الجمـــهوريـــة الجـــزائريـــة الديـمقراطيـــــــة الشـــعبيـــــة
REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
وزارة التعـلـيـم الـعـالـــي و الـبـحـث الـعـلـمي
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE
جـامـعـة عمّـــار ثـلـيـــجـــي بالأغـــواط
UNIVERSITÉ AMAR TELIDJI LAGHOUAT

كـلـيـــــة الـعـلـــــوم
FACULTÉ DES SCIENCES

DEPARTEMENT DE MATHEMATIQUES ET INFORMATIQUE

Mémoire de MASTER

Domaine : *Mathématiques et Informatique*
Filière : *Informatique*
Option : *Réseaux, Systèmes et Applications Répartis*

Par :
OUBEDLI Abderrahmane

THÈME :

Coloriage de graphes

Soutenu publiquement devant le jury composé de :

Mme.	Attika CHETTIH	M.A (A), Université de Laghouat	Présidente
M.	Younes GUELLOUMA	M.A (A), Université de Laghouat	Examinateur
Mlle.	Amel BELABBACI	M.A (A), Université de Laghouat	Examinatrice
Mme	Hadda CHERROUN	Professeur, Université de Laghouat	Encadreur
M.	Attia NEHAR	M.A (A), Université de Djelfa	Co-encadreur

Année Universitaire : 2015-2016

Remerciements

Je remercie Dieu pour son immense générosité ;
Je remercie fortement mes encadreurs :
Madame Hadda Cherroun, professeur à l'université de Laghouat, pour son encadrement et son encouragement.
M. Attia Nehar, maître assistant à l'université de Djelfa, pour son aide précieuse .
Je remercie les membres du jury :
Mme Attika CHETTIH ;
M. Younes GUELLOUMA ;
Mlle. Amel BELABBACI ;
qui ont accepté de juger et d'évaluer ce travail.

Dédicaces

Je dédie ce modeste mémoire à :

Mes chers parents qui sans leurs prières, je n'aurais jamais achevé mon travail ;

mes frères et soeurs ;

mes encadreurs et enseignants ;
tous ceux qui me connaissent.

ملخص

يتطرق هذا العمل إلى مسائل تلوين البيانات، والتي تعد من بين المسائل كاملة التعقيد حيث تمت مناقشة إحدى الخوارزميات التي تعتمد على طريقة التقسيم والتقييم. في مرحلة التقسيم يتم دمج او ربط العقد أما في مرحلة التقييم فتستعمل دالة تيتا (ϑ)كمعيار لضبط عملية التفرع. كما تضمن العمل في المرحلة التجريبية مجموعة من البيانات المستمدة من مشاكل واقعية وأخرى عشوائية، لمعرفة مدى كفاءة الخوارزمية، حيث بينت النتائج فعالية هذه الطريقة بالنسبة للبيانات صغيرة ومتوسطة الحجم والعكس بالنسبة للبيانات كبيرة الحجم قليلة الكثافة، ولمعالجة هذه المشكلة تم تقديم اقتراح نسخة متوازية.

Abstract

This work addresses graph coloring problem, which is ranked among the NP-hard problems. One of the algorithms that is based on the branch and bound method is discussed and evaluated. At the branching stage two vertices are merged or linked by new edge, and at the bounding stage a theta function (ϑ) is used to adjust the branching process. For experiments, we use a set of benchmarks from real and synthetic graphs, to mesure the efficiency of the algorithm. The results prove the suitability of our approach for small and medium graphs. However, for sparse large graphs, the approach behaves worst. To cope with problem, we have proposed a parallel version.

Résumé

Ce travail traite le problème de coloration de graphes, qui se classe parmi les problèmes NP-difficiles. Un des algorithmes qui repose sur la méta-méthode séparation et évaluation est discuté. À l'étape de séparation deux sommets sont fusionnés ou lié par une nouvelle arête, et à l'étape de l'évaluation, la fonction thêta (ϑ) est utilisée pour guider le processus de branchement. Le travail comprend une partie expérimentale pour évaluer l'efficacité de l'algorithme sur un jeu de test réels et synthétiques. Les résultats prouvent la pertinence de notre approche pour les graphes de petites et moyennes tailles. Toutefois, pour les graphes éparses de grande taille, l'approche donne des mauvais résultats. Pour faire face à problème, nous avons proposé une version parallèle.

Table des matières

Table des figures

Liste des tableaux

Liste des Algorithmes

Introduction

Les graphes servent comme modèles mathématiques pour analyser de nombreux problèmes concrets du monde réel. Certains problèmes en physique, chimie, sciences de la communication, la technologie informatique, la génétique, la psychologie, la sociologie et la linguistique peuvent être formulées comme des problèmes dans la théorie des graphes. En outre, de nombreuses branches des mathématiques, comme la théorie des groupes, théorie des matrices, la probabilité et la topologie, ont des liens étroits avec la théorie des graphes.

Plusieurs problèmes pratiques ont joué un rôle déterminant dans le développement de divers sujets dans la théorie des graphes. Le fameux problème des ponts de *Königsberg* a été l'inspiration pour le développement de la théorie des graphes Euleriens. La théorie des graphes hamiltoniens a été développé à partir du jeu "*Around the World*" de *William Hamilton*. La théorie des graphes acycliques a été développé pour résoudre les problèmes des réseaux électriques, et l'étude des ≪arbres≫ a été développé pour énumérer les isomères de composés organiques. Le problème bien connu de quatre couleurs forme la base même pour le développement de planéité dans la théorie des graphes et de la topologie combinatoire. Les problèmes de programmation linéaire et la recherche opérationnelle peuvent être traités par la théorie des flux dans les réseaux. Le problème d'*écolière de Kirkman* et d'ordonnancement sont des exemples de problèmes qui peuvent être résolus par le coloriage de graphes [BR12].

Contexte

Malgré la diversité des modèles de coloriage de graphes discutés dans la littérature fondamentale, le modèle classique (i.e coloriage de sommets) reste l'un des plus importants et largement appliqués dans la pratique. La $\mathcal{NP}-$difficulté du problème de coloriage donne lieu à la nécessité d'employer des méthodes non optimales (approximatives) dans une large gamme d'applications pratiques.

De plus, la large gamme de problèmes résolus par la coloration classique, ainsi que la variété des familles de graphes avec signification pratique dans ce domaine facilite l'évolution et le développement de nouveaux algorithmes sous-optimaux. Il existe plusieurs méthodes relativement simples. Cependant, nous pouvons s'empasser de solutions exactes pour certaines applications.

Objectifs

Dans ce contexte, notre travail a pour but de traiter le problème de coloriage des sommets, en passant par la présentation de quelques méthodes exactes et d'autres approximatives avant de discuter notre approche qui fait partie de la première catégorie.
Les objectifs principaux de ce travail sont :

- Implémenter une version séquentielle améliorée d'un algorithme exacte de coloriage basé sur la méthode de *"Séparation et Évaluation"* [CF07] et la fonction ϑ, considérée comme un caractéristique d'un graphe.
- Mesurer sa performance empirique par un jeu de test de graphes réels et synthétiques.
- Paralléliser l'algorithme afin de l'étendre traiter des problèmes de grande taille.

Organisation du mémoire

En plus de cette introduction, notre mémoire est scindé en quatre (4) chapitres :

Dans un **premier chapitre** on a introduit les notions de base concernant les graphes et qui seront utilisées plus tard dans les chapitres qui suivent. Par la suite, le problème de coloriage est présenté et expliqué d'un point de vue historique et formel en passant par ses applications.

Le **deuxième chapitre** traite les grandes approches où la plus part des algorithmes de coloriage sont classés, à savoir, une classe des méthodes exactes (optimales) et une autre pour les méthodes approximatives (sous-optimales).

En suite, la description de notre algorithme basé sur la méta-méthode *Branch & Bound* sera présenté dans le **troisième chapitre** avec l'utilisation du nombre thêta considéré comme une bonne borne du nombre chromatique d'un graphe, avec une partie expérimentale, utilisant un jeu de test réels ainsi que des graphes aléatoires.

Dans le **quatrième chapitre**, une proposition de parallélisation de notre approche est décrite.

Chapitre 1

Généralités sur les graphes et le coloriage

Introduction

Dans ce chapitre, nous abordons les notions de base sur les graphes en général, et tout particulièrement au problème de coloriage. On commence, dans la section 1.1, par donner les définitions nécessaires à la compréhension de la suite de ce mémoire, puis nous nous concentrons, aux sections suivantes, sur la formalisation du problème de coloriage, ses différents types et des exemples d'applications.

1.1 Définitions

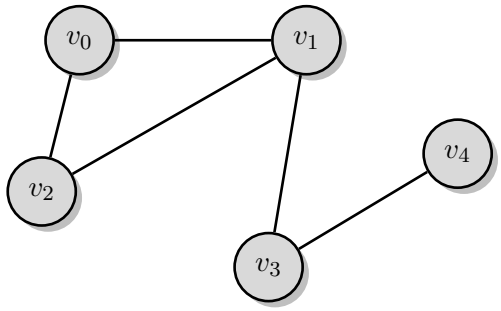
Définition 1.1.1. Un graphe G est une paire ordonnée $G = (V, E)$, où V est un ensemble fini d'éléments appelés sommets ou nœuds, tandis que E est un ensemble fini de paires non-ordonnées de sommets appelés arêtes.

La cardinalité de l'ensemble des sommets V, désigné par le symbole $n = |V|$ est appelée l'ordre du graphe G. De même, la cardinalité d'ensemble des arêtes E, noté $m = |E|$ est appelée la taille du graphe G.

Les sommets $u, v \in V$sont dits adjacents (ou voisins) si $\{u, v\} \in E$. Ils sont dits non-adjacents si $\{u, v\} \notin V$. Les arêtes $e = \{v_i, vj\}, f = \{v_j, v_k\}$ sont dits adjacents en v_j si $e \cap f = \{v_j\}$ et non-adjacents si $e \cap f \neq \emptyset$ [Kub04] .

Si les arêtes sont orientées (arcs) on parlera alors de graphe orienté, et dans le cas où il existe plusieurs arêtes entre deux sommets u,v le graphe est dit multi-graphe.

Dans ce travail, on s'intéresse uniquement aux graphes simples non orientés (la figure 1.1).



$$V = \{v_0, v_1, v_2, v_3, v_4\}.$$
$$E = \{\{v_0, v_1\}, \{v_0, v_2\}, \{v_1, v_2\}, \{v_1, v_3\}, \{v_3, v_4\}\}.$$

FIGURE 1.1 – Exemple d'un graphe simple avec $n = 5$.

Définition 1.1.2. Le degré $deg(v)$ du sommet v dans le graphe G représente le nombre des arêtes incidentes au sommet v dans le graphe G. Il est donné par $|\{e \in E : v \in e\}|$.

Définition 1.1.3. Le degré maximal d'un sommet dans le graphe G est désigné par $\Delta(G) = max\{deg(v)|v \in V\}$, alors que le degré minimal est désigné par $\delta(G) = min\{deg(v)|v \in V\}$.

Définition 1.1.4. La densité d'un graphe G, noté $D(G)$ est définie par le rapport entre le nombre d'arêtes existantes et le nombre d'arêtes possibles $D(G) = 2m/(n(n-1))$.

Définition 1.1.5. Soient $G = (V, E)$ et $G' = (V', E')$ deux graphes. Le graphe G' est un sous-graphe de G si :

$$\begin{cases} V' \subseteq V \\ E' \subseteq E \cap (V' \times V') \end{cases} \tag{1.1}$$

Définition 1.1.6. Un graphe est dit complet si toutes ses paires de sommets sont adjacentes. Un graphe complet avec n sommets est noté K_n.

Définition 1.1.7. Le graphe complémentaire ou graphe inversé d'un graphe simple G est un graphe simple H ayant les mêmes sommets et tel que deux sommets distincts de H soient adjacents si et seulement s'ils ne sont pas adjacents dans G.

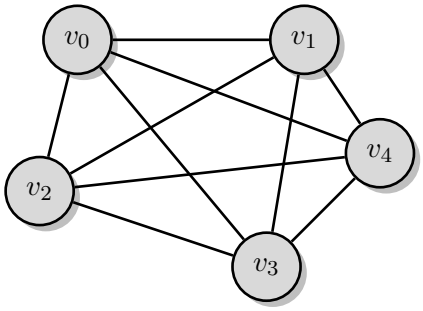


FIGURE 1.2 – Exemple d'un graphe complet avec $n = 5$.

Définition 1.1.8. Une clique est un sous-graphe complet de G. La clique maximale est la plus grande clique d'un graphe.

Définition 1.1.9. Un stable est un ensemble de sommets qui sont non-adjacents mutuellement. Le stable maximal est le plus grand stable d'un graphe.

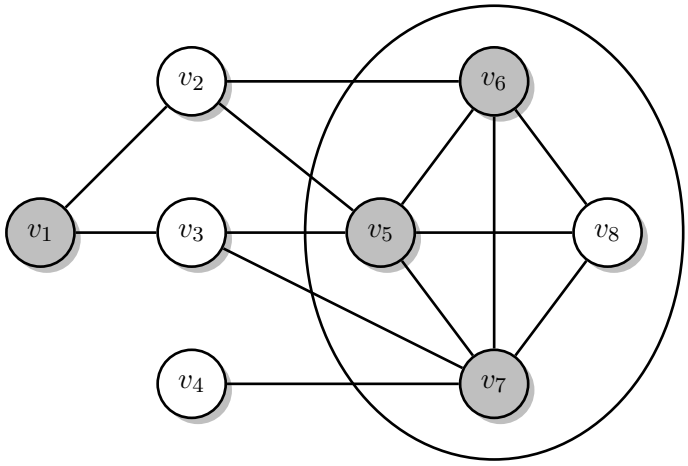


FIGURE 1.3 – Exemple d'une clique maximale (dans l'ellipse) est un stable maximal (les sommets blancs).

1.2 Le coloriage et ses types

1.2.1 Historique

Le problème de coloriage de graphe a été noté la première fois en 1852 par un étudiant de l'University College London, *Francis Guthrie* (1831-1899), qui, tout en colorant une carte des comtés de l'Angleterre, a remarqué que seulement quatre couleurs étaient nécessaires pour veiller à ce que tous les pays voisins ont été alloués à des couleurs différentes.

Afin de simplifier l'idée de coloriage de graphe on peut imaginer un ensemble d'objets qui doivent être regrouper à un ensemble finie de groupe : un coloriage d'un graphe G est une partition de ces sommets à k sous-ensembles $S = \{S_1, S_2, S_3, S_4, ..., S_k\}$.

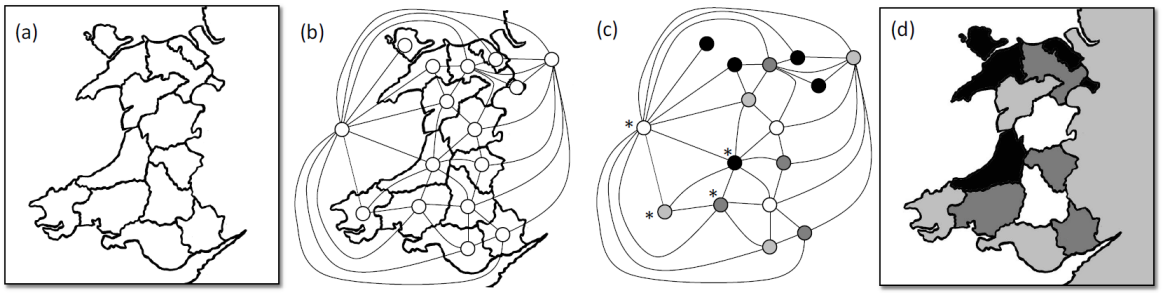


FIGURE 1.4 – Utilisation de graphes pour le coloriage d'une carte [Lew15].

1.2.2 Types de coloriage

En général, le coloriage d'un graphe peut consister en affectant des couleurs à des sommets, des arêtes et des faces d'un graphe planaire ou toute combinaison des dispositions qui précèdent. En outre, pour chaque modèle de coloriage, on a diverses règles sur la légalité ou l'optimalité des solutions. les modèles non classiques peuvent introduire des conditions supplémentaires de l'utilisation de couleurs en permettant d'utiliser plus d'une couleur par élément, ce qui permet la séparation des couleurs en fractions ou en admettant l'andainage de couleurs. D'une manière générale, il existe plusieurs modèles de coloriage de graphes décrits dans la littérature[Kub04].

Coloriage des sommets : ce sont problèmes de coloriages qui ont reçu le plus d'attention et qui ont été étudiés le plus souvent sont ceux visés à colorier les sommets. i.e affecter des couleurs aux sommets.

Coloriage des arêtes : Un coloriage des arêtes d'un graphe G est une affectation de couleurs aux arêtes de G.

Coloriage des surfaces (régions) : Un graphe G est $k-$*région colorable* si chaque région de G peut être attribué à l'un des k couleurs données d'une sorte que les régions voisines (adjacentes) sont colorées différemment.

Coloriage par liste : Le problème de coloriage par liste, est comme le problème de coloriage des sommets, implique l'assignation des couleurs à chaque sommet d'un graphe de telle sorte qu'aucune paire de sommets adjacents ne sont pas affectés à la même couleur. Cependant, en plus de cela, quand un problème est spécifié, des sommets individuels sont également alloués à leur propre liste de couleurs admissibles auxquelles ils peuvent être affectés.

Multi-coloriage : Dans ce cas, la tâche est d'attribuer ensuite des couleurs différentes à chaque sommet v tel que les sommets adjacents n'ont pas de couleurs en commun, et le nombre de couleurs utilisées est minime. le multi-coloriage a des applications pratiques dans des domaines tels que les problèmes d'affectation de fréquences où, dans certains cas, il est souhaitable que les dispositifs radio soient capables de transmettre et de recevoir des messages sur plusieurs fréquences.

1.3 Formalisme et concepts

Etant donné un graphe $G = (V, E)$ et un entier positif k, un $k-$*coloriage* est une fonction $c : V \rightarrow \{1, ..., k\}$ de l'ensemble des sommets dans l'ensemble des nombres entiers positifs inférieurs ou égaux à k. Si on pense à ce dernier ensemble comme un ensemble de c 'couleurs', alors K est une affectation d'une couleur à chaque sommet.

Coloriage propre et coloriage impropre La fonction c est un $k-coloriage\ propre$ de G si pour chaque paire u, v des sommets adjacents, $c(u) \neq c(v)$, i.e les sommets adjacents sont colorés différemment. Si un tel coloriage existe pour un graphe G, on dit que G est $k-colorable$. Dans le cas contraire on parle de coloriage *impropre*.

Le problème de k-coloriage propre se formule comme suit :

Pour chaque sommet $v \in V$ on associe un nombre entier $c(v) \in \{1, 2, ..., k\}$ tel que :

— $c(v) \neq c(u)\ \forall\ \{v, u\} \in E$; et.

— k est minimal.

Pour chaque sous-ensemble V_i de sommets $(1 \leq i \leq k)$ dont la couleur est i, V_i est une $Classe\ de\ couleurs$.

Définition 1.3.1. Un coloriage est dit *complet* si tout sommet $v \in V$ est associé à une couleur $c(v) \in \{1, 2, ..., K\}$.

Définition 1.3.2. Un coloriage est dit $faisable$ s'il est complet et propre.

1.3.1 Nombres et bornes

Pour n'import quel graphe G, plusieurs nombres significatifs sont liés, et dans les cas où il est difficiles de les trouver, des bornes supérieures ou inférieures sont en générale utilisées pour les encadrer.

Nombre de clique le nombre de clique est la taille de la clique maximale noté $\omega(G)$.

Nombre de stable le nombre de stable est la taille du stable maximal noté $\alpha(G)$.

Nombre chromatique Le *nombre chromatique* noté $\chi(G)$ d'un graphe G est le nombre minimal de couleurs pour faire un coloriage propre des sommets de G.

Indice chromatique L'*indice chromatique* noté $\chi'(G)$ d'un graphe G est le nombre minimal de couleurs pour faire un coloriage propre des arrêtes de G.

1.3.2 Encadrement du nombre chromatique

Une idée simple pour estimer le nombre chromatique est de faire un encadrement dont le plus intuitif est : $1 \leq \chi(G) \leq n$. cet encadrement est induit du fait que pour n'importe quel graphe G on a besoin au moins d'une seule couleurs si tous les sommets sont isolés, ou dans le pire de cas n couleurs s'ils sont tous connectés. Cependant cet encadrement est très étroit. et pour cela on examine maintenant quelques-unes des bornes supérieures et inférieures qui peuvent être déclarés sur le nombre chromatique d'un graphe. Ces bornes peuvent être tout à fait utile, et dans de nombreux cas, elles sont trop difficile à calculer.

Bornes inférieures

Si un graphe G contient un sous-graphe complet k_k, un coloriage possible de G exigera évidemment au moins k couleurs. Déclarant ceci d'une autre manière, soit $\omega(G)$ le nombre de sommets contenus dans la plus grande clique dans G. Comme $\omega(G)$ couleurs différentes seront nécessaires pour colorer cette clique, on en déduit que $\chi(G) \geq \omega(G)$.

Dans une autre perspective, on peut aussi considérer le stable d'un graphe. Soit $\alpha(G)$ le nombre de stable d'un graphe G. Dans ce cas, $\chi(G)$ doit être au moins $\lceil n/\alpha(G) \rceil$ puisque pour être inférieure à cette valeur impliquerait l'existence d'un stable plus grand que $\alpha(G)$.
Ces deux bornes peuvent être combinés dans ce qui suit :

$$\chi(G) \geq \max\{\omega(G), \lceil n/\alpha(G) \rceil\} \tag{1.2}$$

La précision des bornes indiquées dans l'inégalité (1.2) varie d'un cas à l'autre base. Leur inconvénient majeur est le fait que les tâches de calcul $\omega(G)$ et $\alpha(G)$ sont eux-mêmes des problèmes $\mathcal{NP}$-*difficiles*.

Bornes supérieures

Les bornes supérieures sur le nombre chromatique sont souvent calculées en tenant compte des degrés de sommets dans un graphe. Par exemple, quand un graphe a une densité élevée, souvent un plus grand nombre de couleurs seront nécessaires, car une plus grande proportion des paires de sommets doivent être séparé en différentes classes de couleur. Cette proposition donne lieu au théorème suivant [Lew15].

Theorem 1.3.1. *Soit G un graphe connexe avec le degré maximal $\Delta(G)$ (qui est, $\Delta(G) = max\{deg(v) : v \in V\}$). Alors $\chi(G) \leq \Delta(G) + 1$.*

1.3.3 Complexité du problème de coloriage

Problématique : Est-ce-il y a un algorithme qui trouve le nombre chromatique pour tout graphe quelque soit sa taille et sa topologie ?

La résolution de ce problème réside dans l'énumération de toutes les possibilité de coloriage et la détermination de celle-ci qui donne le nombre minimum de couleurs ce qu'il fait partie de l'optimisation combinatoire.

L'optimisation combinatoire

L'optimisation combinatoire consiste à trouver un point minimisant une fonction, appelée coût, dans un ensemble dénombrable. Une méthode naïve pour résoudre ce problème, est d'énumérer toutes les solutions du problèmes, de calculer le coût pour chacune, puis de donner le minimum.

Parfois il est possible d'éviter d'énumérer des solutions dont on sait, par l'analyse des propriétés du problème, que ce sont de mauvaises solutions, c'est-à-dire des solutions qui ne peuvent

pas être le minimum. La méthode séparation et évaluation est une méthode générale pour cela.

Un problème d'optimisation combinatoire est formulé de manière générale sous la forme suivante :

— Un ensemble discret N.

— Une fonction d'ensemble $f : 2^N \longrightarrow \mathbb{R}$ dite fonction objectif.

— Un ensemble $\mathcal{R}$ de sous-ensembles de N , dont les éléments sont appelés les solutions réalisables.

Un problème d'optimisation combinatoire consiste à déterminer $\max_{S \subseteq N} \{f(S) : S \in \mathcal{R}\}$.

Espace des solutions et taux d'accroissement

Une méthode possible pour résoudre le problème de coloriage de graphes est de vérifier chaque affectation possible des sommets aux couleurs et revenir ensuite le meilleur d'entre eux. Une telle méthode serait en effet garanti pour retourner une solution optimale, mais serait-il pratique ?

Selon cette approche, étant donné un graphe particulier à n sommets, on doit d'abord déterminer le nombre maximal de couleurs que la solution pourrait utiliser. Dans la pratique, cela pourrait être estimée comme une valeur comprise entre 1 et n, mais pour l'instant on va supposer que cette valeur soit simplement n, car aucune solution réalisable ne sera jamais besoin de plus couleurs que les sommets.

On considère maintenant le nombre d'affectations qui auraient besoin d'être vérifié. Puisqu'il y aurait n choix de couleur pour chacun des n sommets, cela donnerait un espace de solution contenant un total de n^n solutions candidates. Ce nombre augmente très rapidement à l'égard de n, ce qui signifie que le nombre de solutions candidats à vérifier deviendra rapidement trop grand même pour un ordinateur très puissant.

En outre, même si on limitera les solutions en utilisant un maximum de $k < n$ couleurs étiquetés $1, ..., k$, ce serait encore conduire à k^n assignations possibles, une fonction qui est encore soumis à une explosion combinatoire similaire pour tout $k > 1$.

En plus, dans certains cas des solutions sont considérées identiques lorsque l'on a des permutation des étiquettes des couleurs seulement. par exemple la solution :

$$c(v1) = 4 \ , \ c(v2) = 3 \ , \ c(v3) = 4 \ , \ c(v4) = 1 \ , \ c(v5) = 1 \ , \ c(v6) = 2$$

est identique à :

$$c(v1) = 2 \ , \ c(v2) = 3 \ , \ c(v3) = 2 \ , \ c(v4) = 1 \ , \ c(v5) = 1 \ , \ c(v6) = 4$$

Si on échange les couleurs 2 et 4.
Le nombre des assignations identiques à ignorer en utilisant k couleurs est $k! = k \times (k-1) \times (k-2) \times (k-3) \times ... \times 2 \times 1$. ce qui représente une quantité importante. Pour cela une autre

approche peut décrire le problème de coloriage d'une manière plus précise par le partitionnement de l'ensemble des sommets en k classe de couleurs $S = \{S_1, S_2, ..., S_k\}$ de tel façon que chaque $S_i \in S$ est un stable, et k est minimal.

Le nombre de possibilités dans ce cas est réduit considérablement par rapport à n^n. En général, Le nombre de façons de partitionner un ensemble de n éléments en sous-ensembles non vides est donnée par le $n^{ème}$ nombre de **Bell**, notée B_n.

Si on limite le nombre de couleurs à $k < n$ alors on aura le nombre de **Stirling** [1] suivant :

$$\left\{ {n \atop k} \right\} = \frac{1}{k!} \sum_{j=1}^{k} (-1)^{k-j} \binom{k}{j} j^n. \tag{1.3}$$

A la fin, si on estime B_n comme la somme de toutes les possibilité de k on aura :

$$B_n = \sum_{k=1}^{n} \left\{ {n \atop k} \right\} \tag{1.4}$$

Les discussions ci-dessus ont démontré que tout algorithme de coloriage de graphe basé sur l'énumération et la vérification de l'espace de solution entière n'est pas raisonnable parce que, quoi que ce soit, sauf cas de problème trivial, son exécution sera tout simplement trop long.

Toutefois, le taux de croissance exponentielle de l'espace de solution n'est pas la seule raison pour laquelle le problème de coloriage de graphe est $\mathcal{NP}$-*difficile* (concept introduit par *Stephen Cook* dans les années 1970s).

1.4 Applications de coloriage

1.4.1 Emploi du temps

Cette tâche consiste à attribuer les événements à des intervalles de temps en fonction d'un ensemble de contraintes. L'un des plus importants de ces contraintes est ce qui est souvent connu sous le nom de la contrainte "choc d'évènement".

Un choc d'évènement indique que si une personne (ou une autre ressource dont il n'y a qu'une seule) est nécessaire pour être présente dans une paire d'événements, ces événements ne doivent pas être affectés au même intervalle de temps depuis une telle cession se traduira par cette personne (ressource) ayant d'être à deux endroits à la fois.

Les problèmes de l'emploi du temps peuvent être facilement convertis en un problème de coloriage de graphe équivalent en considérant chaque événement comme un sommet, puis en ajoutant des arêtes entre des paires de sommets qui sont soumises à une contrainte de choc d'événement.

1. Il tire son nom de *Stirling James*. Les nombres de Stirling sont introduits au XVIII^e siècle.

Chaque intervalle de temps disponible dans le calendrier correspond alors à une couleur, et la tâche est de trouver un coloriage tel que le nombre de couleurs n'est pas plus grand que le nombre de créneaux horaires disponibles.

1.4.2 Allocation des registres

Le compilateur favorise toujours de mettre les variables dans les registres du processeurs afin d'augmenter la performance des programmes. Les registres généralement sont limités en nombre (dizaines par processeur), par contre le nombre de variables est important. Cette situation impose que le compilateur décide quelles variables à choisir, en évitant tout type de conflits entre celles-ci (dépendance, anti-dépendance ...) c.a.d : deux variables qui interfèrent ne doivent pas être allouer au même registre.

Le problème de décider comment affecter les variables aux registres peuvent être modélisés comme un problème de coloriage de graphe en utilisant un sommet pour chaque intervalle de temps, puis en ajoutant des bords entre des paires de sommets correspondant au chevauchement des intervalles. Un tel graphe est connu comme le graphe d'interférence, et la tâche est de colorer le graphe en utilisant des couleurs égales ou moins au nombre de registres disponibles.

1.4.3 Assignation des fréquences

Les bandes des fréquences est une ressource rare dans le domaine de communication, et pour maximiser l'utilisation de ces bandes, elles sont divisée en certain nombre de sous-bandes ou canaux. Pour arriver à cette optimisation les canaux peuvent être utilisés plusieurs fois, si les émetteurs sont éloignés suffisamment l'un de l'autre de telle sorte que la distance rend l'interférence minimale.

1.4.4 Ordonnancement

Plusieurs exemples pratiques d'ordonnancement par un coloriage de graphes ont été abordé, on peut citer par exemple : l'ordonnancement des vols d'avions, l'ordonnancement des tâches bi-processeurs. Soit n le nombre de tâches à ordonnancer. G est le graphe de conflit de ces tâches : les sommets correspondent aux tâches, et deux sommets sont reliés par une arête si les tâches correspondantes ne peuvent pas être exécutées en même temps (par exemple, elles utilisent la même ressource non partageable). Les couleurs correspondent aux plages horaires disponibles.

1.4.5 Test des circuits imprimés

Garey, et al (1976) [GJS76], ont examiné le problème de test des circuits imprimés sur cartes pour les courts-circuits (causés par des lignes de soudure). Cela donne lieu à un problème de coloriage de graphe, dont les sommets correspondent à des fils, et une arête entre deux sommets s'il existe un risque potentiel pour un court-circuit entre les fils. Un coloriage du graphe correspond à un partitionnement des fils en ensembles, de telle sorte que les fils dans

chaque ensemble peuvent être testés en même temps contre les courts-circuits, ce qui accélère le processus de test.

1.4.6 Compression des images

Considérons une image bitmap partitionnée en régions connectées, par un algorithme de segmentation, et que le but est juste de pouvoir distinguer, pour chaque pixel, la région à laquelle il appartient. Un code mot peut être assigné à chaque pixel de tel sorte que les régions seront constituées de pixels ayant le même code mot. Il est clair qu'on aura une borne supérieure égale à 2 sur le nombre de bits nécessaires par pixel, car le coloriage d'un graphe planaire se fait en utilisant juste 4 couleurs [CFA04].

1.4.7 Problème du pécheur

Un autre problème connu est le problème de pécheur qui possède un ensemble de type de poisson, mais ces poisson ne sont pas tous compatible à cause de la loi proie-prédateur, donc pour connaitre le nombre minimal des compartiments, on modélise avec le coloriage, on obtient : Les nœuds sont les différents types de poisson, et deux extrémité d'une arête représentent de types incompatible, Le nombre de couleurs est donc le nombre de compartiments nécessaires [Neh09].

1.4.8 Carrés latins

Un autre domaine important des mathématiques pour lesquelles les techniques de coloriage de graphes sont naturellement adaptées est le domaine des carrés latins (Sudoku Puzzles). les carrés latins sont $l \times l$ grilles qui sont remplisse avec l symboles différents, chacun se produisant exactement une fois par ligne et une fois par colonne. Ils ont été initialement examinées en détail par *Leonhard Euler*, qui a rempli ces grilles avec des symboles de l'alphabet latin, bien que de nos jours, il est courant d'utiliser les entiers 1 jusqu'à l pour remplir les grilles.

Un carré latin peut être exprimée comme un problème de coloriage de graphe. Les symboles utilisés dans la grille représentent les couleurs. Chaque cellule de la grille est alors associée à un sommet et des arêtes ont été ajoutées entre toutes les paires de sommets dans la même ligne, et toutes les paires de sommets dans la même colonne.

Les carrés latins ont des applications pratiques dans divers domaines, comme la planification et la conception expérimentale. Pour une application dans la planification, On a deux groupes de l personnes et on veut organiser des réunions entre toutes les paires de personnes appartenant à ces différents groupes. Il est clair que dans ce cas, on a besoin de l^2 réunions au total et, étant donné que seuls l réunions peuvent avoir lieu simultanément, au moins l intervalles de temps (*timeslots*) seront nécessaires. les carrés latins donnent des solutions à ces problèmes qui font usage d'exactement l intervalles de temps. Pour voir cela, on nomme les membres de la première équipe comme $r_1, r_2, ..., r_l$, qui sont représentés par les lignes de la grille, et les membres de

deuxième équipe $c_1, c_2, ..., c_l$, représentée par les colonnes. Les caractères dans un $l \times l$ carré latin représentent alors les différents intervalles de temps auxquels les réunions sont assignées.

Conclusion

Dans ce chapitre on a exposé certaines généralités sur les graphes et leur coloriage. Ces concepts sont nécessaires pour la compréhension du reste du document.

Chapitre 2

Etat de l'art

Introduction

Dans la littérature, différents travaux traitent le problème de coloriage, des méthodes proposées pour arriver à des solutions optimales, on les a classé comme algorithmes exacts, il y a aussi d'autres méthodes qui cherchent des solutions approchées (sous-optimales) dans un temps raisonnablement réduit par rapport aux précédentes. on les a classé sous le nom d'algorithmes approximatifs.

2.1 Algorithmes exacts

Parmi les méthodes qui ont été proposé cherchant à trouver une solutions optimale pour le problème des coloriage on a choisi de décrire trois approches : Algorithme de Randall-Brown basé sur l'idée de la solution partielle, la technique de backtracking et la programmation en nombres entiers.

2.1.1 Algorithme de Randall-Brown

L'un des premiers algorithmes séquentiels de coloriage a été introduit par Randall-Brown [Bro72]. Avant de décrire cet algorithme, on introduit d'abord la notion de solution partielle. Soit $G = (V, E)$ un graphe simple, où V est l'ensemble des sommets, E est l'ensemble des arêtes :

Définition 2.1.1. Une solution partielle de niveau p, $(p < n)$ est un coloriage du graphe G dans lequel seuls les sommets $v_1, v_2, ..., v_p$ sont coloriés.

Description : Soit K_{pq} une solution partielle de niveau p, avec q représente le nombre de couleurs utilisées. A partir de K_{pq} on obtiendra toutes les solutions en procédant comme suit :

— Introduire un nouvel sommet v_{p+1}.

— Assigner à v_{p+1} les couleurs $1, 2, ..., q + 1$ successivement.

— Éliminer les colorations non faisables.

— Procéder de la même manière avec toutes les nouvelles solutions partielles.

A la fin, les solutions qui utilisent le plus petit nombre de couleurs sont optimales.

Algorithme 1 : Algorithme séquentiel de coloriage (SC) d'un graphe.

```
Données : G(V, E)
Résultat : χ(G)
début
    C ⟵ ChercherClique(G)
    k ⟵ 0
    pour v ∈ C faire
        k ⟵ k + 1
        Colorier v avec k
    retourner SCrec(G, k, |V| + 1, |C|)
/* ------------------------------------------------------------------ */
Fonction SCrec(G, k, best, lb)
début
    si G est entièrement colorié alors
        retourner k
    sinon
        v ⟵ ChercherSommetNonColorié(G)
        pour c ← 1 à min(k + 1, best − 1) faire
            si ∀ v′ ∈ Voisins(v); c(v′) ≠ c alors
                colorier v avec c ;
                best ⟵ SCrec(G, max(c, k), best, lb)
                décolorer v ;
                si lb = best alors
                    retourner best
        retourner best
```

Complexité Si on considère que l'opération caractéristique de cet algorithme est l'assignation d'une couleur à un sommet, on peut considérer sa complexité au pire cas comme suit : Au début on a une coloration de p sommets avec q couleurs, donc au pire cas $q = p$, et donc on $\sum_1^p k$ opérations caractéristiques.
Au niveau du sommet v_{p+1}, les couleurs possibles sont $1, 2, ..., q + 1$, donc on aura $(p! * p + 1)$ opérations caractéristiques, ce qui donne $\sum_1^{p+1} k$ pour le coloriage des $p + 1$ sommets.
Pour un graphe à n sommets, le nombre d'assignation de couleur est au pire cas égale à $\sum_1^n k$.
On peut conclure que cet algorithme est d'une complexité factorielle $\mathcal{O}(n!)$. L'algorithme 1 [Cou97], est une implémentation de cet méthode.

2.1.2 Le retour sur trace (Backtracking)

Backtracking est une méthode générale qui peut être utilisée pour déterminer une solution (ou éventuellement l'ensemble des solutions optimales) à un problème de calcul tels que la coloration de graphe. En substance, les algorithmes de traçage fonctionnent par la construction de l'espace des solutions complètes a partir des solutions partielles d'une manière systématique. Cependant, au cours de ce processus de construction dès que la preuve est acquise qu'il n'y a aucun moyen de compléter la solution partielle en cours pour obtenir une solution optimale, l'algorithme *backtracks* afin d'essayer de trouver des moyens appropriés de réglage de la solution partielle actuelle.

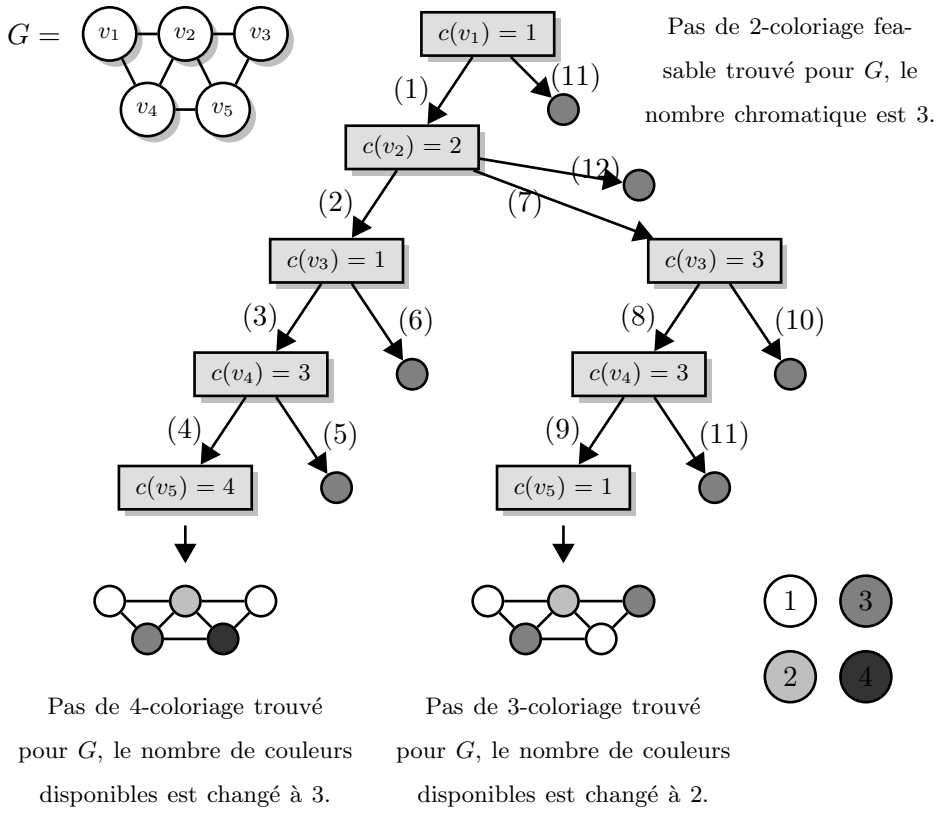


FIGURE 2.1 – Exemple d'exécution d'un algorithme de backtracking.

Un algorithme de backtracking simple pour le coloriage de graphe pourrait fonctionner comme suit. Etant donné un graphe $G = (V, E)$, les sommets sont d'abord triés de telle sorte que le sommet $v_i (1 \leq i \leq n)$ correspond au $i^{ième}$ sommet dans cet ordre. On a également besoin de choisir une valeur k désignant le nombre de couleurs disponibles. Initialement, ce pourrait être tout simplement $k = \infty$. L'algorithme de backtracking effectue alors une série d'étapes *avant* et *arrière*.

Pas en avant : colorier les sommets dans l'ordre donné jusqu'à ce qu'un sommet est identifié qui ne peut être colorée avec une des k couleurs disponibles.

Pas en arrière : sinon remontent à travers les sommets colorés dans l'ordre inverse et iden-

tifier les points où les différentes affectations de couleurs aux sommets peuvent être faites. les pas en avant sont ensuite repris à partir de ces points. Si une coloration complète possible est trouvée, alors k peut être réglé sur le nombre de couleurs utilisées dans cette coloration moins 1, l'algorithme continue alors. A la fin, l'algorithme se termine quand un pas en arrière atteint le sommet v_1 racine, ou quand un autre critère d'arrêt comme un délai maximal est atteint (solutions approchée). La figure 2.1.2 illustre le principe.

2.1.3 Programmation en nombres entiers

Une autre façon de parvenir à un algorithme exact pour le coloriage de graphe est d'utiliser la programmation en nombres entiers (IP : *Integer Programming*), qui est un type spécialisé du modèle de la programmation linéaire. La programmation linéaire peut être considérée comme une méthodologie générale pour parvenir à des solutions optimales pour des modèles mathématiques linéaires. Ces modèles sont composés par des variables, des contraintes linéaires et une fonction linéaire objective. Les variables (ou décisions) prennent des valeurs numériques, et les contraintes sont utilisées pour définir des plages possibles de valeurs pour ces variables. La fonction objectif est ensuite utilisée pour mesurer la qualité d'une solution et de définir quelles affectation particulière des valeurs aux variables est considérée comme optimal.

D'une manière générale, les variables de décision des modèles de programmation linéaires sont continues dans le sens où elles sont autorisées à être fractionnée. D'un autre côté, les problèmes de programmation en nombres entiers sont d'un type particulier des modèles de programmation linéaire dans lequel une partie, sinon la totalité, des variables de décision sont limitées à des valeurs entières.

Considérons maintenant quelques formulations IP du problème de coloriage de graphe. Soit $G = (V, E)$ un graphe avec n sommets et m arêtes. Peut-être la plus simple consiste à utiliser la formulation de deux matrices binaires $X_{n\times n}$ et Y_n pour maintenir les variables du problème. Ceux-ci sont interprétés comme suit :

$$X_{ij} = \begin{cases} 1 \text{ si le sommet } v_i \text{ est associé à la couleur } j. \\ 0 \text{ si non.} \end{cases} \tag{2.1}$$

$$Y_j = \begin{cases} 1 \text{ si au moins un sommet est associé à la couleur } j. \\ 0 \text{ si non.} \end{cases} \tag{2.2}$$

Les éléments de X et Y sont non seulement des entiers, mais aussi des binaires. Ceci est une restriction commune dans les modèles de programmation entiers et est nécessaire ici parce que deux options sont disponibles : le sommet v_i est soit attribué à la couleur j, ou non. L'objectif dans ce modèle est de minimiser maintenant le nombre de couleurs utilisées à travers la fonction objectif.

$$\min \sum_{j=1}^{n} Y_i \tag{2.3}$$

Sous les contraintes suivantes :

$$X_{ij} + X_{lj} \leq Y_j \quad \forall (v_i, v_l) \in E, \quad \forall j \in \{1, ..., n\} \tag{2.4}$$

$$\sum_{j=1}^{n} X_{ij} = 1 \quad \forall v_i \in V. \tag{2.5}$$

La contrainte (2.4) assure qu'aucune paire de sommets adjacents sont affectés à la même couleur et que $Y_j = 1$ si et seulement si un sommet est affecté à la couleur j.
La contrainte (2.5) précise ensuite que chaque sommet doit être affecté à exactement une couleur.

2.2 Algorithmes approximatifs

2.2.1 Heuristiques

Une heuristique est une méthode de calcul qui fournit rapidement (en temps polynomial) une solution réalisable, pas nécessairement optimale, pour un problème d'optimisation $\mathcal{NP}$-*difficile*.

Algorithme glouton

Un algorithme glouton (Greedy) fait toujours le choix qui lui semble le meilleur sur le moment. Autrement dit, il fait un choix localement optimal dans l'espoir que ce choix mènera à une solution globalement optimale. La qualité de la solution dépend de l'ordre de traitement des sommets.

Description

— Trier les sommets sur un ordre arbitraire π.

— Pour chaque sommet π_i chercher une classe adéquate S_j (couleur possible).

— S'il n'est pas possible d'ajouter le sommet à une classe existante, créer une nouvelle classe.

Algorithme 2 : Algorithme glouton.

Données : $G(V, E)$
Résultat : *Nombre de couleurs*
début
  $\pi \longleftarrow Permutation(V)$
  $S \longleftarrow \emptyset$
  pour $i \leftarrow 1$ **à** $|\pi|$ **faire**
    pour $j \leftarrow 1$ **à** $|S|$ **faire**
      si $\{S_j \cup \{\pi_i\}\}$ *est un stable* **alors**
        $S_j \longleftarrow S_j \cup \{\pi_i\}$
        break
      sinon
        $j \longleftarrow j + 1$
    si $j > |S|$ **alors**
      $S_j \longleftarrow \{\pi_i\}$
      $S \longleftarrow S \cup \{S_j\}$
  retourner $|S|$

Complexité : Dans le pire des cas chaque sommet π_i va créer une nouvelle classe de couleurs, ce qui est expliqué par une itération de 1 à $i - 1$.
Pour l'ensemble des sommets : $1+2+3+4+...+(i-1)$. Donc la complexité globale est : $\mathcal{O}(n^2)$.

Un algorithme glouton peut donner une solution optimale si le choix est fait sur le bon ordre des sommets.

Algorithme de DSATUR

L'algorithme de DSATUR (abrégé de «degré de saturation») était à l'origine proposée par Brélaz (1979). Il est très similaire dans le comportement de l'algorithme glouton du fait qu'il prend à son tour, chaque sommet selon un certain ordre, puis assigne à la première classe de couleur appropriée, la création de nouvelles classes de couleurs si nécessaire.

Description : La différence entre les deux algorithmes réside dans la façon dont ces sommets sont ordonnés : Avec l'algorithme glouton l'ordre est décidé avant le processus de coloriage ; d'autre part, pour l'algorithme DSATUR, le choix de la couleur du sommet suivant est décidé d'une manière heuristique basée sur les caractéristiques du courant coloration partielle du graphe. Ce choix est basé principalement sur la saturation du degré des sommets.

Algorithme 3 : Algorithme utilisant le degré de saturation DSATUR.

Données : $G(V, E)$
Résultat : *Nombre de couleurs*
début
  $S \longleftarrow \emptyset$
  $X \longleftarrow V$
  tant que $X \neq \emptyset$ **faire**
    Choisir $v \in X$
    pour $j \leftarrow 1$ **à** $|S|$ **faire**
      si $\{S_j \cup \{v\}\}$ *est un stable* **alors**
        $S_j \longleftarrow S_j \cup \{v\}$
        break
      sinon
        $j \longleftarrow j + 1$
    si $j > |S|$ **alors**
      $S_j \longleftarrow \{v\}$
      $S \longleftarrow S \cup \{S_j\}$
    $X \longleftarrow X - \{v\}$
  retourner $|S|$

Complexité : Par conséquent, dans le pire des cas, la complexité des DSATUR est le même qu'un algorithme glouton à $\mathcal{O}(n^2)$, bien que dans la pratique un peu de comptabilité supplémentaire est nécessaire pour garder une trace des degrés de saturation des sommets non colorés.

Algorithme récursif "Le plus grand en premier" (RLF)

L'algorithme RLF (Recursive Largest First) a été initialement conçu par *Leighton* (1979), en partie pour une utilisation dans la construction de solutions aux grands problèmes de l'emploi du temps.

Description : La méthode fonctionne en coloriant un graphe à la fois avec une couleur, par opposition à un sommet à la fois. Dans chaque étape, l'algorithme utilise des heuristiques pour identifier un stable du graphe, qui est ensuite associé à une couleur. Ce stable est ensuite retiré du graphe, et le processus est répété sur les plus petit sous-graphe résultant. Ce processus se poursuit jusqu'à ce que le sous-graphe est vide ; jusqu'à ce que tous les sommets ont été colorés, et on a une solution réalisable.

Algorithme 4 : Algorithme récursif "Le plus grand en premier" (RLF)

Données : $G(V, E)$
Résultat : *Nombre de couleurs*
début
  $S \longleftarrow \emptyset$
  $X \longleftarrow V$
  $Y \longleftarrow \emptyset$
  $i \longleftarrow 0$
  tant que $X \neq \emptyset$ **faire**
    $i \longleftarrow i + 1$
    $S_i \longleftarrow \emptyset$
    tant que $X \neq \emptyset$ **faire**
      Choisir $v \in X$
      $S_i \longleftarrow S_i \cup \{v\}$
      $Y \longleftarrow Y \cup \Gamma_X(v)$
      $X \longleftarrow X - (Y \cup \{v\})$
    $S \longleftarrow S \cup \{S_i\}$
    $X \longleftarrow Y$
    $Y \longleftarrow \emptyset$
  retourner i

Complexité : Leighton (1979) a prouvé que dans le pire des cas la complexité de RLF sera $\mathcal{O}(n^3)$, en lui donnant un coût de calcul plus élevé que $\mathcal{O}(n^2)$ des algorithmes gloutons et DSATUR ; Cependant, cet algorithme reste de complexité polynomiale bornée.

Welsh Powell

C'est un algorithme "heuristique" séquentiel qui a été développé dans les années soixante par Welsh, D. J. A et Powell, M. B, pour la coloration de graphes dans un temps polynomial [WP67]. L'algorithme donne une solution approchée.

Description :

1. Trier les sommets selon leurs degrés et leur attribuer des numéros d'ordre.
2. Commencer par colorer les sommets selon cet ordre en donnant une couleur à chaque sommet non colorié et la même couleur pour le reste des sommets non coloriés et non adjacents à ce sommets..
3. Répéter l'étape (2) s'il existe des sommets qui ne sont pas encore coloriés.

Complexité : L'algorithme commence par le tri des sommets dans une liste, ce qui représente une complexité de $\mathcal{O}(n^2)$ au pire de cas, puis n parcours de la liste triée avec un

coût de l'ordre $\mathcal{O}(n^2)$. Donc la complexité globale de l'algorithme est $\mathcal{O}(n^2)$.

2.2.2 Méta-heuristiques

Les méta-heuristiques sont généralement des algorithmes stochastiques itératifs, qui progressent vers un optimum global, c'est-à-dire l'extremum global d'une fonction, par échantillonnage d'une fonction objectif. Elles se comportent comme des algorithmes de recherche, tentant d'apprendre les caractéristiques d'un problème afin d'en trouver une approximation de la meilleure solution (d'une manière proche des algorithmes d'approximation). Parmi Les méta-heuristiques utilisé dans le coloriage de graphes La Recherche Locale, la technique du Recuit Simulé [KV$^+$83] et la Recherche Taboue [Glo86].

Recherche locale

Parfois connu sous le nom de "recherche de voisinage", les algorithmes de recherche locale utilisent des opérateurs de voisinage qui sont des systèmes simples pour changer une solution candidate particulière.

Description : Au début un voisinage $N(s)$ est défini pour chaque solution s à l'aide d'un opérateurs de voisinage, et une fonction économique des solutions $f(s)$.

— Générer une solution initiale s et l'utiliser comme optimale $s^* = s$.

— Générer une solution s' dans le voisinage $N(s)$ de s.

— Poser $s = s'$ et mettre à jour s^* si $f(s) < f(s^*)$.

— Répéter les étapes suivantes jusqu'à la satisfaction d'une condition d'arrêt (temps, écart).

Recherche Taboue

les travaux de Chams et al. (1987) et Hertz et de Werra (1987) ont proposé que les deux méta-heuristiques *le recuit simulé* et *la recherche tabou* sont adaptés pour aider au traitement des problèmes de coloriage de graphes. Une méthode de recherche tabou appelé TABUCOL en particulier devient très populaire.

TABUCOL a été utilisé comme un sous-programme de recherche locale dans un certain nombre d'algorithmes hybrides performants, y compris ceux de Dorne et Hao (1998), Galinier et Hao (1999), Avanthay et al. (2003) et Thompson et Dowsland (2008).

La Recherche Taboue utilise une procédure de recherche locale pour se déplacer de manière itérative d'une solution potentielle s à une solution améliorée s' dans le voisinage de s, jusqu'à ce que un critère d'arrêt soit satisfait (en général, une limite de temps ou un seuil de score).

Les procédures des recherches locales deviennent souvent coincés dans les régions de mauvais résultats. Afin d'éviter ces écueils et d'explorer les régions de l'espace de recherche qui serait laissé inexploré par d'autres procédures de recherche locale, la recherche taboue explore soigneusement le voisinage de chaque solution quand la recherche progresse.

Description du TABUCOL : TABUCOL opère dans l'espace de $k-coloriages$ impropres complets à l'aide d'une fonction objectif qui compte simplement le nombre d'affrontements défini par :

$$f(s) = \sum_{\forall (u,v) \in E} g(u,v) \tag{2.6}$$

avec :

$$g(u,v) = \begin{cases} 1 \text{ si } c(u) = c(v). \\ 0 \text{ si non.} \end{cases} \tag{2.7}$$

Compte tenu d'une solution candidate $s \in S$, $s = \{s_1, ..., s_k\}$. Les déplacements dans l'espace des solutions, sont effectuées en sélectionnant un sommet $v \in s_i$ dont l'affectation à la classe de couleur s_i est actuellement responsable d'un choc, puis l'assigner à une nouvelle classe de couleur $s_j \neq s_i$. Une amélioration de cet algorithme (Galinier et Hertz, 2006) consiste à sélectionner seulement les sommets qui ne posent pas des chocs.

La liste de l'algorithme tabou est stockée en utilisant une matrice $T_{n \times k}$. Si, à l'itération l de l'algorithme, l'opérateur de voisinage transfère un sommet v de S_i vers S_j, l'élément T_{vi} devient $l+t$, où t est un nombre entier positif qui signifie que le mouvement de v pour revenir à la classe de couleur s_i est un tabou (c.a.d refusé) pour t itérations de l'algorithme. t est un paramètre de la méthode taboue, il peut être une valeur statique ou dynamique.

L'algorithme de la Recherche Taboue Générique (RTG) suit les étapes suivantes :

— Configuration initiale :générer une coloration initiale (avec k couleurs).

— Régénération de configuration : produire rapidement un coloriage à $k-1$ couleurs avec le minimum de conflits.

— Recherche d'un coloriage propre :essayer de réduire le nombre de conflits à zéro.

L'algorithme s'arrête quand il trouve un coloriage optimal, ou quand le nombre d'itérations permises est atteint sans trouver une coloration propre avec le nombre de couleurs courant k. Le plus petit nombre k trouvé pour une coloration propre est retourné.

Algorithmes évolutifs hybrides (HEA)

Une autre méthode méta-heuristique utilisée dans le coloriage de graphes est l'algorithme évolutionnaire hybride (HEA) de Galinier et Hao (1999). Le HEA fonctionne par le maintien d'une population de solutions candidates qui se génère par l'intermédiaire d'un opérateur de recombinaison spécifique du problème, et un procédé de recherche locale. Comme TABUCOL, le HEA fonctionne dans l'espace de $k-coloriages$ impropres complets en utilisant la fonction de coût f.

L'algorithme commence par la création d'une population initiale de solutions candidates. Chacun des membres de cette population est formé en utilisant une version modifiée de l'algorithme de DSATUR pour laquelle le nombre de couleurs k est fixé au départ.

A fin de fournir la diversité entre les membres, le premier sommet est sélectionné au hasard et affecté à la première couleur. Les sommets restants sont ensuite pris en séquence en fonction du degré de saturation maximal et affectés à la classe la plus basse indexée couleur s_i considérée comme réalisable (où $1 \leq i \leq k$).

Lorsque les sommets sont rencontrés pour lesquels aucune classe de couleur possible existe, ceux-ci sont tenus d'un côté et sont affectés à des classes de couleurs aléatoires à la fin de ce processus. Lors de la construction de cette population initiale, une tentative est alors faite pour améliorer chaque membre en appliquant la routine de recherche locale.

Dans son exécution, l'algorithme fait évoluer la population en utilisant la **recombinaison**, la **mutation** et la **pression évolutive**. A chaque itération deux solutions pères s_1 et s_2 sont choisis parmi la population au hasard.

Principe d'un algorithme évolutionnaire typique :

— deux solutions pères s_1 et s_2 sont choisis parmi la population au hasard.

— Les solutions pères sont utilisées en conjonction avec l'opérateur de *recombinaison* pour produire une solution fils s'.

— Cette solution s' est améliorée via l'opérateur de recherche locale, et est inséré dans la population en remplaçant le plus faible de ses deux parents (la pression évolutive).

En plus de ces méthodes cités dans ce chapitre, une autre approche exacte basée sur la division de l'espace des solutions, c'est la méta-méthode de séparation et évaluation qui sera traitée dans le chapitre suivant.

Conclusion

Une panoplie de familles d'algorithmes de coloriage de graphes a été décrite dans ce chapitre. Parmi ces algorithmes on trouve les algorithmes exactes, dans le chapitre prochain, on décrira une nouvelle approche exacte de coloriage à base de la méta-heuristique "Séparation et Évaluation".

Chapitre 3

Notre approche de coloriage

Introduction

Notre approche de coloriage est une amélioration d'un algorithme proposé par Cherroun et Feautrier dans le cadre de l'utilisation de coloriage dans la synthèse matérielle à haut niveau (HLS) dont l'objectif est de convertir une spécification comportementale pour la totalité ou une partie d'une application, afin de l'effectuer sur un circuit dédié [CF07].

L'algorithme est à l'origine basé sur la méta-méthode Séparation et évaluation (*Branch - and Bound*) et la clique maximale d'un graphe calculé approximativement. Ensuite, une fonction d'évaluation thêta (ϑ) est utilisée dans l'étape d'évaluation, cette fonction est ajoutée comme amélioration par Nehar et al. [Neh09].

3.1 La méta-méthode Évaluation et Séparation

Un algorithme par séparation et évaluation, ou *branch and bound*, est une méthode générique de résolution de problèmes d'optimisation combinatoire. Cette méthode est très utilisée pour résoudre des problèmes $\mathcal{NP}$-*difficiles*, c'est-à-dire des problèmes considérés comme difficiles à résoudre efficacement sur les machines contemporaines.

Dans les méthodes par séparation et évaluation, la séparation permet d'obtenir une méthode générique pour énumérer toutes les solutions tandis que l'évaluation évite l'énumération systématique de toutes les solutions.

3.1.1 Séparation

La phase de séparation consiste à diviser le problème en un certain nombre de sous-problèmes qui ont chacun leur ensemble de solutions réalisables S_i de telle sorte que tous ces ensembles forment un recouvrement (idéalement une partition) de l'ensemble S.

$$S = \bigcup_{i=1}^{n} S_i. \tag{3.1}$$

$$S_i \cap S_j = \emptyset \quad \forall \quad i \neq j. \tag{3.2}$$

Ainsi, en résolvant tous les sous-problèmes et en prenant la meilleure solution trouvée, on est assuré d'avoir résolu le problème initial. Ce principe de séparation peut être appliqué de manière récursive à chacun des sous-ensembles de solutions obtenus, et ceci tant qu'il y a des ensembles contenant plusieurs solutions. Les ensembles de solutions (et leurs sous-problèmes associés) ainsi construits ont une hiérarchie naturelle en arbre, souvent appelée *arbre de recherche* ou *arbre de décision*.

3.1.2 Évaluation

L'évaluation d'un nœud de l'arbre de recherche a pour but de déterminer l'optimum de l'ensemble des solutions réalisables associé au nœud en question ou, au contraire, de prouver mathématiquement que cet ensemble ne contient pas de solution intéressante pour la résolution du problème (typiquement, qu'il n'y a pas de solution optimale). Lorsqu'un tel nœud est identifié dans l'arbre de recherche, il est donc inutile d'effectuer la séparation de son espace de solutions.

Pour déterminer qu'un ensemble de solutions réalisables ne contient pas de solution optimale, la méthode la plus générale consiste à déterminer une borne inférieure pour le coût des solutions contenues dans l'ensemble (s'il s'agit d'un problème de minimisation). Si on arrive à trouver une borne inférieure de coût supérieur au coût de la meilleure solution trouvée jusqu'à présent, on a alors l'assurance que le sous-ensemble ne contient pas l'optimum.

3.1.3 Paramètres d'un Branch & Bound

Tout algorithme basé sur la méta-méthode de séparation et évaluation, doit comporte des deux critères suivants :

- Heuristique donnant la fonction d'évaluation (minorant ou majorant).
- Critère de séparation (selon quel critère on partitionne un nœud en deux (2) ou plusieurs sous nœuds).

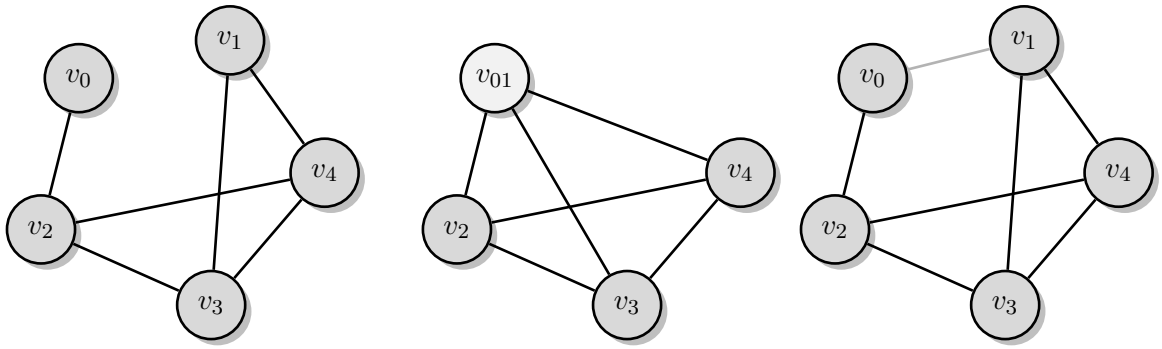
En plus, plusieurs stratégies peuvent être suivies pendant le parcours de l'arbre des solutions [Zha96] :

- Le meilleur en premier (Best-First Search).
- Parcours en profondeur (Depth-First Search).
- Parcours itératifs en profondeur fixe (Itérative deeping).
- Recursive Best-First Search.
- Constant-Space Best-First Search.

3.2 Principe de l'algorithme

Notre approche pour trouver le nombre chromatique d'un graphe G est induite de l'idée que lorsque on veut attribuer les couleurs aux sommets, deux sommets non-adjacents (u, v) ont deux possiblités : (1) soient ils prennent la même couleur $c(u) = c(v)$, (2) soient ils prennent des couleurs différentes $c(u) \neq c(v)$. Cette idée revient à *Bollobas* [Bol98].

Dans le premier cas les deux sommets peuvent être fusionnés et deviennent un seul sommet dans le nouveau graphe G'. Dans le cas contraire une nouvelle arête est ajoutée entre les deux sommets construisant un nouveau graphe G''. (voir Figure 3.1).



(G) Graphe original (G'') Fusion de deux sommets (G') Nouvelle arête

FIGURE 3.1 – Les graphes G, G' et G''.

3.2.1 Stratégie de l'exploration

Cette méthode sera appliquée d'une manière récursive sur les deux nouveaux graphes G' et G'' . En effet, tout coloriage de l'un des deux graphes G' et G'' est aussi le coloriage du graphe original G, et dans l'ensemble on cherche la meilleure solution sur l'arbre des solutions par l'application de la formule suivante :

$$\chi(G) = \min\{\chi(G'), \chi(G'')\} \tag{3.3}$$

L'exploration de l'espace des solutions est faite avec la technique de séparation et celle de l'évaluation. Le parcours de l'arbre des solutions (figure 3.2) est fait par un algorithme DFS (Depth First Search figure) jusqu'à l'arrivée aux graphes complets au niveau des feuilles de l'arbre. où il n'y a pas des sommets non-adjacents et le nombre chromatique est le degré de ce graphe :

$$\chi(G) = |V| \tag{3.4}$$

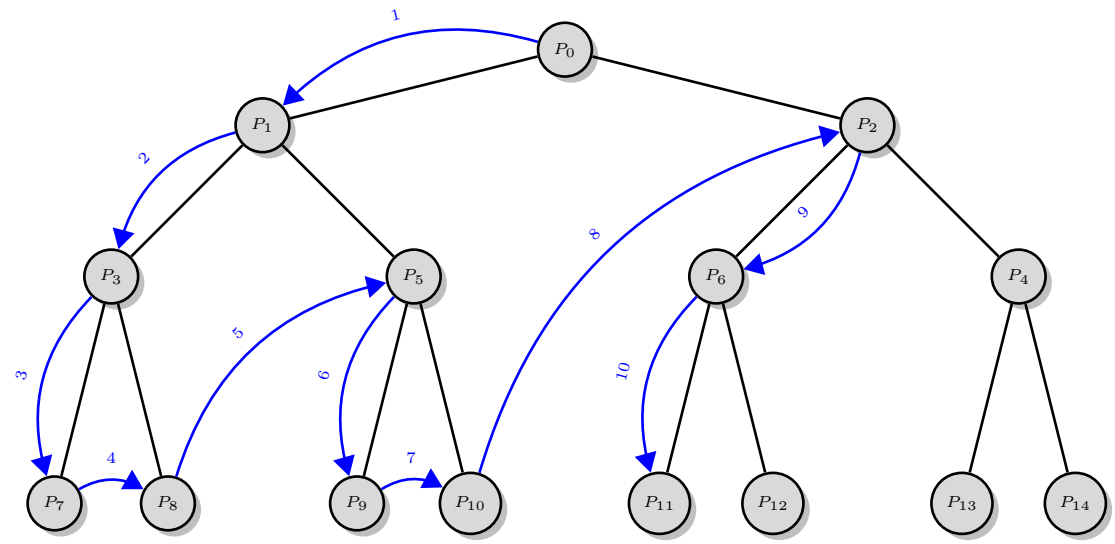


FIGURE 3.2 – Stratégie d'exploration DFS (Algorithme séquentiel).

La séparation

La séparation consiste à diviser l'espace des solutions à chaque étape en deux sous ensembles disjoints, d'où la génération des solutions possibles pour le problème. chaque graphe généré par la fusion des deux sommets choisis et par la création d'une nouvelle arête est considéré comme un nouveau problème de coloriage.

Principe d'évaluation

Cette étape permet d'éviter le parcours de toutes les possibilités en éliminant un sous-ensemble par un critère bien déterminé. Elle est basé sur la borne inférieure connue au nombre chromatique, $\chi(G) \geq \omega(G)$, où $\omega(G)$ est le nombre de clique (clique maximale) de G. Cette borne est garantie du fait que les sommets d'une clique requièrent des couleurs distinctes.

En général, pour n'importe quel graphe, le problème de la clique maximale est $\mathcal{NP}$-*difficile*. Donc on a besoin d'une autre borne qui doit être plus facile à calculer par rapport à la clique maximale et pour cela un nouveau nombre réel (ϑ) est exploité dans cette étape.

Au début, le chromatique sera initialisé par une borne supérieure (Dans notre cas $\Delta(G)+1$), et à chaque nouvelle instance de graphe un calcul de thêta sera effectué et le resultat sera comparé avec le meilleur nombre obtenu jusqu'à ce point pour estimer la possibilité d'une amélioration. Des initialisations différentes du chromatique peuvent affecter le temps d'exploration.

3.2.2 La fonction thêta

La fonction thêta est populairement connu comme le nombre Lovász d'un graphe G [GLS88]. Il existe de nombreuses façons de définir $\vartheta(G)$, et la surprenante variété de différentes ca-

ractérisations indique en elle-même que $\vartheta(G)$ devrai être intéressant. Mais la propriété la plus intéressante de $\vartheta(G)$ est probablement le fait qu'elle peut être calculée de manière efficace, même si elle se trouve entre deux autres nombres de graphes classiques dont le calcul est $\mathcal{NP}$-*difficile*. Cet encadrement est introduit par D.Knuth sous le nom de «théorème du Sandwich» [Knu94].

Théorème du Sandwich

Malgré qu'l est $\mathcal{NP}$-*difficile* pour calculer $\omega(G)$, la taille de la plus grande clique dans un graphe G, et il est $\mathcal{NP}$-*difficile* pour calculer $\chi(G)$, le nombre minimal de couleurs nécessaires pour colorer les sommets de G, mais Grötschel, Lovász et Schrijver ont prouvé [GLS81] que l'on peut calculer en temps polynomial un nombre réel qui est "pris en sandwich" entre ces entiers difficiles à calculer :

$$\omega(G) \leq \vartheta(\overline{G}) \leq \chi(G) \tag{3.5}$$

Calcul de thêta

Le calcul de thêta se base sur une représentation orthogonale des graphes [LV02], puis une optimisation convexe (OC) est résolue à l'aide de programmation semi-définie.

Représentation orthogonale d'un graphe

Soit $G = (V, E)$ un graphe simple. Nous désignerons par $\overline{G} = (V, \overline{E})$ son complément.

Une représentation orthogonale d'un graphe G dans R^d attribue à chaque $v_i \in V$ un vecteur non nul $u_i \in R^d$ telle que $u_i^T u_j = 0$ (vecteurs orthogonaux) $\forall \quad e_{ij} \in \overline{E}$. Une représentation orthonormée est une représentation orthogonale dans laquelle tous les vecteurs ont une longueur unitaire.

On note que les sommets différents ne sont pas forcément représentés par des vecteurs différents, ni que les sommets adjacents sont représentés par des vecteurs non-orthogonaux. Si ces conditions sont tenues en compte, on l'appelle la une représentation orthogonale fidèle.

En effet, tout graphe possède au moins une représentation orthogonale (orthonormale) triviale dans R^n, dans laquelle un sommet v_i est représenté par un vecteur de base euclidienne standard u_i. Cependant, on s'intéresse aux représentions moins triviales, qui sont plus économiques (i.e. $d < n$).

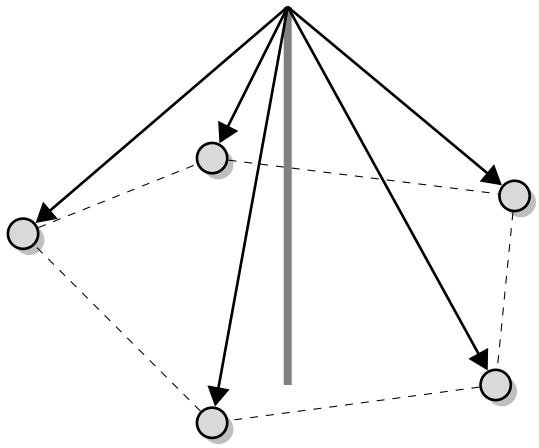


FIGURE 3.3 – Une représentation orthogonale d'un pentagone.

La figure 3.2.2 illustre une représentation orthogonale d'un pentagone. Le pentagone C_5, un graphe G à 5 sommets formant une chaine fermée, peut avoir une représentation orthogonale qui aura la forme d'un parapluie. En effet, les 5 vecteurs formant le parapluie sont assignés aux sommets de C_5 de tel sorte qu'ils forment le même angle avec c (le vecteur représentant le manche du parapluie) et que tous deux sommets non adjacents soit étiquetés par des vecteurs orthogonaux. Ces vecteurs constituent une représentation orthogonale de C_5 dans R^3.

Le plus petit cône et la fonction thêta

A fin de trouver des représentations "économiques" orthogonales, on peut définir cette minimisation de plusieurs façons.Par exemple, on peut vouloir trouver une représentation orthogonale dans une dimension aussi faible que possible.

Parmi les fonctions économiques qui a motivé l'introduction de représentations orthogonales c'est la théorie de l'information et la capacité de *Shannon*. Le but de cette économie est de choisir des mots de longueur k de telle façon qu'aucune confusion entre les mots ne se produit.

Ce problème peut être représenté par le plus petit angle de cône de rotation (en dimension arbitraire) qui contient tous les vecteurs dans une représentation orthogonale du graphe donne lieu à la fonction thêta du graphe. Formellement, on a [Lov79] :

$$\vartheta(G) = \min_{u_i,c} \max_{i \in V} \frac{1}{(c^T u_i)^2}. \tag{3.6}$$

La fonction thêta, notée $\vartheta(G)$ est un cas particulier de la fonction thêta pondérée $\vartheta(G, w)$ pour laquelle $w = 1$.
On peut formuler $\vartheta(G, w)$ comme étant un Problème Semi-Défini (PSD), qui peut être résolu en temps polynomiale. Il établit le théorème suivant, qui montre que $\vartheta(G, w)$ est approximé à un réel t avec une erreur ϵ.

$$|\vartheta(G, w) - t| < \epsilon. \tag{3.7}$$

Notation

On note par S_n l'espace des matrices symétriques $n \times n$. Cet espace est doté du produit :

$$\langle A, B\rangle_{S_n} = tr(A^T B) = \sum_{i=1}^{n}\sum_{j=1}^{n} A_{ij}B_{ji} \tag{3.8}$$

avec : tr représente la trace d'une matrice, A et B deux matrices carrées.

Une matrice symétrique est dite semi-définie positive (et on la note $A \succeq 0$), si toutes ces valeurs propores sont non négatives. De même, $A \succ 0$, $A \preceq 0$, $A \prec 0$ signifient que A est définie positive, semi-définie négative et définie négative, respectivement.

On note par S_n^+ le cône connexe des matrices $n \times n$ semi définies. Ce cône définit un ordre partiel pour $A, B \quad \in S_n$ (noté $A \succeq B$) si $(A-B)$ est une matrice semi définie positive ($(A-B) \succeq 0$).

Programmation semi-définie

La programmation semidéfinie est un sous-champ de l'Optimisation Convexe (OC), concernée par l'optimisation d'une fonction objectif linéaire sur l'intersection du cône de matrices semi définie positives avec un espace affine. On peut dire aussi qu'elle est un problème d'optimisation continue, et elle est considérée comme une extension de la programmation linéaire. De nombreuses méthodes ont été proposées pour la programmation linéaire et ont ensuite été généralisées pour la programmation semi-définie. Un programme semi définie dit primal, sous forme standard s'écrit comme suit :

$$PSDP = \begin{cases} min\langle C, X\rangle \\ \text{s.c } A.X = b \\ X \geq 0 \end{cases} \tag{3.9}$$

avec $b \in R$ et A est un opérateur linéaire de S_n dans R^m tel que :

$$\langle C, X\rangle = \begin{pmatrix} \langle A_1, X\rangle \\ \vdots \\ \langle C_m, X\rangle \end{pmatrix} \tag{3.10}$$

C et A_i, $i = 1, ..., m$, sont des matrices de M_n. Sans perte de généralité on peut supposer que C et les A_i sont symétriques.

Formulation de thêta

Une formulation de $\vartheta(G)$ comme optimum d'un programme semi défini est donnée comme suit [LV02] :

$$Minimiser \quad t$$
$$Sous-contrainte \begin{cases} Y \geq 0 \\ Y_{ij} = -1(\forall \quad e_{ij} \in \overline{E}) \\ Y_{ii} = t - 1 \end{cases} \tag{3.11}$$

Caractérisations de thêta

Donald E. Knuth , a donné quatre caractérisations pour la fonction thêta [Knu94] :

— Caractérisation par les valeurs propres de matrices faisables.

— Caractérisation par les coûts des vecteurs des représentations orthogonales de G.

— Caractérisation par les représentations orthogonales du graphe complémentaire $\overline{G}$ (deux caractérisations différentes).

3.3 Description de l'algorithme

L'algorithme fonctionne comme suit :

— Au niveau de la racine, on commence par le graphe G (problème initial P_0), et par l'initialisation du *MeilleurChrom* par la valeur $\Delta(G) + 1$ qui est considéré comme une borne supérieure (voir Théorème 1.3.1).

— Dans chaque nœud P_i de l'arbre des problèmes générés, deux sommets non-adjacents sont choisis (plusieurs critères sont possibles), et la séparation est faite en utilisant la règle décrite précédemment.

— A chaque nœud interne P_i, autre qu'une feuille, un calcul de thêta $\vartheta(G)$ est réalisé. Si $\lceil \vartheta(G) \rceil \geq$ *MeilleurChrom* le sous arbre ne sera pas construit car il ne mènera plus à une solution meilleure.

— Une feuille est atteinte quand il n'y a plus de sommets non-adjascents dans le graphe G_i, une solution triviale est alors obtenue (égale à $|V|$). Si cette solution est meilleure que *MeilleurChrom*, alors *MeilleurChrom* est mis à jour.

— L'algorithme s'arrête quand toutes les branches sont explorées, et *MeilleurChrom* est retourné comme solution optimale.

Algorithme 5 : Algorithme exact à séparation et évaluation

Données : $G(V, E)$
Résultat : $\chi(G)$
P : Pile
$MeilleurChrom$: Entier
C : Couple de sommets
début
  /* MeilleurChrom est initialisé à la valeur $\Delta(G) + 1$ */
  $MeilleurChrom \longleftarrow ValeurInitiale()$
  $Empiler(G, P)$
  tant que $NonVide(P)$ **faire**
    $G \longleftarrow Depiler(P)$
    si $EstComplet(G)$ **alors**
      si $|V| < MeilleurChrom$ **alors**
        $MeilleurChrom \longleftarrow |V|$
    sinon
      si $\lceil \theta(G) \rceil < MeilleurChrom)$ **alors**
        $C \longleftarrow (v_i, v_j)$ tel que v_i, v_j ne sont pas adjacents.
        $G' \longleftarrow FusionnerSommets(G, C)$
        $G'' \longleftarrow LierSommets(G, C)$
        $Empiler(G'', P)$
        $Empiler(G', P)$
  retourner $MeilleurChrom$

3.4 Complexité de l'algorithme

Les algorithmes de type séparation & évaluation sont -en général- difficiles à estimer leur complexité à cause de l'absence des informations concernant le nombres des sous-problèmes traités et les résultats de l'étape de l'évaluation au niveau de chaque nœud.

Dans notre cas, on peut simplement donner une estimation en utilisant le temps de traitement de chaque sous-problème. Si on suppose que l problèmes sera être traité dont la complexité n^k pour chacun du fait que thêta est calculé dans un temps polynomial ($k \in N$), on obtiendra la complexité globale est $\mathcal{O}(l * n^k)$.

3.5 Expérimentation

Dans cette partie on présente l'étude expérimentale de notre approche. On commence par décrire l'environnement d'exécution, les outils d'implémentation et les jeux de test (benchmarks) sur lesquels on mesure les performances. On achèvera par une discussion des résultats obtenus.

3.5.1 L'environnement et les outils utilisés

Machine & Système d'exploitation : L'environnement de test est constitué d'une machine avec les caractéristiques suivantes :

- Processeur *Intel Core I5* 2.5 *Ghz* avec deux (2) coeurs, et deux (2) threads pour chacun.
- 3 *Mo* de mémoire cache niveau 3.
- 4 *Go* de mémoire vive.
- Espace disque suffisant.

Le système d'exploitation est une version *32bit* de la distribution Debian Linux 8.

Langage & bibliothèques : L'algorithme est implémenté entièrement en langage *ANSI-C* pour des raisons de performance, et pour cela on a utilisé le compilateur *Gnu GCC* version 4.9.2. Le calcul de thêta est effectué à l'aide d'une bibliothèque de résolution des problème de la programmation semi-définie appelé *CSDP* avec une modification pour que les routines de cette bibliothèque seront appeler directement.

La bibliothèque CSDP : Est une bibliothèque de routines écrite par Brian Borchers, et qui implémente une variante de l'algorithme de résolution des programmes semi-défini de Helmberg et al [HRVW96]. L'avantage principale de cet outil est qu'il est écrit pour être utilisé comme un appel à des routines (notamment la routine de calcul de thêta).

La bibliothèque CSDP est écrit en langage ANCI-C pour l'efficacité, et qu'il fait un bon usage des matrices creuses des contraintes à l'aide des bibliothèques **BLAS** (*Basic Linear Algebra Subprograms*) pour le calcul matriciel et **LAPACK** qui sont des routines pour résolution des systèmes d'équations linéaires.

Le jeu de test (benchmarks) : Un ensemble de graphes est choisi pour mesurer la performance de notre algorithme, ces graphes sont extraits à partir des applications réelles. La plus part des instances sont à l'origine de *DIMACS Challenge* dont le but de création est de faciliter les efforts nécessaires pour tester et comparer des algorithmes et des heuristiques en fournissant un ensemble riche d'instances de graphes et d'outils d'analyse.

Un autre ensemble de test est fait sur des instances de graphe générées d'une manière aléatoire.

3.5.2 Résultats et interprétations

Benchmarks

Les résultats obtenus sont résumés dans le tableau 3.1, les significations des colonnes sont :

— **Instance** : le libellé de l'instance.

— $|\mathbf{V}|$: Nombre de sommets.

— $|\mathbf{E}|$: Nombre d'arêtes.

— $\mathbf{Val}_{init}$: la valeur d'initialisation au début de parcours.

— $\vartheta(\overline{G})$: le nombre thêta calculé pour le graphe.

— $\chi(G)$: le nombre chromatique trouvé.

— $\mathbf{Spr}_{Arb}$: le nombre de séparations lors du parcours des nœuds avec un choix arbitraire (par défaut) des sommets.

— $\mathbf{Temps}_{Arb}(s)$: le temps (en secondes) de traitement avec un choix arbitraire (par défaut) des sommets.

— $\mathbf{Spr}_{max}$: le nombre de séparations en choisissant les sommets non-adjacents ayant les plus grands degrés.

— $\mathbf{Temps}_{max}(s)$: le temps (en secondes) de traitement en choisissant les sommets non-adjacents ayant les plus grands degrés.

— $\mathbf{Spr}_{min}$: le nombre de séparations en choisissant les sommets non-adjacents ayant les plus petits degrés.

— $\mathbf{Temps}_{min}(s)$: le temps (en secondes) de traitement en choisissant les sommets non-adjacents ayant les plus petits degrés.

Concernant les case "-", ceci veut dire que l'évaluation a dépassé 1 heure.

D'après le tableau 3.1, dans la majorité des cas, on remarque qui le nombre ϑ est très proche du nombre chromatique, et dans plusieurs cas ils sont égaux. Cette observation indique que ϑ est une bonne approximation et peut être utilisé dans les cas ou on s'intéresse seulement à des solutions approchées.

Une deuxième observation concerne les critères de sélection des sommets à chaque étape de séparation. Le nombre d'opération de séparation est plus petit dans le cas où les sommets avec des degrés élevés sont choisi que celui dans des autres critères. Cependant, le gain en temps de traitement est faible pour ce critère, ce qui peut être expliqué par le surcoût des opérations supplémentaires de tri (de l'ordre $\mathcal{O}(n^2)$).

Graphes aléatoires

Le but d'utiliser des instances de graphes générées aléatoirement est de mesurer, d'une part, l'effet de la taille du graphe sur le temps d'exécution, et d'autre part, l'effet de la densité du graphe. Dans le tableau 3.2, on présente les résultats d'exécution pour diverses instances de taille croissante, et dans le tableau 3.3, on présente les résultats pour des graphes de taille fixé à 36 sommets avec plusieurs valeurs de densité.

Instance	$\|V\|$	$\|E\|$	$\mathbf{Val}_{init}$	$\vartheta(\overline{G})$	$\chi(G)$	$\mathbf{Spr}_{Arb}$	$\mathbf{Temps}_{Arb}(s)$	$\mathbf{Spr}_{max}$	$\mathbf{Temps}_{max}(s)$	$\mathbf{Spr}_{min}$	$\mathbf{Temps}_{min}(s)$
css1	15	30	7	5.00	5	10	0.04	10	0.03	12	0.03
css11	15	25	6	3.00	3	23	0.06	22	0.07	-	-
css12	17	38	8	5.00	5	12	0.06	12	0.06	12	0.06
css2	32	64	8	5.00	5	28	2.11	27	2.11	43	2.05
css3	27	97	12	8.00	8	19	0.47	19	0.47	24	0.45
css5	9	16	6	4.00	4	5	0.01	5	0.01	5	0.01
css6	12	14	5	3.00	3	9	0.02	9	0.02	13	0.02
wss1	44	324	20	12.00	12	34	7.25	32	7.50	70	6.26
wss2	23	95	14	9.00	9	14	0.16	14	0.16	14	0.16
wss3	11	17	5	4.00	4	7	0.01	7	0.01	7	0.01
wss31	11	23	7	5.00	5	6	0.01	6	0.01	6	0.01
wmt22	31	154	17	7.00	7	57	2.01	24	0.911	116	1.45
woc1	13	23	7	5.00	5	8	0.02	8	0.02	9	0.02
woc2	9	12	5	4.00	4	5	0.01	5	0.01	5	0.01
rasm1	9	14	5	4.00	4	6	0.01	5	0.01	5	0.01
rasm2	7	11	5	4.00	4	3	0.01	3	0.01	3	0.01
jac1	19	36	8	5.00	5	26	0.17	14	0.10	14	0.10
myciel3	11	20	6	2.39	4	8	0.02	8	0.02	11	0.02
myciel4	23	71	12	2.52	5	2192	42.24	322	9.12	16418	218.63
queen5_5	25	160	17	5.00	5	41	0.25	38	0.24	127	0.73
queen6_6	36	290	20	6.04	7	175	9.46	150	9.84	1668	121.25
queen7_7	49	476	25	7.00	7	388	94.14	520	183.70	-	-
david	87	406	83	11.00	11	98	2328.54	76	2320.44	-	-
jac2	82	788	29	18.00	18	64	877.21	64	871.66	-	-
jac3	97	878	25	11.00	11	238	3073.07	86	3752.73	-	-

TABLE 3.1 – Résultats Branch-&-Bound sur différentes instances

Nbr_d'instances	**Nbr_de_Sommets**	ϑ_{moy}	**Nbr_Sep**$_{moy}$	**Temps**$_{moy}(s)$
25	5	2.6989	2.2	0.01
25	10	4.0199	7.1	0.01
25	15	4.7942	15.5	0.04
25	20	5.4636	26.2	0.13
25	25	5.8168	37.6	0.36
25	30	6.4146	59.0	1.13
25	35	6.7531	104.7	5.12
25	40	7.0297	622.4	92.46

TABLE 3.2 – Résultats sur des instances aléatoires de densité 0,5 .

Concernant les tests sur des graphes de taille croissante (tableau 3.2), on observe que le nombre moyen des opérations de séparations (**Nbr_Sep**$_{moy}$) pour 25 instances augmente rapidement, et par conséquence, le nombre de graphes traités, ainsi que le temps moyen (**Temps**$_{moy}$).

Nbr_d'instances	**Densité** D	ϑ_{moy}	**Nbr_Sep**$_{moy}$	**Temps**$_{moy}$
20	0.98	27.4000	10.9	0.18
20	0.95	22.4618	26.9	0.37
20	0.90	17.6158	54.4	0.66
20	0.80	13.1013	78.6	1.23
20	0.70	10.3668	124.0	3.36
20	0.60	8.2872	180.9	7.57
20	0.50	6.8893	347.0	27.91
20	0.40	5.6271	148.6	17.51
20	0.30	4.8332	117.0	17.55
20	0.20	3.9534	68.7	8.37
20	0.10	3.0000	63.6	9.54
20	0.05	2.4791	43.9	7.77
20	0.03	2.0500	50.1	8.08

TABLE 3.3 – Résultats sur des instances aléatoires de 36 sommets et de densité variable.

La densité d'un graphe est aussi un facteur essentiel, le tableau 3.3 illustre le fait que les graphes de densités moyenne et faible nécessitent plus de temps que les graphe denses, ceci est du fait que la fonction ϑ est réellement calculée pour le graphe complémentaire. i.e, si on a un graphe G de faible densité alors son graphe complémentaire $\overline{G}$ est de densité élevée.

Discussion :

D'après le tableau 3.1, et en basant sur les colonnes concernant le temps écoulé pour trouver

le nombre chromatique pour chaque instance de graphe, le facteur de choix des sommets non-adjacents à traiter est important, plusieurs critères sont appliqués, et ont donné des résultats différents.

Un autre critère doit être pris en compte, c'est la valeur initiale du nombre de couleurs. Cette valeur peut causer une augmentation considérable du temps de calcul si elle est mal choisie, i.e initialisée à une valeur non proche du nombre chromatique et à la première valeur atteinte dans l'une des feuilles de l'arbre d'exploration (les graphes complets dans ces feuilles). L'initialisation peut être faite par l'un des algorithmes approximatifs les plus rapide tel que l'algorithme glouton ou DSATUR.

En plus, Les traces de calcul pendant les essais montrent que le temps de calcul de thêta (ϑ) est considérablement grand. Malgré que sa complexité est polynomiale, il constitue une part importante dans le temps d'exécution global de l'algorithme..

Conclusion

Dans ce chapitre, nous avons décrit notre approche de coloriage. Cette approche est l'amélioration d'une méthode exacte basée sur le principe *Branch & Bound*. L'étude expérimentale, utilisant des jeu d'essais comportant des graphes réels et aléatoires, a démonté le bon comportement en pratique de notre approche pour les petites et moyennes instances. Concernant les instances de taille importante l'algorithme marqué un accroissement rapide du temps d'exécution. Cependant, l'algorithme s'apprête bien à une parallélisation. Cette idée est le sujet du chapitre suivant.

Chapitre 4

Parallélisation de notre approche

Introduction

En conséquence de résultats obtenus dans la partie d'expérimentation, et afin de pousser notre approche de coloriage pour qu'elle soit capable de traiter des problèmes de grande taille par rapport à l'algorithme (5) précédemment présenté. A fin d'arriver à cette amélioration, on fait appel au *parallélisme*. Cette notion qui devient une tendance dans les applications de calcul à haute performance.

4.1 Le parallélisme

L'avancé technologique, dans tous les domaines de la vie moderne, demande de plus en plus de puissance de calcul. La course à la vitesse des processeurs a atteint ses limites en terme de réchauffement des circuits et de fréquences alors les constructeurs se sont tournés vers la parallélisation des traitements et la création de processeurs multicoeurs et de multiprocesseurs qu'on peut facilement assimiler à des machines parallèles [Lou14].

4.1.1 Motivations et applications du parallélisme

Les vastes augmentations de puissance de calcul qu'ont été appréciées depuis des décennies ont été au cœur des avancées les plus spectaculaires dans des domaines aussi variés que la science, l'Internet, et de divertissement. Par exemple, le décodage du génome humain, l'imagerie médicale plus précise, des recherches rapides et précises sur le Web , et des jeux informatiques toujours plus réalistes, ces traitements auraient été impossible sans ces augmentations.

Cependant, face à ces augmentations de puissance de calcul, il y a aussi une augmentation sérieuse sur le besoin en calcul. Ce qui suit sont quelques exemples [Pac11] :

— La modélisation du climat.

— Repliement des protéines.

— La découverte de médicaments.

— La recherche énergétique.

— L'analyse des données.

4.1.2 Principes des algorithmes parallèles

Un algorithme parallèle est conçu pour s'exécuter sur une machine parallèle a fin de résoudre un problème en améliorant le temps de calcul tout en respectant les contraintes de complexité en temps calcul et mémoire [Lou14].

L'algorithme parallèle nous dicte comment résoudre un problème donné en utilisant plusieurs processeurs. Cependant, la spécification d'un algorithme parallèle implique plus que de spécifier les étapes. Au moins, un algorithme parallèle a la dimension supplémentaire de la *concurrence* et le concepteur de l'algorithme doit spécifier des ensembles de parties qui peuvent être exécutées simultanément. Cela est essentiel pour l'obtention d'un avantage de performance de l'utilisation d'un ordinateur parallèle.

L'une des méthodes de conception en quatre étapes (voir Figure 4.1) a été proposé par *Ian Foster* pour la conception d'algorithmes parallèles [Pac11] :

1. **Partitionnement :** Diviser le calcul à effectuer et les données exploités par le calcul en petites tâches en identifiant les tâches qui peuvent être exécutées en parallèle.
2. **Communication :** Déterminer quel sont les communication entres les tâches identifiées dans l'étape précédente.
3. **Agglomération ou agrégation :** Combiner les tâches et les communications identifiées dans la première étape dans des tâches plus importantes.
4. **Cartographie (Placement) :** Attribuer les tâches composites identifiées dans l'étape précédente pour les processus/threads. Cela devrait être fait de telle sorte que la communication est réduite au minimum, et chaque processus/thread obtient à peu près la même quantité de travail.

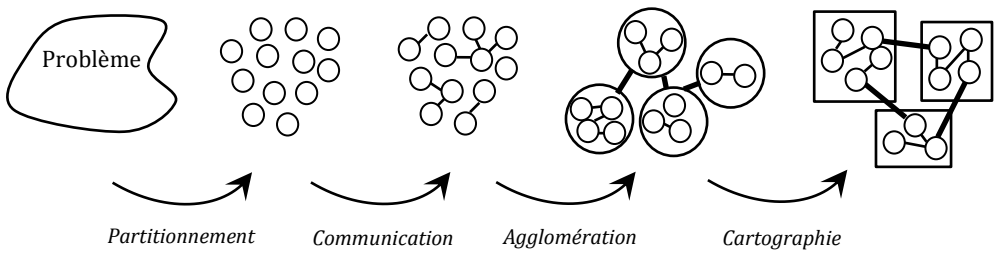


FIGURE 4.1 – Méthode de *Foster* pour la conception d'algorithmes parallèles.

En général, il y a plusieurs choix pour chacune des étapes ci-dessus, mais, relativement peu de combinaisons de choix conduisent à un algorithme parallèle qui donne des performances en rapport avec les ressources de calcul et de stockage utilisées pour résoudre le problème. Souvent, différents choix donnent les meilleures performances sur différentes architectures parallèles ou sous différents paradigmes de programmation parallèle.

4.1.3 Modèles des algorithmes parallèles

Un modèle d'algorithme est typiquement un moyen de structuration d'un algorithme parallèle en sélectionnant une technique de décomposition et à la cartographie et l'application de la stratégie pour minimiser les interactions. Les modèles les plus utilisés sont [KGGK94] :

- *Le modèle de données-parallèles (Data-Parallel Model) :* Dans ce modèle, les tâches sont statiquement ou semi-statique mis en correspondance avec les processus et chaque tâche effectue des opérations similaires sur des données différentes.
- *Le modèle de graphe de tâches (Task Graph Model) :* Dans certains algorithmes parallèles, le graphe tâche-dépendance est explicitement utilisé dans la cartographie. Dans le modèle de graphe de tâches, les relations entre les tâches sont utilisés pour promouvoir la localité ou de réduire les coûts d'interaction.
- *Le modèle de piscine de travail (Work Pool Model) :* Ce modèle se caractérise par une cartographie dynamique des tâches sur les processus avec une équilibrage de charge dans laquelle une tâche peut potentiellement être effectuée par n'import quel processus.
- *Le modèle maître-esclave (Master-Slave Model) :* Un ou plusieurs processus maîtres génèrent le travail et l'affecter à des processus travailleurs.
- *Le modèle producteur-consommateur (Pipeline or Producer-Consumer Model) :* Dans ce modèle, un flux de données est passé par une chaine de processus, chacun d'eux à son action sur ce flux, ce mécanisme est connu sous le nom "parallélisme de flux".
- *Les modèles hybrides (Hybrid Models) :* Un modèle hybrides est composé d'ensemble de modèles appliqués hiérarchiquement ou d'une façon séquentielle dans les différentes phases de l'algorithme.

4.2 Parallélisation de notre approche de coloriage

Notre proposition de parallélisation est plus proche du modèle Maître-Esclave décrit dans la section précédente.

4.2.1 Architecture de l'application

Le schéma général des composantes (voir Figure 4.2) intervenant dans la version parallèle de notre algorithme contient :

— *Un processus/thread organisateur (processus maître) :* Le rôle de ce processus est l'organisation du travail, et la surveillance de l'état global du programme. Ce processus maintient une file d'attente pour garder une liste des problème en attente d'affectation à un processus libéré.
— *Des processus/threads travailleurs (processus esclaves) :* Ces processus servent à effectuer le calcul réel (les opérations de séparation, d'évaluation,et le calcul de thêta ϑ).

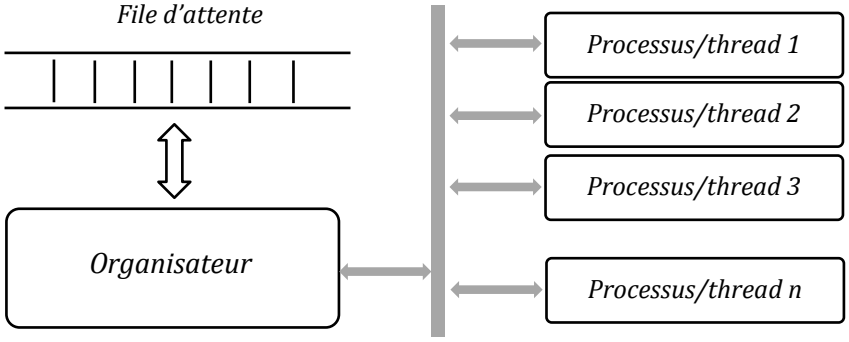


FIGURE 4.2 – Schéma des composants nécessaires pour la version parallèle proposée.

4.2.2 Stratégie d'exploration

L'algorithme parallèle se base sur la version séquentielle avec des modifications pour qu'il soit implémenté avec n'import quel modèle de programmation (mémoire partagée ou distribuée), ce fait est interprété par l'utilisation des primitives (routines) génériques qui ne sont pas liées à un modèle précis.

Le fonctionnement général est comme suit :

— La routine principale d'organisateur commence par la préparation de l'environnement : création de la file d'attente, initialisation des variables et la création du problème racine.
— Ensuite, le problème racine est envoyé à l'un des n processus/threads travailleurs.
— Chaque fois un processus travailleur reçoit un problème à traiter, il applique l'algorithme (6) :
  — Avant le traitement de tout problème, le processus doit vérifier s'il y a une modification sur la meilleur valeur jusqu'a maintenant (si la variable est partagée la valeur est disponible si non vérifier si y a un message de mise à jours).
  — L'étape de l'évaluation est fait comme celui de la version séquentielle, sauf s'il y a une nouvelle meilleur valeur, elle sera diffusée aux autres processus selon le modèle de programmation.
  — Si une séparation est nécessaire, alors l'un des deux problèmes fils est empilé dans la pile locale du processus. et afin de diviser le travail, l'autre problème est envoyé

à la file d'attente par l'intermédiaire du processus organisateur, si la file est pleine, le processus doit aussi empiler le deuxième problème pour le traiter plus tard de la même manière que pour la version séquentielle.

— Le processus travailleur doit déclarer son état s'il n'a pas des problèmes à traiter (la pile est vide).

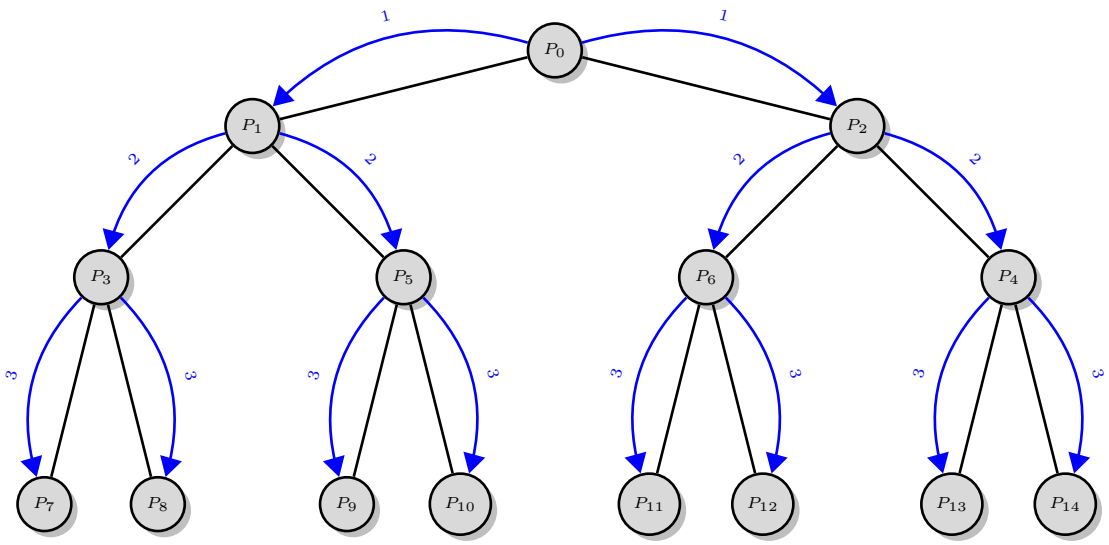


FIGURE 4.3 – Stratégie d'exploration de l'arbre des solutions (Algorithme parallèle).

La figure 4.3 illustre les problèmes qui peuvent être traité en parallèle. Les étapes parallèles sont étiquetées par les numéros, mais ça ne signifie pas forcement qu'ils seront traités exactement avec cet ordre.

Algorithme 6 : Algorithme exécuté par un processus travailleur

```
Données : G(V, E)
P : Pile ;   MeilleurChrom : Entier ;   C : Couple de sommets
début
    Recevoir(G)
    Empiler(G, P)
    tant que NonVide(P) faire
        Vérifier(MeilleurChrom)
        G ⟵ Depiler(P)
        si EstComplet(G) alors
            si |V| < MeilleurChrom alors
                MeilleurChrom ⟵ |V|
                Diffuser(MeilleurChrom)
        sinon
            si ⌈θ(G)⌉ < MeilleurChrom) alors
                C ⟵ (v_i, v_j) tel que v_i, v_j ne sont pas adjacents.
                G' ⟵ FusionnerSommets(G, C)
                G'' ⟵ LierSommets(G, C)
                si la file n'est pas plein alors alors
                    Envoyer(G'')
                sinon
                    Empiler(G'', P)
                Empiler(G', P)
    Envoyer(idle)
```

4.2.3 Analyse de l'algorithme parallèle

L'analyse de l'algorithme proposé se base sur plusieurs critères.

Degré de parallélisme

l'algorithme utilise un problème entier comme unité de donnée traitée dans un processus/thread à part. Donc on parle d'un parallélisme de gros grain (haut niveau). Le nombre des processus travailleurs est un paramètre qui peut être changé pour évaluer quelques métriques liées aux algorithmes parallèles (Travail, Coût, Accélération). Cependant, le nombre des processus est pratiquement limité par les ressources matérielle ou logicielle. Cependant,il reste toujours plus petit par rapport au nombre des problèmes générés qui est théoriquement très grand (la classe du problème de coloriage).

Équilibrage de charge

L'utilisation d'un processus organisateur et un file d'attente principale a pour but de diviser le travail d'un façon équitable entre les processus. Chacun d'eux dès qu'il envoie l'état $Idle$, l'organisateur lui envoi un nouveau problème à traiter, ce qui garantie une utilisation presque parfaite des processus.

La communication

Au début, chaque processus doit diviser sa tâche, lors chaque séparation, en envoyant l'un des deux problèmes à l'organisateur. Ce mécanisme augmente la quantité des données échangé, Mais le temps on peut arriver à un état où tout les processus sont occupés et la file est pleine, dans ce cas, tout les problème fils crées doivent être empiler localement. et ça va diminuer considérablement l'échange.

La terminaison

Comme l'exécution du travail est distribué, il est nécessaire de déterminer l'état globale de cette exécution, i.e si tout les membres (processus/thread) ont terminé leurs têches. A fin de détecter cet état l'organisateur teste périodiquement si la file est vide, et tous les processus sont dans l'état $Idle$.

Une dernière remarque concernant la gestion de la file d'attente. Différents modes de gestion tel que l'utilisation des priorités basées sur la taille du problème et le nombre théta du parent, peuvent être utilisés comme critère d'extraction de problèmes, En effet ceci peut affecter la stratégie globale de parcours, et par conséquence le temps global de l'exécution.

Conclusion

Cette proposition de parallélisation nécessite une implémentation sur une machine parallèle afin de mesurer son comportement en pratique. Vu le temps imparti au travail, on a laissé cette idée comme perspective.

Conclusion & perspectives

Le coloriage de graphes est l'un des problèmes très étudiés et utilisés dans la pratique, Cependant, il est difficile lorsque l'on cible une solution exacte. Nous avons ciblé l'amélioration d'une approche de coloriage de graphes basée sur la méthode *"Séparation et Évaluation"*. Nous pouvons résumer nos contributions dans les points suivants :

— Implémentation de la version séquentielle améliorée de l'algorithme.

— L'évaluation de la performance réelle de notre approche et l'estimation de ses limites.

— Proposition d'une parallélisation qui peut être une base des implémentations sur plusieurs modèles et plateformes parallèles.

Notre travail nous a donné un aperçu sur les grandes efforts faits dans ce domaine. En plus une expérience a été acquise sur les difficultés pratiques pendant les tests, ou en général lors du travail sur des problèmes similaires d'optimisation combinatoire.

Cet effort est aussi considéré comme une initiation à la recherche et nous a permet de découvrir l'un des pilier de l'informatique théorique, le domaine riche en problèmes difficile ou non résolus qui peuvent être un sujet des futures projets de recherche.

Perspectives

Concernant notre approche sur le problème de coloriage, plusieurs idées perspectives peuvent être proposées, On cite les plus importantes parmi elles :

— Minimisation de la complexité spatiale par le choix des structures et des stratégies de dérivation des problèmes (stratégie de sauvegarde).

— Recherche sur des méthodes heuristiques dans l'étape de sélection des sommets. i.e trouver des bons critères ou un mécanisme permettant la connaissance de la bonne séquence des choix.

— Investigation sur des valeurs initiales plus proches du nombre chromatique et de coût

faible, en utilisant les algorithmes approximatifs connus dans le domaine.

— Amélioration du calcul du nombre thêta ou l'utilisation d'une autre borne moins coûteuse dans l'étape de l'évaluation.

— Implémentation de l'algorithme parallèle sur plusieurs modèles (Mémoire partagés, mémoire distribuée).

Bibliographie

[Bol98] Bela Bollobas. *Modern Graph Theory*. Springer, 1998.

[BR12] R. Balakrishnan and K. Ranganathan. *A Textbook of Graph Theory*. Universitext (Berlin. Print). Springer New York, 2012.

[Bro72] J. Randall Brown. Chromatic scheduling and the chromatic number problem. *Management Science*, 19(4-part-1) :456–463, 1972.

[CF07] H. Cherroun and P. Feautrier. An exact resource constrained-scheduler using graph coloring technique. In *2007 IEEE/ACS International Conference on Computer Systems and Applications*, pages 554–561, May 2007.

[CFA04] Jean Cardinal, Samuel Fiorini, and Gilles Van Assche. A graph coloring problem with applications to data compression, 2004.

[Cou97] Olivier Coudert. Exact coloring of real-life graphs is easy. In *Proceedings of the 34th Annual Design Automation Conference*, DAC '97, pages 121–126, New York, NY, USA, 1997. ACM.

[GJS76] M. Garey, D. Johnson, and Hing So. An application of graph coloring to printed circuit testing. *IEEE Transactions on Circuits and Systems*, 23(10) :591–599, Oct 1976.

[Glo86] Fred Glover. Future paths for integer programming and links to artificial intelligence. *Comput. Oper. Res.*, 13(5) :533–549, May 1986.

[GLS81] Martin Grötschel, László Lovász, and Alexander Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1(2) :169–197, 1981.

[GLS88] Martin Grötschel, László Lovász, and Alexander Schrijver. *Geometric Algorithms and Combinatorial Optimization*, volume 2 of *Algorithms and Combinatorics*. Springer, 1988.

[HRVW96] Christoph Helmberg, Franz Rendl, Robert J. Vanderbei, and Henry Wolkowicz. An interior-point method for semidefinite programming. *SIAM Journal on Optimization*, 6(2) :342–361, 1996.

[KGGK94] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing*. Benjamin/Cummings, 1994.

[Knu94] Donald E. Knuth. The sandwich theorem. *Electronic. J. Combinatorics.*, 1 :Article 1, approx. 48 pp. (electronic), 1994.

[Kub04] M. Kubale. *Graph Colorings.* Contemporary mathematics (American Mathematical Society) v. 352. American Mathematical Society, 2004.

[KV+83] Scott Kirkpatrick, Mario P Vecchi, et al. Optimization by simmulated annealing. *science*, 220(4598) :671–680, 1983.

[Lew15] R.M. R. Lewis. *A Guide to Graph Colouring : Algorithms and Applications.* Springer Publishing Company, Incorporated, 1st edition, 2015.

[Lou14] Rédha Loucif. *Parallélisation d'Algorithmes d'Optimisation Combinatoire.* Mémoire de magistère, Université colonel Hadj Lakhdar – Batna, 2014.

[Lov79] L. Lovasz. On the shannon capacity of a graph. *IEEE Transactions on Information Theory*, 25(1) :1–7, Jan 1979.

[LV02] L. Lovász and K. Vesztergombi. Geometric representations of graphs. pages 471–498. Bolyai Soc. Math. Stud.,, 2002.

[Neh09] Attia Nehar. *Algorithmes exactes de coloriage de graphes pour l'ordonnancement.* Mémoire de magistère, Université Amar Telidji - Laghouat, 2009.

[Pac11] P.S. Pacheco. *An Introduction to Parallel Programming.* An Introduction to Parallel Programming. Morgan Kaufmann, 2011.

[WP67] D. J. A. Welsh and M. B. Powell. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 10(1) :85–86, 1967.

[Zha96] W. Zhang. Branch-and-bound search algorithms and their computational complexity. Technical Report ISI/RR-96-443, University of southern California/ Information Sciences Institute., May 1996.