Méthodes exactes et approchées par partition en cliques de graphes

Raksmey Phan

DU : 2396 – EDSPIC : 623

UNIVERSITÉ BLAISE PASCAL

ÉCOLE DOCTORALE SPI

SCIENCES POUR L'INGÉNIEUR

T H È S E

Présentée par

Raksmey PHAN

pour obtenir le titre de

Docteur d'Université

Spécialité : INFORMATIQUE

Méthodes exactes et approchées par partition en cliques de graphes

---

préparée au Laboratoire d'Informatique, de Modélisation et d'Optimisation des Systèmes, Projet ANR ToDo

Soutenue publiquement le 28 novembre 2013,
devant le jury composé de :

Rapporteurs :	
Pascal BERTHOMÉ	Professeur des Universités, ENSI de Bourges
Johanne COHEN	Chargée de Recherche, Université Paris Sud
Examinateurs :	
François DELBOT	Maître de Conférences, Université de Nanterre
Alain QUILLIOT	Professeur des Universités, Université Blaise Pascal
Directeur :	
Christian LAFOREST	Professeur des Universités, Université Blaise Pascal

Remerciements

C'est avec une joie teintée de tristesse que j'écris ces remerciements qui concluent trois années de thèse.

En tout premier lieu et parce que c'est le plus important, je tiens à remercier l'ami, l'encadrant, le directeur de thèse, le professeur Christian Laforest. Un ancien étudiant me disait au début que pour Christian, un thésard c'est avant tout un collaborateur. C'est seulement maintenant que je comprends à quel point cette remarque est importante. Mon directeur fut un guide au début, un mentor durant la thèse et un collègue expérimenté à la fin. Grâce à lui, je n'ai pas seulement appris à apprendre et à découvrir, j'ai aussi appris la persévérance : *il n'y a jamais d'échec à moins que l'on abandonne*. Mon ami fut aussi un solide mât rassurant, auquel je pouvais m'accrocher lors de violentes tempêtes...

Je tiens à remercier les rapporteurs et membres du jury. Je remercie le professeur Pascal Berthomé, rapporteur du mémoire, qui a suivi ce travail depuis ses débuts en participant au comité de thèse. Je remercie madame Johanne Cohen, rapporteur du mémoire, pour sa patience et l'intérêt qu'elle a montré pour notre sujet. Je remercie également le professeur Alain Quilliot, président du jury et directeur du laboratoire. Grâce à lui, le Limos est un laboratoire dans lequel j'ai pris plaisir à travailler quels que soient l'heure et le jour. Et je remercie monsieur François Delbot pour son précieux soutien lors de la conclusion de ces travaux.

Je remercie tout particulièrement mes amis et collègues de bureau Benjamin, Christophe, Diyé, Djelloul, Heitor, Hélène, Jonathan, Libo, Marie, Nikolay, Philippe et Pierre-Alban grâce à qui travailler et faire de la recherche est un bonheur. Plus généralement, je souhaite remercier tout le personnel du Limos et de l'ISIMA qui ont su rendre, par leur professionnalisme et leur amitié, le lieu de travail très agréable. Tous les matins je suis heureux de venir travailler et les longues vacances me rendent impatient de revenir.

Je tiens également à remercier tous mes amis de l'association WorldTop, et plus particulièrement son président honoraire monsieur Bernard Pierre Fenal auprès de qui j'ai tant appris ces dernières années. L'association accueille depuis neuf ans déjà les étudiants internationaux : la fougue de leur jeunesse et leur joie de vivre rendent au vieux thésard que je ne suis plus, le sourire même aux moments les plus difficiles. Je remercie ainsi le professeur David Hill, très impliqué dans le milieu associatif et d'une charité exemplaire. Un remerciement tout particulier à la Maraude de Clermont-Ferrand, pour ce que ses bénévoles font pour les plus démunis d'entre-nous dans (hélas) l'indifférence générale. Puis enfin, une pensée pour la toute jeune association des doctorants Doct'Auvergne, à qui je souhaite plein de réussite.

Pour terminer et non par la moindre, je remercie ma chère et tendre mère qui, par ses défauts et ses qualités, m'a permis de m'épanouir et de devenir l'homme que je suis...

Résumé : Cette thèse se déroule au sein du projet ToDo [1] financé par l'Agence Nationale de la Recherche. Nous nous intéressons à la résolution exacte et approchée de deux problèmes de graphes. Dans un soucis de compromis entre la durée d'exécution et la qualité des solutions, nous proposons une nouvelle approche par partition en cliques qui a pour but (1) de résoudre de manière rapide des problèmes exacts et (2) de garantir la qualité des résultats trouvés par des algorithmes d'approximation. Nous avons combiné notre approche avec des techniques de filtrage et une heuristique de liste. Afin de compléter ces travaux théoriques, nous avons implémenté et comparé nos algorithmes avec ceux existant dans la littérature.

Dans un premier temps, nous avons traité le problème de l'indépendant dominant de taille minimum. Nous résolvons de manière exacte ce problème et démontrons qu'il existe des graphes particuliers dans lesquels le problème est 2-approximable. Dans un second temps nous résolvons par un algorithme exact et un algorithme d'approximation le problème du vertex cover et du vertex cover connexe. Puis à la fin de cette thèse, nous avons étendu nos travaux aux problèmes proches, dans des graphes comprenant des conflits entre les sommets.

Mots clés : exact, approximation, heuristique gloutonne, indépendant dominant, vertex cover, conflits

1. Time versus Optimality in Discrete Optimization ANR 09-EMER-010.

Exact and Approximation Methods by Clique Partition of Graphs

Abstract : This thesis takes place in the project ToDo [2] funded by the french National Research Agency. We deal with the resolution of two graph problems, by exact and approximation methods. For the sake of compromise between runtime and quality of the solutions, we propose a new approach by partitionning the vertices of the graph into cliques, which aims (1) to solve problems quickly with exact algorithms and (2) to ensure the quality of results with approximation algorithms. We combine our approach with filtering techniques and heuristic list. To complete this theoretical work, we implement our algorithms and compared with those existing in the literature.
At the first step, we discuss the problem of independent dominating of minimum size. We solve this problem accurately and prove that there are special graphs where the problem is 2-approximable. In the second step, we solve by an exact algorithm and an approximation algorithm, the vertex cover problem and the connected vertex cover problem. Then at the end of this thesis, we extend our work to the problems in graphs including conflicts between vertices.

Keywords : Exact, Approximation, Greedy Heuristic, Independent Dominating, Vertex Cover, Conflicts

2. Time vs. Optimality in Discrete Optimization ANR 09-EMER-010.

Table des matières

Table des figures

Liste des tableaux

Introduction

Chapitre 1

Algorithmes, complexité et quelques problèmes de graphe

Sommaire

Dans le premier chapitre de cette thèse, nous souhaitons rappeler de manière informelle les notions de base nécessaires pour comprendre la complexité algorithmique, sa mesure et sa comparaison. Nous rappelons ainsi les principes généraux de l'évaluation de la complexité utiles dans la suite de ce document. À la fin de ce chapitre et avant de commencer la présentation de nos travaux, nous introduirons les définitions et notations utilisées dans la suite du document. Une liste des abréviations est aussi disponible à la fin de ce document.

1.1 Complexité algorithmique

Un *algorithme* est un ensemble d'étapes qui permettent de donner une réponse à un problème. Le mot algorithme vient du nom du mathématicien perse Al-Khawarizmi,

connu comme le père de l'algèbre. Sa définition a été donnée bien avant le début de l'informatique par David Hilbert en 1928, lorsqu'il cherchait une suite d'instructions, finies et non-ambigües, pour résoudre son "Entscheidungsproblem" (problème de décision). La non-ambigüité des instructions permet ainsi de distinguer les algorithmes mathématiques, des algorithmes non mathématiques tel que les recettes de cuisine. La *complexité algorithmique*, plus communément appelée *complexité*, a pour but de donner une indication sur la durée d'exécution ou l'espace mémoire occupé par un algorithme.

La notion de complexité que nous présentons ci-dessous, est très liée au modèle de *machine* de calcul d'Alan Turing [Turing 1936]. Dans ce modèle une action est une suite d'*opérations élémentaires*. A chacune des étapes de cette suite, selon l'état de la mémoire de la machine, une opération (élémentaire) est choisie dans l'ensemble des opérations possibles. On dit que cette machine est *déterministe* lorsque, à chaque étape, le choix de l'opération dépend, et est *déterminé*, par l'état courant de la mémoire.

1.1.1 Algorithme et mesure de complexité

La théorie de la complexité fournit des outils pour l'étude et l'évaluation des performances d'un algorithme, du point de vue des ressources (en temps et en espace) essentielles à son exécution. A l'aube de l'informatique (après la seconde guerre mondiale) l'évaluation intrinsèque de la performance était relativement "primitive" : les scientifiques comparaient des mesures qui dépendaient du matériel (le hardware), du compilateur et du langage (le software) utilisés. Ces mesures étaient prises lors des exécutions des algorithmes et on mesurait le nombre d'*opérations élémentaires*. La définition d'une opération élémentaire dépend du détail que l'on souhaite avoir sur la complexité d'un algorithme. Nous considérons ici, une opération élémentaire comme une opération *simple* d'addition, de soustration, d'affectation de valeurs,... Le fait, par exemple, simplement de demander à la machine de faire l'addition de deux nombres a et b peut être décomposé en plusieurs opérations : (1) lire a ; (2) lire b ; (3) additionner a et b ; (4) afficher le résultat. L'évaluation de la complexité essaye de quantifier ces opérations car plus il y a d'opérations élémentaires pour une action, plus cette action va prendre du temps à être exécutée. Or, une même action peut se décomposer en plusieurs suites différentes d'opérations élémentaires. Pour multiplier a et b, une ancienne machine additionnerait $b-1$ fois la valeur de a alors que sur les nouvelles machines, l'opération de multiplication est intégrée, donc la multiplication est réalisée en une seule opération. De la même manière, pour résoudre un même problème, il peut exister plusieurs algorithmes qui peuvent avoir des complexités différentes. Cette section est l'occasion de clarifier tout cela.

C'est en 1968 que Donald Knuth propose dans The Art of Computer Programming [Knuth 1968], une méthode d'évaluation indépendante de l'environnement de développement et mesurant non plus les opérations élémentaires mais la quantité de *res-*

sources utilisées par un algorithme pour son fonctionnement. On entend par ressource, le temps nécessaire à l'algorithme pour se terminer ; ou encore, l'espace mémoire utilisé lors de son exécution. L'addition de a et b peut-être considérée comme une seule étape.

Il propose aussi de mesurer la ressource (en temps ou en mémoire) consommée avec le scénario le plus défavorable : c'est ce qu'on appelle la mesure *en pire cas*. Par exemple, lorsqu'on cherche au hasard (de manière équiprobable) un nombre m parmi une liste de n valeurs différentes, la situation la plus défavorable est celle où m se trouve à la fin de la liste. L'algorithme consistant à parcourir et tester chaque élément de la liste pour trouver m, testera alors les n éléments. La *complexité en pire cas* (généralement appelé *complexité*) est donc n [1], bien qu'avec une autre liste, l'algorithme pourrait très bien rencontrer m en moins de n tests. Ainsi pour comparer les algorithmiques, on peut comparer le nombre *maximum* d'instructions qu'ils nécessitent pour se réaliser. Il est facile de comprendre que ce genre de mesure ne permet pas réellement de comparer les performances des algorithmes dans la pratique. Par la suite il est donc proposé de mesurer la complexité en *moyenne* : évaluer le nombre moyen d'instructions d'un algorithme. Reprenons l'exemple de la recherche du nombre m dans une liste de n nombres (m est parmi les éléments de la liste). De manière assez triviale, on peut calculer qu'en *moyenne* l'algorithme s'arrêtera au bout de $\frac{n}{2}$ tests (modulo quelques hypothèses que nous ne précisons pas ici).

1.1.2 Évaluation et comparaison de complexité

Considérons un problème quelconque P, appelons *instance* de P l'ensemble des données d'entrée et supposons que l'on ait un algorithme qui le résout. Nous souhaitons étudier sa complexité par rapport à la *taille* de l'instance de P. Dans le cas général, cette *taille* est notée n, et représente l'espace mémoire occupée par l'instance. La complexité est alors définie comme le nombre d'instructions exécutées pour résoudre une instance de P en fonction de n.

Prenons par exemple la somme $S_n = \sum_{i=1}^{n} i$ qu'il faut calculer ($S_1 = 1$; $S_2 = 1+2 = 3$; $S_n = 1 + \cdots + n$). L'approche la plus basique consiste à additionner successivement les i. Il faut alors faire $n - 1$ additions. Cela signifie que si on double la valeur de n, l'algorithme va prendre (à une unité près) deux fois plus de temps à s'exécuter. Or il existe une formule appelée l'identité de la somme des premiers entiers qui permet de calculer la somme : $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$. Avec cette identité, on peut écrire un nouvel algorithme nécessitant beaucoup moins de calcul (une addition, une multiplication et une division) quelque soit la valeur de n. Nous constatons ainsi que pour faire le même travail, la comparaison des deux algorithmes permet de choisir la plus rapide.

Notons cependant que cette manière d'évaluer la complexité, communément utilisée,

1. Pour simplifier l'introduction, nous n'entrerons pas dans le détails du codage de n ; ce qui compliquerait un peu les calculs de la complexité.

est discutable car elle ne prend pas en compte le type de données traitées lors de chaque instruction.

Étant donnée la difficulté pratique de ces méthodes de mesures et de comparaisons, on préfère souvent exprimer la complexité en terme de *grandeur de dominance* : $O(.)$. Ainsi un algorithme qui fait $n-1$ opérations (où n est la taille de l'instance) est en $O(n)$ (le terme *dominant* est n, *c'est-à-dire* que la complexité augmente linéairement à n) alors que le calcul de la série S_n avec l'identité de la somme des premiers entiers ($\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$) est en $O(1)$ (le terme dominant est une *constante*, *c'est-à-dire* que quelle que soit la valeur de n la complexité est constante).

On distingue ainsi la complexité des algorithmes par la grandeur qui la domine. Nous donnons ci-dessous une liste non exhaustive :

- $O(1)$: complexité constante, indépendante de n ;
- $O(log(n))$: complexité logarithmique ;
- $O(n^p)$: complexité polynomiale $(p \in \mathbb{R}, p > 1)$;
- $O(a^n)$: complexité exponentielle $(a \in \mathbb{R}, a > 1)$.

1.2 Classement des problèmes et des algorithmes

La section précédente a permis de présenter la notion de complexité algorithmique. Comme nous l'avons vu, pour résoudre un même problème, il peut exister plusieurs algorithmes de complexités très différentes. Dans cette section, nous allons ranger les problèmes dans des classes de complexités. La classe de complexité d'un problème est déterminée par l'algorithme de plus faible complexité (le plus rapide) qui permet de le résoudre. Ainsi ces classes permettent, d'une certaine manière, de classer les problèmes selon la quantité de ressources nécessaires pour les résoudre. Pour cela revenons tout d'abord sur la définition des problèmes de décision et d'optimisation.

1.2.1 Problèmes de décision, d'optimisation et certificat

Un *problème de décision* est un problème informatique dont la réponse est soit *oui* soit *non*. La formulation classique d'un tel problème est : "Existe-t-il une solution telle que <conditions> ?". Les <conditions> sont un ensemble de critères que doit respecter la solution. En d'autres termes, le problème pose la question de l'existence d'un algorithme pour trouver une réponse. Supposons que l'on cherche du chocolat au lait dans un supermarché, alors la question de savoir si ce supermarché en vend ou non est trivialement un problème de décision (pour le savoir il suffirait de passer en revue tous les articles de leur catalogue). Un *problème d'optimisation* est un problème dans lequel il faut déterminer si une solution est la *meilleure c'est-à-dire* si elle est *optimale*. Toujours à la recherche du chocolat au lait, le problème d'optimisation serait de trouver le chocolat au lait le moins cher du supermarché.

Les problèmes de décision et d'optimisation induisent un autre besoin, celui de la *certification*. Il faut pouvoir dire si une réponse est ou non une solution *pertinente* au problème. Un certificat est une information (de taille polynomiale à la taille de l'instance) supplémentaire ajoutée au problème, qui permet de valider (ou non) une solution : c'est en soi, un problème à part entière. Pour dire s'il y a ou non du chocolat au lait dans le supermarché, encore faut-il être capable de reconnaître le chocolat au lait. Pour dire lequel est le moins cher, il faut être capable de comparer les prix.

1.2.2 Classes $\mathcal{P}$, $\mathcal{NP}$, $\mathcal{NP}$-*complet* et $\mathcal{NP}$-*difficile*

La classe $\mathcal{P}$ regroupe les problèmes dont la *résolution* peut se faire en temps polynomial. La classe $\mathcal{NP}$ regroupe les problèmes dont la vérification d'une solution accompagnée du *certificat* se déroule en temps polynomial.

Ainsi en terme de définition, il n'y a pas vraiment de relation entre $\mathcal{P}$ et $\mathcal{NP}$. Cependant il est trivial de constater que les problèmes qui peuvent être résolu en temps polynomial (donc dans $\mathcal{P}$), ont une méthode de validation (certificat) au plus en temps polynomial (donc dans $\mathcal{NP}$). Cela montre la relation $\mathcal{P} \subseteq \mathcal{NP}$. La preuve de l'égalité (ou non) de cette relation est un sujet fondamental de l'informatique. Beaucoup de travaux (y comprit les nôtres) supposent l'inégalité des deux ensembles. La démonstration de leur égalité serait un bouleversement, car cela induirait le fait que nous pouvons résoudre de manière exacte et en complexité polynomiale des problèmes pour lesquels nous ne connaissons actuellement, pour les résoudre, que des algorithmes de complexité exponentielle.

La classe $\mathcal{NP}$-*complet* est un sous ensemble de la classe $\mathcal{NP}$, regroupant les problèmes qui *dominent* tous les autres problèmes de la classe $\mathcal{NP}$. Dans la pratique, cela veut dire que si on trouve un algorithme qui résout un problème de la classe $\mathcal{NP}$-*complet* alors on pourra l'adapter pour résoudre tous les autres problèmes de la classe $\mathcal{NP}$. D'où l'intérêt de traiter les problèmes dits $\mathcal{NP}$-*complet*.

Pour montrer qu'un problème P est $\mathcal{NP}$-*complet* il faut montrer deux choses : (1) une solution peut être *certifiée* en temps polynomial (le problème est alors dans $\mathcal{NP}$) et (2) tous les problèmes de la classe $\mathcal{NP}$ peuvent se réduire de manière polynomiale à P (on dit alors que le problème P est au moins aussi difficile que tous les autres problèmes de la classe $\mathcal{NP}$). Les problèmes qui remplissent la dernière propriété sont de la classe $\mathcal{NP}$-*difficile*. Les figures 1.1 et 1.2 résument ses imbrications de classes dans les cas où $\mathcal{P} \neq \mathcal{NP}$ et $\mathcal{P} = \mathcal{NP}$.

Dans la pratique, il est toujours intéressant de démontrer qu'un problème est $\mathcal{NP}$-*complet*, ou du moins $\mathcal{NP}$-*difficile*. Montrer qu'un problème est dans $\mathcal{NP}$ *c'est-à-dire* que le certificat s'obtient en temps polynomial est souvent évident ; mais réduire tous les problèmes $\mathcal{NP}$ au problème traité est beaucoup plus fastidieux. Habituellement on part d'un problème connu pour être $\mathcal{NP}$-*complet*, que l'on déduit au problème traité.

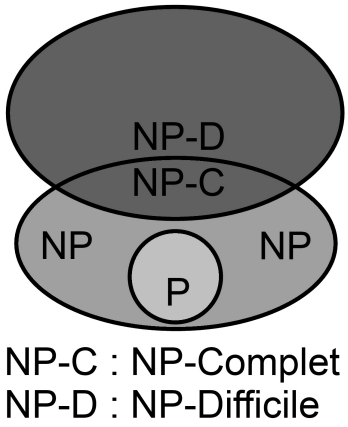

FIGURE 1.1 – Les classes $\mathcal{P}$, $\mathcal{NP}$, $\mathcal{NP}$-*complet* et $\mathcal{NP}$-*difficile* lorsque $\mathcal{P} \neq \mathcal{NP}$

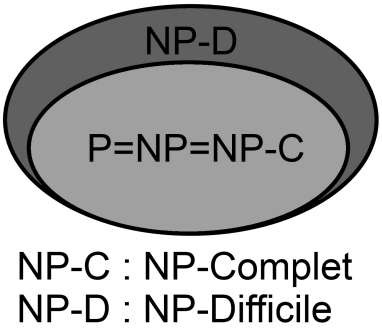

FIGURE 1.2 – Les classes $\mathcal{P}$, $\mathcal{NP}$, $\mathcal{NP}$-*complet* et $\mathcal{NP}$-*difficile* lorsque $\mathcal{P} = \mathcal{NP}$

Comme nous l'avons souligné, montrer que $\mathcal{P} = \mathcal{NP}$ reviendrait à pouvoir résoudre de manière polynomiale tous les problèmes difficiles que nous connaissons actuellement dans le domaine de la recherche opérationnelle. La section suivante donne un exemple d'un problème $\mathcal{NP}$-*complet*.

1.2.3 Exemple de problème $\mathcal{NP}$-*complet* : *Sat*

Nous présentons ici l'un des premiers problèmes $\mathcal{NP}$-*complet* présenté dans l'œuvre fondateur de Garey et Johnson [Garey 1979]. Le propos ici n'est pas de présenter en détail le problème *Sat*, mais de donner une définition formelle de l'un des premiers problèmes $\mathcal{NP}$-*complet* qui est abondamment utilisé dans la recherche opérationnelle pour démontrer la $\mathcal{NP}$-complétude d'autres problèmes.

Soit un ensemble fini de *variables booléennes* $\mathcal{X} = \{x_1, \ldots, x_n\}$ et une *formule propositionnelle* $\mathcal{F} = C_1 \wedge ... \wedge C_l$ composée de *clauses* $C_i = y_{i,1} \vee \cdots \vee y_{i,c_i}$ (avec c_i le nombre de variables booléennes de la clause C_i) telles que $\forall (i,j)$, $y_{i,j}$ est un *littéral* de la variable x_k *c'est-à-dire* : $y_{i,j} = x_k$ ou $y_{i,j} = \neg x_k$. Une affectation $\tau : \mathcal{X} \to \{0,1\}$ ($\{Vrai, Faux\}$) satisfait $\mathcal{F}$ *si et seulement si* pour chaque clause C_i, au moins un littéral est égal à 1. Une solution *Sat* est une affectation $\mathcal{S}_\mathcal{F}$ qui satisfait $\mathcal{F}$. Prenons par exemple un ensemble de 4 variables booléennes $\mathcal{X} = \{x_1, x_2, , x_3, x_4\}$ et une formule $\mathcal{F} = (x_1 \vee x_2) \wedge (\neg x_2 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee x_3)$ composée donc des clauses $C_1 = (x_1 \vee x_2)$, $C_2 = (\neg x_2 \vee \neg x_3 \vee x_4)$ et $C_3 = (\neg x_1 \vee x_3)$. Une affectation $\mathcal{S}_\mathcal{F}$ qui satisfait $\mathcal{F}$ pourrait être $(x_i = 1)_{i=1,3,4}$ (et quelle que soit la valeur de x_2). En effet comme $x_1 = 1$, la clause C_1 est vérifiée ; comme $x_3 = 1$, la clause C_3 est vérifiée ; comme $x_4 = 1$, la clause

C_2 est vérifiée. Donc $\mathcal{F}$ est satisfait. Le théorème de Cook [Cook 1971] montre que le problème de décision du *Sat* (*c'est-à-dire* savoir s'il existe une affectation satisfaisant $\mathcal{F}$) est $\mathcal{NP}$*-complet*.

Ce problème est ainsi l'un des problèmes de référence pour la démonstration de la $\mathcal{NP}$-complétude. S'il existe un algorithme polynomial pour résoudre le problème *Sat*, cet algorithme pourrait résoudre en temps polynomial *tous* les problèmes de $\mathcal{NP}$ et on aurait alors démontré que $\mathcal{P} = \mathcal{NP}$. Pour résoudre ces problèmes *difficiles* il y a différents types d'algorithmes classés selon la qualité des solutions qu'ils retournent et du temps de recherche nécessaire.

1.3 Algorithmes

Nous allons présenter dans ce qui suit le classement des algorithmes en fonction de la qualité des solutions fournies pour les problèmes d'optimisation. Comme nous l'avons vu, la complexité temporelle varie beaucoup d'un algorithme à l'autre. Certains algorithmes résolvent les problèmes en un nombre constant d'étapes quelle que soit la taille de l'instance ; d'autres résolvent les problèmes en un nombre d'étapes proportionnel à la taille des instances ; alors que d'autres encore, le font en un nombre d'étapes exponentielles par rapport à la taille de l'instance. Ci-dessous nous présentons les trois catégories d'algorithmes qui permettent de résoudre les problèmes d'optimisation de la classe $\mathcal{NP}\backslash\mathcal{P}$, *c'est-à-dire* des problèmes dont nous ne connaissons pas d'algorithmes en temps polynomial pour les résoudre de manière exacte.

Nous présentons dans cette section, les *algorithmes exacts* qui permettent de trouver des solutions optimales en temps exponentiel. Puis nous introduirons les *algorithmes d'approximation* qui permettent de trouver des solutions dont la qualité est garantie, en temps polynomial. Enfin, nous présenterons les *heuristiques* qui, bien que la qualité de leurs solutions n'est pas garantie, trouvent souvent en temps acceptable des solutions de bonnes qualités.

1.3.1 Algorithmes exacts

Les algorithmes *exacts* sont des méthodes qui permettent de répondre de manière *exacte* à un problème. Pour les problèmes de décision l'idée la plus *simple* pour résoudre exactement un problème dans $\mathcal{NP}\backslash\mathcal{P}$, serait d'énumérer tous les sous ensembles susceptibles d'être des solutions et de les tester. Dans le cas d'un problème *Sat* de n variables booléennes, un tel algorithme aurait une complexité de 2^n.

Il existe (heureusement) d'autres méthodes que cette dernière, par énumération exhaustive. Par exemple celle de Hwang *et al.* [Hwang 1993] en 1993 qui permet de résoudre en $O^*(n^{O^*(\sqrt{n})})$ le problème du TSP [2] avec des distances euclidiennes dont

2. Le TSP (ou *Travelling Salesman Problem*) est un problème classique d'optimisation dans lequel

la complexité de l'algorithme par énumération exhaustive est de $O^*(n!)$. Une autre approche pour résoudre de manière exacte les problèmes $\mathcal{NP}$-*complet* est la programmation dynamique, notion utilisée pour la première fois par Richard Bellman vers la fin des années 40 et publiée dans *Dynamic Programming* en 1957 (nouvelle édition [Bellman 2003]). Cette approche se fonde sur le principe que pour résoudre optimalement certains problèmes, on peut résoudre optimalement leurs sous-parties. Pour la résolution exacte, cette stratégie est donc applicable seulement à un certain nombre de problèmes dont notamment le sac à dos et les tours de Hanoï.

La résolution exacte par énumération s'est renforcée au fil du temps avec de nouvelles techniques qui permettent de mieux organiser le parcours et les tests de ces sous ensembles. Le but est alors de mieux organiser et de mieux guider la recherche en (1) se concentrant seulement sur les solutions réalisables, en (2) déterminant une fonction d'évaluation efficace de chacune de ces solutions et en (3) déterminant des critères qui permettent de tester seulement les solutions les plus "prometteuses". Cette approche conduit naturellement à définir un parcours de recherche dans lequel chaque étape correspond à un test des critères définis dans l'étape (3) pour guider vers les solutions optimales. Cette stratégie permet souvent d'éviter de tester certains groupes de solutions de mauvaise qualité. Cette méthode a beaucoup inspiré la technique de *séparation et évaluation* (ou branch and bound) proposée pour la première fois par Land et Doig [Land 1960] en 1960, elle est reprise et adaptée à la plupart des problèmes d'optimisation exacts ou non [Troya 1989, Hahn 1998, Allahverdi 2006].

Avec ces algorithmes, nous pouvons évaluer la durée d'exécution des programmes en pire cas, en fonction de la taille des instances. Or, il peut arriver que pour certaines familles de graphes ou de problèmes, on souhaite un algorithme optimisé spécifiquement au groupe. Si on souhaite, par exemple, résoudre un problème sur des graphes de degré maximal borné par une constante Δ, il serait intéressant d'avoir un algorithme dont la complexité dépend aussi de Δ. L'idée de la *complexité paramétrique* est donc de changer le paramètre de mesure de la complexité. Dans la pratique, la complexité exponentielle en fonction de n peut être très grand, mais on espère que la complexité par rapport à un autre paramètre est plus petite, voire polynomiale. Cette idée a été mentionnée par Downey et Fellows [Downey 1992] en 1992. Par exemple dans les problèmes de vertex cover, sur certaines instances, dont on sait que la taille des solutions optimales (notons k) évolue peu en fonction de la taille n du graphe, il est intéressant d'avoir par exemple un algorithme en $O^*(1.2738^k + kn)$ [Chen 2010]. Ainsi si k est constant, l'algorithme serait polynomial, malgré n. De nombreuses études ont permis de développer des algorithmes avec une complexité paramétrique. Citons par exemple le problème de partition de cliques dont la complexité est de 2^{k^2} où k est le nombre de cliques [Mujuni 2008].

Ces dernières années ont également vu la puissance des machines continuer à aug-

il faut trouver un cycle de longueur minimal passant exactement une fois par chacun des n points. La distance entre deux points est la distance euclidienne qui les sépare.

menter et les algorithmes exacts sont capables de résoudre des problèmes de taille de plus en plus importante. Dans le même temps, nous avons vu le retour de la recherche vers les algorithmes exacts tentant d'améliorer leur complexité [Gaspers 2008, Gaspers 2012, Bourgeois 2009a, Bourgeois 2009b, Bourgeois 2010, Havet 2011]. L'analyse approfondie des propriétés des graphes permettent de déterminer des critères précis qui améliorent la complexité en pire cas des algorithmes.

Cependant les algorithmes exponentiels atteignent très vite leurs limites, à cause de la durée d'exécution qui augmente très rapidement par rapport à la taille du graphe. Les puissances des machines ont augmenté, mais en parallèle la taille des instances de problèmes a aussi augmenté. Avec la mondialisation, certains problèmes locaux, se posent maintenant à l'échelle nationale voire internationale. De plus, le développement d'algorithmes de plus en plus complexes et/ou de plus en plus spécifiques à certaines propriétés de certains graphes rend la partie opérationnelle (l'implémentation du code) relativement difficile. Il n'est donc pas étonnant de trouver très peu de résultats expérimentaux dans la littérature récente.

1.3.2 Algorithmes d'approximation

Les algorithmes d'*approximation* permettent de garantir la qualité des solutions trouvées. Un algorithme d'approximation est une approche de résolution en temps polynomial, dont on arrive à borner la taille des solutions trouvées. Le plus souvent associé aux problèmes $\mathcal{NP}$*-difficiles*, ces algorithmes permettent de les résoudre en temps polynomial tout en préservant une *certaine* qualité des solutions trouvées.

Généralement on dit qu'un algorithme est une c-approximation s'il retourne des solutions de taille au plus c fois celle de la solution optimale pour un problème de minimisation et de taille au moins $\frac{1}{c}$ fois la taille de la solution optimale pour un problème de maximisation. La variable c est appelée *facteur d'approximation*. Ainsi une 1-approximation est un algorithme exact en temps polynomial. A partir de cette définition, le but dans ce domaine, est donc de trouver un algorithme d'approximation avec une constante c le plus proche de 1, pour un problème donné.

Pour montrer qu'un algorithme est une c-approximation il faut (1) que l'algorithme soit polynomial, (2) que l'algorithme renvoie bien des solutions réalisables au problème posé et (3) que le rapport de la solution sur l'optimale soit bien borné supérieurement par c pour un problème de minimisation et borné inférieurement par $\frac{1}{c}$ pour un problème de maximisation. Un problème qui admet un algorithme d'approximation de facteur constant c est de la classe $\mathcal{APX}$. Cette classe regroupe de nombreux problèmes $\mathcal{NP}$ [Bar-Noy 1998, Chaudhary 2001, Yannakakis 1980]. Un problème qui admet un algorithme d'approximation de facteur $1 \pm \varepsilon$, avec $\varepsilon \in \mathbb{R}^*$ un réel est de la classe $\mathcal{PTAS}$. Sur ce sujet, la thèse de Kann [Kann 1992] est très complète. Parallèlement aux problèmes de décision et des classes $\mathcal{P}$ et $\mathcal{NP}$, il définit de nouvelles classes pour les

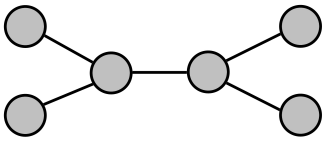

FIGURE 1.3 – Graphe papillon

problèmes d'optimisation. Nous avons repris ci-dessus une partie seulement des notions utiles pour clarifier notre travail.

Prenons par exemple le graphe de la figure 1.3. Supposons qu'on recherche un sous ensemble de sommets qui *couvre* toutes les arêtes, *c'est-à-dire* un groupe de sommets tel que chaque arête du graphe soit liée à au moins l'un d'entre eux (c'est le problème du vertex cover que nous définirons formellement dans la partie II de ce document). Une solution possible consiste à prendre tous les sommets du graphe ; on sera alors sûr de *couvrir* toutes les arêtes. Supposons maintenant que l'on cherche une solution à ce problème ayant le moins de sommets : c'est le problème d'optimisation. Sur ce petit graphe, il est facile de voir que les deux sommets centraux forment la solution de taille minimum (ici de taille 2) et couvrant toutes les arêtes. Appliquons maintenant l'algorithme glouton EDGES DELETION dont le détail sera donné à la section 5.4 : à chaque étape l'algorithme sélectionne une arête et ajoute les deux sommets de ses extrémités dans la solution puis surprime cette arête ainsi que toutes les arêtes *liées* aux deux sommets. Sur les graphes de la figure 1.4, nous avons représenté avec des arêtes en gras toutes les solutions retournées par EDGES DELETION pour le graphe papillon. On constate que ces solutions contiennent au plus 4 sommets, soit deux fois plus que la solution optimale. Pour ce problème et dans tous les graphes en général, on démontre que l'algorithme EDGES DELETION a un rapport d'approximation 2, *c'est-à-dire* que sans prétendre connaître la solution optimale (ni sa taille) on prouve que l'algorithme ne retourne que des solutions de taille au plus deux fois celle de l'optimale. Le problème est de la classe $\mathcal{APX}$. Au delà de cet algorithme glouton, il existe bien entendu d'autres types de méthodes de résolution. Comme par exemple la résolution par programmation linéaire qui s'applique très bien au problème du vertex cover [Vazirani 2001]. La résolution du problème relaxé permet aussi d'obtenir une 2-approximation (le détail de cette modélisation est donné à la section 5.4).

Comme nous pouvons le remarquer cette approche pour mesurer le rapport/facteur d'approximation c est une mesure *en pire cas*, *c'est-à-dire* qu'on prend la situation la moins avantageuse pour l'algorithme. Imaginons une situation très caricaturale dans laquelle, pour un problème donné, un algorithme trouve toujours des solutions optimales sauf pour un cas. La taille de ce dernier peut être très différente de la taille des solutions optimales. Ce sera alors ce dernier cas qui donne le rapport d'approximation de l'algorithme. Or il est très probable que l'algorithme trouve plutôt l'une des autres solutions

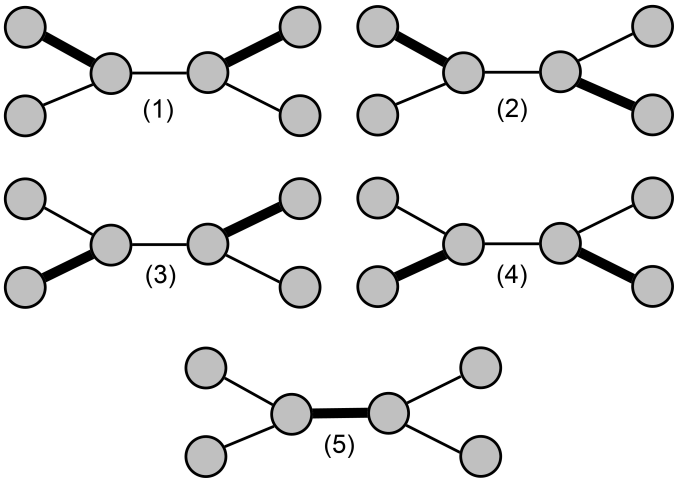

FIGURE 1.4 – Dessins des solutions du graphe papillon

qui sont optimales. Du point de vue du rapport d'approximation, cet algorithme est très mauvais alors que du point de vue pratique il est très bon. Ainsi la comparaison du rapport d'approximation de deux algorithmes différents (comme expliqué plus haut) ne permet pas de choisir le meilleur algorithme dans la pratique. A partir de cette remarque, on comprend que certains travaux se soient tournés vers l'étude des rapport d'approximation *en moyenne*. Cette dernière approche cherche à évaluer théoriquement le rapport d'approximation moyen des solutions que peut retourner un algorithme (voir par exemple la thèse de [Delbot 2009]). Une telle étude peut être bien plus complexe que l'approximation en pire cas. Ainsi la recherche des algorithmes à performance garantie a continué à se développer ces dernières années (voir dans [Vazirani 2001]).

1.3.3 Heuristiques

Contrairement aux algorithmes exacts ou aux algorithmes d'approximation, les heuristiques ne nécessitent pas d'analyses théoriques pour démontrer ou borner la qualité des solutions qu'elles retournent. Cependant certains travaux théoriques ont été réalisés et notamment [Fleury 1993] qui détermine des conditions nécessaires et suffisantes pour démontrer la convergence de certaines heuristiques vers les solutions optimales. Certaines d'entre elles sont aussi très efficaces et peuvent résoudre des problèmes de très grande taille (voir par exemple [Edelkamp 2011]). L'idée de tels algorithmes provient souvent de l'observation de la nature ou de la vie courante. On y distingue les *heuristiques*, les *métaheuristiques* et plus récemment les *hyperheuristiques* [Denzinger 1997].

Une heuristique est une idée que l'on applique à la résolution d'un problème sans garantir les performances. Pour la figure 1.3, l'idée par exemple, d'ajouter successivement le sommet de plus grand degré à la solution jusqu'à ce que toutes les arêtes soient

couvertes est une heuristique. Citons les plus connues comme l'algorithme du *gradient* [Snyman 2005] ou le *recuit simulé* [Aarts 1985].

Les métaheuristiques sont très utilisées de nos jours. Une métaheuristique est une idée de schéma global applicable et adaptable à différents problèmes. En plus de posséder différents outils pour guider et améliorer la qualité des solutions, les métaheuristiques peuvent combiner différentes heuristiques. Le travail consiste alors à bien maîtriser ces outils et les heuristiques pour savoir comment les combiner pour chaque type de problème. L'essaim particulaire, les colonies de fourmis, l'algorithme génétique ne sont que quelques métaheuristiques connues parmi les nombreuses qui sont utilisées.

Le terme d'hyperheuristique a été utilisé pour la première fois en 1997 par Denzinger *et al.* [Denzinger 1997] dans la cadre d'un travail sur l'intelligence artificielle. Une hyperheuristique est une méthode de *recherche et de génération* de métaheuristique. Les hyperheuristiques doivent avoir une connaissance précise des différentes heuristiques et métaheuristiques afin d'être efficaces. Là où les heuristiques et les métaheuristiques recherchent des solutions de bonne qualité, le rôle des hyperheuristiques est de choisir la métaheuristique la plus adéquate.

Comme nous l'avons remarqué, certaines heuristiques sont capables de résoudre des problèmes dans des graphes de grandes tailles, l'incertitude sur la qualité de la solution peut être néanmoins un handicap. Sans connaissance précise de la taille des solutions optimales, les heuristiques comparent leurs solutions avec les bornes inférieures, afin d'évaluer la qualité de leur approche. La validation expérimentale prend ainsi une très grande place dans ce domaine de la recherche. Souvent, les résultats expérimentaux d'une nouvelle heuristique sont comparés à des résultats d'anciennes méthodes, testées sur des instances *benchmark* accessibles à tous. Or, cela ne présage en rien de la qualité de l'heuristique sur les autres instances. Néanmoins de plus en plus de travaux dans le domaine s'efforce de développer (ou d'adapter) les algorithmes sur des instances réelles, modélisées à partir de problèmes (souvent) industriels.

1.3.4 Notre apport

Nous présentons dans ce travail, une approche originale de partition du graphe pour résoudre deux problèmes classiques. Une partie de la thèse consiste à développer des algorithmes pour organiser et guider la recherche de la solution optimale (en complexité exponentielle) et une autre partie consiste à développer un nouvel algorithme d'approximation.

Nous avons ainsi développé des algorithmes exacts et des algorithmes d'approximation pour résoudre les problèmes de l'*indépendant dominant de taille minimum* et du *vertex cover* (dont les définitions seront données dans les parties concernées). L'approche par partitions en cliques nous a permis d'élaborer des algorithmes simples à mettre en œuvre et à modifier. Nous avons ainsi pu tester expérimentalement différents

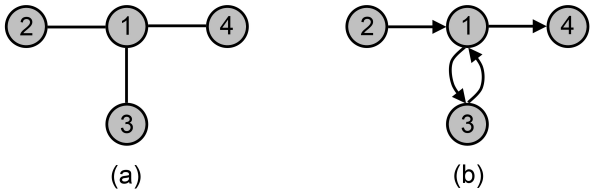

FIGURE 1.5 – Deux graphes non pondérés

filtres (qui permettent d'éviter de tester de *mauvaises* solutions lors de la recherche) pour améliorer ces algorithmes. Après une analyse théorique de leur comportement, nous avons testé tous nos algorithmes sur différentes familles de graphes dont celle des graphes aléatoires.

En faisant des analyses théoriques et des expérimentations numériques, nous essayons dans cette thèse d'apporter une étude *complète* sur notre méthode dite par *partition en cliques*. Cette approche est suffisamment générique pour permettre de concevoir des algorithmes exacts et des algorithmes d'approximation. Nous avons combiné cette approche avec des heuristiques gloutonnes et un algorithme de liste, ce qui nous a permis de résoudre des problèmes dont les instances vont jusqu'à plusieurs centaines de sommets (en quelques secondes dans le cas de l'approximation). Ainsi nous avons implémenté tous nos algorithmes et nous avons comparé nos résultats avec d'autres travaux, lorsque cela est possible. Nous avons également implémenté plusieurs algorithmes, inspirés de méthodes classiques, afin de comparer expérimentalement les résultats, lorsque cela s'avère nécessaire.

1.4 Notations et définitions

Nous introduisons dans cette section quelques **notations** et **définitions** générales sur les graphes, qui seront utiles pour la suite de notre manuscrit. Des rappels, précisions et compléments seront apportés dans les sections lorsque cela s'avère nécessaire.

Pour commencer, nous noterons $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ le *graphe non orienté* et *non pondéré* avec son *ensemble de sommets* $\mathcal{V}$ et son *ensemble d'arêtes* $\mathcal{E}$. *Non pondéré* indique qu'il n'y a pas de poids sur les sommets ou sur les arêtes ; plus précisément, cela indique que les sommets et les arêtes ont le même poids, usuellement unitaire. *Non orienté* signifie que le lien entre deux sommets est à double sens. C'est à dire que l'*arête* uv entre les sommets u et v permet d'aller de u à v et de v à u. Si uv est une arête entre les sommets u et v, alors u et v sont *voisins* ; on note $N(u)$ l'ensemble des voisins de u. Lorsque ce lien est orienté, on l'appelle *arc*. Sur la figure, le graphe (a) est non orienté et non pondéré avec 4 sommets et 3 arêtes. Le graphe (b) est orienté. On note généralement $n = |\mathcal{V}|$ le nombre de sommets.

Dans notre travail et sans précision, le graphe considéré est non orienté et non

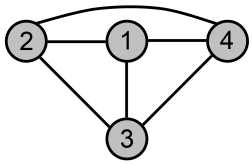

FIGURE 1.6 – Graphe non orienté et non pondéré complet

FIGURE 1.7 – Deux sous graphes induits

pondéré. On dit qu'un tel graphe est un *graphe complet si et seulement si* les sommets sont reliés *deux à deux* par une arête (voir figure 1.6). Il existe alors une formule pour calculer le nombre total d'arêtes en fonction du nombre de sommets n : $NbAretes = \frac{n(n-1)}{2}$. Les graphes complets sont indifféremment nommés *cliques* et noté K_n, pour un graphe de n sommets. A l'opposé, on appelle *stable* ou *indépendant* un ensemble de sommets qui ne sont liés par aucune arête. Soit $H \subseteq \mathcal{V}$ un sous ensemble de sommets de $\mathcal{G}$, on note $\mathcal{G}[H]$ le sous graphe de $\mathcal{G}$ *induit* par H dans $\mathcal{G}$: $\mathcal{G}[H]$ est le graphe d'ensemble de sommets H et d'ensemble d'arêtes qui relient les sommets de H : $\{uv : uv \in \mathcal{E}, u \in H$ et $v \in H\}$. Sur la figure 1.7 : (a) le sous graphe induit de $\mathcal{G}$ par les sommets 1 et 2 qui est une clique de taille 2 noté K_2 ; (b) la sous graphe induit de $\mathcal{G}$ par les sommets 2, 3 et 4 qui est un stable ou un ensemble indépendant de taille 3.

Première partie

Problème de l'indépendant dominant de taille minimum

Chapitre 2

Présentation du problème

Sommaire

De nombreux problèmes ne peuvent être résolus de manière exacte en temps polynomial. Il existe alors des algorithmes exponentiels pour les résoudre. L'objectif est alors de concevoir des algorithmes avec la plus faible complexité exponentielle possible. Cependant de nombreux travaux théoriques ne débouchent pas toujours sur des implémentations et des expérimentations numériques, comparables dans la pratique.

Nous présentons dans cette partie un problème bien connu : l'*indépendant dominant de taille minimum*. Après avoir formellement défini le problème, nous présenterons les premiers résultats sur une famille particulière de graphes : les *line graphs*. Puis dans le chapitre 3, nous présenterons de manière complète une nouvelle approche pour résoudre le problème et nous analyserons les résultats de nos expériences sur huit familles de graphes différentes.

2.1 Indépendant dominant de taille minimum

Soit $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ un graphe non orienté et non pondéré où $\mathcal{V}$ est l'ensemble des *sommets*, $\mathcal{E}$ est l'ensemble des *arêtes* et $n = |\mathcal{V}|$ est sa taille. Si uv est une arête entre les sommets u et v, alors u et v sont *voisins*. Un ensemble *dominant* dans le graphe $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ est un sous-ensemble $D \subseteq \mathcal{V}$ tel que chaque sommet $v \in \mathcal{V} - D$ est voisin d'au moins un sommet dans D. Un ensemble *indépendant* dans $\mathcal{G}$ est un sous-ensemble I de sommets tel que deux sommets quelconque de $I \subseteq \mathcal{V}$ ne sont pas voisins. Le problème d'optimisation de l'*indépendant dominant de taille minimum* (minimum Independent Dominating Set : *mIDS*) consiste à trouver un ensemble indépendant et dominant (*IDS*) de *taille minimum* dans un graphe $\mathcal{G}$ donné. La figure 2.1 montre deux sous-ensembles de type *IDS* (les sommets foncés) pour un graphe de 7 sommets et 11

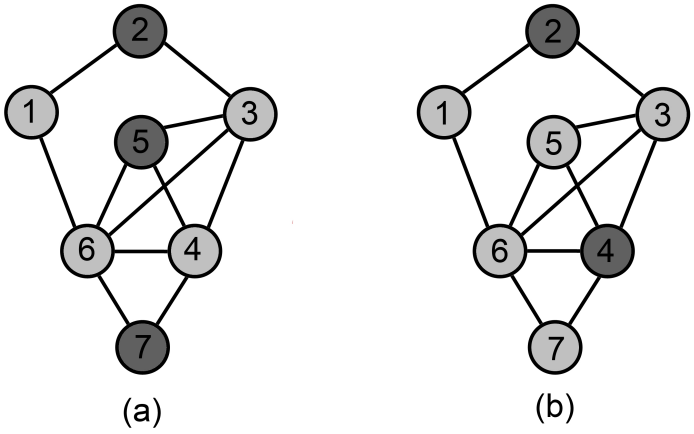

FIGURE 2.1 – Indépendant dominant dans un graphe

arêtes. Le graphe (b) de la figure, montre un *mIDS*. Ce problème est connu pour être $\mathcal{NP}$-*difficile* [Garey 1979]. Magnús M. Halldórsson dans [Halldórsson 1993] a montré en 1993 que dans les graphes généraux, le problème du *mIDS* ne peut pas être approximé sous un facteur $n^{1-\varepsilon}$ avec $\varepsilon > 0$ en temps polynomial, à moins que $\mathcal{P} = \mathcal{NP}$. Les problèmes de types indépendant ou dominant sont liés dans la pratique à des problèmes d'optimisation de réseaux de communication (voir par exemple [Kuhn 2005]).

Par exemple, considérons le réseau des postes de surveillance et de secours du Puy de Dôme représenté figure 2.2. Sur la figure suivante (2.3), on peut voir la zone de couverture qui permet à un point de *transmettre* (émetteur) et de *recevoir* (récepteur) des informations des autres points. Le but est de permettre à tous les postes de recevoir des informations. Un poste qui a un récepteur peut recevoir des informations d'un poste voisin qui possède un émetteur. Dans le réseau actuel, chaque poste possède un récepteur et un émetteur. Lorsqu'une information importante doit être transmise, il suffit de prévenir un des postes de secours et l'information sera successivement transmise à tous les autres. Dans un soucis d'optimisation du budget, le gestionnaire souhaite diminuer le nombre d'émetteurs. Le problème peut ainsi être modélisé en un problème de domination dans le graphe de la figure 2.4, dans lequel deux sommets sont voisins s'ils sont assez proches pour la transmission des informations. Ainsi, avec "seulement" quatre postes émetteurs (les sommets en gras sur la figure 2.4) l'information peut être transmise à tous, puisque ces postes de secours couvrent tous les autres.

Pour commencer, nous allons montrer dans la section suivante que ce problème peut être approximé avec un facteur 2 dans les *line graphs*.

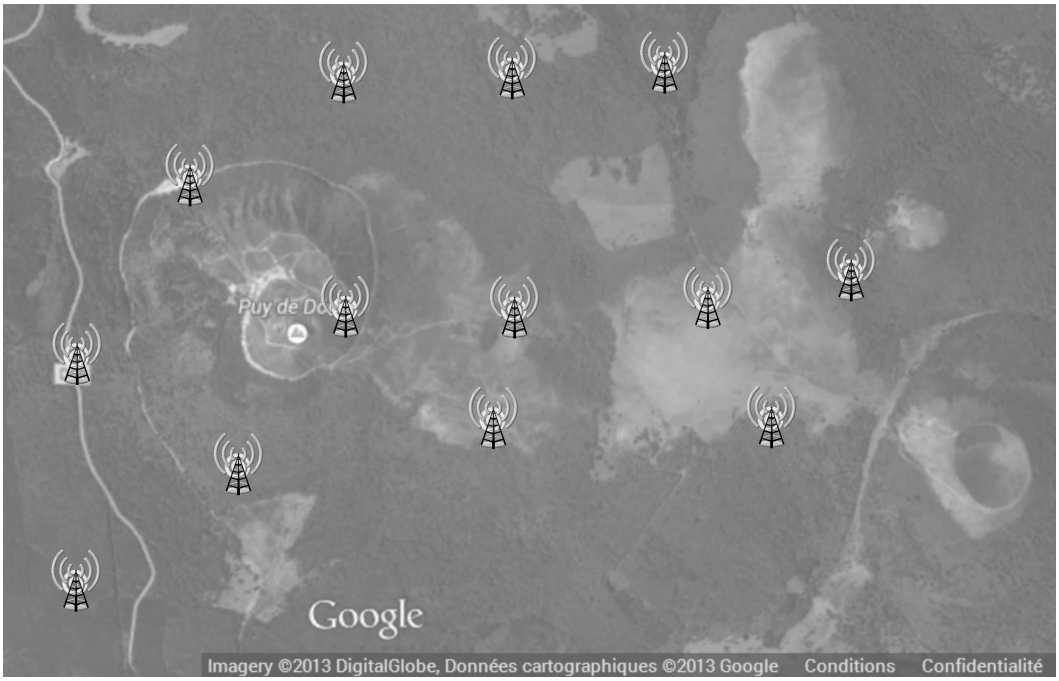

FIGURE 2.2 – Réseau de postes de secours du Puy-de-Dôme

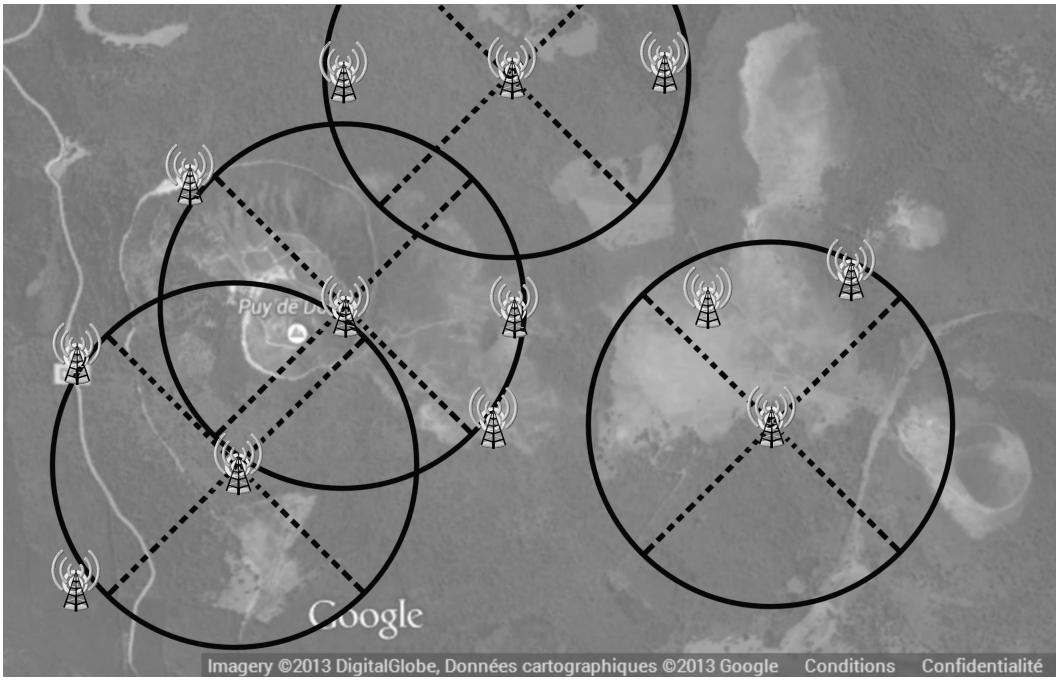

FIGURE 2.3 – Zone de couverture de quatre postes de secours

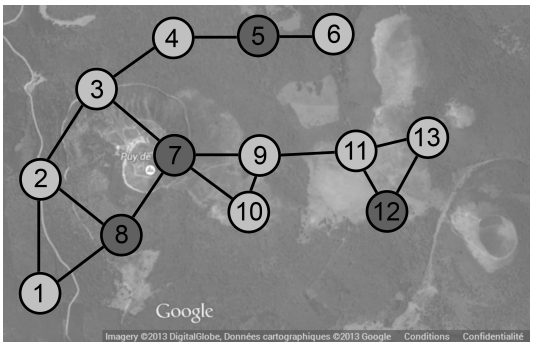

FIGURE 2.4 – Modélisation en problème de domination dans un graphe pour minimiser le nombre d'émetteurs dans le réseau de communication des postes de sécurité du Puy-de-Dôme

2.2 Facteur d'approximation de 2 dans les line graphs

Rappelons pour commencer que le problème d'optimisation du *mIDS* est $\mathcal{NP}$-*difficile* dans les graphes généraux et qu'il n'est pas approximable avec un facteur inférieur à $n^{1-\varepsilon}$ [Halldórsson 1993]. Dans cette section, nous présentons nos premiers résultats sur ce problème. Nous exhibons une famille de graphes particulière dans laquelle le problème est $\mathcal{APX}$-*complet*, avec un facteur d'approximation de 2. Pour ce faire, nous passons par le problème du *couplage maximal de taille minimum*.

Soit $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ un graphe non orienté, un *couplage* $\mathcal{M} \subseteq \mathcal{E}$ (ou matching) est un ensemble d'arêtes deux à deux non adjacentes, *c'est-à-dire* n'ayant pas de sommets en communs. Un couplage est *maximal* au sens de l'inclusion (Maximal Matching : *MM*) s'il n'existe pas d'ensemble strictement plus grand contenant le couplage et qui soit aussi un couplage. Un couplage est *minimum* s'il est de taille minimum. Le graphe (a) de la figure 2.5 illustre (arêtes en gras) un exemple de couplage maximal et le graphe (b) illustre un couplage maximal de *taille minimum* (minimum Maximal Matching : *mMM*). Construire un *MM* peut être fait en temps polynomial avec un algorithme glouton alors que le problème d'optimisation, *c'est-à-dire* construire un *mMM* est $\mathcal{APX}$-*complet* ([Yannakakis 1980]) avec un facteur d'approximation de 2. Remarquons que *n'importe quel MM* est une 2-approximation du *mMM* (voir p.370 de[Ausiello 2000]).

Soit $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ un graphe non orienté, le *line graph* de $\mathcal{G}$, noté $LG = (LV, LE)$ (et aussi appelé *interchange graph* [Ore 1962] (voir pp. 19-21), *adjoint* [Menon 1965], et *edge-to-vertex dual* [Seshu 1961] (voir p.296)) est un graphe dans lequel à chaque sommet correspond une arête différente de $\mathcal{G}$, telle que deux sommets sont voisins *si et seulement si* leurs arêtes correspondantes dans $\mathcal{G}$ sont adjacentes. La figure 2.6 donne un exemple d'une telle transformation : les arêtes de $\mathcal{G}$ sont nommées de a à h et correspondent aux sommets de même nom dans LG. Nous allons montrer dans la section 2.2

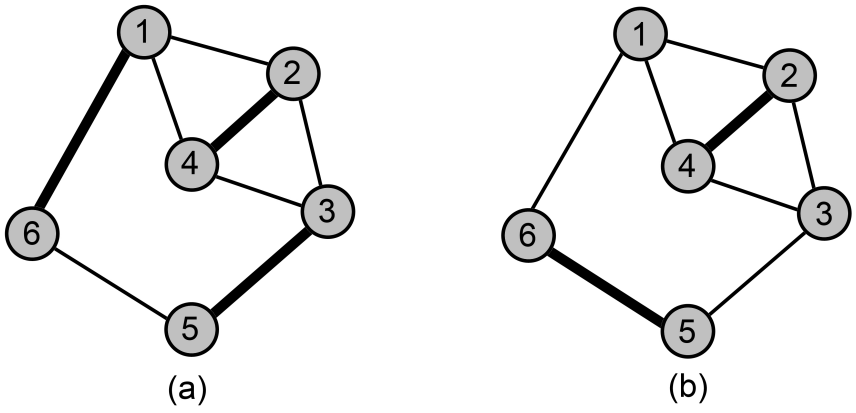

FIGURE 2.5 – Deux couplages dans un graphe

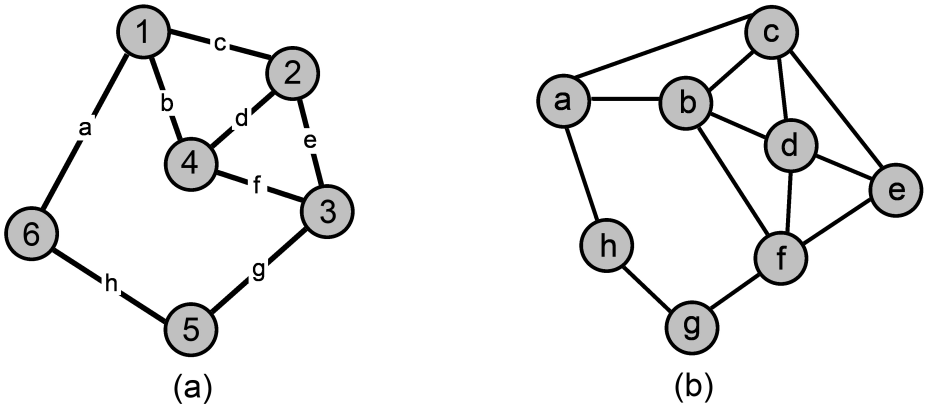

FIGURE 2.6 – (a) Graphe $\mathcal{G}$. (b) Line graph associé à $\mathcal{G}$

que chercher un *mMM* dans $\mathcal{G}$ revient à chercher un *mIDS* dans *LG*.

Le graphe $\mathcal{G}$ est appelé le *graphe racine* (ou root graph) du graphe *LG*. A tout $\mathcal{G}$ correspond un unique *LG* [Lehot 1974] ; mais tous les graphes ne sont pas nécessairement des line graphs. En effet, par exemple, le graphe de la figure 2.7 n'est pas un line graph car il n'est pas possible de découper le graphe en sous-ensembles de clique disjoints tel que chaque sommet appartient à au plus deux cliques [Lehot 1974] : le sommet numéro 1 appartient à trois cliques, quelque soit le découpage (plus généralement, un line graph *LG* est un *claw-free graph*). La transformation inverse (*LG* vers $\mathcal{G}$) n'est donc pas triviale.

La figure 2.8 montre la transformation d'un *line graph* vers son graphe racine. Les arêtes de *LG* forment des cliques numérotées de 1 à 5, de telle sorte qu'un sommet appartient à au plus deux cliques. Chaque clique donne un sommet différent dans le

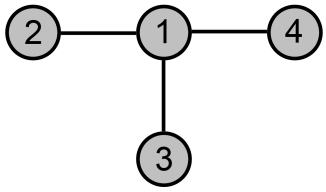

FIGURE 2.7 – Graphe qui n'est pas un line graph

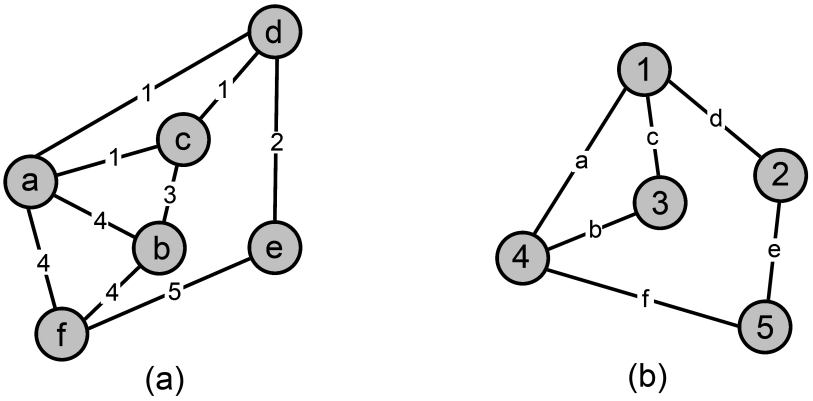

FIGURE 2.8 – (a) Line graph LG. (b) Graphe racine associé à LG

graphe $\mathcal{G}$: il y a correspondance entre les numéros des arêtes et les lettres des sommets dans LG avec les numéros des sommets et les lettres des arêtes dans $\mathcal{G}$. Si deux cliques dans LG sont *voisines* (si elles ont au moins un sommet en commun), alors les deux sommets correspondant dans $\mathcal{G}$ sont liés. A part quelques exceptions données par les auteurs, ce partitionnement en cliques du line graph est unique [Roussopoulos 1973]. Roussopoulos [Roussopoulos 1973] en 1973 et Lehot [Lehot 1974] en 1974 ont également proposé plusieurs méthodes pour déterminer l'existence et retrouver le cas échéant le *graphe racine* en temps polynomial.

Considérons un line graph $LG = (LV, LE)$ et le graphe racine correspondant $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. Nous montrons que quel que soit un indépendant dominant (Independent Dominating Set : *IDS*) $\mathcal{S}_{IDS}$ dans LG, sa taille est forcément au plus *deux fois* la taille d'un *mIDS*.

Théorème 1 *$\mathcal{S}_{\text{IDS}} \subseteq LV$ est un indépendant dominant dans LG si et seulement si il existe un couplage maximal $\mathcal{M}_{\text{MM}} \subseteq \mathcal{E}$ de $\mathcal{G}$ tel que les arêtes de $\mathcal{M}_{\text{MM}}$ correspondent aux sommets de $\mathcal{S}_{\text{IDS}}$. Ainsi on a $|\mathcal{S}_{\text{IDS}}| = |\mathcal{M}_{\text{MM}}|$.*

`Preuve.` Soit un *IDS* $\mathcal{S}_{IDS}$ de LG. Construisons $\mathcal{M}$ en prenant les arêtes de $\mathcal{G}$ correspondant aux sommets de $\mathcal{S}_{IDS}$. On a $|\mathcal{S}_{IDS}| = |\mathcal{M}|$. La figure 2.9 montre une telle

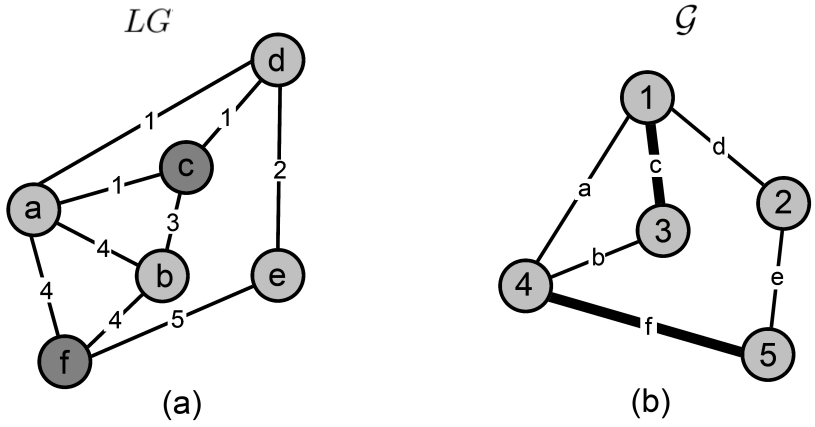

FIGURE 2.9 – (a) *IDS* dans un line graph *LG*. (b) *MM* dans le graphe racine associé à *LG*

transformation, avec le sous-ensemble de sommets $\mathcal{S}_{IDS}$ dans LG (a) et le sous-ensemble d'arêtes $\mathcal{M}$ dans $\mathcal{G}$ (b). La solution $\mathcal{M}$ est un *couplage* : dans le cas contraire, il existerait 2 arêtes de $\mathcal{M}$ qui seraient adjacentes, donc les sommets correspondant dans $\mathcal{S}_{IDS}$ seraient également liés, ce qui serait absurde puisque $\mathcal{S}_{IDS}$ est un indépendant. Ce couplage est *maximal*, sinon il existerait une arête $e \in \mathcal{E}$ qui n'est adjacente à aucune autre arête de $\mathcal{M}$. Donc, le sommet dans LG correspondant à l'arête e ne serait lié à aucun sommet de $\mathcal{S}_{IDS}$, ce qui est absurde puisque $\mathcal{S}_{IDS}$ est un dominant.

Réciproquement, prenons un couplage maximal $\mathcal{M}_{MM}$ de $\mathcal{G}$ et construisons $\mathcal{S}$ en prenant les sommets dans LG qui correspondent aux arêtes de $\mathcal{M}_{MM}$ ($|\mathcal{S}| = |\mathcal{M}_{MM}|$). La figure 2.10 illustre cette construction avec, (a) le *MM* dans le graphe $\mathcal{G}$ et (b) l'*IDS* dans le graphe LG. Montrons que $\mathcal{S}$ est un *indépendant*. En effet, si ce n'était pas le cas, il existerait 2 sommets dans $\mathcal{S}$ qui seraient liés et leurs arêtes associées dans $\mathcal{M}_{MM}$ seraient adjacentes. C'est absurde car $\mathcal{M}_{MM}$ est un couplage. Enfin, $\mathcal{S}$ est un *dominant* : sinon il existerait un sommet dans LG qui n'est lié à aucun sommet de $\mathcal{S}$. Donc l'arête correspondante dans $\mathcal{G}$ ne serait adjacente à aucune arête de $\mathcal{M}_{MM}$. Ce n'est pas possible car $\mathcal{M}_{MM}$ est maximal.

□

Corrolaire 1 *$\mathcal{S}_{\text{mIDS}}$ est un* mIDS *d'un line graph LG si et seulement si il existe un couplage maximum de taille minimum $\mathcal{M}_{\text{mMM}} \subseteq \mathcal{E}$ de $\mathcal{G}$ tel que ses arêtes correspondent aux sommets de $\mathcal{S}_{\text{mIDS}}$. Ainsi on a $|\mathcal{S}_{\text{mIDS}}| = |\mathcal{M}_{\text{mMM}}|$.*

Corrolaire 2 *Dans un line graph LG, trouver un* mIDS *est un problème $\mathcal{APX}$-*complet*, avec un facteur d'approximation de 2. Ce facteur est atteint.*

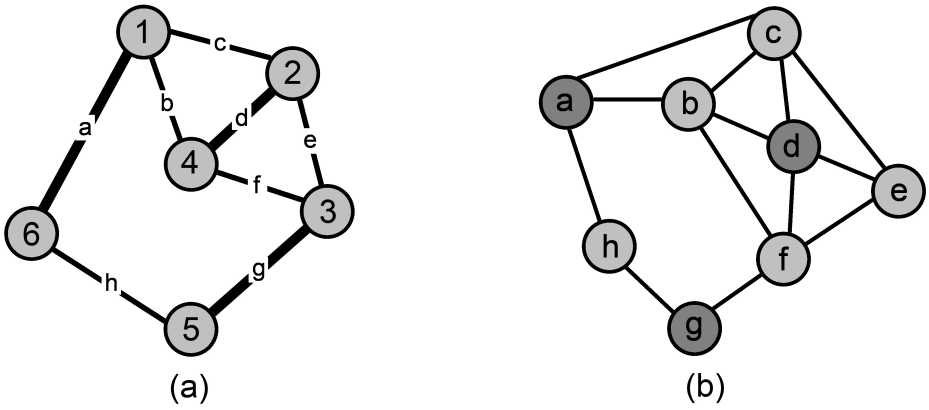

FIGURE 2.10 – Construction d'un *IDS* à partir d'un *MM*

`Preuve.` De manière évidente, le problème est dans $\mathcal{NP}$. Soit $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ un graphe non orienté et non pondéré et $\mathcal{S}_{IMS} \subseteq \mathcal{V}$ un *indépendant maximal* de $\mathcal{G}$, *c'est-à-dire* qu'il n'existe pas de sous-ensembles *indépendants* strictement plus grands que $\mathcal{S}_{IMS}$ contenant $\mathcal{S}_{IMS}$. Dans ce cas, $\mathcal{S}_{IMS}$ est également un *dominant* de $\mathcal{G}$. En effet, si l'on suppose qu'il existe un sommet $v \in \mathcal{V}$ non dominé, alors ni v ni aucun de ses voisins, ne sont dans $\mathcal{S}_{IMS}$. De là, $\mathcal{S}_{IMS} \cup \{v\}$ est un *indépendant* de $\mathcal{G}$, ce qui est absurde. Finalement, $\mathcal{S}_{IMS}$ est un *indépendant dominant*, que nous renommons $\mathcal{S}_{IDS}$.

Le théorème 1 montre qu'à tout *IDS* dans le line graph correspond un *MM* dans le graphe racine de la même taille ; il en est de même pour les *mIDS* et *mMM* par le corrolaire 1. Or, $|\mathcal{M}_{MM}| \leq 2|\mathcal{M}_{mMM}|$ pour tout graphe (voir p.370 de [Ausiello 2000]). De ce fait, on en déduit que dans les line graphs $|\mathcal{S}_{IDS}| \leq 2|\mathcal{S}_{mIDS}|$. Comme la construction d'un indépendant maximal peut être facilement réalisée par un algorithme glouton, le problème est dans $\mathcal{APX}$. De plus, nous avons montré que la procédure pour obtenir un line graph à partir d'un graphe est simple (voir la section 2.2). Le *mMM* ($\mathcal{APX}$*-complet* [Yannakakis 1980]) se réduit donc de manière polynomiale au *mIDS* et nous avons notre résultat.

Le rapport 2 est atteint, par exemple pour le chemin de 3 sommets a, b, c dans cet ordre, qui est le line graph d'un chemin de 4 sommets. Le *mIDS* est constitué du seul sommet b, alors qu'il existe un *IDS* constitué des deux sommets a, c. □

Par ces démonstrations, les line graphs forment une famille dans laquelle le problème du *mIDS* est *intrinsèquement* 2-approximable ; *c'est-à-dire* que quel que soit un *IDS* dans un line graph sa taille est au plus *deux* fois celle du *mIDS*. Ce n'est hélas pas le cas pour les graphes généraux.

2.3 Bilan

Dans ce chapitre, nous avons présenté le problème de l'*indépendant dominant de taille minimum*. Nous avons montré qu'il a un rapport d'approximation de 2 dans les line graphs. Malheureusement, le problème n'est pas approximable dans le cas général. L'approche classique de résolution par énumération de tous les sous-ensembles est alors possible, mais fastidieuse et longue.

Le chapitre suivant sera l'occasion pour nous de présenter une nouvelle méthode exacte (mais moins fastidieuse) pour l'indépendant dominant de taille minimum. Nous présenterons de nombreux tests qui nous permettent de valider et compléter les analyses théoriques. Nous comparerons également notre approche avec celles de deux autres travaux. Pour de futurs tests, notre programme et les instances de nos graphes sont en ligne [1].

1. http ://todo.lamsade.dauphine.fr/spip.php ?article42

Chapitre 3

Algorithmes pour le *mIDS*

Sommaire

Dans ce chapitre, nous présentons une nouvelle approche pour résoudre de manière *exacte* le problème de l'indépendant dominant de taille minimum, dans les graphes généraux. Une idée simple de résolution consisterait à énumérer toutes les solutions possibles. En supposant qu'il faut un millième de seconde pour générer, tester et sauvegarder une solution, il faudrait alors plus de 35000 ans pour traiter un graphe de 50 sommets. Cette approche par *énumération* a une complexité exponentielle en $O^*(2^n)$ [1] (où n est la nombre de sommets). Heureusement, elle a été améliorée en 1988 par Johnson *et al.* [Johnson 1988]. En 2006, Liu et Song [Liu 2006] ont développé une approche élégante en $O^*(\sqrt{3}^n)$ [2] pour résoudre le *mIDS* dans les graphes généraux. Ils utilisent les

1. La notation $O^*(.)$ est utilisée pour exprimer la partie non polynomiale de la complexité d'un algorithme, ignorant les facteurs polynomiaux
2. $\sqrt{3}^n \approx 2^{0.792n}$

couplages maximum pour partitionner le graphe et réduire ainsi la complexité. Bourgeois, Escoffier, et Paschos [Bourgeois 2010] ont, en 2010, proposé un algorithme de branchement pour résoudre le problème en $O^*(2^{0.424n})$. Tous ces travaux ont permis d'évaluer ces techniques dans le pire des cas mais aucun des algorithmes n'a été implémenté et testé par leurs auteurs. Néanmoins, en 2011, Potluri et Negi [Potluri 2011] ont testé l'approche de Liu et Song afin de comparer avec leur méthode appelée Intelligent Enumeration Algorithm.

Dans ce qui suit, nous proposons une méthode qui utilise les *partitions en cliques* du graphe (obtenues de manière gloutonne) et nous prouvons que lorsque le graphe est grand et suffisamment dense, la complexité de la méthode peut être meilleure que n'importe quelle autre méthode. À travers cette analyse de la complexité, nous présentons également une méthode qui fournit une borne inférieure pour notre problème. Basé sur ces travaux algorithmiques, nous avons codé une procédure récursive incluant des "filtrages" pour une recherche efficace de solutions optimales. Nous l'avons testé et elle est capable de traiter des instances de 50 sommets dans un temps raisonnable (de l'ordre de quelques secondes), et des instances d'une centaine de sommets pour certaines familles de graphes.

Pour aller plus loin et traiter des graphes de plus grande taille, nous étudions un algorithme glouton pour l'indépendant dominant dans les graphes généraux. Expérimentalement, l'algorithme fournit de bonnes solutions (au plus 5 fois l'optimal) et il est capable de résoudre des graphes jusqu'à 65000 sommets en moins de 20 secondes. Ce genre d'heuristiques gloutonnes sont de très bonnes méthodes pour obtenir une solution initiale nécessaire pour l'algorithme exact. Nous décrivons également des cas particuliers qui *piègent* cet algorithme.

3.1 Approche par partition en cliques

Dans [Liu 2006], Liu et Song utilisent le couplage maximum pour réduire la complexité du *mIDS* à $O^*(2^n)$. Dans cette partie, nous étendons leur technique en partitionnant les sommets du graphe dans des *cliques* (une clique est un graphe dans lequel chaque paire de sommets est liée par une arête) au lieu de les coupler. Pour décrire notre méthode, nous rappelons quelques **définitions** et **notations**. On appelle une *partition en cliques* d'un graphe $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, une partition des sommets $\mathcal{V}$ en sous-ensembles disjoints $C = \{C_1, ..., C_k\}$ ($\cup_{i=1}^{k} C_i = \mathcal{V}$ et si $i \neq j$, $C_i \cap C_j = \emptyset$) tels que chaque sous-graphe induit par C_i dans $\mathcal{G}$ est une clique. La taille de chaque clique C_i (autrement dit le nombre de sommets dans la clique) est notée c_i ($1 \leq c_i \leq n$). Une clique C_i est dite triviale *si et seulement si* $c_i = 1$. La figure 3.1 fournit un exemple de partition en cliques $C = \{C_1, C_2, C_3\}$ d'un graphe $\mathcal{G}$ de 7 sommets et 10 arêtes. Les cliques ont respectivement 2, 4 et 1 sommets.

Une partition en cliques d'un graphe $\mathcal{G}$ peut facilement être obtenue par un algo-

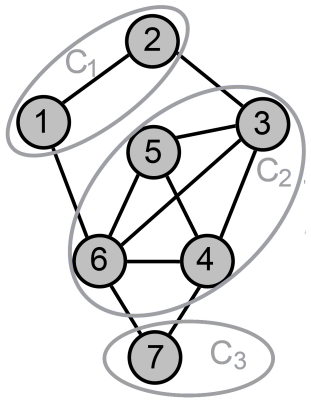

FIGURE 3.1 – Partition en cliques d'un graphe

rithme glouton. Supposons maintenant que nous avons une partition $C = \{C_1, ..., C_k\}$ de $\mathcal{G}$. Comme chaque C_i est une clique, chaque indépendant dominant a *au plus* un sommet dans chaque C_i (autrement la solution ne serait pas indépendante). En se basant sur cette remarque, l'idée de notre algorithme est de construire tous les sous-ensembles de sommets contenant *au plus un sommet* dans chacune des cliques de C et pour chaque sous-ensemble $\mathcal{S}$, tester si c'est un indépendant (il n'y a pas d'arêtes entre les sommets de $\mathcal{S}$) et un dominant (chaque sommet dans $\mathcal{V}\backslash\mathcal{S}$ a au moins un voisin dans $\mathcal{S}$). Si c'est le cas, nous mettons à jour la solution du *mIDS* et retournons l'optimale (la meilleure) à la fin. Notons que chaque test peut s'effectuer en temps polynomial.

A partir de la construction décrite, nous générons tous les sous-ensembles *IDS* (ainsi que les sous-ensembles qui ne le sont pas) et donc aussi toutes les solutions *mIDS*.

Chaque sous-ensemble testé $\mathcal{S}$ se compose d'au plus un sommet dans chaque C_i. Cela représente exactement $c_i + 1$ possibilités pour chaque clique C_i de C. Ainsi, le nombre de *candidats* $\mathcal{S}$ testés pendant notre algorithme est exactement :

$$\prod_{i=1}^{k} (c_i + 1) \ .$$

Comme chaque test est polynomial, notre algorithme est en $O^*\left(\prod_{i=1}^{k} (c_i + 1)\right)$.

La figure 3.2 présente tous les sous-ensembles candidats $\mathcal{S}$ générés et testés par notre algorithme pour deux partitions en cliques différentes du même graphe $\mathcal{G}$. Les sous-ensembles en noir sont des *IDS* (ceux qui sont grisés ne sont pas des *IDS*) et les sous-ensembles entourés sont des *mIDS*.

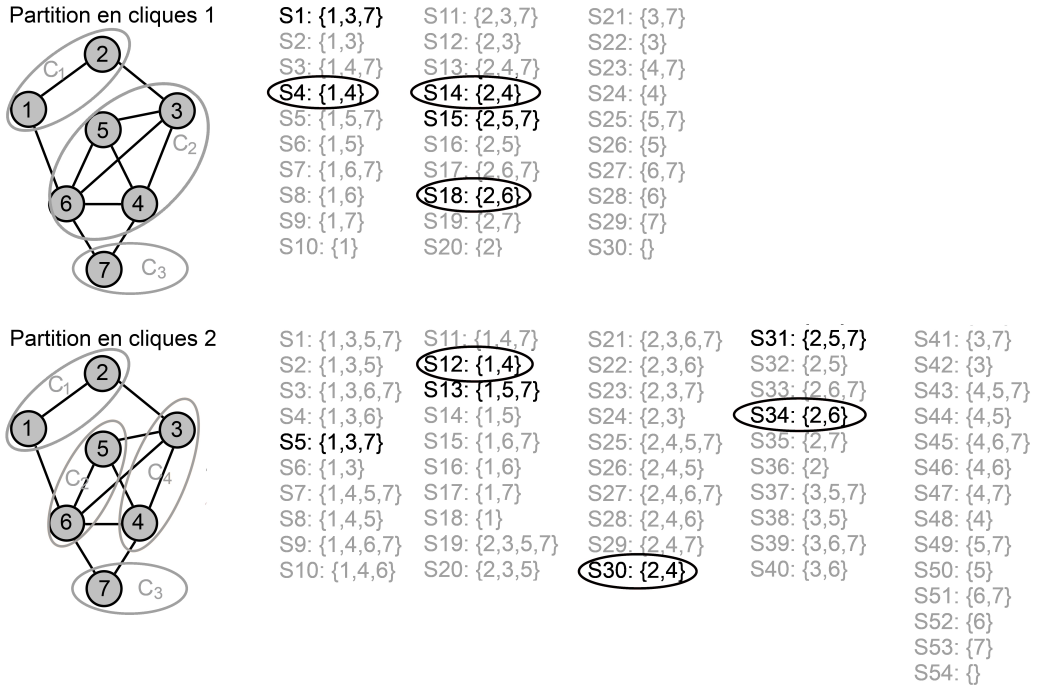

FIGURE 3.2 – Sous-ensembles candidats à partir de deux partitions en cliques différentes. En noir les sous-ensembles qui sont des *IDS*. En noir entouré, les sous-ensemble qui sont des *mIDS*

3.2 Complexité et comparaisons

Le but de cette section est de montrer que lorsque $|\mathcal{V}| = n$ est suffisamment grand, notre algorithme a un fort potentiel d'avoir une complexité meilleure que n'importe quelle autre complexité exponentielle du type A^{bn} où $(A, b) \in (\mathbb{N}\backslash\{0, 1\} \times \mathbb{R}_+^*)$. Plus précisément, nous donnons une condition analytique sous laquelle nous avons :

$$\prod_{i=1}^{k} (c_i + 1) \leq A^{bn} \ . \tag{3.1}$$

Comme dans la section 3.1, les $c_1, \ldots, c_k$ représentent la taille des cliques de la partition initiale de $\mathcal{V}$. Nous avons alors :

$$\sum_{i=1}^{k} c_i = n \ .$$

Les *entiers* $c_1, ..., c_k$ représentent ce qu'on appelle une *partition* de l'entier n. Plus généralement, soit $n > 0$ un entier positif quelconque, une *partition de* n est une liste de $l > 0$ entiers strictement positifs $[n_1, \ldots, n_l]$ tels que :

$$\sum_{i=1}^{l} n_i = n \ .$$

Pour un graphe donné, il est difficile d'énumérer toutes les partitions en cliques différentes ou même seulement de déterminer leur nombre exact. Ainsi, nous faisons une étude probabiliste des partitions de l'entier n. La *taille* de la partition $[n_1, \ldots, n_l]$ est son nombre l d'éléments. Deux listes qui se composent des mêmes éléments (entiers) dans un ordre différent sont considérées comme *égales*. Nous notons $P(n)$ l'ensemble de toutes les partitions de n et $p(n)$ sa taille. Par exemple, si $n = 5$, nous avons :

$$P(5) = \{[1,1,1,1,1], [1,1,1,2], [1,2,2], [1,1,3], [2,3], [1,4], [5]\} \text{ et } p(5) = 7 \ .$$

Nous notons maintenant $Q(n)$ l'ensemble des partitions (et $q(n)$ sa taille) de l'entier $|\mathcal{V}|$ qui ne vérifient pas l'inégalité (3.1), quand A et b sont donnés. Par exemple, si $A = 3$ et $b = \frac{1}{2}$, lorsque $n = 5$ nous avons $Q(5) = \{[1,1,1,1,1], [1,1,1,2], [1,2,2], [1,1,3]\}$.

3.2.1 Analyse

Voyons maintenant que lorsque n tend vers l'infini, $\frac{q(n)}{p(n)}$ tend vers 0, montrant ainsi qu'asymptotiquement, la proportion de partitions de n qui violent (3.1) tend vers 0.

Pour ce faire, nous avons besoin de plusieurs résultats préliminaires. Le lemme 1 donne une condition suffisante pour une partition $c_1, \ldots, c_k$ de n pour vérifier l'inégalité (3.1). Cela prouve que si nous pouvons avoir une partition de n dans laquelle la valeur moyenne est supérieure à une certaine valeur à déterminer (noté $M(A,b)$), alors l'inégalité (3.1) est vérifiée.

Lemme 1 *Soit* $(A,b) \in (\mathbb{N}\backslash\{0,1\} \times \mathbb{R}_+^*)$. *Il existe* $M(A,b) \in \mathbb{R}$ *tel que :*

$$M(A,b) \leq \frac{1}{k}\sum_{i=1}^{k} c_i \Rightarrow \prod_{i=1}^{k}(c_i+1) \leq A^{bn} .$$

`Preuve.` Nous étudions la fonction

$$f(x) = A^x - \frac{1}{b}x - 1$$

et sa dérivée

$$f'(x) = A^x \ln(A) - \frac{1}{b} .$$

Posons $x_1 = \frac{\ln\left(\frac{1}{\ln(A)b}\right)}{\ln(A)}$; $\forall x > x_1$, $f(x)$ est croissante. De plus comme $\lim_{x\to+\infty} f(x) = +\infty$ il existe $g \geq x_1$ tel que $\forall x > g$, $f(x)$ est croissante et supérieure à 0.

Prenons $M(A,b) = \frac{g}{b}$. Supposons maintenant que $M(A,b) \leq \frac{1}{k}\sum_{i=1}^{k} c_i$ alors :

$$\frac{g}{b} = M(A,b) \leq \frac{1}{k}\sum_{i=1}^{k} c_i \quad \Leftrightarrow \quad g \leq b\frac{1}{k}\sum_{i=1}^{k} c_i .$$

Comme nous avons $\forall x \geq g$, $f(x) = A^x - \frac{1}{b}x - 1 \geq 0$, nous en déduisons que :

$$A^{b\frac{\sum_{i=1}^{k} c_i}{k}} - \frac{1}{b}b\frac{1}{k}\sum_{i=1}^{k} c_i - 1 \geq 0 .$$

$$\Leftrightarrow A^{b\frac{\sum_{i=1}^{k} c_i}{k}} \geq \frac{1}{k}\sum_{i=1}^{k} c_i + 1 = \frac{\sum_{i=1}^{k}(c_i+1)}{k} .$$

Nous rappelons l'*inégalité arithmético-géométrique* généralisée aux moyennes pondérées [Steele 2004], pour tout $y_1, ..., y_k \geq 0$:

$$\frac{\sum_{i=1}^{k} y_i}{k} \geq \left(\prod_{i=1}^{k} y_i\right)^{\frac{1}{k}} . \tag{3.2}$$

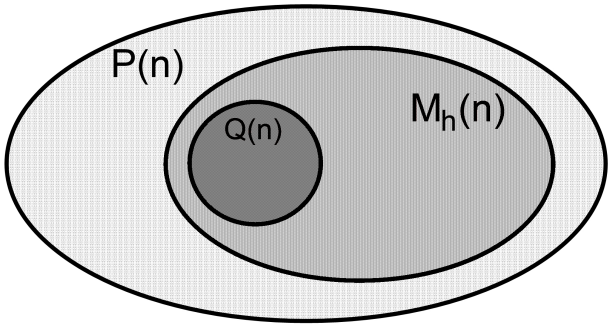

FIGURE 3.3 – Différents sous-ensembles de $P(n)$

Finalement, avec l'inégalité 3.2 et en utilisant $\sum_{i=1}^{k} c_i = n$, nous obtenons :

$$\prod_{i=1}^{k} (c_i + 1) \leq A^{bn} .$$

□

Le lemme 1 prouve que, avec A et b données, il existe une valeur $h = M(A, b)$ telle que, si $h \leq \frac{1}{k} \sum_{i=1}^{k} c_i$ alors $\prod_{i=1}^{k} (c_i + 1) \leq A^{bn}$. A partir de cette valeur h, nous définissons $M_h(n)$ comme l'ensemble (avec $m_h(n)$ sa taille) des partitions $[n_1, \ldots, n_l] \in P(n)$ de n qui vérifient $h > \frac{1}{l} \sum_{i=1}^{l} c_i$, *c'est-à-dire* qui ont une *valeur moyenne* strictement plus petite que h et qui ne respectent pas la condition suffisante du lemme 1 (quand $h = M(A, b)$). Si on note $n = n$ et $h = M(A, b)$, nous avons $Q(n) \subseteq M_h(n)$ car toutes les partitions $[n_1 \ldots, n_l]$ de $Q(n)$ vérifient $h > \frac{1}{l} \sum_{i=1}^{l} c_i$ (la figure 3.3 schématise ces différents sous-ensembles). Autrement, elles satisferaient la condition suffisante du lemme 1 et ainsi vérifieraient (3.1) et donc ne seraient pas dans $Q(n)$. En se basant sur cette remarque, on prouve dans ce qui suit que $\lim_{n \to +\infty} \frac{m_h(n)}{p(n)} \to 0$; ce qui est suffisant pour montrer que $\lim_{n \to +\infty} \frac{q(n)}{p(n)} \to 0$.

Tout d'abord remarquons que $[n_1 \ldots, n_l] \in M_h(n)$ vérifie $h > \frac{1}{l} \sum_{i=1}^{l} c_i$. Comme $\sum_{i=1}^{l} c_i = n$ on a $l > \dfrac{n}{h}$. Cela signifie que $M_h(n)$ est aussi l'ensemble des partitions de n de taille supérieure à $\frac{n}{h}$. C'est la raison pour laquelle nous introduisons maintenant l'ensemble $P(n, k)$ (de taille $p(n, k)$) des partitions de n de taille k, $k \in \mathbb{N}$.

Lemme 2 *$p(n, n - i) \leq p(i)$ pour tout $i \in \{1, \ldots, n\}$.*

`Preuve.` Soit $X = [x_1, ..., x_{n-i}] \in P(n, n - i)$. Soit $Z = [z_1, ..., z_l]$ la liste obtenue après soustraction de 1 à chaque élément de X et en ne conservant que les valeurs strictement supérieures à 0 ($\forall k \leq l \leq n - i, z_k \geq 1$).

Alors $\sum_{k=1}^{l} z_k = \sum_{k=1}^{n-i} x_i - (n-i) = n - (n-i) = i$ et Z est une partition de i. Maintenant, posons $X' = \left[x'_1, ..., x'_{n-i}\right] \in P(n, n-i)$, $X' \neq X$ étant une autre partition. Nous notons Z' la liste obtenue par la même méthode à partir de X'. Supposons que $Z = Z'$. Comme X et X' ont la même taille et que $Z = Z'$, ils ont le même nombre de 1 (qui ont *disparu* dans Z et Z'). La soustraction par 1 des l autres éléments (l est la taille de $Z = Z'$) donne Z et Z'. Cela veut dire que ces éléments strictement supérieurs à 1 sont égaux dans X et X' et que $X = X'$ (l'ordre des éléments dans la liste n'est pas important) ; ce qui est contradictoire avec notre hypothèse.

Cela montre que pour chaque partition différente de n de taille $n-i$, on peut extraire au moins une partition différente de i. Ainsi $p(n, n-i) \leq p(i)$. □

Pour les deux prochains résultats, nous supposons que A et b sont données. Soit $h = M(A, b)$.

Lemme 3 *Avec la notation précédente, nous avons :*

$$m_h(n) \leq 1 + \sum_{i=1}^{n-\lceil \frac{n}{h} \rceil} p(i) \ .$$

`Preuve.` Comme nous l'avons mentionné précédemment, toutes les partitions de $P(n)$ de taille supérieure à $\frac{n}{h}$, ont une valeur moyenne strictement inférieure à h. Ces partitions sont dans $M_h(n)$ ainsi $|M_h(n)| = m_h(n) = \sum_{k=\lceil \frac{n}{h} \rceil}^{n} p(n,k)$. Comme $p(n,n) = 1$, nous pouvons déduire :

$$m_h(n) = 1 + \sum_{k=\lceil \frac{n}{h} \rceil}^{n-1} p(n,k) \ .$$

Maintenant, posons $k = n - i$. On a :

$$m_h(n) = 1 + \sum_{i=1}^{n-\lceil \frac{n}{h} \rceil} p(n, n-i) \ .$$

Avec le lemme 2, on obtient :

$$m_h(n) \leq 1 + \sum_{i=1}^{n-\lceil \frac{n}{h} \rceil} p(i) \ .$$

□

Nous sommes maintenant prêt à montrer que la proportion de partitions de $P(n)$ qui sont dans $M_h(n)$ (et donc dans $Q(n)$) tend vers 0 lorsque n tend vers l'infini.

Théorème 2 *Avec la notation précédente, nous avons :*

$$\lim_{n\to+\infty} \frac{q(n)}{p(n)} = 0 \,.$$

`Preuve.` Comme $q(n) \leq m_h(n)$, il suffit de montrer que $\lim_{n\to+\infty} \frac{m_h(n)}{p(n)} = 0$. Le lemme 3 prouve que $m_h(n) \leq 1 + \sum_{i=1}^{n-\lceil \frac{n}{h} \rceil} p(i)$ et comme $p(n)$ est strictement croissante, nous avons :

$$m_h(n) \leq 1 + \left(n - \left\lceil \frac{n}{h} \right\rceil\right) p\left(n - \left\lceil \frac{n}{h} \right\rceil\right)$$

$$\Leftrightarrow \frac{m_h(n)}{p(n)} \leq \frac{1}{p(n)} + \frac{(n - \lceil \frac{n}{h} \rceil)}{p(n)} p\left(n - \left\lceil \frac{n}{h} \right\rceil\right) \,.$$

Nous avons $\lim_{n\to+\infty} \frac{1}{p(n)} = 0$. Pour la seconde partie de l'équation, nous utilisons la formule asymptotique de Ramanujan [Baez-Duarte 1997] :

$$p(n) \underset{+\infty}{\sim} \frac{1}{4n\sqrt{3}} e^{\pi\sqrt{\frac{2n}{3}}}. \tag{3.3}$$

$$\frac{(n - \lceil \frac{n}{h} \rceil)}{p(n)} p\left(n - \left\lceil \frac{n}{h} \right\rceil\right) \sim \left(n - \left\lceil \frac{n}{h} \right\rceil\right) \frac{\frac{1}{4} \frac{e^{\pi\sqrt{2/3\left(n - \lceil \frac{n}{h} \rceil\right)}}}{(n - \lceil \frac{n}{h} \rceil)\sqrt{3}}}{\frac{1}{4} \frac{e^{\pi\sqrt{2/3n}}}{n\sqrt{3}}} = n e^{\pi\sqrt{2/3}\left(\sqrt{n - \lceil \frac{n}{h} \rceil} - \sqrt{n}\right)} \,.$$

Or $\lim_{n\to+\infty} n e^{\pi\sqrt{2/3}\left(\sqrt{n - \lceil \frac{n}{h} \rceil} - \sqrt{n}\right)} = 0$. Donc :

$$\lim_{n\to+\infty} \frac{m_h(n)}{p(n)} = 0 \,.$$

□

Dans la prochaine section (3.2.2), nous appliquons ces résultats pour comparer notre complexité avec celles de Liu et Song [Liu 2006] et Bourgeois et al., [Bourgeois 2010]. Dans les deux cas, notre algorithme a une meilleure complexité s'il y a une *bonne* partition en cliques ; et la proportion de partitions de n qui induisent une complexité plus grande tend vers 0 lorsque la taille de $\mathcal{G}$ tend vers l'infini.

Bien entendu, à partir d'un graphe $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ donné avec n sommets, il n'y a pas toujours $p(n)$ partitions de l'ensemble $\mathcal{V}$. En particulier quand $\mathcal{G}$ n'est pas dense, le nombre de partitions possibles de $\mathcal{V}$ est inférieur à $p(n)$. Cependant, tant que le nombre de partitions de $\mathcal{V}$ est proche de $p(n)$ la probabilité d'obtenir une *bonne* partition de $\mathcal{V}$ est grande.

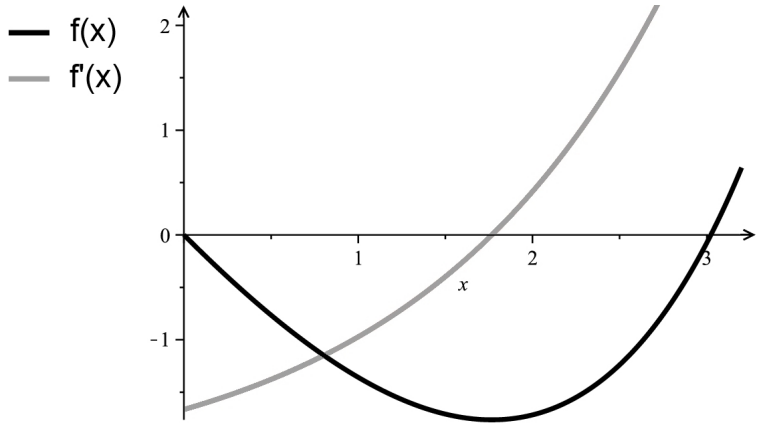

FIGURE 3.4 – Courbe représentatives de $f(x) = 2^x - \frac{1000}{424}x - 1$ et de sa dérivé

3.2.2 Comparaison

Dans cette section, nous appliquons cette méthode de comparaison avec deux approches récentes. Nous donnons un exemple d'application numérique pour trouver h, la valeur moyenne d'une partition, critère de référence de comparaison avec l'heuristique de partition en cliques.

Algorithme de branchement de Bourgeois, Escoffier et Paschos

Dans l'article [Bourgeois 2010], les auteurs développent un algorithme de branchement qui obtient un *mIDS* en $O^*(2^{0.424n})$. Nous souhaitons donc avoir un h tel que :

$$h \leq \frac{\sum_{i=1}^{k} c_i}{k} \quad \Rightarrow \quad \prod_{i=1}^{k} (c_i + 1) \leq 2^{0.424n} .$$

Tout d'abord, posons $f(x) = 2^x - \frac{1000}{424}x - 1$ et $f'(x) = 2^x \ln(2) - \frac{125}{52}$. La figure 3.4 représente la fonction f en noir et sa dérivé en gris. De manière triviale, $f'(x) > 0, \forall x \geq 2$ et $f(x) \geq 0, \forall x \geq 3.03$. Par conséquent, $f(x) \geq 0, \forall x \geq 3.03$.

En utilisant les résultats de la démonstration du lemme 1, nous déduisons $h = 7.2$ et nous avons :

$$2^{0.424n} \geq \prod_{i=1}^{k} (c_i + 1) .$$

Ainsi pour un graphe donné, si nous trouvons une partition en cliques avec une valeur moyenne supérieure à 7.2, notre algorithme aura une complexité exponentielle meilleure que $2^{0.424n}$. De plus le théorème 2 prouve que le nombre de partitions avec une valeur moyenne inférieure à 7.2 est très petit par rapport à $p(n)$ lorsque n est grand.

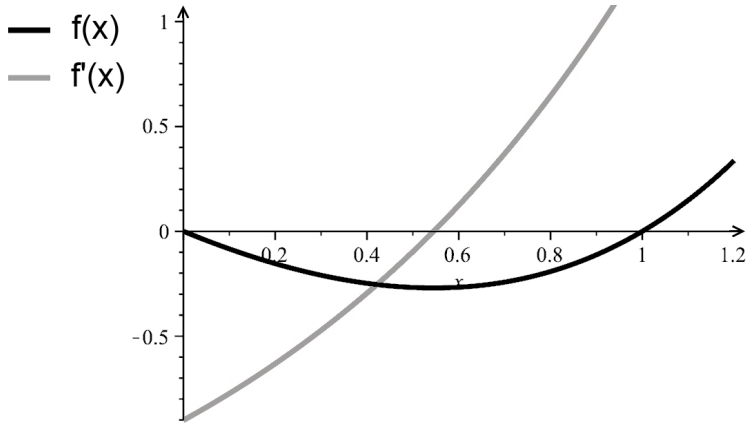

FIGURE 3.5 – Courbe représentatives de $f(x) = 3^x - 2x - 1$ et de sa dérivé

Algorithme de couplage de Liu et Song

Dans cet article, les auteurs proposent un algorithme en $O^*(\sqrt{3}^n)$ pour résoudre le *mIDS* dans les graphes généraux. Nous trouvons une valeur h telle que :

$$h \leq \frac{\sum_{i=1}^{k} c_i}{k} \Rightarrow \prod_{i=1}^{k} (c_i + 1) \leq \sqrt{3}^n \ .$$

Tout d'abord, posons $f(x) = 3^x - 2x - 1$ et $f'(x) = 3^x \ln(3) - 2$. La figure 3.5 représente la fonction f en noir et sa dérivé en gris. De manière triviale, $f'(x) > 0, \forall x \geq 1$ et $f(x) \geq 0, \forall x \geq 1$. Ainsi $f(x) \geq 0, \forall x \geq 1$.

En utilisant le résultat de la démonstration du lemme 1, nous déduisons que $h = 2$ et nous obtenons :

$$\sqrt{3}^n \geq \prod_{i=1}^{k} (c_i + 1) \ .$$

Ainsi, pour un graphe donné, si nous trouvons une partition en cliques avec une valeur moyenne supérieure à 2, notre algorithme aura une complexité exponentielle meilleure que $\sqrt{3}^n$: ce qui est cohérent dans la mesure où les couplages de [Liu 2006] sont des partitions en cliques de taille 2. Le théorème 2 prouve que le nombre de partitions avec une valeur moyenne inférieure à 2 est très petit par rapport à $p(n)$ dès lors que n est grand.

Les sections précédentes ont clarifié l'approche par partition en cliques et nous ont permis de présenter une méthode *atypique* de comparaison des complexités : une comparaison "conditionnelle". Nous avons ainsi noté que, par la propriété de l'indépendance de $\mathcal{S}$, quelle que soit la partition en cliques, $\mathcal{S}$ a *au plus* un sommet dans chacune des

cliques de la partition. Cette remarque nous permet d'élaborer une *borne inférieure* pour le problème du *mIDS*.

3.3 Borne inférieure

Dans cette section, nous présentons un algorithme polynomial pour obtenir une borne inférieure au *mIDS*. Tout d'abord, rappelons et donnons quelques **définitions** préliminaires. Soit $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ un graphe et $C = \{C_1, ..., C_k\}$ une partition en cliques de $\mathcal{G}$. On note $d(C_i) = max\{d(u), u \in C_i\}$ le *degré maximal* de C_i, *c'est-à-dire* le maximum des degrés des sommets de C_i dans $\mathcal{G}$. Soit α la valeur de la *borne inférieure* du *mIDS*. Pour toute solution $\mathcal{S}$ indépendante et dominante, on a $\alpha \leq |\mathcal{S}|$. Par la suite, et sans perte de généralité, supposons que $C = \{C_1, ..., C_k\}$ est triée dans l'ordre décroissant du degré maximal : $d(C_1) \geq d(C_2) \geq ... \geq d(C_k)$.

Maintenant, décrivons notre algorithme (que nous appelons LWB). LWB construit sa solution en ajoutant au fur et à mesure les sommets u_i ($u_i \in C_i, i = 1, .., k$ et $d(u_i) = d(C_i)$) dans l'ensemble $\mathcal{S}$ initialement vide. LWB s'arrête lorsque $\sum_{u \in \mathcal{S}} d(u) \geq |\mathcal{V}| - |\mathcal{S}|$. On note $\mathcal{S}_p = \{u_1, ..., u_p\}$, avec $p \leq k$ la solution retournée par notre algorithme LWB (sans garantir que $\mathcal{S}_p$ est un *IDS*). Le théorème 3 prouve que $|\mathcal{S}_p|$ est une borne inférieure du *mIDS* dans $\mathcal{G}$.

Propriété 1 *Avec les notations précédentes, nous disons qu'une solution $\mathcal{S} \subseteq \mathcal{V}$ vérifie la Propriété 1 si et seulement si :*

1. $\forall i \in \{1, ..., k\}, |\mathcal{S} \cap C_i| \leq 1$.
2. $\sum_{u \in \mathcal{S}} d(u) \geq |\mathcal{V}| - |\mathcal{S}|$.

Lemme 4 *Soit $\mathcal{S}$ un dominant de $\mathcal{G}$. $\mathcal{S}$ vérifie $\sum_{u \in \mathcal{S}} d(u) \geq |\mathcal{V}| - |\mathcal{S}|$.*

`Preuve.` Posons $\mathcal{E}_{out}(\mathcal{S})$ le nombre d'arêtes entre $\mathcal{S}$ et $\mathcal{V} \backslash \mathcal{S}$. Alors, $\mathcal{E}_{out}(\mathcal{S}) \leq \sum_{u \in \mathcal{S}} d(u)$.

Comme $\mathcal{S}$ est un dominant de $\mathcal{G}$, il y a au moins $|\mathcal{V}| - |\mathcal{S}|$ arêtes entre $\mathcal{S}$ et $\mathcal{V} \backslash \mathcal{S}$. Nous avons $|\mathcal{V}| - |\mathcal{S}| \leq \mathcal{E}_{out}(\mathcal{S})$ et $\mathcal{E}_{out}(\mathcal{S}) \leq \sum_{u \in \mathcal{S}} d(u)$ donc :

$$|\mathcal{V}| - |\mathcal{S}| \leq \sum_{u \in \mathcal{S}} d(u) \ .$$

□

Lemme 5 *$\mathcal{S}_p$ est le plus petit sous-ensemble de $\mathcal{V}$ qui vérifie la Propriété 1.*

`Preuve.` Par construction de $\mathcal{S}_p$, il y a au moins un sommet dans chaque C_i *c'est-à-dire* $|\mathcal{S}_p \cap C_i| \leq 1$. De plus, l'algorithme de construction s'arrête lorsque $\sum_{u \in \mathcal{S}_p} d(u) \geq |\mathcal{V}| - |\mathcal{S}_p|$. Ainsi, $\mathcal{S}_p$ vérifie la Propriété 1.

Nous avons $\sum_{1 \leq i \leq p} d(C_i) \geq |\mathcal{V}| - p$ et $\sum_{1 \leq i \leq m} d(C_i) < |\mathcal{V}| - m$, pour tout $m < p$.

Supposons qu'il existe un ensemble $\mathcal{S} = \{v_1, ..., v_m\}$ plus petit que $\mathcal{S}_p$ ($|\mathcal{S}| \leq |\mathcal{S}_p|$) et vérifiant la Propriété 1. On note C_{v_i} la clique de C qui contient v_i.

Alors, $\sum_{1 \leq i \leq m} d(C_{v_i}) \geq |\mathcal{V}| - m$ et :

$$\sum_{1 \leq i \leq m} d(C_{v_i}) > \sum_{1 \leq i \leq m} d(C_i) \ .$$

Cela est en contradiction avec le fait que C est triée dans l'ordre décroissant des degrés maximaux. Donc, $\mathcal{S}_p$ est le plus petit sous-ensemble de $\mathcal{V}$ qui vérifie la propriété 1. □

Théorème 3 *Soit $\mathcal{S}^*$ un* mIDS*, alors $|\mathcal{S}^*| \geq |\mathcal{S}_p|$.*

`Preuve.` $\mathcal{S}^*$ est un ensemble dominant donc par le lemme 4, il vérifie $\sum_{u \in \mathcal{S}^*} d(u) \geq |\mathcal{V}| - |\mathcal{S}^*|$. De plus, comme $\mathcal{S}^*$ est un indépendant nous avons $|\mathcal{S}^* \cap C_i| \leq 1, \forall i \in \{1, ..., k\}$. Ainsi, $\mathcal{S}^*$ vérifie la propriété 1. Mais le lemme 5 prouve que $\mathcal{S}_p$ est le plus petit sous-ensemble de $\mathcal{V}$ qui respecte la propriété 1. Dans ce cas $|\mathcal{S}^*| \geq |\mathcal{S}_p|$. □

Maintenant nous montrons que dans certains cas, la partition en cliques détermine la qualité de notre borne inférieure. Pour illustrer notre exemple, nous utilisons le graphe de la figure 3.6. Soit k un entier pair strictement positif. Le graphe a $2k+1$ sommets et se compose d'un chemin de k sommets à gauche du sommet 1 et d'une clique de $k+1$ sommets (incluant 1 lui-même) à sa droite.

Pour commencer, prenons la partition en cliques $C1 = \{C1_1, ..., C1_{k/2+1}\}$ triée dans l'ordre décroissant du degré maximal avec $|C1_1| = k+1$ ($C1_1$ la clique de $k+1$ sommets à droite) et $|C1_i| = 2, \forall i \in \{2, ..., k/2+1\}$ ($k/2$ cliques de taille 2 dans le chemin de k sommets). Nous avons $d(C1_1) = k+1$ et $d(C1_i) = 2, \forall i \in \{2, ..., k/2+1\}$. L'algorithme LWB construit $\mathcal{S}_p$ en commençant par le sommet de degré $k+1$ de la clique $C1_1$. On cherche à résoudre le système d'inéquations 3.4, où p est la taille de $\mathcal{S}_p$ à partir de laquelle $\mathcal{S}_p$ vérifie la propriété $\sum_{u \in \mathcal{S}} d(u) \geq |\mathcal{V}| - |\mathcal{S}|$.

$$\begin{cases} k+1+2(p-1) & \geq & 2k+1-p \\ k+1+2(p-2) & \leq & 2k+1-(p-1) \end{cases} \tag{3.4}$$

Ce qui donne $p \geq \frac{k+2}{3}$ et $p \leq \frac{k+5}{3}$, en d'autres termes $p = \left\lceil \frac{k+2}{3} \right\rceil$. Ainsi, LWB retourne seulement le sommet de la clique de $k+1$ sommets et $\left\lceil \frac{(k-1)}{3} \right\rceil$ sommets des autres cliques de taille 2.

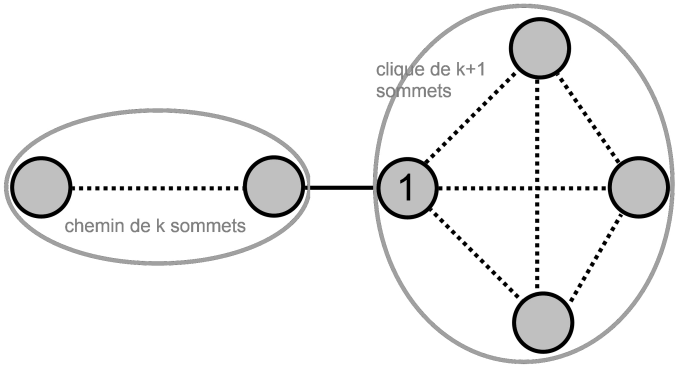

FIGURE 3.6 – Exemple dans lequel la borne inférieure dépend beaucoup de la partition en cliques

Ensuite considérons une seconde partition en cliques $C2$ dans laquelle chaque sommet est une clique (il y a $2k+1$ cliques de ce type). $C2$ est triée dans l'ordre décroissant du degré maximal. Cette fois, l'algorithme LWB retourne une solution contenant seulement deux sommets de la grande clique : $S_2 = \{u_1, u_2\}$ avec $d(u_1) = k+1$ et $d(u_2) = k$. Donc la borne inférieure du *mIDS* avec la partition en cliques $C2$ est seulement de taille 2.

La borne inférieure à partir de la partition en cliques $C1$ dépend de k alors que la borne inférieure de la seconde partition en cliques $C2$ est constante. Cet exemple montre clairement comment une partition en cliques peut influencer la qualité de notre borne inférieure, ce qui parait naturel, dans la mesure où elle se construit à partir de cette partition.

3.4 Algorithme exact pour le *mIDS*

Dans cette section, nous affinons notre méthode pour résoudre l'*indépendant dominant de taille minimum*. Nous rappelons tout d'abord notre algorithme de recherche exhaustive basé sur les partitions en cliques (développé dans la section 3.1). Puis, nous l'améliorons par l'ajout de différentes conditions dans la recherche pour réduire le nombre de solutions testées. Nous décrivons également l'heuristique gloutonne utilisée pour avoir la première solution *IDS*.

3.4.1 Recherche exhaustive

Avant de commencer, rappelons les **définitions** et les **notations** qui seront utilisées par la suite. Nous considérons un graphe $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ et une *partition en cliques* de $\mathcal{G}$, $C = \{C_1, \ldots, C_k\}$, *c'est-à-dire* une famille de k sous-ensembles disjoints des sommets

de $\mathcal{G}$ ($C_i \subseteq \mathcal{V}$ et si $i \neq j$, $C_i \cap C_j = \emptyset$), dont chacun induit une clique dans $\mathcal{G}$: le graphe qui se compose des sommets de C_i et des arêtes de $\mathcal{G}$ qui relient les sommets dans C_i est une clique.

Soit $\mathcal{S}$ un indépendant dominant (*IDS*) de $\mathcal{G}$. Comme $\mathcal{S}$ est un ensemble indépendant et que chaque C_i induit une clique, nous avons : $|C_i \cap \mathcal{S}| \leq 1$. L'idée de notre méthode exhaustive est de construire chaque sous-ensemble $\mathcal{S}$ de $\mathcal{V}$ ayant au plus un sommet dans chaque C_i (ce qui satisfait $|C_i \cap \mathcal{S}| \leq 1$, $\forall i \in \{1, \ldots, k\}$), puis de vérifier si c'est un *IDS* et à la fin de retourner celui de taille minimum. Nous sommes sûr de retourner la solution optimale car cet algorithme construit tous les *IDS*. Le nombre de candidats testés est ainsi $\prod_{i=1}^{k}(c_i + 1)$. Notons que la méthode peut être implémentée en utilisant un algorithme de branchement récursif, explorant les cliques de C une par une dans un ordre arbitraire : C_1, puis C_2, etc. jusqu'à C_k. A chaque étape i de la récursion, il y a $c_i + 1$ choix : ajouter un des c_i sommets de C_i ou n'en ajouter aucun dans la solution courante (la recherche examine toutes ces possibilités). Quand un *IDS* est trouvé, l'algorithme met à jour la *meilleure solution*.

Afin d'initialiser cette meilleure solution, nous avons besoin de trouver un *IDS* avant de lancer la recherche exhaustive. Pour des considérations de performance, nous avons choisi une heuristique gloutonne pour construire cette solution *initiale* (cette heuristique sera explicitée dans la section 3.4.3).

3.4.2 Algorithme exact

Dans la section 3.4.1, nous avons décrit notre méthode générale de recherche d'un *mIDS*. Nous allons maintenant améliorer les performances de cette méthode en ajoutant une méthode de *filtrage* comprenant plusieurs filtres dans le schéma de branchement récursif. Ces filtres nous évitent de tester des mauvais candidats et nous permettent d'ignorer les branches de l'arbre de recherche qui en contient. Cependant, l'ajout de filtres peut également avoir un effet néfaste. Cela ajoute du temps de calcul lié aux tests dont le nombre augmente de manière exponentielle en fonction de la taille du graphe. Ainsi il faut trouver des filtres suffisamment efficaces au regard du coût en temps de calcul. Il faut également prendre soin de combiner les bons filtres dans la mesure où certains rendent d'autres complètement inutiles. Pour illustrer notre approche, nous utilisons le graphe $\mathcal{G}_{Exemple} = (\mathcal{V}, \mathcal{E})$ de 13 sommets et de 21 arêtes de la figure 3.7.

Nous décrivons ci-dessous, les filtres que nous ajoutons à notre méthode générale.

Domination. Si lors d'une étape tous les sommets sont déjà dominés, nous n'avons pas besoin d'aller plus loin. Nous arrêtons alors l'exploration de cette branche de l'arbre. A titre d'exemple, supposons que nous avons la partition en cliques $C = \{\{1,2,3,4\},\{5,6,7\},\{8\},\{9,10,11\},\{12,13\}\}$ du graphe $\mathcal{G}_{Exemple}$ de la figure 3.7. Supposons que nous sommes à une étape où la solution courante $\mathcal{S}$ (en cours de construction) est $\{1,5,8,11\}$. La figure 3.8 représente graphiquement la situation

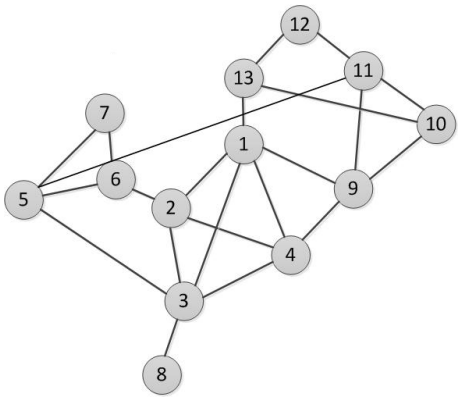

FIGURE 3.7 – Graphe $\mathcal{G}_{Exemple}$ pour illustrer la résolution exacte du *mIDS*

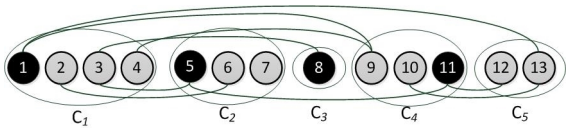

FIGURE 3.8 – Partition en cliques pour illustrer le critère de "Domination"

(les arêtes dans les cliques ne sont pas représentées). A cette étape, la solution $\mathcal{S}$ domine déjà tout le graphe (et est indépendant). Il est donc inutile d'aller plus loin dans la recherche.

Dominé uniquement par sa clique. Considérons un sommet v dans une clique C_i. Maintenant supposons que v a *tous* ses voisins dans C_i. Alors toutes les solutions $\mathcal{S}$ qui n'ont pas au moins un sommet dans C_i ne dominent pas v. Cela veut dire que dans notre recherche, nous pouvons ignorer les solutions qui ne contiennent pas de sommet dans C_i.

Dernière occasion pour dominer. Chaque sommet peut seulement être dominé par un nombre restreint d'autres sommets (ses voisins). Avec une partition en cliques donnée, chaque sommet ne peut être *dominé* que par un nombre limité de cliques (qui contiennent ses voisins). Alors, à partir d'une partition en cliques triées $\{C_1, C_2, \ldots, C_k\}$, pour chaque sommet, nous affectons (dans la phase de préparation des données) le numéro de la dernière clique qui peut le dominer. Nous utilisons cette indication pour détecter les sommets non dominés et qui ne peuvent plus l'être lors du parcours des cliques. Plus précisément, supposons que nous avons une solution $\mathcal{S}$ à une étape, considérons la clique C_i, et qu'il existe un sommet non dominé v dans la clique C_k avec $k < i$ et v n'a pas de voisins dans les cliques $\forall m \geq i$ C_m, alors v ne peut plus être dominé et il est inutile de continuer à construire la solution courante.

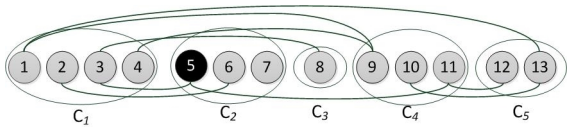

FIGURE 3.9 – Partition en cliques pour illustrer le critère de la "dernière occasion pour dominer"

Par exemple, à partir de la partition en cliques C précédente, supposons que l'on considère la clique C_3 avec $\mathcal{S} = \{5\}$, alors quoi que l'on fasse aux cliques C_3, C_4 et C_5, la solution $\mathcal{S}$ ne sera pas une solution dominante à cause du sommet 2 (voir figure 3.9).

Taille minimum. Ce filtre est un critère classique des problèmes d'optimisation. Dans chaque étape, nous comparons la taille de la solution courante avec la taille de la meilleure solution rencontrée. Nous changeons de branche de recherche si la solution courante est supérieure à la meilleure rencontrée jusqu'à présent par l'algorithme.

Notons que les trois premiers filtres présentés réduisent le nombre de solutions *candidates* testées. Avec eux, nous avons un algorithme d'énumération du *IDS* et du *mIDS*. Seul le dernier filtre restreint les solutions au *mIDS*.

3.4.3 Heuristique de construction d'une solution

Comme nous l'avons dit précédemment, cette méthode a besoin d'un *IDS* initial qui sert de première meilleure solution (utile pour le dernier filtre). Pour ce faire, nous avons codé un algorithme glouton (GREEDY HEURISTIC : GH). À chaque étape de GH, nous ajoutons dans la solution courante le sommet non dominé qui a le plus grand nombre de voisins non dominés. Cette procédure est répétée jusqu'à ce que tous les sommets soient dominés. Il est aisé de comprendre que cet algorithme assure l'obtention d'un *IDS* de $\mathcal{G}$.

Pour illustrer GH utilisons le graphe $\mathcal{G}_{Exemple}$ à la figure 3.7. Soit $NDL = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13\}$ la liste des sommets non dominés. GH s'exécute de la manière suivante.

Étape 1 : GH sélectionne le sommet 1. Il l'ajoute à la solution courante $\mathcal{S}$ (qui est encore vide et $\mathcal{S} = \{1\}$). Les sommets $\{1, 2, 3, 4, 9, 13\}$ sont dominés et $NDL = \{5, 6, 7, 8, 10, 11, 12\}$.

Étape 2 : le sommet 5 est ajouté à $\mathcal{S}$ et $\mathcal{S} = \{1, 5\}$. Les sommets $\{5, 6, 7, 11\}$ sont dominés et $NDL = \{8, 10, 12\}$.

Étape 3 : le sommet 8 est ajouté à $\mathcal{S}$ et $\mathcal{S} = \{1, 5, 8\}$. Le sommet 8 est dominé et $NDL = \{10, 12\}$.

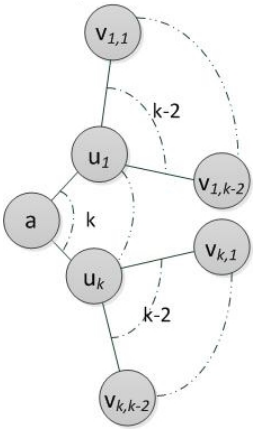

FIGURE 3.10 – Special Star Graphs ($k \in \mathbb{N}$)

Étape 4 : le sommet 10 est ajouté à $\mathcal{S}$ et $\mathcal{S} = \{1, 5, 8, 10\}$. Le sommet 10 est dominé et $NDL = \{12\}$.

Étape 5 : le sommet 12 est ajouté à $\mathcal{S}$ et $\mathcal{S} = \{1, 5, 8, 10, 12\}$. Le sommet 12 est dominé et tous les sommets du graphe sont dominés. La solution est $\mathcal{S} = \{1, 5, 8, 10, 12\}$.

Graphes particuliers qui piègent l'heuristique

Nous montrerons à travers nos expérimentations (section 3.5) que GH construit des solutions de très bonne qualité dans le cas général. Mais ici nous présentons des graphes particuliers qui piègent l'algorithme. La solution trouvée par GH dans ces graphes est alors très loin de l'optimale.

Graphe Special Star. Soit k un entier, et $\mathcal{G}_{SpecialStar} = (\mathcal{V}, \mathcal{E})$ le graphe qui contient $|\mathcal{V}| = k^2 - k + 1$ sommets (voir la figure 3.10). Le graphe se construit comme suit : un premier sommet a avec ses k voisins $u_1, \ldots, u_k$. Chaque u_i $(i = 1, \ldots, k)$ a $k-1$ voisins : a et $v_{i,1}, \ldots, v_{i,k-2}$.

Il est relativement aisé de comprendre que GH construit une solution avec $1+k(k-2)$ sommets (sélectionnant d'abord a puis les sommets restant non encore dominés $v_{i,j}$), alors que la solution optimale a seulement k sommets $(u_1, \ldots, u_k)$. Les figures 3.11 et 3.12 donnent une telle solution lorsque $k = 4$ (la pire solution est de taille 9 alors que la meilleure est de taille 4 seulement).

Graphe Special Two Subsets. Soit k un entier et $\mathcal{G}_{STS} = (\mathcal{V}, \mathcal{E})$ le graphe avec $2k+1$ sommets (voir la figure 3.13) construit comme suit. Le premier sommet a a $k+1$ voisins : b, c et $v_1, \ldots, v_{k-1}$. Le sommet b a k voisins : a et $v_1, \ldots, v_{k-1}$. Le sommet c a k voisins : a et $u_1, \ldots, u_{k-1}$.

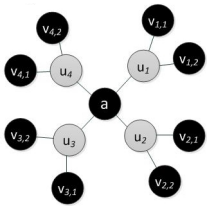

FIGURE 3.11 – Pire solution pour Special Star Graph : $k = 4$

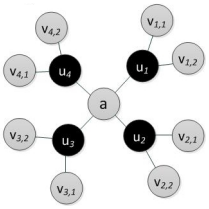

FIGURE 3.12 – Solution optimale pour le Special Star Graph : $k = 4$

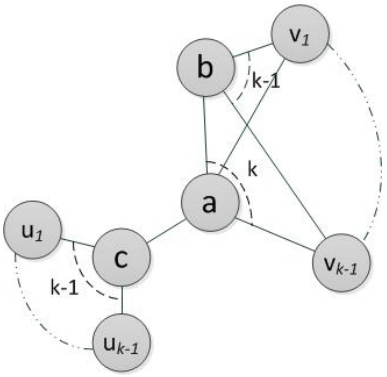

FIGURE 3.13 – Special Two Subsets Graphs ($k \in \mathbb{N}$)

GH retourne une solution avec k sommets (a et $u_1, \ldots, u_{k-1}$) alors que la solution optimale a seulement 2 sommets (b et c). Les figures 3.14 et 3.15 illustrent l'exemple avec $k = 4$ (la pire solution est de taille 4 alors que l'optimal est de taille 2).

3.4.4 Méthode de construction d'une partition en cliques

Notre méthode de résolution du *mIDS* a besoin initialement d'une partition en cliques de $\mathcal{G}$. Nous utilisons une méthode gloutonne de manière stochastique (STOCHASTIC GREEDY APPROACH : SGA) pour construire la partition.

Appelons *sommet libre*, un sommet qui n'est dans aucune clique C_i. Au début de SGA, tous les sommets sont *libres* et SGA s'arête lorsqu'il n'y a plus de sommet libre. Lorsque ce n'est pas précisé, l'algorithme choisit un sommet (parmi un groupe de sommets de mêmes caractéristiques) de façon *équiprobable*. A chaque étape SGA crée une clique C_t avec un sommet libre v (tiré au hasard et de manière équiprobable donc) et ajoute l'un de ses voisins libres dans C_t (si c'est possible). Dans la clique courante C_t, SGA ajoute un sommet libre voisin de tous les sommets de la clique (si un tel sommet existe). SGA construit de manière itérative cette clique C_t jusqu'à ce qu'il n'y ait plus de sommet libre voisin de tous les sommets de la clique. Il crée alors une nouvelle clique

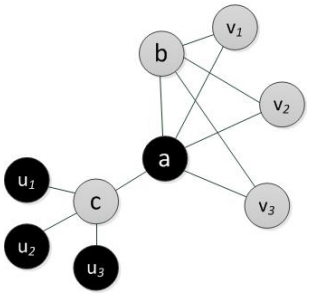

FIGURE 3.14 – Solution retournée par GH pour le graphe Special Two Subsets : $k = 4$

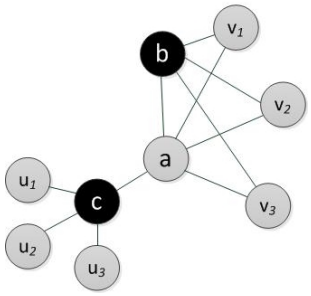

FIGURE 3.15 – Solution optimale pour le graphe Special Two Subsets : $k = 4$

C_{t+1} avec un sommet libre (s'il en reste encore) et continue la construction.

Utilisons notre précédent graphe $\mathcal{G}_{Exemple}$ (figure 3.16) pour illustrer SGA. Appelons LFV, la liste des sommets libres. Au départ de SGA, nous avons : $LFV = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13\}$.

Étape 1 : SGA choisit de façon équiprobable le sommet 1 (parmi les autres sommets) et construit la clique C_1. Il ajoute le sommet 2, l'un des voisins de 1 : $C_1 = \{1, 2\}$. Le sommet 4, l'un des voisins communs à 1 et 2, est choisi aléatoirement et est ajouté : $C_1 = \{1, 2, 4\}$. Le sommet 3, le seul voisin commun à tous les sommets de C_1, est ajouté à la clique : $C_1 = \{1, 2, 3, 4\}$. Il n'y a plus de sommet dans LFV voisin des sommets 1, 2, 3 et 4. Dans ce cas l'algorithme arrête la construction de C_1, et on a $C_1 = \{1, 2, 3, 4\}$ et $LFV = \{5, 6, 7, 8, 9, 10, 11, 12, 13\}$.

Étape 2 : SGA choisit de manière aléatoire le sommet 6 pour construire C_2. Le sommet 5, qui est l'un de ses voisins, est ajouté : $C_2 = \{5, 6\}$. Puis le sommet 7, le seul voisin commun au sommets 5 et 6, est choisi : $C_2 = \{5, 6, 7\}$. Il n'y a plus de voisin commun à tous les sommets de la clique dans LFV et nous avons $C_2 = \{5, 6, 7\}$ et $LFV = \{8, 9, 10, 11, 12, 13\}$.

Étape 3 : On sélectionne le sommet 8 pour construire la clique C_3. Il n'y a pas de voisin de 8 dans LFV. Ainsi, nous avons $C_3 = \{8\}$ et $LFV = \{9, 10, 11, 12, 13\}$.

Étape 4 : On sélectionne de manière aléatoire le sommet 11 pour créer la clique C_4. On ajoute le sommet 10 : $C_4 = \{10, 11\}$. On ajoute ensuite le sommet 9, le seul voisin commun aux sommets 10 et 11 : $C_4 = \{9, 10, 11\}$. Il n'y a plus de voisin commun à tous les sommets de la clique. Donc nous avons $C_4 = \{9, 10, 11\}$ et $LFV = \{12, 13\}$.

Étape 5 : SGA choisit le sommet 12 pour initialiser C_5. Il ajoute le sommet 13 : $C_5 = \{12, 13\}$. La construction est terminée car il n'y a plus de sommet libre.

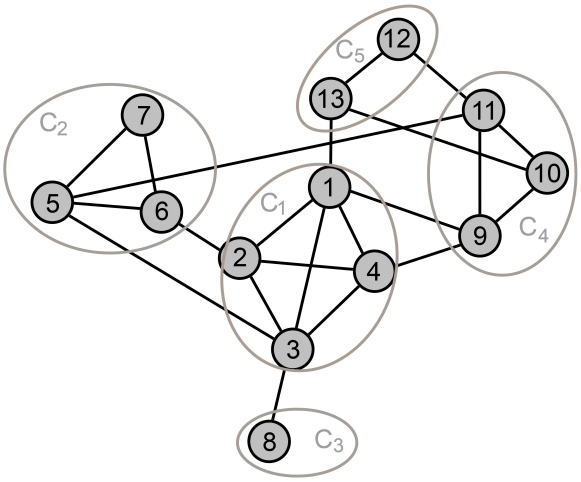

FIGURE 3.16 – Exemple d'une partition en cliques de $\mathcal{G}_{Exemple}$

La figure 3.16 illustre cette partition en cliques. Notons que SGA évite la construction de la partition en cliques *triviales* dans laquelle toutes les cliques ont *un* seul sommet. Il a été démontré dans la section 3.1 qu'une telle partition maximise la complexité de l'approche par partition en cliques.

3.5 Évaluation expérimentale

Dans ce qui suit, nous présentons les résultats expérimentaux de la méthode décrite dans la section 3.4.2 sur différents graphes. Le programme est codé en C. Il a résolu des centaines de graphes de familles différentes, présentées ci-dessous.

Les instances ont été générées avec le logiciel Maple et importées dans notre programme en C. Chaque instance a été exécutée 10 fois sur des processeurs dédiés AMD Opteron 2352 cadencés à 2.1GHz.

3.5.1 Test de huit familles de graphes allant jusqu'à 600 sommets

Ci-dessous les tableaux présentant les résultats des tests. Le nombre de sommets est noté n. Chaque ligne contient :

- le nombre d'exécutions valides $NbExec$ (parfois ce nombre est inférieur à 10 car certaines exécutions ont dépassé le temps maximal autorisé : 6 heures) ;
- la durée d'exécution de l'heuristique gloutonne (GH) pour avoir la première solution $Greedy\_time$ (durée moyenne en secondes) ;
- la taille de la solution trouvée par GH $Greedy\_size$, la durée d'exécution de notre algorithme exact Opt_time (durée moyenne en secondes) ;

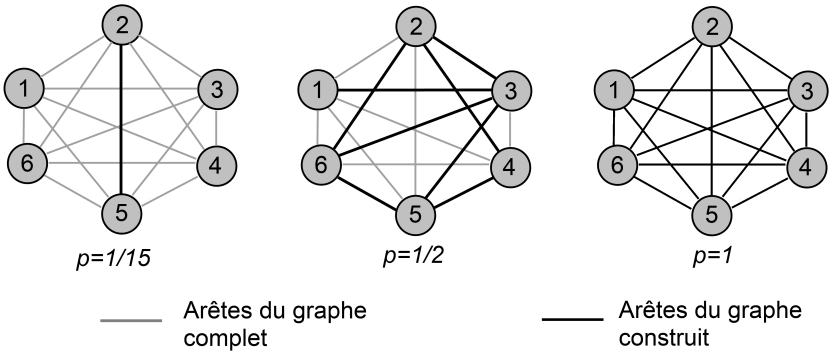

FIGURE 3.17 – Trois graphes construits suivant le modèle des graphes aléatoires

TABLE 3.1 – Résultats de la résolution exacte par *mIDS* des graphes aléatoires

	n=80	*n=90*	*n=110*
NbExec	10	10	5
Greedy_time	0.00	0.00	0.00
Greedy_size	16	14	15
Opt_time	**740.20**	**8049.5**	**126985.00**
Opt_size	10	11	12
NbUpdate	5.00	3.00	3.00

– la taille de la solution optimale *Opt_size*.

Afin de mieux évaluer l'amélioration qu'apporte notre algorithme exact par rapport au GH, nous avons compté le nombre de fois que l'algorithme exact trouve une solution strictement meilleure que la meilleure solution courante (initialisée à la solution de GH) ; la valeur moyenne de cette mesure est *NbUpdate*.

Graphes aléatoires

Nous utilisons le modèle binomial du graphe aléatoire popularisé par Erdős et Rényi [Erdős 1959]. Soit n un entier et p une probabilité, on construit un graphe non orienté et non pondéré de n sommets tel que chaque arête ait une probabilité d'existence p. La figure 3.17 illustre trois graphes de $n = 6$ sommets construits suivant ce modèle, avec respectivement $p = 1/15$, $p = 1/2$ et $p = 1$. Les arêtes en noir sont celles du graphe et les arêtes en gris sont celles du graphe complet.

Le tableau 3.1 rapporte les résultats pour $p = 0.1$. La durée d'exécution augmente très vite passant de 740 secondes pour 80 sommets à plus d'une journée pour 110 sommets.

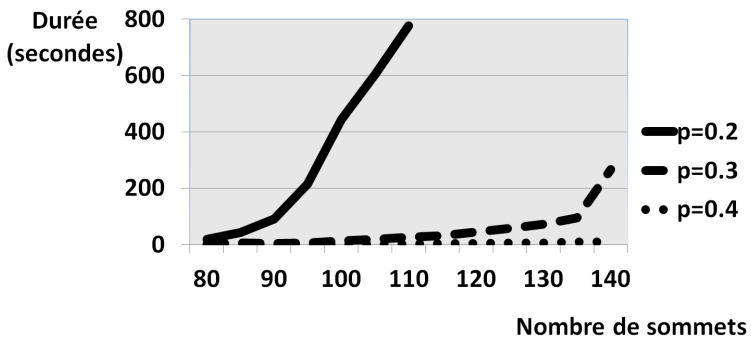

FIGURE 3.18 – Évolution de la durée de résolution de différents graphes de probabilités $p = 0.2$, $p = 0.3$ et $p = 0.4$

Le graphique de la figure 3.18 résume une partie des résultats expérimentaux : sur l'axe des abscisses se trouve le nombre de sommets et sur l'axe des ordonnées la durée de résolution. Par souci de clarté, seules les probabilités 0.2, 0.3 et 0.4 sont présentées. Nous pouvons remarquer que notre algorithme est capable de résoudre de grands graphes dans des durées raisonnables lorsque p est plus grand que 0.3.

Graphes : arbres aléatoires

Suivant les résultats précédents, nous remarquons que notre algorithme est plus lent lorsque la probabilité p est petite (*c'est-à-dire* lorsque $\mathcal{G}$ est peu dense). C'est pourquoi nous avons souhaité tester notre algorithme sur des *arbres*. Un arbre est un graphe non orienté et non pondéré avec n sommets et $n - 1$ arêtes (graphe très peu dense).

Le tableau 3.2 reporte les résultats pour des arbres de tailles différentes. Pour 50 sommets, la durée d'exécution est inférieure à 3 secondes. L'algorithme est capable de résoudre des instances allant jusqu'à 75 sommets en moins de 15 minutes. Nous pouvons aussi remarquer que GH semble être très performant ici. En effet il trouve des solutions dont les tailles sont très proches de l'optimale et ce de manière quasi-instantanée.

Graphe : chemin

Comme nous l'avons vu précédemment, notre méthode est moins performante sur les graphes peu denses. Il est alors intéressant de l'appliquer sur les *chemins* qui sont aussi des arbres avec des sommets de degrés au plus 2.

Le tableau 3.3 reporte ces résultats. La durée d'exécution augmente très rapidement. En comparaison avec les arbres, la méthode semble ici moins efficace : pour le même nombre de sommets (75) la durée d'exécution est d'un peu plus de 14 minutes en moyenne pour les arbres aléatoires et de près de 2 heures pour les chemins. Malgré cela, GH reste très performant ici, puisque la solution optimale est trouvée de manière

TABLE 3.2 – Résultats de la résolution exacte par *mIDS* des arbres aléatoires

	n=50	*n=55*	*n=60*	*n=65*	*n=70*	*n=75*
NbExec	10	10	10	10	10	10
Greedy_time	0.00	0.00	0.00	0.00	0.00	0.00
Greedy_size	18	20	23	26	27	30
Opt_time	2.90	5.30	41.90	37.20	210.70	881.30
Opt_size	17	20	23	24	26	26
NbUpdate	1.00	0.00	0.00	2.00	1.00	4.00

TABLE 3.3 – Résultats de la résolution exacte par *mIDS* des chemins

	n=50	*n=55*	*n=60*	*n=65*	*n=70*	*n=75*
NbExec	10	10	10	9	8	1
Greedy_time	0.00	0.00	0.00	0.00	0.00	0.00
Greedy_size	17	19	20	22	24	25
Opt_time	18.30	71.30	303.10	2515.60	6903.5	7010.00
Opt_size	17	19	20	22	24	25
NbUpdate	0.00	0.00	0.00	0.00	0.00	0.00

instantanée. Toute la durée d'exécution de l'algorithme ne sert qu'à s'assurer que c'est bien une solution optimale.

Graphes : hypercubes

Nous avons vu que notre algorithme est efficace sur les graphes denses. Nous présentons dans cette partie les résultats sur des graphes appelés *hypercubes*. Pour un entier d donné, un hypercube est un graphe $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ avec $|\mathcal{V}| = 2^d$ sommets et $|\mathcal{E}| = 2^{d-1}d$ arêtes ([Harary 1993]). Chaque sommet est étiqueté par un vecteur de d composantes binaires $\{0, 1\}$ et deux sommets sont connectés *si et seulement si* leurs vecteurs se différencient par une seule composante. La figure 3.19 illustre la construction des 5 premiers hypercubes de taille respectivement 2^0, 2^1, 2^2, 2^3 et 2^4 sommets. Chaque sommet a un numéro (de 1 à 16) et un label binaire (chiffres verticaux). Comme on peut le remarquer, la construction d'un hypercube de taille 2^d se fait à partir de l'hypercube de taille 2^{d-1} : les nouveaux sommets et arêtes de l'hypercube de taille 2^d sont grisés sur la figure.

Les hypercubes sont des graphes relativement denses et tous les sommets ont exactement le même degré d (pas d'effets aléatoires liés à la structure du graphe). En dépit de cette densité, la taille maximale des cliques est seulement de 2. Ainsi, notre méthode ne bénéficie pas du gain de complexité grâce à la partition en cliques.

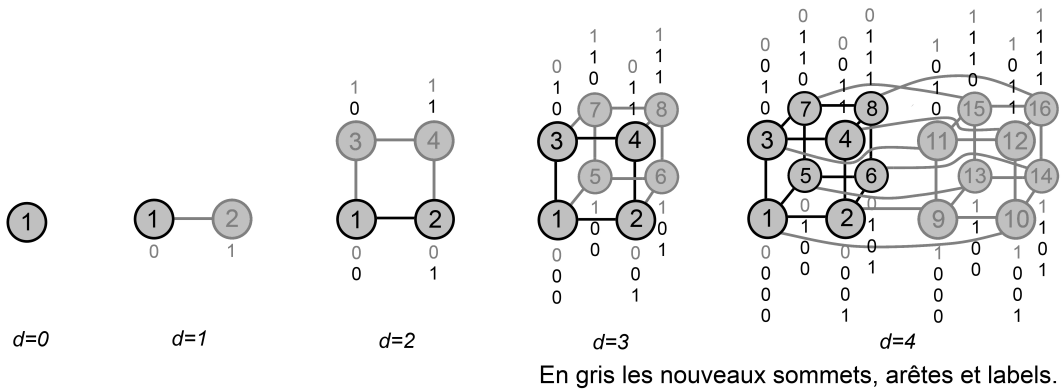

FIGURE 3.19 – Cinq hypercubes de 2^d sommets, $d = 0, \ldots, 4$

TABLE 3.4 – Résultats de la résolution exacte par *mIDS* des hypercubes

	n=4	*n=8*	*n=16*	*n=32*	*n=64*
NbExec	10	10	10	10	10
Greedy_time	0.00	0.00	0.00	0.00	0.00
Greedy_size	2	2	4	8	16
Opt_time	0.00	0.00	0.00	0.00	423.30
Opt_size	2	2	4	8	12
NbUpdate	0.00	0.00	0.00	0.00	2.60

Le tableau 3.4 rapporte les résultats pour des hypercubes allant de 4 à 64 sommets. Cela montre que l'algorithme est capable de résoudre de grands hypercubes en des temps raisonnables (environ 7 minutes). Pour des graphes plus petits, la solution est trouvée de manière instantanée.

Graphes : grilles

Dans le but de comparer notre méthode avec les résultats expérimentaux de Potluri et Negi [Potluri 2011], nous testons notre algorithme sur les grilles de taille allant de 5×5 jusqu'à 8×8. Une grille $n \times m$ est un graphe avec n lignes et m colonnes où chaque intersection est un sommet et chaque sommet est lié aux sommets qui lui sont directement voisins, horizontalement ou verticalement. La figure 3.20 montre trois exemples de graphes de type grille de tailles respectivement 1, 6 et 12 sommets.

Le tableau 3.5 résume cette comparaison avec l'algorithme de Liu et Song [Liu 2006] et l'*Intelligent Enumeration Algorithm* (IEA) de Potluri et Negi [Potluri 2011]. Les durées sont données en secondes. Notons que Potluri et Negi exécutent leur programme

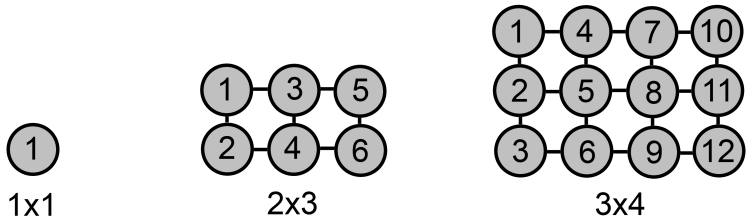

FIGURE 3.20 – Trois grilles

TABLE 3.5 – Comparaisons de l'efficacité de trois algorithmes sur les grilles

	n=25	*n=36*	*n=49*	*n=56*
NbExec	10	10	10	10
Greedy_time	0.00	0.00	0.00	0.00
Greedy_size	8	13	17	20
Opt_time	**0.00**	**0.00**	**9.90**	**630.00**
Opt_size	7	10	12	16
NbUpdate	1.00	2.70	4.10	3.60
IEA	**1.00**	**254.00**	**141242.00**	—
Liu et Song Algorithm	**11.00**	**39225.00**	—	—

sur des processeurs Intel(R) Xeon(TM) cadencés à 3.00GHz, avec 8GB RAM, ce qui est bien plus puissant que notre machine.

Rappelons que nos résultats sont des valeurs moyennes après 10 exécutions. Clairement, IEA est plus rapide que l'approche par couplage de Liu et Song car il n'entre pas dans l'énumération de tous les candidats. Il est également évident que l'approche par partition en cliques avec filtrage est meilleure que les deux autres. Là où IEA teste chacune de ses solutions pour voir si c'est un *IDS* ou non, notre algorithme ignore un grand nombre de solution *IDS*. Ainsi, pour 49 sommets, l'approche par partition en cliques ne prend *que* 9.90 secondes, alors que IEA prend plus d'une journée...

Graphes : Special Star

Les graphes *Special Star* sont une famille de graphes construite dans le but de piéger l'algorithme GH (voir la section 3.4.3). Le tableau 3.6 présente les résultats des tests. GH donne, bien sûr, de très mauvais résultats, mais le reste de l'algorithme corrige rapidement cette erreur. La méthode exacte résout ainsi jusqu'à 381 sommets. Le dernier cas, avec 601 sommets est résolu en moins de 2 heures et 30 minutes.

TABLE 3.6 – Résultats de la résolution exacte par *mIDS* des Special Star Graphs

	n=21	*n=91*	*n=211*	*n=381*	*n=601*
NbExec	10	10	10	10	10
Greedy_time	0.00	0.00	0.00	0.00	0.00
Greedy_size	16	81	196	361	576
Opt_time	0.00	0.00	3.70	227.00	8485.10
Opt_size	5	10	15	20	25
NbUpdate	4.80	9.90	14.30	19.60	24.70

Graphes : Special Two Subsets

Les graphes Special Two Subsets forment une autre famille dans laquelle l'heuristique gloutonne est très peu performante (voir la section 3.4.3). Cependant, notre approche par partition en cliques est très efficace pour améliorer cette solution initiale. Elle résout le *mIDS* de manière quasi instantanée, et ce jusqu'à plus de 900 sommets. Cela peut s'expliquer par le fait que la taille de la solution optimale étant très petite (2 sommets), le filtre sur la taille des candidats (voir la section 3.4.2) permet d'éviter de tester de très nombreuses solutions.

3.5.2 Analyse des résultats

A travers ces tests, nous pouvons noter la pertinence du choix des filtres qui permettent de résoudre le *mIDS* dans des instances allant jusqu'à 100 sommets, avec des durées très courtes. Les tests ont été étendus sur des graphes plus grands, jusqu'à plusieurs centaines de sommets : la résolution se fait en quelques heures. Notons que l'algorithme est bien moins efficace sur les graphes peu denses, bien que dans les *grilles* qui sont considérées comme très peu denses, *comparativement* à d'autres approches, nous sommes bien meilleurs.

Pour conclure ces expériences, nous notons que le GREEDY HEURISTIC (GH) est très rapide (moins d'une seconde dans tous nos tests) et très performant (souvent proche de l'optimal) à l'exception de quelques graphes particuliers comme les deux familles (Special Star et Special Two Subsets) construites pour l'occasion. Ainsi nous avons souhaité tester l'algorithme glouton sur des instances de plusieurs milliers de sommets. Étant donné qu'il est difficile d'obtenir des solutions exactes pour des problèmes de cette taille, GH est comparé à la borne inférieure de l'algorithme présentée à la section 3.3.

3.6 Évaluation de l'heuristique GH et de la borne inférieure

Dans cette section, nous résumons les résultats des tests du Greedy Heuristic (GH) et de l'algorithme de la borne inférieure (LWB). Le programme en C a résolu de grands graphes, allant de 200 sommets à plus de 60000 sommets. Comme dans la section 3.5 chaque instance est exécutée 10 fois sur des processeurs dédiés AMD Opteron 2352 cadencés à 2.1 GHz.

3.6.1 Test de quatre familles de graphes allant jusqu'à 60 000 sommets

Dans les tableaux qui suivent, nous faisons une synthèse des résultats de nos expériences. Le nombre de sommets du graphe est noté n. Chaque ligne contient :

- la taille de la solution construite par GH ($|Greedy|$) ;
- la borne inférieure avec la partition en cliques dégénérées ($|LWBV|$) ;
- la *taille moyenne* de la borne inférieure avec des partitions en cliques aléatoires ($|LWBC|$) ;
- son *écart type* (*Std. Dev*).

Rappelons que la *taille moyenne* et l'*écart-type* sont calculés après 10 exécutions de chaque instance. Ainsi, nous obtenons 10 solutions différentes et 10 partitions différentes.

Graphes aléatoires

Dans la section 3.5, nous avons donné la définition classique du graphe aléatoire que nous allons réutiliser ici. Comme cela a été énoncé auparavant, GH semble très performant lors de nos tests. Dans cette seconde expérience, l'algorithme a construit un *IDS* dans des graphes de 10000 sommets en moins d'une seconde (lorsque $p = 0.9$).

Le tableau 3.7 résume ces tests. Comme attendu, la durée d'exécution augmente lorsque la probabilité p diminue. Les bornes inférieures $|LWBV|$ et $|LWBC|$ sont souvent égales (plus de 90% des cas dans nos tests) et l'écart-type est très petit, atteignant à peine 0.46 pour la probabilité $p = 0.001$. Les tableaux 3.8 et 3.9 rapportent les résultats pour les graphes aléatoires de probabilité $p = 0.001$ pour des tailles allant de 500 sommets à 60000 sommets. GH reste efficace lorsqu'on le compare aux bornes inférieures. Nous pouvons aussi remarquer que pour les petits graphes peu denses, la version du LWB avec les partitions en cliques est nettement plus performante que la version avec la partition en cliques dégénérées. Cette différence est marginale lorsque n est grand.

Graphes : arbres aléatoires

Suivant les résultats précédents, l'écart type est petit lorsque le graphe est peu dense. Nous testons donc l'algorithme GH sur des arbres.

Le tableau 3.10 regroupe les résultats pour des arbres allant de 10000 à 40000 sommets. Pour chaque exécution, la durée est inférieure à 20 secondes. Nous pouvons

TABLE 3.7 – Résultats de l'heuristique GH sur des graphes aléatoires de 10000 sommets

Prob	0.001	0.005	0.01	0.05	0.1	0.5
$\|Greedy\|$	1774	559	338	93	47	10
$\|LWBV\|$	558	144	77	18	10	2
$\|LWBC\|$	558.7	144	77	18	10	2
Std. Dev.	0.46	0	0	0	0	0

TABLE 3.8 – Résultats de l'heuristique GH sur des graphes aléatoires de probabilités 0.001

$n =$	5.10^2	10^3	2.10^3	4.10^3	6.10^3	8.10^3
$\|Greedy\|$	383	626	942	1239	1499	1636
$\|LWBV\|$	238	307	385	459	502	541
$\|LWBC\|$	326.8	338.2	391.5	460.6	503	542.3
Std. Dev.	0.6	2.89	1.02	1.02	0	0.46

TABLE 3.9 – Résultats de l'heuristique GH sur des graphes aléatoires de probabilités 0.001

$n =$	10.10^3	20.10^3	30.10^3	40.10^3	50.10^3	60.10^3
$\|Greedy\|$	1774	2214	2443	2659	2811	2955
$\|LWBV\|$	558	631	670	698	716	732
$\|LWBC\|$	558.7	631	671	698	716	732
Std. Dev.	0.46	0	0	0	0	0

TABLE 3.10 – Résultats de l'heuristique GH sur des arbres aléatoires

$n =$	10.10^3	20.10^3	30.10^3	40.10^3
$\|Greedy\|$	4065	8078	12153	16156
$\|LWBV\|$	2164	4314	6467	8614
$\|LWBC\|$	2175	4337.6	6500.8	8657
Std. Dev.	1.18	2.42	2.6	2.39

TABLE 3.11 – Résultats de l'heuristique GH sur des grilles

$n =$	10.10^3	20.10^3	30.10^3	40.10^3
Dim.	100×100	100×200	100×300	100×400
$\|Greedy\|$	3303	6604	9954	13303
$\|LWBV\|$	2000	4000	6000	8000
$\|LWBC\|$	2000	4000	6000	8000
Std. Dev.	0	0	0	0

remarquer que GH donne des solutions relativement proches (facteur 2) des bornes inférieures.

Graphes : grilles

Les tableaux 3.11 et 3.12 reportent les résultats pour les grilles. La durée d'exécution est toujours inférieure à 20 secondes. Comme pour les arbres aléatoires, GH donne de bons résultats. Remarquons que quelle que soit la version de LWB, le résultat est exactement le même, puisque les grilles sont des graphes presque réguliers.

TABLE 3.12 – Résultats de l'heuristique GH sur des grilles

$n =$	10.10^3	20.10^3	30.10^3	40.10^3
Dim.	10×1000	10×2000	10×3000	10×4000
$\|Greedy\|$	3333	6666	10000	13333
$\|LWBV\|$	2000	4000	6000	8000
$\|LWBC\|$	2000	4000	6000	8000
Std. Dev.	0	0	0	0

TABLE 3.13 – Résultats de l'heuristique GH sur des hypercubes

$n =$	8	16	32	64	128	256	32768	65536
$\|Greedy\|$	2	4	8	16	16	32	2048	4096
$\|Opt.\|$	2	4	8	12	16	32	2048	4096
$\|LWBV\|$	2	4	6	10	16	29	2048	3856
$\|LWBC\|$	2	4	6	10	16	29	2048	3856
Std. Dev.	0	0	0	0	0	0	0	0

Graphes : hypercubes

Comme nous l'avons remarqué précédemment, GH semble très efficace dans les graphes peu denses et réguliers. Ainsi il est intéressant de le tester sur des hypercubes (voir la section 3.5.1 pour une définition). En 1993, Harary et Livingston [Harary 1993] démontrent une formule qui permet de calculer la taille du *mIDS* pour un groupe restreint d'hypercubes. Prenons un sous-ensemble d'hypercubes de dimension $d > 6$ (avec le nombre de sommets $n = 2^d$) telle que $d = 2^k$ ou $d = 2^k - 1$ (k un entier). Dans ce cas $OPT = 2^{d-k}$. Ainsi sur de grands hypercubes, nous pouvons mesurer la qualité du GH et des bornes inférieures par rapport à l'optimale.

Le tableau 3.13 rapporte les résultats pour des hypercubes allant de 8 à 65536 sommets. Cela montre ainsi que GH et LWB sont très efficaces. En effet, GH trouve 7 fois sur 8 les solutions optimales et LWB donne 5 fois sur 8 une borne égale à l'optimale. La durée d'exécution n'excède pas 11 secondes pour des graphes ayant plus de 65000 sommets.

3.6.2 Analyse des résultats

Avec ces tests nous avons confirmé que le GREEDY HEURISTIC (GH) est très rapide : tous nos tests se sont déroulés en moins de 20 secondes. Les résultats montrent que les tailles des solutions retournées par GH n'excèdent pas 5 fois les bornes inférieures, ce qui veut dire qu'elles sont *au plus* 5 fois plus grand que l'optimale. Nous avons aussi vu que pour un même graphe la borne inférieure ne change pas énormément, et ce malgré les partitions en cliques différentes. En effet les écarts types après 10 exécutions ne dépassent pas 1%.

3.7 Bilan

Dans ce chapitre nous avons résolu de manière exacte l'*indépendant dominant de taille minimum* basée sur des *partitions en cliques*. Nous avons montré que cette approche a une complexité potentiellement meilleure que les autres méthodes exponen-

tielles sur des graphes "suffisamment" grands et denses. A partir de ces travaux, nous avons codé un algorithme complet avec une procédure de partitionnement des graphes en cliques, une heuristique de construction d'une solution initiale et une recherche récursive incluant des critères de réduction de l'espace des solutions. Afin de compléter ce travail, nous avons testé notre programme sur de nombreuses familles de graphes pouvant aller jusqu'à 600 sommets pour la méthode exacte et jusqu'à 60000 sommets pour l'heuristique. Les résultats ont montré que notre algorithme est capable de résoudre de manière exacte de "grands" graphes en des durées "raisonnables". Bien qu'il existe peu de résultats numériques dans la littérature, nos tests montrent que l'approche par partition en cliques est meilleure que les *deux* expériences numériques précédentes.

Cette approche que nous avons développée est assez générique. Comme nous l'avons remarqué dans ces premiers chapitres, elle a l'avantage de combiner à la fois les propriétés d'*indépendance* et de *domination* en prenant *un* seul sommet par clique. Dans les chapitres suivants, nous allons résoudre de manière *exacte* et *approchée* le problème du *vertex cover*. En effet, pour couvrir une clique, il faut au moins tous les sommets de la clique sauf *un*. Puis, dans le dernier chapitre, nous étendrons nos travaux vers les problèmes avec conflits et nous montrerons qu'ils sont en général très difficiles à résoudre, même lorsqu'il s'agit seulement de déterminer s'il existe ou non une solution.

Deuxième partie

Problème du vertex cover

Chapitre 4

Présentation du problème

Sommaire

Dans cette partie nous étudions un autre problème $\mathcal{NP}$-*complet* bien connu : le *vertex cover*. Dans la continuité de notre approche par partition en cliques, ce chapitre est l'occasion de présenter le problème et de le résoudre de manière exacte. Nous nous inspirons de la méthode présentée précédemment pour développer un algorithme récursif incluant des filtres pour résoudre le problème. Nous faisons une courte analyse expérimentale de notre programme ainsi que de ses limites et nous comparons différentes versions de notre algorithme. Puis dans le chapitre 5 nous détaillons un nouvel algorithme d'approximation de facteur *strictement* inférieur à 2. Enfin le chapitre 6 est l'occasion pour nous d'introduire la notion de *conflits* dans les problèmes proches de ceux que nous avons étudié. Nous verrons que cela rend encore plus difficile ces problèmes.

4.1 Vertex cover de taille minimum

Soit $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ un graphe non orienté et non pondéré d'ensemble de sommets $\mathcal{V}$ et d'ensemble d'arêtes $\mathcal{E}$. Un *vertex cover* (VC) est un sous-ensemble de sommets $\mathcal{S} \subseteq \mathcal{V}$ tel que chaque arête uv a au moins une extrémité dans $\mathcal{S}$ ($u \in \mathcal{S}$ ou $v \in \mathcal{S}$ ou les deux). Le problème d'optimisation associé consiste à construire un vertex cover de taille *minimum* (minimum vertex cover : mVC). Dans la suite de cette partie, nous dirons qu'une solution est *meilleure* qu'une autre si elle est de taille inférieure. Par exemple, sur la figure 4.1, nous résolvons le VC pour un graphe de 7 sommets et 11 arêtes. Il est clair que les sommets foncés sur la figure (a) forment un VC du graphe car ils couvrent

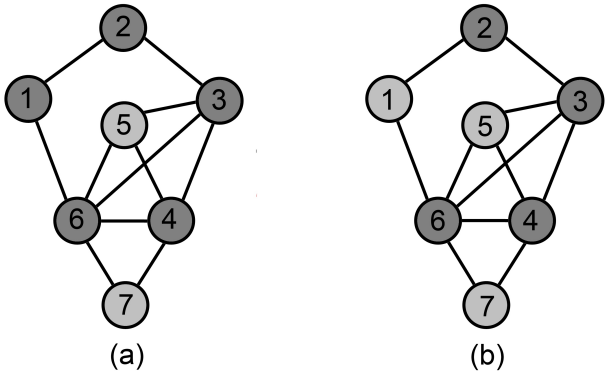

FIGURE 4.1 – Vertex cover dans un graphe

toutes les arêtes ; mais ce sous-ensemble de sommets n'est pas un mVC. Sur la figure (b) les sommets foncés forment bien un minimum vertex cover. C'est un problème classique de la recherche opérationnelle dont la $\mathcal{NP}$-complétude a été démontrée dans l'article fondateur de Karp [Karp 1972] en 1972. De plus, Garey, Johnson et Stockmeyer ont montré en 1976 [Garey 1976] que le problème reste difficile même pour certaines classes de graphes comme les graphes de degrés bornés (le vertex cover est alors désigné sous le nom de node cover).

La recherche d'un vertex cover est un problème qui peut intervenir dans différents domaines scientifiques comme par exemple : la génétique [Roth-Korostensky 2000], la biologie [Stege 2000] ou encore les réseaux de communications [Zhang 2006]. Prenons par exemple le futur câblage sous marin du réseau internet [1] dont un schéma est donné à la figure 4.2. Supposons qu'une agence nationale de sécurité quelconque souhaite surveiller ce réseau afin de protéger les données des utilisateurs de la planète. Afin de limiter le coût logistique de cette protection l'agence souhaite minimiser le nombre de *machines* installées sur les *nœuds* du réseau : une machine installée sur un nœud permet d'enregistrer toutes les informations qui transitent sur les câbles incidents. Modélisons ce problème par le graphe donné à la figure 4.3 dans lequel les nœuds du réseau sont modélisés par des sommets et les câbles qui les lient sont modélisés par des arêtes. Le problème est trivialement un problème de vertex cover dans lequel il faut sélectionner un sous-ensemble de sommets couvrant toutes les arêtes : ces sommets sont en gris foncé.

Dans un certain nombre d'autres domaines, il est important de générer un grand nombre de solutions rapidement quitte à accepter des solutions de mauvaise qualité. Dans cet objectif, de nombreuses heuristiques ont été proposées avec des temps de calculs très petits [Khuri 1994, Yuan 1998, Kotecha 2003, Chen 2004, Shyu 2004,

1. Les informations peuvent être trouvées sur cette page `http://www.cablemap.info`.

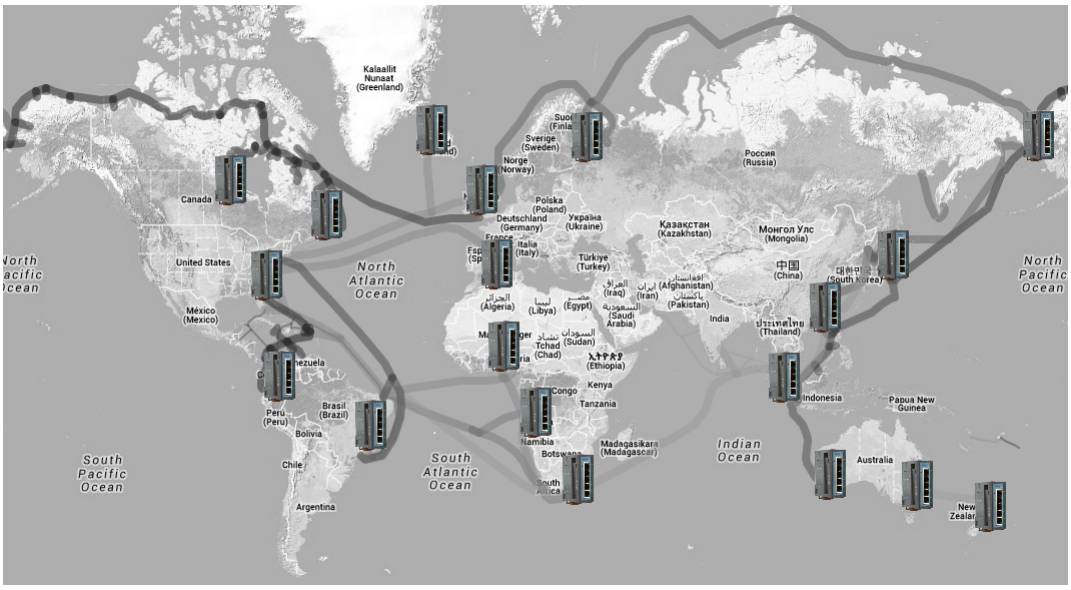

FIGURE 4.2 – Problème modélisable en vertex cover

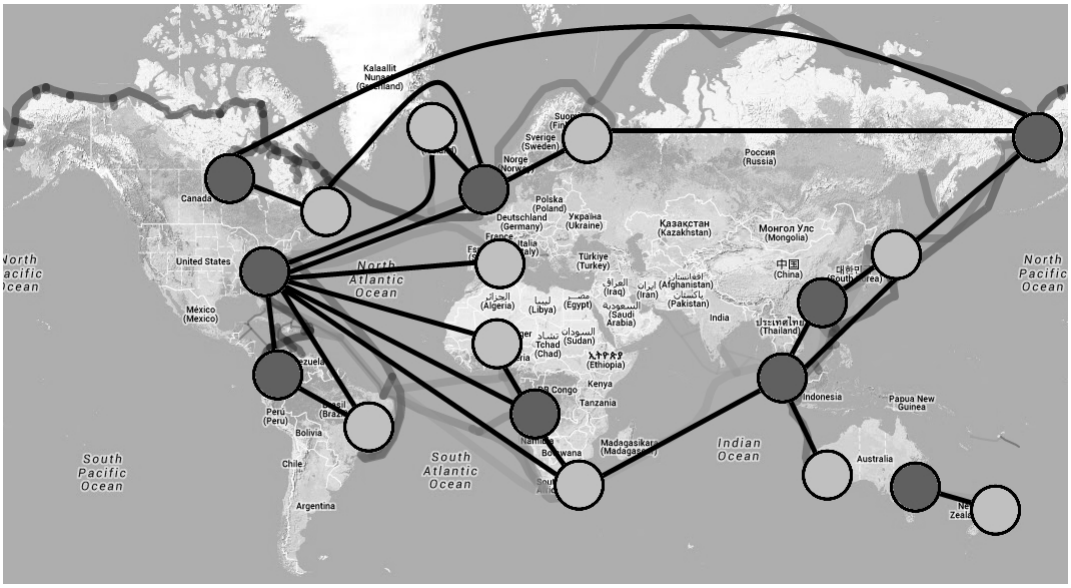

FIGURE 4.3 – Modélisation en vertex cover et solution

Xu 2006].

Le problème d'optimisation du mVC appartient à la classe $\mathcal{FPT}$ (Fixed Parameter Tractable) : il peut être résolu (de manière exacte) en temps polynomial en fonction d'un paramètre k, souvent proportionnel à la taille de la solution optimale. Cette complexité "paramétrique" a suscité beaucoup d'intérêts (voir [Downey 1995]) et beaucoup d'algorithmes exacts ont été proposés (voir [Cheetham 2003, Chen 2010]). Ces algorithmes sont très efficaces dans les applications où le paramètre k est petit, la complexité est bornée par exemple par $O(1.2738^k + kn)$ [Chen 2010], avec n le nombre de sommets dans le graphe. Cependant ces algorithmes ne sont pas utilisés à cause de l'*incertitude* sur la taille de k ; on fait alors appel aux algorithmes d'approximation.

Le problème d'optimisation mVC appartient également à la classe $\mathcal{APX}$: il admet un algorithme polynomial dont le rapport d'approximation est borné par une constante. En 1997, Håstad [Håstad 1997] a montré que le problème n'est pas approximable avec un facteur inférieur à $\frac{7}{6}$ sauf si $\mathcal{P} = \mathcal{NP}$. Certains algorithmes *simples* donnent des rapports d'approximation constants de 2 (voir par exemple la page 3 de [Vazirani 2001] et [Savage 1982]). L'un des plus connus, cité par Papadimitriou et Steiglitz (p.432 de [Papadimitriou 1982]), découvert indépendamment par Gavril et Yannakakis, utilisant le *couplage maximal* est probablement le plus étudié. En dépit des nombreux travaux, aucun algorithme polynomial avec un rapport d'approximation *constant* et *strictement* inférieur à 2 n'a été trouvé. Khot et Regev [Khot 2008] ont conjecturé la difficulté d'approximer le mVC avec un rapport constant stictement inférieur à 2. Bar-Yehuda et Even [Bar-Yehuda 1985] ont proposé un algorithme avec un facteur d'approximation de $2 - \frac{\log\log n}{2\log n}$ que Karakostas a réduit à $2 - \Theta(\frac{1}{\sqrt{\log n}})$[Karakostas 2009].

Ces dernières années Angel, Campigotto et Laforest [Angel 2011, Angel 2012, Angel 2013] ont travaillé sur la résolution de ce problème dans les très grands graphes et Delbot et Laforest [Delbot 2010] ont analysé et comparé 6 algorithmes différents pour résoudre le problème.

Avant d'aller plus loin, rappellons quelques **notations** et **définitions**. Soit un graphe $\mathcal{G}$, nous allons noter OPT la taille de la solution optimale du vertex cover. Un vertex cover $\mathcal{S}$ de $\mathcal{G}$ est *minimal* (au sens de l'inclusion) si tous sous-ensembles de $\mathcal{S}$, strictement plus petit, n'est pas un VC. Si $H \subseteq \mathcal{V}$, on note $\mathcal{G}[H] = (H, F)$ où $F = \{uv : uv \in \mathcal{E}, u \in H \text{ et } v \in H\}$. Rappelons certaines définitions utiles pour la suite. On dit que $H \subseteq \mathcal{V}$ est une *clique* de $\mathcal{G}$ *si et seulement si* $\mathcal{G}[H]$ contient toutes les arêtes possibles entre les sommets de H dans $\mathcal{G}$. On dit que $H \subseteq \mathcal{V}$ est un *indépendant* (ou *stable*) *si et seulement si* $\mathcal{G}[H]$ ne contient pas d'arêtes. On rappelle qu'une *partition en cliques* de $\mathcal{G}$ est une partition de l'ensemble $\mathcal{V}$ des n sommets en sous-ensembles disjoints $C = \{C_1, \ldots, C_k\}$ ($\cup_{i=1}^{k} C_i = \mathcal{V}$ et si $i \neq j$, $C_i \cap C_j = \emptyset$) tel que chaque sous-graphe $\mathcal{G}[C_i]$ induit par C_i dans $\mathcal{G}$ est une *clique* (dont le nombre de sommets est noté c_i $(1 \leq c_i \leq |\mathcal{V}|)$). Une clique qui ne contient qu'un seul sommet est appelé *clique triviale*. Une partition en cliques est *minimale si et seulement si* pour tout $i \neq j$ le

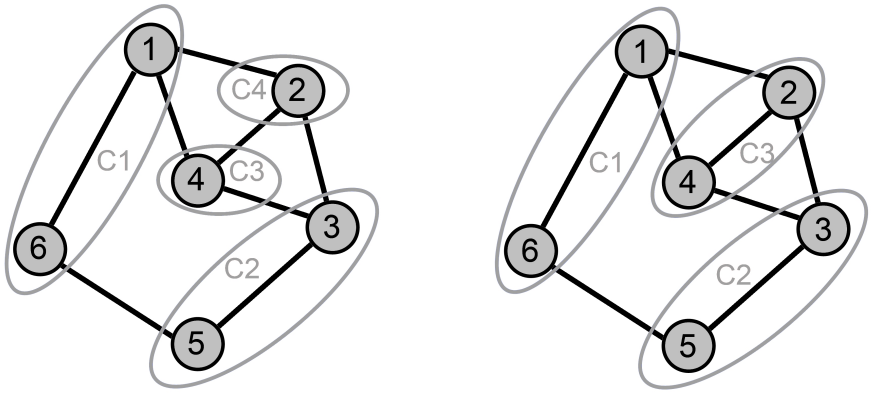

FIGURE 4.4 – Deux partitions en cliques différentes d'un même graphe

graphe $\mathcal{G}[C_i \cup C_j]$ induit par les cliques C_i et C_j *n'est pas* une clique. Sur la figure 4.4 la partition en cliques (entourées en gris) du graphe de droite est minimale alors que celle du graphe de gauche ne l'est pas car les cliques C_3 et C_4 peuvent être regroupées pour former une nouvelle clique.

En s'inspirant de l'approche par partitions en cliques de la première partie nous proposons dans un premier temps un algorithme exact pour résoudre le mVC.

4.2 Algorithme exact basé sur les partitions en cliques

Dans cette section nous adaptons la méthode de la résolution exacte du *mIDS* au problème du *vertex cover* (VC). Dans la première partie nous décrirons la recherche exhaustive du VC basée sur les partitions en cliques. Puis dans la seconde partie, nous présenterons une version raffinée qui intègre une méthode de filtrage des solutions afin d'améliorer les performances de l'algorithme.

4.2.1 Recherche exhaustive

Considérons un graphe $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ et une partition en cliques de $\mathcal{G}$, $C = (C_1, \ldots, C_k)$, $k \in \mathbb{N}^*$. Le lemme 6 donne l'idée générale de la méthode.

Lemme 6 *Soit $\mathcal{S}$ un vertex cover de $\mathcal{G}$ alors pour tout $i \leq k$: $c_i - 1 \leq |\mathcal{S} \cap C_i|$.*

`Preuve.` En effet dans chaque clique C_i, il faut au moins $c_i - 1$ sommets pour couvrir ses arêtes. □

L'idée de l'algorithme est de construire de manière exhaustive chaque sous-ensemble $\mathcal{S}$ de $\mathcal{V}$ ayant au plus un sommet absent dans chaque C_i (respectant ainsi la propriété

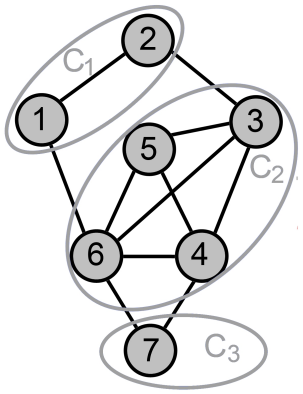

FIGURE 4.5 – Partition en cliques d'un graphe de 7 sommets et 11 arêtes

du lemme 6), et de tester si c'est un VC pour retourner à la fin le mVC. Notons que notre méthode peut être mise en œuvre comme une méthode de type recherche en profondeur : (1) initialiser une solution qui contient tous les sommets puis (2) explorer les cliques les unes après les autres dans un ordre fixé (C_1, puis C_2, etc. jusqu'à C_k). À chaque clique C_i, il y a $c_i + 1$ choix : enlever un des c_i sommets ou n'en enlever aucun. Lorsque toutes les cliques sont couvertes et si le sous-ensemble ainsi construit est un VC de taille strictement plus petite que toutes les solutions rencontrées jusqu'à présent, on met à jour la taille de la meilleure solution rencontrée. Le nombre d'ensembles candidats testés est exactement $\prod_{i=1}^{k}(c_i + 1)$ (identique à la complexité du $mIDS$).

Nous venons de décrire notre méthode générale de recherche d'un vertex cover. Dans ce qui va suivre, nous améliorons les performances de l'algorithme en ajoutant des filtres dans le schéma récursif comme c'est le cas pour l'algorithme exact pour l'*indépendant dominant de taille minimum*. Cela permet d'éviter de tester des candidats de mauvaise qualité et d'ignorer certaines branches de la récursion. Pour illustrer notre propos, nous utiliserons le graphe de la figure 4.5 de 7 sommets et 11 arêtes : la partition en cliques (entourées en gris) se compose de 3 cliques (C_1, C_2 et la clique triviale C_3).

Nous détaillons ci-dessous les filtres que nous avons ajoutées.

Pas d'arête sortante. Considérons les cliques qui ont exactement *un* sommet ayant tous ses voisins dans la même clique, comme par exemple le sommet 5 de la clique C_2 (cf. figure 4.5). Alors ce sommet est inutile dans une solution car il peut être remplacé par un autre sommet de la même clique qui aura l'avantage de couvrir au moins une arête en plus. Ainsi on peut ignorer les solutions (de faible qualité) qui contiennent ce sommet lors de l'exploration car elles ne peuvent pas apporter une solution strictement meilleure.

Un voisin n'est pas dans la solution. Une arête ne peut être dominée que par deux sommets. Si à une étape dans la clique C_i l'algorithme d'exploration enlève de la

solution un sommet v dont au moins un de ses voisins dans C_j (avec $j < i$) n'est déjà plus dans la solution alors il est inutile de continuer l'algorithme avec la solution courante car ce n'est *déjà* plus un VC.

Solution de taille minimum. On s'inspire de l'idée classique dans les problèmes d'optimisation de taille *minimum* : éliminer toutes les solutions dont on est sûr qu'elles seront de taille *supérieure* à la meilleure solution connue. Pour ce faire, à chaque étape, on calcule la taille minimum que pourra avoir la solution courante lorsqu'elle aura parcouru toutes les cliques. Si cette taille est supérieure à la taille de la meilleure solution connue (initialisée à l'ensemble des sommets du graphe), il est inutile de continuer avec la solution courante.

4.2.2 Évaluation expérimentale

Dans cette section, nous comparons les résultats expérimentaux de notre approche de résolution du mVC avec et sans les filtres. Le programme de résolution est codé en langage C. Les instances sont les mêmes que celles de la partie I. Chacune est exécutée 10 fois sur des processeurs AMD Opteron 2352 cadencés à 2.1GHz. Nous avons testé plusieurs milliers d'instances de graphes de familles différentes (pour les détails sur les familles de graphes, voir la section expérimentale 3.5 sur le $mIDS$).

Graphes aléatoires

Pour rappel, les graphes aléatoires sont des graphes dont l'existence de chaque arête est déterminée par une probabilité p utilisée en paramètre lors de la création du graphe. Le tableau 4.1 présente la durée moyenne en secondes des exécutions sur des graphes aléatoires de l'algorithme exact avec la méthode de filtrage : en colonne les probabilités p d'existence des arêtes et en ligne le nombre n des sommets des graphes. L'algorithme exact (avec les filtres) résout en moins de 3 minutes des graphes aléatoires jusqu'à 50 sommets (probabilité de 0.1 à 0.9). En comparaison la version sans filtre n'est pas capable de résoudre en moins d'une heure les graphes de plus de 37 sommets.

L'histogramme 4.6 compare les performances des versions avec et sans filtres de l'algorithme exact sur les graphes aléatoires de probabilité 0.3 : en abscisse le nombre de sommets et en ordonné la durée d'exécution du programme (échelle logarithmique, en millisecondes). Sur certaines instances, le rapport est de plus de 10000. Le programme est particulièrement efficace sur les graphes denses tels que les hypercubes, présentés au paragraphe suivant.

Graphes : hypercubes

Rappelons que les hypercubes sont des graphes de taille $n = 2^d$ (avec d un entier) relativement denses et dont la régularité permet d'éviter les effets de bord liés aux

TABLE 4.1 – Durées (en secondes) du calcul d'une solution optimale dans les graphes aléatoires par l'algorithme exact avec la méthode de filtrage

	$p = 0.1$	$p = 0.25$	$p = 0.3$	$p = 0.5$	$p = 0.7$	$p = 0.9$
$n = 20$	0	0	0	0	0	0
$n = 25$	0	0	0	0	0	0
$n = 30$	0.2	0.02	0.01	0.01	0	0
$n = 35$	0.02	0.14	0.06	0.07	0.02	0
$n = 40$	0.14	1.52	0.49	0.31	0.21	0.01
$n = 45$	4.16	5.61	12.42	2.23	0.54	0.01
$n = 50$	7.31	**142.82**	88.36	17.02	4.8	0.08

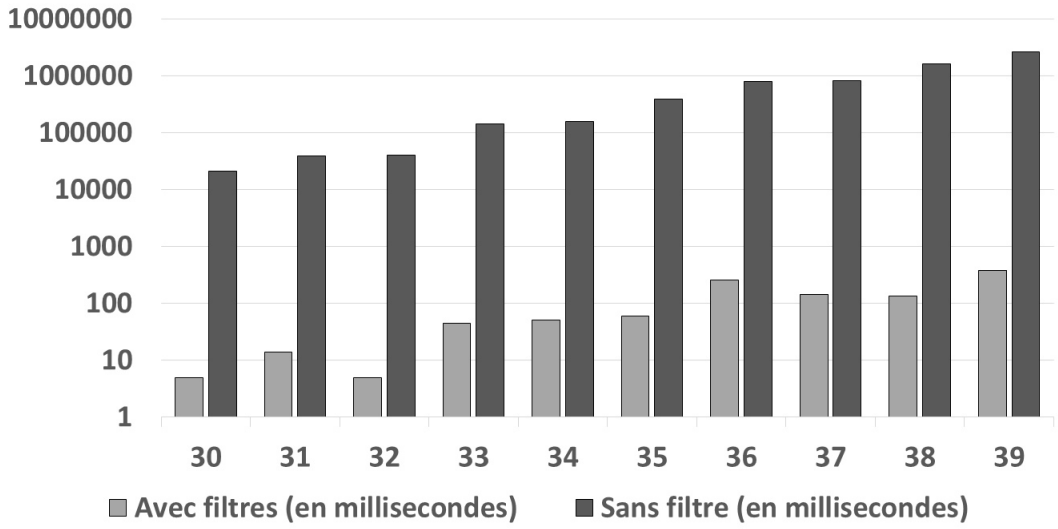

FIGURE 4.6 – Durée (en millisecondes) de deux méthodes de résolution exacte avec et sans filtres sur les graphes aléatoires

TABLE 4.2 – Durée en secondes de résolution des hypercubes par des algorithmes avec et sans filtres

	$n = 8$	$n = 16$	$n = 32$	$n = 64$	$n = 128$
Avec filtres	0	0	**0**	0.51	3398.77
Sans filtre	0	0	**143.28**	-	-

algorithmes stochastiques.

Sur ces graphes l'algorithme résout en moins d'une heure l'instance de 128 sommets. Le tableau 4.2 donne les durées d'exécutions pour les algorithmes avec et sans filtre. La comparaison montre que la différence est encore plus notable sur ces instances puisqu'il faut en moyenne plus de 2 minutes pour résoudre une instance de 32 sommets (sans les filtres) alors que c'est presque instantanée pour la version avec filtres. Au delà de 64 sommets la durée d'exécution de la version sans filtre dépasse le temps maximum fixé à trois heures.

4.3 Bilan

Dans cette section, nous avons présenté un algorithme exact pour le *vertex cover*, l'un des premiers problèmes $\mathcal{NP}$*-complet* présenté par Garey et Jonhson [Garey 1976]. L'algorithme proposé s'inspire de l'algorithme exact par partition en cliques présenté précédemment. Dans le but d'expérimenter notre algorithme, nous avons codé la méthode avec des filtres afin d'éviter de tester des solutions non pertinentes. Sur plusieurs milliers d'instances différentes l'algorithme est capable de résoudre des graphes d'une centaine de sommets. La comparaison entre les versions avec filtres et sans filtre montre l'importance de choisir les bons critères.

Dans la littérature du vertex cover exact, il existe très peu de résultats expérimentaux. Dans la mesure où le problème est approximable, l'effort est tourné vers la recherche d'algorithmes polynomiaux performants en temps et améliorant dans la mesure du possible le rapport d'approximation. Dans le chapitre suivant nous présenterons plusieurs algorithmes de résolution qui ont tous un rapport d'approximation constant de 2. Nous verrons qu'il est possible d'améliorer *un peu* ce résultat théorique. Dans la pratique et après plusieurs millions d'exécutions nous verrons que leurs performances sont très proches.

Chapitre 5

Algorithmes d'approximation pour le vertex cover

Sommaire

Soit un graphe non orienté $\mathcal{G} = (\mathcal{V}, \mathcal{E})$; une solution *optimale* au problème du vertex cover est un sous-ensemble $\mathcal{S}^*$ de sommets de $\mathcal{V}$ de taille *minimum* tel que chaque arête de $\mathcal{G}$ ait au moins une extrémité dans $\mathcal{S}^*$. Dans ce chapitre, nous présentons un nouvel algorithme d'approximation pour ce problème utilisant des *partitions en cliques*. Nous montrerons analytiquement qu'il a un rapport d'approximation en pire cas *strictement inférieur* à 2 et qu'il est potentiellement capable de fournir n'*importe quel* vertex cover *optimal*. Nous étendons notre travail et obtenons un nouvel algorithme pour le problème du *vertex cover connexe*, de rapport d'approximation 2.

5.1 Borne inférieure

Nous présentons ici une *nouvelle* borne inférieure pour le *VC* et nous la comparons avec la borne classique par *couplage*. Un couplage de $\mathcal{G}$ est un ensemble d'arêtes qui

ne partagent pas de sommet en commun. Le couplage est *maximal* s'il n'est pas lui-même un sous-ensemble d'un autre couplage *strictement* plus grand. Dans le lemme 7 nous rappelons la borne inférieure classique de OPT (taille du mVC) basée sur le *couplage maximal*. Puis le lemme 8 prouve que notre méthode donne également une borne inférieure et enfin le lemme 9 montre que pour un graphe $\mathcal{G}$ donné, la meilleure borne inférieure qui peut être obtenue avec notre méthode est toujours plus grande que la meilleure borne obtenue par couplage.

Lemme 7 *Soit $\mathcal{G}$ quelconque et $\mathcal{M}$ un couplage maximal de $\mathcal{G}$. Nous avons : $|\mathcal{M}| \leq$ OPT. Donc $|\mathcal{M}|$ est une borne inférieure de* OPT*, la taille du vertex cover optimal de $\mathcal{G}$.*

Lemme 8 *Soit $\mathcal{G}$ un graphe quelconque à n sommets et $C_1, \ldots, C_k$ ($c_i = |C_i|$ pour tout $i = 1, \ldots, k$) une partition en cliques de $\mathcal{G}$. Alors : $\sum_{i=1}^{k}(c_i - 1) = n - k \leq$ OPT.*

`Preuve.` Notons $\mathcal{S}^*$ un vertex cover optimal ($|\mathcal{S}^*| = OPT$). $\mathcal{S}^*$ doit contenir au moins $c_i - 1$ sommets de chaque clique C_i afin de couvrir toutes les arêtes de C_i. Alors :

$$\sum_{i=1}^{k}(c_i - 1) = n - k \leq OPT.$$

□

On peut remarquer que dans le graphe complet K_n la meilleure borne inférieure par le lemme 7 est $\lfloor n/2 \rfloor$ alors que la meilleure borne inférieure par le lemme 8 est $n-1$, ce qui est égal à OPT dans ce cas. De plus, le lemme 9 montre que la plus grande borne inférieure donnée par le couplage est toujours inférieure à la meilleure borne inférieure donnée par les partitions en cliques.

Lemme 9 *Soit $\mathcal{M}$ un couplage de taille maximum. Soit $n - k$ la plus grande borne inférieure donnée par la méthode de la partition en cliques. Alors : $|\mathcal{M}| \leq n-k \leq$ OPT.*

`Preuve.` Comme $\mathcal{M}$ est une partition en cliques particulière de $\mathcal{G}$ (avec des cliques de taille au plus 2) nous déduisons trivialement le résultat. □

Malheureusement dans certains graphes, cette borne inférieure ne permet pas toujours d'atteindre l'optimal : c'est le cas par exemple du cycle de 5 sommets, où $OPT = 3$ alors que la meilleure borne inférieure est 2.

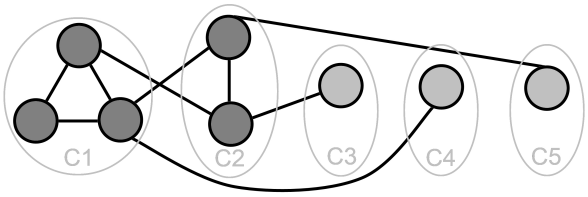

FIGURE 5.1 – Représentation d'une partition en cliques

5.2 Algorithme d'approximation de facteur 2

Dans la section 5.1 nous avons utilisé les partitions en cliques pour trouver des bornes inférieures. Dans cette partie, nous les utilisons pour construire des solutions au problème du vertex cover. Nous décrivons notre méthode appelée CP et nous analysons son rapport d'approximation en fonction de la taille des cliques. Nous montrons qu'il est inférieur à 2 et que ce rapport est atteint.

5.2.1 Algorithme CP

Soit un graphe $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ quelconque. Soit $C_1, \ldots, C_k$ une partition en cliques *minimale* de $\mathcal{G}$ ($\forall i \neq j$, $\mathcal{G}[C_i \cup C_j]$ *n'*est *pas* une clique). Supposons que les cliques triviales sont à la fin de la partition. Soit $l \leq k$ le nombre de cliques non triviales, nous avons donc $c_i \geq 2$ ($i \leq l$) et $c_{l+1} = \cdots = c_k = 1$. L'algorithme CP prend en entrée un graphe $\mathcal{G}$ et une partition en cliques minimale et retourne : $\mathcal{S} = \bigcup_{i=1}^{l} C_i$ (l'union des sommets des cliques non triviales). La figure 5.1 donne un exemple d'une partition du graphe de 8 sommets, partitionné en 5 cliques de taille respectivement trois, deux, un, un et un. La solution $\mathcal{S}$ est composée des sommets en gris foncé des cliques C_1 et C_2. Les arêtes sortantes des deux premières cliques sont couvertes. En fait seules les arêtes entre les cliques C_3, C_4 et C_5 ne seraient pas couvertes. Or si au moins une de ces arêtes existait, la partition ne serait minimale.

Dans la suite, nous supposons que $\mathcal{G}$ contient au moins une arête (autrement : $|\mathcal{S}| = OPT = 0$) et démontrons (entre autre) formellement que $\mathcal{S}$ est un *VC*.

Théorème 4 *$\mathcal{S}$ est un* VC *de $\mathcal{G}$ et :*

$$\frac{|\mathcal{S}|}{\mathrm{OPT}} \leq \frac{l}{\sum_{i=1}^{k}(c_i - 1)} + 1 = \frac{l}{n-k} + 1 \leq 2$$

De plus le rapport 2 *est atteint.*

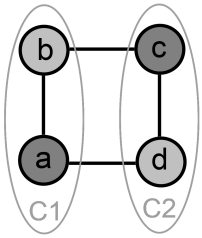

FIGURE 5.2 – Vertex cover de rapport 2

`Preuve.` $\mathcal{S}$ est un vertex cover de $\mathcal{G}$. En effet, toutes les arêtes incidentes à un des sommets de $C_1, \ldots, C_l$ sont couvertes. Les autres arêtes sont entre les cliques triviales. Supposons qu'il existe 2 cliques triviales qui ont deux sommets voisins : $uv \in \mathcal{E}$. Alors $\{u, v\}$ est une clique et cela contredit le fait que la partition en cliques est minimale.

Maintenant nous allons démontrer que le rapport d'approximation est 2. Avec le lemme 8, nous savons :

$$OPT \geq \sum_{i=1}^{k}(c_i - 1) = \sum_{i=1}^{l}(c_i - 1) + \sum_{i=l+1}^{k}(c_i - 1) = \sum_{i=1}^{l}(c_i - 1)$$

Remarquons que pour $i = l + 1$ à k, $c_i = 1$. Par construction : $|\mathcal{S}| = \sum_{i=1}^{l} c_i$. Alors :

$$\frac{|\mathcal{S}|}{OPT} \leq \frac{\sum_{i=1}^{l} c_i}{\sum_{i=1}^{l}(c_i - 1)} = \frac{l + \sum_{i=1}^{l}(c_i - 1)}{\sum_{i=1}^{l}(c_i - 1)} = \frac{l}{\sum_{i=1}^{l}(c_i - 1)} + 1 \leq 2$$

Nous pouvons aussi remarquer que : $\sum_{i=1}^{l}(c_i - 1) = \sum_{i=1}^{k}(c_i - 1) = n - k$ (car $c_i = 1$ pour tout $i > l$).

Pour terminer, considérons le cycle de 4 sommets a, b, c, d dans cet ordre et la partition en cliques $\{\{a, b\}, \{c, d\}\}$ (voir figure 5.2). Dans ce graphe $OPT = 2$ (sommets grisés) mais CP retourne une solution avec les 4 sommets contenus dans des cliques non triviales, ce qui donne un rapport d'approximation d'exactement 2.

□

A partir du théorème précédent on peut montrer que si toutes les cliques non triviales ont au moins $a \geq 3$ sommets alors le rapport d'approximation de notre méthode est au

plus $\frac{a}{a-1}$. Plus généralement, pour tout $r \in [1,2]$ nous avons une relation qui lie r et la taille moyenne des cliques non triviales notée $\bar{n}$: $\bar{n} = \frac{1}{l}\sum_{i=1}^{l} c_i$. Nous avons :

$$\frac{\sum_{i=1}^{l} c_i}{\sum_{i=1}^{l}(c_i - 1)} \leq r \Leftrightarrow \frac{l\bar{n}}{l\bar{n} - l} \leq r \Leftrightarrow 0 \leq r(l\bar{n} - l) - l\bar{n} \Leftrightarrow \frac{r}{r-1} \leq \bar{n}.$$

5.2.2 Algorithme CPGathering

L'algorithme CP prend en entrée un graphe $\mathcal{G}$ et une partition en cliques minimale de $\mathcal{G}$. Dans cette section nous décrivons un algorithme qui permet de construire une partition en cliques minimale de $\mathcal{G}$ (démontré par le lemme 10).

Soit un graphe $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. Au départ chaque sommet u est considéré comme une clique (triviale) $\{u\}$. Alors deux sommets liés par une arête peuvent être *fusionnés* pour former une nouvelle clique. Cette idée nous conduit à l'approche suivante : à chaque étape nous *fusionnons* deux cliques C_i et C_j si $\mathcal{G}[C_i \cup C_j]$ est aussi une clique.

Nous donnons à présent plus de détails sur cet algorithme que nous appelons CPGathering. Tout d'abord nous associons à chaque sommet u une clique triviale $\{u\}$. Tant qu'il existe au moins une arête nous sélectionnons aléatoirement (et de manière équiprobable) une arête uv. Nous créons un nouveau sommet w. La clique associée à w est l'union des cliques associées à u et v. Les sommets u et v sont supprimés (ainsi que toutes les arêtes qui leur sont incidentes). Nous créons de nouvelles arêtes entre w et tous les sommets qui étaient voisins aux *deux* sommets u et v. Ainsi à la prochaine étape, w est voisin des seuls sommets dont la fusion donne une clique. A la fin, il ne reste plus d'arête et la partition en cliques de $\mathcal{G}$ se compose des cliques associées au sommets restant. Cet algorithme est polynomial. La figure 5.3 donne un exemple de cette construction à travers un graphe de 6 sommets et de 8 arêtes. L'algorithme regroupe successivement les sommets grisés pour construire la partition en 3 cliques.

Lemme 10 CPGathering *retourne toujours une partition en cliques* minimale *et peut retourner n'*importe quelle *partition en cliques minimale de* $\mathcal{G}$.

`Preuve.` À chaque étape, le sommet créé est associé à une clique dans le graphe original $\mathcal{G}$. Par construction chaque sommet est exactement dans une seule clique : CPGathering construit donc une partition de $\mathcal{G}$. Cette partition est minimale ; autrement deux cliques pourraient être fusionnées, cela voudrait dire qu'il reste encore une arête entre les deux sommets associés à ces deux cliques. Cela entre en contradiction avec notre algorithme.

Maintenant, posons $P = \{C_1, \ldots, C_k\}$ une partition en cliques minimale de $\mathcal{G}$. Comme le processus de *fusion* de CPGathering est aléatoire, il est possible de

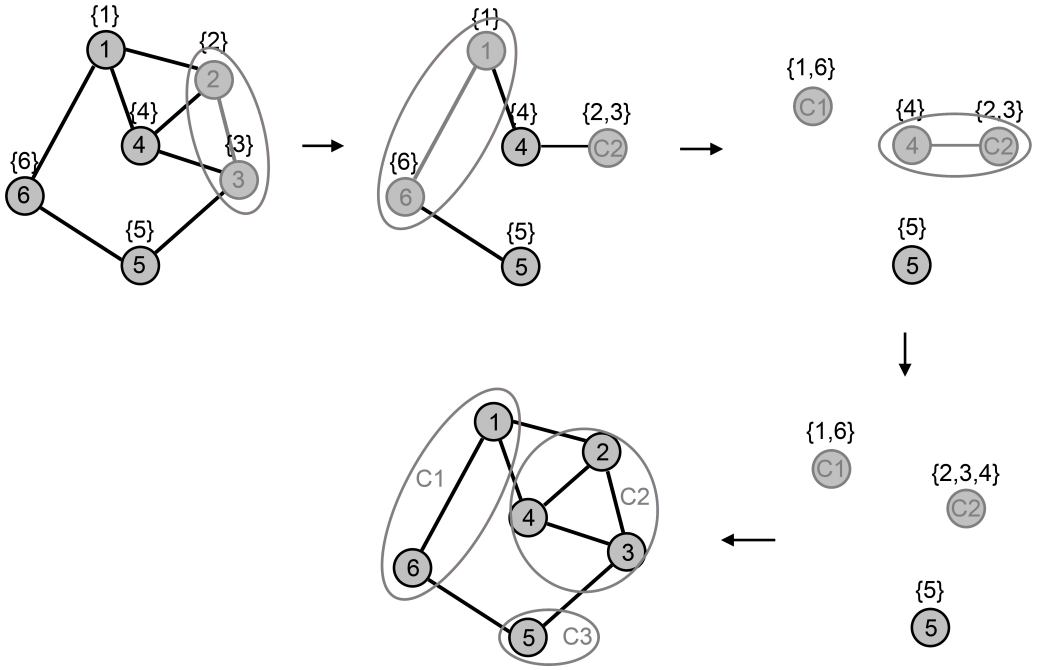

FIGURE 5.3 – Schéma illustrant la construction d'une partition en cliques avec l'algorithme CPGATHERING

construire d'abord C_1 (en $c_i - 1$ étapes). Puis l'algorithme fusionne les arêtes de C_2 et construit de manière itérative C_2, jusqu'à C_k. □

À cette étape, nous avons prouvé que CP retourne un vertex cover de rapport 2 à partir d'une partition en cliques minimale (donnée par CPGathering par exemple). Dans la section 5.3 nous présenterons un nouvel algorithme d'approximation qui améliore ce rapport.

5.3 Algorithme de facteur non constant strictement inférieur à 2

Dans cette section, nous améliorons les solutions données par CP (voir section 5.2) par l'application de l'algorithme ListRight [Birmelé 2009, Delbot 2008]. Nous montrons comment ce nouvel algorithme réduit le domaine de recherche en excluant les solutions non minimales. Nous montrons également que n'importe quelle solution optimale peut potentiellement être trouvée par l'algorithme et qu'en plus le rapport d'approximation est strictement inférieur à 2.

Delbot et Laforest [Delbot 2008] ont proposé en 2008 ListRight, un algorithme de *liste* pour le problème du vertex cover. La méthode est simple : pour un graphe $\mathcal{G}$ donné et une liste quelconque des sommets, parcourir cette liste de droite à gauche ; pour chaque sommet courant u, si u a au moins un voisin (dans $\mathcal{G}$) à sa droite dans la liste et qui n'est pas dans la solution courante $\mathcal{S}$ (initialement vide) alors u est ajouté à $\mathcal{S}$ (dans tous les autres cas, u n'est pas ajouté). Les auteurs ont montré qu'à la fin, $\mathcal{S}$ est un vertex cover minimal avec un rapport d'approximation qui est fonction de Δ, le degré maximal du graphe. Nous allons maintenant décrire l'algorithme VCCP qui combine CP avec ListRight.

5.3.1 Algorithme

Soient un graphe $\mathcal{G}$ et une partition en cliques minimale triée $C = \{C_1, \ldots, C_l, C_{l+1}, \ldots, C_k\}$ ($\forall i \leq l, c_i \geq 2$, et $\forall i$ tel que $l < i \leq k$, $c_i = 1$) de $\mathcal{G}$. Posons $\mathcal{S} = \bigcup_{i=1}^{l} C_i$ la solution retournée par CP. Nous construisons une liste L des n sommets de $\mathcal{G}$ à partir de C : à droite de L nous regroupons dans un ordre quelconque les sommets des cliques triviales $C_{l+1}, \ldots, C_k$; à sa gauche nous regroupons les autres sommets dans un ordre quelconque. Nous appliquons alors l'heuristique ListRight sur cette liste L. Notons $\mathcal{S}'$ la nouvelle solution. Comme les algorithmes CP et ListRight sont polynomiaux alors VCCP est aussi polynomial.

La figure 5.4 illustre un exemple de déroulement de cet algorithme. A partir du graphe $\mathcal{G}$ de 7 sommets et 11 arêtes et une partition minimale en 3 cliques (a), on

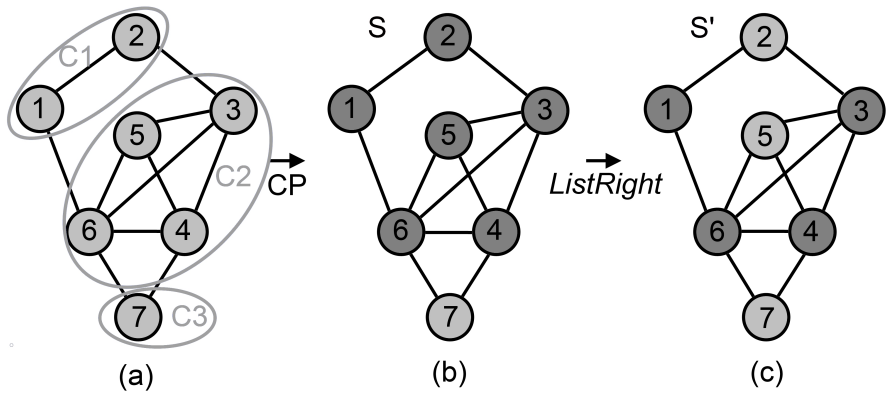

FIGURE 5.4 – Schéma illustrant l'algorithme VCCP

détermine une solution $\mathcal{S} = \{1, 2, 3, 4, 5, 6\}$ (sommets grisés du graphe (b)), union des sommets des cliques non triviales. La liste $L = 3, 1, 2, 6, 4, 5, 7$ est construite avec les sommets de $\mathcal{S}$ à gauche dans un ordre quelconque et le sommet de la clique triviale à droite de la liste. L'heuristique ListRight parcourt cette liste de droite à gauche et ajoute successivement dans la solution $\mathcal{S}'$ les sommets 4, 6, 1 et 3. $\mathcal{S}' = \{4, 6, 1, 3\}$ (sommets grisés du graphe de la figure 5.4 (c)) est un *vertex cover minimale* de $\mathcal{G}$. Le lemme 11 va le démontrer dans le cas général.

Lemme 11 *$\mathcal{S}'$ est un vertex cover minimal et $\mathcal{S}' \subseteq \mathcal{S}$.*

`Preuve.` Il a été démontré dans [Delbot 2008] que ListRight retourne un vertex cover minimal à partir de n'importe quelle liste. Par construction, tous les sommets des cliques triviales $C_{l+1}, \ldots, C_k$ sont à droite de la liste L et ils sont indépendants (car la partition est minimale), ainsi ils n'ont pas de voisins à leur droite. Par conséquent ListRight construit $\mathcal{S}'$ seulement en sélectionnant les sommets de $\mathcal{S} = \bigcup_{i=1}^{l} C_i$, ainsi $\mathcal{S}' \subseteq \mathcal{S}$. □

Le prochain résultat montre que VCCP a un rapport d'approximation *strictement* inférieure à 2.

Théorème 5 *Soit un graphe $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, avec au moins une arête. Alors : $|\mathcal{S}'| < 2\text{OPT}$.*

`Preuve.` $\mathcal{G}$ contient au moins une arête, alors $OPT \geq 1$ (autrement $OPT = |\mathcal{S}| = |\mathcal{S}'| = 0$). Comme $\mathcal{S}'$ est un sous-ensemble de $\mathcal{S}$, le théorème 4 montre que VCCP est un algorithme d'approximation de rapport 2 :

$$\frac{|\mathcal{S}'|}{OPT} \leq \frac{|\mathcal{S}|}{OPT} \leq \frac{l}{n-k} + 1 \leq 2$$

Avec cette inégalité, nous allons montrer que le rapport d'approximation est strictement inférieur à 2. Supposons qu'il existe un graphe $\mathcal{G}$ avec une partition en cliques $\{C_1, \ldots, C_l, C_{l+1}, \ldots, C_k\}$ telle que $|\mathcal{S}'| = 2OPT$. Comme nous avons $|\mathcal{S}| \leq 2OPT$ et $|\mathcal{S}'| \leq |\mathcal{S}|$ alors $|\mathcal{S}| = 2OPT$ et $\frac{l}{n-k} + 1 = 2$, donc $l = n - k$. Nous montrons maintenant que les cliques non triviales de la partition sont de taille exactement 2.

Supposons qu'il existe au moins une clique de taille 3. Si dans chaque clique non triviale nous retirons un sommet alors il reste deux sommets dans cette clique. Nous avons donc retiré $l = n-k$ sommets de ces cliques non triviales et il reste au moins $l+1$ sommets dans ces cliques. Il y a aussi $k - l$ sommets dans les cliques triviales. Ainsi le nombre de sommets n est au moins $n \geq l + (l+1) + (k-l) = l + k + 1$. Avec l'égalité précédente ($l = n - k$) nous avons $n \geq n - k + k + 1 = n + 1$: ce qui est absurde. Donc nous avons $c_1 = \cdots = c_l = 2$ et ces sommets composent la solution : $\mathcal{S} = \bigcup_{i=1}^{l} C_i$.

Nous posons $I = C_{l+1} \cup \cdots \cup C_k$ l'union des cliques triviales alors $I = \mathcal{V} - \mathcal{S}$. Rappelons que $\mathcal{S}$ est supposée être une (exacte) 2-approximation de taille $2l$ et qu'un vertex cover optimal $\mathcal{S}^*$ (de taille $OPT = l$) couvre les l cliques/arêtes C_i ($i \leq l$) car c'est un vertex cover. Ainsi $\mathcal{S}^*$ a au moins un sommet dans chaque arête C_i ($i \leq l$). Or ces arêtes sont indépendantes, donc il faut l sommets de $\mathcal{S}^*$ pour les couvrir et comme $\mathcal{S}^*$ est de taille l on en déduit que $\mathcal{S}^*$ contient exactement un sommet de chaque C_i ($i \leq l$) et $\mathcal{S}^* \subseteq \mathcal{S}$ ($\mathcal{S}^* \cap I = \emptyset$). Donc pour chaque clique (arête uv) C_i ($i \leq l$), si $u \in \mathcal{S}^*$ alors $v \notin \mathcal{S}^*$ et v n'a aucun voisin dans I (sinon cette arête ne serait pas couverte par $\mathcal{S}^*$).

Comme $|\mathcal{S}'| = |\mathcal{S}| = 2OPT$, tous les sommets dans les cliques non triviales sont sélectionnés par ListRight. Considérons alors une clique C_i ($i \leq l$) de deux sommets u et v ($uv \in \mathcal{E}$). Supposons que $u \in \mathcal{S}^*$ et $v \notin \mathcal{S}^*$. Le sommet v n'a pas de voisin w dans I (autrement, l'arête vw ne serait pas couverte par $\mathcal{S}^*$ puisque $\mathcal{S}^* \cap I = \emptyset$). Donc tous les voisins de v sont dans $\mathcal{S}'$ et $\mathcal{S}' - \{v\}$ est aussi un vertex cover. Mais cela est en contradiction avec le fait que $\mathcal{S}'$ est minimal (lemme 11). □

Le théorème 6 prouve ainsi que VCCP est capable de retourner n'importe quelle vertex cover optimal. Nous avons besoin tout d'abord de quelques résultats intermédiaires pour le montrer.

Lemme 12 *Soient un graphe $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ et un vertex cover optimal $\mathcal{S}^*$ de $\mathcal{G}$. Soit $\mathcal{S}_1^* \subseteq \mathcal{S}^*$ (et $\mathcal{S}_2^* = \mathcal{S}^* - \mathcal{S}_1^*$). Alors $\mathcal{S}_1^*$ est un vertex cover optimal du graphe $\mathcal{G}' = \mathcal{G}[\mathcal{V} - \mathcal{S}_2^*]$.*

`Preuve.` $I = \mathcal{V} - \mathcal{S}^*$ est un ensemble indépendant de $\mathcal{G}$. En retirant les sommets $\mathcal{S}_2^*$ de $\mathcal{G}$ nous retirons aussi les arêtes adjacentes aux sommets de $\mathcal{S}_2^*$. Comme il n'y a pas d'arêtes entre les sommets de I, toutes les arêtes de $\mathcal{G}'$ sont couvertes par $\mathcal{S}_1^*$ et $\mathcal{S}_1^*$ est un vertex cover de $\mathcal{G}'$.

Maintenant supposons qu'il existe un vertex cover A de $\mathcal{G}'$ tel que $|A| < |\mathcal{S}_1^*|$. A couvre les arêtes de $\mathcal{G}'$ et $\mathcal{S}_2^*$ couvre les autres arêtes de $\mathcal{G}$; alors $A \cup \mathcal{S}_2^*$ est un vertex cover de $\mathcal{G}$ mais $|A| + |\mathcal{S}_d^*| < |\mathcal{S}_1^*| + |\mathcal{S}_2^*| = |\mathcal{S}^*|$: ceci est en contradiction avec le fait que $\mathcal{S}^*$ est optimal. □

Lemme 13 *Soient un graphe $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ et un vertex cover optimal $\mathcal{S}^*$ de $\mathcal{G}$. Il existe une partition en cliques minimale de $\mathcal{G}$ telle que tous les sommets de $\mathcal{S}^*$ appartiennent aux cliques non triviales.*

`Preuve.` Soit $P = \{C_1, \ldots, C_l, C_{l+1}, \ldots, C_k\}$ une partition en cliques minimale de $\mathcal{G}[\mathcal{S}^*]$, le graphe induit par $\mathcal{S}^*$ dans $\mathcal{G}$. Nous appelons B ("Big") le sous-ensemble de sommets des cliques non triviales C_i $(i \leq l)$ et U ("Unit") le sous-ensemble de sommets des cliques triviales C_i $(l+1 \leq i \leq k)$. Alors $B \cup U = \mathcal{S}^*$ et il n'y a pas d'arête entre les sommets de U (autrement P ne serait pas minimale).

Soit $\mathcal{G}' = \mathcal{G}[\mathcal{V} - B]$, le graphe induit par $\mathcal{V} - B$ dans $\mathcal{G}$. Par le lemme 12, U est un vertex cover optimal de $\mathcal{G}'$.

Nous définissons maintenant $I = \mathcal{V} - \mathcal{S}^*$, un ensemble indépendant de $\mathcal{G}$. U et I sont deux ensembles indépendants de $\mathcal{G}'$ et $I \cap U = \emptyset$ $(U \subseteq \mathcal{S}^*)$ alors $\mathcal{G}'$ est un graphe biparti avec I et U les deux ensembles de la bipartition. Le théorème de König (voir [Plummer 1986] p.4) montre que dans les graphes bipartis, la taille du vertex cover optimal est égale à la taille du couplage maximum. Donc il existe un couplage maximum de $\mathcal{G}'$ qui couvre tous les sommets de U. Maintenant chaque sommet de $\mathcal{S}^*$ appartient à une clique non triviale : certains d'entre eux appartiennent aux cliques dans B et les autres appartiennent aux arêtes (cliques de taille 2) du couplage. Si cette partition en cliques est minimale dans $\mathcal{G}$ alors nous avons notre résultat. Dans le cas contraire, il est facile de rassembler ces cliques pour avoir une partition minimale dans laquelle chaque sommet de $\mathcal{S}^*$ est dans une clique non triviale. □

Lemme 14 *Soient un graphe $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ et une partition en cliques minimale P de $\mathcal{G}$. Supposons que $\mathcal{S}$ est un vertex cover minimal de $\mathcal{G}$ dans lequel chaque sommet appartient à une clique non triviale de P. Alors* VCCP *peut retourner $\mathcal{S}$ à partir de $\mathcal{G}$ et P.*

`Preuve.` $I = \mathcal{V} - \mathcal{S}$ est un ensemble indépendant de $\mathcal{G}$. Comme $\mathcal{S}$ est minimal, chacun de ses sommets a au moins un voisin dans I. Par hypothèse, tous les sommets des cliques triviales de P sont dans I et certains sommets des cliques non triviales de P peuvent aussi être dans I (au plus un sommet par clique). Nous construisons la liste L ainsi : à gauche, nous rassemblons dans un ordre quelconque les sommets de $\mathcal{S}$; à leur droite nous rassemblons aussi dans un ordre quelconque tous les sommets de I qui ne sont pas dans les cliques triviales ; enfin, à droite de L nous regroupons dans un ordre quelconque les sommets des cliques triviales. En appliquant ListRight sur une telle

liste, aucun sommet de I ne sera pris dans la solution car ils n'ont pas de voisin à leur droite dans la liste. Par la suite, tous les sommets de $\mathcal{S}$ seront retournés, donc $\mathcal{S}$ sera retourné à partir du graphe $\mathcal{G}$ et de la partition P. □

Théorème 6 VCCP *peut retourner n'importe quel vertex cover optimal si la partition en cliques est construite par* CPGATHERING.

`Preuve.` Soit $\mathcal{S}^*$ un vertex cover optimal de $\mathcal{G}$. Le lemme 13 prouve qu'il existe une partition en cliques minimale de $\mathcal{G}$ telle que tous les sommets de $\mathcal{S}^*$ appartiennent aux cliques non triviales. Comme $\mathcal{S}^*$ est aussi minimal, le lemme 14 montre que VCCP peut retourner $\mathcal{S}^*$ s'il prend en entrée cette partiton en cliques minimale. Pour terminer, le lemme 10 montre que CPGATHERING peut retourner n'importe quelle partition en cliques minimale. □

5.3.2 Limitations

Maintenant, nous proposons une famille de graphes appelée *MaracasGraph*, dans laquelle le rapport d'approximation de VCCP tend vers 2 quand le nombre de sommets tend vers l'infini.

Soit k un entier, qui est le paramètre de construction de notre famille de graphes. Soient r, x deux sommets du graphe $\mathcal{G}$. Il est composé de k sommets nommés a_i $(i \leq k)$ et de k sommets nommés b_i $(i \leq k)$. Chaque a_i est lié à r et chaque b_i est lié à x. Et $\forall i \leq k$, a_i et b_i sont liés. Un vertex cover optimal est composé des k sommets a_i et de x. Un tel graphe est présenté à la figure 5.5 (a). Considérons maintenant le cas particulier où la partition en cliques minimale est la suivante : $P = \{C_1, \ldots, C_{k+2}\}$ avec $C_i = \{a_i, b_i\}$ $(1 \leq i \leq k)$, $C_{k+1} = \{x\}$ et $C_{k+2} = \{r\}$ (figure 5.5 (b)). Considérons la liste $L = a_1, b_1, a_2, b_2, \ldots, x, r$ construite à partir de P. En appliquant l'algorithme LISTRIGHT, le vertex cover $\mathcal{S}$ retourné est composé de tous les sommets sauf r et x (sommets grisés sur la figure 5.5 (b)). Ainsi la taille de la solution est $2k$ alors que l'optimal est $k+1$ (chaque a_i et x : sommets grisés sur la figure 5.5 (c)). Le rapport $\frac{2k}{k+1}$ tend vers 2 lorsque k tend vers l'infini.

Nous avons démontré que l'algorithme de résolution VCCP par l'approche des partitions en cliques a un rapport d'approximation non constant strictement inférieur à 2. Bien qu'il existe des familles de graphes dans lequel le rapport est très mauvais (tendant vers 2) il serait intéressant d'étudier son comportement après implémentation et tests. Dans la section 5.4, nous procédons à ces expérimentations.

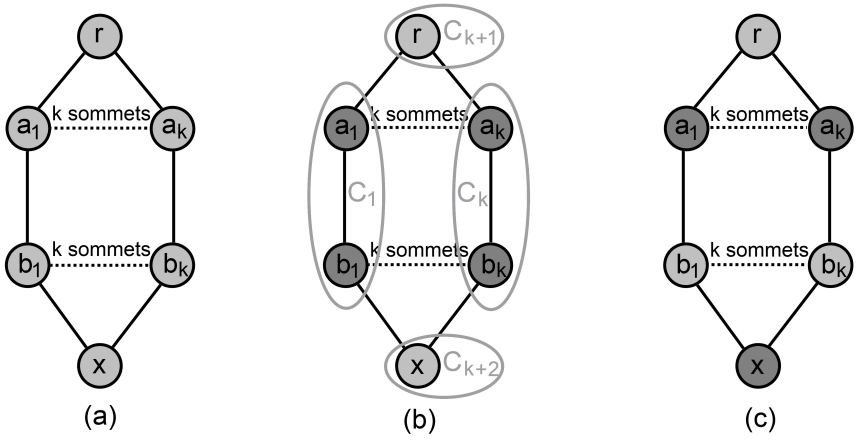

FIGURE 5.5 – Graphes de type maracas

5.4 Évaluation expérimentale

Dans cette section, nous évaluons la qualité de VCCP en comparaison avec trois autres heuristiques. À partir de l'approche par partition en cliques nous avons construit une liste sur laquelle est appliquée l'algorithme LISTRIGHT (voir section 5.3.1). Nous généralisons cette idée en combinant de la même manière LISTRIGHT avec 3 autres heuristiques bien connues (développées dans les paragraphes ci-dessous) : DEPTH-FIRST SEARCH (DFS [Savage 1982]) (prendre les sommets internes d'un arbre construit par un parcours en profondeur du graphe), EDGE DELETION (ED, voir par exemple [Delbot 2010]) proposée par Gavril (prendre les extrémités d'un couplage maximal pour l'inclusion) et LINEAR PROGRAMMING (LP [Vazirani 2001]) (résolution avec *lpsolver* dont le modèle est détaillé plus bas). Les algorithmes DFS et ED sont codés de manière stochastique alors que le problème linéaire a été résolu avec l'utilisation de la *Callable Library* de Cplex. Chacun retourne un vertex cover utilisé par LISTRIGHT : à gauche de la liste on regroupe les sommets de la solution de façon équiprobable et à droite les sommets qui ne sont pas dans la solution (de façon équiprobable). Les algorithmes sont implémentés en C.

L'algorithme DEPTH-FIRST SEARCH est un algorithme récursif qui permet de parcourir tous les sommets d'une composante connexe. Il est utilisé dans le cadre du vertex cover par Savage [Savage 1982] en 1982. Dans le contexte de notre problème nous proposons la version classique qui prend en entrée un graphe puis retourne un vertex cover. Le parcours en profondeur que nous avons mis en œuvre permet de construire un arbre de manière aléatoire : à chaque étape le choix parmi les sommets possibles se fait de façon équiprobable. Le vertex cover est l'ensemble des sommets de l'arbre

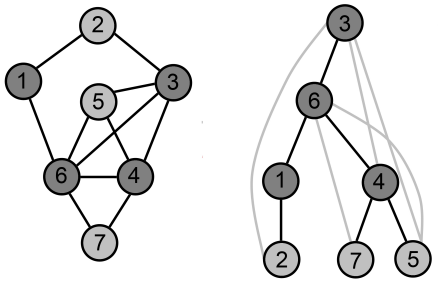

FIGURE 5.6 – Construction d'un arbre (b) à partir d'un graphe (a) par l'algorithme DFS

sans ses feuilles. La figure 5.6 illustre la construction d'un d'arbre (graphe (b)) avec DFS à partir du graphe (a). Dans une première étape, l'algorithme choisit de manière équiprobable (parmi 7 sommets possibles) le sommet numéro 3 en tant que racine. Puis parmi les voisins de 3 l'algorithme choisit le sommet 6 ; puis parmi les voisins de 6 le sommet 1 ; et enfin parmi les voisins de 1 le sommet 2. Le sommet 2 n'a pas de voisin qui ne soit pas encore dans l'arbre en cours de construction. Donc par récursion, DFS revient à 1 (mais ce dernier n'a pas non plus de voisin non encore dans l'arbre) puis revient à 6. De là DFS choisit de manière équiprobable le sommet 4 (parmi les sommets $4, 5, 7$) puis le sommet 7. Ce dernier n'a plus de voisin donc l'algorithme revient à 4 pour choisir 5. Nous obtenons l'arbre de la figure 5.6 (b) (les arêtes en gris clair permettent de retrouver les arêtes initiales du graphe (a)). Les sommets de l'arbre privés de ses feuilles (en gris foncés sur la figure) forment un vertex cover.

L'algorithme EDGES DELETION découle de la remarque simple qu'une arête ne peut être couverte que par au plus deux sommets. Ainsi si le graphe est partitionné en couplages (cliques de taille 2), les sommets du couplage forment un vertex cover de rapport d'approximation 2 [Cormen 1990]. L'algorithme 1 donne une version simplifiée de notre implémentation.

Algorithme 1 : ED

Données : Un graphe $\mathcal{G} = (\mathcal{V}, \mathcal{E})$.
Sortie : Un sous-ensemble $\mathcal{S} \subseteq \mathcal{V}$ initialement vide.

```
1 S := ∅ ;
2 tant que E ≠ ∅ faire
3 |   Choisir une arête uv ∈ E de manière équiproblable ;
4 |   S := S ∪ (u, v) ;
5 |   Enlever de E les arêtes incidentes à u et/ou à v ;
6 fin
```

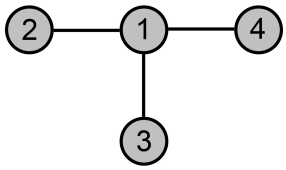

FIGURE 5.7 – Graphe de 4 sommets et 3 arêtes

La modélisation en problème linéaire du vertex cover est présentée par exemple dans [Vazirani 2001]. Chaque sommet de $\mathcal{G}$ est modélisé par une variable x_i dans le problème linéaire et chaque arête est modélisée par une contrainte reliant ses deux sommets. Prenons l'exemple de la figure 5.7 qui représente un graphe de 4 sommets et 3 arêtes. La problème linéaire (relaxé) consiste à minimiser la fonction $x_1+x_2+x_3+x_4$ sous les contraintes suivante :

$$\begin{cases} x_1 + x_2 \geq 1 \\ x_1 + x_3 \geq 1 \\ x_1 + x_4 \geq 1 \\ \forall i \in \{1,2,3,4\}, 0 \leq x_i \leq 1 \end{cases}$$

La résolution de cette version relaxée du problème dans laquelle les valeurs des x_i sont des réels est polynomiale (PL), ce qui n'est bien sûr pas le cas de la version entière (PLNE). Le PL résolu, on déduit une solution du vertex cover en prenant les sommets de $\mathcal{G}$ dont la valeur x_i dans la solution du PL est supérieure ou égale à 1/2. Notons que dans les graphes bipartis, la solution optimale du PL est une solution entière (les x_i sont égales à 0 ou à 1) : de ce fait la résolution du PLNE dans les graphes bipartis est polynomiale. Autrement dit, la résolution du vertex cover dans les graphes bipartis par programmation linéaire est exacte.

5.4.1 Test de plusieurs familles de graphes

Nous avons au total expérimenté plusieurs milliers de graphes de quatre familles différentes. Pour chacune de ces instances et pour chacune des méthodes présentées précédemment, nous avons exécuté 1000 fois chaque méthode afin d'obtenir la *taille moyenne*, de comparer la taille de la *meilleure solution rencontrée* et de calculer l'*écart type*. Ce qui nous permet de comparer les résultats des quatre méthodes entre elles mais aussi avec la valeur optimale de chaque instance. Sur ces graphes, les quatre méthodes ont des temps d'exécution très courts (jusqu'à 10 secondes par instance) et très proches : ce qui explique que nous ne les faisons pas apparaitre dans les tableaux.

Graphes aléatoires

Les figures 5.8 à 5.11 présentent le pourcentage de réduction gagné en combinant CP avec ListRight (algorithme VCCP) par rapport aux solutions données par l'algorithme CP (partition en cliques seules), pour les graphes aléatoires. Par souci de clarté nous avons sélectionné seulement les probabilités 0.1, 0.25, 0.75 et 0.9. Pour chaque graphe, les tailles des instances sont sur l'axe des abscisses et les pourcentages de réduction sont sur l'axe des ordonnées.

Soient, après 1000 exécutions, $\bar{\mathcal{S}}_{\text{VCCP}}$ la taille moyenne des solutions retournées par VCCP, $\mathcal{S}^*_{\text{VCCP}}$ la taille minimum des solutions retournées par VCCP, $\bar{\mathcal{S}}_{\text{CP}}$ la taille moyenne des solutions retournées par CP et $\mathcal{S}^*_{\text{CP}}$ la taille minimum des solutions retournées par CP. Les courbes noires donnent les pourcentages gagnés en calculant $100 * \frac{\bar{\mathcal{S}}_{\text{VCCP}}}{\bar{\mathcal{S}}_{\text{CP}}}$ et les courbes grises donnent les pourcentages gagnés en calculant $100 * \frac{\mathcal{S}^*_{\text{VCCP}}}{\mathcal{S}^*_{\text{CP}}}$.

Sur la courbe de la figure 5.9, qui présente l'analyse pour les graphes aléatoires de probabilité 0.25, de 10 à 100 sommets, on constate que l'utilisation de ListRight comme post traitement peut réduire la taille moyenne des solutions retournées par CP, de 40% (par exemple, pour l'instance de 12 sommets). De même si on ne compare que les meilleures solutions rencontrées après 1000 exécutions, la réduction atteint les 35% pour l'instance de 24 sommets du graphe de probabilité 0.1 de la figure 5.8.

La comparaison des meilleures solutions rencontrées est pertinente dans le sens où, pour 7034 instances sur 7371, l'algorithme VCCP trouve la solution optimale. Les huit courbes montrent que ListRight est très efficace sur les instances de faible densité et de petite taille alors que le gain relatif diminue pour les graphes grands et denses. Cela peut s'expliquer en partie par le fait que pour les graphes denses, la taille du vertex cover optimal est proche de la taille de l'instance.

Les figures 5.12 à 5.15 comparent le rapport moyen (après 1000 exécutions) des solutions retournées par VCCP, DFS+LR, ED+LR et LP+LR par rapport aux optimaux[1], pour les graphes de 10 à 90 sommets de probabilité 0.1, 0.25, 0.75 et 0.90. Par exemple, la figure 5.15 montre que pour le graphe de probabilité 0.9 de 80 sommets, les quatre algorithmes retournent sur 1000 exécutions des solutions dont le rapport moyen par rapport à l'optimal de l'instance est identique : seulement 1.02. Nous constatons ici que la qualité de ces algorithmes sont relativement proche et que le rapport moyen n'atteint pas le 1.2.

Ces résultats sur les graphes d'Erdős-Rényi montrent que l'efficacité des algorithmes est comparable. Après plus de 6 millions d'exécutions, les différents paramètres de mesure (taille de la solution moyenne, taille minimum des solutions rencontrés, taille

1. La taille des solutions optimales a été obtenue par l'algorithme de résolution exacte donnés dans le chapitre 4.

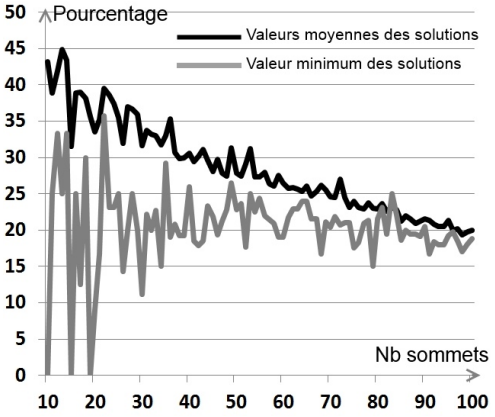

FIGURE 5.8 – Pourcentage de réduction des solutions retournées de VCCP par rapport à CP sur le graphe aléatoire $p = 0.1$

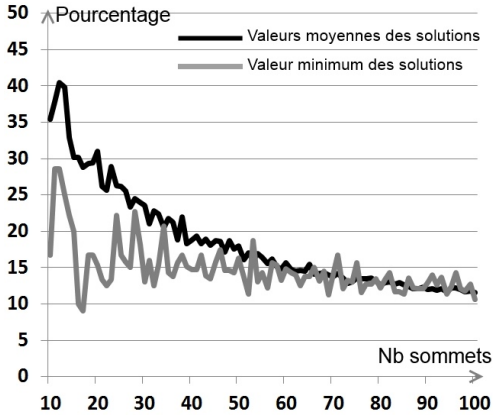

FIGURE 5.9 – Pourcentage de réduction des solutions retournées de VCCP par rapport à CP sur le graphe aléatoire $p = 0.25$

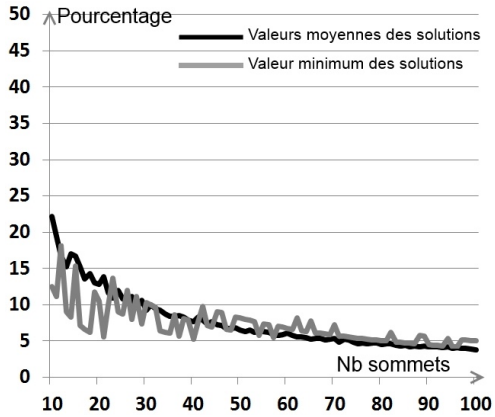

FIGURE 5.10 – Pourcentage de réduction des solutions retournées de VCCP par rapport à CP sur le graphe aléatoire $p = 0.75$

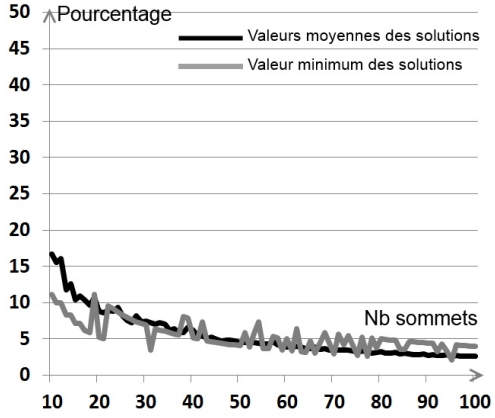

FIGURE 5.11 – Pourcentage de réduction des solutions retournées de VCCP par rapport à CP sur le graphe aléatoire $p = 0.9$

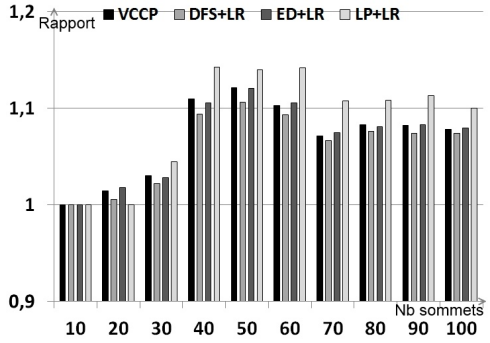

FIGURE 5.12 – Rapport moyen des solutions retournées par quatre algorithmes par rapport à l'optimal sur le graphe aléatoire $p = 0.1$

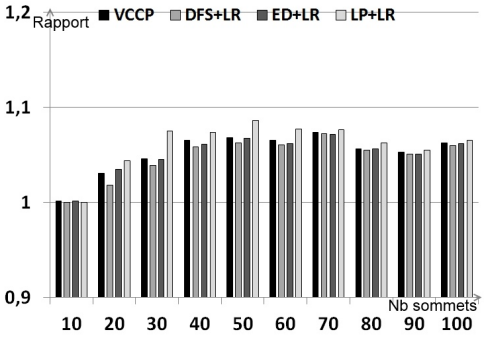

FIGURE 5.13 – Rapport moyen des solutions retournées par quatre algorithmes par rapport à l'optimal sur le graphe aléatoire $p = 0.25$

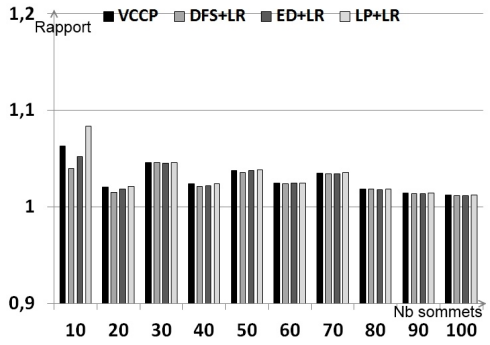

FIGURE 5.14 – Rapport moyen des solutions retournées par quatre algorithmes par rapport à l'optimal sur le graphe aléatoire $p = 0.75$

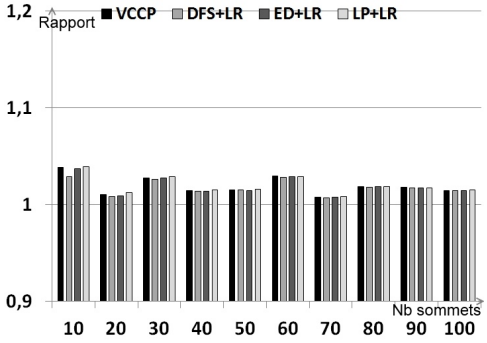

FIGURE 5.15 – Rapport moyen des solutions retournées par quatre algorithmes par rapport à l'optimal sur le graphe aléatoire $p = 0.9$

maximum des solutions rencontrées, écart-type) ont donné des résultats identiques à l'unité près. De plus la différence entre la *valeur moyenne* et la *valeur optimale* (obtenue par résolution du problème linéaire) de chaque instance est inférieure à 5% (en moyenne) et décroit lorsque le graphe est dense.

Après ces expériences, il est difficile de hiérarchiser nos algorithmes. Néanmoins, les bonnes performances de ListRight sont confirmées renforçant ainsi le choix de son utilisation comme post traitement des algorithmes de rapport d'approximation 2. Afin de mieux étudier la différence de performance des algorithmes, nous allons traiter les graphes de type *BHOSLIB*, connus pour être très difficiles pour le problème du vertex cover.

Graphes de type BHOSLIB

Nous présentons dans le tableau 5.1 une partie des résultats sur les graphes *benchmark* de type *BHOSLIB* [2], connus pour représenter des instances difficiles pour le problème du vertex cover. Les valeurs correspondent à la meilleure solution rencontrée après 1000 exécutions sur les 40 instances de 450 à 1534 sommets. Les valeurs optimales sont données par l'auteur [3]. Sur ces instances aussi, les algorithmes trouvent des solutions proches des optimales. Si on compare les méthodes entre elles, on constate que pour 35 instances (valeurs en gras dans le tableau 5.1) sur 40, notre algorithme trouve une solution aussi bonne ou meilleure que les autres, alors que le rapport est seulement de 15/40 pour *DFS*, de 12/40 pour *ED* et de 10/40 pour *LP*.

Les histogrammes des figures 5.16 à 5.19 présentent les rapports de la taille moyenne des solutions (par instance) par rapport à l'optimal, pour certains graphes de type BHOSLIB de 450 sommets à 1534 sommets. Par exemple pour l'instance *frb50-23-1.mis* de 1150 sommets de la figure 5.18, les quatre algorithmes fournissent des solutions qui sont en moyenne *très* proches et qui sont environ 1.5% supérieures à l'optimal. Ces courbes montrent très clairement que les algorithmes retournent des solutions très proches, même sur ces graphes benchmark.

Par ailleurs, les calculs montrent que pour toutes les instances, ce rapport *moyen* est toujours inférieur à 2%. Cependant, pour ces graphes, nous remarquons que l'écart type de l'algorithme VCCP est plus important que les autres algorithmes ce qui pourrait signifier que VCCP couvre un espace de solutions plus large (en terme de taille de solutions).

Avec ce groupe d'instances, nous avons résolu des graphes de plus de 1500 sommets. Il est ensuite intéressant d'essayer de résoudre des instances plus importantes pour voir les limites de nos programmes.

2. http ://www.nlsde.buaa.edu.cn/∼kexu/benchmarks/graph-benchmarks.htm.

3. Ke XU, Professor of Computer Science - National Laboratory of Software Development Environment - Department of Computer Science and Engineering - Beijing University of Aeronautics and Astronautics - Beijing, 100083, P. R. China

TABLE 5.1 – Meilleurs résultats obtenus de 4 algorithmes sur les instances BHOSLIB

	n	Opt	VCCP	DFS_LR	ED_LR	LP_LR
frb30-15-1.mis	450	420	**425**	426	**425**	426
frb30-15-2.mis	450	420	**425**	**425**	**425**	**425**
frb30-15-3.mis	450	420	**425**	426	**425**	426
frb30-15-4.mis	450	420	**425**	426	426	426
frb30-15-5.mis	450	420	**424**	425	425	426
frb35-17-1.mis	595	560	567	567	**566**	567
frb35-17-2.mis	595	560	**566**	567	567	567
frb35-17-3.mis	595	560	**566**	567	**566**	**566**
frb35-17-4.mis	595	560	**566**	567	567	567
frb35-17-5.mis	595	560	**565**	567	567	567
frb40-19-1.mis	760	720	**727**	**727**	728	728
frb40-19-2.mis	760	720	**727**	728	728	**727**
frb40-19-3.mis	760	720	**728**	**728**	729	**728**
frb40-19-4.mis	760	720	**727**	**727**	728	728
frb40-19-5.mis	760	720	**726**	729	728	728
frb45-21-1.mis	945	900	**908**	910	909	910
frb45-21-2.mis	945	900	**908**	910	910	909
frb45-21-3.mis	945	900	**909**	911	910	910
frb45-21-4.mis	945	900	**909**	**909**	910	910
frb45-21-5.mis	945	900	**909**	910	**909**	**909**
frb50-23-1.mis	1150	1100	**1109**	1111	1110	1111
frb50-23-2.mis	1150	1100	**1110**	1111	1111	**1110**
frb50-23-3.mis	1150	1100	**1110**	1112	1112	1112
frb50-23-4.mis	1150	1100	**1110**	1111	1112	1112
frb50-23-5.mis	1150	1100	1111	1112	**1110**	**1110**
frb53-24-1.mis	1272	1219	**1230**	1231	1231	1232
frb53-24-2.mis	1272	1219	**1229**	1231	1231	1230
frb53-24-3.mis	1272	1219	**1230**	1232	**1230**	1231
frb53-24-4.mis	1272	1219	**1231**	1232	**1231**	1232
frb53-24-5.mis	1272	1219	**1230**	1231	1231	**1230**
frb56-25-1.mis	1400	1344	**1356**	1357	1358	1357
frb56-25-2.mis	1400	1344	**1356**	1357	1357	1357
frb56-25-3.mis	1400	1344	**1355**	1357	1358	1357
frb56-25-4.mis	1400	1344	1357	**1355**	1357	1356
frb56-25-5.mis	1400	1344	**1355**	1357	1356	**1355**
frb59-26-1.mis	1534	1475	**1488**	1490	1489	**1488**
frb59-26-2.mis	1534	1475	1488	1488	**1487**	1489
frb59-26-3.mis	1534	1475	1488	1489	**1487**	1489
frb59-26-4.mis	1534	1475	**1489**	**1489**	**1489**	**1489**
frb59-26-5.mis	1534	1475	**1488**	**1488**	1490	1490
Moyenne		967.25	**976.3**	977.32	977.12	977.17

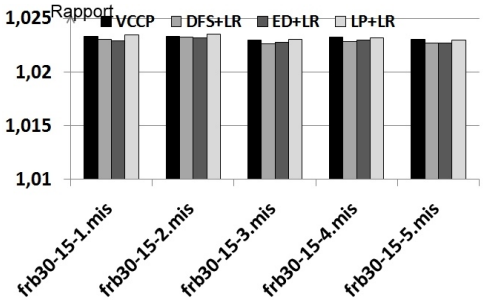

FIGURE 5.16 – Rapport moyen des solutions retournées par quatre algorithmes par rapport à l'optimal pour le graphe BHOSLIB $n = 450$

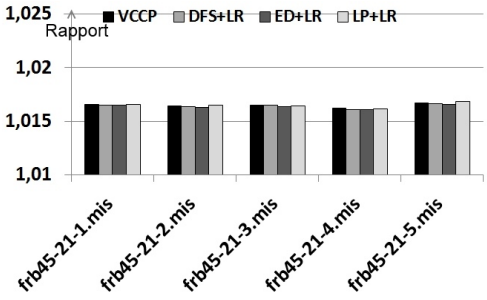

FIGURE 5.17 – Rapport moyen des solutions retournées par quatre algorithmes par rapport à l'optimal pour le graphe BHOSLIB $n = 945$

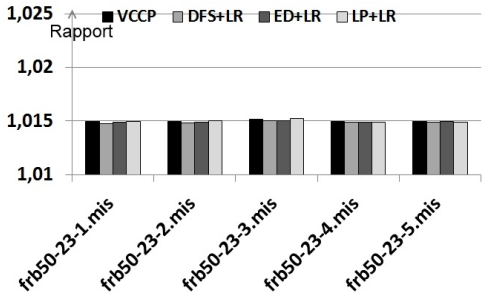

FIGURE 5.18 – Rapport moyen des solutions retournées par quatre algorithmes par rapport à l'optimal pour le graphe BHOSLIB $n = 1150$

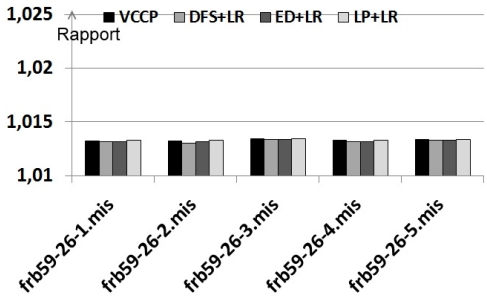

FIGURE 5.19 – Rapport moyen des solutions retournées par quatre algorithmes par rapport à l'optimal pour le graphe BHOSLIB $n = 1534$

TABLE 5.2 – Valeur moyenne de la taille des solutions sur des hypercubes après 1000 exécutions

	n	Opt	VCCP		DFS_LR		ED_LR		LP_LR	
			$\mathcal{S}_{min}$	$\overline{\mathcal{S}}$	$\mathcal{S}_{min}$	$\overline{\mathcal{S}}$	$\mathcal{S}_{min}$	$\overline{\mathcal{S}}$	$\mathcal{S}_{min}$	$\overline{\mathcal{S}}$
CHG_3.edg	8	4	**4**	5	**4**	5	**4**	5	**4**	**4**
CHG_4.edg	16	8	**8**	11	**8**	11	**8**	11	**8**	**8**
CHG_5.edg	32	16	**16**	22	**16**	22	**16**	22	**16**	**16**
CHG_6.edg	64	32	**32**	46	**32**	46	**32**	46	**32**	**32**
CHG_7.edg	128	64	80	94	76	93	84	94	**64**	**64**
CHG_8.edg	256	128	176	191	167	191	174	191	**128**	**128**
CHG_9.edg	512	256	364	388	358	389	368	388	**256**	**256**
CHG_10.edg	1024	512	759	788	748	790	750	788	**512**	**512**
CHG_11.edg	2048	1024	1564	1599	1551	1602	1559	1599	**1024**	**1024**
CHG_12.edg	4096	2048	3170	3239	3179	3245	3171	3238	**2048**	**2048**
CHG_13.edg	8192	4096	6470	6551	6475	6567	6468	6551	**4096**	**4096**
CHG_14.edg	16384	8192	13119	13242	13168	13272	13089	13241	**8192**	**8192**
Moyenne			2147	2181	2149	2186	2144	2181	**1365**	**1365**

Graphes : hypercube

Les graphes de type hypercube sont très intéressants par leur régularité (voir 3.5.1). Quelque soit la taille d'un hypercube, tous les sommets et toutes les arêtes ont exactement les mêmes caractéristiques. De plus comme le graphe est biparti, la résolution exacte se fait facilement avec notre version de l'algorithme *LP*. Nous avons résolu 12 instances de graphes de type hypercube de 8 à 16384 sommets. Le tableau 5.2 résume les moyennes des tests des 12 instances. Pour chacune de ces instances nous avons exécuté 1000 fois chaque méthode afin de comparer la taille de la *meilleure solution rencontrée* ($\mathcal{S}_{min}$) et d'obtenir leur *taille moyenne* ($\overline{\mathcal{S}}$).

Si on se restreint aux trois algorithmes VCCP, DFS_LR et ED_LR, les résultats sont tout à fait comparables : la taille moyenne des solutions ne diffère pas plus de 0.2% sur toutes les instances. Si on regarde la colonne de LP_LR on constate que l'algorithme trouve toujours la solution optimale (avec des durées d'exécution comparables aux autres). En effet dans les graphes bipartis le modèle relaxé du vertex cover en problème linéaire est équivalent à son modèle en problème linéaire en nombre entier. La résolution se fait donc de manière exacte et la partie *post traitement* par LR est inutile.

Graphes : maracas

Nous avons vu dans la section 5.3.2 une famille de graphes dont le rapport d'approximation au *pire cas* tend vers 2 lorsque n est grand. Nous allons dans ce paragraphe, étudier les résultats expérimentaux des algorithmes sur ces graphes. Nous avons résolu

TABLE 5.3 – Valeur moyenne de la taille des solutions sur les graphes de type maracas après 1000 exécutions

	n	Opt	VCCP	DFS_LR	ED_LR	LP_LR
CMaracasGraph_101_50.edg	101	50	51	50.9	51	50
CMaracasGraph_201_100.edg	201	100	101	100.9	101	100
CMaracasGraph_301_150.edg	301	150	151	150.9	151	150
CMaracasGraph_401_200.edg	401	200	201	200.9	201	200
CMaracasGraph_501_250.edg	501	250	251	250.9	251	250
CMaracasGraph_601_300.edg	601	300	301	300.9	301	300
CMaracasGraph_701_350.edg	701	350	351	350.9	351	350
CMaracasGraph_801_400.edg	801	400	401	400.9	401	400
CMaracasGraph_901_450.edg	901	450	451	450.9	451	450
CMaracasGraph_1001_500.edg	1001	500	501	500.9	501	500
CMaracasGraph_1101_550.edg	1101	550	551	550.9	551	550
CMaracasGraph_1201_600.edg	1201	600	601	600.9	601	600
CMaracasGraph_1301_650.edg	1301	650	651	650.9	651	650

68 instances de 41 à 1381 sommets. Dans le tableau 5.3 nous présentons une partie des valeurs moyennes de la taille des solutions retournées par chacun des algorithmes.

Comme ces graphes sont bipartis, il est tout naturel que l'algorithme LP_LR retourne les solutions optimales. Ce qui est plus remarquable c'est que les trois autres algorithmes retournent des solutions *très* proches de l'optimal : *un* sommet de différence en moyenne. Alors que le rapport théorique en pire cas est de 2, ici l'expérience montre qu'il est plus proche de 1. La courbe de la figure 5.20 indique par ailleurs que ce rapport tend vers 1 lorsque l'instance est de taille importante (au delà de 200 sommets).

5.4.2 Analyse des résultats

Dans cette section, nous avons implémenté notre approche par partition en cliques et trois autres méthodes basées sur des algorithmes d'approximation bien connus. Nous les avons comparé et avons ainsi pu résoudre des instances de plus de 1000 sommets. Les résultats montrent que les quatre algorithmes ont un comportement équivalent sur les graphes aléatoires d'Erdős-Rényi bien que le notre semble parcourir un espace de solutions plus large que les autres. Sur les instances "benchmark" de type *BHOSLIB*, il apparait que notre méthode semble être la plus apte à retourner la meilleure solution parmi les algorithmes testés. Dans les graphes bipartis la résolution par programmation linéaire donne des résultats exacts.

Nous avons également proposé un nouvel algorithme d'approximation de rapport

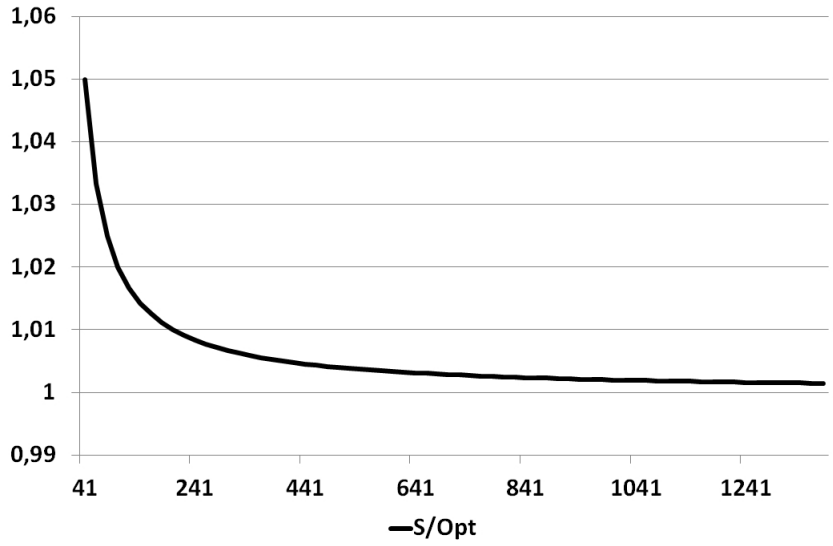

FIGURE 5.20 – Rapport entre la taille moyenne des solutions et l'optimal en fonction de la taille des graphes maracas

strictement *inférieur* à 2 pour le problème classique du vertex cover. Nous avons montré analytiquement que pour n'importe quel graphe, notre algorithme peut potentiellement retourner n'importe quel vertex cover *optimal*. De l'approche par partition en cliques présentée dans la partie I nous avons élaboré un algorithme d'approximation pour le vertex cover de taille minimum. Il est alors naturel d'étendre ces analyses sur une variante classique du problème : le vertex cover connexe.

5.5 Extension au vertex cover connexe

Dans cette section nous adaptons les algorithmes CP et CPGATHERING (voir section 5.2) pour une variante du VC. Un vertex cover $\mathcal{S}$ de $\mathcal{G}$ est *connexe* si le graphe $\mathcal{G}[\mathcal{S}]$ induit par les sommets de $\mathcal{S}$ est connexe. Un vertex cover connexe optimal, noté $\mathcal{S}^*$, de $\mathcal{G}$ est un vertex cover connexe de taille minimum. Remarquons que $OPT_{mVC} \leq |\mathcal{S}^*|$ (où OPT_{mVC} est la taille du vertex cover optimal de $\mathcal{G}$).

Ce problème est $\mathcal{NP}$-*difficile* dans les graphes planaires avec des arbres de degré maximum 4 [Garey 1977] et ne peut pas être approximé avec un rapport inférieur à $10\sqrt{5} - 21$ [Fernau 2006], sauf si $P = NP$. Savage [Savage 1982] propose un algorithme de rapport 2 pour le vertex cover (et donc pour la version connexe) dans les graphes généraux, basé sur la recherche en profondeur (DFS). Escoffier, Gourves et Monnot [Escoffier 2010] ont démontré que le problème est polynomial dans les graphes cordaux, qu'il a un PTAS dans les graphes planaires et qu'il est $\mathcal{APX}$-*difficile* dans les

graphes bipartis.

Nous décrivons ci-dessous le nouvel algorithme *CCPGathering*, pour construire des partitions en cliques "connexes" de $\mathcal{G}$. Nous utilisons comme sous routine un algorithme glouton qui construit une clique maximale C (C est maximale au sens de l'inclusion) : à partir d'un sommet u qui est initialement ajouté à C et tant que c'est possible, on ajoute à chaque étape un sommet v voisin de *tous* les sommets de C.

Construisons tout d'abord une première clique maximale. On commence par une partition en cliques initialement vide. Nous sélectionnons un sommet u quelconque et de manière gloutonne nous construisons une clique maximale qui le contient. Tous les sommets de cette première clique sont marqués (ou supprimés de $\mathcal{G}$).

Ensuite construisons les autres cliques maximales. Nous sélectionnons un autre sommet u non marqué, voisin d'au moins un sommet de la première clique. Comme précédemment nous construisons de manière gloutonne une clique maximale contenant u avec les sommets non encore marqués. Cela donne une seconde clique maximale de $\mathcal{G}$. De manière générale, supposons que nous avons construit plusieurs cliques maximales et que leurs sommets sont marqués. Tant qu'il reste un sommet non marqué, nous en sélectionnons un, appelé u, voisin d'au moins un sommet des cliques déjà construites et nous construisons une nouvelle clique maximale contenant ce sommet u avec les autres sommets non marqués de $\mathcal{G}$. Les sommets de la nouvelle clique sont marqués.

A la fin, chaque sommet de $\mathcal{G}$ appartient à exactement une clique. L'algorithme retourne un ensemble $\mathcal{S}$, union de tous les sommets des cliques non triviales. Nous montrons le théorème 4 ci-dessous.

Lemme 15 *Soit $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ un graphe connexe et $\mathcal{S}$ l'ensemble de sommets retournés par l'agorithme ci-dessus. Alors $\mathcal{G}[\mathcal{V} - \mathcal{S}]$ le graphe induit par les sommets des cliques triviales est un* stable.

`Preuve.` Appelons C la dernière clique construite, à une étape donnée, à partir d'un sommet u. Supposons que C est une clique triviale, *c'est-à-dire* qui ne contient que le sommet u, alors u n'est voisin d'aucune autre clique triviale $C' = \{v\}$, autrement, C' n'aurait pas été maximale lors de sa construction. Ainsi à la fin de l'algorithme, les sommets des cliques triviales induisent un stable dans $\mathcal{G}$. □

Lemme 16 *Avec les notations ci-dessus $\mathcal{G}[\mathcal{S}]$ est connexe.*

`Preuve.` Appelons C la dernière clique construite, à une étape donnée, à partir d'un sommet u. Supposons que C est une clique non triviale alors u est lié à au moins une autre clique non triviale (autrement u ne serait lié qu'à des cliques triviales mais dans ce cas, l'une d'entre elles aurait pu s'étendre à u puisqu'ils sont voisins). Ainsi à chaque étape, le graphe induit par les sommets des cliques non triviales est connexe. □

Théorème 7 *Soit $\mathcal{S}$ l'ensemble construit par l'algorithme polynomial décrit ci-dessus. $\mathcal{S}$ est un vertex cover connexe de $\mathcal{G}$ et : $|\mathcal{S}| \leq 2|\mathcal{S}^*|$. Le rapport 2 est atteint.*

`Preuve.` Le lemme 15 montre que le graphe induit par les sommets des cliques triviales est un stable donc $\mathcal{S}$ est trivialement un vertex cover de $\mathcal{G}$. Le lemme 16 prouve que le graphe $\mathcal{G}[\mathcal{S}]$ induit par $\mathcal{S}$ dans $\mathcal{G}$ est connexe donc $\mathcal{S}$ est un vertex cover connexe de $\mathcal{G}$. De plus, comme la partition en cliques est minimale, par construction, avec le théorème 4, nous avons : $|\mathcal{S}| \leq 2OPT_{mVC}$. Or $OPT_{mVC} \leq |\mathcal{S}^*|$ donc $|\mathcal{S}| \leq 2|\mathcal{S}^*|$.

Enfin, considérons un chemin de 4 sommets a, b, c, d dans cet ordre. Une possible partition minimale en clique est $\{\{a, b\}, \{c, d\}\}$. L'algorithme retourne alors une solution de taille 4 alors que l'optimal est de taille 2 (sommets b, c). □

5.6 Bilan

Dans ce chapitre nous avons présenté un nouvel algorithme d'approximation pour résoudre le problème du *vertex cover de taille minimum*. L'approche par partition en cliques nous a permis de trouver une borne inférieure à partir de laquelle nous avons montré que l'algorithme a un rapport d'approximation *constant* 2. Ce rapport a été amélioré pour être *strictement* inférieur à 2 (valeur non constante). Pour être complet, nous avons également présenté une sous routine qui permet de générer des partitions en cliques minimales donnant ainsi à notre algorithme principal le potentiel de retourner n'importe quelle vertex cover optimal. Les comparaisons (dans les même conditions expérimentales) avec les algorithmes Depth-First Search, Edges Deletion et Linear Programming ont montré que les performances des méthodes sont relativement équivalentes sauf dans les instances benchmark, connus pour être des instances difficiles pour le vertex cover, où notre approche est légèrement meilleure.

Avec l'indépendant dominant de taille minimum, le vertex cover est un second problème qui se résout de manière assez naturelle avec des partitions en cliques. Dans le chapitre suivant nous commenceront l'analyse de problèmes dérivés/proches du *mIDS* et du *mVC*.

Chapitre 6

Problèmes de graphe avec conflits

Sommaire

Précédemment, nous avons vu différentes méthodes pour résoudre le *vertex cover de taille minimum* et l'*indépendant dominant de taille minimum* : des problèmes qui interviennent dans différents domaines scientifiques [Roth-Korostensky 2000, Stege 2000, Zhang 2006, Kuhn 2005]. Dans ce chapitre nous étendons notre étude à des problèmes proches, qui peuvent servir à modéliser d'autres cas réels : vertex cover *sans conflit*, vertex cover connexe *sans conflit*, indépendant dominant *sans conflit*, dominant *sans conflit* et arbre de Steiner *sans conflit*. Dans un graphe, un *conflit* est une relation entre deux sommets (ou deux arêtes) qui s'excluent mutuellement : dans un couple de sommets (respectivement d'arêtes) en conflit si l'un est pris dans la solution l'autre est exclu.

La recherche et la construction d'une solution *sans* conflit dans un graphe (qui en possède) a été traité par [Zhang 2011, Jansen 1997, Szeider 2003, Darmann 2011, Kanté 2013]. Certains problèmes comme le sac à dos s'y prête particulièrement bien lorsqu'il faut, par exemple, éviter de mettre ensemble certains objets [Pferschy 2009]. L'administrateur d'un réseau peut aussi être confronté à des problèmes qui se modélisent en *vertex cover* [Zhang 2006] et les conflits permettent de prendre en compte les ressources limitées (ex. électricité), partagées par différents sommets. Les graphes avec conflits interviennent également dans la résolution de problèmes de programmation en nombres entiers [Atamtürk 2000].

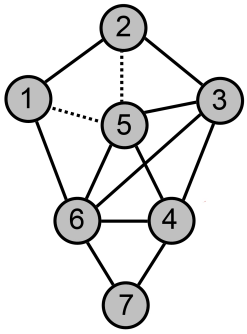

FIGURE 6.1 – Graphe avec conflits

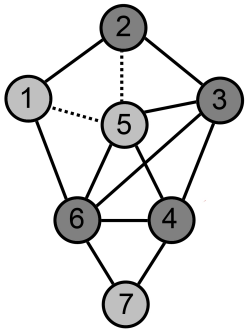

FIGURE 6.2 – Vertex cover sans conflit

6.1 Vertex cover sans conflit

Dans cette section nous traitons une variante du vertex cover : le vertex cover *sans conflit* (vertex cover with no conflict : *VCwnC*). Soit un graphe $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ et un ensemble $\mathcal{C}$ de *paires* de sommets de $\mathcal{G}$. Une paire de sommets de $\mathcal{G}$, $\{u, v\} \in \mathcal{C}$, représente un *conflit* et $\mathcal{C}$ est un *ensemble* de conflits. Un vertex cover sans conflit (*VCwnC*) est un sous-ensemble $\mathcal{S} \subseteq \mathcal{V}$ qui est un vertex cover (voir le chapitre 4 pour la définition) de $\mathcal{G}$ et qui ne contient pas de conflit : $\forall u, v \in \mathcal{S}$, $\{u, v\} \notin \mathcal{C}$. Le problème de décision consiste à savoir s'il existe un tel sous-ensemble de $\mathcal{G}$. Le problème d'optimisation consiste à construire un *VCwnC* de taille minimum. Ce problème est trivialement $\mathcal{NP}$*-difficile* ; en effet lorsque $\mathcal{C}$ est vide, le problème revient à trouver un *vertex cover optimal*.

La figure 6.1 donne un exemple d'un graphe avec 7 sommets, 11 arêtes à couvrir (en noir) et 2 conflits (arêtes en pointillées). Remarquons qu'un conflit peut très bien exister entre les deux extrémités d'une même arête de $\mathcal{G}$ (pour des raisons de clarté, nous ne représentons pas de tel cas dans nos figures). Sur la figure 6.2 les sommets d'un *VCwnC* sont en gris foncé. Une telle solution n'existe pas toujours comme le montre la figure 6.3 : les conflits empêchent de prendre les couples de sommets $1, 3$ ou $2, 4$ qui sont nécessaires dans tous les *VC* de ce graphe.

Dans la suite nous allons utiliser le problème *2-SAT* pour résoudre le *VCwnC*. Rappelons tout d'abord la définition du *2-SAT*.

Soit un ensemble fini de *variables booléennes* $\mathcal{X} = \{x_1, \ldots, x_n\}$ et une *formule propositionnelle* $\mathcal{F} = C_1 \wedge ... \wedge C_l$ composée de *clauses* $C_i = y_{i,1} \vee y_{i,2}$ telles que $\forall (i, j) \in [\![1; n]\!]^2$, $y_{i,j}$ est un *littéral* de la variable x_k $(k = 1, \ldots, n)$ *c'est-à-dire* : $y_{i,j} = x_k$ ou $y_{i,j} = \neg x_k$. Une affectation $\tau : \mathcal{X} \to \{0, 1\}$ satisfait $\mathcal{F}$ *si et seulement si* pour chaque clause C_i, $\tau(y_{i,1}) = 1$ ou $\tau(y_{i,2}) = 1$ (si $y_{i,j} = \neg x_k$ alors $\tau(y_{i,j}) = \neg\tau(x_k)$). Une solution *2-SAT* est une affectation $\mathcal{S}_F$ qui satisfait $\mathcal{F}$. Une solution *optimale* est une

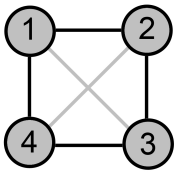

FIGURE 6.3 – Graphe avec conflits sans solution

affectation $\mathcal{S}_F$ satisfaisant $\mathcal{F}$ dont le nombre de variables x_i assignées à 1 ($\mathcal{S}_F(x_i) = 1$) est *minimum*. On note $|\mathcal{S}_F|$ le nombre de variables booléennes telles que $\mathcal{S}_F(x_i) = 1$.

Le problème de décision pour *2-SAT* (*c'est-à-dire* savoir s'il existe une affectation satisfaisant $\mathcal{F}$) est polynomial. Melven Robert Krom [Krom 1967] a proposé un algorithme polynomial par *fermeture transitive* (transitive closure) pour retourner une affectation *si elle existe*. Aspvall, Plass et Tarjan [Aspvall 1979] ont amélioré la qualité de l'approche en 1979. En 1992, Gusfield et Pitt [Gusfield 1992] prouvent que trouver une affectation *optimale* est *$\mathcal{NP}$-difficile*. Dans le même article, ils présentent un algorithme d'approximation de facteur 2 pour le problème.

A présent nous réduisons le *VCwnC* au *2-SAT*. Soit $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ un graphe non orienté et $\mathcal{C}$ un *ensemble de conflits*. Chaque sommet de $\mathcal{G}$ est associé à une variable booléenne x_i (pour simplifier x_i désigne la variable et le sommet correspondant). Pour chaque arête $\{x_i, x_j\} \in \mathcal{E}$ on ajoute une clause *positive* $(x_i \vee x_j)$ dans $\mathcal{F}$. Pour chaque conflit $\{x_i, x_j\} \in \mathcal{C}$ on ajoute une clause *négative* $(\neg x_i \vee \neg x_j)$ dans $\mathcal{F}$. A titre d'exemple, pour le graphe de la figure 6.1, nous avons la formule suivante : $\mathcal{F} = (x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (x_3 \vee x_4) \wedge (x_4 \vee x_7) \wedge (x_7 \vee x_6) \wedge (x_6 \vee x_1) \wedge (x_6 \vee x_5) \wedge (x_5 \vee x_3) \wedge (x_4 \vee x_6) \wedge (x_6 \vee x_3) \wedge (x_5 \vee x_4) \wedge (\neg \mathbf{x_1} \vee \neg \mathbf{x_5}) \wedge (\neg \mathbf{x_2} \vee \neg \mathbf{x_5})$ (les deux dernières clauses représentent les conflits). Une affectation possible $\mathcal{S}_F$ qui satisfait $\mathcal{F}$ est : $(\mathcal{S}_F(x_i) = 1)_{i=2,3,4,6}$ et $(\mathcal{S}_F(x_i) = 0)_{i=1,5,7}$.

Par cette construction polynomiale, $\mathcal{F}$ est donc sous la forme d'une instance *2-SAT*. Soit $\mathcal{S}_F$ une affectation qui satisfait $\mathcal{F}$ et $\mathcal{S}$ l'ensemble des sommets x_i de $\mathcal{G}$ tels que $\mathcal{S}_F(x_i) = 1$.

Théorème 8 *$\mathcal{S}$ est un* VCwnC *de $\mathcal{G}$ si et seulement si $\mathcal{S}_F$ satisfait $\mathcal{F}$. De plus $|\mathcal{S}| = |\mathcal{S}_F|$.*

`Preuve.` Soit $\mathcal{S}_F$ une affectation qui satisfait $\mathcal{F}$; alors les clauses positives $(x_i \vee x_j)$ sont satisfaites ; ainsi toutes les arêtes de $\mathcal{E}$ sont couvertes par $\mathcal{S}$. De la même manière, les clauses négatives $(\neg x_i \vee \neg x_j)$ sont satisfaites ; ainsi chaque paire de sommets dans $\mathcal{C}$ a au plus un de ses deux sommets dans $\mathcal{S}$. Donc $\mathcal{S}$ est un *VCwnC* de $\mathcal{G}$. Réciproquement si $\mathcal{S}$ est un *VCwnC* de $\mathcal{G}$ alors toutes les arêtes de $\mathcal{E}$ ont au moins une extrêmité dans $\mathcal{S}$; ainsi les clauses positives $(x_i \vee x_j)$ sont satisfaites. De plus comme chaque conflit de

$\mathcal{C}$ a au plus un sommet dans $\mathcal{S}$, les clauses négatives $(\neg x_i \vee \neg x_j)$ sont aussi satisfaites. Donc l'affectation $\mathcal{S}_F$ satisfait $\mathcal{F}$. Et par construction $|\mathcal{S}| = |\mathcal{S}_F|$. □

Théorème 9 *Le problème de décision du* VCwnC *est polynomial.*

`Preuve.` En appliquant la transformation polynomiale présentée et en utilisant un des algorithmes de [Krom 1967] ou [Aspvall 1979] pour résoudre le *2-SAT*, il est possible de trouver en temps polynomial un *VCwnC* pour un couple $(\mathcal{G}, \mathcal{C})$ donné, *s'il existe* ou de dire s'*il en existe aucun.* □

Corrolaire 3 *Le problème d'optimisation du* VCwnC *est approximable avec un rapport* 2.

`Preuve.` Le théorème 8 montre qu'à tout *VCwnC* correspond une affectation satisfaisant $\mathcal{F}$, de la même taille ; il en est de même pour les solutions *optimales* (de taille minimum). Ainsi, chercher un optimal du *VCwnC* est équivalent à chercher une affectation de taille minimum qui satisfait le problème *2-SAT* correspondant.

En utilisant la transformation polynomiale et l'algorithme d'approximation de rapport 2 de [Gusfield 1992] pour le problème de l'affectation optimale du *2-SAT*, nous avons naturellement une 2-approximation pour le *VCwnC*. □

Ainsi, à l'instar du *VC*, le *VCwnC* peut être résolu avec un rapport d'approximation de 2. Comme nous avons vu dans le chapitre 4, une de ses extensions est le vertex cover *connexe*. La section suivante va traiter de la forme *connexe* de cette extension.

6.2 Vertex cover connexe sans conflit

Le vertex cover *connexe* sans conflit (connected vertex cover with no conflict : *CVCwnC*) est un cas particulier du vertex cover sans conflit. Soit $(\mathcal{G},\mathcal{C})$, un vertex cover connexe sans conflit (*CVCwnC*) est un sous-ensemble $\mathcal{S}$ des sommets de $\mathcal{G}$ tel que $\mathcal{S}$ est un *vertex cover* ($\forall\{u,v\} \in \mathcal{E}$, $u \in \mathcal{V}$ et/ou $v \in \mathcal{V}$), $\mathcal{G}[\mathcal{S}]$ (le graphe induit par les sommets de $\mathcal{S}$ dans $\mathcal{G}$) est connexe et $\forall u, v \in \mathcal{S}, \{u,v\} \notin \mathcal{C}$. Le problème de décision du *CVCwnC* consiste à savoir s'il existe un tel sous-ensemble. Notons qu'une telle solution peut ne pas exister comme sur l'exemple de la figure 6.4 : le *chemin* de 5 sommets et un conflit, n'admet pas de vertex cover connexe car le conflit entre les sommets 2 et 4 empêche d'avoir les deux sommets dans la même solution.

Pour notre démonstration nous réduisons le *CVCwnC* au problème du *Exact 3-Cover* (*X3C*), un problème *$\mathcal{NP}$-complet* classique [Garey 1979] dont on rappelle la définition. Soit $\mathcal{X} = \{u_1, u_2, \ldots, u_{3q}\}$ ($q \in \mathbb{N}$) un ensemble d'éléments et une famille $\mathcal{F}$ de $k \in \mathbb{N}$ ensembles $C_i \subseteq \mathcal{X}$ et $|C_i| = 3$ ($i \in [\![1;k]\!]$) tel que $\cup_{i=1}^{k} C_i = \mathcal{X}$. Le problème

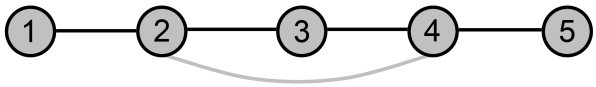

FIGURE 6.4 – Graphe sans solution pour le *CVCwnC*

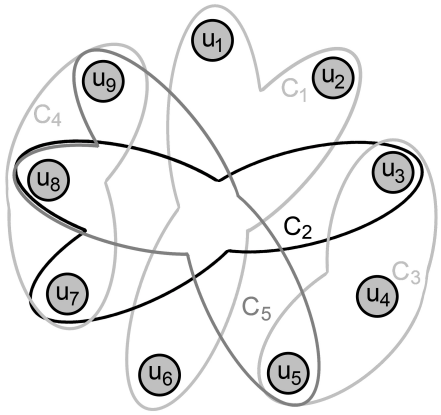

FIGURE 6.5 – Schéma d'une instance de *X3C*

du *X3C* consiste à savoir s'il existe une couverture exacte $\mathcal{S}_\mathcal{F} \subseteq \mathcal{F}$ de $\mathcal{X}$: $\forall C_i \in \mathcal{S}_\mathcal{F}, \forall C_j \in \mathcal{S}_\mathcal{F} - \{C_i\}, C_i \cap C_j = \emptyset$ et $\bigcup_{C_i \in \mathcal{S}_\mathcal{F}} C_i = \mathcal{X}$. La figure 6.5 donne un exemple d'un tel problème : l'ensemble $\mathcal{X} = \{u_1, \ldots, u_9\}$ est composé de $3q = 9$ éléments (les sommets) et la famille $\mathcal{F}$ est composée de $k = 5$ ensembles $C_1 = \{u_1, u_2, u_6\}$, $C_2 = \{u_3, u_7, u_8\}$, $C_3 = \{u_3, u_4, u_5\}$, $C_4 = \{u_7, u_8, u_9\}$ et $C_5 = \{u_5, u_8, u_9\}$ (entourés en gris) de 3 éléments. L'ensemble $\{C_1, C_3, C_4\}$ est une couverture exacte de ($\mathcal{X}$,$\mathcal{F}$).

Théorème 10 *Décider s'il existe un* CVCwnC *est* $\mathcal{NP}$-complet *même dans les graphes bipartis.*

`Preuve.` De manière triviale, le problème est dans $\mathcal{NP}$. Soit $(\mathcal{X}, \mathcal{F})$ une instance quelconque de *X3C*. Construisons une instance $(\mathcal{G}, \mathcal{C})$ de la manière suivante. Chaque $u_j \in \mathcal{X}$ est un sommet de $\mathcal{G}$ (noté u_j aussi). Chaque $C_i \in \mathcal{F}$ est un sommet de $\mathcal{G}$ (noté C_i aussi). On ajoute un nouveau sommet r, connecté à tous les sommets C_i. Dans $\mathcal{G}$ chaque sommet C_i est connecté aux 3 sommets u_a, u_b, u_c *si et seulement si* dans $\mathcal{F}$ l'ensemble $C_i = \{u_a, u_b, u_c\}$. Deux sommets C_i et C_j distincts sont en conflit ($\{C_i, C_j\} \in \mathcal{C}$) *si et seulement si* dans $\mathcal{F}$, $C_i \cap C_j \neq \emptyset$. La construction est polynomiale et le graphe est *biparti.*

Pour illustrer cette transformation, la figure 6.6 présente le couple ($\mathcal{G}$,$\mathcal{C}$) de 15 sommets, 15 arêtes et 5 contraintes (arêtes grisées) construit à partir du problème *X3C* présenté précédemment (voir la figure 6.5).

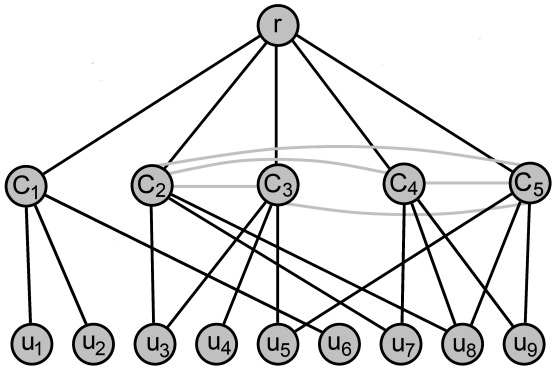

FIGURE 6.6 – Génération d'un graphe à partir d'une instance de *X3C* (les conflits sont en gris clair)

Soit $(\mathcal{X}, \mathcal{F})$ une instance de *X3C* et $(\mathcal{G}, \mathcal{C})$ l'instance du *CVCwnC* associée.

Supposons qu'il existe une solution $\mathcal{S}_\mathcal{F}$ pour le problème du *X3C*. Soit $\mathcal{S} = \{r\} \cup \mathcal{S}_\mathcal{F} \cup \mathcal{X}$. $\mathcal{G}[\mathcal{S}]$ est connexe (les C_i sont connectés entre eux par $r \in \mathcal{S}$ et chaque u_i est connecté à un C_i car $\mathcal{S}_\mathcal{F}$ couvre $\mathcal{X}$). Toute arête $e \in \mathcal{E}$ est couverte par $\mathcal{S}$ (car $\mathcal{S}$ contient r et tous les u_i). Ainsi $\mathcal{S}$ est un vertex cover connexe de $\mathcal{G}$. Il n'y a aucun conflit dans $\mathcal{S}$ car les seuls conflits qui peuvent exister sont entre les sommets de type C_i ; or comme $\mathcal{S}_\mathcal{F}$ est une solution de *X3C*, $\forall C_i, C_j \in \mathcal{S}_\mathcal{F}, C_i \cap C_i = \emptyset$ et ils ne sont donc pas en conflit. Ainsi $\mathcal{S}$ est un *CVCwnC*.

Supposons maintenant qu'il existe un *CVCwnC* de $(\mathcal{G}, \mathcal{C})$. Comme $\mathcal{G}[\mathcal{S}]$ est connexe, $\mathcal{S}$ contient nécessairement des sommets de type C_i. Notons $\mathcal{S}_\mathcal{F}$ l'ensemble des sommets de type C_i de $\mathcal{S}$. Comme $\mathcal{S}$ est sans conflit on a : $\forall C_i, C_j \in \mathcal{S}_\mathcal{F}, C_i \cap C_j = \emptyset$. On a aussi $\bigcup_{C_i \in \mathcal{S}_\mathcal{F}} C_i = \mathcal{X}$, sinon $\exists x \in \mathcal{X}$ tel que $x \notin \bigcup_{C_i \in \mathcal{S}_\mathcal{F}} C_i$ et on est alors dans l'un de ces deux cas : (1) $x \notin \mathcal{S}$, comme x a au moins une arête incidente, celle-ci n'est pas couverte et c'est absurde car $\mathcal{S}$ est un vertex cover ; (2) $x \in \mathcal{S}$, dans ce cas, ses arêtes incidentes sont bien couvertes par $\mathcal{S}$ mais x est un sommet isolé de $\mathcal{G}[\mathcal{S}]$ (aucun voisin de x n'est dans $\mathcal{S}$) et c'est absurde car $\mathcal{G}[\mathcal{S}]$ est connexe. Ainsi, $\mathcal{S}_\mathcal{F}$ est bien une solution du problème *X3C*. □

Nous venons de démontrer que le problème de décision du *CVCwnC* est *$\mathcal{NP}$-complet* par la réduction à *X3C*. Nous pouvons remarquer que dans la preuve du théorème 10, le graphe induit $\mathcal{G}[\mathcal{S}]$ est un arbre et que la solution $\mathcal{S}$ couvre tous les sommets u_i ; cela nous a conduit à adapter cette démonstration au problème de l'arbre de Steiner à la section 6.3.3. Mais avant cela, nous allons présenter la variante avec conflits des problèmes étudiés dans la partie I de ce document. Ainsi la section suivante adapte cette démonstration pour montrer que décider de l'existence d'un indépendant dominant *sans conflit* et d'un dominant *sans conflit* est également *$\mathcal{NP}$-complet*.

6.3 Autres problèmes avec conflits

Ici nous démontrons la $\mathcal{NP}$-complétude de la décidabilité pour quelques problèmes classiques, lorsque des conflits sont ajoutés. Les versions sans conflit sont triviales. Nos réductions sont aussi basées sur le *X3C* dont la description et les notations sont données à la section 6.2.

6.3.1 Indépendant dominant sans conflit

La partie I nous a confronté aux problèmes de type indépendant dominant de taille minimum (*mIDS*). Il est alors naturel de s'interroger sur sa version *sans conflit*.

Soit $(\mathcal{G},\mathcal{C})$, un *indépendant dominant sans conflit* (Independent Dominating with no Conflict : *IDwnC*) est un sous-ensemble $\mathcal{S} \subseteq \mathcal{V}$ tel que $\mathcal{S}$ est un *dominant* de $\mathcal{G}$ ($\forall u \in \mathcal{V}$, $u \in \mathcal{S}$ ou $N(u) \cap \mathcal{S} = \emptyset$), $\mathcal{S}$ est un *indépendant* de $\mathcal{G}$ (pour tout $u \in \mathcal{S}$ et tout $v \in \mathcal{S}$, $uv \notin \mathcal{E}$) et $\mathcal{S}$ est *sans conflit* ($\forall u, v \in \mathcal{S}, \{u, v\} \notin \mathcal{C}$). Le problème de décision du *IDwnC* consiste à savoir s'il existe un tel sous-ensemble dans $\mathcal{G}$.

Pour tout graphe il existe toujours au moins un indépendant dominant : il suffit de construire un indépendant maximal pour l'inclusion grâce à un algorithme glouton polynomial. Mais un *IDwnC* peut ne pas exister (si chaque sommet est en conflit avec tous les autres par exemple).

Théorème 11 *Le problème de décision de l'IDwnC est* $\mathcal{NP}$-complet*, même dans les graphes bipartis.*

`Preuve.` Trivialement, le problème est dans NP. Pour la suite, la démonstration est faite avec le problème du *Exact 3-Cover* (*X3C*). Soit $(\mathcal{X}, \mathcal{F})$ une instance de *X3C*. Construisons une instance $(\mathcal{G},\mathcal{C})$. Chaque $u_j \in X$ est un sommet de $\mathcal{G}$ (noté u_j aussi). Chaque $C_i \in \mathcal{F}$ est un sommet de $\mathcal{G}$ (noté C_i aussi) : on connecte à chaque sommet C_i un nouveau sommet noté r_i (qui n'est connecté qu'à C_i). Dans $\mathcal{G}$ chaque sommet C_i est connecté aux 3 sommets u_a, u_b, u_c *si et seulement si* dans $\mathcal{F}$ l'ensemble C_i contient les éléments u_a, u_b, u_c. Construisons maintenant l'ensemble des conflits $\mathcal{C}$. Les sommets C_i et C_j sont en conflit ($\{C_i, C_j\} \in \mathcal{C}$) *si et seulement si* dans $\mathcal{F}$, $C_i \cap C_j \neq \emptyset$. Chaque sommet u_i est en conflit avec *tous* les autres sommets de $\mathcal{G}$. La construction de l'instance $(\mathcal{G},\mathcal{C})$ est polynomiale et $\mathcal{G}$ est un graphe *biparti*.

Pour illustrer cette transformation, la figure 6.7 présente le couple $(\mathcal{G},\mathcal{C})$ de 15 sommets, 15 arêtes et de nombreuses contraintes (arêtes grisées et arêtes en pointillées autours des u_i) construit à partir du problème *X3C* présenté précédemment (voir la figure 6.5).

Supposons qu'il existe une solution $\mathcal{S}_\mathcal{F}$ pour le problème *X3C*. Soit $\mathcal{S}$ l'ensemble composé de tous les sommets $C_i \in \mathcal{S}_\mathcal{F}$ et, pour chaque $C_j \notin \mathcal{S}_\mathcal{F}$, on ajoute à $\mathcal{S}$ son sommet voisin r_j. Trivialement, $\mathcal{S}$ est un indépendant. $\mathcal{S}$ est aussi un dominant de $\mathcal{G}$

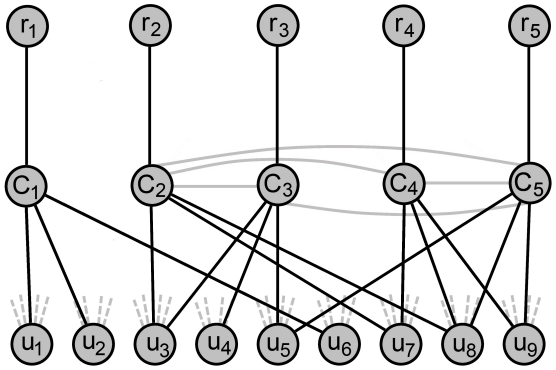

FIGURE 6.7 – Génération d'un graphe à partir d'une instance de *X3C* (les conflits sont en gris clair)

car chaque sommet u_i est dominé par exactement un $C_j \in \mathcal{S}_\mathcal{F}$; chaque C_j qui n'est pas dans $\mathcal{S}_\mathcal{F}$ est dominé par $r_j \in S$; chaque r_i est soit dans $\mathcal{S}$ soit dominé par son C_i ; enfin, $\mathcal{S}$ n'a aucun conflit car C_i et C_j sont en conflit *si et seulement si* $C_i \cap C_j \neq \emptyset$ or $C_i \in \mathcal{S}_\mathcal{F}$, $C_j \in \mathcal{S}_\mathcal{F}$ et $\mathcal{S}_\mathcal{F}$ est une partition de X. De plus, les sommets r_j ne sont en conflit avec aucun sommet de type C_i.

Supposons maintenant qu'il existe un *IDwnC* noté $\mathcal{S}$ de $(\mathcal{G},\mathcal{C})$. Dans ce cas, $\mathcal{S}$ ne contient aucun sommet de $\mathcal{X}$ (car chacun de ses sommets est en conflit avec tous les autres et on ne pourrait donc pas avoir de solution avec plus d'un sommet, ce qui est impossible). $\mathcal{S}$ contient donc obligatoirement des C_i et éventuellement des sommets de type r_j. Notons $\mathcal{S}_\mathcal{F}$ l'ensemble des sommets C_i de $\mathcal{S}$. Comme $\mathcal{S}$ est un dominant $\cup_{C_i \in \mathcal{S}_\mathcal{F}} C_i = \mathcal{X}$. De plus, comme $\mathcal{S}$ est sans conflit, pour tout $C_i \in \mathcal{S}_\mathcal{F}$ et tout $C_j \in \mathcal{S}_\mathcal{F}$ différent on a $C_i \cap C_j = \emptyset$. Ainsi $\mathcal{S}_\mathcal{F}$ est une solution du problème *X3C*. □

Après avoir démontré que savoir s'il existe une solution pour l'indépendant dominant *sans conflit* est $\mathcal{NP}$*-complet*, il est trivial d'en déduire le corolaire pour les dominants *sans conflit*, précisé dans la section suivante.

6.3.2 Dominant sans conflit

Pour un couple $(\mathcal{G},\mathcal{C})$ donné, un dominant *sans conflit* (Domination with no Conflict : *DwnC*) est un sous-ensemble $\mathcal{S} \subseteq \mathcal{V}$ tel que $\mathcal{S}$ est un *dominant* de $\mathcal{G}$ ($\forall u \in \mathcal{V}$, $u \in \mathcal{S}$ et $N(u) \cap S \neq \emptyset$) et *sans conflit* ($\forall u, v \in \mathcal{S}, \{u, v\} \notin \mathcal{C}$). Le problème de décision du *DwnC* consiste à savoir s'il existe un tel sous-ensemble dans $\mathcal{G}$. Notons que pour tout graphe il existe toujours au moins un dominant (on peut prendre par exemple tous les sommets de $\mathcal{G}$) mais qu'un *DwnC* peut ne pas exister (par exemple si chaque sommet est en conflit avec tous les autres).

Corrolaire 4 *Le problème de l'existence du DwnC est* $\mathcal{NP}$-complet *même dans les graphes bipartis.*

`Preuve.` Trivialement, le problème est dans NP (il suffit de parcourir la solution et de vérifier que tous les sommets sont dominés). Remarquons tout d'abord que l'*indépendant dominant* est un cas particulier du *dominant* sans conflit (lorsque $\mathcal{E} = \mathcal{C}$).

Soit $(\mathcal{G}, \mathcal{C})$ une instance du problème $IDwnC$. Soit $\mathcal{C}' = \mathcal{C} \cup \mathcal{E}$ (les arêtes de $\mathcal{G}$ sont considérées comme des conflits dans $\mathcal{C}'$). Il est maintenant facile de montrer que : $\mathcal{S}$ est un $IDwnC$ de $(\mathcal{G}, \mathcal{C})$ *si et seulement si* $\mathcal{S}$ est un $DwnC$ de $(\mathcal{G}, \mathcal{C}')$. Trivialement, quel que soit un indépendant dominant sans conflit $\mathcal{S}_{IDwnC}$ de $(\mathcal{G}, \mathcal{C})$, $\mathcal{S}_{IDwnC}$ est un dominant sans conflit de $(\mathcal{G}, \mathcal{C}')$ puisque $\mathcal{C}' = \mathcal{C} \cup \mathcal{E}$. De même, quel que soit un dominant sans conflit $\mathcal{S}_{DwnC}$ de $(\mathcal{G}, \mathcal{C}')$, il est aussi indépendant car pour tout $u \in \mathcal{S}_{DwnC}$ et tout $v \in \mathcal{S}_{DwnC}$, $uv \notin \mathcal{E}$ et $uv \notin \mathcal{C}$. Cette équivalence et le théorème 11 nous donne le résultat. □

Remarquons que lorsque l'ensemble des conflits est un sous-ensemble des arêtes du graphe ($\mathcal{C} \subseteq \mathcal{E}$), le problème de l'existence d'un $IDwnC$ ou d'un $IDwnC$ est polynomial. En effet, dans cette situation, il existe au moins un $IDwnC$ (dont la construction se fait en temps polynomiale avec un simple algorithme glouton) ; de plus quel que soit un $IDwnC$, c'est aussi un $DwnC$.

Cette section a permis de démontrer que le problème d'existence des problèmes d'indépendant dominant et de dominant *sans conflit* sont très difficiles et ce, même dans les graphes bipartis.

Jusqu'à présent, l'utilisation du problème de $X3C$ semble très efficace pour démontrer la difficulté de savoir s'il existe ou pas une solution en temps polynomial de nos problèmes *sans conflit*. Nous avons alors essayé de l'étendre à d'autres problèmes, comme l'arbre de Steiner présenté dans la section suivante.

6.3.3 Arbre de Steiner sans conflit

Soient $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ un graphe non orienté, $\mathcal{C}$ un *ensemble de conflits* de $\mathcal{G}$ et $\mathcal{M} \subseteq \mathcal{V}$ un sous-ensemble de sommets. L'arbre de Steiner sans conflit (Steiner Tree with no Conflict : $STwnC$) pour l'instance $(\mathcal{G}, \mathcal{C}, \mathcal{M})$ est un arbre $\mathcal{S}_T = (\mathcal{V}_T, \mathcal{E}_T)$ couvrant $\mathcal{M}$ ($\mathcal{M} \subseteq \mathcal{V}_T \subseteq \mathcal{V}$) dans $\mathcal{G}$ ($\mathcal{V}_T \subseteq \mathcal{V}$ et $\mathcal{E}_T \subseteq \mathcal{E}$) qui ne contient aucun conflit ($\forall u, v \in \mathcal{S}, \{u, v\} \notin \mathcal{C}$). Le problème de décision du $STwnC$ consiste à savoir s'il existe un tel arbre pour l'instance $(\mathcal{G}, \mathcal{C}, \mathcal{M})$. Notons qu'une telle solution peut ne pas exister comme sur l'exemple 6.8 : l'arête en gris représente les conflits et les sommets 1 et 5 (en gris foncé) sont à couvrir. Il est impossible de joindre ces deux sommets sans passer par les sommets 2 et 4. Avec seulement $(\mathcal{G}, \mathcal{M})$, il est trivial de savoir s'il existe ou non un arbre couvrant $\mathcal{M}$ dans $\mathcal{G}$. Avec les conflits, le problème est $\mathcal{NP}$*-complet* comme dans les sections précédentes.

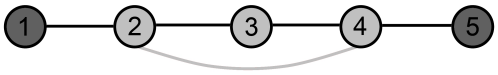

FIGURE 6.8 – Graphe sans solution pour le *STwnC*

Théorème 12 *Décider s'il existe un* STwnC *est un problème* $\mathcal{NP}$-complet*, même dans les graphes bipartis.*

`Preuve.` Le problème est dans $\mathcal{NP}$. La démonstration se fait, encore une fois, avec le problème du *X3C* (voir la section 6.2 pour les notations). Soit $(\mathcal{X}, \mathcal{F})$ une instance quelconque de *X3C*. Construisons une instance $(\mathcal{G}, \mathcal{C}, \mathcal{M})$ de la manière suivante. Chaque $u_j \in \mathcal{X}$ est un sommet de $\mathcal{G}$ (noté u_j aussi). Chaque $C_i \in \mathcal{F}$ est un sommet de $\mathcal{G}$ (noté C_i aussi). On ajoute un nouveau sommet r, connecté à tous les sommets C_i. Dans $\mathcal{G}$ chaque sommet C_i est connecté aux 3 sommets u_a, u_b, u_c *si et seulement si* dans $\mathcal{F}$ l'ensemble $C_i = \{u_a, u_b, u_c\}$.

On pose $\mathcal{M} = \mathcal{X}$. Deux sommets C_i et C_j distincts sont en conflit ($\{C_i, C_j\} \in \mathcal{C}$) *si et seulement si* dans $\mathcal{F}$, $C_i \cap C_j \neq \emptyset$. La transformation est donc identique à la section précédente et la figure 6.6 dessine le triplet ($\mathcal{G}$, $\mathcal{C}$, $\mathcal{M}$) construit à partir du problème *X3C* : $\mathcal{X} = \{u_1, \ldots, u_9\}$ et $C_1 = \{u_1, u_2, u_6\}$, $C_2 = \{u_3, u_7, u_8\}$, $C_3 = \{u_3, u_4, u_5\}$, $C_4 = \{u_7, u_8, u_9\}$ et $C_5 = \{u_5, u_8, u_9\}$.

Supposons qu'il existe une solution $\mathcal{S}_\mathcal{F}$ pour le problème du *X3C*. Soit $\mathcal{S} = \{r\} \cup \mathcal{S}_\mathcal{F} \cup \mathcal{X}$. $\mathcal{S}$ couvre tous les sommets de $\mathcal{M}$ car $\mathcal{M} = \mathcal{X} \subset \mathcal{S}$. $\mathcal{G}[\mathcal{S}]$ est connexe : les C_i sont connectés entre eux par $r \in \mathcal{S}$ et chaque $u_i \in \mathcal{X}$ est connecté à un $C_i \in \mathcal{S}_\mathcal{F}$ car $\mathcal{S}_\mathcal{F}$ couvre $\mathcal{X}$. Donc $\mathcal{G}[\mathcal{S}]$ contient un arbre qui couvre $\mathcal{M}$. Il n'y a aucun conflit dans $\mathcal{S}$ car les seuls conflits qui peuvent exister sont entre les sommets de type C_i ; or $\mathcal{S}_\mathcal{F}$ est une solution de *X3C* ($\forall C_i, C_j \in \mathcal{S}_\mathcal{F}, C_i \cap C_i = \emptyset$). Ainsi $\mathcal{G}[\mathcal{S}]$ contient un *STwnC*.

Supposons maintenant qu'il existe un *STwnC*, $\mathcal{S}_T = (\mathcal{V}_T, \mathcal{E}_T)$ de $(\mathcal{G}, \mathcal{C}, \mathcal{M})$. Comme $\mathcal{S}_T$ est connexe, il contient nécessairement des sommets de type C_i. Notons $\mathcal{S}_\mathcal{F}$ l'ensemble des sommets de type C_i de $\mathcal{S}_T$. Comme $\mathcal{S}_T$ est sans conflit on a : $\forall C_i, C_j \in \mathcal{S}_\mathcal{F}, C_i \cap C_j = \emptyset$. On a aussi $\bigcup_{C_i \in \mathcal{S}_\mathcal{F}} C_i = \mathcal{M} = \mathcal{X}$, sinon $\exists x \in \mathcal{M}$ tel que $x \notin \bigcup_{C_i \in \mathcal{S}_\mathcal{F}} C_i$. Mais on est alors dans un de ces cas : (1) $x \notin \mathcal{S}_T$, c'est absurde car $\mathcal{S}_T$ couvre tout $\mathcal{M}$; (2) $x \in \mathcal{S}_T$, dans ce cas c'est un sommet isolé de $\mathcal{S}_T$ (aucun voisin de x n'est dans $\mathcal{S}_T$) : absurde car $\mathcal{S}_T$ est un arbre. □

6.4 Bilan

Dans ce chapitre nous avons étendu nos travaux vers cinq nouveaux problèmes de type *sans conflit* : le vertex cover sans conflit, le vertex cover connexe sans conflit, l'indépendant dominant sans conflit, le dominant sans conflit et l'arbre de Steiner sans

conflit. Par une transformation en *2-SAT* nous avons montré que le problème d'optimisation du vertex cover sans conflit est $\mathcal{APX}$ avec un rapport d'approximation de 2. Par une réduction polynomiale aux problèmes de *X3C*, nous avons montré que savoir s'il existe une solution (même non optimale) était déjà un problème $\mathcal{NP}$-*complet* même dans les graphes bipartis pour l'indépendant dominant, le dominant et l'arbre de Steiner sans conflit.

Ces problèmes étant très proches nous nous sommes tout d'abord demandé s'il était possible d'appliquer nos idées pour résoudre le problème d'optimisation. Cependant ces résultats montrent qu'il faut dans un premier temps avoir un autre objectif que la résolution exacte. Pour commencer il s'agira de trouver un *bon* algorithme pour décider s'il existe ou non une solution et (peut-être) la retourner si elle existe. Dans le cadre du *mIDS* et *VC*, il serait intéressant d'essayer de nous inspirer de l'heuristique de recherche à partir d'une partition en cliques pour les résoudre.

Conclusion

L'évolution de la puissance de calcul permet de résoudre de manière exacte des problèmes de taille de plus en plus importante. Là où il y a 10 ans il fallait plusieurs jours pour résoudre certains problèmes de graphe de seulement quelques sommets, il est possible à présent de résoudre ces mêmes problèmes en seulement quelques minutes et jusqu'à plusieurs vingtaines de sommets. Nous avons souhaité, à travers ce travail, revenir sur les **algorithmes exacts** afin de développer de nouvelles approches de résolution ; mais aussi de tester expérimentalement ces approches afin d'**analyser leur comportement**. Par la suite, et dans un souci de compromis entre temps et performance, nous avons travaillé sur les **algorithmes d'approximation**. Ces derniers nous permettent expérimentalement de résoudre en quelques secondes le problème du vertex cover dans des graphes de **plusieurs centaines de sommets**.

Dans un premier temps nous avons traité le problème de l'*indépendant dominant de taille minimum* (*mIDS*). Ce problème est de la classe $\mathcal{NP}$*-complet* et il n'existe pas d'algorithme d'approximation de facteur constant pour le résoudre. Néanmoins, en travaillant sur les problèmes de couplage (partition du graphe en arêtes) dans les *line graphs*, nous avons montré que dans cette famille de graphes, le *mIDS* est 2-approximable. La résolution du problème par couplage permet facilement de prendre en compte la propriété de dominance du problème. En voulant généraliser cette méthode de résolution aux autres graphes, nous avons trouvé une approche originale, par *partition en cliques*, pour résoudre de manière exacte le *mIDS*. Notre tentative d'étudier la complexité *en moyenne* nous a conduit à proposer une nouvelle manière d'analyser la complexité algorithmique. Nous avons ainsi montré que, lorsque le graphe est suffisamment grand et suffisamment dense, la méthode par partition en cliques a une complexité exponentielle qui tend à être plus petite que n'importe quelle autre méthode. Nous continuons néanmoins à penser qu'il est possible de calculer la complexité en moyenne : qui pourrait se faire dans un premier temps, par rapport à la partition en cliques utilisée. Cette étude nous a également permis de proposer une nouvelle borne inférieure pour le problème. Les expériences numériques ont montré que le programme, codé en C++, est capable de résoudre dans un délais raisonnable (de l'ordre de quelques minutes) des problèmes de graphe jusqu'à une centaine de sommets (selon la densité et la structure du graphe). La littérature relative à la résolution exacte du *mIDS* étant assez maigre en ce qui concerne les expérimentations numériques, nous avons trouvé un seul article qui a publié et comparé de manière claire, deux algorithmes de résolution du *mIDS*, sur les grilles. En nous comparant avec ces résultats, nous avons montré que l'approche par partition en cliques est de loin la plus efficace.

Tous ces travaux nous ont aidé à remarquer que dans une clique de taille c, il faut au moins $c-1$ sommets de la clique pour avoir une couverture des arêtes, autrement dit un *vertex cover* (*mVC*). Nous avons alors traité le *mVC*. Ce problème est un problème $\mathcal{NP}$*-complet* classique, déjà présent dans l'ouvrage de Garey et Johnson. En s'inspirant de la résolution du *mIDS* nous avons résolu de manière exacte le *mVC* et la remarque

précédente nous a conduit à développer un nouvel algorithme de rapport d'approximation, *en pire cas*, constant égal à 2 et en raffinant, un autre non constant strictement inférieur à 2. Rappelons qu'il existe déjà beaucoup d'algorithmes de rapport d'approximation égal à 2 pour résoudre le mVC, mais (étonnamment) il y a très peu de résultats numériques dans la littérature. Nous avons donc pris le parti de coder et de comparer quatre algorithmes classiques de rapport d'approximation 2. Les résultats montrent qu'en pire cas, comme en moyenne, les solutions trouvées sont très proches. On peut cependant noter que sur les instances *benchmark*, réputées pour être des instances difficiles pour ce problème, l'approche par partition en cliques semble parcourir un espace de solutions plus grand et trouve plus souvent des solutions de meilleure qualité.

Dans l'objectif d'élargir le panel des problèmes qui peuvent être résolus par notre approche par partition en cliques, nous avons, dans un troisième temps, traité un nouveau type de problèmes de graphe, qui intègre des conflits. Nous avons notamment étudié les problèmes du vertex cover connexe sans conflit, de l'indépendant dominant sans conflit et de l'arbre de Steiner sans conflit, sur des graphes généraux avec des conflits. Une première analyse permet de démontrer que les problèmes de décision liés sont $\mathcal{NP}$-*complets*, présageant ainsi une forte difficulté pour l'optimisation des problèmes dans les graphes avec conflits. Malgré cette difficulté, nous avons démontré que le problème du vertex cover sans conflit est 2-approximable dans les graphes généraux. Pour les autres problèmes, une étude future serait de chercher des familles de graphes dans lesquelles les problèmes de décision liés seraient polynomiaux et dont les problèmes d'optimisation seraient $\mathcal{NP}$-*complets*.

Ces travaux mènent à plusieurs perspectives de recherche qu'il serait intéressant d'approfondir dans un futur proche. Tout d'abord, il est intéressant de voir que la complexité de l'algorithme par partition en clique, pour résoudre le $mIDS$, dépend de la qualité de la partition en cliques de départ. De même, le rapport en pire cas de l'algorithme d'approximation, pour résoudre le mVC, dépend aussi de la qualité de la partition en cliques de départ. Il serait alors utile de chercher à caractériser ce qu'est une *bonne* partition en cliques et d'étudier les probabilités et/ou la loi de distribution de l'algorithme qui génère les partitions. Cette étape réussie, il serait possible de calculer pour un graphe donné, la complexité en moyenne de l'algorithme pour résoudre le $mIDS$ et le rapport d'approximation en moyenne de l'algorithme pour résoudre le mVC. On peut également noter la généricité de cette approche par partition en cliques qui permet d'intégrer, de manière intrinsèque, la notion d'indépendance (les partitions) et de dominance (les cliques). Ces propriétés pourraient aider à adapter la méthode à d'autres problèmes de type indépendant et/ou dominant : indépendant de taille maximum, dominant de taille minimum, par exemple.

Travaux et publications

Voici la liste des soumissions et publications liées à la thèse.

Revues internationales (avec comité de lecture)

- Christian Laforest et Raksmey Phan. Experimentations of an Exact Algorithm for the Minimum Independent Dominating Set Problem in Graphs Using Clique Partition. RAIRO–RO, 47 :199-221, 2013.

Conférences nationales et internationales (avec comité de lecture)

- (Article court de 5 pages, présenté sous forme de poster) François Delbot, Christian Laforest, Raksmey Phan. New Approximation Algorithms for The Vertex Cover Problem. International Workshop on Combinatorial Algorithms (IWOCA 2013), 10 juillet 2013, Rouen.
- François Delbot, Christian Laforest et Raksmey Phan. Vertex Cover by Clique Partition. ECCO XXVI, 30 mai 2013, Paris.
- François Delbot, Christian Laforest et Raksmey Phan. Un nouvel algorithme d'approximation pour le vertex cover. ROADEF'2013, 13 février, Troyes.
- Christian Laforest et Raksmey Phan. Résolution exacte du Minimum Independent Dominating Set. ROADEF'2012, 14 avril 2012, Angers.

Workshops nationaux et internationaux

- Christian Laforest et Raksmey Phan. Solving the Minimum Independent Domination Set Problem in Graphs by Exact Algorithm and Greedy Heuristic : Experimental Approach. APEX 2013, 7 juillet 2013, Riga, Lettonie.
- Christian Laforest et Raksmey Phan. Experimentations of an Exact Algorithm for the Minimum Independent Dominating Set Problem in Graphs Using Clique Partition. JGA, 31 janvier 2012, Clermont-Ferrand.

Rapports de recherche

- François Delbot, Christian Laforest et Raksmey Phan. New approximation algorithms for the vertex cover problem variants. Rapport de recherche RR-13-02, Limos, Clermont Ferrand, France (2013).
- Christian Laforest et Raksmey Phan. Experimentations of an exact algorithm for the minimum in-dependent dominating set problem in graphs using clique partition. Rapport de recherche RR-12-01, Limos, Clermont Ferrand, France (2012).

Article soumis

- François Delbot, Christian Laforest, Raksmey Phan. New Approximation Algorithms for The Vertex Cover Problem and Variants With Conflicts. Discrete Applied Mathematics, numéro spécial lié à la conférence ECCO XXVI.

Voici la liste des autres travaux publiés durant ces trois années de thèse.

Conférences nationales (avec comité de lecture)

- Frédéric Chassard, Patrick Admirat, Christophe Duhamel, Philippe Lacomme et Raksmey PHAN. Intégration du problème de collecte des déchets ménagers dans le SI du SICTOM des Couzes. ROADEF'2011, 2 Mars 2011, Saint-Etienne.

Livre

- Philippe Lacomme, Raksmey Phan, Libo Ren et Nikolay Tchernev. Introduction aux nouvelles technologies d'accès aux données : Java, JEE, ASP.NET, XCode. Juillet 2012. Editeur : Ellipse.

Liste des Abréviations

$\mathcal{G}[H]$ graphe d'ensemble de sommets H et d'ensemble d'arêtes qui relient les sommets de H : $\{uv : uv \in \mathcal{E}, u \in H \text{ et } v \in H\}$

$\mathcal{S}$ solution réalisable

$\mathcal{S}^*$ solution optimale

C une partition en cliques

C_i clique d'indice i

c_i taille de la clique d'indice i

K_n grpahe complet de n sommets

n nombre de sommet du graphe $\mathcal{G}$

$\mathcal{C}$ ensemble de conflits

$CVCwnC$ vertex cover connexe sans conflit

$DwnC$ dominant sans conflit

$\mathcal{E}$ ensemble des arêtes

$\mathcal{G}$ graphe non orienté et non pondéré

IDS indépendant dominant

$IDwnC$ indépendant dominant sans conflit

$mIDS$ indépendant dominant de taille minimum

MM sous ensemble d'arêtes formant un couplage maximal

mMM sous ensemble d'arêtes formant un couplage maximal de taille minimum

mVC vertex cover de taille minimum

OPT taille de la solution optimale (de taille minimum) pour un problème donné

$STwnC$ arbre de Steiner sans conflit

VC vertex cover

$VCwnC$ vertex cover sans conflit

$\mathcal{V}$ ensemble des sommets

N(u) ensemble des voisins du sommet u

Bibliographie

[Aarts 1985] E. H. L. Aarts et P. J. M. van Laarhoven. *Statistical Cooling : A General Approach to Combinatorial Optimization Problems.* Philips Journal of Research, vol. 40, pages 193–226, 1985. (Cité en page 14.)

[Allahverdi 2006] Ali Allahverdi et Fawaz S. Al-Anzi. *A branch-and-bound algorithm for three-machine flowshop scheduling problem to minimize total completion time with separate setup times.* European Journal of Operational Research, vol. 169, no. 3, pages 767 – 780, 2006. (Cité en page 10.)

[Angel 2011] Eric Angel, Romain Campigotto et Christian Laforest. *Analysis and Comparison of Three Algorithms for the Vertex Cover Problem on Large Graphs with Low Memory Capacities.* Algorithmic Operations Research, vol. 6, no. 1, pages 56–67, 2011. (Cité en page 66.)

[Angel 2012] Eric Angel, Romain Campigotto et Christian Laforest. *Implementation and Comparison of Heuristics for the Vertex Cover Problem on Huge Graphs.* In SEA, volume 7276 of *Lecture Notes in Computer Science*, pages 39–50. Springer, 2012. (Cité en page 66.)

[Angel 2013] Eric Angel, Romain Campigotto et Christian Laforest. *A new lower bound on the independence number of graphs.* Discrete Applied Mathematics, vol. 161, no. 6, pages 847–852, 2013. (Cité en page 66.)

[Aspvall 1979] Bengt Aspvall, Michael F. Plass et Robert Endre Tarjan. *A Linear-Time Algorithm for Testing the Truth of Certain Quantified Boolean Formulas.* Inf. Process. Lett., vol. 8, no. 3, pages 121–123, 1979. (Cité en pages 101 et 102.)

[Atamtürk 2000] Alper Atamtürk, George L. Nemhauser et Martin W.P. Savelsbergh. *Conflict graphs in solving integer programming problems.* European Journal of Operational Research, vol. 121, no. 1, pages 40 – 55, 2000. (Cité en page 99.)

[Ausiello 2000] G. Ausiello, P. Crescenzi, V. Kann, Marchetti-Sp, Giorgio Gambosi et Alberto M. Spaccamela. Complexity and approximation : Combinatorial optimization problems and their approximability properties. Springer, jan 2000. (Cité en pages 22 et 26.)

[Baez-Duarte 1997] Luis Baez-Duarte. *Hardy-Ramanujan's Asymptotic Formula for Partitions and the Central Limit Theorem.* Advances in Mathematics, vol. 125, pages 114–120, 1997. (Cité en page 37.)

[Bar-Noy 1998] Amotz Bar-Noy, Mihir Bellare, Magnús M Halldórsson, Hadas Shachnai et Tami Tamir. *On Chromatic Sums and Distributed Resource Allocation.* Information and Computation, vol. 140, no. 2, pages 183 – 202, 1998. (Cité en page 11.)

[Bar-Yehuda 1985] R. Bar-Yehuda et S. Even. *A local ratio theorem for approximating the weighted vertex cover problem.* Annals of Discrete Mathematics, vol. 25, pages 27 – 45, 1985. (Cité en page 66.)

[Bellman 2003] Richard Ernest Bellman. Dynamic programming. Dover Publications, Incorporated, 2003. (Cité en page 10.)

[Birmelé 2009] Etienne Birmelé, François Delbot et Christian Laforest. *Mean analysis of an online algorithm for the vertex cover problem.* Information Processing Letters, vol. 109, no. 9, pages 436–439, 2009. (Cité en page 79.)

[Bourgeois 2009a] N. Bourgeois, F. Croce, B. Escoffier et V.Th. Paschos. *Exact Algorithms for Dominating Clique Problems.* In Yingfei Dong, Ding-Zhu Du et Oscar Ibarra, editeurs, Algorithms and Computation, volume 5878 of *Lecture Notes in Computer Science*, pages 4–13. Springer Berlin Heidelberg, 2009. (Cité en page 11.)

[Bourgeois 2009b] Nicolas Bourgeois, Bruno Escoffier et Vangelis Th. Paschos. *Fast algorithms for min independent dominating set.* CoRR, vol. abs/0905.1993, page ., 2009. (Cité en page 11.)

[Bourgeois 2010] N. Bourgeois, B. Escoffier et V. Th. Paschos. *Fast Algorithm for Min Independent Dominating Set.* SIROCCO, vol. LNCS 6058, pages 247–261, 2010. (Cité en pages 11, 30, 37 et 38.)

[Chaudhary 2001] Amitabh Chaudhary et Sundar Vishwanathan. *Approximation Algorithms for the Achromatic Number*, 2001. (Cité en page 11.)

[Cheetham 2003] James Cheetham, Frank Dehne, Andrew Rau-Chaplin, Ulrike Stege et Peter J. Taillon. *Solving large FPT problems on coarse-grained parallel machines.* Journal of Computer and System Sciences, vol. 67, no. 4, pages 691–706, 2003. (Cité en page 66.)

[Chen 2004] Xiaoming Chen, Zheng Tang, Xinshun Xu, Songsong Li, Guangpu Xia et Jiahai Wang. *An Algorithm Based on Hopfield Network Learning for Minimum Vertex Cover Problem.* In Fu-Liang Yin, Jun Wang et Chengan Guo, editeurs, Advances in Neural Networks - ISNN 2004, volume 3173 of *Lecture Notes in Computer Science*, pages 430–435. Springer Berlin Heidelberg, 2004. (Cité en page 66.)

[Chen 2010] Jianer Chen, Iyad A. Kanj et Ge Xia. *Improved upper bounds for vertex cover.* Theoretical Computer Science, vol. 411, no. 40-42, pages 3736 – 3756, 2010. (Cité en pages 10 et 66.)

[Cook 1971] Stephen A. Cook. *The complexity of theorem-proving procedures.* In Proceedings of the third annual ACM symposium on Theory of computing, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM. (Cité en page 9.)

[Cormen 1990] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest et Clifford Stein. Introduction à l'algorithmique. MIT Press, 1990. (Cité en page 85.)

[Darmann 2011] Andreas Darmann, Ulrich Pferschy, Joachim Schauer et Gerhard J. Woeginger. *Paths, trees and matchings under disjunctive constraints.* Discrete Applied Mathematics, vol. 159, no. 16, pages 1726–1735, 2011. (Cité en page 99.)

[Delbot 2008] François Delbot et Christian Laforest. *A better list heuristic for vertex cover.* Information Processing Letters, vol. 107, no. 3-4, pages 125 – 127, 2008. (Cité en pages 79 et 80.)

[Delbot 2009] F. Delbot. *Au delà de l'évaluationvaluation en pire cas : comparaison et évaluationvaluation en moyenne de processus d'optimisation pour le problème du vertex cover et des arbres de connexion de groupes dynamiques.* PhD thesis, Université d'Evry Val d'Essonne, 2009. (Cité en page 13.)

[Delbot 2010] François Delbot et Christian Laforest. *Analytical and experimental comparison of six algorithms for the vertex cover problem.* ACM, Journal of Experimental Algorithmics, vol. 15, pages 1.4 :1.1–1.4 :1.27, November 2010. (Cité en pages 66 et 84.)

[Denzinger 1997] Jorg Denzinger, Marc Fuchs et Matthias Fuchs. *High performance ATP systems by combining several AI methods.* In Proceedings of the 15th international joint conference on Artifical intelligence - Volume 1, IJCAI'97, pages 102–107, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc. (Cité en pages 13 et 14.)

[Downey 1992] Rodney G. Downey et Michael R. Fellows. *Fixed-Parameter Intractability.* In Structure in Complexity Theory Conference, pages 36–49, 1992. (Cité en page 10.)

[Downey 1995] Rod G. Downey et Michael R. Fellows. *Fixed-parameter tractability and completeness II : On completeness for $W[1]$.* Theoretical Computer Science, vol. 141, no. 1–2, pages 109–131, 1995. (Cité en page 66.)

[Edelkamp 2011] S. Edelkamp et S. Schroedl. Heuristic search : Theory and applications. Elsevier Science, 2011. (Cité en page 13.)

[Erdős 1959] P. Erdős et A. Rényi. *On random graphs. I.* Publ. Math. Debrecen, vol. 6, pages 290–297, 1959. (Cité en page 50.)

[Escoffier 2010] Bruno Escoffier, Laurent Gourvès et Jérôme Monnot. *Complexity and approximation results for the connected vertex cover problem in graphs and hypergraphs.* J. Discrete Algorithms, vol. 8, no. 1, pages 36–49, 2010. (Cité en page 95.)

[Fernau 2006] Henning Fernau et David Manlove. *Vertex and Edge Covers with Clustering Properties : Complexity and Algorithms.* In ACiD, pages 69–84, 2006. (Cité en page 95.)

[Fleury 1993] Gérard Fleury. *Méthodes stochastiques et déterministes pour les problèmes np-difficiles*. PhD thesis, Université Blaise Pascal, Clermont II, 1993. (Cité en page 13.)

[Garey 1976] M.R. Garey, D.S. Johnson et L. Stockmeyer. *Some simplified NP-complete graph problems*. Theoretical Computer Science, vol. 1, no. 3, pages 237 – 267, 1976. (Cité en pages 64 et 71.)

[Garey 1977] M. R. Garey et David S. Johnson. *The Rectilinear Steiner Tree Problem is NP-Complete*. SIAM Journal of Applied Mathematics, vol. 32, pages 826–834, 1977. (Cité en page 95.)

[Garey 1979] Michael R. Garey et David S. Johnson. Computers and intractability : A guide to the theory of NP-completeness (series of books in the mathematical sciences). W. H. Freeman & Co Ltd, first edition édition, Janvier 1979. (Cité en pages 8, 20 et 102.)

[Gaspers 2008] Serge Gaspers, Dieter Kratsch et Mathieu Liedloff. *On Independent Sets and Bicliques in Graphs*. In Hajo Broersma, Thomas Erlebach, Tom Friedetzky et Daniel Paulusma, editeurs, Graph-Theoretic Concepts in Computer Science, volume 5344 of *Lecture Notes in Computer Science*, pages 171–182. Springer Berlin Heidelberg, 2008. (Cité en page 11.)

[Gaspers 2012] Serge Gaspers, Dieter Kratsch et Mathieu Liedloff. *On Independent Sets and Bicliques in Graphs*. Algorithmica, vol. 62, no. 3-4, pages 637–658, 2012. (Cité en page 11.)

[Gusfield 1992] Dan Gusfield et Leonard Pitt. *A Bounded Approximation for the Minimum Cost 2-Sat Problem*. Algorithmica, vol. 8, no. 2, pages 103–117, 1992. (Cité en pages 101 et 102.)

[Hahn 1998] Peter Hahn, Thomas Grant et Nat Hall. *A branch-and-bound algorithm for the quadratic assignment problem based on the Hungarian method*. European Journal of Operational Research, vol. 108, no. 3, pages 629 – 640, 1998. (Cité en page 10.)

[Halldórsson 1993] Magnús M. Halldórsson. *Approximating the Minimum Maximal Independence Number*. Inf. Process. Lett., vol. 46, no. 4, pages 169–172, 1993. (Cité en pages 20 et 22.)

[Harary 1993] Frank Harary et Marilynn Livingston. *Independent domination in hypercubes*. Applied Mathematics Letters, vol. 6, no. 3, pages 27 – 28, 1993. (Cité en pages 52 et 59.)

[Håstad 1997] Johan Håstad. *Some Optimal Inapproximability Results*. In STOC, pages 1–10, 1997. (Cité en page 66.)

[Havet 2011] Frédéric Havet, Martin Klazar, Jan Kratochvil, Dieter Kratsch et Mathieu Liedloff. *Exact Algorithms for L(2,1)-Labeling of Graphs*. Algorithmica, vol. 59, no. 2, pages 169–194, 2011. (Cité en page 11.)

[Hwang 1993] R.Z. Hwang, R.C. Chang et R.C.T. Lee. *The searching over separators strategy to solve some NP-hard problems in subexponential time.* Algorithmica, vol. 9, no. 4, pages 398–423, 1993. (Cité en page 9.)

[Jansen 1997] Klaus Jansen et Sabine Öhring. *Approximation Algorithms for Time Constrained Scheduling.* Information and Computation, vol. 132, no. 2, pages 85 – 108, 1997. (Cité en page 99.)

[Johnson 1988] David S. Johnson, Christos H. Papadimitriou et Mihalis Yannakakis. *On Generating All Maximal Independent Sets.* Inf. Process. Lett., vol. 27, no. 3, pages 119–123, 1988. (Cité en page 29.)

[Kann 1992] Viggo Kann. *On the approximability of NP-complete optimization problems.* PhD thesis, Royal Institute of Technology Stockholm, 1992. (Cité en page 11.)

[Kanté 2013] Mamadou Moustapha Kanté, Christian Laforest et Benjamin Momège. *An Exact Algorithm to Check the Existence of (Elementary) Paths and a Generalisation of the Cut Problem in Graphs with Forbidden Transitions.* In SOFSEM, volume 7741 of *Lecture Notes in Computer Science*, pages 257–267. Springer, 2013. (Cité en page 99.)

[Karakostas 2009] George Karakostas. *A better approximation ratio for the vertex cover problem.* ACM Transactions on Algorithms, vol. 5, pages 41 :1–41 :8, November 2009. (Cité en page 66.)

[Karp 1972] Richard M. Karp. *Reducibility Among Combinatorial Problems.* In Complexity of Computer Computations, pages 85–103, 1972. (Cité en page 64.)

[Khot 2008] Subhash Khot et Oded Regev. *Vertex cover might be hard to approximate to within* $2-\varepsilon$. Journal of Computer and System Sciences, vol. 74, no. 3, pages 335–349, 2008. (Cité en page 66.)

[Khuri 1994] Sami Khuri et Thomas Bäck. *An Evolutionary Heuristic for the Minimum Vertex Cover Problem.* In KI-94 Workshops (Extended Abstracts, pages 86–90, 1994. (Cité en page 66.)

[Knuth 1968] Donald E. Knuth. The art of computer programming, volume i : Fundamental algorithms. Addison-Wesley, 1968. (Cité en page 4.)

[Kotecha 2003] Ketan Kotecha et Nilesh Gambhava. *A Hybrid Genetic Algorithm for Minimum Vertex Cover Problem.* In In Prasad, B. (Ed.), The First Indian International Conference on Artificial Intelligence, pages 904–913, 2003. (Cité en page 66.)

[Krom 1967] M. R. Krom. *The Decision Problem for a Class of First-Order Formulas in Which all Disjunctions are Binary.* Mathematical Logic Quarterly, vol. 13, no. 1-2, pages 15–20, 1967. (Cité en pages 101 et 102.)

[Kuhn 2005] Fabian Kuhn, Tim Nieberg, Thomas Moscibroda et Roger Wattenhofer. *Local approximation schemes for ad hoc and sensor networks*. In DIALM-POMC, pages 97–103, 2005. (Cité en pages 20 et 99.)

[Land 1960] A. H. Land et A. G Doig. *An Automatic Method of Solving Discrete Programming Problems*. Econometrica, vol. 28, no. 3, pages 497–520, 1960. (Cité en page 10.)

[Lehot 1974] Philippe G. H. Lehot. *An Optimal Algorithm to Detect a Line Graph and Output Its Root Graph*. J. ACM, vol. 21, no. 4, pages 569–575, 1974. (Cité en pages 23 et 24.)

[Liu 2006] C. Liu et Y. Song. *Exact Algorithms for Finding the Minimum Independent Dominating Set in Graphs*. ISAAC, vol. LNCS 4288, pages 439–448, 2006. (Cité en pages 29, 30, 37, 39 et 53.)

[Menon 1965] V. V. Menon. *The isomorphism between graphs and their adjoint graphs*. j-CAN-MATH-BULL, vol. 8, pages 7–16, 1965. (Cité en page 22.)

[Mujuni 2008] Egbert Mujuni et Frances Rosamond. *Parameterized complexity of the clique partition problem*. In Proceedings of the fourteenth symposium on Computing : the Australasian theory - Volume 77, CATS '08, pages 75–78, Darlinghurst, Australia, Australia, 2008. Australian Computer Society, Inc. (Cité en page 10.)

[Ore 1962] O. Ore. Theory of graphs. American Mathematical Society : Colloquium publications. American Mathematical Society, 1962. (Cité en page 22.)

[Papadimitriou 1982] Christos H. Papadimitriou et Kenneth Steiglitz. Combinatorial optimization : Algorithms and complexity. Prentice-Hall, 1982. (Cité en page 66.)

[Pferschy 2009] Ulrich Pferschy et Joachim Schauer. *The Knapsack Problem with Conflict Graphs*, 2009. (Cité en page 99.)

[Plummer 1986] D. Plummer et L. Lovász. Matching theory. Numeéro n°121 de Annals of discrete mathematics. Elsevier Science, 1986. (Cité en page 82.)

[Potluri 2011] Anupama Potluri et Atul Negi. *Some Observations on Algorithms for Computing Minimum Independent Dominating Set*. In IC3, volume 168 of *Communications in Computer and Information Science*, pages 57–68. Springer, 2011. (Cité en pages 30 et 53.)

[Roth-Korostensky 2000] Chantal Roth-Korostensky. *Algorithms for building multiple sequence alignments and evolutionary trees*. PhD thesis, Zürich, ETH, Zürich, 2000. Diss. : technical sciences. (Cité en pages 64 et 99.)

[Roussopoulos 1973] Nick Roussopoulos. *A MAX{m, n} Algorithm for Determining the Graph H from Its Line Graph C*. Inf. Process. Lett., vol. 2, no. 4, pages 108–112, 1973. (Cité en page 24.)

[Savage 1982] Carla Savage. *Depth-First Search and the Vertex Cover Problem.* Information Processing Letters, vol. 14, no. 5, pages 233–235, Juillet 1982. (Cité en pages 66, 84 et 95.)

[Seshu 1961] S. Seshu. Linear graphs and electrical networks. Addison-Wesley, 1961. (Cité en page 22.)

[Shyu 2004] ShyongJian Shyu, Peng-Yeng Yin et BertrandM.T. Lin. *An Ant Colony Optimization Algorithm for the Minimum Weight Vertex Cover Problem.* Annals of Operations Research, vol. 131, no. 1-4, pages 283–304, 2004. (Cité en page 66.)

[Snyman 2005] J. Snyman. Practical mathematical optimization : An introduction to basic optimization theory and classical and new gradient-based algorithms. Applied Optimization. Springer, 2005. (Cité en page 14.)

[Steele 2004] J.M. Steele. The cauchy-schwarz master class : an introduction to the art of mathematical inequalities. MAA problem books series. Cambridge University Press, 2004. (Cité en page 34.)

[Stege 2000] Ulrike Stege. *Resolving conflicts in problems from computational biology.* PhD thesis, Technische Wissenschaften ETH Zürich, Zürich, 2000. (Cité en pages 64 et 99.)

[Szeider 2003] Stefan Szeider. *Finding paths in graphs avoiding forbidden transitions.* Discrete Applied Mathematics, vol. 126, no. 2-3, pages 261–273, 2003. (Cité en page 99.)

[Troya 1989] J.M Troya et M Ortega. *A study of parallel branch-and-bound algorithms with best-bound-first search.* Parallel Computing, vol. 11, no. 1, pages 121 – 126, 1989. (Cité en page 10.)

[Turing 1936] Alan M. Turing. *On Computable Numbers, with an application to the Entscheidungsproblem.* Proceedings of the London Mathematical Society, vol. 42, pages 230–265, 1936. (Cité en page 4.)

[Vazirani 2001] Vijay V. Vazirani. Approximation algorithms. Springer, 2001. (Cité en pages 12, 13, 66, 84 et 86.)

[Xu 2006] Xinshun Xu et Jun Ma. *An efficient simulated annealing algorithm for the minimum vertex cover problem.* Neurocomputing, vol. 69, no. 7-9, pages 913 – 916, 2006. New Issues in Neurocomputing : 13th European Symposium on Artificial Neural Networks</ce :title> <xocs :full-name>13th European Symposium on Artificial Neural Networks 2005. (Cité en page 66.)

[Yannakakis 1980] Mihalis Yannakakis et Fanica Gavril. *Edge dominating sets in graphs.* SIAM J. Appl. Math., vol. 38, pages 364–372, 1980. (Cité en pages 11, 22 et 26.)

[Yuan 1998] Shih-Yi Yuan et Sy-Yen Kuo. *A new technique for optimization problems in graph theory.* Computers, IEEE Transactions on, vol. 47, no. 2, pages 190–196, 1998. (Cité en page 66.)

[Zhang 2006] Yong Zhang, Qi Ge, Rudolf Fleischer, Tao Jiang et Hong Zhu. *Approximating the minimum weight weak vertex cover.* Theoretical Computer Science, vol. 363, no. 1, pages 99 – 105, 2006. (Cité en pages 64 et 99.)

[Zhang 2011] Ruonan Zhang, Santosh N. Kabadi et Abraham P. Punnen. *The minimum spanning tree problem with conflict constraints and its variations.* Discrete Optimization, vol. 8, no. 2, pages 191 – 205, 2011. (Cité en page 99.)