node1 - 1] = 1

    return cls(matrix)

# Générer un graphe aléatoire de taille `n`
@classmethod

```

```

def rand_graph(cls, n):
    mat = [[0 for _ in range(n)] for _ in range(n)]
    for i in range(n):
        for j in range(i + 1, n, 1):
            mat[i][j] = random.randint(0, 1)
            mat[j][i] = mat[i][j]
    return cls(mat)

```

FIGURE 3.15 – Contenu du fichier graph.py (heuristiques)

color_graph.py

```

from config import GRAPH, COLOR_FILE
import matplotlib.pyplot as plt
import time
import random
import itertools
import numpy as np
import networkx as nx

# Retourne le nombre de couleurs utilisées par le graphe
def nbColorsUsed(coloredGraph):
    return len(set(coloredGraph))

# Retourne le nombre de noeuds ayant la couleur "color_number"
def get_amount_color(solution, node_indexes, color_number):
    color_counter = 0
    for index in node_indexes:
        if solution[index] == color_number:
            color_counter += 1

    return color_counter

```

```
# Retourne une coloration du graphe avec l'algorithme glouton
def greedyColoring(graph):
    numVertices = graph.getVerticesNum()
    # Coloration du graphe (résultat)
    coloring = [-1] * numVertices

    # Assigner la première couleur au premier noeud
    coloring[0] = 0
```

```
available = [False] * numVertices

# Assigner des couleurs aux noeuds restants
for u in range(numVertices):
    # Traiter tous les noeuds adjacents et marquer leurs couleurs
    # comme non disponibles
    neighbors = graph.getNeighbors(u)
    for i in neighbors:
        if coloring[i] != -1:
            available[coloring[i]] = True

    # Trouver la première couleur disponible
    cr = 0
    while cr < numVertices:
        if available[cr] == False:
            break

        cr += 1

    # Assigner la couleur trouvée
    coloring[u] = cr

    # Remettre les valeurs à faux pour la prochaine itération
    for i in neighbors:
        if coloring[i] != -1:
            available[coloring[i]] = False
```

```

    return coloring

# Algorithme de coloration de graphe en utilisant l'heuristique
DSatur
def dSatur(graph):
    color_counter = 1

    degrees = list()
    saturation_degrees = [0] * len(graph)

    solution = [0] * len(graph)
    uncolored_vertices = set(range(len(graph)))

    index_maximum_degree = 0
    maximum_degree = 0

```

```

for index_node in range(len(graph)):
    # Calculer le degré des noeuds du graphe en entrée
    degrees.append((sum(graph[index_node]), index_node))

    # Et trouver le noeud de degré maximal en même temps
    if degrees[index_node][0] > maximum_degree:
        (maximum_degree, node_index) = degrees[index_node]
        index_maximum_degree = index_node

    # Mise à jour de la saturation
    neighbors = graph.getNeighbors(index_maximum_degree)
    for index_neighbor in neighbors:
        saturation_degrees[index_neighbor] += 1

    # Coloration du premier noeud
    solution[index_maximum_degree] = color_counter
    uncolored_vertices.remove(index_maximum_degree)

while len(uncolored_vertices) > 0:
    # Trouver le degré de saturation maximal
    maximum_satur_degree = -1
    for index in uncolored_vertices:
        if saturation_degrees[index] > maximum_satur_degree:
            maximum_satur_degree = saturation_degrees[index]

    # Récupérer la liste des noeuds avec le degré de saturation
    . .

```

```

indexes_maximum_satur_degree = [
    index
    for index in uncolored_vertices
    if saturation_degrees[index] == maximum_satur_degree
]

coloring_index = indexes_maximum_satur_degree[0]
if (
    len(indexes_maximum_satur_degree) > 1
): # Si plus d'un noeud ayant un degré de saturation maximal
    # Alors choisir le noeud de degré maximal pour le
colorer prochainement
    maximum_degree = -1
    for index in indexes_maximum_satur_degree:
        (degree, node_index) = degrees[index]
        if degree > maximum_degree:

```

```

coloring_index = node_index
maximum_degree = degree

# Essayer de colorer du noeud choisi avec des couleurs déjà
utilisées
node_index_neighbors = graph.getNeighbors(coloring_index)
for number_color in range(1, color_counter + 1, 1):
    if get_amount_color(solution, node_index_neighbors,
number_color) == 0:
        solution[coloring_index] = number_color
        break
    else:
        # Si le noeud n'a pas encore été coloré (i.e il faut ajouter
        une nouvelle couleur pour pourvoir le colorier)
        if solution[coloring_index] == 0:
            color_counter += 1 # Ajouter une nouvelle couleur
            solution[coloring_index] = color_counter

# Le retirer des noeuds non colorés
uncolored_vertices.remove(coloring_index)

# Mise à jour des degrés de saturation
for index_neighbor in node_index_neighbors:
    subneighbors = graph.getNeighbors(index_neighbor)

```

```

        if get_amount_color(solution, subneighbors,
solution[coloring_index]) == 1:
            saturation_degrees[index_neighbor] += 1

    # Retourner la solution
    return solution

# Extraire le plus grand stable de G
def _maximal_independent_set(G):
    result = set()
    # Initialiser le graphe restant à G
    remaining = set(G)
    # Pour tout noeud dans le graphe restant
    while remaining:
        # Choisir le noeud avec le degré minimum (pour éviter la
        suppression de beaucoup de noeuds)
        G = G.subgraph(remaining)

```

```

v = min(remaining, key=G.degree)

# Ajouter le noeud v au stable
result.add(v)
# Enlever le noeud v du graphe ainsi que ses voisins
remaining -= set(G[v]) | {v}
return result

def strategy_independent_set(G):
    remaining_nodes = set(G)
    while len(remaining_nodes) > 0:
        nodes = _maximal_independent_set(G.subgraph(remaining_nodes))
        remaining_nodes -= nodes
        yield from nodes
    return nodes

# Heuristique spécifique pour le problème de coloration des
# graphes par le max stable
def graphColoringMaxStable(G):
    G = nx.from_numpy_array(np.array(G))
    colored_graph = G
    if len(G) == 0:
        return {}
    colors = {}

```

```

nodes = strategy_independent_set(G)
for u in nodes:
    # Ensemble des voisins du noeud courant
    neighbour_colors = {colors[v] for v in G[u] if v in colors}
    # Trouver la première couleur non utilisée
    for color in itertools.count():
        if color not in neighbour_colors:
            break
    # Donner la couleur au noeud courant
    colors[u] = color
    |

color_map = []
for node in colored_graph:
    color_map.append(colors[node])

return color_map

```

```

# Retourne une liste de couleurs à partir d'un fichier texte de
couleurs
def parse_colors(color_file, randomize=False):
    with open(color_file) as f:
        colors = f.read().split("\n")
    randomize and random.shuffle(colors)
    return colors

# Afficher les résultats d'un algorithme de coloration donné
def printAlgResults(algname, nbColors, coloration, timeTaken):
    print(f"Solution obtenue avec l'algorithme '{algname}' :")
    print(f"\t- Nombre de couleurs utilisées : {nbColors}")
    print(f"\t- Coloration : {coloration}")
    print(f"\t- Temps pris par l'algorithme : {timeTaken} seconde(s)")

# Générer les couleurs pour l'affichage à partir d'une coloration
def generateColorKey(coloration, colors=None,
colorfile=COLOR_FILE):
    if colors is None:
        colors = parse_colors(colorfile)
    return {i: colors[color_idx] for i, color_idx in
enumerate(coloration)}

```

```
# Exécuter un algorithme de coloration sur un graphe
def execColoringAlgorithm(graph, algname, algfunc):
    # Démarrer le minuteur
    t = time.time()
    # Exécuter l'algorithme de coloration
    coloration = algfunc(graph)
    # Mesure du temps pris par l'algorithme
    dt = time.time() - t
    # Nombre de couleurs utilisées par la coloration
    nbColors = nbColorsUsed(coloration)
    # Afficher les résultats de l'algorithme
    printAlgResults(algname, nbColors, coloration, dt)
    # Génération des couleurs
    color_key = generateColorKey(coloration)
    # Afficher le résultat graphiquement
```

```
title = f"Graphe coloré avec l'algorithme '{algname}'"
GRAPH.draw(color_key=color_key, title=title)

if __name__ == "__main__":
    GRAPH.draw(title="Graphe en entrée")

    # Exécuter l'algorithme de coloration gloutonne (Greedy
    algorithm)
    execColoringAlgorithm(GRAPH, "Coloration gloutonne",
greedyColoring)

    # Exécuter l'algorithme DSatur
    execColoringAlgorithm(GRAPH, "DSatur", dsatur)

    # Exécuter l'algorithme du max stable
    execColoringAlgorithm(GRAPH, "Max stable",
graphColoringMaxStable)

    # Afficher la fenêtre graphique
    plt.show()
```

FIGURE 3.16 – Contenu du fichier *colorgraph.py(heuristiques)*

Chapitre 4

Méthodes approchées : métahéuristiques

4.1 Explication des algorithmes :

4.1.1 Recherche Tabou :

Décrivons maintenant comment la technique de recherche tabou peut être utilisée pour trouver des colorations de grands graphiques. On va essentiellement essayer de trouver une coloration d'un graphe donné G qui utilise une nombre fixe k de couleurs. On peut alors faire varier k comme on veut.

Etant donné un graphe $G = (V, E)$ une solution réalisable sera une partition $s = (V_1, V_2, \dots, V_k)$ de l'ensemble de noeud V into un nombre fixe k de sous-ensembles. Si $E(V_i)$ est la collection des arêtes de G avec les deux extrémités dans V_i , nous pouvons définir la fonction objectif f comme le nombre de arêtes pour lesquelles les deux extrémités sont dans le même V_i (c'est-à-dire ont la même couleur) : $f(s) = \sum_{i=1}^k |E(V_i)|$.

Il est clair que s sera une coloration des noeuds de G avec k couleurs si et seulement si $f(s) = 0$. En fait on peut estimer la meilleure valeur possible de $f(s)$ avec $f^* = 0$; cela est la condition d'arrêt dans l'algorithme. À partir de s , nous générerons un voisin s' (c'est-à-dire une autre partition en k sous-ensembles de noeuds) comme suit : On choisit au hasard un noeud x parmi tous ceux qui sont adjacents à une arête dans $E(V_1) \cup \dots \cup E(V_k)$. Puis en supposant $x \in V_i$, on choisit une couleur aléatoire $j \in [1, k]$ et on obtenir s' à partir de $s = (V_1, \dots, V_k)$ en posant : Après avoir généré des voisins rep de s (qui ne conduisent pas à des

mouvements tabou), nous reprenons le meilleur et nous y passons. La liste tabou est obtenue comme suit : chaque fois qu'un nœud x est déplacé de V_i à V_j pour obtenir la nouvelle solution, le couple (x, i) devient tabou : le nœud x ne peut pas être ramené à V_i pour quelques itérations. Comme décrit précédemment, la liste T des mouvements tabou est cyclique.

Maintenant, nous allons continuer les itérations jusqu'à ce que soit nous obtenions une solution s telle que $f(s) = f^*$ ou jusqu'à atteindre le nombre maximum $nbmax$ d'itérations. Dans ce cas, on n'aura pas obtenu de coloration si pour la dernière solution s on a $f(s) \notin O$.

Nous utilisons une fonction $A(z)$ qui est le niveau d'aspiration de la prochaine valeur de la fonction objectif à atteindre lorsque la valeur courante est $z-f(s)$. Il est utilisé comme ceci : si un déplacement vers un voisin s' est tabou mais donne $f(s') \in A(z)$, alors on laisse tomber le statut tabou de ce mouvement et nous le considérons comme un membre normal de l'échantillon qui est généré. Initialement, nous fixons $A(z) = z - 1$ pour toutes les valeurs de z . Ensuite, chaque fois que nous générerons un s' avec $f(s') \in A(f(s))$ alors on pose $A(f(s)) = f(s') - 1$. La formulation complète de la technique de recherche tabou pour notre problème de coloration est donné dans le tableau 1 ; nous l'appellerons TABUCOL. Plusieurs améliorations locales ont été introduites pour réduire le calcul temps. Tout d'abord, pendant le processus de génération des voisins s' de s , nous pouvons obtenir à un certain stade un s' (pas dans la liste taboue) avec $f(s') \in f(s)$. Au lieu de continuer jusqu'à ce que nous ayons généré rep voisins, on passe directement de s à s' . Voici le pseudo algorithme :

```

Input    $G = (V, E)$ 
          $k = \text{number of colors}$ 
          $|T| = \text{size of tabu list}$ 
          $\text{rep} = \text{number of neighbours in sample}$ 
          $\text{nbmax} = \text{maximum number of iterations.}$ 

Initialization
    Generate a random solution  $s = (V_1, \dots, V_k)$ 
     $\text{nbiter} := 0$ ; choose an arbitrary tabu list  $T$ .

While  $f(s) > 0$  and  $\text{nbiter} < \text{nbmax}$ 
    generate rep neighbours  $s_l$  of  $s$  with move  $s \rightarrow s_l \notin T$  or  $f(s_l) \leq A(f(s))$ 
    (as soon as we get an  $s_l$  with  $f(s_l) < f(s)$  we stop the generation).
    Let  $s'$  be the best neighbour generated
    update tabu list  $T$ 
    (introduce move  $s \rightarrow s'$  and remove oldest tabu move)
     $s := s'$ 
     $\text{nbiter} := \text{nbiter} + 1$ 

endwhile

Output If  $f(s) = 0$ , we get a coloring of  $G$  with  $k$  colors:  $V_1, \dots, V_k$  are the color sets. Otherwise no
coloring has been found with  $k$  colors.

```

FIGURE 4.1 – Algorithme de la recherche Tabou

4.1.2 AG : Algorithme génétique :

Explication de l'AG :

L'algorithme génétique est une métaheuristique évolutionnaire inspirée de l'évolution. Cet algorithme se base sur le principe qu'en démarrant d'un nombre de couleurs K, qui est un majorant du nombre chromatique, on peut tester toutes les colorations inférieures à K c'est-à-dire tester K-1, K-2, ... jusqu'à trouver le nombre chromatique.

Comme initialisation, nous savons que le nombre chromatique est majoré par le plus grand degré + 1. Nous avons essayé dans notre algorithme de prendre le minimum entre le plus grand degré +1 et une coloration générée par un algorithme glouton Greedy. Cette amélioration permettra de réduire le nombre des itérations qui auront pour point de départ cette initialisation.

Nous avons défini la fitness comme étant le nombre de conflits que cette solution génère. Chaque conflit dans la solution incrémente la fitness de 1. Un conflit existe

lorsque deux noeuds voisins dans le graphe sont colorés de la même couleur.

Un individu est une solution, donc un tableau contenant la coloration du graphe. Les individus sont générés aléatoirement. Bien entendu, les solutions générées ne sont pas forcément toutes correctes.

Après avoir initialisé le point de départ des couleurs à tester, nous allons pour chaque itération initialiser la fitness comme étant celui du premier individu et initialiser ce dernier comme étant le meilleur individu. Nous allons ensuite créer une population contenant un certain nombre d'individus aléatoirement générés. Un nombre prédéfini d'itérations à effectué est initialisé. Tant que nous n'avons pas encore atteint ce nombre et que la meilleure fitness est non nulle, nous allons procéder aux quatre étapes de tout algorithme génétique :

- La sélection en tournoi : nous allons prendre une population de taille prédéfinie, mélanger l'ordre des individus, les confronter deux à deux et garder l'individu ayant la meilleure fitness et le sélectionner.
- Le croisement : utiliser un croisement à point unique et générer deux enfants à partir de chaque deux individus sélectionnés. Ces enfants remplaceront leurs parents dans la génération courante
- La mutation : nous allons créer une perturbation dans les individus résultant du croisement. Nous avons pour cela utilisé une probabilité de mutation que nous allons prendre assez petite pour permettre une diversification et ainsi éviter les minimums locaux.
- Mise à jour de la meilleure fitness et du meilleur individu dans la nouvelle population générée.

Une fois toutes ces étapes effectuées, nous allons tester la valeur de la meilleure fitness, deux cas se présentent :

- Si la meilleure fitness est non nulle, cela signifie que l'individu ayant le plus petit nombre de conflits n'est pas une solution. Nous sommes sortis de la boucle parce que nous avons atteint le nombre maximum d'itérations à effectuer pour la coloration courante. Cela signifie donc que le graphe n'est pas colorable avec

ce nombre de couleurs, notons le n . Cela implique donc que le graphe est $(n-1)$ colorable. Cela permet d'être presque sûr (car cela dépend du nombre d'itérations maximum plus il est grand plus nous sommes sûrs) que le graphe n'est pas n colorable puisque nous avons donner plusieurs chances à n pour trouver une solution.

- Si la fitness est nulle, alors le meilleur individu est une solution et donc le graphe est n colorable (avec n le nombre de couleurs courante à tester dans l'itération actuelle). Puisque le graphe est n colorable alors nous voudrions savoir s'il y a possibilité de le colorer avec $n-1$ couleurs, puisque nous cherchons le plus petit nombre de couleurs à utiliser pour colorer le graphe. Nous allons donc réitérer l'algorithme avec $n-1$ couleurs.

Il est à noter que pendant chaque itération parmi celle du nombre maximum des itérations à effectuer, nous allons générer une population. Pour récapituler, notons ci-dessous tous les paramètres à définir :

- Nombre d'itération maximum
- Nombre d'individus dans une génération (taille de la population)
- Probabilité de mutation

Chapitre 5

Etude des résultats

5.1 Etude empirique

5.1.1 Paramètres des algorithmes génétiques :

Chaque algorithme évolutionnaire (métaheuristique) a des paramètres à régler. L'algorithme génétique possède également certains paramètres. Les paramètres de l'algorithme génétique sont définis comme suit :

```
POPLUATION_SIZE = 200
# Pour l'effet de diversification
MUTATION_PROBABILITY = 0.4
# Nombre de générations (itérations) à produire (effectuer) à partir d'une population
GENERATIONS_NUM = 300
# TABU_NUMBER_OF_COLORS = 25
DEBUG = True
```

FIGURE 5.1 – Paramètres AG

L'image ci-dessus fait référence aux valeurs par défaut qui ont déjà été définies. On peut simplement copier ce code d'ici, changer les valeurs et utiliser le dictionnaire modifié comme argument de `geneticalgorithm`. Une autre façon d'accéder à ce dictionnaire est d'utiliser la commande ci-dessous :

Pour changer d'autres paramètres, il suffit de remplacer les valeurs des paramètres dans le fichier `config.py`.

@ GENERATIONS_NUM : Le critère de terminaison de GA. Si la valeur de ce paramètre est `None`, l'algorithme fixe automatiquement le nombre maximum d'itérations en fonction de la dimension, des limites et de la taille de la population. L'utilisateur peut entrer le nombre d'itérations qu'il souhaite par défaut c'est 300. Il est fortement recommandé que l'utilisateur détermine lui-même le `GENERATIONS_NUM` et de ne pas utiliser `None`.

@ POPULATION_SIZE : détermine le nombre de solutions d'essai dans chaque itération. La valeur par défaut est 200.

@ mutation_probability : détermine la probabilité que chaque gène de chaque solution individuelle soit remplacé par une valeur aléatoire. La valeur par défaut est 0,4 .

Effet de la MUTATION_PROBABILITY :

```
def mutation_probability_impact(g):
    new_list = range(math.floor(0), math.ceil(g.num_vertices + 1))
    plt.yticks(new_list)

    num_iterations = 50
    probabilities = [0.0, 0.2, 0.5, 0.8, 1.0]
    iterations = list(range(num_iterations))

    for proba in probabilities:
        result = genetic_algorithm(
            g,
            pool_size=200,
            selection_strategy="roulette",
            selection_percentage=0.6,
            crossing_proba=0.6,
            crossing_manner="1",
            mutation_proba=proba,
            nbr_iterations=50,
            param_tuning=True,
        )

        if len(result) < num_iterations:
            result.extend([result[-1]] * (num_iterations - len(result)))
        plt.plot(iterations, result, label=str(proba))

        # re-init the the graph
        g.re_initialize_graph()

    plt.ylabel("Nombre optimal des couleurs")
    plt.title("Effet du paramètre : MUTATION_PROBABILITY")
```

FIGURE 5.2 – Code pour tester l'impact de la MUTATION_PROBABILITY

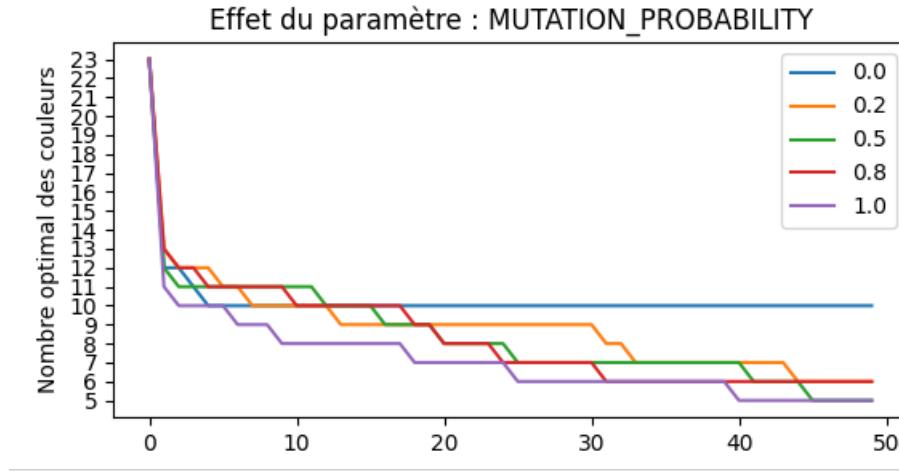


FIGURE 5.3 – Effet de la MUTATION_PROBABILITY

Effet de la POPULATION_SIZE :

```

def pool_size_impact(g):
    new_list = range(math.floor(8), math.ceil(g.num_vertices + 1))
    plt.yticks(new_list)

    nbr_iterations = 45
    iterations = [__ for __ in range(nbr_iterations)]

    pool_sizes = [size for size in range(100, 701, 300)]
    for pool_size in pool_sizes:
        result = genetic_algorithm(
            g,
            pool_size=pool_size,
            selection_strategy="roulette",
            selection_percentage=0.6,
            crossing_proba=0.6,
            crossing_manner="1",
            mutation_proba=0.5,
            nbr_iterations=nbr_iterations,
            param_tuning=True,
        )

        if len(result) < nbr_iterations:
            result.extend([result[-1]] * (nbr_iterations - len(result)))
        plt.plot(iterations, result, label=str(pool_size))

        # re-init the the graph
        g.re_initialize_graph()

    plt.ylabel("Nombre optimal des couleurs")
    plt.title("Effet du paramètre : POPULATION_SIZE")

```

FIGURE 5.4 – Code pour tester l'impact de la POPULATION_SIZE

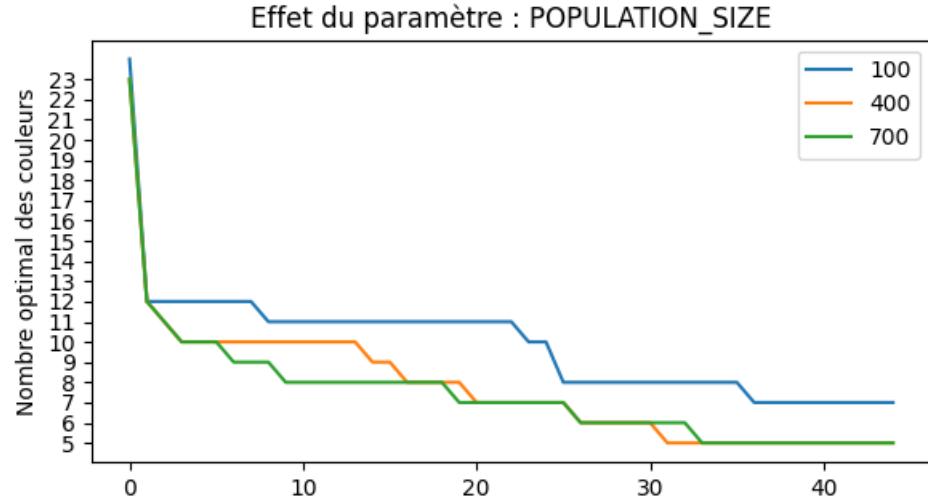


FIGURE 5.5 – Effet de la POPULATION_SIZE

Effet de la GENERATIONS_NUM :

```

def num_generations_impact(g):
    new_list = range(math.floor(0), math.ceil(g.num_vertices + 1))
    plt.yticks(new_list)

    num_generations = [_ for _ in range(20, 101, 20)]
    iterations = list(range(num_generations[-1]))

    for num in num_generations:
        result = genetic_algorithm(
            g,
            pool_size=200,
            selection_strategy="roulette",
            selection_percentage=0.6,
            crossing_proba=0.6,
            crossing_manner="1",
            mutation_proba=0.5,
            nbr_iterations=num,
            param_tuning=True,
        )

        if len(result) < num_generations[-1]:
            result.extend([result[-1]] * (num_generations[-1] - len(result)))
        plt.plot(iterations, result, label=str(num))

        # re-init the the graph
        g.re_initialize_graph()

    plt.ylabel("Nombre optimal des couleurs")
    plt.title("Effet du paramètre : GENERATIONS_NUM")

```

FIGURE 5.6 – Code pour tester l'impact de la GENERATIONS_NUM

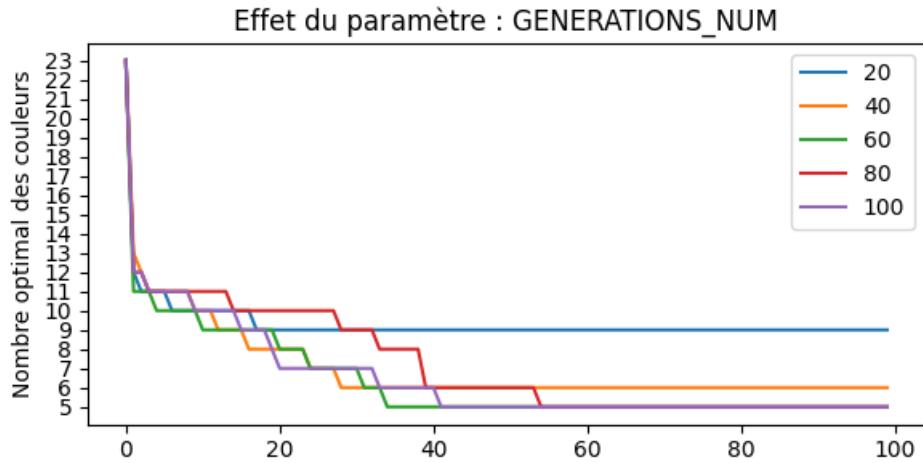


FIGURE 5.7 – Effet de la GENERATIONS_NUM

5.1.2 Effet de la taille de la liste Tabou :

Taille de la liste Tabu	Myciel3.col	Myciel5.col
5	0.0009546279907226562	0.007002353668212891
7	0.002960205078125	0.005002498626708984
8	0.003000020980834961	0.005930423736572266
9	0.0029587745666503906	0.004977226257324219
10	0.0039539337158203125	0.006000995635986328
11	0.0030014514923095703	0.005002260208129883
12	0.005959749221801758	0.008003950119018555
13	0.0029637813568115234	0.005965232849121094
14	0.003958702087402344	0.005001068115234375
15	0.0020110607147216797	0.007963895797729492
16	0.003995418548583984	0.007969379425048828
17	0.002001047134399414	0.007944107055664062
100	0.0019996166229248047	0.003998994827270508

FIGURE 5.8 – Effet de la taille de la liste sur des benchmarks

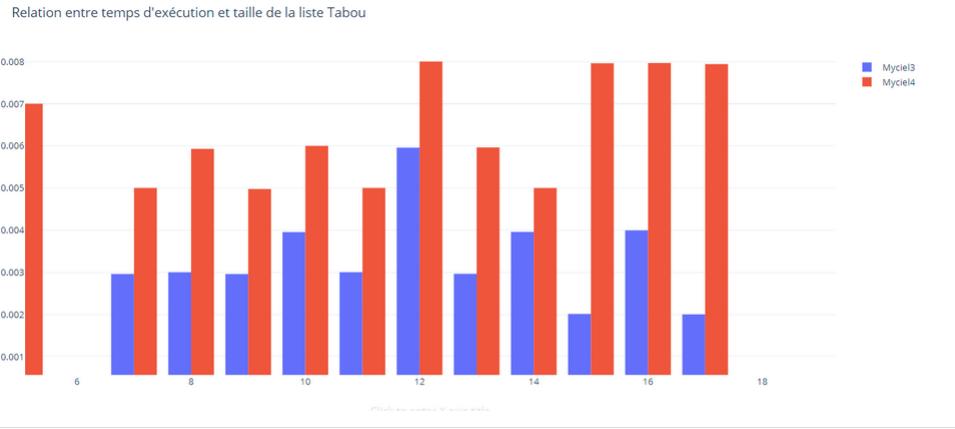


FIGURE 5.9 – Histogramme associé

5.2 Etude comparative

	Instance myciel4	Instance queen5_5	Instance queen7_7
Méthodes exactes			
Branch & Bound	5.0406s	0.0015s	0.7767s
Heuristiques			
Coloration gloutonne	0.00005s	0.00007s	0.0002s
DSatur	0.0003s	0.0007s	0.0031s
Max stable (avec coloration gloutonne)	0.0017s	0.0023s	0.0078s
Méta-heuristiques			
Recherche Tabou (RT)	0.0007s	0.0011s	0.0034s
Algorithme génétique (AG)	34.72391s	90.6011s	183.8560s

FIGURE 5.10 – Comparaison des temps d'exécution

	Instance myciel4	Instance queen5_5	Instance queen7_7
Méthodes exactes			
Branch & Bound	5 couleurs	5 couleurs	7 couleurs
Heuristiques			
Coloration gloutonne	5 couleurs	8 couleurs	10 couleurs
DSatur	5 couleurs	5 couleurs	11 couleurs
Max stable (avec coloration gloutonne)	5 couleurs	7 couleurs	10 couleurs
Méta-heuristiques			
Recherche Tabou (RT)	11 couleurs	14 couleurs	23 couleurs
Algorithmé génétique (AG)	5 couleurs	5 couleurs	10 couleurs

FIGURE 5.11 – Comparaison du nombre de couleurs trouvé

D'après les deux derniers tableaux, on peut tirer les conclusions suivantes :

- Pour l'instance myciel4, on remarque que les algorithmes Branch Bound, Coloration gloutonne, DSatur, Max stable et AG donnent tous le résultat optimale (5 couleurs), mais c'est l'algorithme Coloration gloutonne qui semble la meilleur avec le temps d'exécution le plus petit (0.00005 secondes).
- Pour l'instance queen5_5, on remarque que les algorithmes Branch Bound (bien sûr) et DSatur donnent le meilleur résultat possible (5 couleurs), mais c'est l'algorithme DSatur qui est le plus rapide (0.0007 secondes).
- Pour l'instance queen7_7, on remarque que seulement l'algorithme Branch Bound donne le résultat optimal (7 couleurs) à un temps d'exécution de presque 1 seconde (0.7767 secondes). Or pour un résultat moins optimisé mais assez bon (10 couleurs), on retrouve l'algorithme Coloration gloutonne avec 0.0002 secondes.

RÉFÉRENCES

- [1]Recherche Tabou
- [2]Algorithme génétique
- [3]Problème de coloration
- [4]The objective function for branch and bound
- [5]Source code for networkx.algorithms.coloring.greedy_coloring
- [6]Genetic algorithms for graph coloring