TER : Coloration de graphes

Compte-rendu

Tutrice :
Nadia Brauner

Etudiant :
Michaël Gabay

Laboratoire G-SCOP

Année 2010

Table des matières

Remerciements

Je tiens à remercier **Nadia Brauner** pour son encadrement, ses conseils éclairés et ses suggestions tout au long du TER et **Julien Darlay**, notamment, pour l'aide qu'il m'a apporté avec `CPLEX` et pour trouver des instances de test.
Je remercie également le laboratoire **G-SCOP** et ses membres de m'avoir accueilli pour ce TER.

1 Introduction

La coloration de graphe est un problème omniprésent dans l'industrie de nos jours. Ce problème est très difficile et fait l'objet de nombreux travaux de recherche. Toutefois, dans un contexte d'évolution constante des techniques, de la puissance de calcul et des logiciels, on peut être amené à se poser la question suivante :
Quelle taille de graphe peut-on colorier optimalement avec un ordinateur aujourd'hui ?

Colorier un graphe c'est affecter une couleur à chacun de ses sommets de sorte que deux sommets adjacents ne soeint pas de même couleur (on peut de la même façon définir la coloration des arêtes mais ce problème se ramène très facilement à un problème de coloration des sommets en considérant le graphe de lignes correspondant).
Une coloration optimale d'un graphe est une coloration utilisant le nombre minimum de couleurs (appelé nombre chromatique et noté $\chi(G)$).
Colorier optimalement un graphe est un problème $\mathcal{NP}-complet$. C'est donc un problème très difficile qu'on ne peut résoudre en temps polynomial avec un algorithme déterministe (sauf si $\mathcal{P}=\mathcal{NP}$).

L'objectif de ce TER est de déterminer - compte-tenu des outils et ressources disponibles aujourd'hui - jusqu'à quelles limites on peut essayer de colorier optimalement un graphe.

Afin d'y répondre, j'ai commencé par me documenter puis à envisager plusieurs solutions pour la coloration de graphes. J'ai ensuite implémenté une solution sous forme de programme linéaire en nombres entiers et j'ai utilisé `CPLEX` pour résoudre des instances du problème. Puis, j'ai amélioré mon modèle en ajoutant des coupes permettant de réduire le temps de calcul. Après ceci, j'ai modélisé le problème en utilisant la programmation par contrainte et je l'ai résolu avec `Choco` et `CP`. Enfin, j'ai soumis mes algorithmes à une campagne de tests dont j'ai interprété les résultats.

Dans mon processus de réflexion, je me suis focalisé sur des graphes simples et connexes, toutefois j'ai pris soin de faire en sorte que mes modèles soient valides pour des multigraphes sans surcoût (on considère alors le graphe simple correspondant au multigraphe) et des graphes non connexes. Toutefois, dans le cas de graphes non connexes il serait plus judicieux de passer par une étape de préprocessing identifiant et séparant les composantes connexes pour les soumettre séparément au solveur plutôt que de soumettre le graphe directement.

2 Bibliographie

Comme indiqué dans l'introduction, le première partie de mon travail a été un travail bibliographique. J'ai lu les chapitres concernant la coloration de graphe (coloration des arêtes [1] et des sommets [2]) du livre Graphes et Hypergraphes [3] de Claude Berge. Je me suis également intéressé au livre Graph coloring problems [4] de Jensen et Toft.
Bien entendu, je me suis également servi d'internet et de wikipedia [5].

De cette étude sont ressortis plusieurs résultats et en particulier la minoration suivante :

Théorème 1. *Dans un graphe simple G de n sommets et m arêtes, on a :*

$$\chi(G) \geq \frac{n^2}{n^2 - 2m}$$

Cette minoration donne une borne inférieure pour le nombre chromatique très intéressante, particulièrement lorsque le graphe est dense.

3 La coloration comme Programme Linéaire en Nombres en Entiers

3.1 Modélisation

La première partie de mon travail a été de modéliser le problème de coloration de graphe en programme linéaire en nombres entiers.
Le modèle obtenu et utilisé est le suivant :
Pour un graphe $G = (V, E)$:

$$\begin{cases} \min z & \\ c_i \leq z & \forall i \in V \\ c_i \leq c_j - 1 + n \times y_{i,j} & \forall (i,j) \in E \\ c_j \leq c_i - 1 + n \times (1 - y_{i,j}) & \forall (i,j) \in E \\ y_{i,j} \in \{0,1\} & \forall (i,j) \in E \\ c \in (\mathbb{N}^*)^{|V|} & \end{cases} \tag{1}$$

c correspond au vecteur de la coloration et y à l'activation des contrainte (en fait y permet de linéariser la contrainte $|c_i - c_j| \geq \delta_{i,j}$ où $\delta_{i,j} = 1$ si $(i,j) \in E$ et $\delta_{i,j} = 0$ sinon - dit autrement, cette contrainte correspond à "deux sommets adjacents ne sont pas de même couleur"). z est l'indice de plus grande couleur. Minimiser z c'est donc trouver une coloration c telle que $z = \chi(G)$.

3.2 Résolution

Pour modéliser et résoudre ce problème, j'ai utilisé `CPLEX` et `OPL`.
`CPLEX` est un solveur permettant de résoudre des programmes linéaires en nombres entiers de manière exacte. C'est un logiciel d'`IBM-ILOG` [6].

`OPL` est le langage de modélisation d'`IBM-ILOG`. Il permet de modéliser facilement des problèmes complexes pour les soumettre à `CPLEX`.

Toutefois, la programmation linéaire en nombres entiers est un problème $\mathcal{NP}-complet$. Il ne faut donc pas espérer pouvoir résoudre facilement (1) pour un graphe quelconque.

Afin de pouvoir résoudre le problème, je l'ai modélisé et j'ai recherché des majorations et des contraintes permettant de le résoudre plus rapidement. J'ai également essayé d'optimiser l'aspect algorithmique du problème en évitant des opérations inutiles (placement des contraintes, initialisation et ajout de contraintes sous certaines conditions...).
J'ai pris le soin de démontrer les coupes utilisées et de vérifier leurs compatibilités afin de garantir mon modèle.

Pour représenter les graphes, j'ai utilisé la matrice d'adjacence du graphe car compte-tenu de la complexité du problème, celle-ci sera toujours de dimension réduite. De plus, c'est l'outil idéal pour manipuler les données car le problème de coloration de graphe est un problème d'adjacence des sommets (deux sommets adjacents ne sont pas de même couleur).

J'ai modélisé y comme une matrice de $\mathcal{M}_n(\{0,1\})$ où n est l'ordre du graphe à colorier (i.e. le nombre de sommets du graphe).

Remarque : les programmes et modèles écrits permettent de colorier des multigraphes (les graphes simples étant des multigraphes particuliers).

3.2.1 Validation

A chaque amélioration du modèle, je l'ai testé sur de nombreux graphes afin de permettre d'une part de valider l'intérêt des améliorations apportées en termes d'efficacité de la résolution et d'autre part, de valider le modèle (vérifier que le nombre de couleurs obtenu est bien le nombre chromatique et que la coloration optimale renvoyée est valide).

Pour vérifier la bonne coloration du graphe, j'ai testé les modèles sur des graphes de petite taille et je les ai vérifiés manuellement.
De plus, afin d'effectuer les tests d'efficacité et de valider mes modèles, j'ai choisit dans un premier temps d'utiliser comme instances de tests des graphes complets. J'ai fait ce choix pour deux raisons. La première est qu'ils me permettent de vérifier facilement la bonne coloration. La seconde est qu'ils constituent le pire cas pour le nombre chromatique (celui-ci est majoré par $d+1$ où d est le degré maximum d'un sommet dans le graphe simple correspondant et dans le cas d'un graphe complet, $\chi(G) = d+1$), dans le cas où on ne dispose pas d'une minoration intéressante ils forcent donc le solveur a énumérer et tester un très grand nombre de solutions.

Toutefois, une minoration intéressante est fournie pas le théorème 1 (en particulier, elle vaut n pour le graphe complet d'ordre n : K_n). Une fois cette minoration implémentée, j'ai donc du tester mes modèles sur d'autres exemples.
J'ai choisit d'utiliser les instances du challenge DIMACS de coloration de graphe [7], celles-ci offrant

des exemples variés et dont on connaît, pour une partie, le nombre chromatique. J'ai également testé l'efficacité de la résolution sur des multigraphes connexes générés en utilisant la librairie `Boost`. La répartition des arêtes suivant une loi uniforme.

Les instances utilisées sont disponibles aux adresses suivantes (fichiers .col), sur la page de Michael Trick [8] : http://mat.tepper.cmu.edu/COLOR04/ et http://mat.gsia.cmu.edu/COLOR/instances.html.
`Boost` [9] est une collection de librairies `C++` open source. Vous trouverez plus d'informations sur `Boost` à cette adresse : http://www.boost.org/. J'ai particulièrement utilisée la `Boost Graph Library` [10].

3.2.2 Optimisations

Comme expliqué précédemment, j'ai testé et amélioré les performances de mes algorithmes. Je vais détailler ici les améliorations apportées ainsi que les gains associés en performances (donnés en pourcents - ces gains correspondent à la moyenne de gains de performances par rapport au modèle précédent sur des graphes testés dans les deux cas).
Pour tester ces exemples, j'ai utilisé `CPLEX` 10 pour architectures 32 et 64 bits. Mes tests m'ont rapidement poussé à adopter `CPLEX` 10 pour architectures 64 bits, le gain en performances par rapport la version 32 bits étant non négligeable.

Remarque : afin de vous faire ressentir les améliorations apportées en termes de performances, j'indique dans la suite de cette partie certains temps écoulés pour la coloration et le nombre d'itérations de l'algorithme. Les données en temps ne sont qu'un indicateur subjectif : elles dépendent fortement de la machine, du nombre d'itérations de l'algorithme et du nombre de noeuds explorés lors de la résolution. Les calculs ayant été effectués sur mon ordinateur portable, les données en temps ne sont en aucun cas représentative de ce qui aurait été obtenue sur un serveur dédié. Des tests de performance plus poussés et détaillés se trouvent dans la section 6.

Le modèle initial ne correspond pas exactement au modèle (1) - j'y ai apporté la modification évoquée plus haut : le vecteur des couleurs est un vecteur d'entiers compris entre 1 et $d+1$ (où d est le degré maximum d'un sommet du graphe simple correspondant). Pour ce premier modèle, les performances sont exécrables : l'algorithme colore K_{10} en 52 secondes et 428761 itérations de l'algorithme (K_n est le graphe complet à n sommets).

Les amélioration apportées sont les suivantes (le gain est le pourcentage de temps économisé par rapport à la résolution utilisant toutes les optimisations précédentes) :

1. **Fixer la couleur du premier sommet** (`c[1] == 1` dans le modèle).
   Coloration de K_{10} en 3,8 secondes et 30470 itérations de l'algorithme.
   Coloration de K_{30} non terminée après 2 heures et presque 15 millions d'itérations.
   Gain : 93%.
2. **Ajout de la minoration obtenue par le théorème 1** (`z >= chi_min`).
   Cette minoration du nombre chromatique est particulièrement efficace pour les graphes très denses.

En témoigne l'amélioration des performances sur les graphes complets :
Coloration de K_{10} en moins de 0,1 secondes et 72 itérations de l'algorithme.
Coloration de K_{30} en 1,2 secondes et 2275 itérations de l'algorithme.
Coloration de K_{100} en 36 secondes et 34645 itérations de l'algorithme.
Gain :
– $> 99\%$ sur les graphes complets (car on a alors `chi_min`$= n = \chi(G)$).
– élevé sur les graphes denses.
– faible sur les graphes peu denses

3. **Initialisation de `y[1][i]` et `y[i][1]`.**
On avait précédemment rajouté l'initialisation du premier sommet, on peut également initialiser les valeurs de y correspondantes.
Gain : presque nul sous `CPLEX` 10 car 1 étant le premier sommet, `CPLEX` 10 fixe les valeurs de y correspondantes au début de chaque itérations avant de passer aux autres sommets. C'est en quelque sorte une duplication de contrainte.
Sous `CPLEX` 12, le comportement est tout autre : ces contraintes peuvent induire un gain considérable de performances, aussi bien qu'une perte considérable.

4. **Test de la valeur de `m[i][j]` pour ajouter les contraintes de non adjacence des couleurs.**
`m` est la matrice d'adjacence du graphe.
Jusqu'à présent, les contraintes de non adjacence des couleurs (les deux dernières inéquations dans 1) étaient de la forme :
`m[i][j] * c[i] <= c[j] - 1 + V * y[i][j]`, on avait donc dans le modèle un nombre important de contraintes inutiles et pénalisant l'exécution de l'algorithme (car elles doivent quand même être évaluées pour fixer les valeurs de y). De plus, le modèle ne fonctionnait alors que pour des graphes **simples**.
En ajoutant la contrainte `c[i] <= c[j] - 1 + V * y[i][j]` si et seulement si `m[i][j] != 0` (c'est le test), on élimine des contraintes pénalisant l'algorithme et on peut désormais colorier des **multigraphes** - en prenant soin, bien sûr, de bien calculer les autres variables : par exemple, d, le degré maximum d'un sommet utilisé pour certaines majorations doit être celui du graphe simple correspondant, il n'est donc pas égal à $\frac{1}{2}\max_i(\sum_{j=1}^{n} m_{i,j})$ mais à $\frac{1}{2}\max_i(\sum_{j=1}^{n} \mathbb{1}_{m_{i,j}\neq 0})$, il en va de même pour les autres variables qui sont à calculer dans le graphe simple correspondant.
Gain : 14% sur K_{100} (colorié en 31s).

5. **Ajout des contraintes de non adjacence des couleurs pour $j > i$ seulement.**
Il s'agit en fait d'une contrainte inutilement dupliquée car on a :

$$c_i \leq c_j - 1 + n \times y_{i,j} \Leftrightarrow c_i \leq c_j - 1 + n(1 - \times y_{j,i})$$

et de même :

$$c_j \leq c_i - 1 + n \times (1 - y_{i,j}) \Leftrightarrow c_j \leq c_i - 1 + n \times y_{j,i}$$

car $y_{i,j} = 1 - y_{j,i}$.
Il suffit donc d'écrire ces contraintes une seule fois.
Ceci réduit le nombre de contraintes total de plus de la moitié (on passe de $2n^2$ contraintes de non adjacence des couleurs à $n^2 - n$ contraintes).

6. **Teste si le graphe est complet** (`chi_min == n`).
   C'est l'ajout d'un cas particulier coloriable trivialement.
   Gain : $> 99\%$ si le graphe est complet, 0% sinon (pas de pertes de performances).
7. **Ajout de la contrainte** $y_{i,j} = 1 - y_{j,i}$.
   C'est l'intérêt des $y_{i,j}$ qui permettent l'activation des contraintes de non adjacence des même couleurs.
   Il suffit de remplacer $y_{i,j}$ par sa valeur dans le modèle (1) pour constater que dans le cas où $y_{i,j} = 0 = y_{j,i}$, la première inéquation appliquée à i et j donne $c_i \leq c_j - 1$ et $c_j \leq c_i - 1$ ce qui est absurde. De même avec la deuxième équation si $y_{i,j} = 1 = y_{j,i}$.
   Notons que lorsque cette contrainte est mal placée dans le programme (i.e. après les deux inéquations), l'effet est désastreux et on perd fortement en efficacité.
   Gain : 46% sur des multigraphes dont les arêtes sont distribuées selon une loi uniforme.
8. **Ajout de la contrainte** $c_i \leq i$.
   Il existe toujours une coloration optimale telle que pour tout i, $c_i \leq i$. En ajoutant cette contrainte, on limite grandement le nombre de colorations que le solveur va tester. Même si on limite également le nombre de solutions avec une coloration optimale qu'il peut trouver, cette diminution est négligeable devant celle du nombre de noeuds explorés par le solveur.
   Gain : 87% sur des multigraphes dont les arêtes sont distribuées selon une loi uniforme.
9. **Prise en compte d'une clique dans la coloration**.
   Une clique est un sous-graphe complet du graphe d'origine.
   Les sommets d'une clique étant tous adjacents entre eux, ils sont deux à deux de couleurs différentes. On peut donc, pour initialiser la résolution, commencer par affecter des couleurs aux sommets d'une clique. De plus, le nombre chromatique est minoré par le nombre de sommets de la clique.
   Toutefois, si on veut préserver la contrainte très puissante : $c_i \leq i$, il faut prendre soin de mettre les k sommets de la clique en positions 1 à k de la matrice d'adjacence afin de pouvoir initialiser leurs couleurs en garantissant qu'il existera toujours une coloration optimale avec $c_i \leq i$.
   Gain : le gain dépend fortement des caractéristiques du graphe. Plus la taille de la clique est proche du nombre chromatique, plus le gain sera substantiel (celui-ci allant de 0 à 70%).

J'ai également essayé d'ajouter la contrainte $c_i \leq d(v_i) + 1$ ($d(v)$ est le degré de v) mais celle-ci ne permet pas de gagner en performances. Dans certains cas, elle dégrade même les performances de la résolution.

Toutes ces optimisations ont permis de passer d'un modèle exécrable à un modèle permettant d'envisager de colorier optimalement bon nombre de graphes.
Vous trouverez section 6 des tests de performances et des analyses.

4 La coloration comme Programme Par Contraintes

La programmation par contrainte est un moyen puissant permettant de décrire et résoudre des problèmes. Elle consiste à décrire le problème en utilisant des contraintes. Contrairement à la programmation linéaire en nombres entiers, nous ne sommes pas limités à des inéquations mais nous

pouvons également utiliser toute sortes d'outils tels que les inégalités, les minimum, les maximum, des valeurs absolues...
La puissance de la programmation par contrainte réside dans sa simplicité comme en témoigne la modélisation du problème de coloration de graphe section suivante.

Tout comme la résolution d'un programme linéaire en nombres entiers, la résolution d'un programme par contraintes est un problème $\mathcal{NP} - complet$. Toutefois, les méthodes employées pour résoudre un problème par contrainte diffèrent de la programmation linéaire en nombres entiers. Employer une deuxième modélisation, permettra donc de voir quelle méthode se prête le mieux à la résolution du problème de coloration de graphes.

4.1 Modélisation

Le problème de coloration de graphe se modélise en programmation par contraintes de la manière suivante :

$$\begin{cases} \min z & \\ c_i \leq z & \forall i \in V \\ c_i \neq c_j & \forall (i,j) \in E \\ c \in (\mathbb{N}^*)^{|V|} & \end{cases} \tag{2}$$

Ce modèle est beaucoup plus simple et naturel que le modèle 1.

4.2 Résolution du programme par contraintes

Afin de résoudre le problème, j'ai utilisé `CHOCO` 2.1.1 [11] et `CP` 2.
Le premier est un solveur open source développé, entre autres, par des chercheur nantais. Il offre une API en `Java`.
Le second est développé par `IBM - ILOG` [6], tout comme `CPLEX`.

L'utilisation de ces deux solveurs m'a permis de tester leurs performances, l'un contre l'autre et contre la programmation linéaire en nombres entiers.

Pour représenter les graphes, j'ai utilisé la matrice d'adjacence.

Remarque : sous `CHOCO`, j'ai remplacé la contrainte $c_i \leq z \ \forall i \in V$ par $z = \max_{v_i \in V}(c_i)$. Des tests que j'ai effectué avec `CP` montrent que la contrainte $c_i \leq z \ \forall i \in V$ est en fait meilleure (on explore quelques noeuds de moins pour la résolution).

4.2.1 Optimisations

De même que précédemment, le modèle peut être amélioré.
J'ai choisit d'optimiser celui-ci dans le programme pour `CP` car ce solveur se révèle plus performant que `CHOCO`.

J'y ai introduit toutes les optimisations vu précédemment (cf sous-section 3.2.2) n'ayant pas trait au programme linéaire, ce qui correspond aux points : 1, 2, 3, 4, 6, 7, 8 et 9.

5 Programmes et outils

5.1 Programmes utilisés

Comme expliqué plus haut, j'ai utilisé `OPL` (modélisation), `CPLEX` (programmation linéaire en nombres entiers) et `CP` (programmation par contraintes) de `IBM - ILOG` [6] pour modéliser et résoudre les problèmes.
J'ai également utilisé `CHOCO` [11] pour la programmation par contraintes.

Outre les solveurs, j'ai utilisé `nauty` [12] de Brendan D. McKay pour générer quelques graphes (`nauty` permet d'énumérer des graphes non isomorphes). Toutefois, celui-ci énumérant tous les graphes, il devient inutilisable dès qu'on veut obtenir des graphes ayant plus de 10 sommets. J'ai donc écrit des programmes me permettant de générer des graphes. Vous trouverez plus de détails à leurs propos dans la section suivante.

Pour trouver une clique maximum dans un graphe, j'ai utilisé `Cliquer` [13] de Sampo Niskanen et Patric R. J. Östergård. Ce programme m'a permis d'évaluer les performances de mon heuristique et s'il était envisageable de chercher la clique max dans un graphe pour colorier celui-ci (pour ceci, il faudrait que le coût pour la recherche d'une clique max soit négligeable devant le coût de la coloration).

5.2 Mes programmes

J'ai écrit les modèles pour `CPLEX` et `CP` en `OPL` (c'est le langage de modélisation de `IBM - ILOG` [6]).

Comme expliqué plus haut, j'avais besoin d'un outil me permettant de générer des graphes. J'ai donc utilisé la `Boost Graph Library` [10] pour écrire des programmes (en `C++`) me permettant d'importer des graphes du challenge DIMACS [7] de coloration et de générer des multigraphes connexes dont la répartition des arêtes suit une loi uniforme.

Afin de visualiser ces graphes (soit générés, soit importés), j'ai également introduit une option dans mes programmes qui permet de générer un fichier correspondant au graphe au format `dot` de `graphviz` [14]. On peut ensuite compiler ce fichier avec `dot` pour générer une image correspondant au graphe.
J'ai également codé une heuristique très basique de recherche de clique dans un graphe, l'objectif étant de démontrer l'intérêt où non de prendre en compte une clique pour la coloration. L'heuristique est sans garantie. Elle prend le sommet de plus grand degré et son voisin de plus grand degré puis cherche le sommet voisin des deux premiers de plus grand degré et ainsi de suite.
Afin de pouvoir prendre en compte cette clique, mon programme réordonne ensuite les sommets pour que ceux de la clique soient en tête de la matrice d'adjacence.

Pour résoudre un problème, trois possibilités se présentent donc :

1. on dispose d'un fichier au format DIMACS pour la coloration de graphes.
   Il suffit alors de le soumettre au parser qui génère un fichier qu'on peut soumettre au solveur.

2. on génère aléatoirement un graphe.
   Le fichier généré est alors au format adéquate.
3. on dispose de la matrice d'adjacence du graphe.
   Il suffit de la mettre en forme pour pouvoir soumettre le graphe au solveur.

6 Expérimentations / Analyse

La question posée était :

"Quelle taille de graphe peut-on colorier optimalement avec un ordinateur aujourd'hui ?"

Afin de répondre à cette question, j'ai soumis mes modèles à une campagne de tests. L'objectif était de déterminer l'efficacité de la coloration sur des graphes variés et d'en fixer les limites.

Pour ces tests de performance j'ai utilisé `CPLEX` 10 et 12 afin de déterminer dans un premier temps s'il y avait des différences majeures entre ces deux versions. Puis j'ai fixé la version à utiliser pour la suite, pour les raisons expliquée plus loin (sous-section 6.1).

J'ai effectué tous les tests sur mon ordinateur personnel afin que les durées aient un sens comparativement. C'est un ordinateur portable équipé d'un processeur `Intel(R) Core(TM) 2 Duo P8400` (deux coeurs cadencés à 2,26 GHz).
`CPLEX` 12 a fonctionné en parallèle, utilisant 75% des capacités des coeurs. Les autres solveurs ont été lancés en mode monocoeur.

Dans la suite de cette partie, je vais décrire un certain nombre de problèmes résolus et donner les résultats expérimentaux correspondants.
J'utilise pour mesurer les performances plusieurs indicateurs :

- La durée de résolution.
  Cet indicateur est subjectif et dépend très fortement de la machine et des autres indicateurs. Toutefois, il est nécessaire pour pouvoir répondre à la question posée.
  Note : pour `CP` et `CPLEX`, ce temps correspond au temps de résolution indiqué par `CPLEX` (ils n'incluent pas le temps de chargement du graphe, de chargement du modèle et de lancement du solveur). Pour `CHOCO`, il correspond au temps indiqué par la commande `time`. Il inclut le temps de parsing du fichier DIMACS, de chargement du graphe, de chargement du modèle et de lancement du solveur. Il est donc normal que quand les temps sont petits, ceux de `CHOCO` soient plus grands.
- Le nombre de noeuds (pour `CPLEX`) ou de branches (pour `CP`) explorés lors de la résolution.
  C'est une donnée objective puisqu'elle est indépendante de la machine. Toutefois, le temps nécessaire pour explorer un noeud varie grandement d'un graphe à un autre.
- Le nombre de noeuds (pour `CPLEX`) ou de branches (pour `CP`) explorés lorsqu'une solution optimale est trouvée.
  Cette donnée permet de mesurer le temps passé par le solveur pour démontrer qu'une solution est optimale (ou plutôt démontrer qu'aucune autre n'est meilleure).

Je ne donne pas le nombre d'itérations de l'algorithme car il est fortement lié au nombre de noeuds explorés, une seule des deux données suffit donc.

Afin de décrire les graphes, j'ai notamment utilisé la densité. Celle-ci correspond en fait à la densité du graphe simple correspondant au multigraphe. Elle est égale au nombre d'arêtes du graphes divisé par le nombre d'arêtes du graphe complet ayant autant de sommets : si le graphe simple G a n sommets et m arêtes, alors $densité(G) = \frac{2m}{n^2-n}$

J'ai utilisé mes deux dernières modélisations pour tester les performances : avec (`mod9`) et sans (`mod8`) la prise en compte de la clique.
J'ai choisit dans le modèle prenant en compte la clique de ne pas initialiser les valeurs des $y_{1,j}$ et $y_{j,1}$ mais seulement celles de la clique (i.e. les $y_{i,j}$ où i et j sont des sommets de la clique) et ce pour les raisons expliquées dans le point 3 des optimisations et parce que j'ai observé une dégradation des performances lorsque les deux contraintes étaient couplées.

Remarque : Les instances de test du challenge DIMACS viennent toutes avec la clique maximum en tête du graphe, il ne faut donc pas s'étonner que les performances pour les resoudre en l'état soient meilleures que lorsque j'utilise mon programme pour trouver une clique et réordonner les sommets.
Bien que je spécifie que la taille de la plus grande clique soit 1 pour les instances où je n'utilise pas mon heuristique, le solveur prend tout de même en compte la clique pour les deux modèles (notamment sous `CPLEX` 10 où il explore les sommets dans l'ordre).

Remarque 2 : dans les tests, lorsque j'ai interrompu le calcul, j'ai noté $> y$ pour indiquer que lorsque le calcul a été interrompu, la valeur était y.

6.1 Problème des n reines

Ce problème consiste à trouver si on peut (et si oui comment) placer n reines sur un échiquier de taille $n \times n$ sans qu'elles ne se menacent.

Pour ce faire, on modélise l'échiquier par n^2 sommets, chacun représentant une case. Deux sommets sont adjacents s'ils sont sur la même ligne, même colonne ou même diagonale.
La réponse à la question est oui si et seulement si le nombre chromatique est n.

Les instances utilisées proviennent du challenge DIMACS de coloration.

Les fichiers utilisés été créés par Michael Trick [8] à partir de graphes issus de "The Stanford GraphBase" de Donald Knuth [15].

Queen i_i correspond au problème des i reines et Queen i_i-heuristique correspond au problème des i reines dont les sommets sont réordonnés pour placer en tête du graphe la clique obtenue par l'heuristique.

Propriétés des graphes simples correspondants :

	Nombre chromatique	Arêtes	Sommets	Degré max	Densité
Queen 5-5	5	160	25	16	0,53
Queen 6-6	7	290	36	19	0,46
Queen 7-7	7	476	49	24	0,40

	Plus grande clique	Clique en tête	Taille indiquée
Queen5_5-heuristique	5	5	5
Queen5_5	5	5	1
Queen6_6-heuristique	6	4	4
Queen6_6	6	6	1
Queen7_7-heuristique	7	5	5
Queen7_7	7	7	1

Temps de calcul (en secondes) :

	CPLEX 12		CPLEX 10		CHOCO	CP
	mod8	mod9	mod8	mod9	PPC	PPC
Queen5_5-heuristique	0,09	0,09	0,27	0,31	3,05	0
Queen5_5	0,19	0,23	0,4	0,39		0
Queen6_6-heuristique	7,58	7,98	688,1	699,7	6,31	0,08
Queen6_6	75,36	16,32	15,6	15,07		0,04
Queen7_7-heuristique	> 2460	48,57	246	210	6,24	0,26
Queen7_7	491,38	1006,57	x	> 2640		0,09

Nombre de noeuds explorés :

	CPLEX 12		CPLEX 10	
	mod8	mod9	mod8	mod9
Queen5_5-heuristique	0	0	31	31
Queen5_5	165	165	84	84
Queen6_6-heuristique	4200	4200	589900	589900
Queen6_6	59000	13100	17700	17700
Queen7_7-heuristique	> 1464000	30970	117300	117300
Queen7_7	314754	640512	x	> 2051700

Nombre de noeuds explorés lorsqu'une solution optimale est trouvée :

	CPLEX 12		CPLEX 10	
	mod8	mod9	mod8	mod9
Queen6_6-heuristique	2793	2733	54241	54241
Queen6_6	57030	11629	5123	5123
Queen7_7-heuristique	> 1464000	4248	95929	95929
Queen7_7	314754	640512	x	> 2051700

Concernant ce problème, on remarque plusieurs chose :

- **Les résultats avec les deux modélisations sont exactement les mêmes sous `CPLEX` 10**.
  Ceci montre que le solveur traite linéairement les contraintes. En particulier, s'il commence par explorer le premier sommet, il va commencer par initialiser les $y_{1,j}$ et les $y_{j,1}$. De plus, la clique étant toujours en tête, il va automatiquement lui affecter les couleurs correspondantes (il y a une unique solution réalisable pour les couleurs des sommets de la clique car on a $c_i \leq i$). Sous `CPLEX` 10, les deux modèles sont en fait strictement équivalents.
- à contrario, sous `CPLEX` 12, les deux modèles ne sont pas du tout équivalents dès lors que le graphe est un peu complexe. Ceci est dû à la mécanique interne du solveur qui doit probablement essayer de casser les symétries du problème et ne pas traiter les contraintes linéairement.
- **Une nette amélioration des performances entre `CPLEX` 10 et `CPLEX` 12**.
  Cette amélioration se confirme sur de très nombreux autres graphes testés, j'ai donc décidé de n'utiliser que `CPLEX` 12 pour la suite des tests.
  Toutefois, dans le cas du problème des 6 reines, avec la modélisation 8, on remarque que les performances sont moins bonne sous `CPLEX` 12. Ceci s'explique par les raisons évoquées dans le point précédent : parfois il peut s'avérer plus judicieux de traiter les contraintes linéairement.
- **Les temps de résolution / nombre de noeuds explorés sont très sensibles à l'ordre des sommets**.
  C'est un résultats à la fois étonnant et très intéressant : on constate sur les problèmes des 6 et des 7 reines que ce n'est pas forcément en connaissant la plus grande clique qu'on colore le plus vite. Pour vérifier ceci, j'ai relancé les tests Queen6_6 et Queen7_7 en indiquant la taille de la clique. Les temps de calcul et nombre de noeuds explorés diminuent alors mais restent supérieurs à ceux des Queen-heuristique qui correspondent aux même graphes mais avec les sommets de la clique obtenu par l'heuristique en tête de la matrice d'adjacence (les sommets sont réordonnés).
- **Le modèle 9 semble meilleur que le modèle 8**.
  Il semble qu'on obtienne de meilleures performances avec le modèle 9. Toutefois, dans le cas de Queen7_7 c'est l'inverse même si le gain est flagrant pour ce même graphe lorsque les sommets sont réordonnés (Queen7_7-heuristique), la supériorité d'un des modèles sur l'autre reste à démontrer.
- **Les performances de la résolution en utilisant la programmation par contraintes sont très supérieures aux performances de la résolution du programme linéaire en nombre entiers**.
  Dans ce cas particulier, on peut expliquer ceci par la structure du problème des reines qui se soumet très bien à la programmation par contrainte. Toutefois, on est ici dans le cas d'un modèle de coloration de graphe et il est peu probable que le solveur ai détecté qu'il s'agisse en fait d'un modèle beaucoup plus simple. De plus, les différences de performances sont telles qu'il faut s'attendre à les retrouver pour d'autres graphes.

Ce premier jeu de tests a donc permis de cerner un nombre important de différence entre les modèles et les solveurs. Il reste à confirmer ceci et à trouver éventuellement d'autres différences.

De plus, les résultats obtenus ici permettent d'espérer fortement pouvoir colorier des graphes de taille allant jusqu'à 50 sommets et de densité 0,5 avec la programmations linéaire en nombres

entiers et de pouvoir aller beaucoup plus loin avec la programmation par contraintes.

6.2 Allocation de registres et Graphes de livres

D'autres instances de test utilisées provenant du challenge DIMACS sont les problème de coloration des graphes suivants :

- Graphe d'allocation de registres pour des variables (issus de cas réels).
  C'est un problème complexe pour lequel les compilateurs utilisent des heuristiques.
  Fichier : `fpsol2.i.3`, réalisé par Gary Lewandowski.
- Graphes des relations entre personnages dans des oeuvres célèbres.
  Réalisé par Michael Trick [8] à partir de graphes issus du livre de Donald Knuth [15].
  Chaque personnage est un sommet et deux sommets sont reliés par une arête si les deux personnages correspondants se sont rencontrés dans le livre.
  Livres : Anna Karenina de Tolstoy (`anna`), David Copperfield de Dicken (`david`), l'Iliade d'Homère (`homer`), Huckleberry Finn de Twain (`huck`) et Les Misérables de Victor Hugo (`jean`).

Les caractéristiques des graphes simples correspondants sont les suivantes :

	Nombre chromatique	Arêtes	Sommets	Degré max	Densité
fpsol2.i.3	30	8688	425	346	0,096
anna	11	493	138	71	0,05
david	11	406	87	82	0,11
homer	13	1629	561	99	0,01
huck	11	301	74	53	0,11
jean	10	254	80	36	0,08

	Plus grande clique	Clique en tête	Taille indiquée
fpsol2.i.3	30	15	15
anna	11	7	7
david	11	11	11
homer	13	10	10
huck	11	5	5
jean	10	4	4

Temps de calcul (en secondes) :

	CPLEX 12		CHOCO	CP
	mod8	mod9	PPC	
fpsol2.i.3	> 2574	x	x	> 12042
anna	2,30	3,51	580	0,19
david	0,22	0,21	317	0,01
homer	x	x	> 2507	> 460
huck	> 5360	> 3745	482	134
jean	> 2907	> 4827	128	0,86

Nombre de noeuds / branches explorés :

	CPLEX 12		CP
	mod8	mod9	PPC
fpsol2.i.3	> 44 100	x	> 74 427 000
anna	1 600	3 200	13 852
david	62	62	76
homer	x	x	> 6 328 000
huck	> 12 723 900	> 8 437 600	12 881 318
jean	> 6 862 900	> 115 17 900	63 997

Nombre de noeuds / branches explorés lorsqu'une solution optimale est trouvée :

	CPLEX 12		CP
	mod8	mod9	PPC
fpsol2.i.3	> 44 100	x	413
anna	64	64	132
david	62	62	76
homer	x	x	8 770
huck	122	122	70
jean	117	117	77

Cette fois encore, les performances de la résolution en utilisant la programmation par contrainte sont frappantes : elle surpassent de très loin celles de la programmation linéaire en nombre entiers.

On voit également apparaître des limites de la coloration optimale : on n'arrive ni à colorier les personnages de l'Iliade, ni le graphe d'allocation des registres. Tous deux ont un nombre de sommets supérieur à 400. De plus, la coloration en tant que solution d'un PLNE semble atteindre ses limites puisque ni "Huckleberry Finn", ni "Les Misérable" ne sont coloriables. De surcroît, ceci montre bien à quel point la méthode est sensible à la structure du graphe car "David Copperfield" qui a la même densité que "Huckleberry Finn" et plus de sommets est coloriable.

Un autre point intéressant est de voir que les deux modèles 8 et 9 ont l'air ici presque équivalents. En fait, on retrouve quelque chose qu'on pouvait constater dans l'exemple précédent : lorsque la solution optimale est trouvée rapidement (moins de 1000 noeuds), elle est trouvée après le même nombre de noeuds explorés par le solveur pour les deux modèles. Il en va de même pour les nombres totaux de noeuds explorés qui, lorsqu'ils sont petits, sont égaux pour les deux modèles.

Enfin, le point le plus intéressant soulevé par ces tests est la vitesse à laquelle une solution optimale est trouvée. Même lorsque la résolution ne termine pas, une solution optimal est trouvée dans les premiers dixièmes de secondes de calcul. Ce n'étais pas le cas dans les exemples précédents pour

la programmation linéaire en nombre entiers. Pour ce qui est de la programmation par contraintes, on ne peut rien dire sur le problèmes des 5, 6 et 7 reines car les temps de résolution sont trop court. Toutefois, j'ai utilisé la programmation par contraintes pour résoudre le problème des 8 reines et on a bien le même résultat (optimal trouvé après 11390 branches et problème résolu après 3 666 516 branches et 147 secondes).
Afin de résoudre un problème de coloration dans le monde industriel, on peut donc envisager d'allouer un certain temps à la résolution et, lorsque ce temps est écoulé, on prend la meilleure coloration trouvée. Cette coloration sera soit la coloration optimale si l'algorithme a terminé, soit la meilleure bonne coloration obtenue si la résolution n'était pas terminée et on aura de fortes chances d'avoir en fait une coloration optimale.

6.3 Graphes de Mycielski

Les graphes de Mycielski sont les cas pathologiques par excellence pour la coloration de graphe. Le graphe de Mycielski de rang n, M_n, est le plus petit graphe (i.e. avec le moins de sommets et d'arètes) sans triangle (c'est à dire que K_3 n'est pas un sous-graphe de M_n) tel que $\chi(G) = n+1$. L'absence totale de clique de taille 3 ou plus dans le graphe rend la coloration très difficile car on a très peu de cas où on peut déduire directement la couleur d'un sommet des contraintes et des sommets coloriés précédemment, l'optimum est donc très difficile à trouver et à prouver.

C'est un test de performance important car si on peut colorier optimalement un graphe de Mycielski à k sommets alors on peut probablement colorier optimalement tous les graphes à k sommets ou moins en un temps très inférieur.

Les caractéristiques des graphes simples correspondants sont les suivantes :

	Plus grande clique	Clique en tête	Taille indiquée
M_3	2	2	2
M_4	2	2	2
M_5	2	2	2
M_6	2	2	2

	Nombre chromatique	Arêtes	Sommets	Degré max	Densité
M_3	4	20	11	5	0,36
M_4	5	71	23	11	0,28
M_5	6	236	47	23	0,22
M_6	7	775	95	47	0,17

Temps de calcul (en secondes) :

	CPLEX 12		CHOCO	CP
	mod8	mod9	PPC	
M_3	0,03	0,03	0,74	0,00
M_4	0,62	0,65	3,76	0,03
M_5	851	1 456	221,00	9,43
M_6	x	$>$ 2 641	$>$ 2625	$>$ 8288

Nombre de noeuds / branches explorés :

	CPLEX 12		CP
	mod8	mod9	PPC
M_3	0	0	20
M_4	1 200	1 200	1 598
M_5	1 441 200	2 468 000	499 704
M_6	x	$>$ 353 300	$>$ 303 985 k

Nombre de noeuds / branches explorés lorsqu'une solution optimale est trouvée :

	CPLEX 12		CP
	mod8	mod9	PPC
M_3	0	0	10
M_4	38	38	22
M_5	78	78	46
M_6	284	284	94
M_7	3 102	3 102	2514

Cette fois encore, les performances de la programmation par contraintes outrepassent largement celles de la programmation linéaire en nombre entiers. Toutefois, ce n'est pas suffisant pour réussir à colorier M_6 en deux heures, même si, de même que pour M_7, le nombre chromatique est trouvé très rapidement.

Les résultats obtenus précédemment et ceux obtenus sur les graphes de Mycielski nous prouvent qu'on peut essayer, avec assurance de colorier optimalement les graphes à moins de 50 sommets et que la limite pour la programmation linéaire en nombre entiers est probablement autour de 60.
Pour ce qui est de la programmation par contrainte, la limite est en revanche plus élevée mais grâce à ce test, on sait qu'à partir de moins 97 sommets celle-ci peut s'avérer problématique. Je vais essayer de déterminer cette limite dans les tests suivants.

La modélisation 8 est ici plus intéressante que la neuvième. C'est tout à fait normal car la prise en compte d'une clique est complètement inutile dans le cas des graphes de Mycielski puisque la plus grande clique est constituée de 2 sommets.
Bien que combiner les optimisations consistant à fixer les $y_{1,i}$, $y_{i,1}$ et prendre en compte une clique

fait perdre du temps sur le meilleur temps de résolution des deux modèles, il peut être intéressant de faire un compromis et de combiner ces deux optimisations afin d'obtenir un modèle qui, à coup sûr, a un temps de résolution compris entre les deux temps de résolution (c'est ce qui se passe en pratique lorsqu'on les combine).

6.4 Graphes aléatoires

Afin de déterminer les limites de la coloration optimale, j'ai généré un grand nombre de multigraphes aléatoires en fixant le nombre de sommets, d'arêtes et en imposant que le graphe soit connexe (dans le cas où le graphe généré n'est pas connexe, le programme de génération ajoute des arêtes pour le rendre connexe). Les arêtes sont distribuées selon une loi uniforme.
J'ai ensuite soumis ces graphes aux solveurs.

Contrairement aux exemples précédents issus du challenge DIMACS, ces graphes étant générés aléatoirement, je ne connais ni leurs nombres chromatiques (du moins pas avant de l'avoir calculé), ni les tailles de leurs plus grandes cliques.

Les graphes dont le nombre chromatique est calculé avec succès par toutes les méthodes sont les graphes suivants (la colonne arêtes du multigraphe correspond au nombre d'arêtes demandé, ce nombre peut-être inférieur à la valeur réelle) :

Sommets	Arêtes du multigraphe	Arêtes du graphe simple	densité du graphe simple
10	30 / 50 / 100	22 / 31 / 39	0,49 / 0,69 / 0,87
20	50 / 100 / 200	42 / 80 / 124	0,22 / 0,42 / 0,65
30	100 / 200 / 300 / 500	89 / 160 / 205 / 304	0,20 / 0,37 / 0,47 / 0,70
50	100 / 500	99 / 415	0,08 / 0,39
70	100 (trivial)	102	0,04

Dès 50 sommets, lorsque la densité dépasse $\frac{1}{2}$, la résolution du programme linéaire en nombres entiers devient très difficile. Ces observations confirme les observations précédentes : la limite de la taille d'un graphe coloriable en utilisant la programmation linéaire en nombres entiers, avec une densité de $\frac{1}{2}$ est de 50 sommets. On peut toutefois aller plus loin lorsque la densité est faible (comme en témoigne la coloration de "David Copperfield" - 84 sommets - et "Anna Karenina" - 138 sommets).

Il est à noter que lors de la résolution de ces problèmes, le modèle 9 est plus efficace que le modèle 8 et que tous ces graphes sont coloriés en quelques secondes.

Avec `CP`, j'ai également pu colorier les graphes suivants :

Sommets	Arêtes du multigraphe	Arêtes du graphe simple	densité du graphe simple
50	1000	699	0,57
70	500 / 1000 / 1500 / 2000	447 / 845 / 1093 / 1376	0,19 / 0,35 / 0,45 / 0,57
80	100 / 500 / 1000	106 / 466 / 847	0,03 / 0,15 / 0,27
100	1000	911	0,18

Ces graphes ont été coloriés pour la plupart en quelques secondes voire quelques minutes. Toutefois, la coloration du graphe à 70 sommets et 1000 arêtes a duré à peu près une demi heure et celle du graphe à 70 sommets et 1500 arêtes un peu plus d'une heure.
Ces durées qui se rallongent m'ont fait sentir qu'on se rapproche de la limite.
En effet, dès 80 sommets, on ne parvient plus à colorier les graphes à 1500 arêtes (densité 0,37 ; arrêté après plus de 6 heures) à 2000 arêtes (densité 0,47 ; arrêté après deux heures). J'ai également essayé de colorier sans succès un graphe à 100 sommets et 3000 arêtes (densité 0,45 ; arrêté après plus de 9 heures de calcul).
La limite pour la coloration de graphe en utilisant la programmation par contraintes semble donc se situer autour de 75 à 80 sommets pour une densité de $\frac{1}{2}$. Même si bien sûr on peut aller plus loin (voire beaucoup plus loin) lorsque la densité est faible.

7 Conclusion

J'ai utilisé la programmation linéaire en nombre entiers et la programmation par contraintes pour résoudre optimalement le problème de coloration de graphe.
En utilisant des logiciels à la pointe pour résoudre ces problèmes, j'ai pu estimer quelles sont les limites de la coloration optimale des graphes.
À la question "Quelle taille de graphe peut-on colorier optimalement avec un ordinateur aujourd'hui ?", je peux répondre qu'avec les modèles que j'ai mis au point et les logiciels que j'ai utilisé (notamment `CP`), on peut colorier optimalement des graphes ayant une densité de $\frac{1}{2}$ jusqu'à au moins 75 sommets et qu'on peut même aller beaucoup plus loin lorsque la densité est plus faible.
Lorsque le nombre de sommets du graphe est inférieur à 500, je pense qu'on peut tirer un très grand bénéfice de l'heuristique proposée plus haut (partie 6.2). Lorsque celui-ci est supérieur, il est à mon avis plus judicieux de recourir directement à une heuristique spécialisée.

Mes modèles peuvent de plus être encore améliorés. En particulier, on peut introduire une étape de pré-résolution dans laquelle on trouverait une coloration avec une heuristique afin d'obtenir un majorant du nombre chromatique qu'on injecterait ensuite dans le modèle. De plus, on pourrait utiliser une meilleure heuristique que celle que j'ai utilisé pour trouver une clique. On pourrait même éventuellement trouver une clique de taille maximum avec un algorithme exacte puisque cliquer, par exemple, permet de trouver rapidement une clique dans des graphes de moins de 200 sommets.
J'avais également envisagé de résoudre le problème en parallèle mais le saut de complexité entre le calculable et le non calculable est tel qu'il faudrait un nombre de processeurs exponentiel pour repousser les limites du calcul. Mettre au point des méthodes de résolution parallèles n'a de réel intérêt que pour diminuer le temps de calcul des solutions déjà calculables.

J'ai modélisé le problème de coloration de graphe en d'autres problèmes pour le résoudre. Il également est possible d'utiliser d'autres approches. En particulier, il reste une piste importante à explorer qui est celle de l'écriture d'algorithmes spécialisés, utilisant par exemple le principe de reliements-contractions et de la séparation des pièces.

Références

[1] C. Berge. *Graphes et hypergraphes*, chapter 12. Indice chromatique, pages 236–259. Dunod, Paris, 1970.

[2] C. Berge. *Graphes et hypergraphes*, chapter 15. Nombre chromatique, pages 314–346. Dunod, Paris, 1970.

[3] C. Berge. *Graphes et hypergraphes*. Dunod, Paris, 1970.

[4] T.R. Jensen and B. Toft. Graph coloring problems. *Bull. Amer. Math. Soc. 33 (1996), 287-288. DOI : 10.1090/S0273-0979-96-00651-9 PII : S 0273-0979 (96) 00651-9 Copyright of article : Copyright 1996, American Mathematical Society*, 1996.

[5] Wikipedia, l'encyclopédie libre. http://en.wikipedia.org/wiki/Main_Page.

[6] IBM − ILOG. http://www-01.ibm.com/software/websphere/products/optimization/.

[7] DIMACS Challenges. http://dimacs.rutgers.edu/Challenges/.

[8] Michael Trick's Operations Research Page. http://mat.tepper.cmu.edu/.

[9] Boost library. http://www.boost.org/.

[10] Boost Graph Library (BGL). http://www.boost.org/doc/libs/1_43_0/libs/graph/doc/table_of_contents.html.

[11] CHOCO. http://www.emn.fr/z-info/choco-solver/index.html.

[12] Brendan D. McKay. nauty. http://cs.anu.edu.au/~bdm/nauty/.

[13] Sampo Niskanen et Patric R. J. Östergård. Cliquer. http://users.tkk.fi/pat/cliquer.html.

[14] Graphviz Graph Visualization Software. http://www.graphviz.org/.

[15] D.E. Knuth. *The Stanford GraphBase*. ACM Press, 1994.