Coloration de graphes

Projet de Recherche Documentaire

Encadrant : M. Julien Bernard

Adrien Avenia Claire Marche

Table des matières

Introduction

Ce projet de recherche documentaire a été réalisé dans le cadre de l'unité d'enseignement du même nom, et a pour thème la coloration de graphes.

Plus particulièrement, on s'attachera ici à l'optimisation d'algorithmes de coloration de graphes.

Pour cela, nous allons tout d'abord dans un chapitre introductif présenter les notions principales sur les graphes qui seront abordées tout au long de ce rapport. Ensuite, nous aborderons le thème plus particulier de la coloration de graphes, au travers de définitions et de descriptions des avancées. Enfin, nous répertorierons les algorithmes et heuristiques les plus efficaces pour la coloration de graphes, avant de conclure par un bilan général.

Chapitre 1

Les graphes

Définition 1. *Graphe Un graphe est un ensemble de nœuds reliés par des arêtes ou des arcs.*

On utilise souvent des graphes pour modéliser des réseaux. Cela assure, par la nombre de types de réseaux existants, une grande variété au niveau des graphes. Ainsi, donner une définition plus précise et incluant toujours la totalité des graphes serait impossible.

1.1 Vocabulaire utile

Adjacence L'adjacence est la propriété de deux sommets d'être connectés par la même arête ou de deux arêtes de présenter une extrémité commune.

Arête Une arête est la connexion entre sommets. Dans un graphe orienté, on parle d'arc, le terme arête servant alors à désigner les deux arcs reliant deux sommets dans un sens puis dans l'autre.

Boucle Une boucle est une arête partant d'un sommet et arrivant sur lui-même.

Chemin Un chemin est une suite consécutive d'arcs dans un graphe orienté. Cette même structure dans un graphe non orienté s'appelle une chaîne.

Degré Le degré $d(x)$ d'un sommet x est, dans le cas d'un graphe non orienté, le nombre d'arêtes reliées à x. Dans le cas d'un graphe orienté, il existe $d^-(x)$ le degré entrant de x comptant le nombre d'arcs dans sa direction, et $d^+(x)$ le degré sortant de x comptant le nombre d'arcs qui en sortent.

Indépendance Deux sommets sont indépendants s'ils ne sont pas adjacents.

Line Graph Le line graph d'un graphe G est le graphe dans lequel deux sommets adjacents dans le line graph correspondent à deux arêtes incidentes à un même sommet dans G et deux arêtes incidentes dans le line graph correspondes à deux sommets adjacents dans G.

Ordre L'ordre d'un graphe est son nombre de sommets.

Sommet Les sommets d'un graphes sont les éléments du graphes qui sont reliés entre eux par des arêtes. Ils sont aussi appelés nœuds.

Sous-ensemble indépendant de sommets Un sous-ensemble indépendant de sommets est un ensemble de sommets ayant pour propriété d'être deux à deux indépendants.

1.2 Différents types de graphes

Graphes valués Les différences peuvent d'abord porter sur les informations apportées par le graphes. En effet, dans un graphe valué, on donne une valeur aux éléments du graphes (on peut valuer indifféremment les nœuds ou les arêtes ou arc, bien que le second soit plus courant), afin de leur donner un poids.

Graphes orientés On peut également définir le type de graphe par la navigabilité de ses arêtes ou arcs. Si ils sont tous navigables indifféremment dans les deux sens (c'est à dire que pour tout (A, B) du graphe, on peut aussi bien naviquer de A à B que de B à A), le graphe et dit non orienté et on parle d'arêtes. Sinon, s'il existe des arêtes du graph non navigables dans les deux sens (c'est à dire que pour (A, B), on pourrait naviguer de A à B mais pas de B à A ou l'inverse), le graphe est dit orienté et on parle d'arcs.

Graphes simple Un graphe simple est tout simplement un graphe dans lequel aucune boucle (arête qui rejoint le même sommet) ni aucunes arêtes parallèles (reliant les deux mêmes sommets) ne sont trouvables.

Graphes connectés On dit d'un graphe qu'il est connecté si on peut trouver un chemin dans ce graphe entre chacune de ses paires de sommets. Au contraire, s'il y a au moins une paire qu'il est impossible de relier, on dit de ce graphe qu'il est déconnecté.

Graphes réguliers Un graphe est dit régulier si le degré de tous ses sommets est le même. Si ce degré est k, on dit alors que le graphe est k-régulier.

Graphes complets Un graphe complet est un graphe dans lequel tous les sommets sont reliés à tous les autres sommets, c'est à dire un graphe dans lequel le degré de tous les sommets est le degré maximal du graphe.

Graphes cycles Un graphe à n sommets ($n \geq 3$) qui forme un cycle avec l'entièreté de ses sommets est un graphe cycles.

Graphes roues On peut former un graphe roue à partir d'un graphe cycle en lui ajoutant un sommet, appelé le « hub » relié à tous les autres sommets du graphe.

Graphes cycliques Un graphe qui contient un cycle en son intérieur est un graphe cyclique. Au contraire, un graphe sans aucun cycle est appelé un graphe acyclique.

Graphes bipartis Un graphe $G = (V, E)$ comprenant une partition de sommets $\{V_1, V_2\}$ est dit biparti si toutes les arêtes du graphes relient un sommet de V_1 à un sommet de V_2.

Graphes complets bipartis Un graphe complet biparti est un graphe biparti dans lequel tous les sommets de chaque partition sont reliés à tous les sommets de l'autre partition.

Graphes étoilés Un graphe complet biparti pour lequel une partition ne contient qu'un sommet et l'autre contient tous les autres est un graphe étoilé.

Complément d'un graphe Le complément d'un graphe G est un graphe $\bar{G}$ tel que, si deux sommets soient adjacents dans G, ils ne le soient pas dans $\bar{G}$ et si deux sommets ne soient pas adjacents dans G, alors ils le soient dans $\bar{G}$.

Cographes Un cographe est un graphe simple construit récursivement, sachant que :
- le graphe à un sommet est un cographe,
- le complémentaire d'un cographe est un cographe,
- l'union de deux cographes est un cographe.

1.3 Coloration de graphe

La coloration de graphes est un des domaines les plus importants de la théorie des graphes. Cela comprend notamment le fameux « Théorème des quatre couleurs », qui dit que tout graphe planaire peut être coloré avec uniquement quatre couleurs sans que ses éléments adjacents ne soient de la même couleurs. Ce type de coloration avec les éléments adjacents de couleur différente est la coloration la plus souvent utilisée et celle pour laquelle des algorithmes sont les plus recherchés. Chacune de ces colorations peut être appliquée à différents éléments d'un graphe.

1.3.1 La coloration la plus basique

Le type de coloration le plus basique est également le plus connu du grand public. En effet, c'est simplement l'idée classique qui consiste à colorer les éléments adjacents de couleurs différentes. C'est ce type de coloration qui a donné lieu, sur les graphes planaires, à l'énonciation du théorème des quatre couleurs, mais on peut également l'appliquer à un groupe plus étendu de graphes. On peut

voir un exemple de cette coloration dans la figure 1.1. Elle existe évidemment également pour les arêtes, avec le même type de spécifications.

Pour chaque graphe, il existe un nombre minimal de couleurs permettant la coloration du graphe selon ces spécifications. Ce nombre est appelé nombre chromatique $\chi(G)$. Il permet de trouver de manière triviale la coloration optimale du graphe ; le calculer le plus rapidement possible est donc un enjeu majeur.

Exemple d'utilisation

Les antennes téléphoniques émettent sur une certaine fréquence. Or deux antennes téléphoniques trop proches émettant sur une même fréquence créent des interférences entre elles. Il est donc nécessaire que, quand deux antennes sont trop proches, elles émettent sur des fréquences différentes. Les fréquences étant une ressource payante, il peut être judicieux pour un opérateur de trouver la coloration optimale du graphe représentant les antennes téléphoniques (avec les arêtes reliant les antennes trop proches) afin d'utiliser le moins de fréquences possibles.

1.3.2 Coloration équitable

Une coloration équitable suit les même spécifications que la coloration simple, mais y ajoute une contrainte supplémentaire : pour toute paire de couleurs, le nombre d'éléments qui portent une couleur doit différer de l'autre d'au maximum 1. Un exemple de coloration équitable peut être observé à la figure 1.2 Il a été prouvé par Hajnal et Szeremedi [1] qu'il existe une k-coloration équitable d'un graphe si $k \geq \Delta(G) + 1$.

Pour cette coloration spécifique, il existe également un nombre minimal de couleurs à utiliser pour obtenir la coloration optimale du graphe, appelé nombre chromatique équitable $\chi_{=}(G)$.

Exemples d'utilisation

Il est parfois nécessaire d'établir un emploi du temps optimal, contenant des tâches impossible à réaliser simultanément. Dans ce cas, modéliser ces tâches en tant que sommets d'un graphes, chaque sommet étant reliés à tous les sommets représentant les tâches qu'il est impossible d'effectuer en un même temps, puis en réalisant une coloration équitable optimale de ce graphe, on affecte les tâches de la manière la plus effective possible.

1.3.3 Coloration circulaire

Une k-coloration circulaire consiste en l'association de nombre fractionnaires au sommets du graphe, tel que les valeurs des sommets adjacents soient distant

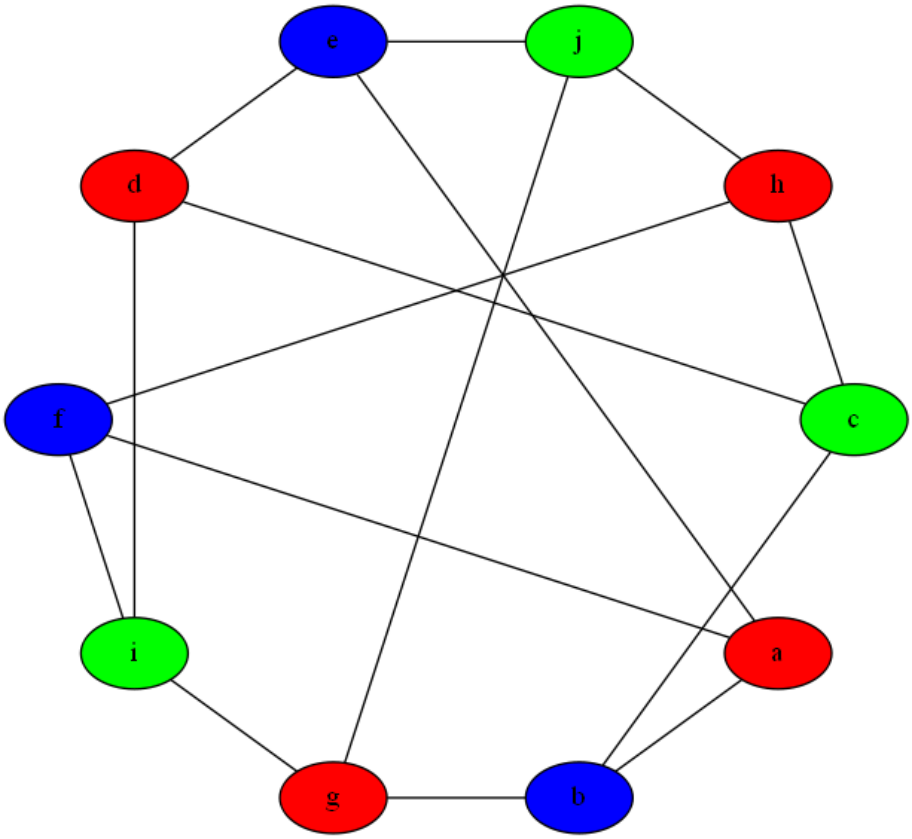

FIGURE 1.1 – Exemple de coloration simple de graphes

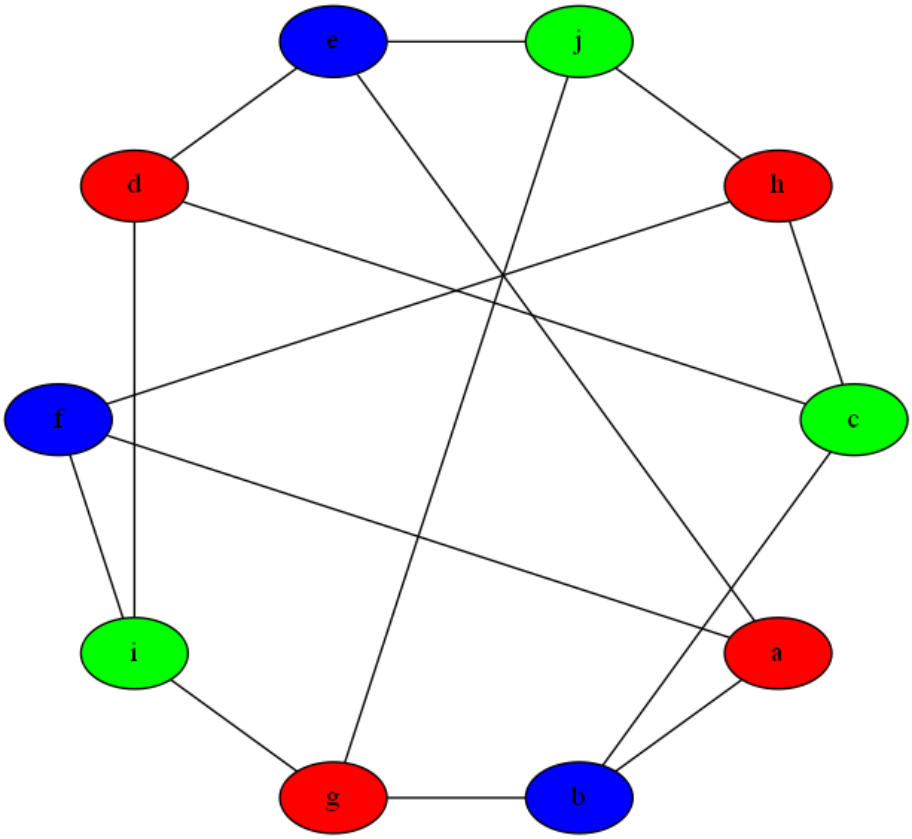

FIGURE 1.2 – Exemple de coloration équitable de graphes

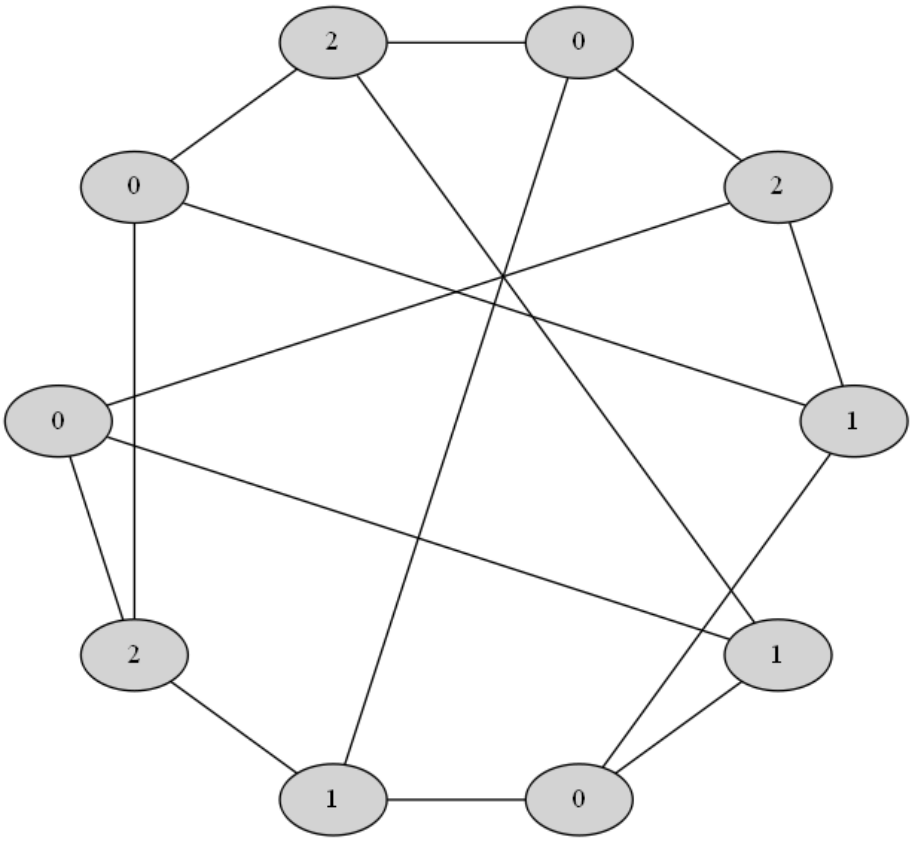

FIGURE 1.3 – Exemple de coloration circulaire de graphes

au minimum d'une valeur dixe (par exemple 1) et au maximum de la valeur $k-1$. On peut observer à la figure 1.3 une 3-coloration circulaire d'un graphe.

Il existe un nombre minimal de valuation des sommets, appelé nombre chromatique circulaire $\chi_C(G)$.

Exemple d'utilisation

Dans un atelier où les ressources sont renouvelables et les tâches à effectuer indépendantes les unes des autres, la réalisation d'une coloration circulaire sur une modélisation de l'atelier par un graphe permet l'affectation optimale des ressources aux différentes tâches.

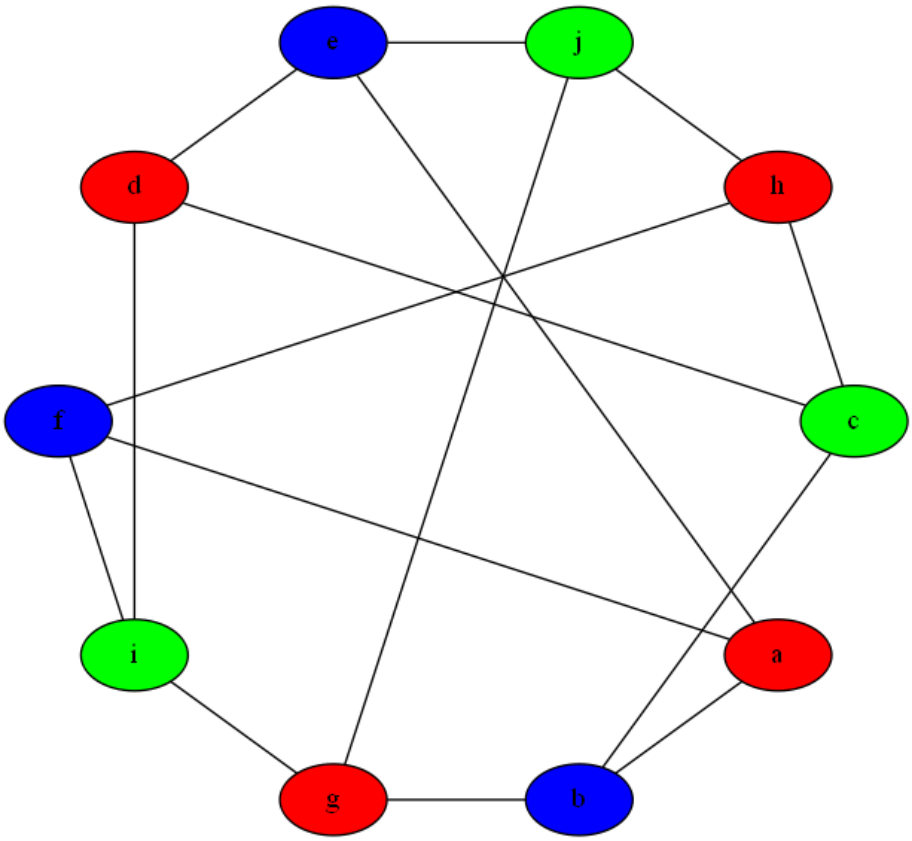

FIGURE 1.4 – Exemple de coloration acyclique de graphes

1.3.4 Coloration acyclique

La coloration acyclique suit les même spécifications que la coloration simple de graphes, mais y ajoute en plus la nécessité de non existence d'un cycle de deux couleurs (c'est-à-dire sans alternance). On peut observer en figure 1.4 une coloration acyclique. Cette coloration existe avec les mêmes spécifications pour les arêtes d'un graphe.

Il existe un nombre minimal de couleurs pour une coloration acyclique optimale, appelé nombre chromatique acyclique $A(G)$. Il est prouvé que, pour $\Delta(G) \to \infty$, on a $A(G) = O(\Delta(G)^{\frac{4}{3}})$.

Pour des graphes de degrés inférieur ou égal à trois, il existe un algorithme linéaire de coloration acyclique du graphe, rédigé par Skulrattanakulchai.

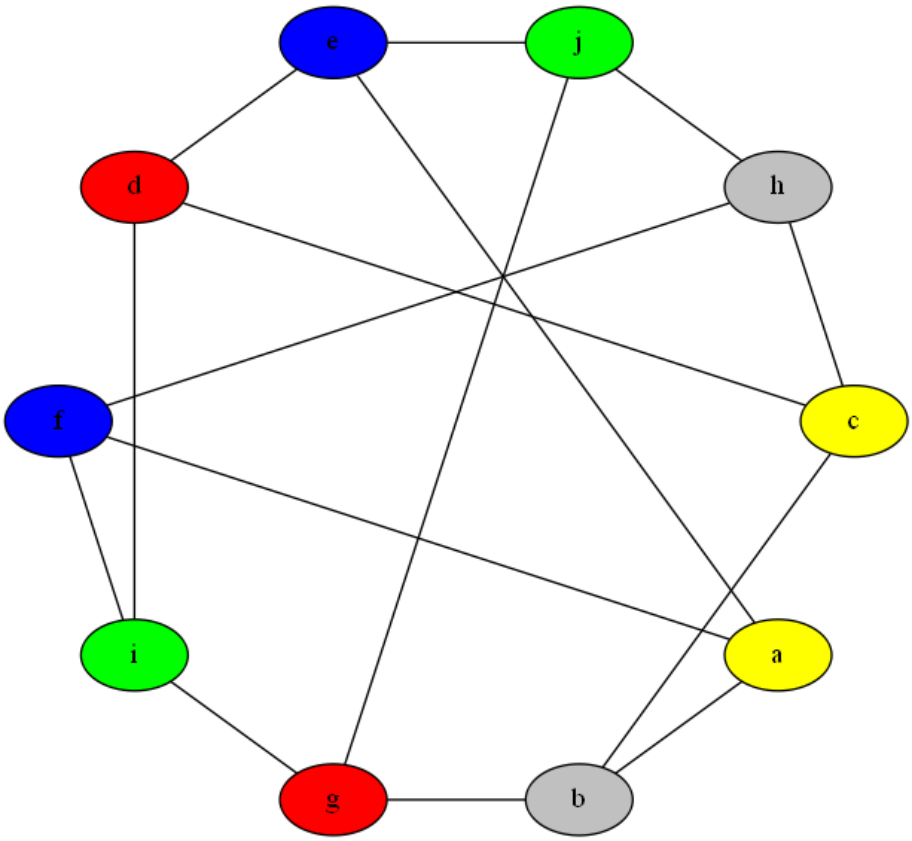

FIGURE 1.5 – Exemple de coloration étoilée de graphes

1.3.5 Coloration étoilée

Une coloration étoilée d'un graphe est une coloration simple de ce même graphe pour laquelle tout chemin de taille quatre dans ce graphe contient au moins trois couleurs différentes. La figure 1.5 montre une coloration étoilée d'un graphe.

Il existe un nombre minimal de couleurs permettant d'obtenir une coloration étoilée d'un graphe, appelé nombre chromatique étoilé $\chi_s(G)$.

Il a été prouvé par Lyons que toute coloration acyclique d'un cographe est aussi une coloration étoilée.

1.3.6 Coloration par liste

La coloration par liste est une coloration telle que chaque sommet à colorer possède une liste (qui peut être identique ou non à celle d'un autre sommet)

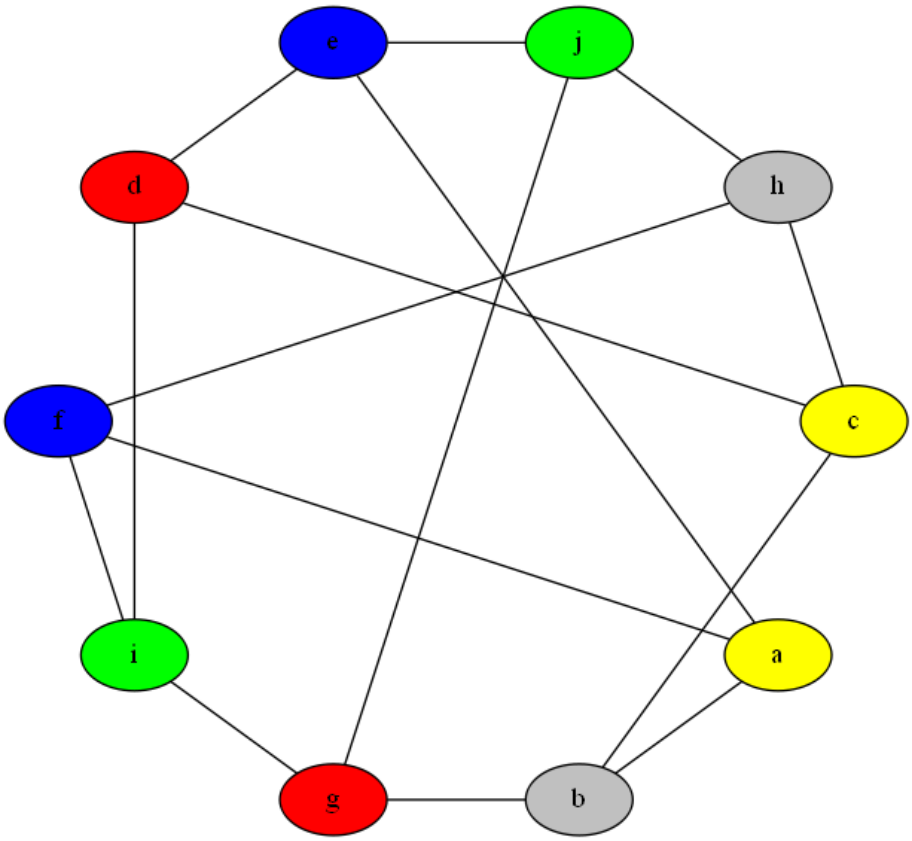

FIGURE 1.6 – Exemple de coloration par liste de graphes

de couleurs qu'on peut lui appliquer. Sinon, en respectant les listes, on réalise simplement une coloration simple du graphe. On peut voir à la figure 1.6 un exemple de coloration par liste.

Exemples d'utilisation

Si certains individus ne sont qualifiées que pour certaines tâches, on préfèrera leur confier celles-ci plutôt que d'autres. Ainsi, pour réaliser un emploi du temps optimal pour ces personnes, on se met dans le même cas que pour l'utilisation de coloration simple, mais on associe en plus à chaque sommet qui représente un individu une liste de couleurs possibles, chaque couleur représentant une tâche pour laquelle il est qualifié.

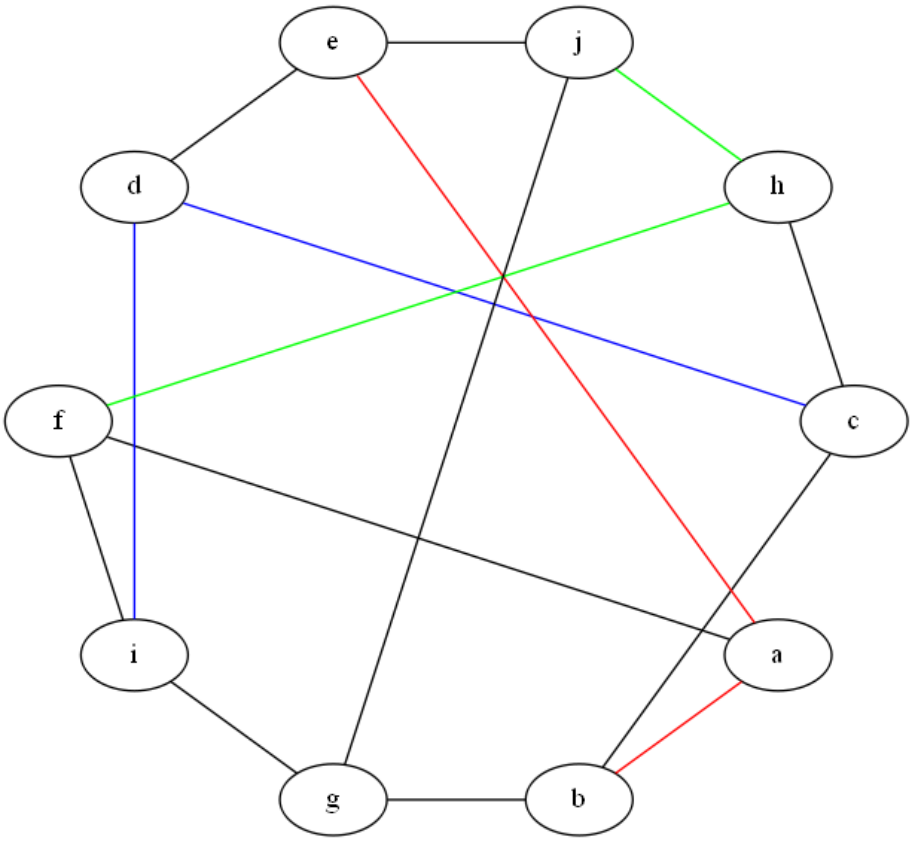

FIGURE 1.7 – Exemple de coloration de chemins

1.3.7 Coloration de chemins

La coloration de chemins est un type spécifique de colorations d'arêtes, qui consiste à colorier les liens entre certaines paires de sommets, telle qu'aucun sommet n'appartienne à plusieurs chemins à la fois. La figure 1.7 est un exemple de coloration de chemins.

Exemples d'utilisation

Lorsqu'on souhaite faire passer des messages entre différents nœuds d'un réseau, on ne souhaite pas que les nœuds intermédiaires ne traitent plusieurs messages à la fois. Dans ce cas exact, une coloration de chemin permet de trouver le chemin optimal en respectant ces restrictions.

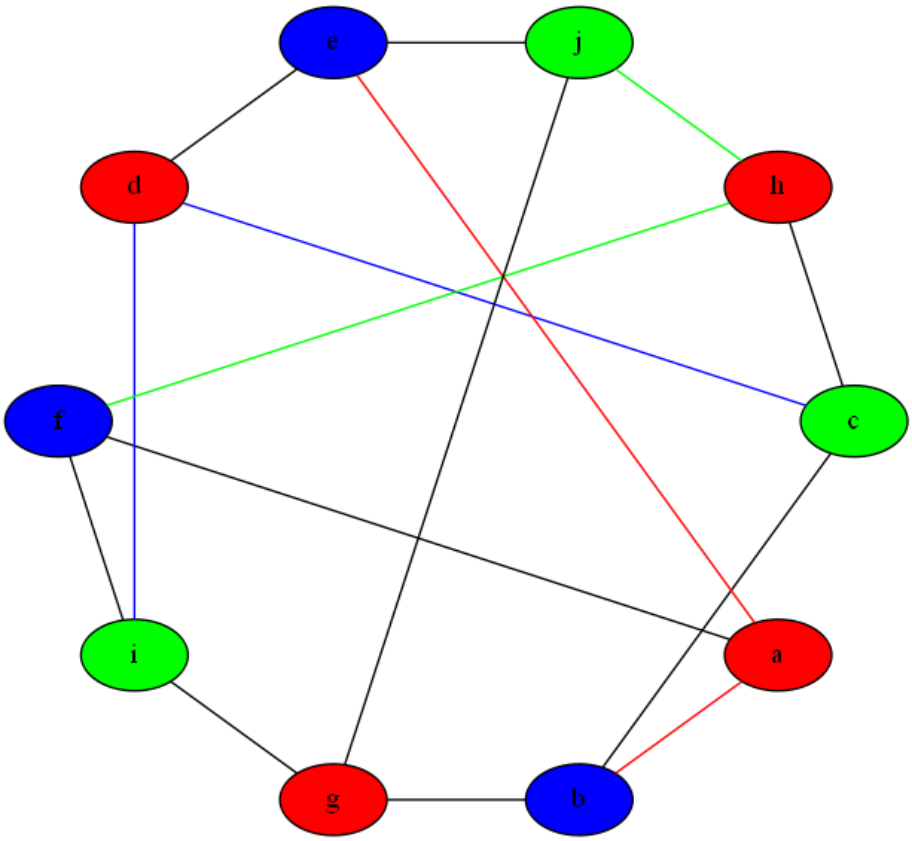

FIGURE 1.8 – Exemple de coloration totale de graphes

1.3.8 Un cas particulier de coloration : la coloration totale

Dans les cas précédents, on se concentrait uniquement sur la coloration d'un type d'élément du graphe , que ce soit les sommets ou les arêtes. Une coloration totale d'un graphe colore à la fois les arêtes et les sommets, avec sur chacun la même restriction que pour une coloration simple, à laquelle on ajoute que chaque paire incidente (sommet, arête) ne doit pas être de la même couleur. Cette coloration est montrée à la figure 1.8.

Tout comme pour les autres colorations, il existe un nombre minimal de couleurs permettant de réaliser une coloration optimale, appelé nombre chromatique total $\chi''(G)$.

Il a été conjecturé par Behzad et Vizing que $\chi''(G) \leq \Delta(G) + 2$. Si cette conjecture est vérifiée, il serait possible de diviser les graphes en deux classes selon leurs colorations totales.

1.4 Notations

Dans la suite de ce rapport, nous nous référerons à un graphe $G = (V, E)$ d'ordre n, où V correspond à l'ensemble de ses sommets, et E à l'ensemble de ses arrêtes. Nous désignerons par $\chi_s(G)$ le nombre chromatique du graphe G. Le *degré* d'un sommet w de G sera noté $d(w)$, son *degré de saturation* $d_s(w)$. Soit S un sous-ensemble indépendant de sommets, nous noterons S_i la classe de couleur i.

Chapitre 2

Des algorithmes de coloration de graphes

La coloration de graphes a toujours suscité un vif entrain auprès de la communauté scientifique, de par sa complexité, et ses nombreuses applications possibles ayant été énoncé dans le chapitre précédent.

C'est ainsi que de nombreux algorithmes ont fait leur apparition, s'inspirant dans la plupart des cas de leurs prédécesseurs, en tentant toujours d'améliorer les nombreuses faiblesses récurrentes, et en mettant en œuvre de nouveaux concepts.

Il est l'un des problèmes NP-complet les plus étudiés, et la recherche reste toujours très active, fournissant des résultats en constante amélioration.

2.1 La difficulté de ces algorithmes

La considération de ce problème en tant que problème NP-complet d'optimisation combinatoire, implique une difficulté de résolution en temps exponentiel, et la nécessité de se tourner vers des algorithmes d'approximation ou l'utilisation d'heuristiques, voir de méta-heuristiques. C'est ainsi qu'aucun algorithme de résolution en temps polynomial n'existe à ce jour, et que la recherche de nouvelles stratégies et le développement d'algorithmes les utilisant se trouve toujours être actifs.

Chaque algorithme possède de nombreuses faiblesses (temps d'exécution instable sur de grandes instances de graphe, mémoire à long terme beaucoup trop consommatrice *etc.*), conséquences fréquentes d'imperfections inhérentes à la stratégie de résolution appliquée (répétition de mêmes opérations, stockages en mémoire de données superflues, persistance sur de mauvaises colorations *etc.*). Des ajustements (paramétrage de constantes) doivent souvent être réalisées, notamment en fonction de la taille de l'instance ; ils permettent soit d'améliorer les performances d'exécution, soit de modifier la solution donnée par l'algorithme (lorsque ces ajustements concernent pour certains, la modification de l'initialisation d'une coloration arbitraire par exemple).

Une part d'aléatoire (ou de détermination arbitraire de certaines données) se retrouve dans de nombreux algorithmes (comme les algorithmes génétiques), renforçant l'instabilité de la résolution.

Ainsi, la réutilisation de différents concepts, leur mélange (donnant lieu à des algorithmes hybrides), et la combinaison avec de nouveaux ajouts sont des comportements fréquemment rencontrés dans les algorithmes modernes. Néanmoins, ceci ne suffit pas encore à satisfaire une résolution optimale. L'hybridation permet l'amélioration de certaines contraintes, mais en créé généralement d'autres, que l'algorithme se charge de masquer à travers de nouveaux paramétrages.

2.2 Présentation d'algorithmes

2.2.1 Recursive Largest First

L'une des méthodes de coloration les plus simples consiste à procéder *séquentiellement* . Elle consiste à définir une relation d'ordre pour les nœuds du graphe, tel que $O = [a_0, ..., a_{n-1}]$. À a_0 est assigné la classe de couleur 1, puis jusqu'à a_{n-1}, la classe de couleur assignée au nœuds a_i se retrouve être la plus faible n'étant pas déjà assignée à un nœud voisin à a_i. Au final, bien que l'on se retrouve dans une situation d'*optimum local*, le résultat est hautement influencé par la relation d'ordre initiale : certains ordonnancements mèneront à une coloration plus pauvre que d'autres. Un algorithme trivial utilisant cette méthode sur un ordre aléatoire est l'algorithme *Randomly Ordered Sequential* (RND)[2].

Des variantes existent, modifiant simplement la relation d'ordre.

L'algorithme *Largest First* (LF) se base sur un ordonnancement par degrés décroissants, et l'algorithme *Smallest Last* (SL) sur un ordonnancement par degrés décroissants, mais en déterminant récursivement les nœuds de degré minimal (a_i correspond ainsi au nœud de degré minimal dans le sous-graphe $U = V - \{v_n, ..., v_{i+1}\}$).

L'ordonnancement peut aussi se faire selon le degrés de saturation d'un nœud, comme dans l'algorithme *Degree of Saturation* (DSATUR) [3], où le premier nœud correspond à celui de plus haut degré, et les suivants sont déterminés par degrés de saturation décroissants.

Ces différents algorithmes se retrouvent avoir une complexité relativement faible ($\theta(n^2)$ pour l'ordre relatif aux degrés, $\theta(n^3)$ pour l'ordre relatif aux degrés de saturation), mais ne garantissent pas de trouver $\chi_s(G)$.

L'algorithme *Backtracking Sequential Coloring* (BSC) retourne lui la valeur exact de $\chi_s(G)$, au détriment d'un mécanisme plus complexe : l'ordre, initialement déterminé par degrés décroissants, se retrouve être dynamiquement modifié.

2.2.2 Tabucol

La recherche locale est une autre méthode générale, permettant d'obtenir d'une coloration initiale - légale ou non -, une solution dite voisin : on procède

par itérations successives afin de progressivement atteindre une coloration de qualité, même si il n'y a aucune garantie quant à l'obtention dans la pratique d'une solution optimale.

L'un des premiers algorithmes de recherche local, et probablement le plus populaire, est *Tabucol* [4]. Malgré son ancienneté (il a été développé en 1986), son mécanisme est toujours utilisé dans des algorithmes hybrides.

Se basant sur une coloration générée aléatoire (et non forcément légale), l'algorithme se charge de retourner, pour un nombre k de couleurs donné, une coloration légale dans la mesure du possible. Ainsi, une stratégie d'utilisation afin de trouver une approximation de $\chi_s(G)$ peut être d'employer une heuristique quelconque afin d'obtenir une première coloration légale, d'en extraire son nombre k de couleurs, et d'appeler de manière itérative l'algorithme *Tabucol* en décrémentant k, jusqu'à ce que celui-ci retourne une coloration illégale.

La spécificité de cet algorithme de recherche locale consiste en l'utilisation d'une *liste tabou* (représentée par une structure de données adéquate), interdisant le retour sur une solution récemment explorée, afin d'éviter le problème d'optimal local.

En effet, à chaque itération, l'algorithme modifie la couleur d'un sommet en conflit avec un autre, ceci afin de progressivement diminuer le nombre de conflits. Cette modification est ensuite ajoutée pour un certain nombre d'itérations dans la *liste tabou*.

Une difficulté réside dans les paramètres à fixer pour calculer le nombre dit d'itérations pendant lesquels la modification est considérée *tabou*. Ces paramètres s'avèrent devoir être adaptés pour chaque instance de graphe.

Aussi, le nombre maximal d'itérations doit être adéquatement défini ; n'étant pas possible de savoir avec certitude si une coloration légale pour k couleurs fixées existe, il est nécessaire de stopper l'algorithme quand on juge que celui-ci ne peut plus améliorer la solution, et de le laisser retourner la solution en l'état.

Au final, lorsque après plusieurs exécutions, l'algorithme retourne une solution illégale, il est possible d'admettre qu'il n'existe aucune coloration en k couleurs légale, et considérer $k+1$ comme la valeur recherchée $\chi_s(G)$.

Malgré sa simplicité, les performances de l'algorithme peuvent être nuancées [5] : dans de nombreux cas, *Tabucol* se retrouve à réaliser un nombre conséquent d'itérations, sans modifier la solution. Ceci peut être lié à la définition même d'une itération de l'algorithme. Aussi, l'algorithme demande une place en mémoire importante, afin de mémoriser la *liste tabou*, dans laquelle pour chaque paire de sommets, de nombreuses informations peuvent devoir être enregistrées. Enfin, comme il a été énoncé plutôt, *Tabucol* demande plusieurs paramétrages précis (taille de la *liste tabou*, critère d'arrêt etc.) ; ce qui compromet ses performances globales.

Data: A graph $G = (V, E)$, an integer $k > 0$
Paramater: $MaxIter, L$ and λ
Result: Solution $c*$
Build a random solution c;
$c* = c$;
$iter = 0$;
Set the tabu list to the empty set; `/* No move is tabu */`
repeat
  Set $iter := iter + 1$;
  Choose a candidate 1-move (v, i) with minimum value $\delta(v, i)$;
  Introduce move $(v, c(v))$ into the tabu list for $L + \lambda F(c)$ iterations ;
  $c = c + (v, i)$;
  if $f(c) < f(c*)$ **then**
    $c* = c$;
  end
until $f(c) = 0$ *or* $iter = MaxIter$; `/* Stopping criterion */`

Algorithm 1: Algorithm Tabucol

2.2.3 HCD

Une méthode particulière de recherche locale se base sur la définition d'un ordre de priorité des sommets.

Un algorithme implémentant cette méthode est *HCD* [6], conçu dans l'idée d'un algorithme à la fois simple, et rapide, pour une utilisation dans un contexte industriel plutôt qu'à une fin de pure recherche.

Inspiré des techniques de recherche *tabou*, l'ordre de priorité se substitue à l'utilisation d'une liste tabou, dans la même optique d'éviter les boucles et optimums locaux.

Le programme se découpe en quatre fonctions principales. Une fonction d'initialisation assigne à chaque nœud i une couleur c_i lui étant propre, ainsi qu'une priorité p_i égale à sa couleur. Le maximum de couleurs requis (UB) est alors initialisé à $|V|$.

Une fonction nommée *Pull-colors* se charge de sélectionner le nœud ayant la plus haute priorité, et tente de « l'abaisser » en lui assignant la plus faible couleur compatible avec celles données à ses voisins, puis en mettant à jour sa priorité à la couleur lui ayant été assigné. Enfin, UB est modifié si besoin. Si la tentative échoue, la fonction passe au nœud suivant. Au bout de $|V|$ tentatives échouées (soit, quand plus aucun sommet ne peut être « abaissé »), la fonction *Push-colors* est appelée.

Cette fonction *Push-colors* va tenter d'harmoniser les couleurs de chaque sommet, en assignant si possible à chaque nœud une couleur plus élevée que celle déjà assignée, sans excéder UB ni créer de conflits.

La dernière fonction, *Pop-colors* est appelée lorsque *Push-colors* n'a pas réussi à modifier la coloration : elle a pour rôle d'échapper de l'optimum local, en remplaçant la classe de couleur S_1 par S_{UB+1}, et en associant à chaque nœud

de cette classe modifiée une priorité minimale, afin de ne pas se retrouver à la prochaine itération dans la même situation.

Concernant le critère d'arrêt de l'algorithme, un paramètre l détermine le nombre d'accès limite à la fonction *Push-colors*, et une couleur maximale doit être fixée, afin de s'arrêter lorsqu'une coloration légale de ce niveau est trouvée.

Comparé à l'algorithme de recherche *tabou*, nous pouvons remarquer que *HCD* travaille exclusivement sur des solutions légales. Ainsi, il peut être stoppé à n'importe quel moment et retourner une solution valide.

Néanmoins, il est à noté que la solution finale dépend toujours de la solution initiale (c'est à dire de l'ordre initiale attribué), et qu'il peut être nécessaire d'exécuter l'algorithme sur une même instance plusieurs fois, avec une initialisation différente, afin d'obtenir une solution plus précise.

2.2.4 Checkcol

Une rétro-ingénierie de *HCD* a été proposée sous le nom de *Checkcol* [5], rajoutant un mécanisme de *checkpoint*, correspondant à un régulier vidage de la mémoire considérée comme superflue à long terme.

Plus précisément, l'algorithme combine l'utilisation de mémoire à long terme, et de mémoire à court terme.

La conception de l'algorithme s'est faite dans le but de réduire les conséquences de deux des problèmes de *HCD* : l'effet de *grande portion*, correspondant à un temps passé trop conséquent sur des grandes portion de l'espace de recherche, et l'effet de *coloration similaire*, consistant à la génération de solutions trop proches les unes des autres.

La ré-initialisation de la mémoire s'effectue ainsi dans l'optique d'éviter l'effet de *grande portion*. Néanmoins, il est nécessaire après ces ré-initialisations de conserver une initialisation des priorités adéquate, afin d'éviter l'effet de *coloration similaire*.

L'intégration de ces deux mécanismes (checkpoint et gestion de priorités) permet de réaliser des recherches locales successives, interagissant les unes avec les autres, sans consommer toujours plus de mémoire.

Le fonctionnement de l'algorithme reste globalement similaire à celui de *HCD*, en rajoutant notamment des informations concernant la mise à jour d'un sommet : chaque sommet possède en plus de sa couleur et de sa priorité, un nombre indiquant son nombre de mise à jour de couleur effectuées depuis le dernier checkpoint, ainsi que l'itération de sa dernière mise à jour. Ceci permet d'éviter plus simplement l'effet de *coloration similaire*.

```
Function Initialize()
    Starting_upper_bound();
    foreach i ∈ V do
        update_count_i = 0;
        last_update_i = 0;
        p_i = c_i;
    end
    Set the checkpoint interval;
    Set the stopping criterion;
    Sort_nodes();
    k = 1;
    iterations = 0;
end
```

Algorithm 2: Algorithm Checkcol : Initialize

```
Function Pull-colors()
    Choose the highest priority vertex, i.e., i = Order[k] ;
    Assign to i the lowest admissible color c ;
    iterations = iterations + 1 ;
    p_i = c ;
    if c ≠ c_i then
        update_count_i = update_count_i + 1 ;
        last_update_i = iterations ;
    end
    if the stopping citerion is met then stop;
    if iterations > checkpoint then
        Start_New_Phase() ;
        return ;
    end
    if k = |V| then
        Sort_nodes() ;
        Update_UB() ;
        Push-colors() ;
    else
        k = k + 1 ;
    end
end
```

Algorithm 3: Algorithm Checkcol : Pull-colors

```
Function Push-colors()
    for k = |V| downto 1 do
        iterations = iterations + 1;
        i = Order[k];
        Let c be the largest color in [c_i, UB] that can be assigned to i;
        if c ≠ c_i then
            c_i = c;
            update_count_i = update_count_i + 1;
            last_update_i = iterations;
        end
        p_i = (1/c_i);
        if iterations > checkpoint then
            Start_New_Phase();
            return;
        end
        if the stopping citerion is met then stop;
        if coloring not changed then
            Pop-colors();
            return;
        else
            k = 1;
            Sort_nodes();
            Update_UB();
        end
    end
end
```

Algorithm 4: Algorithm Checkcol : Push

```
Function Pop-colors()
    for all vertices i ∈ V having smallest last_update_i do
        c_i = UB + 1;
        update_count_i = update_count_i + 1;
        iterations = iterations + 1;
        last_update_i = iterations;
    end
    if the stopping citerion is met then stop;
    foreach i ∈ V do p_i = (1/c_i);
    k = 1;
    Sort_nodes();
    Update_UB();
end
```

Algorithm 5: Algorithm Checkcol : Pop

2.2.5 Union Independant Set

Une alternative à la recherche locale, est l'utilisation d'algorithmes génétiques, prenant pour base de fonctionnement des mutations aléatoires des sommets.

Dans un algorithme génétique de coloration de graphe, une coloration spécifique est définie comme étant un individu de la populations, constituée de l'ensemble de toutes les colorations déjà réalisées par l'algorithme à un instant donné.

Un exemple d'algorithme utilisant cette méthode, est l'algorithme *Union Independant Set* (*UID*) [7], ayant la particularité d'être hybride, c'est à dire de combiner la recherche locale et le *croisement*.

De la même façon que pour *Tabucol*, *UID* se charge de retourner la meilleure coloration trouvée avant que le critère d'arrêt soit rempli, pour k couleurs. Ainsi, il faut appeler plusieurs fois l'algorithme pour k décroissant.

L'algorithme commence pas générer arbitrairement un première individu (représentant une coloration non forcément légale), qui constitue ainsi la population de départ. Il désigne par une variable s le meilleur individu (c'est à dire celui ayant la meilleure coloration), et par f le nombre de conflits dans cette solution s.

Par la suite, l'algorithme modifie la population jusqu'à ce qu'un critère d'arrêt soit rencontré, en appliquant successivement un *croisement* des populations, et des mutations par *recherche tabou*

Le processus de croisement consiste à *croiser* selon une probabilité fixée (déterminant l'importance du croisement par rapport aux mutations) chaque paire possible d'individus. Si deux individus se *croisent*, les deux individus sont remplacés par deux individus correspondant à une union des sous-ensemble indépendants de chaque coloration, afin de minimiser certaines classes de couleur, tout en valorisant d'autres : c'est en fait une tentative d'unification des sous-ensembles indépendants.

La mutation par *recherche tabou* s'effectue selon le principe usuelle, d'interdire les positions précédentes en les enregistrant dans une structure, puis de choisir selon une certaine probabilité au hasard un voisin parmi tous, ou d'appliquer une sélection *tabou* classique.

Data: G, a graph
Result: the number of conflicts with k fixed colors

```
/* f, f* : fitness function and its best value encountered so
   far                                                        */
/* s* : best individual encountered so far                   */
/* i, MaxIter : the current and maximum number of iterations
   allowed                                                    */
/* best(P) : returns the best individual of the population
   P                                                          */
i = 0;
generate(P_0);
s* = best(P_0);
f* = f(s*);
while f* > 0 and i < MaxIter do
    P'_i = crossing(P_i, T_x) ;          /* using UIS crossover */
    P_{i+1} = mutation(P'_i) ;             /* using tabu search */
    if f(best(P_{i+1})) < f*) then
        s* = best(P_{i+1});
        f* = f(s*);
    end
    i = i + 1;
    return f*;
end
```

Algorithm 6: Algorithm UIS

Conclusion

Pour résumer, la coloration de graphes est un problème qui existe depuis longtemps et qui n'a toujours pas été résolu. En effet, depuis l'émergence de l'informatique, des algorithmes ont été créés afin de tenter de trouver une coloration optimale de tout graphe en un temps réduit. Cependant, malgré la réalisation d'algorithmes suivant différentes approches au problème, telle que la recherche locale ou son hybridation avec des algorithmes génétiques, ou l'amélioration d'algorithmes existant suivant les époques, situation illustrée par HCD, qui a pu évoluer en CheckCol, il est toujours impossible d'obtenir un algorithme fiable trouvant une coloration optimale en temps polynomial sur tous les types de graphes. La difficulté est notamment qu'aucun algorithme ne présente en même temps les critères que sont la rapidité d'exécution et la fiabilité de la solution trouvée ; il est en effet parfois nécessaire de relancer plusieurs fois l'algorithme, avec des paramètres différents par exemple, pour obtenir une solution satisfaisante, dont on n'a toujours aucune certitude qu'elle est la meilleure. La coloration de graphe étant un problème NP-complet, l'existence ou non d'un tel algorithme est invariablement liée au problème historique des mathématiques et de l'algorithmique qu'est le problème $P = NP$.

Résumé

Ce document a été réalisé dans le cadre de l'unité d'enseignement Recherche Documentaire de deuxième année de Licence informatique mention Cursus Master en Ingénierie.

Il a pour sujet la coloration de graphe, et présente un bref résumé d'informations générales sur les graphes et leur coloration, ainsi que la présentation de différents algorithmes permettant de résoudre ce problème.

Mots-clé : graphes, coloration, algorithmes, recherche documentaire

Abstract

This document was realized as part of the course "Recherche Documentaire" (Literature Search), from the Second Year of the Bachelor of computer science, Cursus Master en ingénierie (Engineering Master's Degree course) mention.

It discusses graph coloring, and pressents a brief summary of general informations on graphs and their coloring, as well as presentations of different algorithms allowing the resolution of the problem.

Keywords: graphs, coloring, algorithms, literature search

Bibliographie

[1] E. Szemerédi A. Hajnal. Proof of a conjecture of erdős. *Colloq Math Soc Já nos Bolyai*, 1970.

[2] Frank Thomson Leighton. A graph coloring algorithm for large scheduling problems. *Journal of Research of the National Bureau of Standards*, 1979.

[3] Walter Klotz. *Graph Coloring Algorithms*. 2002.

[4] Alain Hertz Philippe Galinier. A survey of local search methods for graph coloring. *Computers & Operations Research*, pages 2547–2562, 2006.

[5] Giuseppe F. Italiano Massimiliano Caramia, Paolo Dell'Olmo. Checkcol : Improved local search for graph coloring. *Journal of Discrete Algorithms*, pages 277 – 298, 2006.

[6] Paolo Dell'Olmo Massimiliano Caramia. A fast and simple local search for graph coloring. In *Proceedings of the 3rd International Workshop on Algorithm Engineering*, pages 316–329. Springer-Verlag, 1999.

[7] Jin-Kao Hao Raphaël Dorne. A new genetic local search algorithm for graph coloring. In *Parallel Problem Solving from Nature — PPSN V*, pages 745–754. Springer Berlin Heidelberg, 1998.