Le coloriage de graphe, un jeu d'enfant ?

Maurice Clerc

Le coloriage de graphe, un jeu d'enfant ?

Maurice Clerc

January 31, 2023

Le jeu est l'occupation la plus sérieuse de l'enfant

Albert Brie (Le mot du silencieux)

Table des Matières

Avant-propos

Dans les écoles maternelles et primaires on amuse parfois les enfants en leur demandant de colorier un dessin formé de ronds et de traits entre eux, avec deux règles :

- deux ronds reliés par un trait ne doivent pas être de la même couleur ;
- le meilleur coloriage est celui qui utilise le moins de couleurs différentes.

Il est d'ailleurs instructif d'observer comment les enfants tentent de trouver une bonne solution, surtout si, au lieu de crayons de couleurs, ils peuvent utiliser des jetons colorés et les déplacer.

Cependant le problème, sous le terme générique de *coloriage optimal de graphe*, est parfois d'une redoutable complexité et ce petit livre n'a nullement la prétention d'être un exposé complet de l'état de l'art. Tout au plus essayé-je d'exposer des éléments de difficulté croissante, depuis des jeux d'apparence simple jusqu'aux algorithmes quantiques. Entre les deux j'insiste surtout sur des méthodes de résolution déterministes car elles me semblent sous-estimées, au détriment de celles faisant appel au hasard. Pour ces dernières, dont je dis quand même quelques mots, il existe une vaste littérature que le lecteur intéressé pourra trouver facilement, surtout en anglais, par des recherches sur les mots-clés *graph coloring, stochastic methods*.

Plus spécifiquement d'autres informations et théorèmes sont disponibles dans des ouvrages et articles déjà publiés ou sur l'Internet. Il s'agit essentiellement de divers algorithmes de coloriage et d'études portant sur des classes particulières de graphes, comme planaires (dessinable sur un plan sans croisement de traits) ou en arbre. Par exemple un graphe planaire ne nécessite jamais plus de quatre couleurs et un graphe en arbre jamais plus de deux.

Comme toute présentation d'un travail scientifique doit donner les éléments nécessaires à une éventuelle reproduction j'ai inclus les principaux codes sources qui ont été utilisés.

Un dernier point et non des moindres : le but de ce livre est surtout de donner des éléments pour inciter à méditer sur les fascinants problèmes des coloriages de graphes. Ainsi, par exemple, certaines définitions de codages sont juste présentées, sans exemple d'utilisation. C'est volontaire, à vous de voir si elles peuvent vous intéresser. Et, dans le même esprit, certaines affirmations

sont énoncées sans être prouvées en détail (néanmoins tous les éléments pour le faire assez facilement sont là).

Alors bonne lecture et bonnes réflexions !

Outils

Une part importante de la documentation pour cet ouvrage a été rassemblée par consultation de l'Internet grâce au métamoteur de recherche Startpage (https://startpage.com). Son principal avantage est d'envoyer simultanément la même requête à plusieurs moteurs de recherche et d'agréger les résultats, tout en ne communiquant aux-dits moteurs que le minimum d'information sur l'origine de la requête.

La rédaction a été faite sous LyX (http://lyx.org), qui permet de générer simplement des fichiers LaTeX sans avoir à connaître ce langage. La bibliographie a été gérée par Zotero (https://www.zotero.org) et son extension LyZ pour l'interfacer avec LyX. Les graphiques ont été réalisés à l'aide des logiciels suivants :

- LibreOffice Calc (http://www.libreoffice.org) ;
- Gimp (http://www.gimp.org) ;
- Inkscape (https://inkscape.org/) ;
- GNU Octave (https://www.gnu.org/software/octave), un quasi-clone du logiciel commercial Matlab© et également ce dernier.
- Mathematica (https://www.wolfram.com/mathematica/)

Les programmes informatiques ont été écrits en langage Matlab© ou Octave. Enfin, les traitements informatiques ont été réalisés sous le système d'exploitation Linux Ubuntu 22.04, sur un micro-ordinateur 64 bits, dont l'epsilon-machine est $2,22\times10^{-16}$ (pour mémoire, cela signifie que tout calcul portant sur des nombres de valeur absolue censée être inférieure à cette valeur doit être regardé avec la plus grande suspicion).

Contacter l'auteur

Si vous souhaitez envoyer des commentaires, signaler des erreurs ou apporter des suggestions, vous pouvez me contacter par courriel, à l'adresse Maurice.Clerc@WriteMe.com.

Chapitre 1

Des jeux

À première vue ce chapitre peut sembler futile, voire inutile. Mais dans les trois jeux présentés, surtout les deux derniers, plus complexes, il est intéressant d'observer et d'analyser les stratégies plus ou moins intuitives mises en œuvre. En effet la plupart des algorithmes de résolution déterministes que nous verrons ensuite s'appuient sur de telles stratégies.

Considérons naïvement le graphe de la figure 1.0.1 et formalisons déjà un peu. Les ronds s'appellent les *sommets* du graphe et les traits entre les sommets sont les *arêtes* du graphe. On utilise aussi les mots *nœuds* et *arcs*. Néanmoins le mot arc est en général plutôt utilisé quand les traits sont des flèches, indiquant un sens de parcours.

Pour chaque nœud on peut compter le nombre d'arêtes qui s'y rattachent : c'est le *degré* du nœud. Sur la figure, on voit qu'il y a un nœud de degré six, trois de degré cinq, deux de degré quatre, et un de degré trois. La somme des degrés est 32 et le nombre d'arêtes est donc la moitié, 16.

Nous nous intéressons ici aux nombreuses manières de colorier les nœuds d'un graphe en se donnant des règles et des contraintes. Il est possible de les considérer comme des jeux, dont certains sont d'ailleurs difficiles à résoudre**(author?)** [16], mais commençons par des choses simples. Déjà, réglons une petite question de dénomination : doit-on parler de coloriage ou de coloration ? Consultons le dictionnaire de l'Académie française :

- coloriage : Action de colorier ; résultat de cette action. Le coloriage d'une carte géographique. Exposer des coloriages d'enfants.
- coloration : Action de colorer, le fait de se colorer ; état de ce qui est coloré. La coloration des fruits par l'action du soleil. À l'automne, les feuillages prennent une coloration jaune et rousse. La coloration de la peau, sa pigmentation. Une coloration artificielle, lumineuse, sombre, terne. Fam. : coiffure. Se faire une coloration, teindre ses cheveux, en modifier le reflet, l'éclat.
- colorier : Teinter un dessin, une surface, de diverses couleurs. Colorier

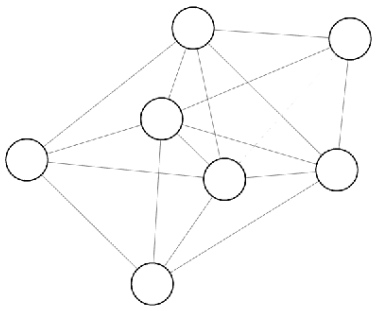


Figure 1.0.1: Un graphe à sept nœuds et seize arêtes.

une estampe, une image. Des planches coloriées. Un album à colorier. Colorier à l'aquarelle, au lavis.

- colorer : Donner à une matière une couleur déterminée. Colorer un bois, une étoffe. Colorer une coupe microscopique. L'art de colorer le cristal, les vitraux. Du verre coloré en rouge, en bleu.

Clairement, d'après ces définitions, le couple colorier/coloriage est le plus pertinent pour décrire ce que nous étudions ici. Si l'on voit pourtant souvent l'autre couple, colorer/coloration dans la littérature c'est sans doute sous l'influence des publications en anglais (américain), qui parlent de *color/coloration*.

Notez que ce chapitre est le premier de la progression « facile » (jeux) vers « difficile » (algorithmes quantiques) évoquées et peut donc être ignoré sans inconvénient si vous connaissez déjà bien le sujet.

1.1 Le jeu de Col

En général il se pratique sur un graphe *planaire*, mais ce n'est pas une obligation. Notez que malgré les apparences, le graphe de la figure 1.0.1 est bel et bien planaire, en le redessinant autrement (figure 1.1.1 où, pour mieux s'y repérer, les nœuds ont été numérotés). En effet, il n'est pas nécessaire que les arêtes soient rectilignes, car, en fait, chacune représente simplement un lien entre deux nœuds.

1.1.1 Règles

Le jeu de Col peut se pratiquer dès six ans, voire avant en modifiant un peu le vocabulaire et la présentation des règles. Le nom est à la fois une allusion au créateur du jeu (Colin Vout) et au fait qu'il s'agit d'un jeu de coloriage.

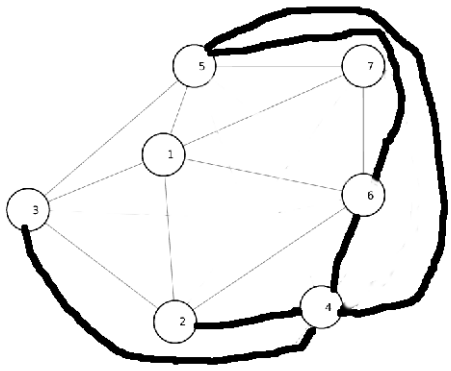


Figure 1.1.1: On peut redessiner le graphe de la figure 1.0.1 sans croisement d'arêtes : il est planaire.

En général on joue à deux, avec deux crayons de couleur, par exemple bleu et rouge.

Le joueur qui a le crayon bleu commence. Il colorie un des sommets de son choix. Ensuite c'est au tour du joueur ayant le crayon rouge, il colorie un des sommets restants. Puis le joueur au crayon bleu colorie un des sommets restants et ainsi de suite. Mais il ne faut jamais colorier un sommet dans la même couleur qu'un des sommets voisins (auxquels il est *adjacent*, c'est-à-dire relié par un trait).

Notons au passage, même si c'était implicite jusqu'ici, que nous ne considérons que des graphes *connexes*, c'est-à-dire pour lesquels on peut toujours aller d'un nœud à un autre en suivant des arêtes et bidirectionnels (une arête peut etre parcourue dans les deux sens).

Le premier qui est bloqué (parce que tous les sommets non encore coloriés sont adjacents à un sommet de la couleur de soncrayon et que c'est à son tour de jouer) a perdu la partie.

Sur des grands graphes, on peut jouer à plus de deux et avec plus de couleurs.

1.1.2 Stratégies

Même pour ce jeu tout simple on ne connaît pas de méthode générale pour gagner à coup sûr[1] mais il y a quand même quelques stratégies qui fonctionnent souvent. Définissons d'abord la *distance* entre deux nœuds : c'est le nombre d'arêtes du plus court chemin pour aller d'un nœud à l'autre. Si l'on considère les distances pour tous les couples de nœuds, la plus grande est appelée le *diamètre* du graphe. Sur la figure 1.1.2 il est de deux, même si cela ne saute pas aux yeux. On a alors comme règles empiriques :

[1] Écrit en octobre 2022.

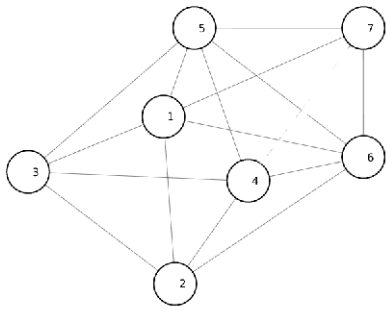


Figure 1.1.2: Entre les nœuds 3 et 7, la distance est de 2, en suivant le chemin 3-1-7. Sur ce graphe il n'y a d'ailleurs aucune distance supérieure à 2, qui est donc la valeur de son diamètre.

- colorier un nœud du plus petit degré possible ;
- s'il y en a plusieurs, en choisir un qui soit le plus « loin » possible de ceux déjà coloriés avec la couleur en cours ;

À noter aussi la règle du (pair, perd) : si le diamètre est pair, alors le joueur qui commence risque fort de perdre, si son adversaire joue bien.

Une variante du jeu de Col est présentée dans le livre de Dorian Mazauric sur les jeux de graphes « grandeur nature » (**(author?)** [22]), même si ce n'est pas explicitement indiqué. On dessine un graphe sur le sol et des équipes s'affrontent, chacune d'une couleur donnée (chasuble, foulard, brassard, etc.). Chaque équipe place à tour de rôle un de ses membres sur un nœud libre du graphe.

La première équipe qui est forcée de placer un joueur « à côté » d'un autre de sa couleur a perdu. Ici « à côté » signifie bien sûr qu'il y a une arête reliant les deux positions. Un joueur, une fois placé, ne peut pas être déplacé. Les équipes doivent donc appliquer une stratégie « gloutonne » telle que celles décrites plus loin, dans la section 4.2.

1.2 Le Colorigraphe

Le Colorigraphe est un jeu de réflexion ... et de rapidité ! Praticable à partir de 5-6 ans, mais ne vous y trompez pas : même des adultes « coincent » parfois !

D'une part, il peut amener à devoir tenter de colorier des graphes à douze nœuds (nombre qui, d'ailleurs, dans le principe, pourrait être augmenté) ce qui, en soi, n'est déjà pas simple, et, d'autre part comme dans ce jeu on se limite à trois couleurs au maximum, il n'y a pas toujours de solution. Encore faut-il

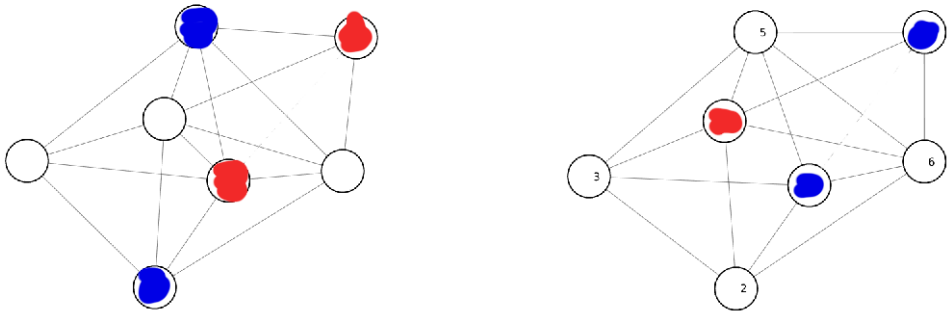


Figure 1.1.3: Jeu de Col. Une fin de partie. À gauche le joueur «bleu » a mal commencé et perd. À droite il a mieux commencé, sur un nœud de faible degré, et il gagne, le « rouge » ayant à tort joué sur le nœud 1 au lieu du 3.

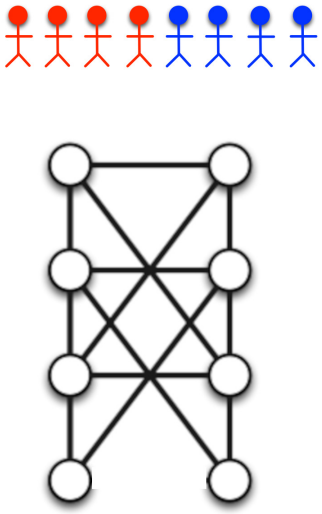


Figure 1.1.4: Le jeu de Col par équipe en extérieur. Si l'équipe qui commence joue bien, elle gagne.

d'ailleurs en être sûr. En cas de doute on peut utiliser *a posteriori* un algorithme sur ordinateur, capable de dire s'il existe une solution ou pas. Et s'il en existe, d'en donner une et même la meilleure possible.

Faisons donc une description détaillée de ce jeu, avec des exemples.

La version « de luxe » nécessite un matériel spécifique (voir la liste ci-dessous et la photo 1.2.5), mais on peut parfaitement se contenter de dessiner les graphes sur une feuille. Néanmoins des jetons de couleur sont quand même utiles, pour tenter plus facilement diverses possibilités.

1.2.1 Matériel

- Plateau avec 12 plots numérotés disposés sur un cercle (ou des fiches prédessinées).
- De quoi matérialiser les arêtes. Tiges avec annneaux aux extrémités (6 tiges pour chacune des 7 tailles possibles). On peut aussi utiliser des élastiques. Ou simplement un crayon si l'on utilise des fiches.
- 12 jetons plats troués bicolores, par exemple bleu d'un côté, rouge de l'autre.
- 6 jetons plats troués noirs.
- Deux dés 12 faces (numérotées). Si quatre dés sont disponibles, deux peuvent être utilisés pour noter l'enjeu en cours.
- De quoi noter les enjeux et les points gagnés (papier+crayon ou barrette avec billes, par exemple).
- Sabliers (30 s, 1mn, 2 mn) ou minuteur. Un minuteur un peu stressant (tic-tac et sonnerie finale) augmente la difficulté. .

1.2.2 Déroulement d'une partie

Selon les joueurs, on peut augmenter ou réduire le temps accordé pour chaque problème.

Les joueurs conviennent du total de points à accumuler pour gagner, par exemple au moins 100 points, pour une partie à deux joueurs ou 70 points pour une partie à trois joueurs (voir ci-dessous le calcul des points) ou moins si l'on veut une partie plus rapide. Chaque joueur, à tour de rôle (sauf en cas de défi, voir ci-après) :

- construit un problème ;
- tente de le résoudre dans le délai accordé.

1.2.3 Construire un problème

Le joueur dont c'est le tour définit un enjeu. C'est un nombre entre 2 et 24. Noter cet enjeu. Le joueur à sa gauche lance les dés autant de fois que cet enjeu. Si les deux mêmes numéros apparaissent, refaire un lancer. À chaque fois, matérialiser l'arête correspondant, c'est-à-dire placer une tige (ou tendre un élastique, ou tracer un trait) entre les plots (les nœuds) dont les numéros sont donnés par les dés. Note : si le lancer des dés redonne une arête existante, le refaire.

Cette phase est omise si les graphes sont prédessinés sur des cartes (voir la variante 3 ci-après). . Dans ce cas il convient de les tirer au hasard.

1.2.4 Résoudre un problème (colorier)

Le coloriage consiste à mettre un jeton sur chaque plot, dans la durée accordée. Un coloriage est valide si aucune arête n'a de jetons de même couleur à ses deux extrémités. Régler le minuteur à une ou deux minutes, selon le niveau des joueurs. Note : Les plots isolés (qui ne supportent aucune arête) peuvent être ignorés.

Trois cas possibles :

1. Coloriage valide, uniquement rouge et bleu. Le joueur gagne un nombre de points égal à l'enjeu.

2. Coloriage valide, mais avec des jetons noirs. Chaque jeton noir fait perdre deux points. Le joueur gagne donc l'enjeu diminué de deux fois le nombre de jetons noirs utilisés, sauf s'il y a défi (voir ci-dessous).

3. Coloriage non valide. Dans les cas 2 et 3 l'autre joueur (ou un des deux autres dans le cas d'une partie à trois) peut lancer un défi, en disant « Je prends ! », « Défi ! » ou « Je peux faire mieux ».

1.2.5 Défi

Le joueur qui a lancé un défi tente lui-même de résoudre le problème. Comme il a eu le temps de l'observer, le minuteur est réglé à la moitié du temps initial. Si, dans le temps imparti, il réussit un meilleur coloriage, il gagne des points comme indiqué ci-dessus.

Pour le cas 2, un meilleur coloriage est un coloriage également valide mais avec moins de jetons noirs, voire pas du tout.

Pour le cas 3, un meilleur coloriage est un coloriage valide quelconque, avec deux ou trois couleurs.

Attention : si le joueur qui a lancé le défi ne réussit pas, son adversaire gagne l'enjeu !

Note : après un défi on continue le tour normal des joueurs. Pour une partie à deux, le joueur qui a lancé le défi joue donc deux fois de suite.

1.2.6 Exemple d'une partie entre Sara et Samuel

1. Sara est la plus jeune, elle commence. Elle choisit un enjeu très prudent de 5. Samuel lance 5 fois les dés (5-3, 7-12, 11-12, 12-3, 6-3) et Sara met à chaque fois en place l'arête correspondante. Le minuteur est réglé à une minute. Elle trouve facilement une solution et gagne 5 points (voir la figure 1.2.1).

2. Au tour de Samuel. Il choisit un enjeu un peu plus risqué de 10. Sara lance 10 fois les dés, et Samuel construit un problème à 10 arêtes. Le minuteur est réglé à une minute. Samuel ne trouve pas de solution à deux couleurs (d'ailleurs c'est impossible, voir la figure 1.2.2), mais avec un jeton noir, il obtient quand même un coloriage valide. Il gagne alors 10-2=8 points.

3. Pour se rattraper, Sara tente un enjeu de 12. Avec 12 lancers des dés (par Samuel), elle construit un problème assez difficile. Le minuteur est réglé à une minute. Elle ne trouve pas de solution à deux couleurs (c'est impossible, voir la figure 1.2.3). Elle tente alors avec des jetons noirs, mais la minute est dépassée.

4. Samuel lance un défi. Le minuteur est réglé à 30 secondes. Samuel trouve une solution dans le temps imparti, avec deux jetons noirs. Il gagne donc 12-4=8 points et c'est à nouveau à lui de jouer. S'il n'avait pas trouvé de solution à temps, Sara aurait gagné 12 points.

5. Samuel tente un gros coup : un enjeu de 24. Il place les 24 arêtes définies par les dés lancés par Sara (cela peut être un peu long !). Le minuteur est réglé à une minute. Samuel voit rapidement que le bicoloriage est impossible (voir la figure 1.2.4) et tente un tricoloriage en utilisant des jetons noirs. Mais la minute est dépassée.

6. Sara, ayant bien observé le problème, lance un défi. Avec le minuteur réglé à 30 s, elle trouve une solution à quatre jetons noirs. Elle gagne donc 24-8=16 points

7. etc., jusqu'à ce que l'un des joueurs atteigne au moins le total défini au départ.

1.2.7 Variantes

Vous pouvez facilement inventer des variantes. En voici par exemple quelques-unes.

1.2.7.1 Variante 1

L'enjeu est choisi au hasard en lançant les dés. Sa valeur est donnée par la somme des points.

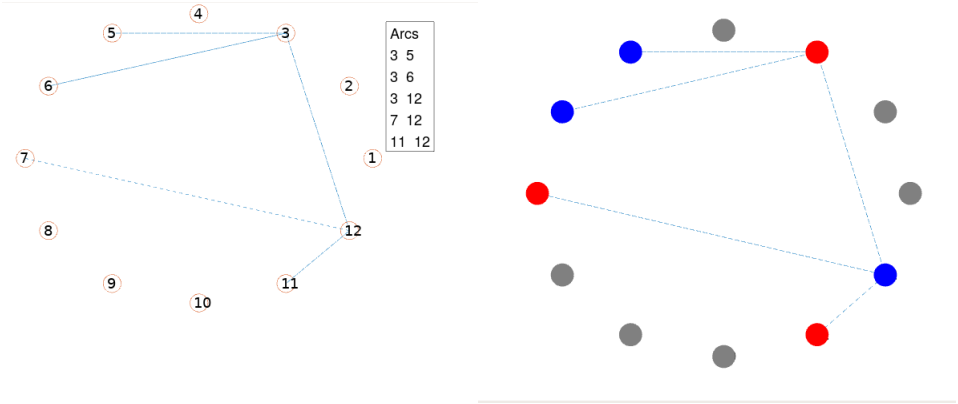


Figure 1.2.1: Enjeu de 5 pour Sara. Le problème et la solution trouvée.

1.2.7.2 Variante 2 (simplification)

Construire un problème (lancer les dés, placer les arêtes). Placer 12 jetons bicolores sur les 12 plots, la même couleur visible. Démarrer le minuteur. Une tentative de résolution consiste alors à retourner des jetons pour essayer de trouver un bicoloriage valide. En cas d'échec, on ne cherche pas de tricoloriage.

1.2.7.3 Variante 3 (avec du matériel supplémentaire)

Les problèmes sont prédéfinis sur des cartes. Le paquet de cartes est mélangé, posé sur la table face cachée et le joueur dont c'est le tour tire une carte.

1.2.8 Quelques astuces

- S'il y a trois arêtes formant un triangle, le bicoloriage est impossible.
- Si vous repérez des triangles, placez un jeton noir sur le plot ayant le plus d'arêtes.
- Plus généralement, si vous ne trouvez pas rapidement de bicoloriage, commencer par placer un jeton noir sur le plot ayant le plus d'arêtes (degré maximum) est souvent une bonne stratégie pour trouver un coloriage correct (mais pas forcément le meilleur).

1.3 En solitaire

Même si le Colorigraphe se joue en général à deux où à la rigueur trois, en réalité le principe de base peut être généralisé pour jouer seul : étant donné un graphe, le colorier avec un nombre de couleur minimum, le *nombre chromatique*

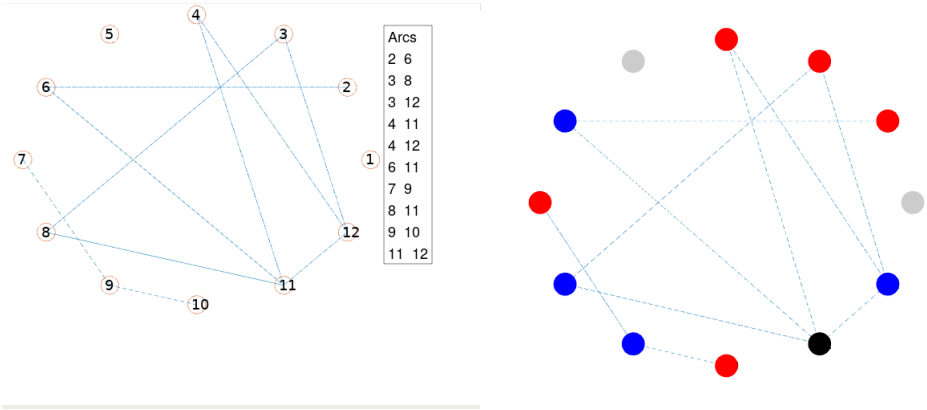


Figure 1.2.2: Enjeu de 10 de Samuel. Le problème et la solution trouvée, avec un jeton noir.

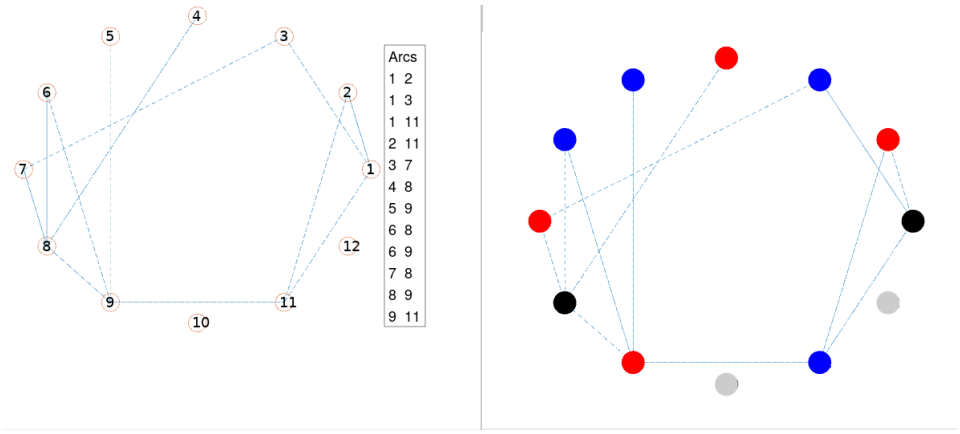


Figure 1.2.3: Enjeu de 12 de Sara. Elle ne trouve pas de solution en moins d'une minute, mais Samuel a lancé un défi et en a trouvé une avec deux jetons noirs, en moins de 30 secondes.

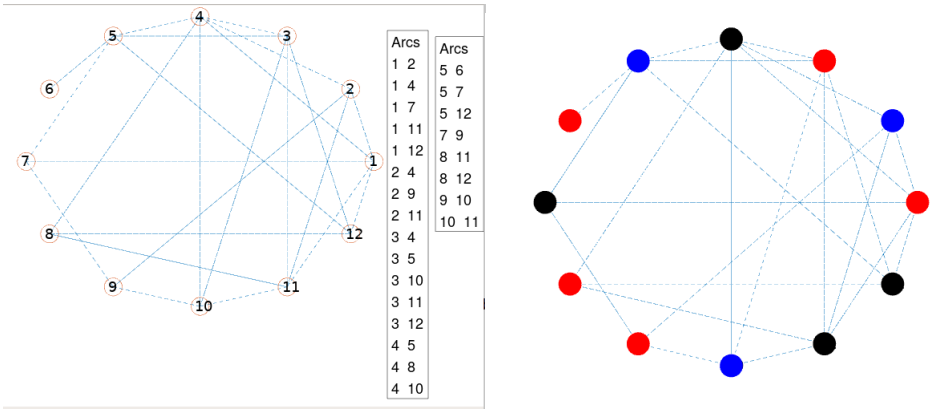


Figure 1.2.4: Enjeu de 24 de Samuel. Mais c'est Sara qui trouve une solution à quatre jetons noirs, après avoir lancé un défi. Notez qu'il existe une solution avec seulement trois jetons noirs, en 2, 5 et 10.

du graphe. Pour générer et dessiner un graphe, on peut procéder d'une manière analogue, avec des dés :

- un premier lancer donne le nombre de nœuds ;
- un deuxième lancer donne le nombre d'arêtes ;
- les lancers suivants donnent pour chaque arête les numéros de son origine et de son extrémité.

Mais il est bien plus simple d'utiliser un programme informatique de génération aléatoire d'un graphe connexe bidirectionnel, suivi d'un affichage ou d'une impression. Un code source est donné dans l'annexe 8.7.1.

Bien sûr, on ne considére que des graphes sans boucle, c'est-à-dire sans arête allant d'un nœud à lui-même, car sinon le problème est évidemment impossible.

D'ailleurs je vous invite à effectivement tenter de résoudre de tels graphes aléatoires, en explicitant les stratégies mises en œuvre. En effet, fort probablement, vous vous apercevrez, dans le chapitre sur les méthodes de résolution, que nombre d'entre elles ne sont que des formalisations inavouées de stratégies intuitives.

La figure 1.3.1 montre quelques exemples de graphes à huit nœuds et leur coloriage le plus économique en nombre de couleurs.

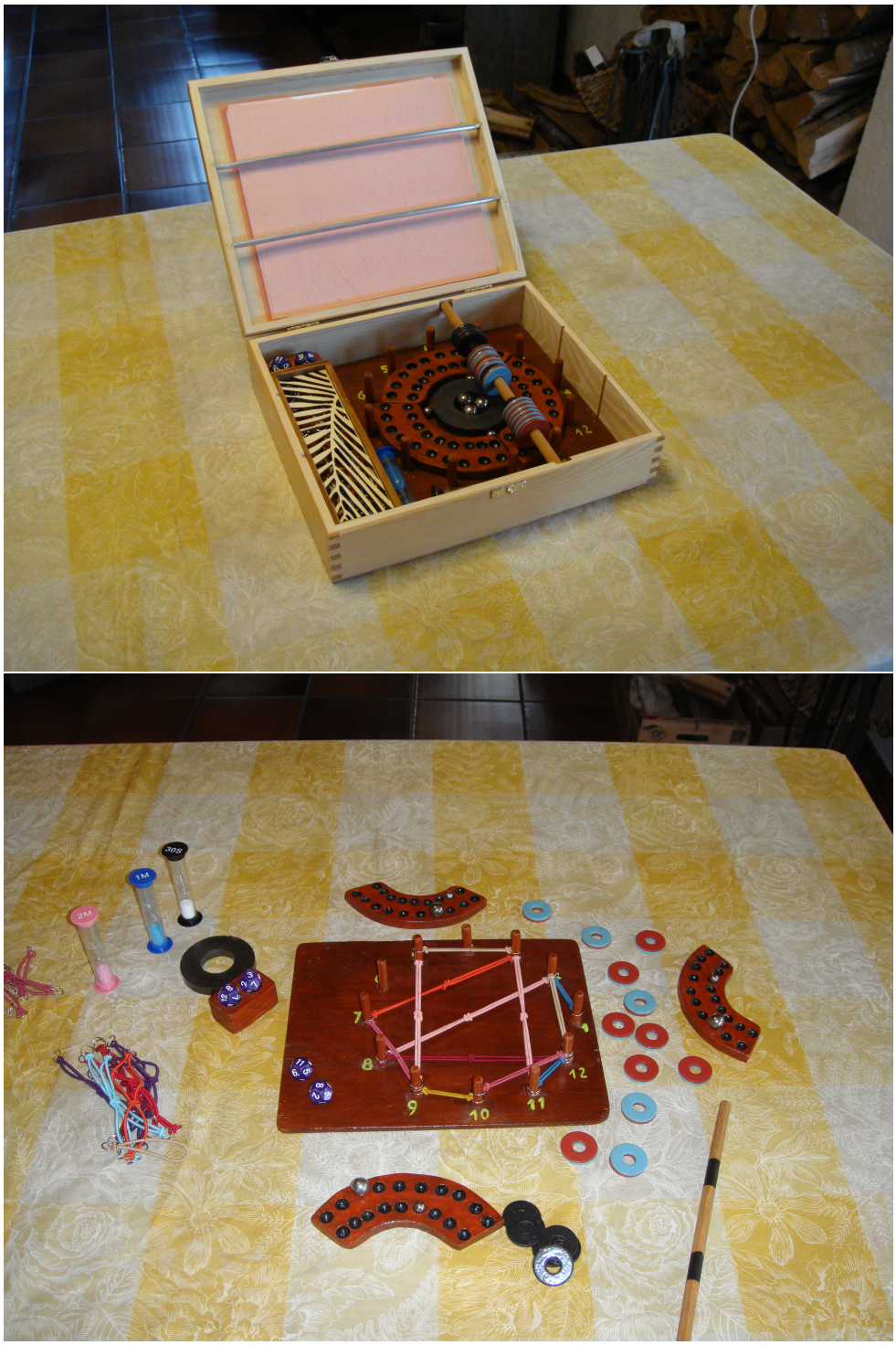
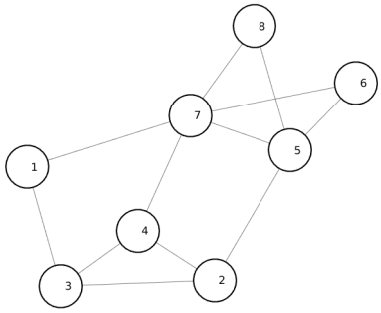
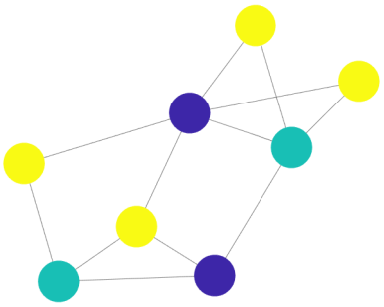
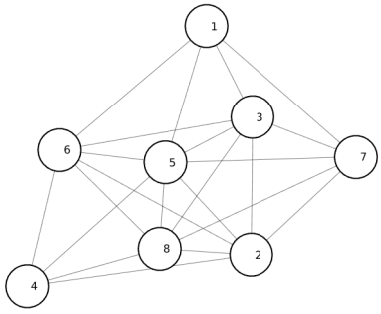


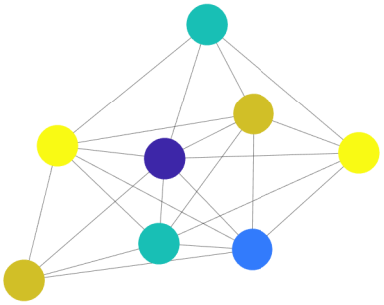
Figure 1.2.5: Colorigraphe. Boite ouverte et une partie en cours.





(a) Douze arêtes. Le graphe est planaire et ne nécessite que trois couleurs.





(b) Avec vingt-deux arêtes, il faut ici cinq couleurs.

Figure 1.3.1: Exemples de graphes aléatoires à huit nœuds.

Chapitre 2

Quelques applications

Toute affectation de ressources avec des contraintes d'exclusions mutuelles (planning de cours, partage de bandes de fréquence, ordonnancement de tâches devant utiliser une même machine, etc.) peut être modélisée par un graphe et résolue par un coloriage valide, c'est-à-dire que pour chaque arête les nœuds extrémités ont des couleurs différentes. Il y a toujours un tel coloriage, mais le but est aussi d'utiliser le moins de ressources (c'est-à-dire de couleurs) possibles.

Voyons quelques exemples simples, essentiellement ludiques, juste à titre d'illustration. Le lecteur intéressé en trouvera facilement de nombreux autres plus conséquents dans la littérature (**(author?)** [4], **(author?)** [27],**(author?)** [21]).

2.1 Covoiturage

Alice, Bob, Samuel, Sara et Nicolas doivent aller à une même réunion. Un covoiturage serait souhaitable mais Bob déteste tout le monde, Samuel ne supporte ni Alice ni Nicolas, qui, lui-même ne s'entend pas avec Sara. Combien faut-il de véhicules au minimum ?

Dessinons donc un graphe dont les sommets sont nos cinq personnages et traçons des arêtes entre les individus incompatibles. On voit alors qu'un coloriage valide nécessite trois couleurs. Rouge (de rage ?) pour Bob, vert pour Alice et Nicolas et bleu pour Sara et Samuel. Donc trois véhicules sont nécessaires et suffisent. Remarquez que Bob le misanthrope doit prendre une voiture pour lui tout seul. Et s'il n'y a que deux voitures de disponibles ? Un compromis sera à trouver et ce problème plus contraint est évoqué dans le chapitre 7 sur les algorithmes diplomates.

Bien sûr, ici, une solution peut facilement être trouvée « manuellement ». Mais dès que le problème se complique, il faut appliquer des algorithmes de résolution plus ou moins sophistiqués. Les plus simples à coder sont les *algorithmes gloutons* (voir la section 4.2). Très rapides et garantissant un coloriage valide dans tous les cas de figure, mais rarement le meilleur et même

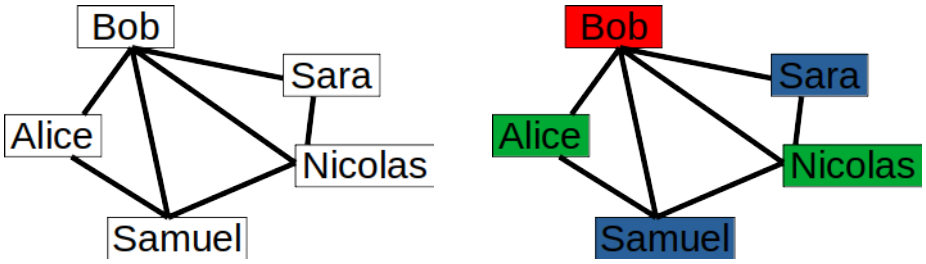


Figure 2.1.1: Les arêtes du graphe représentent les incompatibilités. Un tricoloriage est possible ; il faudra trois voitures, dont une juste pour Bob.

quelquefois le pire possible.

Donc nous verrons aussi d'autres méthodes, soit avec résultat optimum garanti, mais lentes, voire inutilisables en pratique sur de grands graphes, soit stochastiques, avec un résultat non garanti, comme les gloutons, mais quand même en général plutôt bon.

2.2 Sudoku

Pour illustrer l'utilisation du coloriage de graphe, considérons le petit sudoku 4x4. La figure 2.2.1 indique la numérotation des cases et le graphe qui en résulte. Une arête indique que les cases extrémités ne peuvent contenir le même nombre. Par exemple les cases 1, 2, 3 et 4 ne peuvent contenir que des nombres différents. De même pour les case 1, 2, 5 et 6, etc.

La recherche d'un coloriage minimal par le plus simple des algorithmes que nous verrons plus loin (Glouton 0, à la section 4.2.1) propose une solution à quatre couleurs, qui, codées de 1 à 4, résolvent le problème (voir la figure 2.2.2).

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

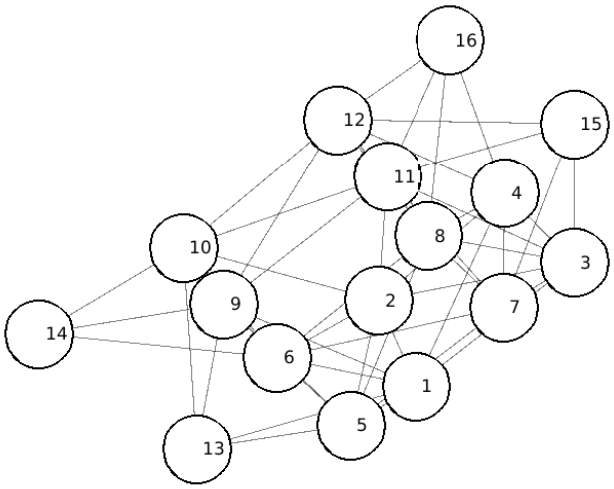


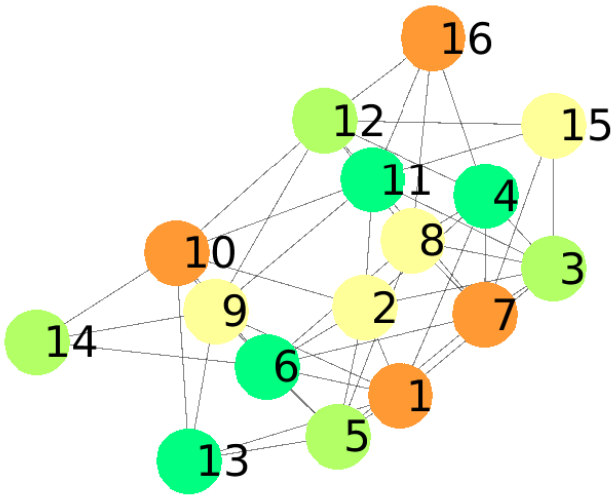
Figure 2.2.1: Sudoku - Numérotation des cases et graphe correspondant

2.3 Salles de réunion

Un certain nombre de réunions, disons huit, doivent se tenir avec des horaires de début et de fin donnant lieu à des chevauchements. Combien faut-il de salles ?

On peut d'abord former un simple tableau avec le temps en abscisse, qui fait apparaître les chevauchements temporels des réunions. Pour construire le graphe, les nœuds sont les réunions et si deux réunions se chevauchent on trace une arête entre elles (figure2.3.1). La résolution du graphe nous indique que trois salles sont nécessaires (figure 2.3.2).

Notons qu'il s'agit ici d'un graphe d'intervalles. Pour ce type de graphe une solution peut toujours être trouvée en temps polynomial même sur un ordinateur classique (non quantique).



1	2	3	4
3	4	1	2
2	1	4	3
4	3	2	1

Figure 2.2.2: Sudoku - Coloriage solution

	8h	9h	10h	11h	12h	13h	14h
Réunion 1	x	x					
Réunion 2		x	x	x			
Réunion 3	x						
Réunion 4			x	x	x		
Réunion 5		x	x				
Réunion 6					x	x	x

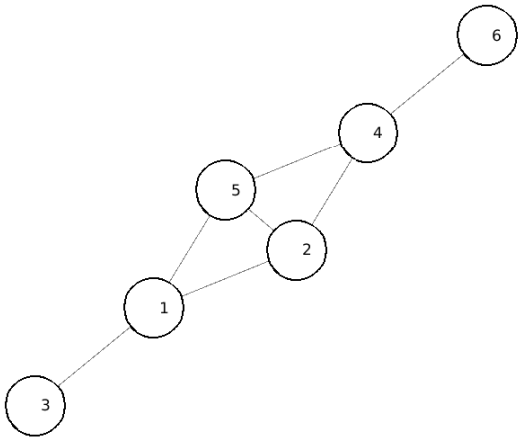
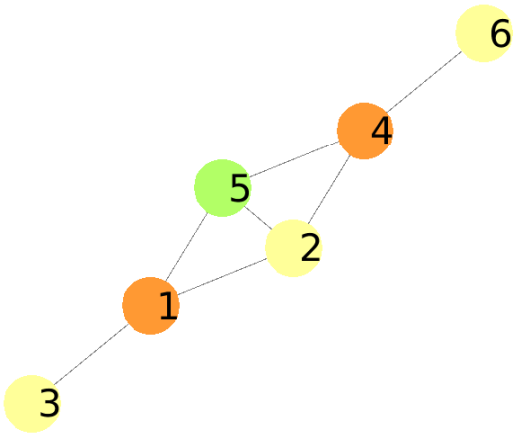


Figure 2.3.1: Réunions - Diagramme temporel et graphe résultant.



	8h	9h	10h	11h	12h	13h	14h
Réunion 1							
Réunion 2							
Réunion 3							
Réunion 4							
Réunion 5							
Réunion 6							

Figure 2.3.2: Réunions - Graphe résolu et affectation des trois salles.

Chapitre 3

Codages

Pour écrire des programmes de résolution à faire exécuter sur ordinateur, il est nécessaire de savoir représenter les diverses entités du problème à résoudre selon des codes exploitables par la machine (et le langage utilisé). Le plus souvent on s'appuie sur des codages binaires ou plus généralement entiers.

3.1 Définitions

Nous avons vu la définition d'un coloriage valide, que l'on peut réécrire comme suit :

Un coloriage est *valide* si l'on n'a jamais deux nœuds adjacents de la même couleur.

Nous aurons besoin d'une autre notion de degré :

Le *degré* d'un coloriage est le nombre de couleurs différentes utilisées. On le notera souvent K. À ne pas confondre avec les degrés des nœuds.

À partir de là, on peut préciser ce que vont rechercher les algorithmes présentés par la suite :

Un coloriage est *optimum* s'il est valide de degré K et s'il n'en existe pas de valide de degré inférieur.

Pour simplifier les présentations et les algorithmes, on convient de représenter les couleurs par des entiers. D'où une autre définition :

Un codage entier de coloriage est *admissible* s'il contient au moins une fois chaque entier de 1 à K et aucun autre. Par la suite et pour simplifier nous utiliserons souvent l'expression métonymique *coloriage admissible*, en particulier dans l'annexe 8. Évidemment elle ne qualifie pas le coloriage lui-même, mais bien le codage entier choisi parmi l'infinité des possibles (voir ci-dessous les coloriages équivalents).

Comme les couleurs sont arbitraires, sous certaines conditions, deux coloriages peuvent être en fait deux représentations du même objet :

Deux coloriages sont *équivalents* s'ils sont de même degré et si tous les nœuds d'une même couleur dans l'un sont également de même couleur dans l'autre.

Table 3.1: Deux coloriages équivalents

Alice	vert	rouge
Bob	rouge	bleu
Nicolas	vert	rouge
Samuel	bleu	vert
Sara	bleu	vert

Par exemple, pour le graphe 2.1.1 les deux coloriages de la table 3.1 sont équivalents.

Sauf mention contraire, tous les coloriages considérés ci-après seront supposés admissibles et on notera que pour tout coloriage il en existe un équivalent minimal, dont le code couleur entier soit le plus petit possible (voir la section 3.2.2). Ainsi le coloriage (3,1,2,5) peut être remplacé par (3, 1, 2, 4). Un code source Matlab© permettant ce « compactage » est donné en annexe 8.7.2.

3.2 Coder un graphe

Soit un graphe à N nœuds. Nous avons déjà vu qu'il était pratique et assez naturel de les numéroter de 1 à N. Mais il faut aussi indiquer les arêtes. Dans le cas le plus général les graphes sont *orientés* : chaque arête est représentée par une flèche et ne peut dès lors être parcourue que dans le sens de celle-ci. Il peut donc y en avoir une du nœud i au nœud j, mais pas forcément dans l'autre sens. Cependant on ne s'intéresse ici qu'aux graphes non orientés (bidirectionnels) eet sans boucle (pas d'arête d'un nœud vers lui-même).

3.2.1 Codages binaires

Le graphe peut être représenté par une matrice G de dimension $N \times N$, dite d'*adjacence*, telle que $G(i,j) = 1$ si l'arête i-j existe (ou l'arête $j-i$) et $G(i,j) = 0$ sinon. Comme le graphe est non orienté, la matrice est symétrique. Par exemple le graphe à cinq nœuds de la figure 3.2.1 a pour matrice

$$G_{adj} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Du point de vue stockage en mémoire, et comme la diagonale est nulle, la symétrie permet de n'utiliser que $N(N-1)/2$ bits, par exemple ceux du triangle supérieur droit.

Lorsque le graphe a peu d'arêtes comparativement au nombre de nœuds, la matrice est éparse, ne contenant pratiquement que des zéros. Un codage

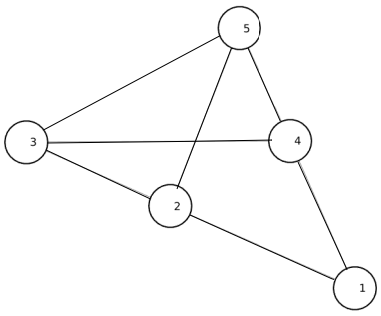


Figure 3.2.1: Graphe 5 nœuds 6 arêtes

condensé peut alors être intéressant. Les séquences de zéros consécutifs sont remplacées par leur nombre.

Un autre codage binaire consiste en une matrice d'*incidence*, parfois appelée matrice nœuds-arêtes ou sommets-arêtes. Les arcs sont supposés numérotés de 1 à A. La matrice est alors de taille $N \times A$. L'élément (i, j) vaut 1 si et seulement si le nœud i est une des extrémités de l'arc j et 0 sinon. Le graphe de la figure 3.2.1, en supposant les arêtes numérotées en ordre lexicographique (soit $(1,2) \to 1, (1,4) \to 2, (2,3) \to 3, (3,5) \to 4, (4,3) \to 5, (4,5) \to 6$), donne comme matrice d'incidence :

$$G_{incid} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}$$

Par exemple la colonne 3 de la matrice nous dit que les nœuds 2 et 3 sont reliés. Une propriété évidente est que chaque colonne contient exactement deux 1. Là encore un codage condensé peut être utile.

Cependant, contrairement à la matrice d'adjacence, la matrice d'incidence ne décrit pas à elle seule entièrement le graphe : il faut connaître aussi les codages entiers des arêtes. Par exemple, dans l'exemple ci-dessus, il faut savoir que l'arête 2 est $(1,4)$ et non $(1,3)$.

On peut pallier ce défaut, mais au prix d'une augmentation considérable de la taille de la matrice, en utilisant $N(N-1)/2$ colonnes, une par arête, existante ou non (voir la section 3.4.3).

Plus généralement, tous les codages binaires, s'ils sont faciles à manipuler et même quasiment indispensables pour des calculs quantiques, sont peu économiques en taille mémoire. Le codage entier et la liste explicite des arêtes, que nous allons voir maintenant, sont beaucoup plus concis.

3.2.2 Codage entier

On peut considérer la liste des $N(N-1)/2$ éléments 0 ou 1 de la matrice G_{adj} comme étant l'écriture binaire d'un entier. Avec notre exemple : 1010101111, soit 687 en base 10.

On peut ainsi définir, pour chaque N, « l'ensemble des entiers qui codent les graphes connexes non orientés sans boucle d'ordre N ». Ces ensembles ont des propriétés intéressantes non triviales, à commencer par leur taille[1] qui augmente très vite avec N : 1, 1, 4, 38, 728, 26704, 1866256, etc. Sur cet ensemble on peut estimer l'efficacité moyenne d'un algorithme de recherche de coloriage optimum (voir plus loin la section Comparaisons 4.2.10).

3.2.3 Liste explicite des arêtes

Le plus intuitif est d'indiquer les arêtes comme paires de nœuds. Ainsi le graphe de la figure 3.2.1 est représenté par l'ensemble de paires suivant

$$\{(1,2),(2,3),(3,4),(3,5),(4,5),(2,5),(4,1)\}$$

Comme il s'agit d'un ensemble, l'ordre de présentation des paires n'importe pas. Et comme le graphe est non orienté, l'ordre dans chaque paire non plus.

Cependant, on peut utiliser un codage plus compact en admettant que toutes les paires de nœuds sont en ordre lexicographique. Alors il suffit de donner les rangs dans cette liste des paires correspondant à une arête existante. Dans notre exemple, il y a dix paires, dont six correspondent à une arête existante

$$\{(\mathbf{1},\mathbf{2}),(1,3),(\mathbf{1},\mathbf{4}),(1,5),(\mathbf{2},\mathbf{3}),(2,4),(\mathbf{2},\mathbf{5}),(\mathbf{3},\mathbf{4}),(\mathbf{3},\mathbf{5}),(\mathbf{4},\mathbf{5})\}$$

et le codage est alors

$$\{1,3,5,7,8,9,10\}$$

(car l'arête $(4,1)$ est aussi l'arête $(1,4)$). Ce codage est plus économique en espace mémoire, mais nous ne l'utiliserons pas ici, car plus difficile à interpréter pour une présentation.

3.3 Coder un coloriage

Bien évidemment, un coloriage nécessite au plus N codes. Par convention, même si l'on parle de couleurs, on utilise alors des nombres entiers de $I_N = \{1,\ldots,N\}$.

[1] Formule récursive $T_N = 2^{\frac{N(N-1)}{2}} - \sum_{k=1}^{N-1} \left(\begin{array}{c} N-1 \\ k-1 \end{array} \right) T_k 2^{\frac{(N-k)(N-k-1)}{2}}$

3.3.1 Codages naturels

Comme les nœuds sont aussi supposés être numérotés de 1 à N, le codage peut être simplement une liste d'entiers tirés de I_N (en général avec des répétitions) avec la convention que cette liste est dans l'ordre des numéros des nœuds. Par exemple le coloriage du graphe de la figure 3.3.1a est $(1, 2, 1, 2, 1)$et $(1, 2, 2, 1, 2)$ pour celui de la figure 3.3.1b. Notons au passage que l'on peut toujours imposer que la couleur du nœud 1 soit également 1. Plus généralement, s'il y a K couleurs, on peut toujours s'arranger pour que le codage ne comprenne que des chiffres au plus égaux à K. Par exemple le coloriage (1, 3, 5, 3) d'un graphe à quatre nœuds est à trois couleurs et peut être remplacé par l'équivalent (1, 2, 3, 2). C'est ce que nous supposons maintenant.

Nous verrons alors que des manipulations purement algébriques sur les codages du graphe et du coloriage permettent de dire si ce dernier est valide ou non.

Cependant, pour certains algorithmes de recherche de coloriage (voir 4.2.11.2), il est préférable d'utiliser un codage par un entier unique. Soit $(c_1, \ldots, c_N)$ le codage selon la méthode ci-dessus. Par définition un c_n est au plus égal à N. On peut donc considérer, dans la base $N + 1$, l'entier $code\,(C) = c_1 c_2 \ldots c_N$. Ainsi

$$code\,(C) = \sum_{n=1}^{N} c_n\,(N+1)^{n-1} \tag{3.3.1}$$

Nous définissons ainsi une relation d'ordre sur l'ensemble des coloriages admissibles. Notons d'emblée qu'une borne inférieure $code_{inf}$ est pour tous les c_n égaux à 1 (tous les nœuds de même couleur, même si le coloriage est forcément invalide) et la valeur maximale $code_{max}$ est pour $c_n = N - n + 1$ pour le graphe complet[2]. Avec cette relation d'ordre le plus petit coloriage valide est optimal.

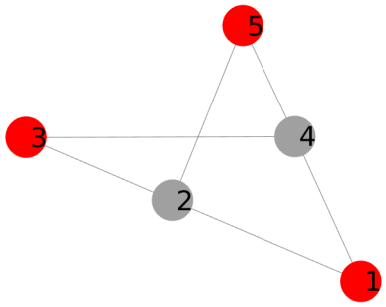
Ce type de codage entretient des liens formels étroits avec ce que l'on appelle les entiers *pannumériques* (voir l'annexe 8.1).

3.3.2 Codage binaire

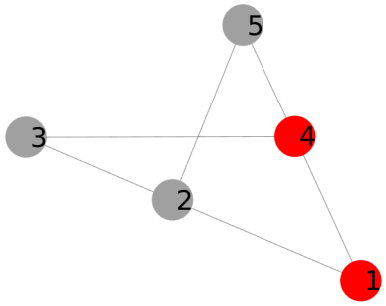
Pour certaines autres méthodes, en particulier la recherche par programmation linéaire (4.2.11) et l'approche quantique (6) il est utile d'avoir une représentation purement binaire, même si beaucoup moins compacte. Supposons que les nœuds du graphe aient K couleurs numérotées 1, 2, ..., K. On définit alors la matrice de coloriage C de dimension $N \times K$ telle qu'en chaque ligne n il y ait un seul élément égal à 1, en colonne k si la couleur du nœud n est k. Tous les autres éléments de la ligne sont nuls.

Ainsi le codage binaire du coloriage de la figure 3.3.1a est

[2]Rappelons qu'un graphe bi-directionnel sans boucle est dit « complet » s'il possède tous les $\frac{N(N-1)}{2}$ arêtes possibles.



(a) Un coloriage valide, de codage (1, 2, 1, 2, 1).



(b) Un coloriage invalide, de codage (1, 2, 2, 1, 2)

Figure 3.3.1: Deux coloriages d'un graphe 5 nœuds 5 arêtes.

$$C = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Ce code peut être représenté sous forme d'une chaine de bits, en concaténant les lignes. L'exemple ci-dessus donne 1001100110. Il suffit de connaître N (ou K) pour recréer la matrice.

Remarque
Lorsqu'il s'agit de présenter sous forme binaire des coloriages trouvés par un algorithme, la contrainte « un seul 1 par ligne » peut être négligée. Considérons par exemple les deux coloriages de la figure 3.3.2. Ils peuvent être représentés par la seule matrice

$$C = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

dont l'interprétation est que le nœud 1 peut avoir au choix la couleur 1 ou la couleur 3.

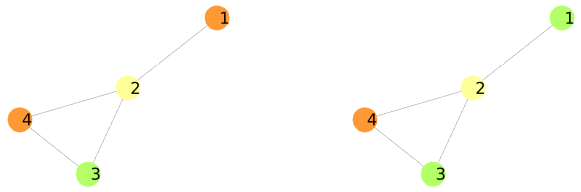


Figure 3.3.2: Deux coloriages valides dont les codages binaires peuvent être fusionnés en une seule matrice.

3.4 Coder un graphe colorié

Jusqu'ici nous avons codé séparément le graphe G et le coloriage C. Cependant il peut être intéressant d'avoir un codage qui incorpore toutes les informations : la structure du graphe et son coloriage. Ce type de codage est simplement présenté ici, sans être exploité ultérieurement.

3.4.1 Utilisation de la diagonale

Une première méthode consiste simplement à partir de la matrice d'adjacence du graphe et à la compléter par le codage en entiers de C. Ceux-ci peuvent

être mis sur la diagonale ou, avec redondance, sur les valeurs 1 de la matrice. Reprenons l'exemple du graphe à cinq nœuds de la figure 3.3.1. Avec le coloriage (1, 2, 2, 1, 2) la matrice devient

$$G_C = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 \\ 1 & 2 & 1 & 0 & 1 \\ 0 & 1 & 2 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 2 \end{bmatrix}$$

3.4.2 Couleurs d'extrémités

Une seconde méthode consiste à simplement indiquer la couleur de l'extrémité de chaque arête. Ce qui donne

$$G_C = \begin{bmatrix} 0 & 2 & 0 & 1 & 0 \\ 1 & 0 & 2 & 0 & 2 \\ 0 & 2 & 0 & 1 & 0 \\ 1 & 0 & 2 & 0 & 2 \\ 0 & 2 & 0 & 1 & 0 \end{bmatrix}$$

La matrice se lit alors ainsi

- l'extrémité de l'arête $1 \to 2$ est de couleur 2
- l'extrémité de l'arête $1 \to 4$ est de couleur 2
- l'extrémité de l'arête $2 \to 1$ est de couleur 1
- l'extrémité de l'arête $2 \to 3$ est de couleur 2

etc.

3.4.3 Codage binaire selon les arêtes

Dans cette méthode, on construit une chaine de bits en considérant successivement chaque arête (ordre lexicographique). Si l'arête est valide (extrémités de couleurs différentes) on ajoute 1, sinon on ajoute 0.

Le résultat ne contient en fait pas toute l'information du couple (G, C), car on ne peut en déduire que le nombre d'arêtes, égal à la longueur de la chaine, mais le code obtenu peut être utile lors d'une recherche (voir la section 4.1.3).

Dans notre exemple, le code obtenu est 100011, car
arête (1,2), valide, donne 1
arête (1,4) invalide, donne 0,
etc.

Chapitre 4

Résolutions déterministes

Comme évoqué précédemment, nous allons présenter plusieurs types de méthodes de recherche d'un coloriage optimal, plus précisément :

- déterministes sans garantie ;
- garanties (donc forcément déterministes) ;
- quantiques ;
- heuristiques et métaheuristiques (intrinsèquement stochastiques), mais très succinctement.

Toutes ont des avantages et des inconvénients et ne sont pas utilisées dans les mêmes circonstances. Par exemple si le temps calcul n'est pas un problème, autant choisir une méthode garantie, même si très lente. À l'inverse si le budget de temps calcul est très réduit, une méthode déterministe sans garantie s'impose. Les heuristiques et métaheuristiques sont des compromis entre ces deux extrêmes et souvent très efficaces, mais, comme précisé en introduction, il en sera peu parlé ici.

Rappelons enfin que nous ne nous intéressons ici qu'aux méthodes généralistes, valables pour tout type de graphe car n'exploitant pas d'éventuelles particularités structurelles (planéité, arborescence, cycles, etc.), qu'elles soient connues à l'avance ou détectées par un prétraitement. Le seul bémol est que certains algorithmes utilisent quand même les degrés des nœuds.

4.1 Difficulté du problème et paysages

Pour un graphe à N nœuds le nombre total de coloriages possibles est N^N. C'est la taille de l'espace de recherche, qui augmente donc très rapidement avec celle du graphe. De plus la proportion de coloriages admissibles, elle, diminue avec la taille du graphe (voir l'annexe 8.3). Les coloriages admissibles n'étant même pas tous valides, on voit que résoudre des problèmes de taille croissante revient à chercher des aiguilles de plus en plus dispersées dans une meule de foin de plus en plus imposante.

On peut considérer le problème de recherche d'un coloriage minimum comme un problème d'optimisation. Pour cela, il suffit d'être capable d'affecter à chaque point de l'espace de recherche — chaque coloriage possible — une valeur, de façon que le coloriage cherché ait la plus petite possible. On peut même se contenter de définir une relation d'ordre entre les coloriages et de pouvoir dire que tel coloriage est « inférieur » à tel autre. C'est ce qui est fait par exemple pour la recherche séquentielle de la section 4.2.11.2.

Si l'on affecte des valeurs, on peut visualiser le « paysage » du problème, du moins des sections 2D ou 3D. Faisons-le ici, en calculant la valeur d'un coloriage C d'un graphe G à N nœuds de la manière suivante :

- Soit $f_{1,G}(C)$ le nombre d'arêtes dont les extrémités sont de même couleur. La valeur maximale est $\frac{N(N-1)}{2}$.
- Soit $f_2(C)$ le nombre de couleurs différentes utilisées. La valeur maximale est N.
- La valeur du coloriage est calculée par $f_G(C) = f_{1,G}(C) + \alpha f_2(C)$.

On souhaite que, pour deux coloriages C_1 et C_2:

- Si $f_{1,G}(C_1) = 0$ et $f_{1,G}(C_2) > 0$ alors $f_G(C_1) < f_G(C_2)$. Autrement dit un coloriage valide est toujours meilleur qu'un invalide, même s'il utilise plus de couleurs.
- Si $f_{1,G}(C_1) = f_{1,G}(C_2)$, alors c'est le nombre de couleurs qui prime.

Il suffit pour cela d'avoir $\alpha < \frac{1}{N-1}$, par exemple $\alpha = \frac{1}{N}$.

4.1.1 Paysage en dimension N

Considérons alors le petit graphe de la figure 4.1.1 et construisons le paysage de notre fonction f sur l'espace de recherche des coloriages. Il est de dimension quatre, donc difficile à représenter, mais la figure 4.1.2 en donne deux sections 3D, pour lesquelles deux couleurs ont été fixées. Bien que l'espace de recherche contienne déjà 256 points, on constate que le paysage présente certaines régularités. Ceci est dû au fait que la proposition « plus près c'est mieux (en probabilité) » est vraie même pour des problèmes *a priori* combinatoires comme ici (voir **(author?)** [9] et la discussion sur les problèmes à corrélation positive dans **(author?)** [10]).

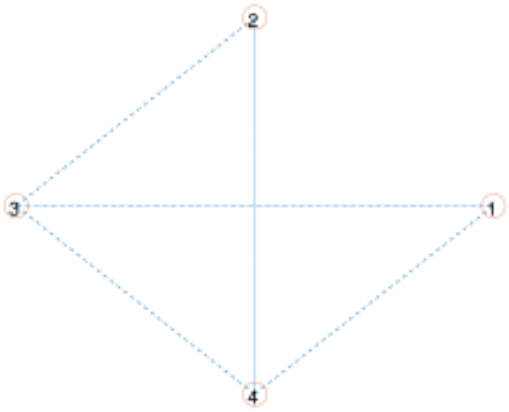


Figure 4.1.1: Graphe 4 nœuds, 5 arêtes.

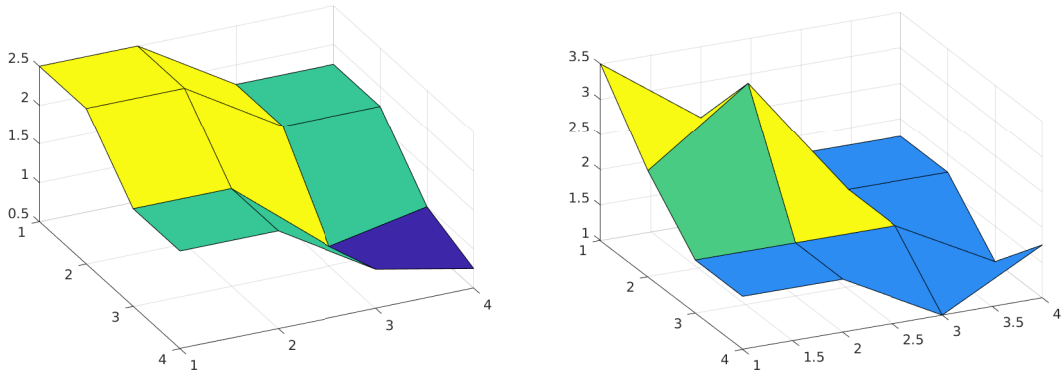


Figure 4.1.2: Paysage des valeurs de coloriages pour le graphe 4.1.1. Deux sections transversales. La première contient d'ailleurs la valeur minimale 0,75 (coloriage valide à trois couleurs).

Les surfaces présentées sur la figure α4.1.2 sont formées d'interpolations linéaires par morceaux entre les points de positions entières. Elles suggèrent qu'il est possible de coder les couleurs avec des valeurs réelles continues (à l'approximation près de l'ordinateur utilisé) et de faire travailler l'optimiseur sur une fonction elle-même continue. Ceci sort du cadre de cette étude, mais en notant $c_i = C(i)$, A le nombre d'arêtes existantes et $\bar{A} = \frac{N(N-1)}{2} - A$ le nombre d'arêtes inexistantes, la fonction suivante est utilisable :

$$f_G(C) = \frac{1}{\bar{A}} \sum_{G(i,j)=0} (|c_i - c_j| > 1) + \frac{\alpha}{A} \sum_{G(i,j)=1} (|c_i - c_j| < 1) \qquad (4.1.1)$$

La première partie compte le nombre de fois que deux nœuds non adjacents ont des valeurs de couleur éloignées et tendra à minimiser le nombre de couleurs. La seconde partie compte le nombre de fois que deux nœuds adjacents ont des valeurs de couleur proches et tendra à minimiser de tels conflits. Le coefficient α permet d'accorder plus ou moins d'importance à ce critère. Une fois un minimum proposé par l'optimiseur il suffira alors de remplacer les valeurs c_i (éventuellement arrondies à l'entier le plus proche) par leurs rangs dans un classement par ordre croissant pour retrouver des valeurs entières codant un coloriage admissible.

Ainsi, sur notre petit exemple, avec $\alpha = 2$, l'algorithme standard d'optimisation par essaim particulaire *SPSO 2007* (*Particle Swarm Central* **(author?)** [24]) trouve un coloriage optimal à trois couleurs en 40 évaluations en moyenne. Pour le graphe cubique (8 nœuds, 12 arêtes) présenté à la section 4.2.1, il lui faut en moyenne 300 évaluations pour trouver un bi-coloriage. Il ne peut certes pas rivaliser avec d'autres méthodes stochastiques sophistiquées, mais le point important est qu'il est possible de faire bien mieux que le hasard pur car les paysages des problèmes de coloriage ne sont pas aussi chaotiques que l'on pourrait croire.

Notre exemple étant vraiment simpliste (graphe planaire), considérons un graphe un peu plus compliqué, comme celui de la figure 4.2.15 : 7 nœuds, 20 arêtes et non planaire.

Pour autant, ainsi que le montrent les deux sections de la figure 4.1.4 le paysage présente bel et bien des régularités exploitables par des algorithmes d'optimisation, même si la présence de plateaux complique la recherche.

4.1.2 Paysage sur monocode

En fait, c'est surtout l'examen des paysages sur l'espace de recherche des codes des coloriages qui peut donner l'impression de chaos. Par codes, on entend ici les valeurs uniques attribuées à chaque coloriage, comme dans la section 3.3.1.

Utilisons une valorisation du coloriage plus simple que celle de la formule 4.1.1. Soit $A_v$le nombre d'arêtes valides, dont les extrémités sont de couleurs différentes.

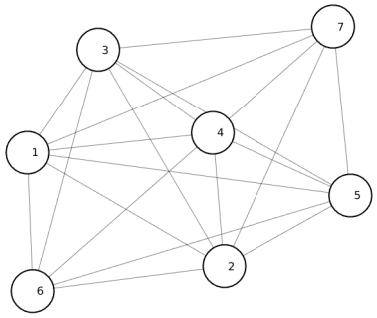


Figure 4.1.3: Graphe à 7 nœuds et 20 arêtes. Il est non planaire (des arêtes se croisent nécessairement).

$$f_G(C) = \frac{A_v}{A} \tag{4.1.2}$$

La figure 4.1.5 (avec interpolations linéaires) montre un exemple pour le graphe cubique. On y voit bien le minimum, mais un algorithme itératif classique aurait du mal à le trouver.

4.1.3 Paysage sur bicodes

Un compromis est de définir le coloriage du graphe à l'aide de deux codes et ainsi d'avoir une visualisation possible en 3D quel que soit N.

Il y a évidemment plusieurs manières de construire un tel bicode. Pour illustrer la méthode considérons-en deux.

4.1.3.1 (code binaire, complément à 1)

Le premier code du coloriage est simplement son code binaire selon ses arêtes, sous forme de chaîne de bits (voir 3.3.2). Le second code est son complément à 1. Par exemple 1001100110 et 0110011001. Ces deux nombres sont transformés en entier (base 10) et, pour faciliter la représentation, on ajoute 1 et on prend les logarithmes (base 2). La valorisation est la même que pour le paysage monocode.

C'est tout-à-fait artificiel et il y a redondance d'information, mais le paysage (avec régressions quadratiques) est plus « sympathique ». La figure 4.1.6 montre un exemple pour le même graphe cubique.

Le minimum devrait être facilement trouvé par un optimiseur itératif classique, à condition de savoir manipuler des « coloriages » avec des valeurs non entières.

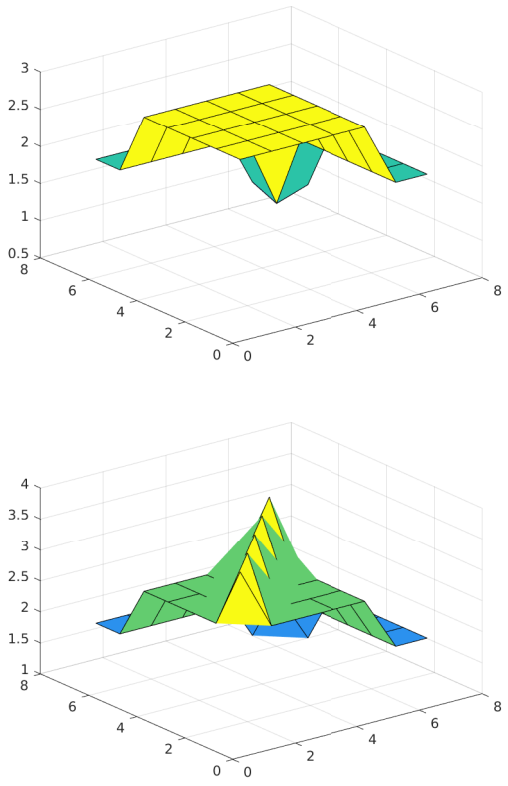


Figure 4.1.4: Graphe à 7 nœuds et 20 arêtes. Deux sections 3D du paysage des coloriages. Des régularités, mais les plateaux compliquent la recherche du minimum.

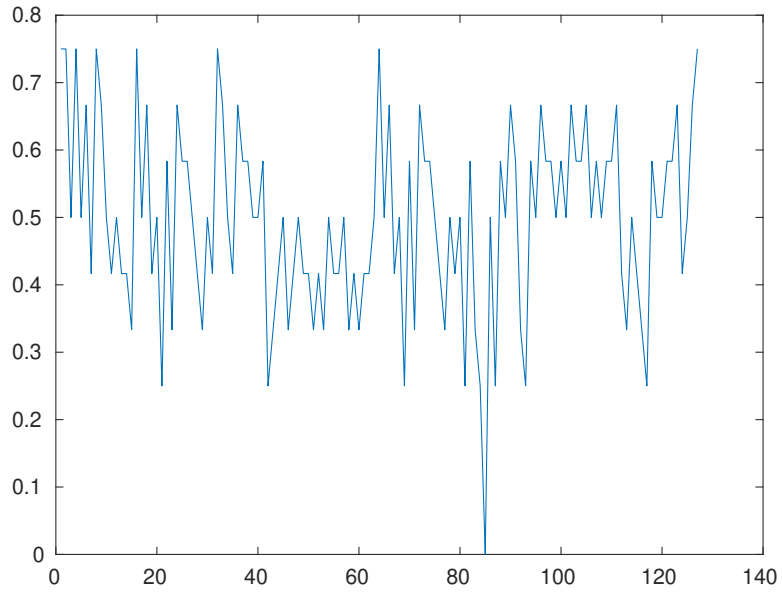
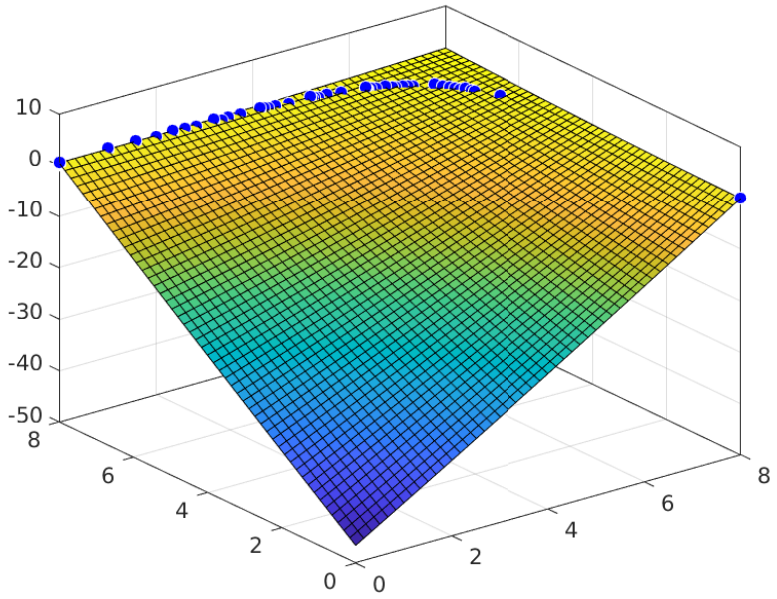


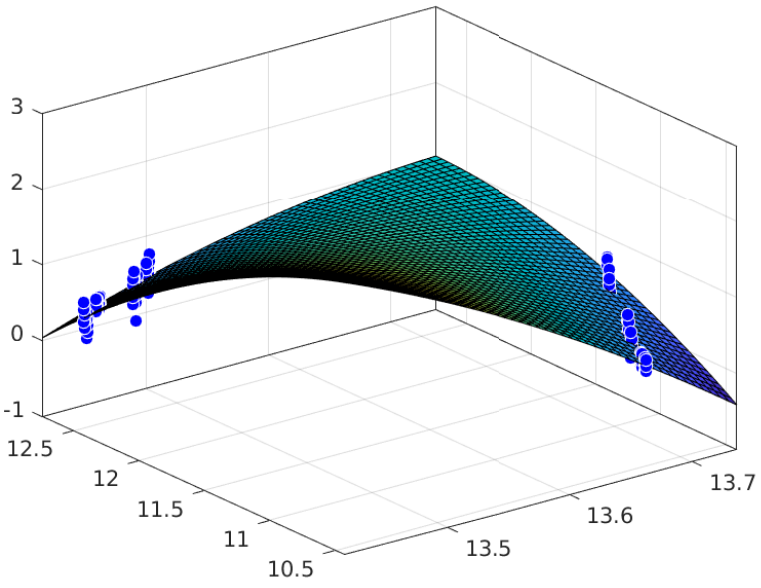
Figure 4.1.5: Graphe cubique. Paysage sur les codes binaires à deux couleurs. Le minimum est difficile à trouver.

4.1.3.2 (bits impairs, bits pairs)

Avec la seconde méthode, tout aussi artificielle, mais sans redondance, on se contente d'extraire les bits de rang impairs pour former le premier code, et ceux de rang pair pour former le second. Par exemple 1001100110 donne $(10101, 01010)$. Cependant, comme le montre la figure, le résultat est moins probant, d'autant qu'un éventuel algorithme devrait alors trouver les deux meilleurs minimas pour reconstruire un coloriage optimal.



(a) Méthode 1 (code, complément à 1)



(b) Méthode 2 (bits de rang impair, bits de rang pair)

Figure 4.1.6: Graphe cubique. Paysages sur des bicodes à deux couleurs.

4.2 Quelques méthodes

L'avantage d'un algorithme déterministe[1] est évidemment qu'il est inutile de l'exécuter plusieurs fois, au contraire des algorithmes stochastiques, heuristiques et métaheuristiques (5). L'inconvénient est que pour les déterministes non garantis, en général très rapides et de complexité faible le résultat peut être assez mauvais, voire très mauvais, sur certains problèmes.

Les déterministes garantis sont tous, pour l'instant, de complexité exponentielle en fonction du nombre de nœuds et d'arêtes[2]. Cependant, dans certains cas, il en existe de complexité polynomiale, par exemple sur les graphes 3-coloriables (algorithme de Wigderson **(author?)** [29]) ou ceux ayant une structure particulière telle qu'un arbre, dont on voit facilement qu'il est toujours 2-coloriable.

Comme déjà vu, on convient de représenter les couleurs par des entiers de $\{1, 2, \ldots, N\}$. Un nœud est dit *vacant* s'il n'est pas colorié. Une classe importante d'algorithmes déterministes non garantis est celle de ceux dits *gloutons*. Leur pseudo-code général est :

Modèle Glouton

```
Tant qu'il existe un nœud vacant
- en choisir un
- le colorier avec une couleur possible
  (c'est-à-dire qui n'est portée par aucun de ses voisins)
```

Notez que ce modèle, est une formalisation de la plupart des stratégies mises en application par des tout-petits dans des jeux comme ceux du chapitre 1 . Pour ce que j'ai pu en constater, les enfants un peu plus grands, disons 5-6 ans, utilisent tous, sans bien sûr les nommer ainsi, des algorithmes avec retour (voir la section 4.2.9).

Comme on peut le voir, quand les algorithmes gloutons affectent une couleur à un nœud ils n'y reviennent pas : pas de repentir au sens artistique du terme. Ainsi le coloriage complet se fait en N itérations. À partir de ce modèle général, les variantes portent sur la manière de choisir, parmi les nœuds vacants, celui qui va être colorié et sur le choix de la couleur qui lui est affectée. En voici quelques-unes (plusieurs codes sources sont donnés à l'annexe 8.7).

4.2.1 Glouton 0

On choisit le nœud vacant de plus faible numéro et on lui affecte la couleur la plus « petite » possible qui n'entre pas en conflit avec celles de ses voisins

[1] Parfois appelé constructif, ce qui est un peu restrictif, car certaines des méthodes déterministes sont parfois amenées à détruire un coloriage partiel pour tenter d'en reconstruire un meilleur.

[2] Écrit en septembre 2022. C'est lié au fameux problème $P \stackrel{?}{=} NP$. Si vous pouvez prouver que la réponse est positive, alors d'une part il existe un algorithme de complexité polynomiale sur ordinateur classique (non quantique) et, d'autre part, vous remportez le prix d'un million de dollars mis en jeu par l'Institut de mathématiques Clay en 2000 ...

déjà coloriés. Selon l'ordre de numérotation des nœuds l'algorithme trouve un coloriage optimal ... ou pas !

Même sur graphe très simple « linéaire » à quatre nœuds l'algorithme peut être mis en échec, comme on le voit sur la figure 4.2.1 montrant l'évolution du coloriage. Deux couleurs suffiraient mais au final il en propose trois.

L'intérêt théorique est qu'il existe toujours au moins une numérotation pour laquelle un coloriage optimal sera bel et bien trouvé. L'inconvénient est cependant que le nombre total de numérotations possibles est $N!$, qui augmente très vite avec N et rend irréaliste la méthode consistant à les essayer toutes. Une stratégie possible consiste cependant, si l'on soupçonne, pour une numérotation donnée, que le coloriage proposé n'est pas optimal, à essayer quand même plusieurs autres numérotations.

Mais on peut aussi construire des algorithmes moins sensibles à cette numérotation.

Autre exemple : le graphe « cubique ». Il est classiquement appelé ainsi, bien qu'en fait il soit planaire, comme le montre la figure 4.2.2.

Il est bi-coloriable mais spécialement conçu pour mettre en défaut certains types d'algorithmes : tous les nœuds ont le même degré et la numérotation soigneusement choisie.

4.2.2 Glouton 1

C'est une légère variante de DSATUR (**(author?)** [7]). Comme Glouton 0, mais les nœuds sont préalablement classés par ordre décroissant de degré. Dans la variante 1a, les nœuds sont classés par ordre croissant de degré, mais globalement c'est moins efficace (voir le tableau 4.1 dans la section Comparaisons).

L'algorithme est moins sensible à la numérotation des nœuds que Glouton 0. Mais il l'est quand même un peu car si plusieurs nœuds ont le même degré, il les sélectionnera successivement selon l'ordre donné par cette numérotation. Pour chaque nœud sélectionné la couleur affectée est la plus petite possible.

Sur le graphe cubique il ne fait évidemment pas mieux que Glouton 0 (quatre couleurs), puisque les nœuds ont tous le même degré.

4.2.3 Glouton 2

On choisit le nœud vacant qui a le plus grand nombre de voisins vacants. La couleur affectée est la plus petite possible. Comme Glouton 1 il est légèrement sensible à la numérotation des nœuds, si plusieurs ont le même nombre maximum de voisins vacants.

Mais pour le graphe cubique il ne propose également que quatre couleurs, encore une fois du fait que tous les nœuds ont le même degré.

4.2.4 Glouton 3

On choisit le nœud vacant auquel on peut affecter la couleur de plus faible numéro. Même risque de sensibilité à la numérotation des nœuds, s'il y a

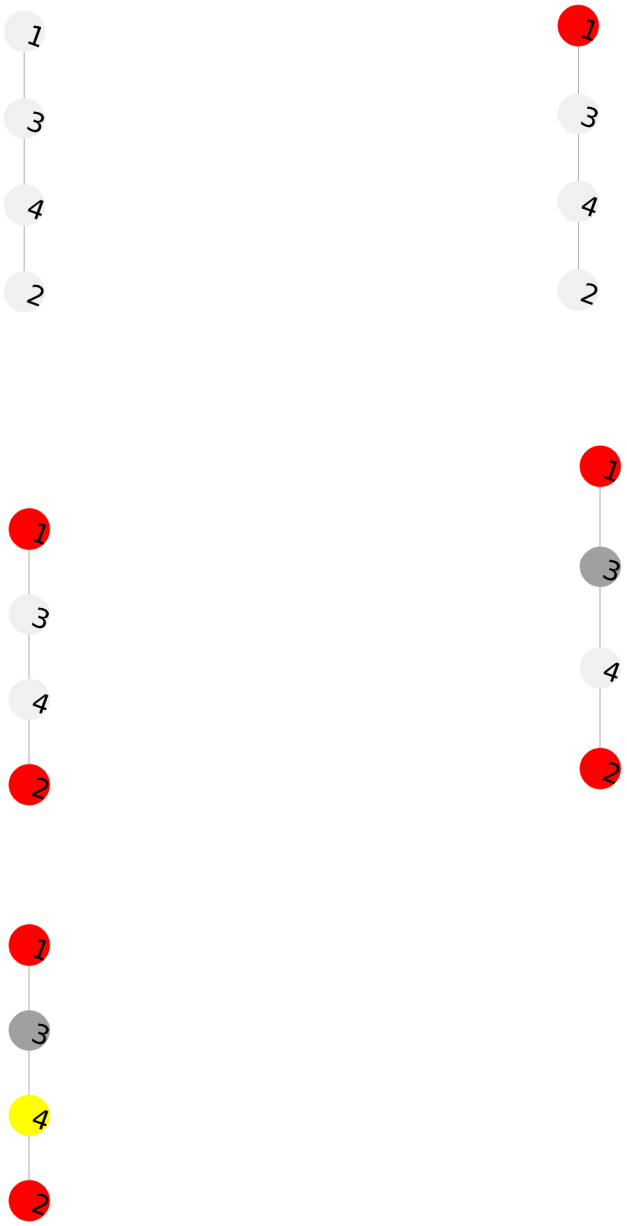


Figure 4.2.1: Glouton 0. Échec sur un graphe à quatre nœuds « mal » numéroté. La solution proposée a trois couleurs mais deux couleurs suffiraient. Ce que trouverait d'ailleurs l'algorithme avec la numérotation de haut en bas (1, 2, 3, 4).

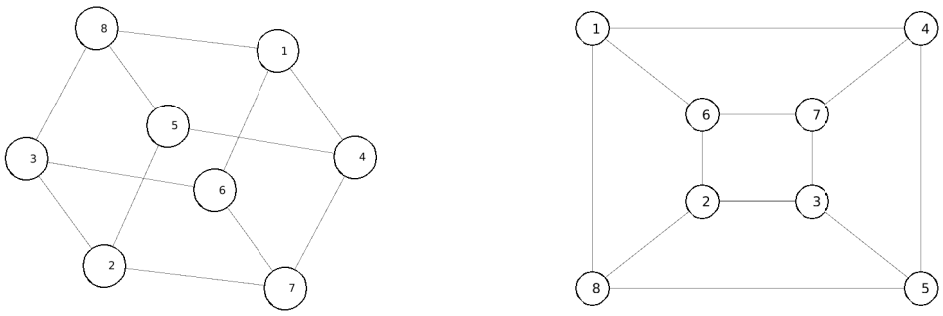


Figure 4.2.2: Graphe cubique et représentation planaire

plusieurs candidats possibles. Néanmoins, sur le graphe cubique il trouve bien un bicoloriage.

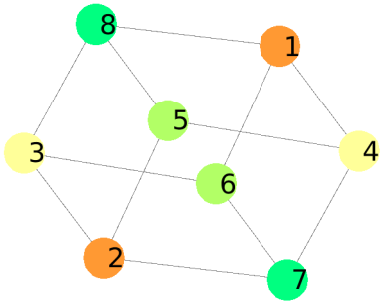
4.2.5 Glouton 4

On considère tous les couples de nœuds (i, j) dont au moins un est vacant. On sélectionne celui dont la longueur du chemin de i à j en nombre d'arêtes est la plus petite. On lui affecte la couleur de plus faible numéro possible. Même remarque que pour les précédents : en cas de plusieurs choix équivalents, la numérotation des nœuds a une influence. Un peu plus compliqué que les précédents et plus gourmand en temps calcul, pour une efficacité pas vraiment meilleure.

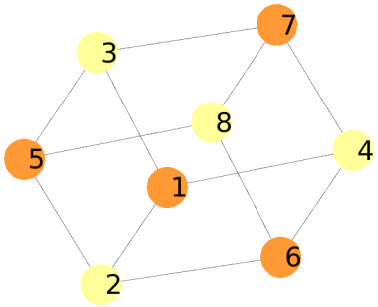
4.2.6 Johnson

Encore un algorithme un peu compliqué mais assez ancien (**(author?)** [18]) et moyennement efficace (voir le tableau 4.1) même si sur le graphe cubique il trouve bien une bicoloration. Il est présenté ici à titre historique. Il fonctionne par réduction itérative du graphe à son sous-graphe non encore colorié.

Cela n'apparaît pas explicitement, mais il est sensible à la numérotation des nœuds, comme Glouton 0 et comme lui il peut être tenu en échec sur des graphes très simples, comme le linéaire à quatre nœuds (voir la figure 4.2.6). Cependant il trouve bien une solution optimale pour le graphe cubique et même pour d'autres plus compliqués.



(a) Numérotation « déroutante ». L'algorithme ne propose qu'une solution à quatre couleurs.



(b) Numérotation « facilitante ». L'algorithme trouve un bicoloriage optimal.

Figure 4.2.3: Glouton 0 sur le graphe cubique

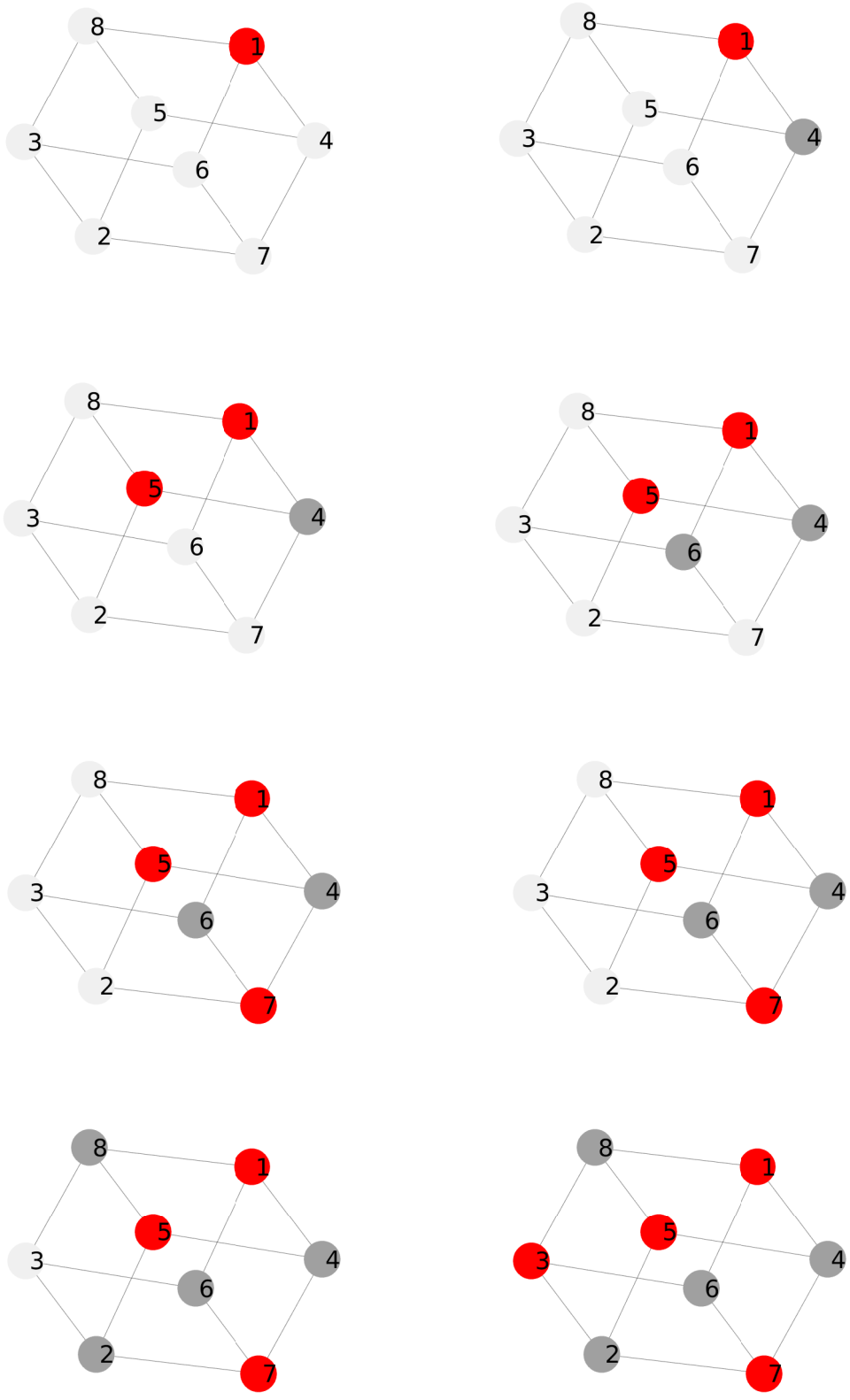


Figure 4.2.4: Sur le graphe cubique Glouton 3 trouve un bicoloriage.

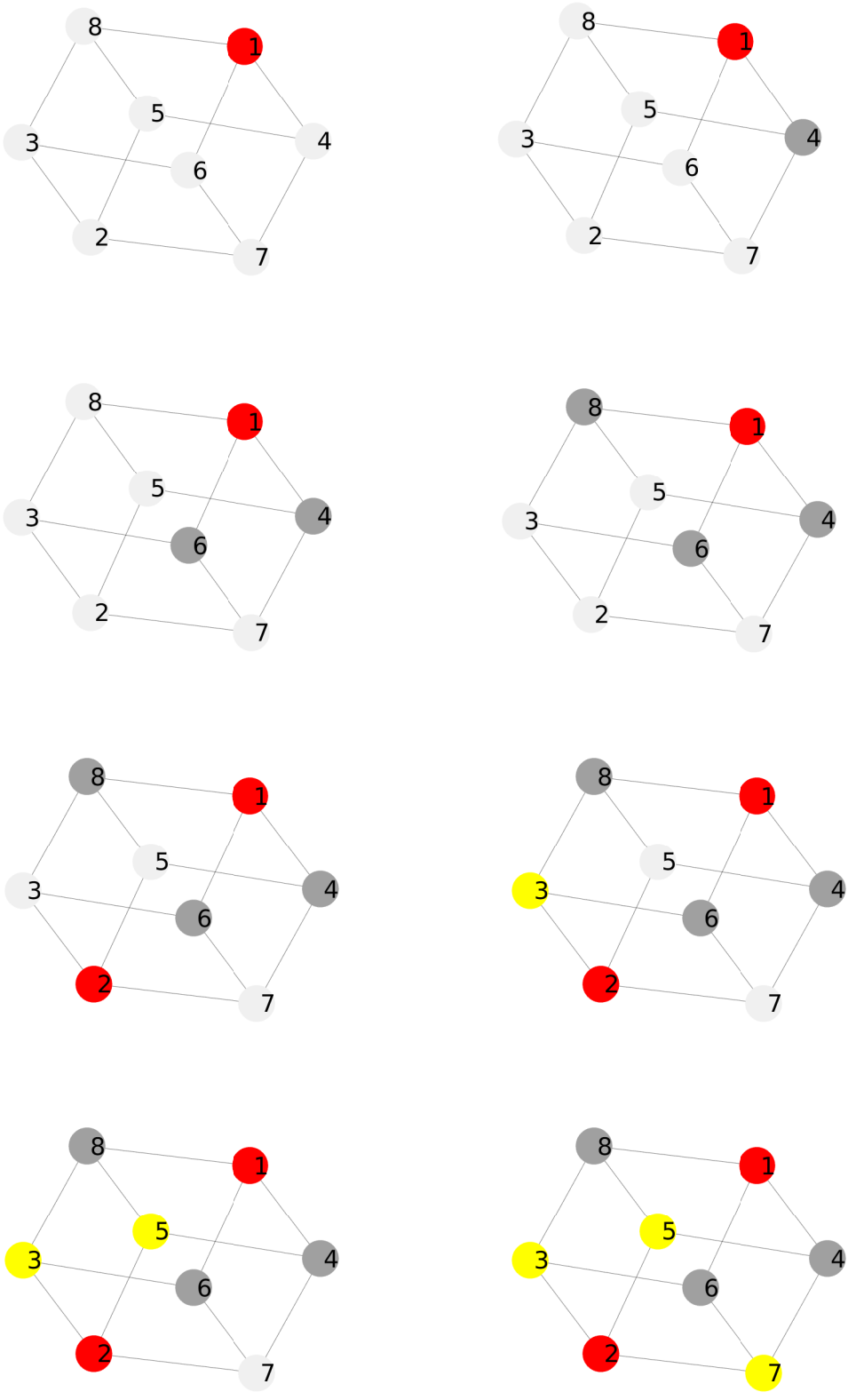
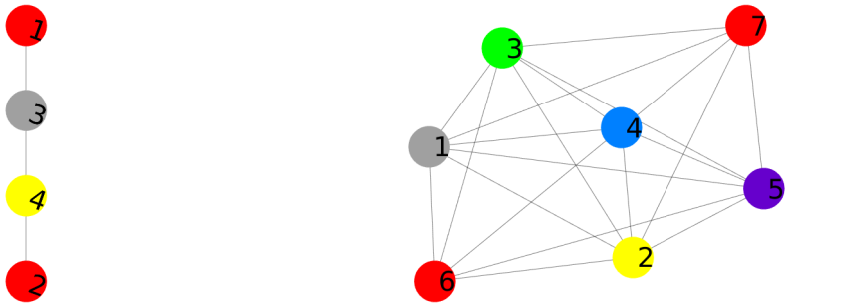


Figure 4.2.5: Sur le graphe cubique Glouton 2 ne trouve qu'un tricoloriage. Remarquez la différence d'avec Glouton 3, à partir de la troisième itération.



(a) Solution non optimale sur le graphe « linéaire ». (b) 6 nœuds, 20 arêtes, solution optimale (6 couleurs).

Figure 4.2.6: Algorithme de Johnson, quelques exemples.

Algorithme de Johnson (pseudo-code)

```
Couleur=0;
Tant qu'il y a des nœuds non coloriés
  Couleur=Couleur+1;
  G_courant=G;
  Tant que G_courant est non vide
G_courant = G - (nœuds coloriés et arêtes incidentes)
Tant que G_courant est non vide
  Trouver son nœud de plus petit degré
  Lui affecter Couleur
  Supprimer ce nœud et ses voisins
```

4.2.7 RLF (*Recursive Large First*)

Cette méthode est sensiblement plus sophistiquée et conçue pour les grands graphes ayant relativement peu d'arêtes (**(author?)** [19]). Sur un petit graphe comme le cubique il ne trouve qu'un 4-coloriage, mais sur un dix nœuds il trouve en général un coloriage optimal.

. Tout d'abord on définit $v_c(i,j)$ comme le nombre de voisins communs à i et à j. Ensuite la boucle itérative est décrite dans l'encadré 4.2.7. Les sous-ensembles de nœuds de même couleur sont progressivement contractés en un seul nœud.

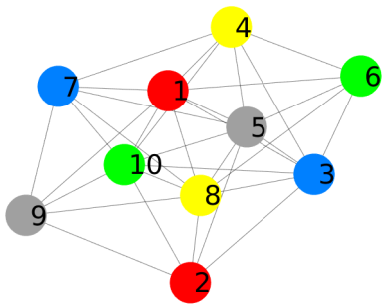


Figure 4.2.7: Sur ce graphe à dix nœuds RLF trouve un coloriage optimal à cinq couleurs.

RLF

1. Choisir un nœud i de degré maximum.
2. Choisir un nœud j non voisin de i tel que $v_c(i,j)$ est maximum et contracter (fusionner) j en i.
3. Répéter 2 jusqu'à ce que i soit voisin de tous les autres nœuds.
4. Enlever le nœud i et reprendre à l'étape 1. Tous les nœuds contractés en i auront la même couleur.

4.2.8 Glouton excentrique

Définissons un algorithme glouton *raisonnable*, même si le terme semble un peu un oxymore :

- à chaque itération il colorie un seul nœud ;
- et avec la plus petite couleur possible (au sens du codage entier) qui n'entre pas en conflit avec celles de ses voisins.

La première condition n'est pas vraiment contraignante puisque l'on peut toujours s'y ramener. Elle permet simplement d'affirmer que pour un graphe de N nœuds un tel algorithme construit toujours un coloriage en N itérations.

Un glouton qui ne serait pas raisonnable (excentrique ? aventureux ?) affecterait par exemple la couleur 3 à un nœud alors que 2 serait possible.

Tous les gloutons vus jusqu'ici sont raisonnables. Il peut donc être instructif d'en définir et étudier un qui ne le soit pas.

Par exemple il suffit de partir de Glouton 0, sauf qu'au lieu d'affecter la plus petite couleur possible, il affecte la plus grande déjà utilisée (sauf par les voisins,

bien sûr). C'est un peu contre-intuitif mais il apparaît qu'il est à peine moins efficace que Glouton 0 (voir le tableau de comparaisons 4.1).

4.2.9 Un algorithme à retours

Jusqu'ici nous n'avons vu que des algorithmes qui coloriaient progressivement le graphe, sans possibilité de revenir en arrière. Mais, bien sûr, on peut construire des algorithmes qui admettent un certain « droit à l'erreur ».

En voici un, dont le principe est le suivant :

- on suppose que le graphe est coloriable (de façon valide) avec K couleurs. Sauf information plus précise sur la structure du graphe (par exemple présence d'au moins un sous-graphe triangulaire) on commence par $K = 2$;
- on considère les nœuds un à un, dans un certain ordre, et on cherche à les colorier validement avec une couleur au plus égale à K ;
- si impossible on revient au nœud précédent et on change sa couleur, puis on reprend ;
- si à force de revenir en arrière on arrive au premier nœud on recommence tout avec $K = K + 1$.

Ici encore, cette méthode peut être vue comme une formalisation d'une stratégie utilisée dans un jeu tel que, par exemple, le Colorigraphe (chapitre 1).

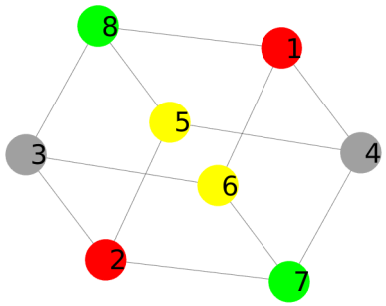
Dans sa version la plus simple (voir le code 8.7.5.10) l'ordre des nœuds est celui de leur numérotation, mais on peut raffiner, par exemple en classant les nœuds par ordre de degré décroissant. Même ainsi l'algorithme est sensible à la numérotation dès qu'il y a des nœuds de même degré. Il est facilement mis en échec sur un graphe simple (voir la figure 4.2.8), mais, contrairement aux autres son efficacité moyenne augmente avec la taille du graphe (cf. le tableau 4.1).

Une autre manière de présenter le fonctionnement de cet algorithme est de le considérer comme un parcours d'arbre de coloriages incomplets. La racine est le coloriage avec uniquement le premier nœud colorié à 1. On peut « élaguer » certaines branches au cours de la recherche, ce qui l'accélère, mais cependant ne change pas le niveau d'efficacité. Néanmoins, même ainsi, son temps calcul reste bien supérieur à celui des algorithmes gloutons.

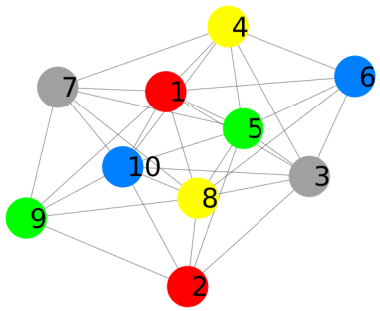
D'autres algorithmes à retour plus sophistiqués existent, mais pas vraiment généralistes. Par exemple celui défini dans **(author?)** [30] ne fonctionne bien que si un prétraitement de recherche de clique maximale est réalisé et, surtout, nécessite des paramétres définis par l'utilisateur en particulier le temps consacré à la recherche des cliques et le nombre de contraintes implicites à calculer, comme celle de la figure 4.2.9.

4.2.10 Comparaisons

Définissons l'*efficacité* d'un algorithme sur un graphe donné :



(a) Graphe cubique. La solution trouvée est à quatre couleurs, alors que deux suffisent.



(b) Graphe à 10 nœuds. Un coloriage optimal à 5 couleurs est trouvé, mais pas dans le même ordre qu'avec RLF.

Figure 4.2.8: Algorithme à retours.

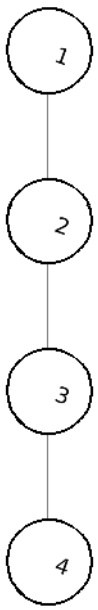


Figure 4.2.9: Une contrainte implicite est que les nœuds 1 et 4 ne peuvent avoir la même couleur.

Table 4.1: Efficacités moyennes d'algorithmes gloutons. On remarque que Johnson, bien que sensiblement plus compliqué que plusieurs Gloutons, leur est inférieur et que RLF est légèrement moins efficace que Glouton 2. L'algorithme Retours est très mauvais pour les petits graphes mais, contrairement aux autres, sont efficacité augmente avec le nombre de nœuds.

Nombre de nœuds ⇒	4	5	7	10
Glouton 0	0,992	0.953	0.839	0,627
Glouton 1	1	1	0,987	0,905
Glouton 1a	0,964	0,799	0,618	0,376
Glouton 2	1	1	0,994	0,936
Glouton 3	1	0,934	0,752	0,504
Glouton 4	0,994	0,965	0,842	0,624
Johnson	0,987	0,940	0,815	0,636
RLF	1	1	0,987	0,905
Glouton excentrique	0,991	0,952	0,826	0,599
Retours	0,446	0,549	0,649	0,657
Séquentiel 1	0,934	0,952	0,880	

- elle vaut 1 si un coloriage optimum est trouvé ;
- elle vaut 0 sinon.

Si deux algorithmes ont la même efficacité moyenne sur l'ensemble des graphes d'ordre N, on dira qu'ils sont *N-équivalents*.

Par exemple, pour $N = 4$, Glouton 1, Glouton 2, Glouton 3 et RLF sont équivalents et même d'efficacité parfaite (égale à 1), mais ce n'est plus le cas pour des valeurs plus grande. Pour $N = 10$ (avec mon petit ordinateur portable, je ne peux guère aller plus loin), on trouve que l'efficacité moyenne de Glouton 0 est 0,627, alors que celle de Glouton 3 n'est que de 0,504. Ce résultat est d'ailleurs un peu inattendu car Glouton 3 est plus sophistiqué que Glouton 0. Le tableau 4.1 donne plus de détails.

On y remarque que les algorithmes les plus compliqués ne sont pas forcément les meilleurs (en moyenne), RLF vs Glouton 2, par exemple.

Notons cependant que comme le nombre de graphes possibles augmente très rapidement avec N, les efficacités indiquées ne sont que des estimations par échantillonnage aléatoire. Naturellement, pour que les comparaisons soient à peu près fiables, la même liste de graphes aléatoires est utilisée pour tous les algorithmes.

On peut aussi, plus classiquement, comparer les résultats sur quelques grands graphes donnés. Par exemple le problème `wap05a.col` de la librairie DIMACS (**(year?)**) comporte 905 nœuds et 43081 arêtes et son nombre chromatique est 50. Voici les résultats fournis par quelques algorithmes du tableau 4.1 :

- Glouton 0 trouve un coloriage avec 64 couleurs, Johnson et Retours avec 63 couleurs ;

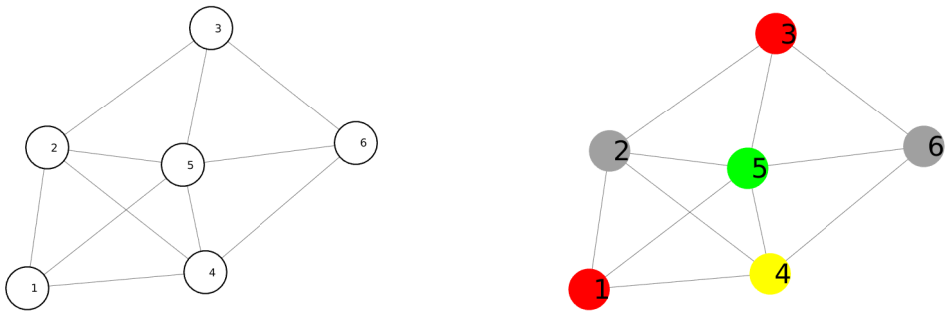


Figure 4.2.10: Le sous-graphe formé sur les nœuds (1, 2, 4, 5) forment une clique maximale. Il est alors certain qu'il faut au moins quatre couleurs et, donc, le coloriage présenté (trouvé par Glouton 0) est optimal.

- Glouton 1 et RLF en trouvent un avec 51 couleurs.
- Glouton 2 trouve un coloriage optimal de 50 couleurs.

Certains algorithmes très sophistiqués effectuent un prétraitement d'analyse du graphe, en particulier déterminent ce que l'on appelle ses cliques, (**(author?)** [15]), Nous n'utiliserons pas ici cette notion, mais une *clique* est un sous-graphe complet, c'est-à-dire que deux sommets quelconques y sont toujours adjacents (voir la figure 4.2.10). Il est immédiat que si une clique est de taille K, il faut K couleurs pour la colorier de façon valide et, donc, au moins autant pour le graphe complet. Une clique est maximale s'il n'y en a pas de plus grande (hormis éventuellement le graphe complet).

Pour autant un tel prétraitement ne donne pas forcément de meilleurs résultats, même sur d'autres exemples.

4.2.11 Algorithmes garantis

Ne serait-ce que pour comparer avec les algorithmes qui donnent une solution non forcément optimale, il est utile d'avoir au moins un algorithme complètement sûr. Le plus simple est la recherche exhaustive qui, évidemment, demande un temps considérable voire inacceptable dès que le graphe est un peu grand.

Comme déjà évoqué, à l'heure où ceci est écrit[3], tous les algorithmes garantissant une solution optimale dans tous les cas de figure ont une complexité exponentielle (complexité) en taille du graphe, du moins sur une machine classique (de Türing). Mais certains savent quand même tirer parti de la structure du graphe pour, assez souvent, être relativement rapides.

[3] janvier 2023

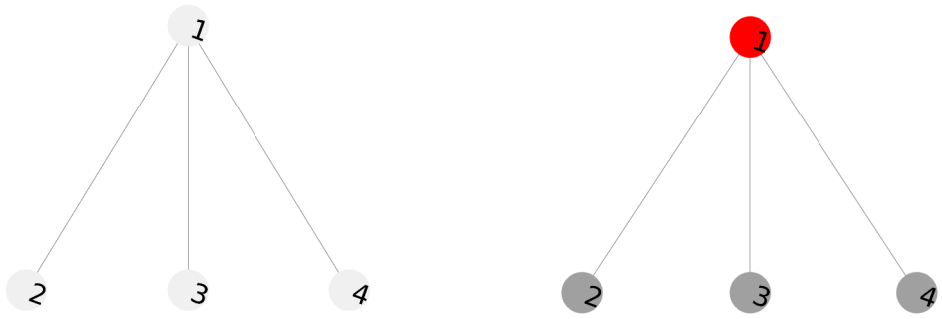


Figure 4.2.11: Un graphe à quatre nœuds et son coloriage optimal.

4.2.11.1 Programmation linéaire

Une méthode fondée sur la programmation linéaire est donnée en annexe 8.7.5.1. On utilise le codage binaire des coloriages. Si l'on recherche un coloriage à K couleurs pour un graphe à N nœuds, il peut donc être représenté par un vecteur binaire x de longueur NK. Le principe de base est d'essayer différentes valeurs de K, à partir de 2 (voire d'un peu plus si l'on fait une analyse de structure préalable) jusqu'à en trouver une pour laquelle le système d'équations et d'inéquations admet une solution. Pour chaque K :

- on construit une matrice A_{eq} et un vecteur b_{eq} tels que l'on doit avoir $A_{eq}x = b_{eq}$
- on construit une matrice A et un vecteur b tels que l'on doit avoir $Ax \leq b$
- on applique un algorithme de recherche d'une solution x, avec, donc, en plus, les contraintes que toutes ses composantes soient 0 ou 1.

Détaillons un peu à partir d'un exemple. Considérons le graphe à quatre nœuds à gauche de la figure 4.2.11. Il est facile de trouver manuellement son coloriage optimal, à deux couleurs, montré à droite de la figure.

Pour le trouver par la programmation linéaire sous contraintes, considérons son codage binaire lorsque l'on essaie avec $K = 2$. Il doit être de la forme

$$x = \begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \\ x_5 & x_6 \\ x_7 & x_8 \end{bmatrix}$$

Rappelons que la ligne du nœud i indique sa couleur par un 1 dans la colonne correspondante. Par définition, donc, il doit y avoir un seul 1 dans chaque ligne.

Une manière de le formaliser est de dire que la somme sur chaque ligne doit être égale à 1. On réécrit x sous la forme d'un vecteur « vertical » à huit composantes.

D'où la matrice

$$A_{eq} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

et le vecteur

$$b_{eq} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

de façon que l'on ait $A_{eq}x = b$.

Pour les inégalités on peut utiliser un calcul algébrique direct à partir de la matrice G du graphe

$$G = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

mais il est plus intuitif de construire progressivement la matrice A en considérant les arêtes une par une. Le vecteur b est ici

$$b = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Par exemple il y a une arête entre les nœuds 1 et 2 qui ne peuvent donc pas être de la même couleur. Donc $x_1 + x_3 \leq 1$. En effet , s'ils étaient de la même couleur cela se traduirait par $x_1 + x_3 = 2$. Ainsi la première ligne de la matrice A sera [10100000]. Pour la même raison $x_2 + x_4 \leq 1$ et la seconde ligne sera [01010000] etc.

Au final la matrice A est

$$A = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

et les inégalités sont bien de la forme $Ax \leq b$.

Le système à résoudre est alors

$$\begin{cases} A_{eq}x = b_{eq} \\ Ax \leq b \\ x_i \in \{0,1\} \end{cases}$$

avec N égalités et NK inégalités.

Il donnera comme solution possible (il y en a deux équivalentes) le vecteur coloriage $x = (0,1,1,0,1,0,1,0)$ qui, remis en forme, devient

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 0 \\ 1 & 0 \end{bmatrix}$$

ce qui, réinterprété en codage entier, donne $(1,2,2,2)$. Naturellement, s'il n'y a pas de solution, il faudra essayer une autre valeur de K. La stratégie peut être d'augmenter à partir d'une borne inférieure sûre ou bien de partir d'une valeur plausible et d'augmenter de un en cas d'échec et de diminuer de un en cas de succès. Ou, plus économique, de choisir les valeurs successives de K par dichotomie.

4.2.11.2 Un algorithme séquentiel

Une autre manière de construire un algorithme garanti est de définir une relation d'ordre sur les coloriages telle que le « plus petit valide » soit optimal.

Le coloriage C_1 est dit inférieur au coloriage C_2 dans les cas suivants :

1. $\max(C_1) < \max(C_2)$
2. ou bien $\max(C_1) = \max(C_2)$ et $code(C_1) < code(C_2)$

Avec cette relation d'ordre, l'algorithme de recherche est relativement simple (voir l'encadré 4.2.11.2), surtout si l'on code les couleurs de 0 à N-1, car les codages et décodages peuvent alors se faire en base N. Mais il faut indiquer comment calculer le *suivant* d'un code.

Recherche séquentielle

```
K=2; % Ou autre borne inférieure si l'on en connaît une
     % suite à une analyse de structure
Tant que 1=1 % A priori infini, mais on trouve toujours
             % une solution, au pire avec K=N
  C=[0 <N-K+1 fois>, 1, 2, ...,K-1]; % Cmin, admissible
  Cmax=[0, K-1 <N-K+1 fois>]; % On peut toujours supposer
                              % la couleur 0 pour le premier nœud
  Tant que C n'est pas égal à Cmax

    si C est admissible ET valide
    Proposer C comme solution
    FIN

  C=suivant(C, K)
  K=K+1 % Impossible avec K, on essaie une couleur de plus
```

Un point important est d'estimer le nombre de fois que les boucles *Tant que* seront parcourues.

La méthode la plus simple consiste à ajouter 1 au code de C, puis décoder pour générer le coloriage suivant. En pratique, on peut d'ailleurs générer directement le résultat, sans avoir à coder/décoder.

On notera qu'avec cette définition du suivant on génère de nombreux coloriages inadmissibles (voir l'exemple 4.2).

Considérons le pire des cas pour cet algorithme, à savoir le graphe entièrement connecté. Le premier coloriage considéré en base N, avec deux couleurs, est $C_{min} = (0, 0, \ldots, 0, 1)$. Ensuite on incrémente.

On pourrait penser qu'il suffit d'ajouter 1. Cela donnerait la séquence du tableau 4.2. Elle fonctionnerait d'ailleurs pour le graphe complet à quatre nœuds car seul le dernier élément (0 1 2 3) est valide et donne la solution.

Mais ce n'est pas le cas général. Avec le graphe linéaire de la figure4.2 on trouverait un coloriage valide à trois couleurs *avant* la solution optimale à deux couleurs.

D'où la nécessité de considérer en fait successivement des séquences par valeur de K. D'abord celle pour deux couleurs puis, si échec (aucun graphe valide), celle pour trois couleurs, etc. (voir le pseudo-code 4.2.11.2 et un code source à l'annexe 8.7.5.11). Ce qui, pour quatre nœuds, nous donne les séquences du tableau 4.3. Naturellement ces séquences ne sont entièrement générées que pour le graphe complet.

Table 4.2: Séquence des coloriages générés pour quatre nœuds, en incrémentant simplement de 1. Les coloriages sont indiqués en base quatre. Les inadmissibles sont signalés par un astérisque. Elle fonctionnerait pour le graphe complet avec la solution (0 1 2 3) mais pour le graphe linéaire on trouve un coloriage à trois couleurs (0 0 1 2) car son code est plus petit que celui à deux couleurs (0 1 0 1).

C
0 0 0 1
0 0 0 2*
0 0 0 3*
0 0 1 0
0 0 1 1
0 0 1 2
0 0 1 3*
0 0 2 0*
0 0 2 1
0 0 2 2*
0 0 2 3*
0 0 3 0*
0 0 3 1*
0 0 3 2*
0 0 3 3*

C (suite)
0 1 0 0
0 1 0 1
0 1 0 2
0 1 0 3*
0 1 1 0
0 1 1 1
0 1 1 2
0 1 1 3*
0 1 2 0
0 1 2 1
0 1 2 2
0 1 2 3

Table 4.3: Graphe à quatre nœuds. Séquences générées pour les valeurs successives de K (nombre de couleurs essayé). Quand il trouve une solution l'algorithme propose en fait en sortie le coloriage « minimum » équivalent. Par exemple (0 1 2 3) pour le graphe complet au lieu de (2 3 0 1). Pour le graphe linéaire de la figure 4.2 il trouve presque immédiatement la solution (0 1 0 1).

	$K = 3$	
	1 1 2 0	
$K = 2$	1 2 0 0	
0 0 0 1	1 2 0 1	
0 0 1 0	1 2 0 2	
0 0 1 1	1 2 1 0	
0 1 0 0	1 2 2 0	
0 1 0 1	2 0 0 1	
0 1 1 0	2 0 1 0	$K = 4$
0 1 1 1	2 0 1 1	2 3 0 1
1 0 0 0	2 0 1 2	
1 0 0 1	2 0 2 1	
1 0 1 0	2 1 0 0	
1 0 1 1	2 1 0 1	
1 1 0 0	2 1 0 2	
1 1 0 1	2 1 1 0	
1 1 1 0	2 1 2 0	
	2 2 0 1	
	2 2 1 0	

Table 4.4: Nombre de coloriages testés (les séquences) et temps calcul avant d'un trouver un optimal, sur dix graphes générés au hasard pour chaque nombre de nœuds N. Sans surprise la croissance est exponentielle en fonction de N, comme on le voit bien à partir de $N = 10$.

	Nombre de coloriages testés					Temps calcul				
N =>	4*	5	6	7*	10*	4	5	6	7	10
essai 1	6	44	177	1946	789379	2.097e-03	5.524e-03	7.562e-03	1.221e-02	2.647e+00
essai 2	28	45	610	2545	863579	2.247e-03	4.234e-03	1.17e6-02	1.095e-02	2.913e+00
essai 3	13	65	71	2084	768079	2.109e-03	3.715e-03	5.321e-03	8.613e-03	2.570e+00
essai 4	13	45	273	2195	38335	1.787e-03	3.825e-03	1.424e-02	9.016e-03	1.304e-01
essai 5	17	14	173	1114	881996	2.515e-03	4.122e-03	7.238e-03	5.919e-03	2.955e+00
essai 6	13	63	143	2227	840218	3.144e-03	4.568e-03	8.426e-03	9.174e-03	2.795e+00
essai 7	6	145	153	1882	847084	3.190e-03	4.626e-03	6.115e-03	8.208e-03	2.833e+00
essai 8	17	25	590	8079	884994	3.500e-03	2.890e-03	1.582e-02	2.609e-02	2.932e+00
essai 9	13	137	253	210	696314	2.980e-03	4.659e-03	1.079e-02	1.932e-03	2.330e+00
essai 10	19	62	73	1200	789439	2.750e-03	5.461e-03	6.453e-03	6.261e-03	2.627e+00

Ou mieux, comme déjà signalé, on peut, plus économiquement, sélectionner les K successifs par dichotomie.

Pour le graphe complet à cinq nœuds, il y aura alors 318 coloriages testés, dont 195 inadmissibles, avant de trouver le premier valide et optimal.

Notons qu'en pratique, comme les graphes sont rarement complètement connectés, la recherche s'arrête souvent avant. Le tableau 4.4 indique le nombre de coloriages testés pour en trouver un optimal, pour dix graphes aléatoires de différentes dimensions, ainsi que les temps de calcul. Comme on peut s'y attendre ces nombres croissent exponentiellement avec N, même si la recherche est relativement rapide car dans l'immense majorité des cas l'algorithme élimine le coloriage comme étant inadmissible[4] et n'a donc pas besoin de tester sa validité.

Les figures 4.2.12 et 4.2.13 donnent deux exemples de graphes générés, avec un coloriage optimal obtenu.

Sur le graphe cubique l'algorithme séquentiel trouve une solution optimale après avoir testé seulement 85 codes coloriages, d'ailleurs tous admissibles, ce qui est exceptionnel, la moyenne pour huit nœuds étant d'environ 54 % (voir le tableau 4.5).

[4]Rappelons qu'un coloriage $C = (c_1, \ldots, c_N)$ est admissible ssi il contient au moins une fois chaque entier de l'intervalle $[\min_i (c_i), \max_i (c_i)]$

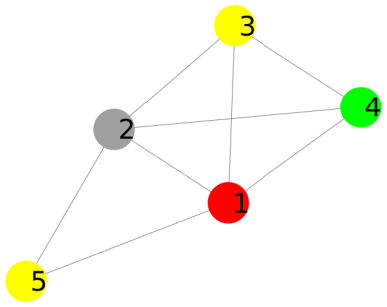


Figure 4.2.12: Recherche séquentielle. Graphe à 5 nœuds, coloriage optimal (quatre couleurs) trouvé après 173 testés.
.

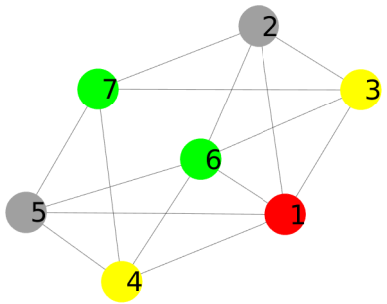
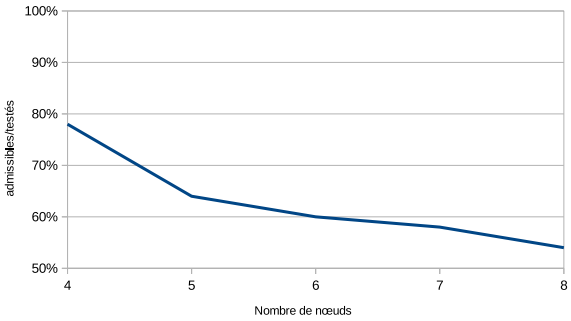


Figure 4.2.13: Recherche séquentielle.Graphe à 7 nœuds, coloriage optimal (quatre couleurs) trouvé après 2403 testés.

Table 4.5: Proportion moyenne de coloriages admissibles dans ceux testés avant de trouver une solution valide. Estimations sur 10 000 graphes au hasard au plus (pour dix nœuds).

Nombre de nœuds	admissibles/testés
4	78 %
5	64 %
6	60 %
7	58 %
8	54 %



Nombre de coloriages
Pour un graphe à N nœuds, on pourrait croire que le nombre de coloriages possibles est simplement N^N puisque chacun des N nœuds peut *a priori* être de l'une quelconque des N couleurs. Mais en fait des séquences équivalentes ne comptent que pour un seul coloriage (voir en annexe la section **??**). Ainsi, par exemple, les séquences (3,4,4,3) et (2,3,3,2) représentent le « même » coloriage (1,2,2,1).

À première vue la recherche séquentielle n'est donc pas très efficace, les algorithmes gloutons nous ayant habitué à beaucoup mieux. Mais la différence essentielle est que ces derniers ne donnent pas toujours un coloriage optimal alors qu'ici il est garanti.

On peut parfois légèrement diminuer le nombre de codes à tester. D'une part, comme pour les autres algorithmes, on peut modifier la numérotation des nœuds en considérant plus finement la structure du graphe (degrés des nœuds, sous-graphes, etc.). Plus spécifiquement, dans certains cas, il est possible de calculer une borne inférieure du nombre chromatique supérieure à deux. Par exemple même une analyse de structure rudimentaire peut mettre en évidence des « triangles » d'arêtes, auquel cas l'algorithme peut démarrer avec $K = 3$.

Ou bien, autre exemple, ce pourrait être la classique borne inférieure d'Hofmann**(author?)** [6]

$$K = 1 - \frac{\lambda_{max}(G)}{\lambda_{min}(G)}$$

où $\lambda_{max}(G)$ est la plus grande valeur propre de la matrice d'adjacence G et $\lambda_{min}(G)$ la plus petite (notez qu'elle est négative). Ou encore d'autres plus sophistiquées et souvent meilleures**(author?)** [12]. Néanmoins, en pratique, ces améliorations restent cependant marginales. D'autres sont légèrement plus efficaces.

4.2.11.3 Améliorations

Une première amélioration, évidente, est applicable lorsque l'on arrive à $K = N$. Nul besoin de tester, un coloriage optimal est $(1, 2, \ldots, N)$.

Une seconde amélioration est applicable au tout début, lorsque l'on essaie $K = 2$. En effet, tous les coloriages sont admissibles, sauf le dernier $(1, 1, \ldots, 1)$. Donc pas besoin de tester l'admissibilité des coloriages successifs.

Une troisième amélioration est moins évidente : il s'agit de « sauter » certains coloriages non admissibles. La figure 4.2.14 montre par exemple les tailles de ces séquences improductives pour le graphe à sept nœuds de la figure 4.2.13. Elles représentent au total près de 45% des coloriages testés et, selon le pseudo-code 4.2.11.2 l'algorithme est obligé de considérer tous ces coloriages.[5]. Mais pour chaque valeur de K il est relativement facile de situer au moins la plus grande de ces séquences (cf. l'annexe 8.4) et, donc, d'éviter d'avoir à tester l'admissibilité des coloriages la composant. En appliquant la méthode indiquée dans l'annexe on ne teste plus que 2318 coloriages au lieu de 2403, soit un gain de 3,5%. C'est assez peu et, surtout, ce gain est généralement décroissant avec la taille du graphe du fait que, pour chaque K, on ne saute qu'une seule séquence. Il y a donc là une piste pour faire mieux et sauter plusieurs séquences.

De plus, comme le montre l'histogramme de la figure 4.2.14, l'écrasante majorité des sauts potentiels (des intervalles de coloriages inadmissibles) sont nuls ou très petits. Les grands intervalles intéressants à sauter sont, *a contrario*, très rares.

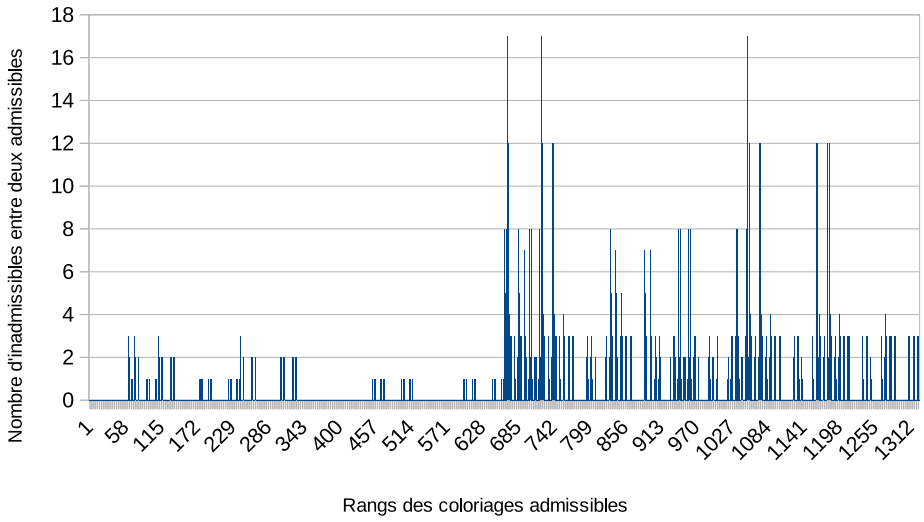
Une quatrième amélioration se fonde sur un constat empirique : sachant que le nombre d'arêtes maximum est $m = N(N-1)/2$, plus celui du graphe s'approche de ce maximum plus il est probable que le nombre chromatique soit lui-même proche de N. En pratique il est alors souvent plus efficace de démarrer sur une valeur de K assez grande et, selon le résultat, d'opter ensuite pour une diminution ou une augmentation.

Prenons un cas extrême pour illustrer, le graphe de la figure 4.2.15. Il a 7 nœuds et 20 arêtes. Il est donc quasi-complet ($m = 21$). L'algorithme séquentiel partant de $K = 2$ et même avec la technique des sauts que nous venons de voir, teste 28 541 coloriages avant de trouver un coloriage optimal à six couleurs.

Mais si l'on part de $K = 6$, pour lequel on trouve un coloriage valide, il suffit de considérer ensuite $K = 5$, pour lequel aucun coloriage valide n'est trouvé. On se convainc ainsi que six couleurs est bien le nombre chromatique. Ce faisant, l'algorithme aura testé seulement 23 772 coloriages, soit un gain de 16,7 %.

On peut d'ailleurs tout aussi bien partir de $K = 5$ et l'algorithme, après échec, essaiera 6 avec succès. Plus généralement, si A est le nombre d'arêtes du graphe et d_{max} le degré maximum des nœuds, la formule empirique suivante pour la valeur initiale de K semble donner plutôt de bons résultats, surtout pour les grands graphes :

[5] Au passage notons que pour l'affichage d'un nombre en base N la plupart des logiciels (Matlab© ici) se limitent aux symboles pris dans $\{0, 1, \ldots, 9\} \cup \{A, B, \ldots Z\}$ et, donc, refusent une base supérieure à 36. Pour les graphes supérieurs à cette taille il faut alors parfois écrire un programme spécifique.



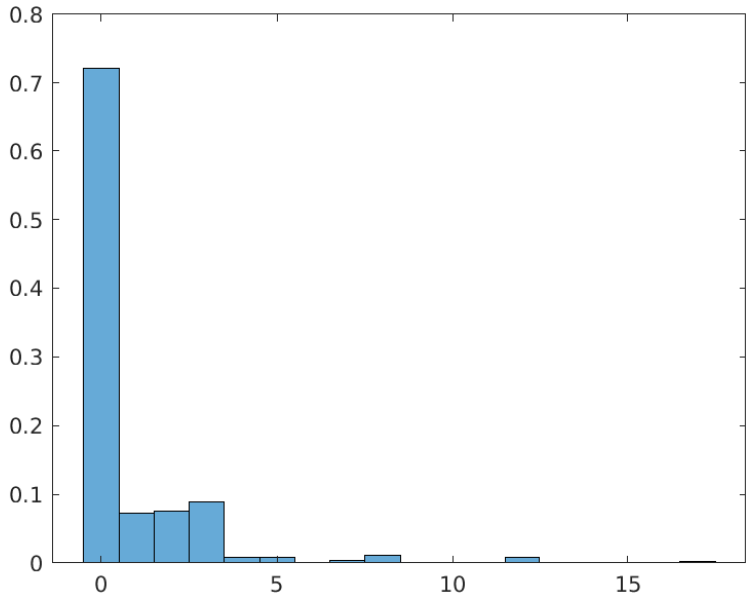


Figure 4.2.14: Méthode séquentielle sur le graphe à sept nœuds de la figure 4.2.13. Nombre de coloriages inadmissibles entre deux admissibles. Ils représentent plus de 45 % du total des graphes examinés. Néanmoins, comme le montre l'histogramme de répartition des tailles des sauts. la plupart sont nuls ou très petits. Ce sont les quelques rares grands intervalles qui expliquent ce pourcentage.

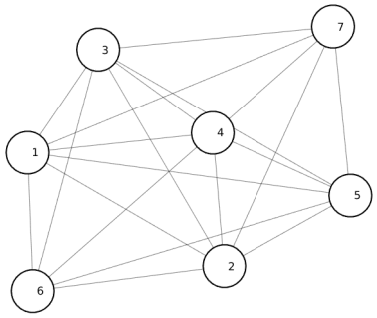


Figure 4.2.15: Recherche séquentielle. Graphe à 7 nœuds et 20 arêtes. Coloriage optimal à 6 couleurs trouvé après 23 772 testés, en partant de $K = 6$.

$$K_{initial} = \left\lfloor a \left(\frac{d_{max}}{A} \right)^b + 2 \right\rfloor \tag{4.2.1}$$

avec $a = 0,037$ et $b = -1,4$.

Pour les petits graphes, elle a tendance à sous-estimer le nombre chromatique. Par exemple, pour le graphe de la figure 4.2.15 elle ne donne que 2, au lieu de 6.

Par contre, pour le graphe `dsjc1000.9` (1000 nœuds, $d_{max} = 924$, $A = 449449$) de la bibliothèque DIMACS elle donne 215, ce qui est intéressant, car l'on sait à ce jour[6] que le nombre chromatique est au plus égal à 222. Ainsi, en partant de 215 on ferait très probablement une économie considérable de temps calcul.

Pour un graphe comme `wap05a` (905 nœuds, $d_{max} = 162$, $A = 21695$) le gain est même certain, puisque la formule donne 58 alors que le nombre chromatique est 50.

Notons que de toute façon, même si une formule telle que 4.2.1 est très grossière, elle permet toujours un gain (par rapport au fait de partir de 2) si $|K_{initial} - \chi| < |2 - \chi|$, où χ est le nombre chromatique du graphe.

4.2.11.4 Comparaisons

Nous avons considéré deux algorithmes déterministes garantis, par programmation linéaire et par recherche séquentielle. Comparons-les sur quelques exemples, en appliquant, pour la recherche séquentielle, les améliorations vues ci-dessus. Les temps de calcul ne sont que des estimations pour les graphes de plus de six

[6] janvier 2023

Table 4.6: Comparaison de la programmation linéaire et de la recherche séquentielle. Temps de calcul moyen en secondes pour des graphes construits au hasard pour chaque valeur du nombre de nœuds. La recherche séquentielle est plus efficace mais l'écart diminue avec la taille du graphe.

Nombre de nœuds	Nombre de graphes possibles	Nombre de graphes au hasard	Programmation linéaire	Recherche séquentielle
4	38	100	1,2731e-02	7,314e-04
5	728	2000	1,741e-02	7,273e-04
6	26 704	40 000	1,829e-02	2,758e-03
7	1 866 256	50 000	7,020e-02	1,551e-02

nœuds. Il faudrait pour eux traiter plusieurs millions de graphes aléatoires, ce que mon petit ordinateur ne permet pas de faire en un temps raisonnable.

Le tableau 4.6 permet cependant de constater que la recherche séquentielle est sensiblement plus efficace que la programmation linéaire, même si l'écart relatif diminue avec la taille du graphe.

Chapitre 5

Méthodes stochastiques

Les méthodes garanties induisent des calculs importants, voire hors de portée des machines classiques (non quantiques). Pour pallier cet inconvénient, nous avons vu des algorithmes déterministes rapides, mais non garantis.

Une autre approche non garantie est de faire appel au hasard mais au hasard contrôlé (**(author?)** [10]).

La recherche d'un coloriage optimal peut en effet être vue comme un problème de minimisation d'une fonction objectif judicieusement choisie. On fait alors appel à des *heuristiques* ou des *métaheuristiques*. La différence entre les deux est que les heuristiques sont spécifiquement conçues pour le type de problème considéré, alors que les métaheuristiques ont un domaine d'application plus général, quitte quand même à procéder parfois à des adaptations.

La distinction est toutefois un peu artificielle : si les adaptations sont suffisamment importantes, une métaheuristique n'est alors guère autre chose qu'une heuristique. Dans ce qui suit, heuristique voudra alors simplement dire « faisant appel au hasard », stochastique, donc, par opposition aux méthodes déterministes.

Bien sûr un coloriage vraiment minimum n'est pas toujours trouvé, mais au moins, même pour des grands graphes, on obtient souvent un résultat acceptable en un temps raisonnable. Reste quand même à les comparer aux algorithmes déterministes, gloutons ou non qui, après tout, ne fournissent pas toujours non plus le meilleur coloriage possible.

Notons d'ailleurs que la quasi-totalité de ces méthodes utilisent des générateurs algébriques de nombres pseudo-aléatoires, c'est-à-dire, en dernière analyse, une liste de nombres parfaitement définie. Donc, à strictement parler, ce sont aussi souvent des algorithmes déterministes, puisque la séquence des opérations est reproductible d'une exécution à l'autre (à condition, bien sûr, d'utiliser à chaque fois la même graine pour le générateur algébrique).

Si on les appelle stochastiques c'est que la liste est souvent si longue que lors d'une exécution on ne risque pas d'avoir à l'utiliser cycliquement. On peut d'ailleurs s'amuser à utiliser des listes relativement courtes et, donc, précisément, recyclées, ce qui est loin d'être inefficace (**(author?)** [10] chapitre 7). Ou une liste infinie mais dont chaque élément est directement connaissable, comme les

décimales de π.

Cependant, en principe, ce type d'algorithme *peut* utiliser des nombres vraiment aléatoires, issus de divers phénomènes physiques et leur dénomination n'est donc pas totalement usurpée.

On peut reprendre ici le modèle générique utilisé pour les algorithmes déterministes (4.2), en ajoutant simplement « au hasard » pour la sélection. Le pseudo-code général devient alors :

Modèle stochastique

```
Tant qu'il existe un nœud vacant
- en choisir un au hasard
- le colorier avec une couleur possible
  (c'est-à-dire qui n'est portée par aucun de ses voisins)
```

Naturellement, « au hasard » signifie en fait « selon un hasard plus ou moins dirigé », selon différents critères comme, par exemple, les degrés des nœuds ou, plus finement, les degrés de saturation (le nombre d'arêtess incidentes dont l'autre extrémité est déjà coloriée).

On voit d'ailleurs que, à l'extrême, la part de hasard peut être rendue nulle et que, donc, le modèle déterministe n'est qu'un cas particulier du modèle stochastique.

5.1 Quelques exemples

Toutes les heuristiques et métaheuristiques cherchent le minimum d'une fonction objectif[1], définie sur un espace de recherche. En fait, plus généralement, il suffit même qu'elles soient simplement capables de décider si telle position dans l'espace de recherche est « inférieure » à telle autre. Autrement dit, ces méthodes ont juste besoin d'une relation d'ordre sur cet espace.

Ici, l'espace de recherche est, pour un graphe donné, l'ensemble des coloriages admissibles, sur lequel nous avons déjà défini une telle relation d'ordre (voir la section 3.3.1). Mais bien d'autres sont envisageables. Pour deux coloriages C_1 et C_2, on peut ainsi s'appuyer sur les notions de validité ou non des coloriages et sur leur degrés, notés ici K_{C_1} et K_{C_2}, avec les deux règles suivantes :

- si C_1et C_2 sont de même statut (valide ou invalide), alors la comparaison est faite sur les degrés. Par exemple $K_{C_1} < K_{C_2} \Rightarrow C_1 \prec C_2$ (lire « C_1inférieur à C_2 »).
- si C_1est valide et pas C_2 alors $C_1 \prec C_2$.

Mais ce n'est pas forcément le plus judicieux. Retenons donc simplement que dès qu'une relation d'ordre est définie sur l'espace des coloriages, alors n'importe quelle méthode stochastique est *a priori* utilisable. Mais quelles sont les plus efficaces ?

[1] Sauf, bien sûr, celles qui traitent spécifiquement des problèmes multi-objectifs et cherchent plutôt des compromis entre ceux-ci.

Table 5.1: Comparaisons entre un glouton et des (méta)heuristiques. Degré du meilleur coloriage valide trouvé. Pour les méta)heuristiques est donné en plus le nombre d'évaluations qui ont été effectuées, quand il est important.

Graphes ⇒	Cubique	myciel3	myciel5	dsjc125.1	wap05a	dsjc1000.9
Nœuds (arêtes)	8 (12)	11 (20)	47 (236)	125 (736)	905 (43 081)	1000 (449449)
Coloriages possibles	16 777 216	$2,853 \times 10^{11}$	$3,878 \times 10^{78}$	$1,299 \times 10^{262}$	∞[2]	∞[3]
Nombre chromatique	2	4	6	5	50	$\leq$222
Algorithmes						
Glouton 2	2	4	6	7	50	308
ACO	2	4	6 (10^8)	5 (10^8)	50 (10^8)	252 (10^8)
Backtracking	2	4	6 (10^8)	5 (10^8)	50 (10^8)	302 (10^8)
Hybrid	2	4	6 (10^8)	5 (10^8)	50 (10^8)	283 (10^8)

À titre d'exemples et pour effectuer quelques comparaisons, j'ai juste retenu trois méthodes décrites dans l'ouvrage de R. M. R. Lewis, *A Guide to Graph Colouring: Algorithms and Applications* (**(author?)** [20]) :

- une adaptation de l'algorithme à colonie de fourmis (*ACO*)
- un algorithme avec retours (*Backtracking*)
- un algorithme hybride (*Hybrid*)

Leur descriptions détaillées sont dans l'ouvrage cité, mais je m'intéresse ici uniquement aux comparaisons avec Glouton 2, données dans le tableau 5.1. On peut constater que ce dernier est loin d'être ridicule, même pour un graphe de 1000 nœuds, et pour un effort de recherche bien inférieur. Ceci suggère qu'un bonne stratégie pourrait être d'appliquer d'abord Glouton 2 puis un algorithme stochastique à partir de la solution trouvée.

5.2 Le moindre effort : le bi-objectif

Une approche intuitive est d'indiquer à un algorithme nos désidérata :

1. minimiser le nombre de couleurs ;
2. minimiser le nombre d'arêtes invalides (même couleur aux extrémités).

[2]Pour mon micro-ordinateur 64 bits.
[3]Pour mon micro-ordinateur 64 bits.

et de lui dire simplement ensuite « Débrouille-toi ! ».

Ces deux objectifs sont contradictoires, d'où l'idée d'utiliser, justement, un algorithme bi-objectif qui produira un front de Pareto de solutions qui, pour lui, sont équivalentes (figure 5.2.1).

Cette méthode est évoquée ici car en pratique tous les algorithmes bi-objectifs efficaces sont stochastiques (**(author?)** [28], **(author?)** [3]). Cependant, en fait, nous ne retiendrons qu'un point du front pour lequel l'objectif 2 vaut zéro. Sur l'exemple cela correspond à la valeur 5 pour l'objectif 1, qui, en l'occurrence, est bien le nombre chromatique du graphe. Bien sûr ce n'est pas toujours le cas. D'une part l'algorithme peut ne pas trouver de solution avec zéro pour l'objectif 2 et, d'autre part, il peut, et plus fréquemment, comme les algorithmes plus classiques, ne proposer que des solutions non optimales, avec trop de couleurs.

Donnons deux exemples, réalisés avec l'algorithme génétique bi-objectif `gamultiobj` intégré à Matlab©. Un code, volontairement rudimentaire pour être plus explicite, est donné en annexe 8.7.6.

Pour le graphe `myciel5` de la bibliothèque DIMACS, qui comprend 47 nœuds et dont le nombre chromatique est 6, le front de Pareto montre que la meilleure solution trouvée comporte 13 couleurs (figure 5.2.2).

Il est certes possible de raffiner cette approche, par exemple en attribuant des poids différents aux objectifs, mais, comme déjà signalé, ce livre ne concerne pas vraiment les algorithmes stochastiques et, donc, nous ne développerons pas plus cette présentation.

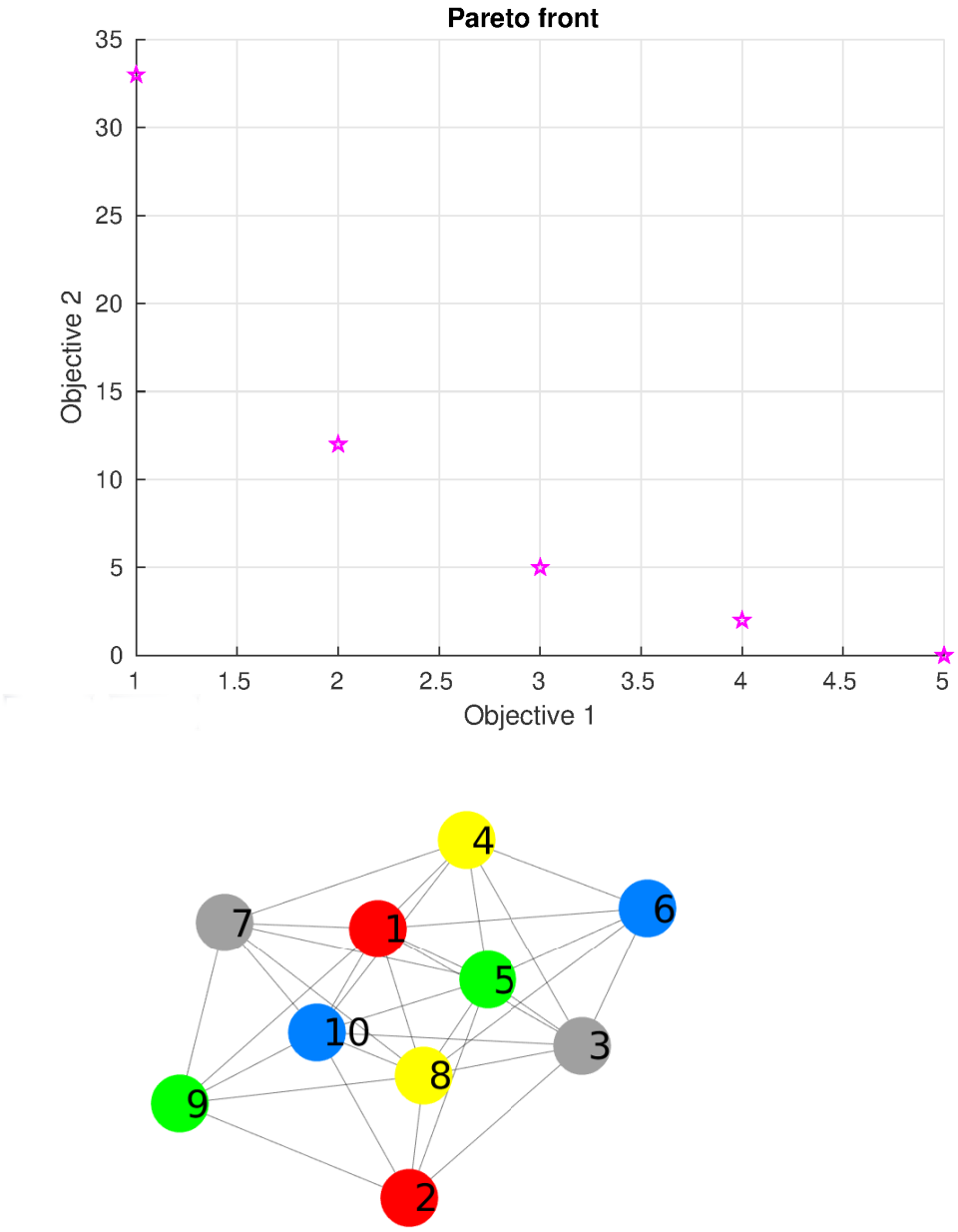


Figure 5.2.1: Graphe 10 nœuds traité par algorithme bi-objectif : front de Pareto et coloriage optimal.

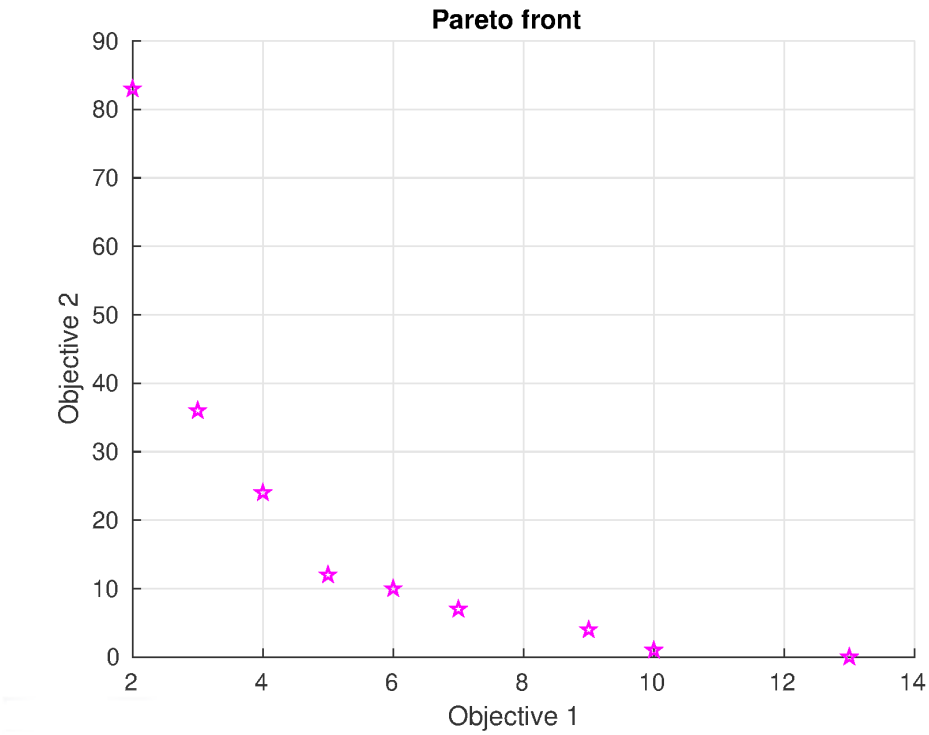


Figure 5.2.2: Graphe `myciel5` (47 nœuds) traité par algorithme bi-objectif : front de Pareto. La meilleure solution (objectif 2 à zéro) comporte 13 couleurs, alors que le nombre chromatique est 6.

Chapitre 6

Une méthode quantique

Comme signalé plus haut, il est en général impossible, pour les grands graphes (sauf structures particulières), d'obtenir un coloriage vraiment minimum en un temps raisonnable, même sur des super-calculateur[1], tant qu'ils sont du type « machine de Türing ». Il n'en est plus de même sur une machine quantique. Certes, on n'est jamais *absolument* sûr que le coloriage obtenu soit le vrai minimum, puisqu'il y a une part de hasard. Mais on peut augmenter drastiquement la probabilité que ce soit bien le cas en répétant les exécutions, tout en restant dans des temps de calcul raisonnables.

Il y a cependant deux contraintes :

- Disposer d'un ordinateur quantique puissant. Les progrès sont constants[2] et l'on peut espérer atteindre rapidement un nombre suffisant de qubits (les « bits » quantiques) pour résoudre le coloriage de graphes de grande taille.
- Disposer d'algorithmes adaptés. En fait c'est le plus incertain, car pour développer un tel algorithme, il faut, comme le préconisait Sartre dans un tout autre contexte, « se briser les os de la tête », raisonner autrement que pour un ordinateur classique.

À titre d'exemple de validation conceptuelle un algorithme et une variante sont présentés ci-après. Si vous n'avez pas quelques notions de calcul quantique vous pouvez aisément l'ignorer et retenir simplement cette conclusion : sa complexité n'augmente que de façon polynomiale avec la taille du graphe (au lieu d'exponentiellement en algorithmique classique). Ce qui peut se comprendre intuitivement car la puissance d'un calculateur quantique augmente, elle, exponentiellement avec le nombre de qubits utilisables. Plus précisément, si n est ce nombre, le nombre d'états représentables simultanément est 2^n.

[1] Comme Frontier (Oak Ridge National Laboratory, Tennessee) qui, en 2022, peut réaliser 1102 PFlops/s (environ 10^{18}opérations/s).

[2] En juillet 2021 le processeur chinois Zuchongzhi possédait 66 qubits. En novembre l'ordinateur Eagle d'IBM en alignait 127. Et en décembre QuEra (Université de Harvard et MIT), annonçait 256 qubits.

Alors, très grossièrement, comme le nombre de coloriages de degré K d'un graphe à N nœuds est K^N, et si K est de l'ordre de 2^k, on peut comprendre qu'un calcul quantique avec $n = kN$ qubits devrait permettre de les tester tous d'un seul coup. En pratique, cependant, pour les grands graphes, N est sensiblement supérieur à n, et il faut fractionner la recherche, pour différentes valeurs de K.

Les algorithmes sont donnés ici sous la forme de circuits quantiques. Les constructions de ces circuits s'inspirent de la méthode mathématique détaillée dans l'annexe 8.5, mais peuvent être comprises indépendamment.

Sinon, à défaut d'affronter les chaussetrappes dissimulées et autres embûches de ces algorithmes, vous pouvez toujours vous délasser avec un autre petit jeu, donné en annexe 8.8 : la Course quantique !

6.1 Rudiments de calcul quantique

Cette section peut évidemment être sautée si vous connaissez déjà le calcul quantique. D'ailleurs il ne s'agit ici que d'une présentation très simplifiée, juste suffisante (et pas tout à fait orthodoxe !) pour pouvoir comprendre les algorithmes détaillés ensuite.

Les bits quantiques ou *qubits* sont des vecteurs à deux dimensions, de longueur 1. Par convention le vecteur $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ sera noté $|0>$ et le vecteur $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$ sera noté $|1>$. Ces deux notations sont des *kets*. Ainsi tout qubit $\begin{pmatrix} \alpha \\ \beta \end{pmatrix}$ peut s'écrire $\alpha|0> + \beta|1>$.

Notez que l'on impose $\alpha^2 + \beta^2 = 1$ et que, donc, α^2 et β^2 sont tous deux dans $[0, 1]$ et peuvent être interprétés comme des probabilités [3]. Mais probabilités de quoi ?

Sur les qubits on définit trois types d'opérations :

- la *superposition* ;
- les *transformations* ;
- la *mesure*.

Un algorithme est alors souvent représenté par un *circuit* qui indique, à chaque pas de temps, quelles opérations sont appliquées à quels qubits.

Mesure

Commençons par la mesure qui, est, très généralement, la dernière opération d'un circuit. Un des principes du calcul quantique est que, tant qu'un qubit n'est pas mesuré, non seulement l'on ne connait pas sa valeur, ce qui semble

[3] En toute généralité les coefficients sont des nombres complexes tels que $|\alpha^2| + |\beta^2| = 1$ et, donc, le qubit peut être vu comme un point sur la sphère unité de dimension 4.

normal, mais, plus étrangement, il n'a en fait *pas* de valeur vraiment définie, ce qui est nettement moins intuitif.

Le résultat de la mesure est alors d'attribuer la valeur 0 avec la probabilité α^2 ou la valeur 1 avec la probabilité β^2. Ainsi, après une mesure, un qubit est devenu un bit classique. La méthode habituelle de définition d'un algorithme est donc de manipuler des qubits un peu « à l'aveugle », sans pouvoir connaître leur état à chaque instant, mais, évidemment, selon des règles qui, dans le meilleur des cas, donneront les résultats souhaités au final, après les mesures.

Transformation

Une transformation peut concerner un ou plusieurs qubits. On utilise généralement le terme de *porte*, en référence à la réalisation physique d'un qubit par un photon qui est transformé en traversant effectivement divers dispositifs.

Il en existe de nombreuses, mais commentons seulement celles qui seront utilisées pour la méthode NK (section 6.2). Toutes les portes ont une représentation algébrique, à l'aide de matrices unitaires[4].

Porte H

Au départ, tous les qubits sont à $|0>$. Alors, pour ne priviliégier aucune configuration, on les fait « passer » par une porte H (le nom vient de Hadamard, qui les positionne en l'état intermédiaire, qui donnerait 0 ou 1 avec des probabilités égales à $1/2$, si on les mesurait.

Une telle porte induit donc la transformation

$$|0> \rightarrow \frac{|0> + |1>}{\sqrt{2}} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

En représentation matricielle, on a

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{6.1.1}$$

On voit qu'elle induit aussi la transformation

$$|1> \rightarrow \frac{|0> - |1>}{\sqrt{2}} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

mais nous n'en aurons pas besoin.

C'est l'analogue de la classique initialisation aléatoire uniforme souvent utilisée dans les algorithmes à population d'agents.

[4] Rappelons qu'une matrice unitaire est de norme 1 et que, donc, appliquée à un vecteur, elle le transforme en un autre de même norme.

Porte X

X comme *exchange*. Parfois appelée Pauli-X ou porte NOT ou *bit-flip*. Quand un qubit « passe » par une telle porte, il devient, en quelque sorte son opposé : $|0>$ devient $|1>$ et $|1>$ devient $|0>$. Plus généralement les composantes α et β sont échangées.

Sa matrice est

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{6.1.2}$$

et appliquer cette porte revient à effectuer le produit

$$X \times \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \beta \\ \alpha \end{pmatrix}$$

C'est l'analogue du Non logique en informatique classique.

Porte CX

CX comme *controlled exchange*. On l'appelle aussi CNOT car il s'agit en effet d'une « négation » contrôlée. Cette porte concerne deux qubits. Quand le premier est à $|1>$ une porte X est appliquée au second. Mais si le premier est à $|0>$, tout se passe comme si la porte X n'avait aucun effet sur le second. On peut dire qu'il la traverse en restant inchangé.

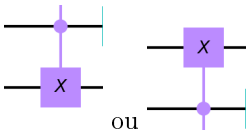
La matrice est donc maintenant 4×4 :

$$CX = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{6.1.3}$$

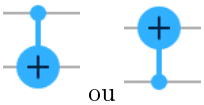
D'un point de vue algébrique, appliquer cette porte consiste à former un vecteur ψ à quatre éléments en concaténant « verticalement » ceux des deux qubits, puis à effectuer le produit

$$CX \times \psi = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} \alpha_1 \\ \beta_1 \\ \alpha_2 \\ \beta_2 \end{pmatrix} = \begin{pmatrix} \alpha_1 \\ \beta_1 \\ \beta_2 \\ \alpha_2 \end{pmatrix}$$

Dans un circuit, cette porte est souvent représentée plus ou moins ainsi

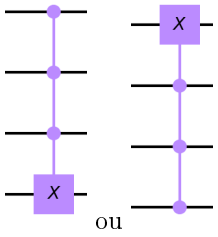


ou, si l'on pense surtout à l'aspect « négation »



Porte CnX

C'est une généralisation de la porte CX. Parfois notée MCX n (MC pour ***m**ultiple **c**ontrolled*). Au lieu d'un seul qubit de contrôle, il y en a n, déclenchant la porte X s'ils sont tous à $|1>$. Une telle porte peut évidemment être remplacées par un ensemble de portes plus simples, mais quand l'ordinateur quantique la supporte directement, cela permet d'être plus concis dans la description de l'algorithme. Par exemple, dans un circuit, une porte C3X peut être schématisée ainsi



Superposition et intrication

Ce sont sans aucun doute les notions les plus délicates de la physique quantique. Il n'y a d'ailleurs pas consensus sur leur interprétation.

Pour la superposition essayez d'imaginer comment un photon peut être *en même temps* polarisé à droite et à gauche. Et pour l'intrication comment deux photons créés ensemble d'une certaine manière (que l'on appelle, justement, intriquée) continuent à s'influencer l'un l'autre, instantanément, quelle que soit ensuite la distance qui les sépare (principe de non-localité).

Même si ceci a été prouvé expérimentalement pour la première fois en 1982 (**(author?)** [5]), puis confirmé à plusieurs reprises (**(author?)** [25, 23]).

Notons d'ailleurs que l'intrication peut être vue comme de la superposition étendue à plusieurs objets quantiques.

Mais si l'on s'en tient au calcul lui-même, sans chercher à savoir à quoi il peut bien correspondre dans le monde réel et se perdre dans des considérations quasi-philosophiques, il n'y a pas vraiment de difficulté car les prédictions de ces calculs sont en bon accord avec les mesures physiques et, en ce qui nous concerne ici, avec les résultats produits par des ordinateurs quantiques (ou leur simulations).

Ainsi la superposition est dans le fait qu'un qubit a deux paramètres (appelés plus haut α et β). Si ce qubit représente un photon, on peut dire que ce dernier est polarisé à droite avec une probabilité α^2 et à gauche avec une probabilité β^2.

Quant à l'intrication elle se manifeste par la manipulation simultanée de plusieurs qubits, une opération algébrique à base de matrices unitaires. C'est ce que font par exemple les portes CX, mais on peut bien sûr traiter un plus grand nombre de qubits et c'est ce que nous allons faire pour le coloriage de graphe.

6.2 Méthode NK

On cherche un coloriage comportant un nombre de couleurs K donné. Le nombre de qubits pour définir une matrice de coloriage est NK (d'où le nom de cette méthode générale). En pratique, pour construire un circuit quantique, l'approche algébrique vue à la section 8.5 est juste une ligne directrice que nous n'avons pas besoin de suivre strictement. Les différentes étapes peuvent être les suivantes :

- Définir le graphe à l'aide de qubits[5].
- Générer une superposition de tous les coloriages possibles.
- Identifier les paires de nœuds ayant la même couleur.
- Comparer au graphe. S'il y a une arête entre deux nœuds de même couleur, « détruire » la matrice de coloriage (il y a une astuce ici, voir ci-dessous).
- Mesurer (et afficher/sauvegarder les solutions).

Chacune de ces étapes peut être réalisée par un sous-circuit quantique. Un code Qiskit complet est donné dans l'annexe 8.7.7.3.

Parmi les solutions proposées, certaines peuvent parfois utiliser moins de K couleurs. Aussi est-il utile de les examiner, ce qui peut facilement être fait sur un ordinateur classique et en temps polynomial.

6.2.1 Définir le graphe

Les $\frac{N(N-1)}{2}$ arêtes possibles du graphe (supposé non orienté et sans boucle, mais la généralisation est facile) sont représentés par le même nombre de qubits $|0>$ (pas d'arête) ou $|1>$ (il y a une arête).

6.2.2 Générer une superposition de tous les coloriages possibles

Il suffit d'appliquer la porte de Hadamard aux qubits q_i. Ceux-ci peuvent être présentés sous la forme d'une matrice $N \times K$ qui peut être vue comme la superposition de toutes les matrices de coloriages

[5]Nous procédons ainsi pour avoir une représentation purement quantique et pouvoir appliquer des portes CX aux éléments du graphe mais, en fait, avec une approche hybride, nous pourrions utiliser une représentation binaire classique.

$$Q = \begin{pmatrix} q_0 & q_1 & \cdots & q_{K-1} \\ q_K & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ q_{(N-1)K} & \cdots & \cdots & q_{NK-1} \end{pmatrix}$$

Chacune des N lignes de K éléments doit comporter un 1 et un seul, donc de K manières possibles. Ainsi, uu total le nombre des matrices de coloriage est K^N.

Le code Qiskit de la section 8.7.7.3 donne alors, par exemple (après 1000 exécutions) les 27 coloriages possibles du graphe triangulaire :

{'010100100': 99, '010001010': 6, '001100100': 36, '010010001': 6, '010010010': 9, '001001010': 4, '100001010': 24, '001100010': 22, '001001100': 7, '100010001': 17, '100001001': 11, '100010100': 99, '001010100': 22, '100001100': 52, '001100001': 4, '010001100': 23, '100010010': 47, '100100001': 49, '100100100': 255, '001010001': 2, '010010100': 41, '100100010': 100, '010001001': 4, '001001001': 2, '010100001': 17, '001010010': 8, '010100010': 34}

Chacune des séquences binaires représente une matrice de coloriage. Par exemple 010100100 est [6]

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

signifiant

- couleur 3 pour le nœud 1
- couleur 3 pour le nœud 2
- couleur 2 pour le nœud 3

ce qui est d'ailleurs un coloriage invalide. Notez que pour simplifier le circuit (voir la figure 6.2.1) on réinitialise $N \times \begin{pmatrix} K \\ 2 \end{pmatrix}$ fois des qubits auxiliaires. Cette opération n'est pas une porte logique mais elle est néanmoins utilisable sur un ordinateur quantique, car il faut la comprendre comme une mesure (qui projette le qubit sur 0 ou 1) suivie d'une réutilisation du-dit qubit avec forçage à l'état $|0>$. Le circuit est donc en fait dynamique (**(author?)** [17], **(author?)** [11]).

Et, évidemment, si ce code est placé au début d'un plus complet et sur une vraie machine quantique, les mesures finales sont à omettre.

6.2.3 Identifier les paires de nœuds ayant la même couleur

Il y a $\frac{N(N-1)}{2}$ paires possibles, aussi allons-nous utiliser le même nombre de qubits auxiliaires[7]. À l'aide des portes C2X chacun d'entre eux est positionné à $|1>$ si les deux nœuds ont la même couleur ou $|0>$ sinon.

[6] Il faut lire de droite à gauche, même si en fait ce n'est pas important ici.

[7] Il est tout-à-fait possible d'en utiliser moins voire un seul, au prix d'un augmentation du nombre de portes et, au final, de la complexité.

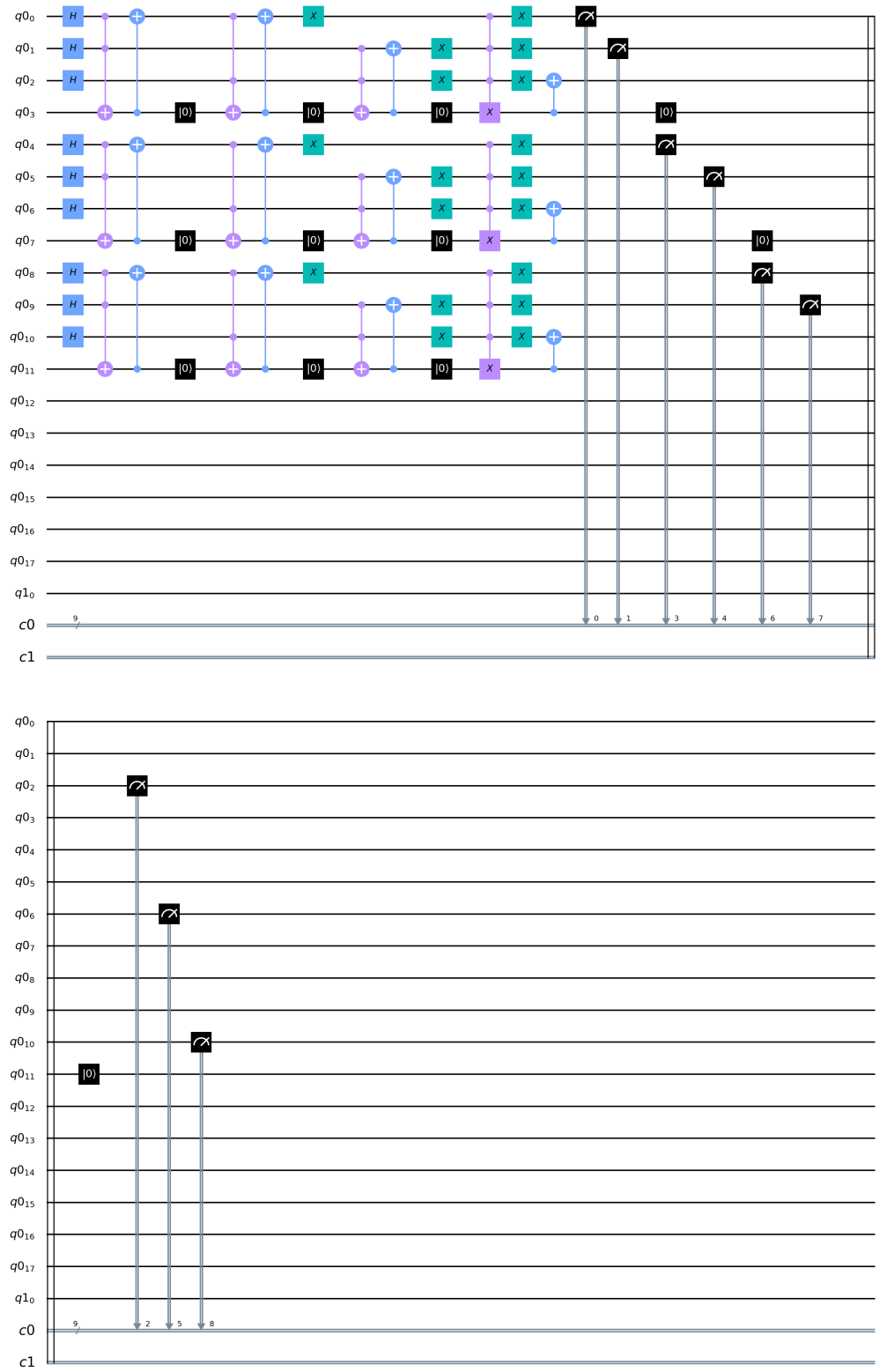


Figure 6.2.1: Méthode NK. Circuit pour générer toutes les matrices de coloriage (3 nœuds, 3 couleurs)

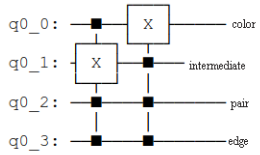


Figure 6.2.2: « Détruire » un coloriage si deux nœuds ont la même couleur (q0_2=$|1>$) et une arête entre eux (q0_3=$|1>$).

6.2.4 Comparer au graphe

Ensuite, on applique des portes C3X. Un qubit représente l'état d'une paire de nœuds (même couleur ou non), un autre l'état de l'arête correspondante (existante ou non) et un troisième est juste utilisé comme intermédiaire.

L'astuce pour « éliminer » les coloriages invalides est la suivante : le quatrième qubit (qui fait partie de la matrice de coloriage) est positionné à $|0>$ si les deux premiers sont dans l'état $|1>$. Voir le principe sur la figure 6.2.2. Bien sûr ceci peut être simplifié si vous disposez d'une porte spécifique de réinitialisation 3-contrôlée.

6.2.5 Mesurer et proposer les solutions

On provoque la mesure des qubits représentant la superposition des matrices de coloriage et on ne garde que les séquences de bits qui ne sont pas 0^*. Ensuite, on peut mettre les séquences obtenues sous forme matricielle, pour une présentation plus lisible.

6.2.6 Exemples

Deux exemples rudimentaires, juste pour illustrer la méthode.

6.2.6.1 3 nœuds, 3 couleurs

Le graphe est un triangle et on cherche un 3-coloriage.

Une simulation Qiskit donne comme solutions

{'010001100': 17, '100010001': 13, '100001010': 23, '001010100': 26, '010100001': 23, '001100010': 9, '000000000': 889}

On rejette la séquence nulle et il reste, comme attendu, six coloriages valides. Par exemple 100001010 donne la matrice de coloriage

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

soit le coloriage effectivement acceptable

- couleur 2 pour le nœud 1

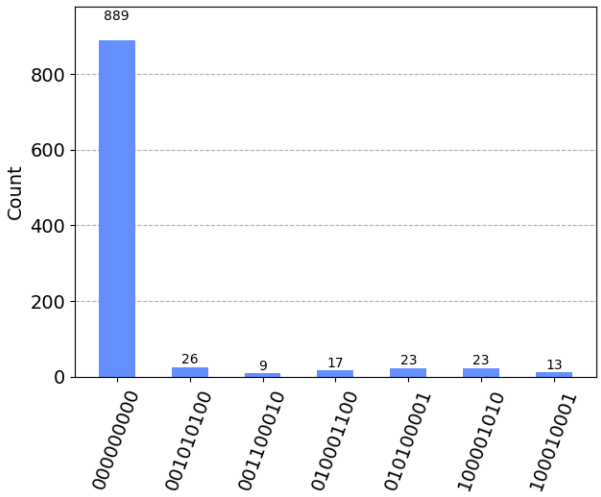


Figure 6.2.3: Dans les résultats la séquence binaire nulle est largement majoritaire.

- couleur 1 pour le nœud 2
- couleur 3 pour le nœud 3

Notons que le circuit produit majoritairement la séquence nulle 000000000, comme le montre aussi l'histogramme de la figure 6.2.3. Il faut donc exécuter suffisamment de fois le circuit pour être presque certain d'obtenir au moins une solution acceptable.

6.2.6.2 3 nœuds, 2 couleurs

On conserve le même graphe, en cherchant un 2-coloriage. C'est impossible et la simulation, après 1000 exécutions, donne simplement deux séquences nulles : {'0 000000': 350, '1 000000': 650}

6.2.7 Complexités

Pour $K \geq 3$ déterminer si un graphe quelconque est K-coloriable est un problème NP-Complet. En pratique cela signifie que sur un ordinateur classique (machine de Türing) nos simulations faites avec Qiskit nécessitent des ressources (mémoire, temps calcul) croissant exponentiellement avec la taille du graphe.

Ainsi mon portable de 6 Gb ne peut même pas résoudre le 3-coloriage d'un graphe à 4 nœuds et 5 arêtes (message d'erreur : *Insufficient memory for 29-qubits circuit*) en utilisant un circuit analogue à celui de la figure 6.2.1.

Néanmoins on peut essayer d'estimer des complexités théoriques, selon différentes mesures. Elles sont toutes polynomiales. D'autres approches sont de complexité exponentielle, même pour le problème de décision de la K-colorabilité (**(author?)**

[26]). D'autres encore suggèrent une complexité effectivement polynomiale ((**author?**) [13]), mais seulement expérimentalement.

6.2.7.1 Nombre de qubits

On utilise

- $\frac{N(N-1)}{2}$ qubits pour le graphe. Ceci pourrait d'ailleurs être ramené au même nombre de bits classiques avec une approche hybride.
- NK qubits pour la superposition des matrices de coloriage.
- $\frac{N(N-1)}{2}$ qubits pour les paires de nœuds.
- $N+1$ qubits auxiliaires.
- $NK+1$ bits classiques pour les mesures et les opérations conditionnelles.

Comme $K \leq N$ la complexité en nombre de qubits est $O\left(N^2\right)$, même si certains qubits auxiliaires sont réinitialisés et réutilisés.

6.2.7.2 Nombre de portes

En analysant le code Qiskit on peut construire une table comptant le nombre de portes. On note ici C_iX la porte X multi-contrôlée par i qubits. D'après cette table 6.1, la complexité en nombre de portes est donc $O\left(N^4\right)$.

Table 6.1: Portes utilisées par la méthode NK. Pour chaque porte la taille de la matrice unitaire équivalente est $2^w \times 2^w = 2^{2w}$.

	Nombre	Porte	Poids w
Définir le graphe	$\frac{N(N-1)}{2}$	X	1
Initialisation	NK	H	1
Matrice de coloriage	$N\frac{K(K-1)}{2}$	C_2X	3
	$N\frac{K(K-1)}{2}$	CX	2
	NK	X	1
	NK	C_KX	K+1
	NK	X	1
	NK	CX	2
Test de « même couleur »	$K\frac{N(N-1)}{2}$	C_2X	3
Comparer au graphe	$2NK\frac{N(N-1)}{2}$	C_3X	4
Total du pire cas ($K=N$)	$N^4 + N\frac{N(N-1)}{2} + 8N^2$		

Nœuds	Profondeur	Largeur	Complexité
2	19	8	152
3	58	18	1044
4	130	32	4160
5	244	50	12200
6	409	72	29448
7	634	98	62132
8	928	128	118784

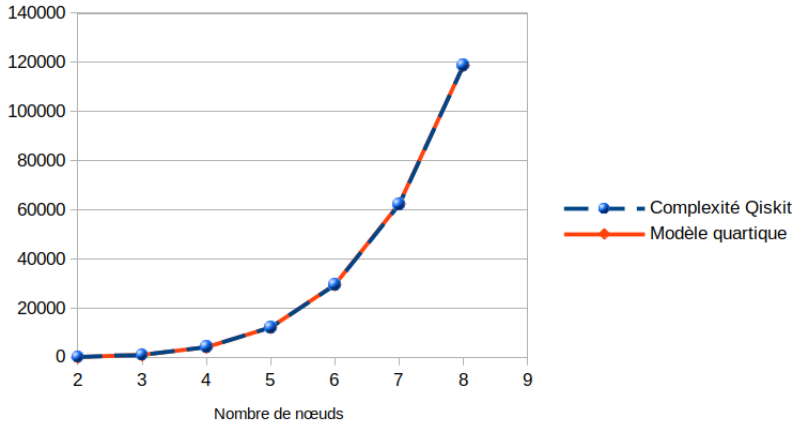


Table 6.2: Complexité du circuit pour la méthode *NK*. Elle croît de façon polynomiale avec la taille du graphe.

6.2.7.3 Taille du circuit

Une mesure classique de complexité d'un circuit quantique est le produit largeur×profondeur. La largeur est le nombre de qubits utilisés, qui est ici une fonction quadratique du nombre de nœuds et du nombre de couleurs.

La profondeur est de l'ordre du nombre de portes. Au final la complexité est donc $O\left(N^4\right)$. La profondeur réelle peut être demandée dans le code Qiskit. Le tableau 6.2 et la figure jointe confirment une complexité quartique.

Ainsi, pour chaque nombre de couleurs testé par l'algorithme la complexité est polynomiale, plus précisément $O(N^4)$. Même avec la plus mauvaise stratégie possible, consistant à essayer successivement 2 couleurs, puis 3, ... puis $N-1$, la complexité de la recherche d'un coloriage optimal est donc au plus $O(N^5)$.

Plus astucieusement, on peut procéder par dichotomie sur le nombre de couleurs K :

- on essaie $K_1 = 2$
- si pas de solution, on essaie $K_2 = N - 1$.
- si pas de solution, il faut N couleurs, une différente pour chaque nœud.

- si une solution, on sait que l'optimum est dans $]K_1, K_2]$ et l'on essaie l'entier K_3 le plus proche de son milieu
- etc.

Au maximum on devra essayer k valeurs avec $2^{k-1} \leq N-2 \leq 2^k$. La complexité devient alors

$$O\left(\ln\left(N-2\right)N^4\right) \tag{6.2.1}$$

6.2.8 Discussion

À l'évidence nous utilisons trop de qubits (NK) pour décrire un coloriage. En effet, théoriquement, pour un K-coloriage il n'est besoin que de Q qubits avec

$$Q = N\left\lceil\frac{\ln\left(K\right)}{\ln\left(2\right)}\right\rceil \tag{6.2.2}$$

où $\lceil x \rceil$ signifie « le plus petit entier supérieur ou égal à x ». Pour $N = 4$ et $K = 3$ nous avons $NK = 12$ et $Q = 8$. Et pour le 3-coloriage d'un graphe à 10 nœuds $NK = 30$ et $Q = 20$. Pour un K donné, le gain en pourcentage est évidemment constant ($1/3$ pour $K = 3$), mais augmente rapidement en valeur absolue.

6.3 Variante Q

Il est donc théoriquement possible d'utiliser moins de qubits que dans la méthode générale NK. La contre-partie est que construire le circuit devient de plus en plus compliqué quand le nombre de couleurs K augmente.

C'est pourquoi je présente ici la méthode seulement pour un tri-coloriage. Mais d'abord voyons comment l'on peut coder trois couleurs avec deux qubits et ensuite comment construire un circuit complet.

6.3.1 Deux qubits pour trois valeurs

Avec deux qubits nous avons quatre kets possibles : $|00>$,$|10>$, $|01>$ et $|11>$. Aussi, avant de construire le circuit complet, devons-nous utiliser un sous-circuit qui en élimine un. Disons, par exemple, que les trois premiers sont pour coder les trois couleurs. Nous avons donc besoin d'un petit circuit dont les entrées sont les deux qubits dans n'importe quel état et dont la sortie n'est jamais $|11>$, par exemple celui de la figure 6.3.1. Notez cependant que les probabilités des trois sorties possibles ne sont pas égales.

Le comportement de ce circuit peut être résumé ainsi :
$|00> \Rightarrow |00>$
$|11> \Rightarrow |00>$
$|10> \Rightarrow |10>$
$|01> \Rightarrow |01>$

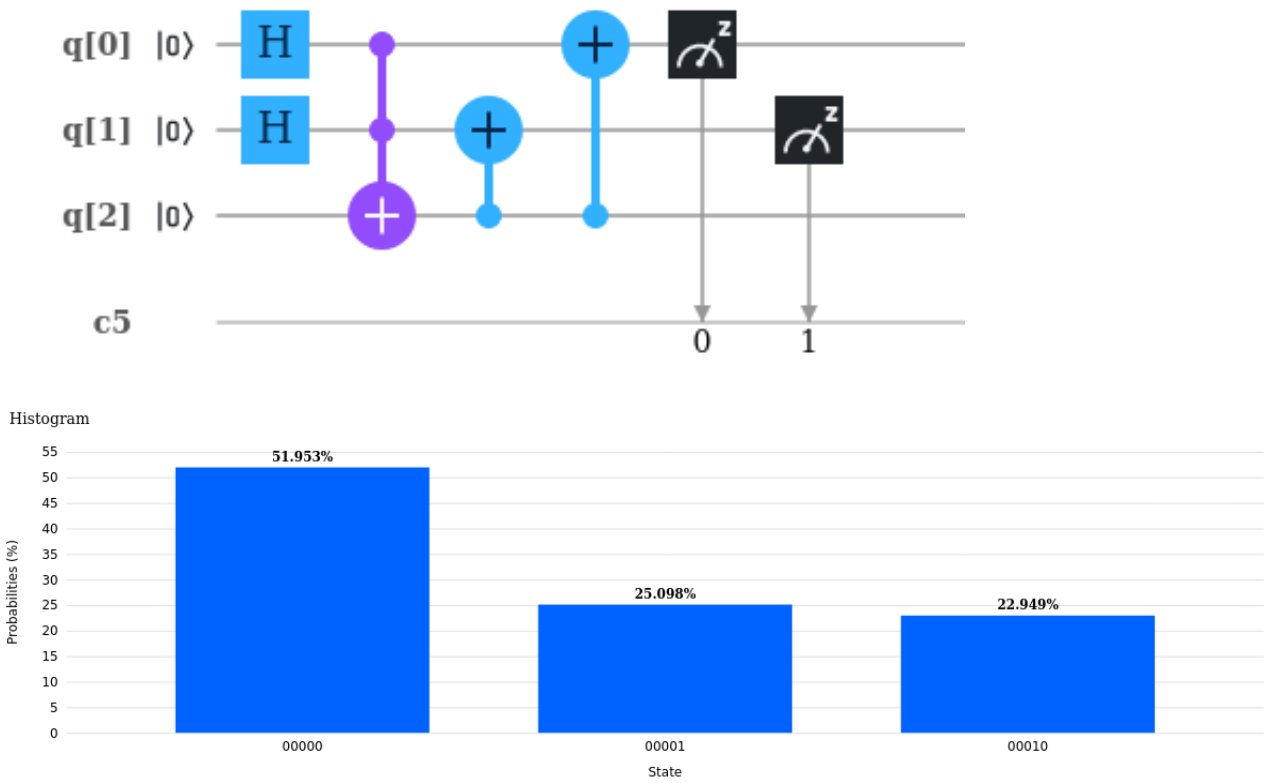


Figure 6.3.1: La sortie de ce circuit n'est jamais $|11>$. La figure de droite donne les probabilités estimées après 1000 exécutions pour les trois sorties possibles, 00, 01 et 10.

Alors qu'au début les quatre probabilités sont 1/4, on voit bien que les trois finales sont 1/2, 1/4 et 1/4. En pratique les valeurs trouvées après un certain nombre d'exécutions sont bien sûr un peu différentes.

Un tel manque d'uniformité n'est pas très élégant, même si sans grande importance dans le cas présent. Pour les puristes un circuit « parfait » est présenté un peu plus loin (6.3.4).

6.3.2 Un circuit complet

La différence principale d'avec la méthode *NK* est la manière dont sont repérées les paires de nœuds ayant la même couleur. Comme $|11>$ a été éliminé nous avons juste à considérer les trois autres possibilités. Dans le sous-circuit de la figure 6.3.2, le cinquième qubit est $|1>$ si et seulement si les deux paires des autres qubits codent pour la même couleur.

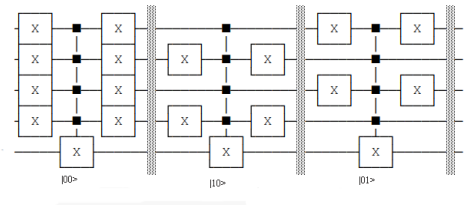


Figure 6.3.2: Méthode Q, 3-coloriage. Sous-circuit d'identification des paires de nœuds de même couleur 00, 10, ou 01.

Le reste du circuit est très semblable à celui de la méthode *NK* et dans le cas « même couleur et arête existante » nous avons à nouveau tous les qubits codant la couleur mis à $|0>$. Le circuit complet est montré par les figures 6.3.3, 6.3.4 et 6.3.5.

Un code Qiskit est donné dans la section 8.7.7.2.

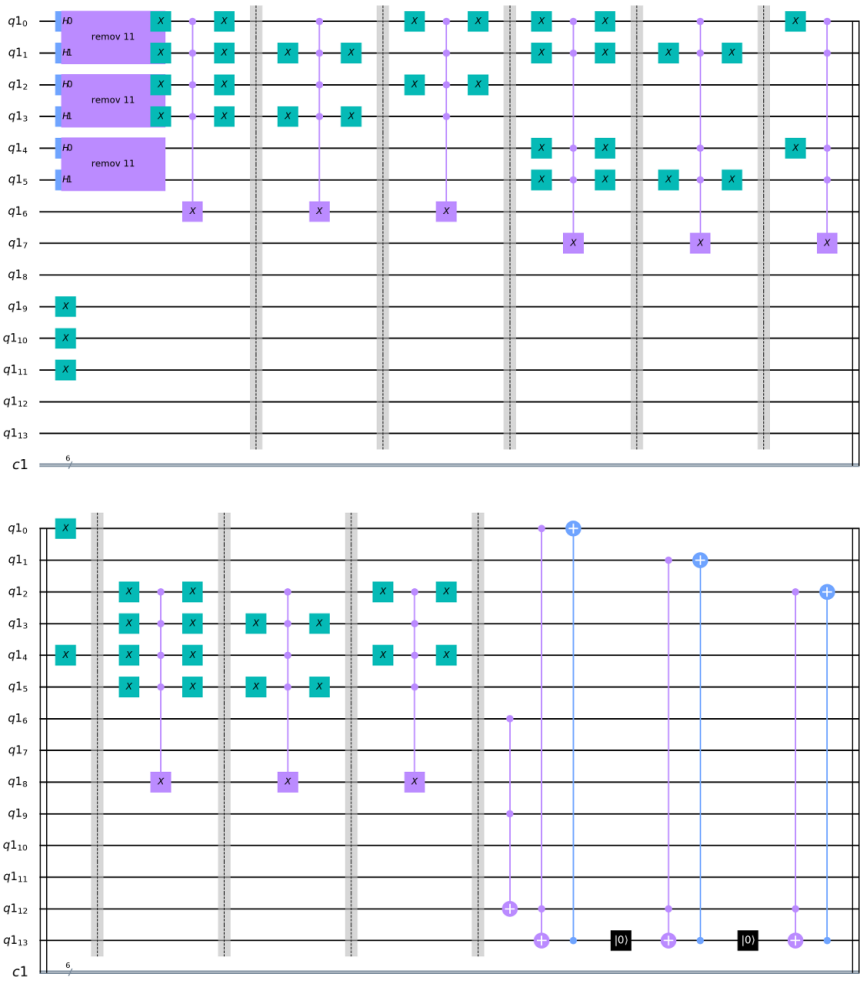


Figure 6.3.3: Méthode Q. Circuit pour le graphe triangulaire et trois couleurs (1/3).

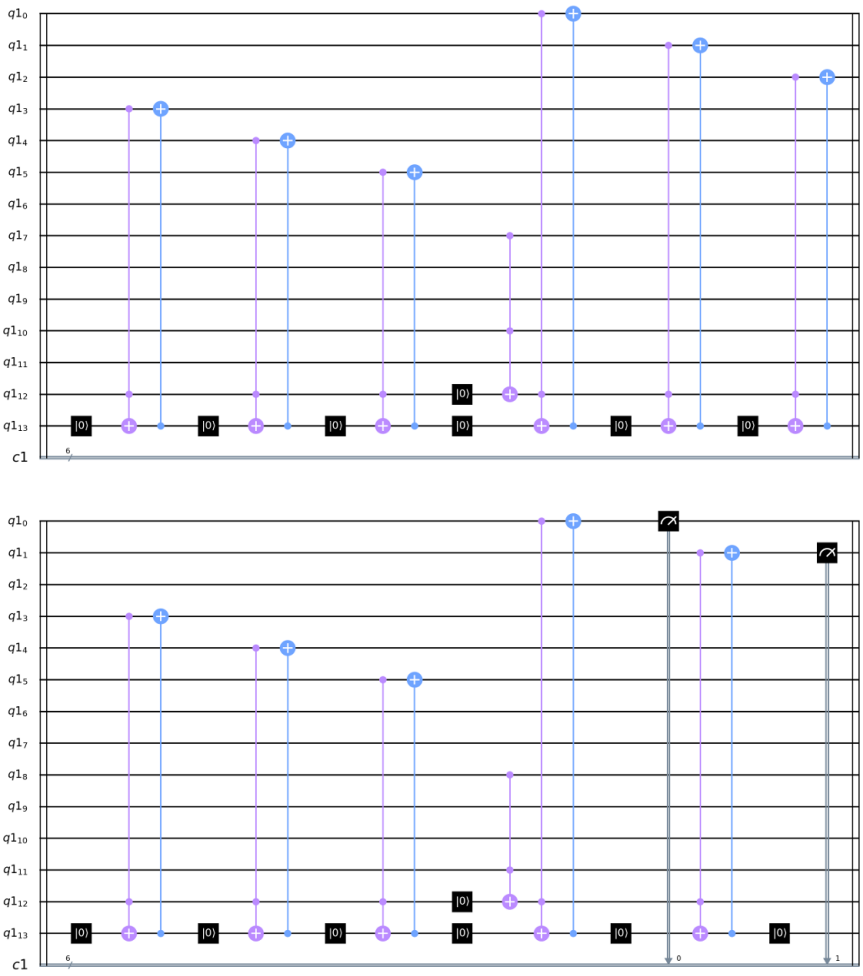


Figure 6.3.4: Méthode Q. Circuit pour le graphe triangulaire et trois couleurs (2/3) .

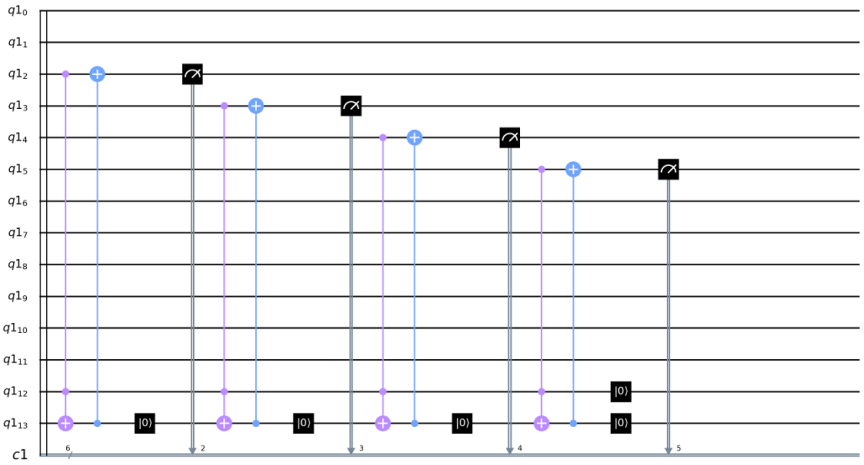


Figure 6.3.5: Méthode Q. Circuit pour le graphe triangulaire et trois couleurs (3/3).

Au-delà de trois

Supposons que nous voulions appliquer la méthode Q pour $K > 3$. Si $K = 2^k$ il n'est pas nécessaire d'éliminer quoi que ce soit. Mais si $2^k < K < 2^{k+1}$ nous devons éliminer $K - 2^k$ configurations. Construire le sous-circuit correspondant peut être très difficile. Un palliatif hybride possible, quoique pas très satisfaisant, est le suivant :

1. Ne rien éliminer avant le circuit principal.
2. Après mesure, éliminer par un traitement non quantique, les solutions qui nécessitent plus de K couleurs.

Sur le graphe triangulaire on trouve après mille exécutions :

{'000110': 31, '011000': 29, '010010': 44, '001001': 37, '100100': 34, '100001': 30, '000000': 795}

Par exemple '000110' est pour (01, 10, 00), car nous devons le lire de droite à gauche. Il représente les couleurs des trois nœuds. Dans ce cas simpliste les autres solutions sont juste des permutations de celle-ci.

Comme moins de qubits sont nécessaires, les coloriages valides de la figure 8.5.1 peuvent être trouvés sans message d'erreur pour cause de dépassement de mémoire même sur un petit ordinateur portable. Le résultat de la simulation est

{'01001010': 6, '00011010': 9, '00000000': 940, '01100000': 11, '00100101': 5, '10000101': 14, '10010000': 15}

Et les coloriages valides sont les six qui ne sont pas nuls. Notons que la sortie nulle est de loin la plus probable et il est donc nécessaire d'exécuter un grand nombre de fois le circuit (ici 1000) pous avoir les autres.

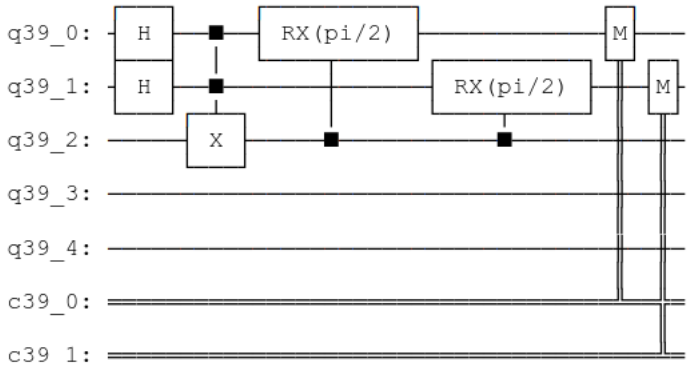


Figure 6.3.6: Rotations des deux qubits. Elles diminuent sérieusement la probabilité de 11.

6.3.3 Complexités

Quoique le nombre total de qubits nécessaires soit inférieur à celui requis par la méthode NK, nous avons toujours besoin de $\frac{N(N-1)}{2}$ qubits pour définir le graphe et les complexités sont donc du même ordre polynomial que pour la méthode NK.

Remarque : Il est aussi possible de diminuer le nombre de qubits tout en conservant une implémentation généralisable à tout nombre de couleurs, mais au prix d'une augmentation du nombre de portes et, au final, de la complexité totale. Le code de la section 8.7.7.4 présente une telle construction de circuit, de complexité largeur×profondeur $O\left(N^7\right)$.

6.3.4 Deux qubits pour trois valeurs - Méthode équilibrée

Méthodes 1 et 2

Nous pouvons appliquer une rotation aux deux qubits. Avec le petit circuit de la figure 6.3.6, nous obtenons par exemple la répartition suivante après 1000 exécutions : {'11' : 55, '10' : 302, '01' : 340, '00' : 303}. Elle n'est pas parfaite mais la probabilité de 11 est déjà assez petite (55/1000).

Pour la rendre vraiment nulle nous pouvons compléter le circuit, comme sur la figure 6.3.7. Après les rotations le cas 11 est « transformé » en 00. Nous obtenons maintenant, après 1000 exécutions, la distribution {'10' : 319, '01': 312, '00': 369}. Naturellement la probabilité de 00 est un peu trop élevée, mais la répartition n'est pas si mauvaise., l'important étant que la probabilité de 11 soit bien nulle.

Notez que nous pourrions n'avoir qu'un seul qubit auxiliaire et le réinitialiser à $|0>$ après les rotations. Néanmoins une telle réinitialisation peut être difficile sur un véritable ordinateur quantique et c'est pourquoi nous utilisons deux

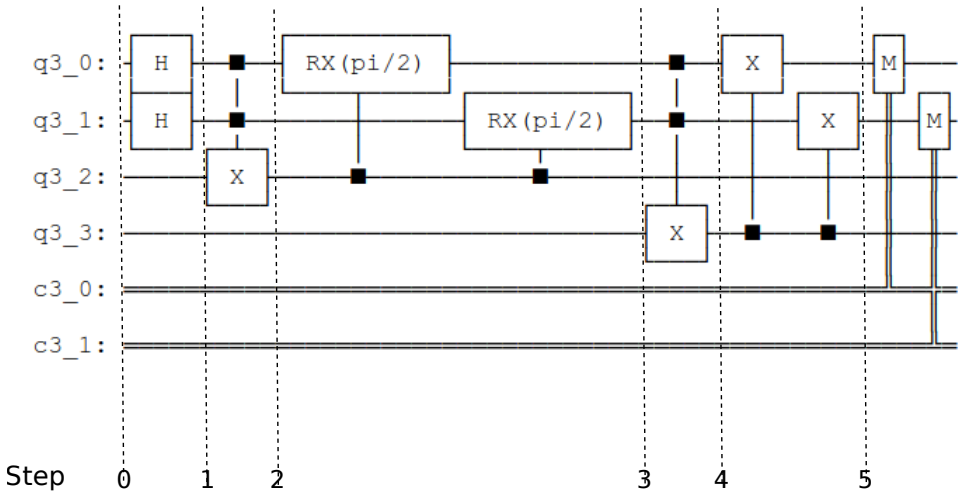


Figure 6.3.7: Élimination du cas 11.

qubits auxiliaires.

Il pourrait être utile de visualiser les qubits sur leurs sphères de Bloch[8], pas à pas, et et aussi d'examiner l'évolution du vecteur d'état (de taille $2^4 = 16$). Mais ceci ne peut être fait que sur une simulation. Sur un véritable ordinateur quantique observer un qubit le détruit (plus précisement le projette sur 0 ou 1).

Définissons (α_i, β_i) comme les coefficients du qubit q_i sur la base $(|0>, |1>)$. Alors la forme générale du vecteur d'état d'un système de quatre qubits est la suivante

$$\Psi = \begin{matrix}(\alpha_3\alpha_2\alpha_1\alpha_0, \\ \alpha_3\alpha_2\alpha_1\beta_0, \\ \alpha_3\alpha_2\beta_1\alpha_0, \\ \alpha_3\alpha_2\beta_1\beta_0, \\ \alpha_3\beta_2\alpha_1\alpha_0, \\ \alpha_3\beta_2\alpha_1\beta_0, \\ \alpha_3\beta_2\beta_1\alpha_0, \\ \alpha_3\beta_2\beta_1\beta_0, \\ \beta_3\alpha_2\alpha_1\alpha_0, \\ \beta_3\alpha_2\alpha_1\beta_0, \\ \beta_3\alpha_2\beta_1\alpha_0, \\ \beta_3\alpha_2\beta_1\beta_0, \\ \beta_3\beta_2\alpha_1\alpha_0, \\ \beta_3\beta_2\alpha_1\beta_0, \\ \beta_3\beta_2\beta_1\alpha_0, \\ \beta_3\beta_2\beta_1\beta_0)\end{matrix}$$

[8]Ou de Poincaré. Voir par exemple la définition sur Wikipédia.

Chque composante Ψ_j de Ψ peut être vue comme « correspondant » à la chaine binaire que nous trouverions après une mesure. En l'élevant au carré nous avons la probabilité de cette séquence. Pour se souvenir de quelle séquence correspond à quelle composante il suffit de remplacer α par 0, β par 1 et d'ignorer les indices. Ainsi la quatrième composante (élevée au carré) donnerait la probabilité de la chaine 0011.

Au tout début (étape 0), tous les qubits sont dans l'état $|0>$. Comme $|0>=(1,0)$ cela implique $\Psi_1 = 1$, et les autres nuls.

Étape 1 (après les portes de Hadamard)

Les qubits 0 et 1 sont dans l'état « intermédiaire » $\frac{|0>+|1>}{\sqrt{2}}$, les deux qubits auxiliaires restant inchangés. Le vecteur d'état donné par Qiskit est alors

[0.5+0.j 0.5+0.j 0.5+0.j 0.5+0.j 0. +0.j 0. +0.j 0. +0.j 0. +0.j 0. +0.j 0. +0.j 0. +0.j 0. +0.j 0. +0.j 0. +0.j 0. +0.j 0. +0.j]

ce qui est exactement ce que nous attendions

$$\left\{ \begin{array}{rcl} \Psi & = & \frac{|0>+|1>}{\sqrt{2}} \otimes \frac{|0>+|1>}{\sqrt{2}} \otimes |0> \otimes |0> \\ & = & \left(\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\right) \end{array} \right.$$

En effet, nous avons

$$\begin{array}{c} \alpha_0 = \frac{1}{\sqrt{2}} \\ \beta_0 = \frac{1}{\sqrt{2}} \\ \alpha_1 = \frac{1}{\sqrt{2}} \\ \beta_1 = \frac{1}{\sqrt{2}} \\ \alpha_2 = 1 \\ \beta_2 = 0 \\ \alpha_3 = 1 \\ \beta_3 = 0 \end{array}$$

Étape 2 (après la porte C2X sur le qubit 2)

Le qubit 3 reste dans l'état $|0>$, le qubit 2 est modifié mais, bizarrement, les qubits 0 et 1 sont aussi modifiés; légèrement. Le vecteur d'état donné par Qiskit est

[0.5-2.77555756e-17j 0.5-2.77555756e-17j 0.5-2.77555756e-17j 0. +0.00000000e+00j 0. +0.00000000e+00j 0. +0.00000000e+00j 0. +0.00000000e+00j 0.5-5.55111512e-17j 0. +0.00000000e+00j 0. +0.00000000e+00j 0. +0.00000000e+00j 0. +0.00000000e+00j 0. +0.00000000e+00j 0. +0.00000000e+00j 0. +0.00000000e+00j 0. +0.00000000e+00j]

Nous constatons de très petites valeurs venant de nulle part : la simulation Qiskit n'est pas parfaite. Le vecteur théorique est en fait

$$\Psi = \left(\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, 0, 0, 0, 0, \frac{1}{2}, 0, 0, 0, 0, 0, 0, 0, 0\right)$$

qui est bien ce que l'on retrouve en mettant à zéro ces très petites valeurs. Les probabilités non nulles sont données dans le tableau 6.3.

Chaine	Probabilité
0000	$\frac{1}{4}$
0001	$\frac{1}{4}$
0010	$\frac{1}{4}$
0111	$\frac{1}{4}$

Table 6.3: Méthode 1, étape 2. Probabilités non nulles des chaines de bits.

Chaine	Probabilité
0000	$\frac{1}{4}$
0010	$\frac{1}{4}$
0001	$\frac{1}{4}$
0100	$\frac{1}{16}$
0101	$\frac{1}{16}$
0110	$\frac{1}{16}$
0111	$\frac{1}{16}$

Table 6.4: Méthode 1, étape 3. Probabilités non nulles des chaines de bits.

Étape 3 (après les deux rotations contrôlées)
Le vecteur d'état de la simulation, une fois rectifié comme ci-dessus, devient :

$$\Psi = \left(\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, 0, -\frac{1}{4}, -\frac{1}{4}i, -\frac{1}{4}i, \frac{1}{4}, 0, 0, 0, 0, 0, 0, 0, 0\right)$$

Étape 4 (après la porte C2X sur le qubit 3)
On obtient

$$\Psi = \left(\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, 0, -\frac{1}{4}, -\frac{1}{4}i, -\frac{1}{4}i, 0, 0, 0, 0, 0, 0, 0, 0, \frac{1}{4}\right)$$

Chaine	Probabilité
0000	$\frac{1}{4}$
0010	$\frac{1}{4}$
0001	$\frac{1}{4}$
0100	$\frac{1}{16}$
0101	$\frac{1}{16}$
0110	$\frac{1}{16}$
1111	$\frac{1}{16}$

Table 6.5: Méthode 1, étape 4. Probabilités non nulles des chaines de bits.

Chaine	Probabilité
0000	$\frac{1}{4}$
0010	$\frac{1}{4}$
0001	$\frac{1}{4}$
0100	$\frac{1}{16}$
0101	$\frac{1}{16}$
0110	$\frac{1}{16}$
1100	$\frac{1}{16}$

Table 6.6: Méthod 1, étape 5. Probabilités non nulles des chaines de bits.

Étape 5 (après les deux portes NOT contrôlées)

$$\Psi = \left(\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, 0, -\frac{1}{4}, -\frac{1}{4}i, -\frac{1}{4}i, 0, 0, 0, 0, 0, \frac{1}{4}, 0, 0, 0\right)$$

Comme attendu, toutes les chaines xx11 ont une probabilité nulle. Si nous mesurons les deux premiers qubits, nous trouvons

- pour 00, $\frac{1}{4} + \frac{1}{16} + \frac{1}{16} = \frac{6}{16} = 0,375$;
- pour 01 et 10, $\frac{1}{4} + \frac{1}{16} = \frac{5}{16} = 0,3125$.

Faisons fonctionner à nouveau le circuit, mais 100 000 fois. Nous trouvons {'00': 37508, '10': 31162, '01': 31330 }. Les probabilités estimées sont alors très proches des théoriques.

Méthode 3, théoriquement parfaite

Une autre méthode est de considérer l'état quantique Ψ_0 après les portes de Hadamard et l'état Ψ_1 que nous souhaitons. Comment passer de l'un à l'autre ? Sur la base $(|00>, |00>, |10>, |01>, |11>)$ nous avons (ne pas oublier que la norme doit toujours être égale à 1) :

$$\begin{cases} \Psi_0 &= \frac{1}{2}(1,1,1,1) \\ \Psi_1 &= \frac{1}{\sqrt{3}}(1,1,1,0) \end{cases}$$

?ous devons trouver une matrice unitaire U telle (t est pour « transposition »)

$$U \times \Psi_0^t = \Psi_1^t$$

Nous savons que cela doit être une matrice de rotation 4×4 dont la forme générale est

$$U = \begin{pmatrix} a & -b & -c & -d \\ b & a & -d & c \\ c & d & a & -b \\ d & -c & b & a \end{pmatrix}$$

sous la contrainte $det\,(U) = 1$. Quelques manipulations algébriques simples donnent

$$\begin{cases} a & = & \frac{\sqrt{3}}{2} \\ b & = & -\frac{\sqrt{3}}{6} \\ c & = & \frac{\sqrt{3}}{6} \\ d & = - & \frac{\sqrt{3}}{6} \end{cases}$$

$$U = \frac{1}{2\sqrt{3}} \begin{pmatrix} 3 & 1 & -1 & 1 \\ -1 & 3 & 1 & 1 \\ 1 & -1 & 3 & 1 \\ -1 & -1 & -1 & 3 \end{pmatrix} \tag{6.3.1}$$

Il facile de vérifier que nous avons bien $U \times U' = U' \times U = I$. Mais la difficulté principale est de construire un circuit quantique équivalent à cette matrice U. C'est le problème bien connu de la *synthèse de circuit quantique*.

Pouvons-nous le faire avec uniquement des portes mono-qubit ? Si c'est le cas, cela implique que nous pouvons trouver deux matrices 2×2 U_1et U_2 telles que $U_1 \otimes M_2 = U$, où $\otimes$ est le produit externe.

Ceci est équivalent à un système de seize égalités comme $U_1\,(1,1)\,U_2\,(1,1) = a$, $U_1\,(1,1)\,U_2\,(1,2) = -b$, etc., mais on voit rapidement qu'elles sont incompatibles [9].

Nous devons donc utiliser au moins quelques portes à deux qubits. Considérons celle qui transforme la base initiale en la base « magique » (see**(author?)** [8][10]):

$$M = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & i & 0 & 0 \\ 0 & 0 & i & 1 \\ 0 & 0 & i & -1 \\ 1 & -i & 0 & 0 \end{pmatrix}$$

Rappelons que cette base magique est donnée par

$$\begin{cases} |m_1> & = & \frac{|00>+|11>}{\sqrt{2}} \\ |m_2> & = & \frac{i|00>-i|11>}{\sqrt{2}} \\ |m_3> & = & \frac{i|01>+i|10>}{\sqrt{2}} \\ |m_4> & = & \frac{|01>+|10>}{\sqrt{2}} \end{cases}$$

Un circuit possible est donné sur le figure 6.3.8, où S est la porte de phase $\begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$ et H la porte de Hadamard.

Comme U est réelle orthogonale nous savons (c'est un théorème) qu'il existe deux matrices 2×2 A_1 et A_2 telles que

[9] Plus techniquement c'est en relation avec la décomposition de Schmidt de l'état Ψ_1. Les coefficients de cette décomposition ne sont pas tous strictement positifs, ce qui signifie que l'état est intriqué, non séparable.

[10] *Attention* - Le circuit présenté dans cet article est erroné : inversion des deux dernières portes CNOT.

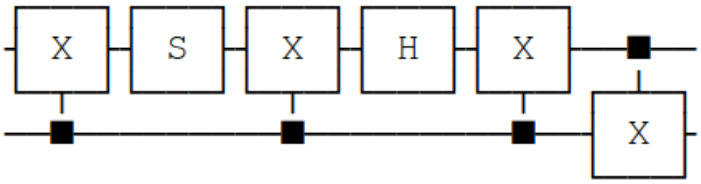


Figure 6.3.8: Un circuit réalisant la porte magique.

$$U = M' \left(A_1 \otimes A_2 \right) M$$

D'où

$$A_1 \otimes A_2 = MUM' = \frac{1}{2\sqrt{3}} \begin{pmatrix} 3-i & 1+i & 0 & 0 \\ -1+i & 3+i & 0 & 0 \\ 0 & 0 & 3-i & 1+i \\ 0 & 0 & -1+i & 3+i \end{pmatrix}$$

Et nous avons alors immédiatement

$$A_1 \otimes A_2 = I \otimes A$$

où I est la matrice unitaire

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

avec

$$A = \frac{1}{2\sqrt{3}} \begin{pmatrix} 3-i & 1+i \\ -1+i & 3+i \end{pmatrix}$$

Ainsi nous avons juste à trouver comment implémenter A. Comme pour toute porte mono-qubit A peut être décomposée ainsi

$$A = e^{i\theta_0} \begin{pmatrix} e^{-i\theta_1/2} & 0 \\ 0 & e^{i\theta_1/2} \end{pmatrix} \begin{pmatrix} \cos\left(\frac{\theta_2}{2}\right) & -\sin\left(\frac{\theta_2}{2}\right) \\ \sin\left(\frac{\theta_2}{2}\right) & \cos\left(\frac{\theta_2}{2}\right) \end{pmatrix} \begin{pmatrix} e^{-i\theta_3/2} & 0 \\ 0 & e^{i\theta_3/2} \end{pmatrix}$$

$$A = \begin{pmatrix} \cos\left(\frac{\theta_2}{2}\right) e^{i(2\theta_0-\theta_1-\theta_3)/2} & -\sin\left(\frac{\theta_2}{2}\right) e^{i(2\theta_0-\theta_1+\theta_3)/2} \\ \sin\left(\frac{\theta_2}{2}\right) e^{i(2\theta_0+\theta_1-\theta_3)/2} & \cos\left(\frac{\theta_2}{2}\right) e^{i(2\theta_0+\theta_1+\theta_3)/2} \end{pmatrix}$$

soit, en portes de rotation

$$A = e^{i\left(\theta_0 - \frac{\theta_1}{2} - \frac{\theta_3}{2}\right)} R_z\left(\theta_1\right) R_y\left(\theta_2\right) R_z\left(\theta_3\right)$$

en se souvenant que, par exemple

$$R_z\left(\theta_1\right) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\theta_1} \end{pmatrix}$$

d'où les coefficients $e^{-\frac{\theta_1}{2}}$, etc.

Par identification nous formons alors quatre équations complexes pour trouver quatre valeurs θ :

$$\begin{cases} \cos\left(\frac{\theta_2}{2}\right) e^{i(2\theta_0-\theta_1-\theta_3)/2} &= \frac{3-i}{2\sqrt{3}} \\ \sin\left(\frac{\theta_2}{2}\right) e^{i(2\theta_0+\theta_1-\theta_3)/2} &= \frac{-1+i}{2\sqrt{3}} \\ \sin\left(\frac{\theta_2}{2}\right) e^{i(2\theta_0-\theta_1+\theta_3)/2} &= \frac{1+i}{2\sqrt{3}} \\ \cos\left(\frac{\theta_2}{2}\right) e^{i(2\theta_0+\theta_1+\theta_3)/2} &= \frac{3+i}{2\sqrt{3}} \end{cases}$$

ce qui correspond à huit équations réelles, mais en fait seuls quelques sous-systèmes présentent quatre équations indépendantes, par exemple

$$\begin{cases} \cos\left(\frac{\theta_2}{2}\right)\cos\left(\frac{2\theta_0-\theta_1-\theta_3}{2}\right) &= \frac{3}{2\sqrt{3}} \\ \cos\left(\frac{\theta_2}{2}\right)\sin\left(\frac{2\theta_0-\theta_1-\theta_3}{2}\right) &= \frac{-1}{2\sqrt{3}} \\ \sin\left(\frac{\theta_2}{2}\right)\cos\left(\frac{2\theta_0-\theta_1+\theta_3}{2}\right) &= \frac{1}{2\sqrt{3}} \\ \cos\left(\frac{\theta_2}{2}\right)\cos\left(\frac{2\theta_0+\theta_1+\theta_3}{2}\right) &= \frac{3}{2\sqrt{3}} \end{cases}$$

Ce qui donne

$$\begin{cases} \cos\left(\frac{\theta_2}{2}\right) &= \pm\ \frac{\sqrt{5}}{\sqrt{2}\sqrt{3}} \\ \sin\left(\frac{2\theta_0-\theta_1-\theta_3}{2}\right) &= \pm\frac{1}{\sqrt{2}\sqrt{5}} \\ \cos\left(\frac{2\theta_0-\theta_1+\theta_3}{2}\right) &= \mp\frac{1}{\sqrt{2}} \\ \cos\left(\frac{2\theta_0+\theta_1+\theta_3}{2}\right) &= \pm\frac{3}{\sqrt{2}\sqrt{5}} \end{cases}$$

Définissons

$$\begin{cases} \varphi_0 &= -2\arcsin\left(\frac{1}{\sqrt{2}\sqrt{5}}\right) \\ \varphi_1 &= 2\arccos\left(\frac{1}{\sqrt{2}}\right) \\ \varphi_2 &= 2\arccos\left(\frac{\sqrt{5}}{\sqrt{2}\sqrt{3}}\right) \\ \varphi_3 &= 2\arccos\left(\frac{3}{\sqrt{2}\sqrt{5}}\right) \end{cases}$$

Une solution pour les angles θ est alors

$$\begin{cases} \theta_0 &= \frac{\varphi_0+\varphi_3}{4} \\ \theta_1 &= \frac{\varphi_3-\varphi_1}{2} \\ \theta_2 &= -\varphi_2 \\ \theta_3 &= \frac{\varphi_1-\varphi_0}{2} \end{cases}$$

Notez que $\theta_0 = 0$ et nous pouvons donc l'ignorer. La porte correspondant au coefficient $e^{-i\left(\frac{\theta_1}{2}-\frac{\theta_3}{2}\right)}$ est représentée par une matrice qui a juste cette valeur sur sa diagonale, les autres étant nulles. Le circuit final est donné sur la figure 6.3.9. Il est nettement plus compliqué que celui de la méthode 2 de la figure 6.3.7 ne serait-ce que parce que nous devons appliquer trois rotations.

En simulation Qiskit et 100 000 exécutions on obtient {'00': 33458, '01': 33200, '10': 33342}, très proche du $\left(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right)$ théorique attendu.

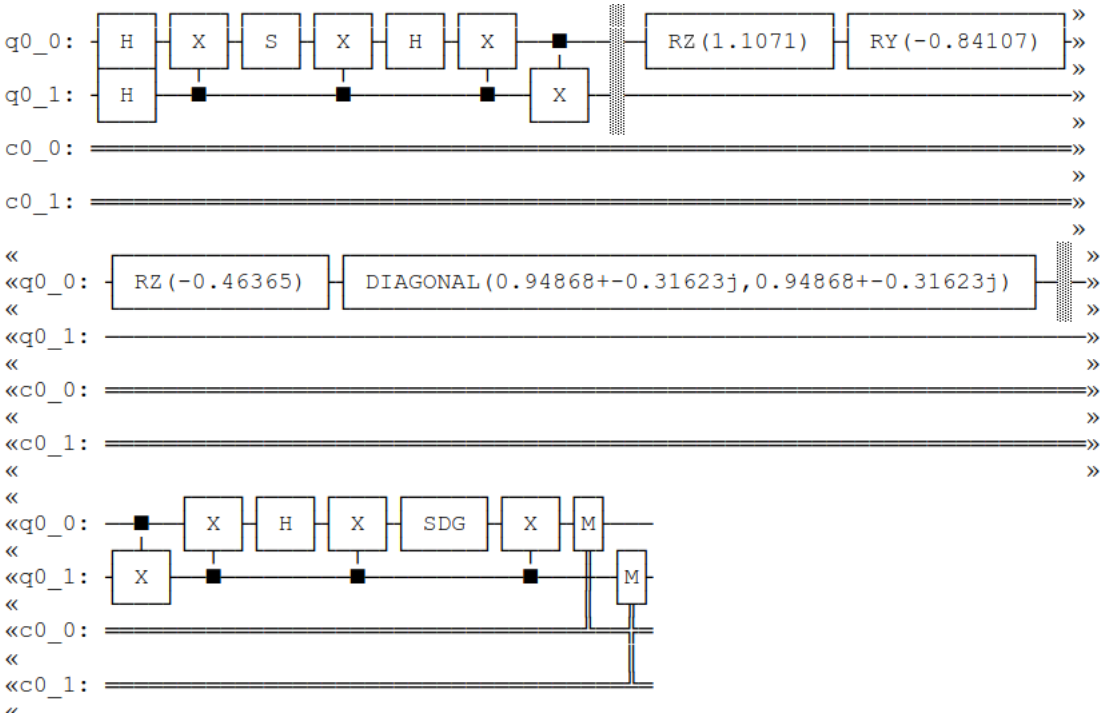


Figure 6.3.9: Circuit d'une méthode exacte pour éliminer 11.

Chapitre 7

Algorithmes diplomates

La définition d'un coloriage valide est fondée sur des contraintes du type « pas la même couleur pour deux nœuds adjacents ». Pour chaque paire de nœuds adjacents une telle contrainte n'a que deux valeurs : respectée ou non. En ce sens elle est binaire et les algorithmes qui les appliquent sont intransigeants.

Cependant il est parfois souhaitable d'être plus nuancé et de pouvoir indiquer dans quelle mesure deux nœuds adjacents de même couleur sont acceptables. Ceci peut être formalisé en affectant des niveaux d'*intolérance* d'arête invalide (extrémités de même couleur), par exemple de 0 à 1. La valeur 1 signifiant « totalement inacceptable » et la valeur 0 « parfaitement acceptable » (ce qui revient à dire que l'arête en question n'existe pas).

Alors on peut calculer l'*indice de frustration* d'un coloriage[1],

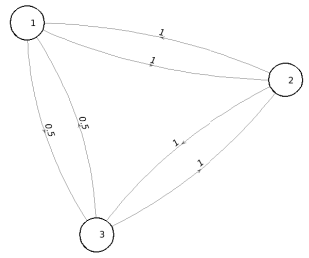
$$\frac{\textit{somme des intolérances des arêtes invalides}}{\textit{somme des intolérances}}$$

Si le coloriage est optimal l'indice vaut 0, la frustration est nulle, car il n'y a pas d'arêtes invalides. Mais si, comme dans certains cas, l'on est contraint d'utiliser moins de couleurs que le nombre chromatique, il convient quand même d'en réduire au mieux le nombre. Un algorithme qui cherche à le faire, grâce à des compromis judicieusement choisis, pourra être qualifié de *diplomate*.

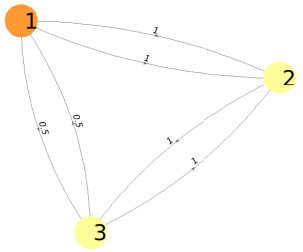
La figure 7.0.1 donne un exemple avec le graphe triangulaire. Ici les tolérances sont symétriques, mais ce n'est pas le cas général (voir plus loin d'exemple du covoiturage).

Tout algorithme de coloriage intransigeant peut être assoupli pour qu'il devienne diplomate. Par exemple le modèle Glouton vu à la section 4.2 devient

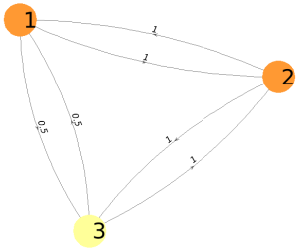
[1] Par analogie avec le concept de frustration en géométrie et en physique (par exemple pour des spins en interaction antiferromagnétique)



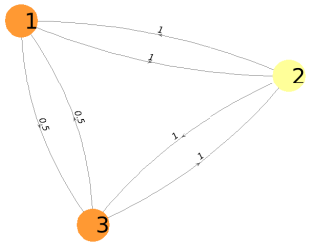
(a) Intolérances



(b) Frustration $\frac{0,8+0,8}{4,6} \simeq 0,348$



(c) Frustration $\frac{1+1}{4,6} \simeq 0,435$



(d) Frustration $\frac{0,5+0,5}{4,6} \simeq 0,217$

Figure 7.0.1: Graphe triangulaire - Intolérances et bi-coloriages possibles. Le meilleur compromis est d'affecter la même couleur aux nœuds 1 et 3.

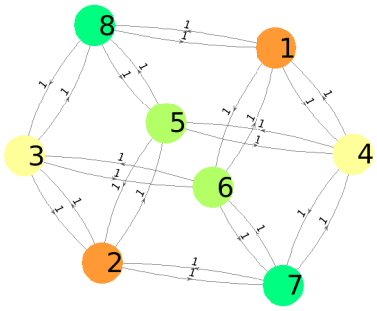


Figure 7.0.2: Graphe cubique, traité par Diplomate 0, équivalent alors Glouton 0.

Modèle Glouton diplomate

```
Tant qu'il existe un nœud vacant
- déterminer les couples (nœud vacant, couleur)
  qui augmentent le moins possible la frustration
- s'il n'y en a qu'un, le choisir
- sinon, appliquer un ou plusieurs critères
  pour en choisir un parmi ces équivalents
```

C'est la clause après le « sinon » qui définit les variantes possibles. Le choix de la couleur se fait dans une liste prédéfinie. Si la longueur de cette liste est supérieure ou égale au nombre chromatique il existe une solution de frustration nulle, qui ne sera d'ailleurs pas forcément trouvée, comme pour toute méthode gloutonne.

Plus généralement, même si cette longueur est inférieure au nombre chromatique, la solution proposée, pour laquelle la frustration sera nécessairement non nulle, ne sera pas toujours la meilleure.

Pour l'algorithme Diplomate 0, dont le code source est donné dans l'annexe 8.7.8, la clause « sinon » est simplement « prendre le premier rencontré ». Si toutes les intolérances sont égales, on retrouve évidemment Glouton 0, comme le montre la figure 7.0.2 : il lui faut quatre couleurs pour arriver à une frustration nulle, alors que deux suffisent.

Mais c'est surtout dans le cas de frustration nulle impossible qu'il est intéressant d'utiliser un algorithme diplomate. Reprenons l'exemple du covoiturage de la section 2.1, mais en supposant qu'il n'y a que deux voitures de disponibles et en évaluant les niveaux de frustration si deux personnes doivent partager la même voiture.

La figure 7.0.3 représente les intolérances estimées et une solution obtenue par Diplomate 0.

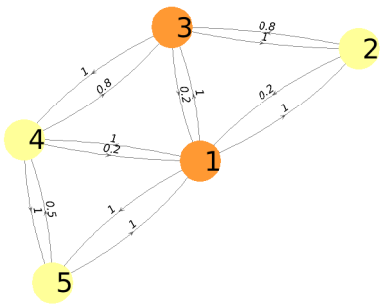


Figure 7.0.3: Covoiturage de cinq personnes avec deux voitures et intolérances.

La frustration totale trouvée est minimale et vaut $\frac{1+0,2+1+0,5}{10,7} \simeq 0,25$. Le pauvre Bob (nœud 1) qui, rappelons-le, déteste tout le monde, est contraint de partager une voiture.

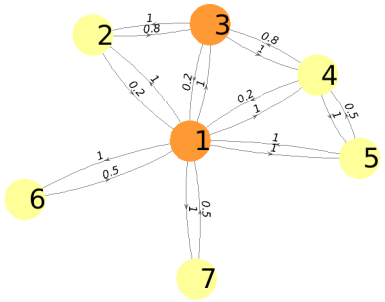
Bien souvent il y a cependant des contraintes qui imposent une borne inférieure au nombre de couleurs (de ressources). Reprenons l'exemple du covoiturage mais avec sept personnes. La figure 7.0.4 nous montre d'abord une solution avec deux voitures, qui est inacceptable si chaque voiture ne peut transporter que quatre personnes. Il faut trois voitures, Bob étant seul dans la sienne et la frustration est alors même nulle.

En pratique, donc, on peut être amené à lancer un post-traitement vérifiant si les contraintes supplémentaires sont respectées et, si elles ne le sont pas, à relancer l'algorithme avec plus de couleurs/ressources. Ceci peut évidemment être automatisé mais si le nombre de ressources est imposé, il y a deux cas de figure :

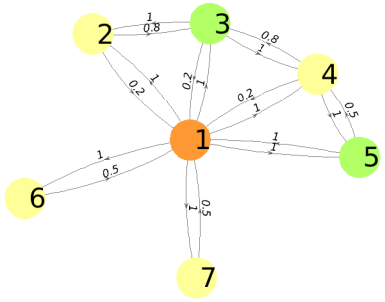
- il n'y a pas de solution sous cette contrainte ;
- il y a une solution, mais l'algorithme a été incapable de la trouver.

Avec un algorithme déterministe sans garantie on peut tenter d'en essayer d'autres de même type car, comme nous l'avons vu, ils ne sont pas tous équivalents. Mais si l'algorithme est déterministe avec garantie, c'est sans espoir. Cependant, en général, il est rarement possible d'utiliser un tel algorithme sur des grands problèmes, alors il reste le recours aux algorithmes stochastiques, en les relançant plusieurs fois. et en comptant sur la chance. Ou le recours aux algorithmes quantiques, plus rapides et de convergence plus assurée, mais à condition de disposer d'un véritable ordinateur quantique et non d'un simple simulateur.

Pour conclure cet bref aperçu sur les algorithmes diplomates, notons que le champ d'application n'est pas restreint aux problèmes de coloriages de graphes, même valués. Sont concernés tous les problèmes de satisfaction de contraintes



(a) Avec deux voitures, mais l'une des deux doit emporter cinq personnes.



(b) Avec trois voitures. La meilleure solution est encore que Bob soit seul dans la sienne et la frustration est alors nulle.

Figure 7.0.4: Covoiturage sept personnes et intolérances.

– pour lesquels il existe déjà un vaste corpus de méthodes – dans le cas où il n'existe pas, justement, de solution totalement satisfaisante. On entre alors là dans le domaine de l'intelligence artificielle.

Et c'est une tout autre histoire.

Chapitre 8

Annexe

Ce chapitre présente quelques dénombrements, une étude mathémathique sous-tendant les deux algorithmes quantiques du chapitre 6, et un certain nombre de codes sources, afin que le lecteur puisse, s'il le souhaite, vérifier les résultats donnés et, éventuellement, procéder à ses propres expérimentations et améliorations.

Rappelons que « coloriage admissible de degré K » est juste un raccourci pour « codage entier de coloriage, ne contenant que les entiers de 1 à K (ou de 0 à $K-1$) et chacun au moins une fois ». Ceci est étroitement lié à la notion d'entiers pannumériques. Le nombre de coloriages admissibles est une indication de la difficulté du problème. Contrairement à ce que l'on pourrait croire, il n'augmente pas systématiquement avec K.

8.1 Entiers pannumériques

La définition générale : un entier pannumérique est un nombre entier dont l'écriture comporte tous les chiffres (avec ou sans 0) dans une base donnée.

Mais dans le contexte de la recherche séquentielle d'un coloriage optimal (voir 4.2.11.2) modifions-la un peu : on considère les nombres dont l'écriture en base K comprend exactement N chiffres, avec éventuellement des zéros en tête, et qui contient tous les chiffres de 0 à $K-1$. Un tel nombre pourrait être appelé NK-pannumérique, mais par la suite nous dirons simplement pannumérique.

Par exemple, et selon cette définition, pour $N = 7$ et $K = 6$ le nombre 0035412 est pannumérique, mais pas 0035422 (il manque le 1).

Ainsi, il y a bijection entre les coloriages admissibles et les entiers pannumériques. Par exemple, pour un graphe à sept nœuds le coloriage $(1, 1, 4, 6, 5, 2, 3)$ est admissible et « identifiable » au pannumérique 0035412. Ceci nous permet quelques calculs intéressants en particulier concernant les intervalles de coloriages non admissibles (voir ci-dessous 8.4).

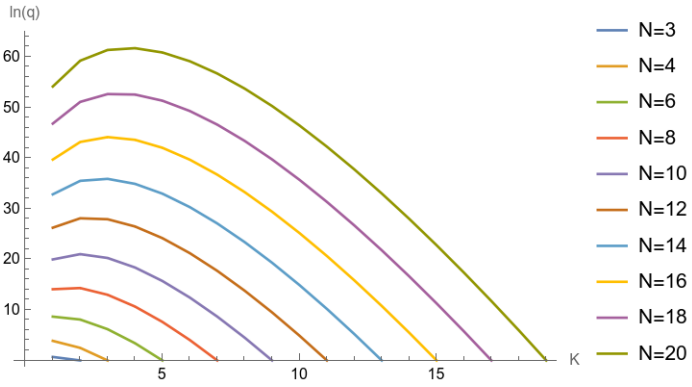


Figure 8.2.1: Nombre de classes d'équivalence (échelle logarithmique).

8.2 Équivalences

Considérons un coloriage de degré K. Il définit un sous-ensemble N_1 de nœuds de couleur 1, un sous-ensemble N_2 nœuds de couleur 2, ..., un sous-ensemble N_K de nœuds de couleur K. Cela peut être représenté par la liste $L_K = (1, 2, \ldots, K)$ dont l'élément de rang k est la couleur commune aux nœuds du sous-ensemble N_k.

Alors les $K!$ permutations de L_K correspondent à des coloriages tous équivalents entre eux. Comme les « couleurs » sont arbitraires si, par exemple, on remplace tous les 2 par des 1 et simultanément tous les 1 par des 2, cela ne change pas vraiment le coloriage.

Ainsi chaque coloriage de degré K appartient à une classe d'équivalence de taille $K!$. En résumé, l'ensemble des coloriages de degré K peut être partitionné en classes d'équivalence au nombre de

$$q_{N,K} = \frac{A_{N,K}}{K!} = K^{N-K} \prod_{i=K+1}^{N} i$$

Parmi celles-ci une et une seule contient des coloriages optimaux. Un algorithme de recherche devra la trouver ou, plus précisément, un de ses représentants.

Comme on peut le voir sur la figure 8.2.1 pour K donné $q_{N,K}$ augmente rapidement avec N, mais pour N donné il passe par un maximum pour une faible valeur de K, pour ensuite décroître assez vite.

8.3 Coloriages admissibles

Cherchons le nombre total de coloriages admissibles pour un graphe à N nœuds. Un réflexe possible est de dire que chaque nœud peut prendre au plus N valeurs et le nombre de séquences de N entiers pris dans $\{1, 2, \ldots N\}$est alors

$$A(N) = N^N$$

Mais une séquence n'est pas forcément un coloriage admissible. Pour un coloriage de degré K nous avons la contrainte que chaque chiffre de $\{1, 2, \ldots, K\}$ doit être utilisé au moins une fois.

En fait ce problème est celui du calcul du nombre de surjections $S(N, K)$ d'un ensemble à N éléments sur un ensemble à K éléments. En effet si l'on considère une surjection de $\{1, \ldots, N\}$ sur $\{1, \ldots, K\}$, on voit bien que les antécédents de cette surjection satisfont précisément la contrainte indiquée.

La formule de calcul est classique :

$$S(N, K) = \sum_{i=1}^{K} (-1)^{K-i} \left(\begin{array}{c} K \\ i \end{array} \right) i^N \tag{8.3.1}$$

qui peut, par exemple, se démontrer par récurrence en utilisant le fait que

$$K^N = \sum_{i=1}^{K} \left(\begin{array}{c} K \\ i \end{array} \right) S(N, i)$$

ou grâce à la formule du crible de Poincaré.

Finalement le nombre total de coloriages admissibles pour un graphe à N nœuds est la somme de ceux pour chaque degré K

$$S_{tot}(N) = \sum_{K=1}^{N} S(N, K) \tag{8.3.2}$$

8.4 Intervalles non admissibles

En recherche séquentielle (section 4.2.11.2), appelons « saut » le nombre de coloriages entre deux admissibles. Quel est le saut maximum ?

On ne s'intéresse ici qu'au cas $2 < K < N$. En effet, pour $K = N$ trouver un coloriage optimal est trivial. Par ailleurs, pour $K = 2$, tous les sauts sont nuls.

Il est alors facile de voir que le plus grand saut se produit pour $K = N - 1$ et se situe entre le coloriage $C_1 = (0, 0, N-2, N-3, \ldots, 1)$ et $C_2 = (0, 1, 0, 2, 3, \ldots, N-2)$.

Les codes de ces deux coloriages sont

$$\begin{array}{rcl} c_1 & = & \sum_{i=1}^{N-2} i\,(N-1)^{i-1} \\ & = & (N-1)^{N-2} - \frac{(N-1)^{N-2}-1}{(N-2)^2} \end{array}$$

et

$$\begin{array}{rcl} c_2 & = & (N-1)^{N-2} + \sum_{i=2}^{N-2} i\,(N-1)^{N-2-i} \\ & = & \frac{1}{N-2}\left((N-1)^{N-1} + 2\,(N-1)^{N-3} - (N-1)^{N-2}\right) + \frac{(N-1)^{N-3}-N+1}{(N-2)^2} - 1 \end{array}$$

Par exemple, pour $N = 7$, on considère donc, en base six, les nombres 0054321 et 0102345, soit, en base dix, 7465 et 8345. Leur différence 880 indique qu'il y a 879 non admissibles entre eux.

La taille de ce saut maximum augmente exponentiellement avec N. Pour $N = 10$ elle vaut déjà 1943079 .

Une application évidente est que si l'on fait une recherche séquentielle d'un coloriage optimal et si l'on arrive jusqu'à C_1 (parce que l'on n'a pas trouvé de solution pour $K < N - 1$) alors on peut passer directement à C_2.

Mais pourquoi s'arrêter en si bon chemin ? On peut appliquer un calcul analogue pour $K = N - 2$, $N - 3$, ..., 3. Faisons alors la démarche inverse, en partant de $K = 3$, puisque la recherche séquentielle teste des valeurs croissantes de K. Ainsi, d'une manière générale, si l'on arrive à $C_1 = (0, \ldots, K-1, K-2, \ldots, 1)$ on peut sauter directement à $C_2 = (0, \ldots, 1, 0, 2, 3, \ldots, K-1)$. Cependant pour chaque valeur de K on ne saute qu'une seule séquence et, bien que ce soit la plus grande, le gain reste faible, ou, plus précisément, il n'est intéressant que si le nombre chromatique est juste légèrement inférieur à N. En effet, on doit tester alors d'autant plus de valeurs de K et, donc, on effectue plus de sauts.

8.5 Coloriages valides

Pour un coloriage, le fait d'être admissible ne dépend pas de la structure du graphe étudié, seulement de son nombre de nœuds, mais bien sûr ce qui est réellement recherché est un coloriage valide et ayant un minimum de couleurs.

Il est possible de définir les conditions de validité d'un coloriage par une approche purement matricielle. Cela est utile par exemple pour une résolution par programmation linéaire et également pour une approche quantique. Le coloriage C est supposé codé par sa matrice binaire (qui a, rappelons-le, une et une seule valeur 1 par ligne, les autres étant nulles) et le graphe G par sa matrice d'adjacence.

Nous définissons

$$\Sigma = C \oplus C \tag{8.5.1}$$

$$G_K = \sqcup_K G \tag{8.5.2}$$

$$\Gamma = G_K \odot \Sigma \tag{8.5.3}$$

où $\oplus$ est la somme extérieure, $\sqcup_K$ l'opérateur de concaténation « horizontale » répété K fois, et $\odot$ le produit élément par élément.

Ensuite l'indicateur est

$$v_{C,G} = \max(\Gamma) \tag{8.5.4}$$

de telle sorte que la validité d'un coloriage C pour le graphe G soit donnée par la condition

$$v_{C,G} < 2 \tag{8.5.5}$$

Ce qui est équivalent à

$$v_{C,G} = 0 \vee v_{C,G} = 1 \tag{8.5.6}$$

qui est plus simple à vérifier par manipulation de qubits. On peut aussi utiliser une somme extérieure modifiée de telle sorte que Γ soit binaire. La condition est alors simplement

$$\Gamma = \mathbf{0} \tag{8.5.7}$$

où $\mathbf{0}$ est une matrice nulle.

L'algorithme en langage Octave/Matlab©

```
Sigma=outerSum(C);
% Variant:
% Sigma=max(0,Sigma-1);
GK=repmat(G, K,1);
Gamma=GK.*Sigma;
valid=max(Gamma(:))<2 ;
% Variants:
% valid=max(Gamma(:))<1; % valid=Gamma=zeros(size(C));
...
function Sigma=outerSum(C)
 % C'est une version simplifiée
 % La version normale complète admet deux matrices en entrée
[~,K]=size(C);

    Sigma=[];
        for k=1:K
           Ck=meshgrid(C(:,k));
           Ck=Ck+Ck';
           Sigma=[Sigma Ck];
        end

end
```

Preuve

1. Les seules valeurs possibles dans Σ sont 0, 1 et 2.
2. Les seules valeurs possibles dans Γ sont 0, 1 et 2.
3. Considérons $\Sigma(n,m)$. Nous avons $m = (k-1)N+j$, avec $j \in [1,2,\cdots,N]$.
   (a) Si $\Sigma(n,m) = 0$ ni le nœud n ni le nœud j ont la couleur k.
4. Si $\Sigma(n,m) = 1$ le nœud n a la couleur k et le nœud j une autre ou vice versa.
5. Si $\Sigma(n,m) = 2$ les nœuds n et j ont la couleur k (ce qui est bien sûr également vrai quand $n = j$).

6. La valeur 2 est « éliminée » dans Γ ssi (si et seulement si) $\Gamma(n,m)=0$, i.e. $G(n,j)=0$, qui signifie « pas d'arête entre les nœuds n et j ».

7. Donc, s'il n'y a pas de valeur 2 dans Γ alors le coloriage C est valide.

Exemple pour un 3-coloriage (voir la figure 8.5.1)

$$G=\begin{pmatrix}0&0&1&1\\0&0&1&1\\1&1&0&1\\1&1&1&0\end{pmatrix}$$

$$C=\begin{pmatrix}1&0&0\\1&0&0\\0&1&0\\0&0&1\end{pmatrix}$$

$$\Sigma=\begin{pmatrix}1&1&1&1\\1&1&1&1\\0&0&0&0\\0&0&0&0\end{pmatrix}+\begin{pmatrix}1&1&0&0\\1&1&0&0\\1&1&0&0\\1&1&0&0\end{pmatrix}$$

$$\sqcup\begin{pmatrix}0&0&0&0\\0&0&0&0\\1&1&1&1\\0&0&0&0\end{pmatrix}+\begin{pmatrix}0&0&1&0\\0&0&1&0\\0&0&1&0\\0&0&1&0\end{pmatrix}$$

$$\sqcup\begin{pmatrix}0&0&0&0\\0&0&0&0\\0&0&0&0\\1&1&1&1\end{pmatrix}+\begin{pmatrix}0&0&0&1\\0&0&0&1\\0&0&0&1\\0&0&0&1\end{pmatrix}$$

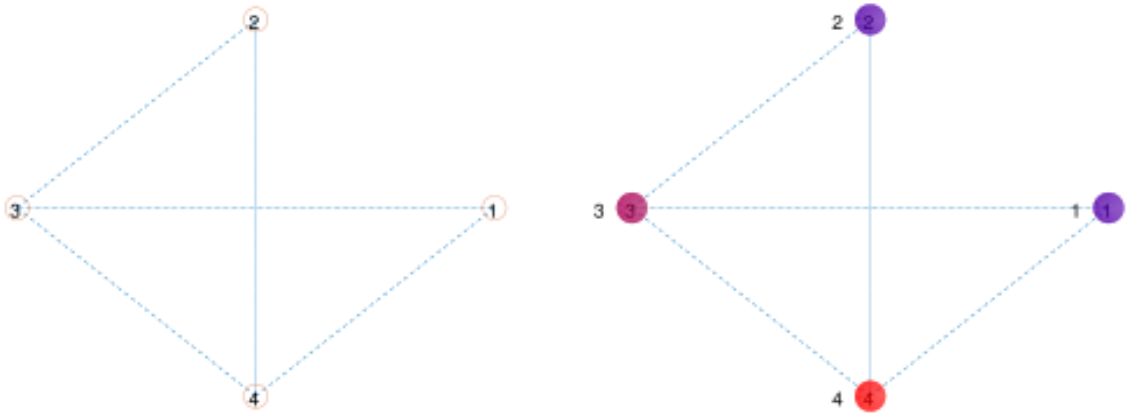
$$=\begin{pmatrix}2&2&1&1&0&0&1&0&0&0&0&1\\2&2&1&1&0&0&1&0&0&0&0&1\\1&1&0&0&1&1&2&1&0&0&0&1\\1&1&0&0&0&0&1&0&1&1&1&2\end{pmatrix}$$

$$G_K=\begin{pmatrix}0&0&1&1&0&0&1&1&0&0&1&1\\0&0&1&1&0&0&1&1&0&0&1&1\\1&1&0&1&1&1&0&1&1&1&0&1\\1&1&1&0&1&1&1&0&1&1&1&0\end{pmatrix}$$

$$\Gamma=\begin{pmatrix}0&0&1&1&0&0&1&0&0&0&0&1\\0&0&1&1&0&0&1&0&0&0&0&1\\1&1&0&0&1&1&0&1&0&0&0&1\\1&1&0&0&0&0&1&0&1&1&1&0\end{pmatrix}$$

Notez que les valeurs 2 de Σ sont « éliminées » grâce au produit par G_K élément par élément. Regardons soigneusement comment

Σ est construite afin de trouver les indices des valeurs 2. Pour chaque couleur k :



(a) Le graphe (b) Un 3-coloriage possible

Figure 8.5.1: Coloriage optimal d'un graphe 4 nœuds 5 arêtes.

- dupliquer «horizontalement » N fois la colonne correspondante de la matrice C afin de construire une matrice carrée Σ_k;
- l'ajouter à sa transposée ;
- si $k > 1$,concaténer « horizontalement » le résultat au précédent.

Ainsi une valeur 2 est générée ssi nous avons

$$\Sigma_k(i,j) = \Sigma_k(j,i) = 1 \tag{8.5.8}$$

Mais $\Sigma_k(i,j) = \Sigma_k(i,1)$ et $\Sigma_k(j,i) = \Sigma_k(j,1)$, indiquant que la même couleur k est donnée aux nœuds i et j. Ainsi, après un long détour, la règle est finalement très simple :

Il y a une valeur 2 dans Σ ssi deux nœuds ont la même couleur.

Cela est utile entre autres pour construire un circuit quantique. Voici un exemple de coloriage invalide sur notre graphe 4 nœuds 5 arêtes.

$$C = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\Gamma = \begin{pmatrix} 0 & 0 & 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

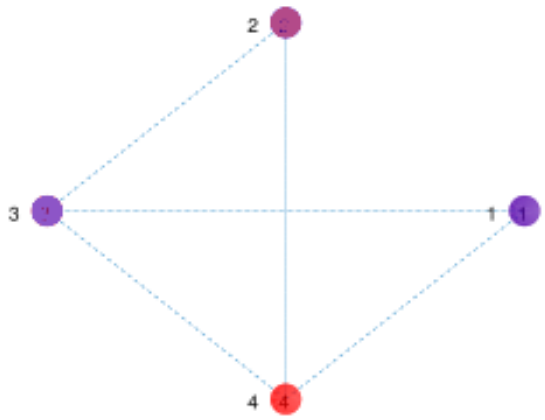


Figure 8.5.2: Un 3-coloriage invalide. Les nœud 1 et 3 ont la même couleur, alors qu'ils sont reliés par une arête.

Polynôme chromatique
Un dénombrement classique est celui qui fait appel au polynôme chromatique $H_G(K)$ qui donne le nombre de coloriages valides de degré inférieur ou égal à K. Par exemple pour le graphe triangulaire on a

$$H_G(K) = K(K-1)(K-2)$$

qui montre bien qu'il n'y a pas de coloriage valide avec une ou deux couleurs et qu'il y en a six avec trois couleurs. Ici, comme tous les 3-coloriages sont valides, on peut vérifier que $A_{N,3} = P_G(3)$.
Bien sûr ce n'est pas toujours aussi simple. Par exemple pour le graphe cubique on a

$$H_G(K) = K^8 - 12K^7 + 66K^6 - 214K^5 + 441K^4 - 572K^3 + 423K^2 - 133K \tag{8.5.9}$$

En général le polynôme chromatique est donc assez compliqué. Il peut se construire par récursion sur des opérations de suppressions d'arêtes et de contractions de graphe. Le lien avec le nombre chromatique est le fait que ce dernier est le plus petit entier pour lequel le polynôme est positif. Une fois le polynôme construit, il est donc facile d'en déduire le nombre chromatique, par exemple pour vérifier si un algorithme de coloriage trouve une solution optimale ou non ((un code source utilisable pour de petits graphes est donné en annexe 8.7.4).
Ainsi, pour le graphe cubique, le polynôme est nul pour $K = 0$ et $K = 1$ (qui de toute façon, est forcément inacceptable pour un graphe connecté) et vaut 1 pour $K = 2$. Il est donc certain qu'il est 2-coloriable.

Il serait intéressant d'avoir une formule pour le nombre de coloriages valides $V(N, K)$. On a de façon évidente

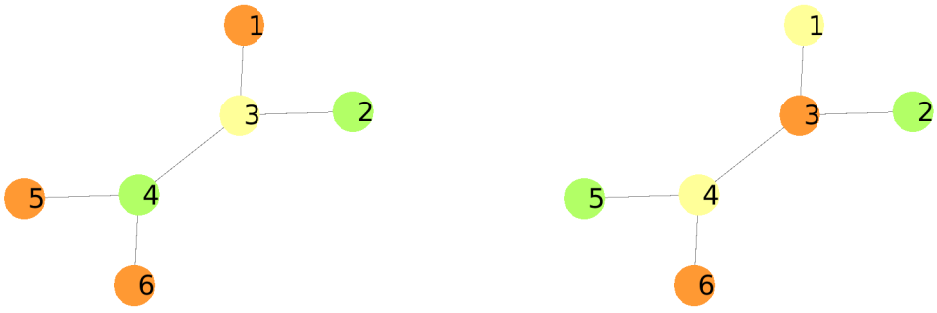
$$V(N, N) = N!$$

car puisque si tous les nœuds sont de couleurs différentes, on peut permuter celles-ci sans risque.

Plus généralement supposons que nous ayons un coloriage valide à K couleurs, pas nécessairement optimal. Il définit une partition des N nœuds en K sous-ensembles de même couleur de tailles $(N_1, ..., N_K)$. Toute permutation de couleurs entre ces ensemble définit un autre coloriage valide et tous ces coloriages ainsi générés sont équivalents. . Donc on a

$$V(N, K) \geq K! \tag{8.5.10}$$

La valeur $K!$ est juste une borne inférieure, car il peut y avoir d'autres partitions valides, comme le montre la figure 8.5.3.



(a) Partition $(3, 1, 2)$ (b) Partition (2,2,2)

Figure 8.5.3: Un même graphe peut supporter plusieurs coloriages valides non équivalents

8.6 Estimation de difficulté

Pour avoir une idée de la difficulté du problème du coloriage, nous pouvons comparer, pour un nombre de nœuds donné N, le nombre de coloriages valides au nombre d'admissibles. Mais pour cela il nous faut le nombre de coloriages valides ou, à défaut, une estimation, même grossière, alors que nous n'avons que

la borne inférieure. À ce jour[1], en effet, il n'y a apparemment pas de formule exacte publiée.

Commençons par estimer la densité moyenne des N-graphes connexes. Pour ces graphes le nombre minimum d'arêtes est $N-1$, d'où une densité égale à 2/N, et le nombre maximum $N*(N-1)/2$, donnant une densité de 1. On en déduit que la densité moyenne est

$$d(N) = \frac{1}{2} + \frac{1}{N} \tag{8.6.1}$$

Maintenant inspirons-nous d'une étude expérimentale (**(author?)** [14]), suggérant une inégalité empirique de la forme suivante

$$V(N,K) < p + p^{\beta}$$

où p est le nombre de coloriages optimums et $\beta \simeq 2$. Cependant elle est difficilement applicable, car trouver p devient en pratique impossible dès que N augmente trop.

8.7 Codes sources

Les codes sont écrits en langage Octave/Matlab©. Cependant certaines instructions sont spécifiques à Matlab et ne fonctionnent pas sous Octave, essentiellement celles pour dessiner les graphes. Quelques petits programmes simples ont été écrits dans le langage de Mathematica©, mais ne sont pas donnés ici.

8.7.1 Génération aléatoire d'un graphe

Rappelons que nous ne considérons que les graphes non orientés connexes sans boucle.

```
function edges=randConnect(N)
% Generate a connected graph with N nodes
Plot=false;
% Connect node 1 to another one j, at random
connected(1)=1;
j=randi(N-1)+1;
connected(j)=1;
notConnectedList=2:N; % Remove 1
notConnectedList=notConnectedList(notConnectedList~=j); % Remove j
nbNotConnected=N-2;
connectedList=[1 j ];
nbConnected=2;
edges(1,1)=1; edges(1,2)=j;
nEdg=1; % Number of edges
% Connect the others
```

[1] 25 janvier 2023

```
while nbNotConnected>0
  % fprintf('\n Connected \n'); disp(connectedList);
  %fprintf('\n Not connected \n'); disp(notConnectedList);

  % Select at random a non connected j
  j=notConnectedList(randi(nbNotConnected));
  %fprintf('\n j %i',j)
  % Randomly choose the number of links
  nbLinks=randi(nbConnected);

  % Randomly select nbLinks origins
  origins=randperm(nbConnected,nbLinks);
  %fprintf('\n  to connnect to \n'); disp(origins)

  % For each origin, create the link to j
  for n=1:nbLinks
    i =connectedList(origins(n));

    % Save the new edge
    nEdg=nEdg+1;
    edges(nEdg,1)=i; % i is connected ...
    edges(nEdg,2)=j; % ... to j
  end

  % j is now connected
  nbConnected=nbConnected+1;
  connectedList=[connectedList j];

  % Remove j from the list of unconnected
  notConnectedList=notConnectedList(notConnectedList~=j);
  nbNotConnected=nbNotConnected-1;
end
if Plot
  figure;
  G=graph(edges(:,1),edges(:,2));
  h=plot(G,"-o",'MarkerSize',30,'NodeColor',[0.9 0.9 0.9]);
  h.NodeFontSize = 20;
end
end
```

Cette méthode de construction appelle quelques commentaires. Dans quelle mesure, pour un nombre de nœuds N donné, les graphes obtenus sont-ils vraiment aléatoires ? Ici « aléatoire » est un raccourci pour « engendré par un processus aléatoire uniforme ».

Nous avons vu que tout graphe peut être décrit par une séquence binaire s. En pratique, comme nous ne considérons que des graphes non orientés sans

boucle on utilise simplement la séquence donnée par le triangle supérieur droit de la matrice d'adjacence. La séquence est alors de longueur $n = \frac{N(N-1)}{2}$.

Soit m le nombre de 1 de cette séquence. Si cette dernière est aléatoire, la probabilité que $m = k$ suit une loi de Bernouilli, en l'occurence ici

$$\mathbb{P}(m = k) = \begin{pmatrix} m \\ k \end{pmatrix} \frac{1}{2^n}$$

Simulons par exemple le processus comme si nous voulions générer 10^6 graphes quelconques de $N = 30$ nœuds purement aléatoirement, sans contrainte de connexité, en générant des séquences de $n = \frac{30\times29}{2} = 435$ bits et traçons l'histogramme des rapports

$$p = \frac{m}{n}$$

Alors m est le nombre d'arêtes du graphe représenté par la séquence binaire. Comme prévu, on obtient bien une distribution de Bernouilli, symétrique de moyenne 1/2.

Maintenant procédons de même en utilisant le code de génération de graphes connexes ci-dessus. Dans ce cas on a

$$m \geq N - 1$$

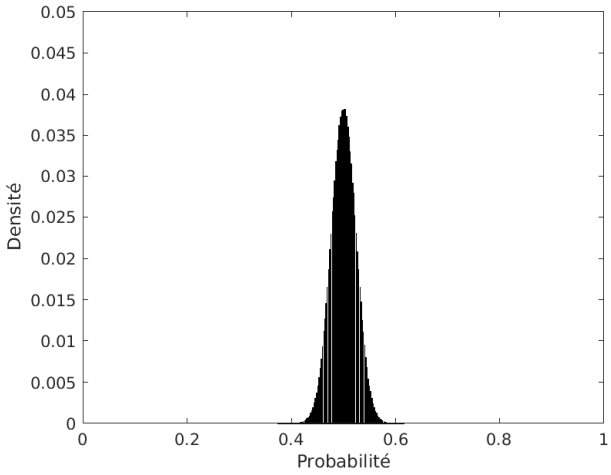
ce qui implique

$$p \geq \frac{2}{N}$$

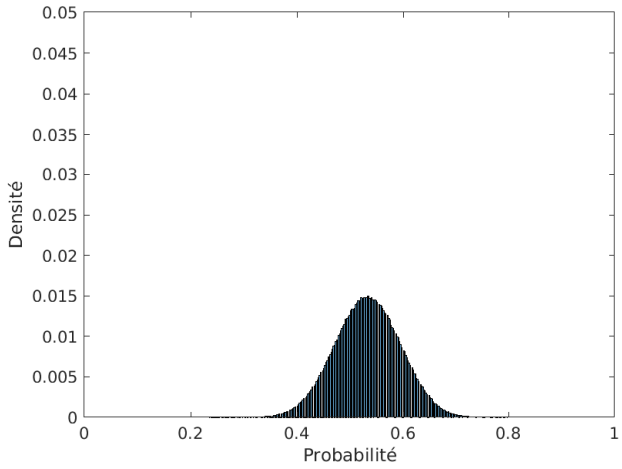
Si le code génére bien des graphes aléatoires connexes, on s'attend à ce que l'histogramme obtenu ait l'allure d'une distribution de Bernouilli de moyenne $\left(\frac{2}{N} + 1\right)/2$ soit, dans notre exemple, 0,53333. Et c'est en effet ce que nous trouvons. La distribution de la figure 8.7.1 a la « bonne allure » et sa moyenne est 0,533295.

8.7.2 Coloriage équivalent

```
%{
(Modified to more easily generate C++ code)
Equivalent coloring, but so that it is as "small" as possible
(according to the later coding of Cequiv by an unique integer)
Example
[5 5 4 3 3 2 2 2 2 4] => [1 1 2 3 3 4 4 4 4 2 )
%}
% Would be quicker by using unique(), but there is an issue when
% translating it into C++ with Matlab Coder.
Cequiv=zeros(1,N);
ind=find(C==C(1));
Cequiv(ind)=1;
```



(a) Graphes quelconques



(b) Graphes connexes sans boucle

Figure 8.7.1: Graphes de 30 nœuds. Distribution du nombre d'arêtes, estimée sur 10^6 générations aléatoires. Pour les graphes connexes sans boucle la moyenne est nécessairement supérieure à 1/2.

```
k=1;
for n=2:N
    if Cequiv(n)>0 continue;end
        k=k+1;
        ind=find(C==C(n));
        Cequiv(ind)=k;
    end
end
```

8.7.3 Nombre de graphes connexes

C'est juste l'application directe de la formule classique. En pratique, sauf sur un ordinateur très puissant, ce code est inexploitable au-delà de quelques dizaines de nœuds. Notez que le nombre calculé correspond aux graphes structurellement différents. Pour certains algorithmes la numérotation des nœuds a une influence sur l'efficacité. Alors le nombre de graphes qu'ils « voient » comme différents est en fait $TN!$.

```
% Code Octave/Matlab
function T=nbConnectG(N)
%{
Number of undirected connected graphs with N nodes
See http://oeis.org/A001187)
1, 1, 4, 38, 728, 26704, 1866256, ...
Recursive code
%}
T=2^(N*(N-1)/2);
for k=1:N-1
c1=(N-k)*(N-k-1)/2;
u=nchoosek(N-1,k-1)*2^c1*nbConnectG(k);
T=T-u;
end
end
```

Envisager tous les graphes connexes d'ordre N suggère une expérience de pensée intéressante. Bien sûr les nombres deviennent vite énormes ($1,57 \times 10^{57}$ pour $N = 20$),mais supposons qu'un utilisateur ait souvent à chercher un coloriage optimal pour des graphes de taille toujours au plus égale à un certain N. Comme nous l'avons vu, il est possible de coder chaque graphe par un entier unique.

Alors on peut imaginer que, une fois pour toute, on établisse une (gigantesque !) liste triée contenant, pour chaque graphe, son code et un coloriage optimal trouvé par une méthode sûre (et donc longue).

Encore faut-il ensuite être capable, si l'on doit étudier un graphe donné, de le retrouver dans la liste. Considérons par exemple la recherche dichotomique.

Le nombre d'essais maximum t est tel que

$$2^t \leq T(N) \leq 2^{t+1}$$

soit

$$t \simeq \frac{\ln(T(N))}{\ln(2)}$$

Or il se trouve que $\ln(T(N))$ suit une loi quadratique (voir la figure 8.7.2). Ainsi, une fois la liste constituée – mais c'est malheureusement irréaliste dès que N est un peu grand – la recherche d'un coloriage optimal est elle-même de complexité quadratique.

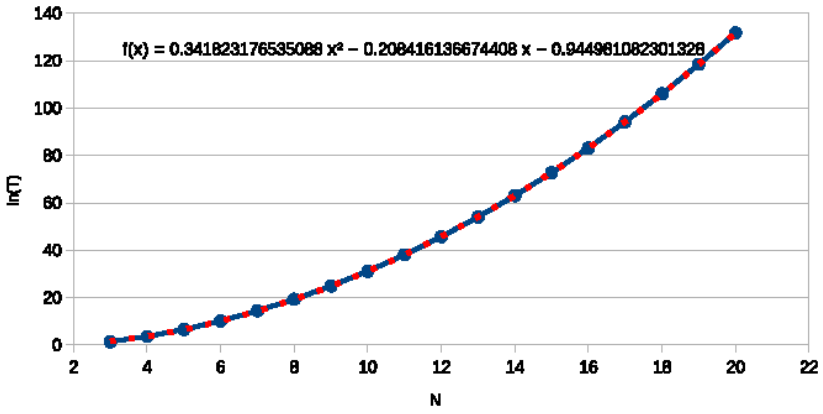


Figure 8.7.2: Logarithme du nombre de graphes connexes en fonction du nombre de nœuds.

8.7.4 Polynôme chromatique

```
function chi=chromaticNumber(G)
% G is a binary adjacency matrix
% Matlab command:
P=chromaticPoly(G); % Build the chromatic polynom
N=length(P);
% Find the smallest integer chi so that P(chi)>0
% This is the chromatic number
for chi=1:N-1 % In fact 1 can not be acceptable for a connected graph
  pnx=evalP(P,N,chi);
  if evalP(P,N,chi)>0 return; end
end
end
%=========================================================================
function Px=evalP(P,N,x)
```

```
Px=0;
u=1;
for n=2:N-1
  u=u*x+P(n);
end
Px=u+P(N);
end
```

8.7.5 Algorithmes déterministes

Certaines instructions peuvent ne pas fonctionner sous Octave, en particulier pour les graphiques.

Pour les algorithmes garantis (Programmation linéaire et Recherche séquentielle) ayant des temps de calcul forcément longs, en pratique on peut générer des fichiers .mex C++ compilés sous Matlab. De plus les codes pourraient être écrits de manière plus concise, à l'aide d'instructions spécifiques, mais j'ai évité de le faire pour faciliter une éventuelle traduction vers d'autres langages.

8.7.5.1 Programmation linéaire

```
function CC=colorAlgoLP(G) % If called by graphColor
%{
   2021-12 Maurice Clerc
Use linear programming under constraints for optimal graph coloring.
The solution is sure, but it takes a very long time when N increases.

Note: this code is neither elegant nor concise, so that it should be
       easier to understand. The only difficult point is how to build
       the matrix A for inequalities constraints.

  G     = adjacency matrix of the graph

%}
[N,~]=size(G);
CC=zeros(1,N);
if N>30 % Of course, it depends on your computer ...
  fprintf('\n WARNING: It may be looong!'); fprintf('\n')
end
K=2; % To start, try this number of colors. You may perform first
     % a structural analysis, in order to find a higher lower bound.
OK=false;
while ~OK
  NK=N*K;
  %-------------------------------------------------------------------------
  % Build Aeq and beq so that Aeq*x=beq
 % fprintf('\n Build Aeq and beq, for equality constraints');
```

```
beq=ones(N,1);
Aeq=zeros(N,NK);
for i=1:N
  j1=K*(i-1);
  for j=j1+1:j1+K
    Aeq(i,j)=1;
  end
end

s=sum(G(:));
A=zeros(s ,NK);

edg=0;
for i=1:N-1
  for j=i+1:N
    if G(i,j)==0 continue; end
    edg=edg+1;
    for k=1:K
      rxi=(i-1)*K + k;
      rxj=(j-1)*K + k;
      A(edg+k-1,rxi)=1;
      A(edg+k-1,rxj)=1;
    end
    edg=edg+K-1;
  end
end

[s,~]=size(A);
b=ones(s,1);

% Now we try to solve
intcon=1:NK; % All xi are integers
lb=zeros(1,NK); % Actually the only acceptable values are 0 and 1
ub=ones(1,NK);

options=optimoptions(@intlinprog,'Display','off');
%options=optimoptions(@intlinprog,'Display','iter');
%  If A is sparse it could be better to use mldivide or decomposition.
[x,~,exitflag] = intlinprog([],intcon,A,b,Aeq,beq,lb,ub,options);

if exitflag~=1
  %fprintf('\n No valid coloring with %i colors \n',K)
  K=K+1; % So we try with more colors
else
  % A solution has been found. Build the NxK binary representation
  OK=true;
```

```
    C=reshape(x,K,N);
    Ct=C';
   [~,l]=size(Ct);
    % Convert the binary representation into the classical one
    %try
    for i=1:N
      CC(i)=find(Ct(i,1:l)>0);
    end


    %{
    % Check the validity, according to the mathematical proof
    GK=G;
    for k=1:K-1
     GK=[GK,G]; % size N,N*K
    end
    Sigma=outerSum(C,C);
    Gamma=GK.*Sigma;
    m=max(Gamma(:));
    if m>1
      fprintf('\n Invalid coloring ');
      fprintf('\n There should be no 2 here: \n');
      disp(Gamma);
    end
    %}

   % disp(Aeq)
   % disp(A)

  end
end
```

8.7.5.2 Glouton 0

```
function C=colorAlgo0(G)
% Classical Greedy
fprintf('\n Algo 0');
  [N,~]=size(G);

sequence=false; % If true plot the successive colorings
                % Just to see what happens. Do NOT use it if N is big!
% Check
if sequence && N>10
  fprintf('\n Risk of out of memory')
  error(" ")
end
```

```
edges=G2edges(G); % Just for plot
 C=zeros(1,N); % Initial colors

if sequence graphPlot(edges, C); end

  colored=0; % Number of colored nodes

  while colored<N
     for n=1:N
         % List of colors used by its neighbours
         if C(n)>0 continue; end
         L=[];
         for nn=1:N
           if G(n,nn)==1 && C(nn)>0
             L=[L C(nn)];
           end
         end % for nn=1:N

         if isempty(L)
          k=1;
          else
            % Find the smallest color k that is NOT used by its neighbours
            % and assign it to n
            for k=1:N
             I=find(L==k);
              if isempty(I)
                break;
             end
             end %  for c=1:N
         end % if isempty(L)

          C(n)=k;
          colored=colored+1;
          %fprintf('\n %i colored %i',n,k);
     if sequence graphPlot(edges, C); end

     end %  for n=1:N
  end % while colored<N

end
%--------------------------------------------------
function graphPlot(edges,C)
%{
   Plot a colored graph
    edges = edges of the graph
    C = colors of the nodes. May be empty
```

```
Warning: specific to Matlab. Does not work with Octave (2021)
%}
colors=[240 240 240;
  255 0 0;160 160 160;
  255 255 0;0 255 0;
  0 128 255;102 0 204;
  204 153 255; 255 153 153;
  255 255 204;229 255 204;
  153 204 255
  ];
colors=colors/255;

G=graph(edges(:,1),edges(:,2));
N=length(C);
figure('Color',[1 1 1]); % White background
if ~isempty(C)
   h=plot(G,"-o",'EdgeColor','k');
  h.MarkerSize=30;
   h.NodeFontSize = 20;
  %h.NodeLabel = {}; % Remove the labels of the nodes
  % Coloring
    if N<=length(colors)
      h.NodeColor=colors(C+1,:);
    else
       h.NodeCData=C;
    end

else
  h=plot(G,"-o",'EdgeColor','w');
  h.NodeColor="black";
  h.MarkerSize=32;
   h.NodeLabel = {};
  hold on
  h2=plot(G,"-o",'MarkerSize',30,'EdgeColor','k');
   h2.NodeColor="white";
  %h2.NodeFontSize = 20;
  %h2.NodeFontName="Times";
  %h2.NodeFontAngle="normal";
 h.NodeLabel = {};
end
axis off; % Remove the axis
end
```

8.7.5.3 Glouton excentrique

```
function C=colorAlgoEcc(G)
```

```
% Based on Algo 0 (classical greedy), but more eccentric:
% it does not always assign the smallest possible color
fprintf('\n Algo Eccentric');
  [N,~]=size(G);
   C=zeros(1,N); % Initial colors
   C(1)=1;
  colored=1; % Number of colored nodes
  Lused=[1];  % List of colors of colored nodes
  while colored<N
     for n=1:N % For node n
         if C(n)>0 % Already colored
           continue;
         end

         % List of colors used by its neighbours
         Lneigh=[];
         for nn=1:N
           if G(n,nn)==1 && C(nn)>0
             Lneigh=[Lneigh C(nn)];
           end
         end % for nn=1:N
            % Find the HIGHEST color that is already used
            % but NOT by its neighbours
            % and assign it to n
            Lused=sort(Lused,'descend');
            OK=false;
            for k=1:length(Lused)
              I=find(Lneigh==Lused(k));
              if isempty(I)
                OK=true;
                c=Lused(k);
                break;
              end
             end %  for k=1:length(Lused)

             if  ~OK
               c=max(C)+1;
             end

             C(n)=c;
                Lused=[Lused,c];
                colored=colored+1;

     end %  for n=1:N % For node n
  end % while colored<N
end
```

8.7.5.4 Glouton 1

```
function C=colorAlgo1(G)
% Greedy. Nodes sorted once by degree
fprintf('\n Algo 1');
[N,~]=size(G);
C=zeros(1,N);
  colored=0;

  % Sort nodes by degree
  for n=1:N
    deg(n)=sum(G(n,1:N));
  end

  [~,indDeg]=sort(deg,'descend');
   while colored<N
     for m=1:N
       n=indDeg(m);

         % List of colors used by its neighbours
         L=[];
         for nn=1:N
           if G(n,nn)==1 && C(nn)>0
             L=[L C(nn)];
           end
         end % for nn=1:N

         if isempty(L)
          k=1;
          else
            % Find the smallest color k that is NOT used by its neighbours
            % and assign it to n
            for k=1:N
             I=find(L==k);
              if isempty(I)
                break;
              end
            end %  for k=1:N
         end % if isempty(L)

          C(n)=k;
          colored=colored+1;
     end %  for n=1:N
   end % while colored<N
```

8.7.5.5 Glouton 2

```
function C=colorAlgo2(G)
   % Greedy. Color the node that has the max number of non colored neighbours
fprintf('\n Algo 2');
   [N,~]=size(G);
    C=zeros(1,N); % No initial colors
  colored=0; % Number of colored nodes
  while colored<N
    % Find the non colored node n that has the max number
    % of uncolored neighbours
    nMax=0;
    for i=1:N % Loop on nodes
      if C(i)>0 continue; end % If already colored, skip
        nbNeigh=0;
        for nn=1:N % Number of non colored neighbours of i
          if G(i,nn)==1 && C(nn)==0
            nbNeigh=nbNeigh+1;
          end

          if nbNeigh>=nMax
            n=i;
            nMax=nbNeigh;
          end
        end
    end
   %fprintf('\n Node %i:',n)
         % List of colors used by its neighbours
         L=[];
         for nn=1:N
           if G(n,nn)==1 && C(nn)>0
             L=[L C(nn)];
           end
         end % for nn=1:N

         if isempty(L)
          k=1;
          else
            % Find the smallest color k that is NOT used by its neighbours
            % and assign it to n
            for k=1:N
             I=find(L==k);
              if isempty(I)
                break;
              end
            end %  for c=1:N
```

```
          end % if isempty(L)
     %fprintf('  assign color %i',k);
           C(n)=k;
           colored=colored+1;
  end
    end
```

8.7.5.6 Glouton 3

```
function C=colorAlgo3(G)
  % Greedy less sensitive to the numbering of the nodes
fprintf('\n Algo 3');
[N,~]=size(G);
sequence=false; % If true plot the successive colorings
                % Just to see what happens.
                % Do NOT use it if N is big!
% Check
if sequence && N>10
  fprintf('\n Risk of out of memory')
  error(" ")
end
edges=G2edges(G); % Just for plot
   C=zeros(1,N); % Initial colors
   C(1)=1;
  colored=1; % Number of colored nodes
  coloredNodes=1;
  degr=sum(G,1);
  [~,permut]=sort(degr);
if sequence graphPlot(edges, C); end
  while colored<N
    % Consider the set of colored nodes
    % Find a non colored node n that is connected to this set

   % for n=1:N % Loop on nodes
   for n=1:N
      if C(n)>0 continue; end % Already colored => ignore

        % Let Nk be the subset of colored nodes connected to n
        Nk=[];
        for k=1:colored
          c=coloredNodes(k);
         if G(c,n)==1
           Nk=[Nk c];
         end
        end
```

```
          if ~isempty(Nk)
           Ck=C(Nk);

              % Find the smallest color colorK that is not used in Nk
           for colorK=1:N
             used=find(Ck==colorK);
             if isempty(used)
               % Assign colorK to n.
              C(n)=colorK;
              break
             end
           end  % for colorK=1:N

         coloredNodes=[coloredNodes n];
         colored=colored+1;
         if sequence graphPlot(edges, C); end
          end %  if ~isempty(Nk)
     end % for n=1:N % Loop on nodes

   end % while colored<N
   end
```

8.7.5.7 Glouton 4

```
function C=colorAlgo4(G)
fprintf('\n Algo 4');
[N,~]=size(G);
sequence=true; % If true plot the successive colorings
               % Just to see what happens.
               % Do NOT use it if N is big!
% Check
if sequence && N>10
  fprintf('\n Risk of out of memory')
  error(" ")
end
edges=G2edges(G); % Just for plot
% Depth indices
for i=1:N-1
  for j=i+1:N
    depth(i,j)=pathG(G,i,j);
  end
end
  % Initialisation
  C=zeros(1,N) ; % Initial colors
  k=1;
   colored=0;
```

```
if sequence graphPlot(edges, C); end

 while colored<N
   % Find the max depth(i,j) with at least i or j not colored
   minDepth=Inf;
  for i=1:N-1
    for j=i+1:N
      if C(i)*C(j)>0 continue;end % Both nodes are colored
        if depth(i,j)<minDepth
          minDepth=depth(i,j);
        if C(i)==0
          n=i; % Node to color
        else % C(j)=0
         n=j;  % Node to color
        end
      end
    end
  end

   % List of colors used by its neighbours
        L=[];
        for nn=1:N
          if G(n,nn)==1 && C(nn)>0
            L=[L C(nn)];
          end
        end % for nn=1:N

        if isempty(L)
         k=1;
         else
           % Find the smallest color k that is NOT used by its neighbours
           % and assign it to n
           for k=1:N
            I=find(L==k);
             if isempty(I)
               break;
             end
           end %  for k=1:N
        end % if isempty(L)

         C(n)=k;
         colored=colored+1;
         if sequence graphPlot(edges, C); end

 end % while colored<N
```

```
  end
%=========================================================
function l=pathG(G,i,j)
% Length of the shortest path between i and j
% The graph G is connected and undirected
[N,~]=size(G);
if G(i,j)==1
  l=1;
  return
end
% Neighbours
% j is not amongst them
l=2;
 neigh1=[i];
 used=zeros(1,N);
 used(i)=1;
  n1=i;

  isj=false;
  while ~isj
    l1=length(neigh1);
    neigh2=[];

    for n2=1:l1
      for  k=1:N
        if G(neigh1(n2),k)==1
          if k==j
            isj=true;
            break;
          else
            if used(k)==0
              neigh2=[neigh2 k];
              used(k)=1;
            end
          end
        end
      end % for  k=1:N

      if isj
        l=l-1;
        return
      end

    end  % for n2=1:l1
    l=l+1;
    neigh1=neigh2;
```

```
  end % while ~isj
end
```

8.7.5.8 Johnson

```
function C = colorAlgoJohnson(G)
% Thanks to Stephan (2022)
% Warning: this Matlab code may not work for Octave
fprintf('\n Johnson')
  edges=G2edges(G);
 s=edges(:,1);
 t=edges(:,2);
 Graph = graph(s,t);
 N=numnodes(Graph);
% Initialisation
    Ccurrent = 0;
    C = zeros(1,N )';
    C(:) = Inf; % To start, set the colors to infinity
    node = cell(N,1);
    for n = 1:N
        node{n} = num2str(n);
    end
    Graph.Nodes.Name = node;
    GraphNew = Graph;
 %------------------------------------------------- Algorithm
    while sum(isinf(C)) > 0 % While there are non colored nodes
        Ccurrent = Ccurrent + 1;
        GraphNew = Graph;
        % Remove the colored nodes. It creates a subgraph
        GraphNew = rmnode(GraphNew, find(~isinf(C)));
        while numnodes(GraphNew) > 0 % If there still are non colored nodes
           % Find the node of the subgraph  of uncolored nodes
           % that has the smallest degree
          [~, nodeIndex] = min(degree(GraphNew, GraphNew.Nodes.Name));
            % Assign the current color
            C(str2double(cell2mat(GraphNew.Nodes.Name(nodeIndex))))...
              = Ccurrent;
            % Find the neighbours of these nodes
            neigh = neighbors(GraphNew,...
              (cell2mat(GraphNew.Nodes.Name(nodeIndex))));

            % Remove this node
            GraphNew = rmnode(GraphNew, GraphNew.Nodes.Name(nodeIndex));
            % Remove its neighbours
            GraphNew = rmnode(GraphNew, neigh);
        end
```

```
    end
end
%=========================================================
function edges=G2edges(G)
 [N,~]=size(G); % Number of nodes (connected symmetric graph)
 nE=N*(N-1)/2;
 edges=zeros(nE,2);
 nbEdges=0;
 for i=1:N-1
   for j=i+1:N
     if G(i,j)>0
       nbEdges=nbEdges+1;
      edges(nbEdges,1)=i;
      edges(nbEdges,2)=j;
     end
   end
 end
 edges=edges(1:nbEdges,:);
end
```

8.7.5.9 RLF

```
%{
 See  MRLF
https://hal.archives-ouvertes.fr/hal-00451266
%}
function C=colorAlgoRLF(G)
fprintf('\n Algo RLF');
[N,~]=size(G);
degr=sum(G,2);
[~,Ind]=sort(degr,'descend');
C=zeros(1,N);  % Initial colors = no color at all
colored=0; % Number of colored nodes
k=1;
% Assign color k
while colored<N
  add=false;
  for n=1:N
    m=Ind(n);
    if C(m)>0 continue; end % Already colored

    % Uncolored node, check the neighbours
    possible=true;
    for nn=1:N
      if nn==m continue; end
      if  G(m,nn)==1 && C(nn)==k % A neighbour is colored with k
```

```
          possible=false;
          break;
        end
      end % for nn=1:N

      if possible % No neighbour is colored with k
        C(m)=k;
        colored=colored+1;
        add=true;
      end
    end % for n=1:N

    if ~add
      k=k+1;
    end
  end
  end
```

8.7.5.10 Retours (*Backtracking*)

```
function C=colorAlgoBack(G)
%{

    Recursive coding.

%}
fprintf('\n Backtracking')
[N,~]=size(G);
K=2;
OK=false;
iterK=0; % Just for information
while ~OK
  iterK=iterK+1; fprintf('\nK %i,  %i',K,iterK)
C=zeros(1,N);
C(1)=1;
j=1;
C=Color(C, j,K,N,G);
Ind=find(C==0);
if isempty(Ind)
  OK=true;
else
  K=K+1;
  iterK=0;
end
end
end
%=============================================================
```

```
function [C]=Color(C, j,K,N,G)
if j==N+1
  return;
end
for i=1:K
  C(j)=i;
  if valid(C,G,j)
   [C]= Color(C,j+1,K,N,G);
   break;
  end
end
end
```

8.7.5.11 Recherche séquentielle

```
function C=colorAlgoSeqImprov2 (edges,K0)
%{
 Garanteed version , but time consuming
K0 = initial number of colors to try.

    Then the algorithm may increase or decrease it.

%}
saveSkip=false;
if saveSkip skipFile=fopen('skip','w'); end
saveColor=false;
if saveColor saveC=fopen('saveC','w'); end
verbose=false;
N=max(edges(:));
[nbEdges,~]=size(edges);
valid=false;
if  K0==0
  u=nbEdges/(N*(N-1)/2);
  K=floor(max(2,N*u^(N/2)));% Try first this number of colors
else
  K=K0;
end
decreaseK=true;
admissible=1;
%skip=[]; % For information. Number of inadmissible after each admissible
nskip=0;
END=false;
C=[zeros(1,N-K+1), 1:K-1]; % Cmin. Admissible
Cmax=[0,(K-1)*ones(1,N-1)]; %
while 1==1 % Seems infinite, but one always finds a solution and return
  if K==N  % For the complete graph
    Cbest=1:N;
```

```
  return
end

% Check validity
valid=checkValid2(edges,nbEdges,C);

if valid
  if  decreaseK
    if K>2
      K=K-1;
      C=[zeros(1,N-K+1), 1:K-1]; % Cmin. Admissible
      Cmax=[0,(K-1)*ones(1,N-1)]; %
    else % K=2, and valid => END
      END=true;
    end
  else % Increasing K and valid => END
    END=true;
  end
else % If not valid
  if sum(C==Cmax)==N % All coloring with K have been checked
    K=K+1; % I increase K
    decreaseK=false;
    C=[zeros(1,N-K+1), 1:K-1]; % New Cmin. Admissible
    Cmax=[0,(K-1)*ones(1,N-1)]; %

  else % Try the next coloring with K
     [C,skipped]=nextSkip2(C,N,K);
    if skipped % Either K=2 or skip possible
      admiss=true;
    else % Check admissibility
       admiss=checkAdmiss(C,K);
      if admiss
        admissible=admissible+1;
      end
    end

    if admiss
      % Save C
      if saveColor
        for n=1:N
          fprintf(saveC,'%i ',C(n))
        end
        fprintf(saveC,'\n')
      end
      %skip=[skip; nskip]; nskip=0; % For information
    else
```

```
          nskip=nskip+1; % Interval between successive admissibles
        end
      end
    end

    if END
      if saveSkip
        for s=1:length(skip)
          fprintf(skipFile,'%i \n',skip(s));
        end
        fclose(skipFile);
      end
     C=C+1;
      return %***** END
    end

  end
  end
  %=========================================================
  function admiss=checkAdmiss(C,K)
  admiss=true;
  for k1=1:K
    isk=C==k1-1;
    if isempty(isk) % One colour in (0,..., K-1) is missing
      admiss=false;
      return
    end
  end
  %=========================================================
  function valid=checkValid2(edges,nbEdges,C)
  valid=false;
  for i=1:nbEdges
     if C(edges(i,1))==C(edges(i,2))
      return
    end
  end
  valid=true;
  end
  %=========================================================
  function [C,skipped]=nextSkip2(C0,N,K)
  %  Note there is a repetition of nextC
  C=zeros(1,N);
  skipped=false;
  if K==2
   % C=nextC(C0,N,K);
    n=N;
```

```
    stop=false;
    C=C0;
    while ~stop
      C(n)=C(n)+1;
      if C(n)==K
        C(n)=0;
        n=n-1;
        if n==0
          stop=true;
        end
      else
        stop=true;
      end
    end
    %----------
    skipped=true; % Actually skip=0, but the coloring is admissible for sure
  else
    C1=[zeros(1,N-K+1), K-1:-1:1];
    if sum(C0==C1)<N
     % C=nextC(C0,N,K);
     n=N;
    stop=false;
    C=C0;
    while ~stop
      C(n)=C(n)+1;
      if C(n)==K
        C(n)=0;
        n=n-1;
        if n==0
          stop=true;
        end
      else
        stop=true;
      end
    end

     %-----------------
      skipped=false;
    else % We can directly skip to C2
      C=[ zeros(1,N-K), 1,0,2:K-1];
      skipped=true;
    end
  end
  end
```

8.7.6 Bi-objectif

```
function C=colorAlgoBiobj(G0,edges,nbEdges)

    global G % To avoid nested functions
    G=G0;
    [N,~]=size(G); % Number of nodes
    lb=ones(1,N);
    ub=N*lb;
    options = optimoptions('gamultiobj','ParetoFraction',0.7,'PlotFcn',@gaplotpareto);
    [solution,f12] = gamultiobj(@biobj,N,[],[],[],[],lb,ub,options);
    solution=round(solution); % List of all solutions
    % Find a valid solution, if any
    [nbSol,~]=size(solution);
    Cvalid=[];
    nbValSol=0;
    for s=1:nbSol

        Cs=colorEquiv2(solution(s,:),N); % Transform the coloring
                                         % into a minimal equivalent one
        valid=checkValid2(edges,nbEdges,Cs); % True iif each arc is valid
        if valid
            Cvalid=[Cvalid;Cs];
            nbValSol=nbValSol+1;
            f1(nbValSol)=f12(s,1); % Number of colors of this coloring
        end
    end
    if isempty(Cvalid)

        fprintf("\n Warning: invalid colouring!\n")
        C=Cs(1,:);
        return
    end
    % Select the valid solution with the minimum number of colors
    [~,Ind]=sort(f1);
    C=Cvalid(Ind(1),:);
    end

%---------------------
function f=biobj(x,G)

    global G
    C=round(x);
    N=numel(C);
    % Number of colors, to minimise
    f(1)=numel(unique(C));
```

```
    % Number of invalid arcs, to minimise
    nbInval=0;
    for i=1:N-1
        for j=i+1:N
            if G(i,j)>0 && C(i)==C(j)
              nbInval=nbInval+1;
            end
        end
    end
    f(2)=nbInval;
end
```

8.7.7 Algorithmes quantiques

Les codes sont en langage de simulation Qiskit (**(author?)** [2]).

8.7.7.1 2-coloriage

```
#----------------------------- General code for 2-colouring
import numpy as np
from qiskit import *
%matplotlib inline
# Define the graph by giving the ends of each edge
end1=[0,1,2,3,1,3]
end2=[1,2,3,0,4,4]
#----
nNodes=max(end1+end2)+1
nEdges=len(end1)
nodes=np.arange(nNodes)
# Create a Quantum Circuit
q = QuantumRegister(nNodes+nEdges)
c=ClassicalRegister(nNodes)
circ = QuantumCircuit(q,c)
# Gates
circ.h(nodes)
k=nNodes
for n in range(nEdges):
    i=end1[n]
    j=end2[n]
    circ.ccx(i,j,k)
    circ.x(i);circ.x(j); circ.ccx(i,j,k); circ.x(i);circ.x(j)
    circ.cx(k,j)
    circ.barrier(i,j,k)
    k=k+1
```

```
#----------------------- Measure
circ.barrier(q)
circ.measure(nodes,c)
#----------------------- Run
from qiskit import Aer
# Use Aer's qasm_simulator
backend_sim = Aer.get_backend('qasm_simulator')
job_sim = execute(circ, backend_sim, shots=1024)
# Grab the results from the job.
result_sim = job_sim.result()
counts = result_sim.get_counts(circ)
print(counts)
#--------------- Drawing
circ.draw()
```

8.7.7.2 3-coloriage, méthode Q

```
#-------------------------- Q method, Generate all 3-colorings
'''
N= number of nodes
K= number of colors
Each color is coded by at most Q=ceil(ln(K)/ln(2)) qubits
The main difficulty is to "eliminate", if any, Q-K states
Example, K=3
needs 2 qubits, but it implies 4 states 00, 10, 01, 11
so we use a circuit that eliminates 11
'''
import numpy as np
from qiskit import *
%matplotlib inline
from qiskit.quantum_info.operators import Operator
from qiskit import Aer
backend_sim = Aer.get_backend('qasm_simulator')
nNodes=3
nColors=3 # nColors <= nNodes. WARNING, only 3 with this code
nqcode=2 # WARNING, only 2, for nColors=3
nn2=round((nNodes-1)*nNodes/2) # Number of pairs of different nodes
                               # and of possible edges
sc=round(nqcode*nNodes) # Beginning of the list of qubits
                        # that describes the pairs of nodes
sg=round(sc + nn2) # Beginning of the list of qubits
                   # that describes the graph
nqbits=sg + nn2+2 # Total number of qubits
# Create a Quantum Circuit
q = QuantumRegister(nqbits)
```

```
c=ClassicalRegister(nqcode*nNodes) # To measure to "extract" the coloring
qc = QuantumCircuit(q,c)
# Add the graph (binary list of nNodes*(nNodes-1)/2 elements,
# because the graph is symmetric and there is no i=>i edges.
# Set to |1> the qubits corresponding to an edge.
'''
Graphs
3 nodes, needs 3 colors
0 1 1
1 0 1
1 1 0
=> 1 1 1
4 nodes, needs 3 colors
0 0 1 1
0 0 1 1
1 1 0 1
1 1 1 0
=> 0 1 1 1 1 1
'''
qc.x(sg) # Set to |1> iif there is an edge
qc.x(sg+1)
qc.x(sg+2)
#qc.x(sg+3)
#qc.x(sg+4)
#qc.x(sg+5)
# Initialisation
# Hadamard gate for qubits that represent the coloring
s=0
for n in range(nNodes):
    for k in range(nqcode):
        qc.h(s)
        s=s+1
# Eliminate |11>
remov_op = Operator([[3, 1, -1, 1],
[-1, 3, 1, 1],
[1, -1, 3, 1],
[-1, -1, -1, 3]]/(2*np.sqrt(3)))
# We can also use a sub-circuit
for n in range(nNodes):
    q1=n*nqcode
    q2=q1+1
    qc.unitary(remov_op, [q1, q2], label='remov 11')
# At this point, if we measure, we find 3 possible colors for each node:
# 00, 10, 01
```

```
print('end of coloring')
# Switch the ancillary qubits corresponding to pairs of nodes
# that have the same color
q4=sc # Ancillary qubit to use
for n1 in range(nNodes-1):

    q0=2*n1
    q1=q0+1
    for n2 in range(n1+1,nNodes):

        q2=2*n2
        q3=q2+1
        # Case |00>
        qc.x(q0);qc.x(q1);qc.x(q2);qc.x(q3)
        qc.mcx([q0,q1,q2,q3],q4)
        qc.x(q0);qc.x(q1);qc.x(q2);qc.x(q3)
        qc.barrier()
        # Case |10>
        qc.x(q1);qc.x(q3)
        qc.mcx([q0,q1,q2,q3],q4)
        qc.x(q1);qc.x(q3)
        qc.barrier()
        # Case |01>
        qc.x(q0);qc.x(q2)
        qc.mcx([q0,q1,q2,q3],q4)
        qc.x(q0);qc.x(q2)
        qc.barrier()
        q4=q4+1 # Will use the next ancillary qubit

# At this point, if we measure the (nNodes-1)*nNodes)/2 ancillary qubits,
# we get binary strings,
# in which 1 means "same color" for n1 and n2
print('end of pairs of nodes')
# Compare to the graph.
qnc=nqbits-2
for n in range(sc,sc+nn2): # For each pair of nodes


    # If same color and there is an edge "destroy" (set to |0*>) the coloring
    qc.ccx (n,n+nn2,qnc) # Set to |1> if same color and edge
    for code in range(sc):


        qc.ccx(code,qnc,qnc+1) # Set code to |0> if qnc is |1> ...
        qc.cx(qnc+1,code) # ...
        qc.reset(qnc+1)
        qc.reset(qnc)

print('end of compare to graph')
```

```
# Measure (only the qubits describing the coloring)
cb=0
for code in range(sc):

    qc.measure(code,cb)
    cb=cb+1

print('end of measures')
# Execute the circuit
result=job.result()
#print(result)
print('end of execute')
# Grab the results from the job.
counts = result.get_counts(qc)
print(counts)
print('The solutions are given by the strings that are not 0*')
print('Please check: some of them may need LESS than', nColors,'colours')
#qc.draw()
#qc.draw(filename='circuit_triangle_3_colors, Q method')
```

8.7.7.3 Méthode NK

```
Code Qiskit de la méthode NK
   '''
    NK method for the K-coloring of a N-graph
    This version generates many times the completely uncolored "solution"
    So it has to be removed then by classical post-processing
   ---
   Binary coloring matrix:
   1 row for each node
   1 column for each color
   One and only one 1 in each row, other values 0
   It means we need at least NK qubits to describe such a matrix
   ---
   On my laptop I can consider only very small graphs
   but in principle it should work for any N and K.
   Used here under Anaconda / Jupyter /Python
   Badly coded, I know.
   Warnings:
   The algorithm searches a solution for a given number of colors (K).
   For a complete resolution it should loop on different K.
   The worst approach is to check for K=2, then 3, ... N-1
   but clever strategies do exist (dichotomy, for example)
   '''
   import math
   import random
   from random import randrange
```

```
import numpy as np
from qiskit import *
import qiskit.tools.jupyter
from qiskit.visualization import plot_histogram
# %qiskit_version_table
# %qiskit_copyright
%matplotlib inline
#from qiskit import Aer
#from qiskit.providers.aer import AerError, QasmSimulator
#backend_sim = Aer.get_backend('statevector_simulator')
#backend_sim = Aer.get_backend('extended_stabilizer')
backend_sim = Aer.get_backend('qasm_simulator')
# Describe the graph
# as a binary sequence  (upper right half-triangle of the incidence matrix)
# read row by row from left to right
#G=[1] # 2 nodes, needs 2 colors
#G=[1,1,1 ] # 3 nodes, needs 3 colors
#G=[1,0,1] # 3 nodes, needs 2 colors
G=[1,0,0,1,0,1] # 4 nodes, needs 2 colors
#G=[0, 1, 1, 1, 1, 1] #  4 nodes, needs 3 colors
#G=[1, 0, 0, 1, 1, 1] #  4 nodes, needs 3 colors
#G=[1,1,1,1,1,1] # 4 nodes, needs 4 colors
nColors=2 # nColors <= nNodes. Number of colors to try.
nshots=100 # Number of runs
nNodes=round(math.sqrt(2*len(G)+1/4)+1) .
print([nNodes,' nodes'])
#------------------------------------------------- Build the circuit
# Check for a given number of colors
nc=nColors+1 # For each node, nColors qubits that will be measured, + one ancill
nn2=round((nNodes-1)*nNodes/2) # Number of pairs of different nodes and of possi
sc=round(nc*nNodes) # Number of qubits to skip before the ancillary qubits for p
sg=round(nc*nNodes + nn2) # Beginning of the list of qubits that describe the gr
nqbits=sc + 2*nn2
# Total number of qubits
# Create a Quantum Circuit
q = QuantumRegister(nqbits)
c=ClassicalRegister(nColors*nNodes) # To measure to "extract" the coloring matri
qc = QuantumCircuit(q,c)
# Add the graph (binary list of nNodes*(nNodes-1)/2 elements, because the graph
# and there is no i=>i edges. Set to |1> the qubits corresponding to an edge.
for n in range(nn2):
    if G[n]>0:
        qc.x(sg+n)
# ---------------------------------------Generate a coloring matrix
# Initialisation
# Hadamard gate for qubits that represent the coloring matrix
```

```
s=0
for n in range(nNodes):
    for k in range(nColors):
        qc.h(s+k)
    s=s+nc
#print(qc)
#-----------------------------
# Constraints
# A 1 and only one 1 in each of the coloring matrix
s=0
for n in range(nNodes):
    for k in range(nColors-1):
        for l in range(k+1,nColors):
           # Eliminate 11
            cb=[s+k,s+l]  # |1> => qubit has the color k
                          # |1> => qubit has also the color l
            qc.mcx (cb,s+nColors) # If so, set to |1> the ancillary bit ...
            qc.cx (s+nColors,s+k) # ... and use it to remove the color k
            qc.reset(s+nColors)

    # Eliminate 0* row (no color assigned to the node n)
    for k in range(nColors): qc.x(s+k) #  if 0 => 1. Not needed if you can use a

    cb=list(range(s,s+nColors) ) # If now all |1>,
    qc.mcx (cb,s+nColors)        #...  set to |1> the ancillary bit ...

    for k in range(nColors): qc.x(s+k) # if 1 => 0  (restore)

    #
    #qc.cx (s+nColors,s+nColors-1) # If no color at all, assign |1> to the last
    qc.cx (s+nColors,s) # If no color at all, assign |1> to the first one
    #tosetto1=randrange(s,s+nColors-1) # Assign a random color
    #qc.cx (s+nColors,tosetto1)
    qc.reset(s+nColors) # Reset and reuse the ancillary qubit

    s=s+nc

 # At this point, if we measure, we find coloring matrices (nNodes lines, nColor
# one and only one 1 in each line (a node does have a color, and only one)
print('end of coloring matrices')
print(qc)
#qc.draw(output='mpl',filename="./NK, triangle, superposition.eps")
# Switch the ancillary qubits corresponding to pairs of nodes that have the same
for k in range(nColors):
    s=nc*nNodes
    for n1 in range(nNodes-1):
```

```
        for n2 in range(n1+1,nNodes):
            n11=nc*n1+k # If q[n11]=|1> it means the node n1 has the color k
            n22=nc*n2+k # If q[n22]=|1> it means the node n2 has the color k

            qc.ccx(n11,n22,s) # If same color k, set s to |1>. Notice it can hap
            s=s+1
            #print([n11,n22])
 # At this point, if we measure the (nNodes-1)*nNodes)/2 ancillary qubits, we ge
# in which 1 means "same color" for n1 and n2
print('end of pairs of nodes')
# Compare to the graph.
for n in range(sc,sc+nn2): # For each pair of nodes
    # If same color and there is an edge "destroy" (set to |0*>) the coloring
    for node in range(nNodes):
        qnode=nc*node
        qnc=qnode+nColors
        for k in range(nColors):
            cb=[n,n+nn2,qnode+k]
            qc.mcx (cb,qnc)
            cb=[n,n+nn2,qnc]
            qc.mcx (cb,qnode+k)
            qc.reset(qnc)
print('end of compare to graph')
# Measure (only the qubits describing the coloring matrices)
cb=0
for n in range(nNodes):
    s=n*(nColors+1)
    for k in range(nColors):
        qb=s+k
        qc.measure(qb,cb)
        cb=cb+1
print('end of measures')
# print(qc) # Display the circuit
#qc.draw(output='mpl',filename="./NK_triangle.eps") # Save the circuit
d=qc.depth()
print("Circuit depth: ",d)
print("Circuit width: ",nqbits)
print("Complexity: ",d*nqbits)
# Execute the circuit on a statevector simulator
#job = execute(qc, backend_state,shots=1000)
# Execute the circuit on the qasm simulator.
# Quick on small graphs, but memory error for 4 nodes, 3 colors
job = execute(qc, backend_sim,shots=nshots)
#  This method should handle more qubits, but is awfully slow
#qobj = assemble(qc, backend=QasmSimulator(), shots=1000)
#job = QasmSimulator().run(qobj, backend_options={'method': 'extended_stabilizer
```

```
result=job.result()
#print(result)
print('end of execute')
# Grab the results from the job.
counts = result.get_counts(qc)
print(counts)
print('The solutions, if any,  are given by the strings that are not 0*')
if nColors>2:
    print('Please check: some of them may be a coloring with LESS than', nColors
plot_histogram(counts)
# Coloring binary matrices and color lists (by node)
str=list(counts.keys()) #, list(counts.values()))
nClass=len(str)
nbSol=0
for n in range(nClass):
    binary=str[n] # Bit string, a solution (if not 0*)
    reverse_str = binary[::-1] # Should be read from right to left

    if int(reverse_str)>0:
        print(reverse_str)
        nbSol=nbSol+1
        print("Binary coloring solution")
        Color=[]
        for i in range(nNodes):

            start=i*nColors
            end=start+nColors
            row = reverse_str[start : end]
            print(row)
            for j in range(nColors):
                if int(row[j])>0:
                    Color.append(j+1)

        #print(" ")
        print("Color list (by node)")
        print(Color)
        print(" ")

if nbSol==0: print("Sorry, I didn't find any solution")
```

8.7.7.4 Méthode NK, variante 6

```
'''
 NK-6 (low cost) method for the K-coloring of a N-graph
```

```
 Here "low cost" just means less qubits than the NK method
 but in fact it needs more gates,
 and the depth*width complexity is higher.
 This version generates many times the completely uncolored "solution"
 So it has to be removed then by classical post-processing
---
Binary coloring matrix:
1 row for each node
1 column for each color
One and only one 1 in each row, other values 0
It means we need at least NK qubits to describe such a matrix
---
On my laptop I can consider only very small graphs
but in principle it should work for any N and K.
Used here under Anaconda / Jupyter /Python
Badly coded, I know.
Warnings:
The algorithm searches a solution for a given number of colors (K).
For a complete resolution it should loop on different K.
The worst approach is to check for K=2, then 3, ... N-1
but clever strategies do exist (dichotomy, for example)
'''
import math
import numpy as np
from qiskit import *
import qiskit.tools.jupyter
from qiskit.visualization import plot_histogram
%matplotlib inline
backend_sim = Aer.get_backend('qasm_simulator')
# Describe the graph
# as a binary sequence  (upper right half-triangle of the incidence matrix)
# read row by row from left to right
#G=[1] # 2 nodes, needs 2 colors
G=[1,1,1 ] # 3 nodes, needs 3 colors
# G=[1,0,1] #3 nodes, needs 2 colors
#G=[0, 1, 1, 1, 1, 1] #  4 nodes, needs 3 colors
#G=[1, 0, 0, 1, 1, 1] #  4 nodes, needs 3 colors
nColors=3 # nColors <= nNodes. Number of colors to try.
# Number of runs
nshots=100
#------------------------------------------------- Build the circuit
nNodes=round(math.sqrt(len(G)+1/4)+1) # It should be len(G)+1/4)+1/2, but Python ...
print([nNodes,' nodes'])
nnc=nNodes*nColors
nn2=round((nNodes-1)*nNodes/2) # Number of pairs of nodes
ancill=round(nnc+nn2)
```

```
nqbits=ancill +2 # Total number of qubits
# Create a Quantum Circuit
q = QuantumRegister(nqbits)
c=ClassicalRegister(nnc) # To measure to "extract" the coloring matrices
qc = QuantumCircuit(q,c)
# Add the graph (binary list of nNodes*(nNodes-1)/2 elements, because the graph is symm
# and there is no i=>i edges. Set to |1> the qubits corresponding to an edge.
for pair in range(nn2): # For each pair of nodes ...
    if G[pair]>0:
        qc.x(nnc+pair) # ... set to |1> if the edge does exist
#print(qc)
# ---------------------------------------Generate a superposition of all coloring matr
# Initialisation
# Hadamard gate for qubits that represent the coloring matrix
qc.h(range(nnc))
#print(qc)
#-----------------------------
# Constraints
# A 1 and only one 1 in each row of the coloring matrix
s=0
for n in range(nNodes):
    for k in range(nColors-1):
        for l in range(k+1,nColors):
           # Eliminate 11
            qc.ccx (s+k,s+l,ancill)
            qc.cx (ancill,s+k)
            qc.reset(ancill)

    # Eliminate 0* (no color assigned to the node n)
    for k in range(nColors):
        qc.x(s+k) #  if 0 => 1. Not needed if you can use a negative multicontrolled ga

    cb=list(range(s,s+nColors) )
    qc.mcx (cb,ancill)

    for k in range(nColors): qc.x(s+k) # if 1 => 0

    qc.cx (ancill,s+nColors-1)
    qc.reset(ancill)
    s=s+nColors

 # At this point, if we measure, we find coloring matrices (nNodes rows, nColors columm
# one and only one 1 in each row (a node does have a color, and only one)
print('end of coloring matrices')
#qc.draw(output='mpl',filename="./NK6, mycircuit.eps")
# Check conflicts (same color at the ends of an edge)
```

```
n=-1
for n1 in range(nNodes-1):
    qnode1=n1*nColors # rank of the first qubit for the node n1
    for n2 in range(n1+1,nNodes):
        qnode2=n2*nColors # rank of the first qubit for the node n1
        n=n+1
        edge=nnc+n # rank of the qubit for the pair (n1, n2).  |1> if there is an edge

        for k in range(nColors): # For each color k ...
            nk1=qnode1+k # |1> if node n1 has the color k
            nk2=qnode2+k # |1> if node n2 has the color k

            cb=[nk1,nk2,edge] # control qubits
            qc.mcx (cb,ancill) # |1> if same color (i.e. conflict) and there is an edge

            # If ancill=|1> remove all colors
            for nn in range(nnc): # loop on colorings of node
                cb=[ancill,nn] # ancill=|1> if same color and there is an edge
                               # nn=|1> if there is a color
                qc.mcx(cb,ancill+1) # set to |1> a second ancillary qubit
                qc.cx(ancill+1,nn) # and use it to set to |0> the color qubit
                qc.reset(ancill+1)

            qc.reset(ancill)  # Reset and reuse the ancillary qubit (dynamic circuit)

print('end of check conflicts')
# Measure (only the qubits describing the coloring matrices)
cb=0
for n in range(nNodes):
    s=n*nColors
    for k in range(nColors):
        qb=s+k
        qc.measure(qb,cb)
        cb=cb+1
print('end of measures')
# print(qc) # Display the circuit
#qc.draw(output='mpl',filename="./NK6.eps") # Save the circuit
d=qc.depth()
print("Circuit depth: ",d)
print("Circuit width: ",nqbits)
print("Complexity: ",d*nqbits)
# Execute the circuit on a statevector simulator
#job = execute(qc, backend_state,shots=nshots)
# Execute the circuit on the qasm simulator.
# Quick on small graphs, but memory error for 4 nodes, 3 colors
# nshots
```

```
job = execute(qc, backend_sim,shots=nshots)
#  This method should be able to handle more qubits, but is awfully slow
#qobj = assemble(qc, backend=QasmSimulator(), shots=nshots)
#job = QasmSimulator().run(qobj, backend_options={'method': 'extended_stabilizer'})
result=job.result()
#print(result)
print('end of execute')
# Grab the results from the job.
counts = result.get_counts(qc)
print(counts)
print('The solutions, if any,  are given by the strings that are not 0*')
if nColors>2:
    print('Please check: some of them may be a coloring with LESS than', nColors,' colo
plot_histogram(counts)
# Coloring binary matrices and color lists (by node)
str=list(counts.keys()) #, list(counts.values()))
nClass=len(str)
nbSol=0
for n in range(nClass):
    binary=str[n] # Bit string, a solution (if not 0*)
    reverse_str = binary[::-1] # Should be read from right to left

    if int(reverse_str)>0:
        print(reverse_str)
        nbSol=nbSol+1
        print("Binary coloring solution")
        Color=[]
        for i in range(nNodes):
            start=i*nColors
            end=start+nColors
            row = reverse_str[start : end]
            print(row)
            for j in range(nColors):
                if int(row[j])>0:
                    Color.append(j+1)

        #print(" ")
        print("Color list (by node)")
        print(Color)
        print(" ")

if nbSol==0: print("Sorry, I didn't find any solution")
```

8.7.8 Diplomate 0

Un code Octave/Matlab volontairement non optimisé, de façon à bien mettre en évidence les étapes de la méthode.

```
function C=colorAlgoDiplomat0(G,frust,nbColors)
% frust= NxN values (one for each arc)
%       Not the most compact way, but easier to use
fprintf('\n ==================================Diplomat 0');
[N,~]=size(G);
frustTot=0;
frustMax=sum(frust(:));
edges=G2edges(G); % Just for plot
C=[1,zeros(1,N-1)]; % Initial colors
coloredNb=1; % Number of colored nodes
while coloredNb<N
  frstMin=Inf;
  for i=1:N % For each uncolored node
    if C(i)>0 continue; end
    % List of colored neighbours
    coloredNeigh=[];

    for j=1:N
      if G(i,j)==0 continue; end % No link
      if C(j)==0 continue; end % A link, but the node is uncoloured
      coloredNeigh=[coloredNeigh,j];
    end

    % Try colors
    for k=1:nbColors
      % If assigned to the node, compute the generated frustration
      frst=0;
      for neigh=1:length(coloredNeigh)
        j=coloredNeigh(neigh);
        if C(j)==k
          frst=frst+frust(i,j)+frust(j,i);
        end
      end

      % Keep the best pair (node, color)
      if frst<frstMin
        frstMin=frst;
        iBest=i;
        kBest=k;
      end
    end
  end %  for n=1:N
```

```
    fprintf('\n Node %i colored by %i => frustration %f',iBest,kBest,frstMin)
    frustTot=frustTot+frstMin;

    C(iBest)=kBest;
    coloredNb=coloredNb+1;
  end %  while coloredNb<N
  fprintf("\n Total frustration %f/%f = %f",frustTot, frustMax,...
    frustTot/frustMax);
  end
```

8.8 La Course quantique

Ce petit jeu de plateau est vaguement inspiré des circuits quantiques, d'où son nom. On peut imaginer que chaque jeton représente un qubit et la couleur son état : blanc pour $|0>$ et noir pour $|1>$. Et ce jetons traversent des portes qui les transforment.

8.8.1 Matériel

Les photos 8.8.1 montrent une version « de luxe » en bois, dans laquelle les barrettes à neuf trous servent à compter le nombre de parties gagnées, mais tout le jeu peut être réalisé en carton et les scores simplement notés sur papier.

- Une aire de jeu, sur laquelle sont dessinées les pistes ou lignes de temps (horizontales, une par joueur) et les lignes d'instant (verticales, en nombre impair, par exemple de 0 à 6 pour des parties simples).
- Pour chaque joueur, un jeton noir et un jeton blanc ou, mieux, un biface noir-blanc. On peut aussi utiliser plus de jetons, pour les laisser en place et mieux visualiser la partie terminée (voir les exemples ci-dessous).
- Un jeu de plaquettes ou de cartes pour les « portes quantiques », chacune en plusieurs exemplaires :
  1. La porte X. Elle bascule le jeton qui la franchit (blanc donne noir et inversement).
  2. La porte H. Après franchissement, la couleur du jeton est tirée au hasard.
  3. La porte cX. (c pour « contrôle »). À l'instant en cours, si le jeton du joueur est noir, elle lui donne le droit de remplacer la porte d'un adversaire au même « temps » par une porte X, piochée à part.
  4. La porte I. Le jeton qui la franchit reste identique.
- Un dispositif quelconque permettant de faire un choix binaire au hasard (pile ou face, dé noir et blanc, etc.).

8.8.2 Règles et déroulement du jeu

On suppose ici qu'il y a 7 lignes d'instant, de 0 à 6.

- Au début chaque joueur pose son jeton blanc sur sa piste, à l'instant 0. Le but du jeu est d'arriver avec un jeton noir à l'instant 6.
- On distribue au hasard 6 portes, face cachée, à chaque joueur.
- Un paquet spécial de 6 portes X est placé à part.
- Chaque tour, où les joueurs jouent chacun un coup, correspond à une progression d'un instant.
- Chaque joueur choisit une porte dans son jeu et la pose face cachée devant lui. Quand tous les joueurs l'on fait, ces portes peuvent être retournées, chaque joueur place la sienne sur sa piste, sur la ligne d'instant en cours.
- Si un joueur a présenté une porte cX et que son jeton est actuellement noir (sous-entendu « actif ») il remplace la porte de l'adversaire de son choix par une X prise dans le paquet spécial, sauf si cette porte est également une cX.
- Ensuite, chaque joueur fait franchir sa porte à son jeton, en appliquant la transformation correspondante :
  - Pour une porte I, il garde la couleur.
  - Pour une porte cX, il garde la couleur.
  - Pour une porte X, il change la couleur.
  - Pour une porte H, il tire la couleur au hasard.

8.8.3 Exemple 1, avec deux joueurs

Sam reçoit les cartes suivantes : X I X cX H H . Il joue sur la première piste.

Sara reçoit les cartes I cX H H X X . Elle joue sur la seconde piste.

Le tableau 8.1 présente une partie complète, où l'on a laissé en place les jetons successifs, pour mieux voir l'évolution.

- Instant 1. Chaque joueur place une porte X. En les traversant, les jetons deviennent noirs.
- Instant 2. Sam place une porte I. Son jeton va rester noir. Sara place une porte H, tire au hasard la couleur et tombe sur blanc.
- Instant 3. Sam place une porte H, tire au hasard et tombe sur blanc. Sara place une porte I. Son jeton reste blanc.
- Instant 4. Sam place une porte H, tire au hasard et tombe sur blanc. Sara place une porte H, tire au hasard et tombe sur noir.

Table 8.1: Course quantique, exemple 1

	0	1		2		3		4		5		6	
Sam	○	X	●	I	●	H	○	H	○	cX	○	X	●
Sara	○	X	●	H	○	I	○	H	●	X	○	cX	○

- Instant 5. Sam place une cX. Comme son jeton est blanc, elle sera en fait inopérante sur Sara. Son propre jeton ne change pas et reste blanc. Sara place une X. Au passage, son jeton devient blanc.
- Instant 6. Sam place sa dernière porte, une X. Son jeton devient noir. Sara place sa dernière porte, une cX qui ne sert à rien. Son jeton reste blanc. Sam a gagné.

8.8.4 Exemple 2, avec portes cX activées

Le tableau 8.2 présente une autre partie, Sam et Sara ayant les mêmes cartes que dans l'exemple précédent. Ici, c'est Sara qui gagne.

- Instant 1. Sam et Sara placent tous deux une porte X. Leurs jetons deviendront noirs en la traversant.
- Instant 2. Sam a placé une I, mais Sara une cX. Comme son jeton est noir, elle pioche une X dans le paquet spécial et remplace la porte I de Sam, dont le jeton va devenir blanc en traversant cette porte X.
- Instant 3. Sam place une porte H, tire au hasard et tombe sur noir. Sara place une porte H, tire au hasard et tombe sur blanc.
- Instant 4. Sam place une porte H, tire au hasard et tombe sur noir. Sara place une porte H, tire au hasard et tombe sur noir.
- Instant 5. Sara a placé une I, Sam une cX. Comme son jeton est noir, il doit piocher une X pour remplacer la I de Sara. Le jeton de Sara deviendra blanc, mais en fait ça va l'arranger.
- Instant 6. Sam doit poser sa dernière porte, qui est une X. Son jeton passe la porte et devient blanc. Sara pose également sa dernière porte, une X. Son jeton passe la porte et devient noir. Sara a gagné.

Table 8.2: Course quantique, exemple 2

	0	1		2		3		4		5		6	
Sam	○	X	●	I, X	●○	H	●	H	●	cX	●	X	○
Sara	○	X	●	cX	●	H	○	H	●	I,X	●○	X	●

Figure 8.8.1: La Course quantique, version « de luxe »

Bibliographie

[1] DIMACS Graphs and Best Algorithms. http://cedric.cnam.fr/ porumbed/graphs/.

[2] Qiskit, https://qiskit.org/.

[3] Saman Ahmadi, Guido Tack, Daniel D. Harabor, and Philip Kilby. Bi-Objective Search with Bi-directional A*. *Proceedings of the International Symposium on Combinatorial Search*, 12(1):142–144, July 2021. Number: 1.

[4] Shamim Ahmed. Applications of Graph Coloring in Modern Computer Science.

[5] Alain Aspect, Jean Dalibard, and Gérard Roger. Experimental Test of Bell's Inequalities Using Time-Varying Analyzers. *Physical Review Letters*, 49(25):1804–1807, December 1982. Publisher: American Physical Society.

[6] Yonatan Bilu. Three extensions of Hoffman's bound on the graph chromatic number. *Journal of Combinatorial Theory*, (Series B 96), 2006.

[7] Daniel Brélaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22(4):251–256, April 1979.

[8] Stephen S. Bullock and Igor L. Markov. Arbitrary two-qubit computation in 23 elementary gates. *Physical Review A*, 68(1):012318, July 2003. Publisher: American Physical Society.

[9] Maurice Clerc. When Nearer is Better. Technical report, Open archive https://hal.archives-ouvertes.fr/hal-00137320, 2007.

[10] Maurice Clerc. *L'aléatoire contrôlé en optimisation*. ISTE (International Scientific and Technical Encyclopedia), April 2015.

[11] Matthew DeCross, Eli Chertkov, Megan Kohagen, and Michael Foss-Feig. Qubit-reuse compilation with mid-circuit measurement and reset, October 2022. arXiv:2210.08039 [quant-ph].

[12] C. S. Edwards and C. H. Elphick. Lower bounds for the clique and the chromatic numbers of a graph. *Discrete Applied Mathematics*, 5(1):51–64, January 1983.

[13] Alex Fabrikant and Tad Hogg. Graph Coloring with Quantum Heuristics. Edmonton, Alberta, Canada, 2002.

[14] Alexandre Gondran and Laurent Moalic. Optimality Clue for Graph Coloring Problem, December 2018.

[15] Stefano Gualandi and Federico Malucelli. Exact Solution of Graph Coloring Problems via Constraint Programming and Column Generation. *INFORMS Journal on Computing*, 24(1):81–100, February 2012. Publisher: INFORMS.

[16] Adrien Guignard. *Jeux de coloration de graphes.* Thèse de doctorat, Bordeaux 1, December 2011.

[17] Fei Hua, Yuwei Jin, Yanhao Chen, John Lapeyre, Ali Javadi-Abhari, and Eddy Z. Zhang. Exploiting Qubit Reuse through Mid-circuit Measurement and Reset, November 2022. arXiv:2211.01925 [quant-ph].

[18] David S. Johnson. Worst case behaviour of graph coloring algorithms. pages 513–527, Winnipeg, Manitoba, 1974. Utilitas Math.

[19] F. T. Leighton. *A graph coloring algorithm for large scheduling problems.* National Bureau of Standards, 1979.

[20] R. M. R. Lewis. Algorithm Case Studies. In R. M. R. Lewis, editor, *Guide to Graph Colouring: Algorithms and Applications*, Texts in Computer Science, pages 113–154. Springer International Publishing, Cham, 2021.

[21] R.M.R. Lewis. *A Guide to Graph Colouring.* Springer International Publishing, Cham, 2016.

[22] Dorian Mazauric. *Graphes et Algorithmes – Jeux grandeur nature.* Inria, 2016. Pages: 1.

[23] Paul-Antoine Moreau, Ermes Toninelli, Thomas Gregory, Reuben S. Aspden, Peter A. Morris, and Miles J. Padgett. Imaging Bell-type nonlocal behavior. *Science Advances*, 5(7):eaaw2563, July 2019. Publisher: American Association for the Advancement of Science.

[24] PSC. Particle Swarm Central, http://particleswarm.info.

[25] Dominik Rauch, Johannes Handsteiner, Armin Hochrainer, Jason Gallicchio, Andrew S. Friedman, Calvin Leung, Bo Liu, Lukas Bulla, Sebastian Ecker, Fabian Steinlechner, Rupert Ursin, Beili Hu, David Leon, Chris Benn, Adriano Ghedina, Massimo Cecconi, Alan H. Guth, David I. Kaiser, Thomas Scheidl, and Anton Zeilinger. Cosmic Bell Test Using

Random Measurement Settings from High-Redshift Quasars. *Physical Review Letters*, 121(8):080403, August 2018. Publisher: American Physical Society.

[26] Kazuya Shimizu and Ryuhei Mori. Exponential-Time Quantum Algorithms for Graph Coloring Problems. *Algorithmica*, June 2022.

[27] Satish Thadani, Seema Bagora, and Anand Sharma. Applications of graph coloring in various fields. *Materials Today: Proceedings*, 66:3498–3501, January 2022.

[28] Carlos Hernández Ulloa, William Yeoh, Jorge A. Baier, Han Zhang, Luis Suazo, and Sven Koenig. A Simple and Fast Bi-Objective Search Algorithm. *Proceedings of the International Conference on Automated Planning and Scheduling*, 30:143–151, June 2020.

[29] Avi Wigderson. A new approximate graph coloring algorithm. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, STOC '82, pages 325–329, New York, NY, USA, May 1982. Association for Computing Machinery.

[30] Zhaoyang Zhou, Chu-Min Li, Chong Huang, and Ruchu Xu. An exact algorithm with learning for the graph coloring problem. *Computers & Operations Research*, 51:282–301, November 2014.

Index