Investigating decomposition methods for the maximum common subgraph and sum colouring problems

Maël Minot

THÈSE DE DOCTORAT DE L'UNIVERSITÉ DE LYON
opérée au sein de l'INSA de Lyon

École doctorale nº 512
InfoMaths

Spécialité de doctorat :
Informatique

Soutenue publiquement le 19/12/2017, par

Maël MINOT

Investigating decomposition methods for the maximum common subgraph and sum colouring problems

---

Devant le jury composé de :

M. Simon DE GIVRY · Centre de recherches de Toulouse, INRA	Rapporteur
M. Christophe LECOUTRE · IUT de Lens	Examinateur
M. Chu Min LI · Université de Picardie Jules VERNE	Rapporteur
M. Samba Ndojh NDIAYE · Université Lyon 1, LIRIS	Examinateur (Encadrant)
Mme Christine SOLNON · INSA Lyon, LIRIS	Directrice de thèse

Département FEDORA – INSA Lyon – Écoles doctorales
Quinquennal 2016–2020

SIGLE	ÉCOLE DOCTORALE	NOM ET COORD. RESPONSABLE
CHIMIE	**CHIMIE DE LYON** http://www.edchimie-lyon.fr Sec. : Renée EL MELHEM Bât. Blaise PASCAL, 3e étage **secretariat@edchimie-lyon.fr** INSA : R. GOURDON	M. Stéphane DANIELE Institut de recherches sur la catalyse et l'environnement de Lyon IRCELYON-UMR 5256 Équipe CDFA 2 Avenue Albert EINSTEIN 69 626 Villeurbanne CEDEX **directeur@edchimie-lyon.fr**
E.E.A	**ÉLECTRONIQUE, ÉLECTROTECHNIQUE, AUTOMATIQUE** http://edeea.ec-lyon.fr Sec. : M.C. HAVGOUDOUKIAN **ecole-doctorale.eea@ec-lyon.fr**	M. Gérard SCORLETTI École Centrale de Lyon 36 Avenue Guy DE COLLONGUE 69 134 Écully ☎ 04.72.18.60.97 🖷 04.78.43.37.17 **gerard.scorletti@ec-lyon.fr**
E2M2	**ÉVOLUTION, ÉCOSYSTÈME, MICROBIOLOGIE, MODÉLISATION** http://e2m2.universite-lyon.fr Sec. : Sylvie ROBERJOT Bât. Atrium, UCB Lyon 1 ☎ 04.72.44.83.62 INSA : H. CHARLES **secretariat.e2m2@univ-lyon1.fr**	M. Fabrice CORDEY CNRS UMR 5276 Lab. de géologie de Lyon Université Claude Bernard Lyon 1 Bât. Géode 2 Rue Raphaël DUBOIS 69 622 Villeurbanne CEDEX ☎ 06.07.53.89.13 **cordey@univ-lyon1.fr**
EDISS	**INTERDISCIPLINAIRE SCIENCES-SANTÉ** http://www.ediss-lyon.fr Sec. : Sylvie ROBERJOT Bât. Atrium, UCB Lyon 1 ☎ 04.72.44.83.62 INSA : M. LAGARDE **secretariat.ediss@univ-lyon1.fr**	Mme Emmanuelle CANET-SOULAS INSERM U1060, CarMeN lab, Univ. Lyon 1 Bâtiment IMBL 11 Avenue Jean CAPELLE INSA de Lyon 69 621 Villeurbanne ☎ 04.72.68.49.09 🖷 04.72.68.49.16 **emmanuelle.canet@univ-lyon1.fr**
INFOMATHS	**INFORMATIQUE ET MATHÉMATIQUES** http://edinfomaths.universite-lyon.fr Sec. : Renée EL MELHEM Bât. Blaise PASCAL, 3e étage ☎ 04.72.43.80.46 🖷 04.72.43.16.87 **infomaths@univ-lyon1.fr**	M. Luca ZAMBONI Bât. Braconnier 43 Boulevard du 11 novembre 1918 69 622 Villeurbanne CEDEX ☎ 04.26.23.45.52 **zamboni@maths.univ-lyon1.fr**
MATÉRIAUX	**MATÉRIAUX DE LYON** http://ed34.universite-lyon.fr Sec. : Marion COMBE ☎ 04.72.43.71.70 🖷 04.72.43.87.12 Bât. Direction **ed.materiaux@insa-lyon.fr**	M. Jean-Yves BUFFIÈRE INSA de Lyon MATEIS Bât. Saint-Exupéry 7 Avenue Jean CAPELLE 69 621 Villeurbanne CEDEX ☎ 04.72.43.71.70 🖷 04.72.43.85.28 **ed.materiaux@insa-lyon.fr**
MEGA	**MÉCANIQUE, ÉNERGÉTIQUE, GÉNIE CIVIL, ACOUSTIQUE** http://edmega.universite-lyon.fr Sec. : Marion COMBE ☎ 04.72.43.71.70 🖷 04.72.43.87.12 Bât. Direction **mega@insa-lyon.fr**	M. Philippe BOISSE INSA de Lyon Laboratoire LAMCOS Bâtiment Jacquard 25 bis Avenue Jean CAPELLE 69 621 Villeurbanne CEDEX ☎ 04.72.43.71.70 🖷 04.72.43.72.37 **philippe.boisse@insa-lyon.fr**
SCSO	**SCSO** * http://ed483.univ-lyon2.fr Sec. : Viviane POLSINELLI Brigitte DUBOIS INSA : J.Y. TOUSSAINT ☎ 04.78.69.72.76 **viviane.polsinelli@univ-lyon2.fr**	M. Christian MONTES Université Lyon 2 86 Rue Pasteur 69 365 Lyon CEDEX 07 **christian.montes@univ-lyon2.fr**

* ScSo : Histoire, Géographie, Aménagement, Urbanisme, Archéologie, Science politique, Sociologie, Anthropologie.

Thanks (*Version française plus bas*)

I am not really used to thanking people. I think that I perceive as some kind of weakness the feeling of owing something to someone. Truth be told, there are quite a lot of things that I see as weaknesses – even the ability to forgive, for example. And when you start showing weaknesses to the world, you expose yourself to judgement. Yet another thing that I hate. It would be far easier to address my thanks to my stuffed animals, or even to my gaming consoles; it sure would prove to be less hazardous. But there it is: sometimes, one needs to put his beliefs and habits aside, especially the bad ones.

Facing the eventuality of making this section predictable, I will start by thanking my advisers: Christine, who seemed to agree to take me as her student without real concern despite the lack of information. She always knew how to make sure we went forward, even when a few painful detours were required in order to eventually provide some rigorous and thorough work. Samba, then, who first chose me as his intern at the end of my master's degree, even though I felt like I was just some random person, with my doubts for only luggage. More than once, he put me back on track – often as if involuntarily or by nature – when I was feeling down due to some particular task happening to be more challenging than the others.

I should also thank the other members of my jury. Having spent long and painful nights proofreading my thesis, I cannot help but be surprised at the thought that people actually volunteered to do such a thing.

Additional thanks go to my colleagues, even though I have always felt that this term seems oddly inappropriate to designate PhD students. Bearing with my continued presence in the office must have been hard, sometimes. Thanks, too, to all those people I kept running into while walking the corridors, even though I never could remember most of your names.

Thanks to these friends that I see far from often enough but that definitely still exist, somewhere.

Thanks to my family that offers me some kind of vital drop point. It was nice being able to spend my Sundays baking cakes for my colleagues for a change of pace.

Thanks to my flatmates, past and present, who help me explore the real and social world, with its assets as well as its bad aspects. Six years living on my own was probably too much.

Thanks to those people of all kinds that I started meeting a few months ago. I still think that gathering in noisy places to chat is utterly absurd and blatantly suboptimal, but at least it helps me to clear my mind, and maybe to become someone better.

Ultimately, I think that these three years can be considered the most eventful of the beginning of my existence. I learned as much about the world and myself than about computer science. More than ever, I am eager to find out what lies in my future, even if, undoubtedly, a few ordeals will come to rise now and then.

Maël MINOT, October 2017

Remerciements

Je n'ai pas vraiment l'habitude de remercier des gens. Je crois que je vois comme une faiblesse le fait d'avoir le sentiment de devoir quelque chose à qui que ce soit. À vrai dire, je vois beaucoup de choses comme des faiblesses – même la faculté de pardonner à quelqu'un, par exemple. Et lorsque l'on commence à montrer des faiblesses, on prend le risque d'être jugé. Encore une chose que je déteste. Il me serait plus facile de remercier mes animaux en peluche ou même ma console de jeux ; cela serait moins dangereux. Mais voilà : il faut parfois mettre de côté ses croyances et ses habitudes, surtout les mauvaises.

Au risque de rendre cette section banale, je commencerai par remercier mes deux encadrants : Christine, qui semble avoir accepté sans trop hésiter de me prendre en charge en ne disposant pas de tant d'information que ça. Elle a toujours su veiller à ce que les choses aillent de l'avant, quitte à devoir faire quelques détours difficiles mais nécessaires pour présenter un travail rigoureux et complet. Samba, enfin, qui m'a tout d'abord pris comme stagiaire alors que j'avais le sentiment de sortir un peu de nulle part, avec mes doutes pour seuls bagages. Il m'a maintes fois remis moralement d'aplomb, bien souvent en semblant ne pas le faire vraiment exprès, lorsque je me posais les mauvaises questions face à des tâches plus ardues que d'ordinaire.

Merci également à mes rapporteurs. Après avoir peiné des nuits durant pour relire cette thèse, je suis assez étonné à l'idée que des gens se portent volontaires pour de telles choses.

Merci à mes collègues, bien que ce terme me semble toujours étrangement inadapté pour désigner des thésards. Supporter ma présence dans un bureau à longueur de journée n'a pas dû être toujours facile. Merci même à tous ces gens que je croisais dans les couloirs et dont je n'ai jamais réussi à retenir le nom.

Merci à mes amis, que je ne vois plus du tout assez souvent, mais qui existent encore bel et bien.

Merci à ma famille, qui m'offre un point de chute parfois capital. J'ai ainsi pu passer mes dimanches à me concentrer sur les gâteaux que je préparais pour mes collègues.

Merci à mes colocataires, anciens comme actuels, qui m'aident à explorer le monde réel et social, avec ses avantages comme ses inconvénients. Six ans à vivre seul, c'était probablement un peu trop.

Merci à ces gens variés et étonnants que j'ai commencé à rencontrer il y a quelques mois. Je trouve toujours que se retrouver dans des lieux bruyants pour parler est complètement absurde et sous-optimal, mais ça me change les idées et m'aide à me construire.

En définitive, je crois que ces trois années ont été les plus riches en événements de ce début d'existence. J'en ai appris autant sur le monde et sur moi-même que sur l'informatique. Plus que jamais, je suis curieux de découvrir de quoi l'avenir sera fait, quitte à affronter quelques épreuves au passage.

Maël MINOT, octobre 2017

d

Abstract

The objective of this thesis is, from a general standpoint, to design and evaluate decomposition methods for solving constrained optimisation problems. Two optimisation problems in particular are considered: the *maximum common induced subgraph problem*, in which the largest common part between two graphs is to be found, and the *sum colouring problem* [KS89], where a graph must be coloured in a way that minimises a sum of weights induced by the employed colours.

The maximum common subgraph (MCIS) problem is notably difficult, with a strong applicability in domains such as biology, chemistry and image processing, where the need to measure the similarity between structured objects represented by graphs may arise. The outstanding difficulty of this problem makes it strongly advisable to employ a decomposition method, possibly coupled with a parallelisation of the solution process. However, existing decomposition methods are not well suited to solve the MCIS problem: some lead to a poor balance between subproblems, while others, like tree decomposition, are downright inapplicable.

To enable the structural decomposition of such problems, CHMEISS et al. proposed an approach, TR-decomposition, acting at a low level: the *microstructure* of the problem. This approach had yet to be applied to the MCIS problem. We evaluate it in this context, aiming at reducing the size of the search space while also enabling parallelisation. Moreover, we introduce a post-decomposition step that focuses on alleviating redundancies between subproblems. Our experiments show that the time employed to decompose the initial problem eventually proves to be profitable.

The second problem that caught our interest is the sum colouring problem. It is an $\mathcal{NP}$-hard variant of the widely known classical graph colouring problem. As in most colouring problems, it basically consists in assigning colours to the vertices of a given graph while making sure no neighbour vertices use the same colour. In the sum colouring problem, however, each colour is associated with a weight. The objective is to minimise the sum of the weights of the colours used by every vertex. This leads to generally harder instances than the classical colouring problem, which simply requires to use as few colours as possible.

Only a few exact methods have been proposed for this problem. Among them stand notably a constraint programming (CP) model, a branch and bound approach, as well as an integer linear programming (ILP) model.

We led an in-depth investigation of CP's capabilities to solve the sum colouring problem, while also looking into ways to make it more efficient. These experiments proved that even though CP seems, at first glance, to be at a disadvantage, it still has some assets, notably its reduced memory needs.

Additionally, we evaluated a combination of integer linear programming and constraint programming, with the intention of conciliating the strong points of these highly complementary approaches. We took inspiration from the classical *backtracking bounded by tree decomposition* (BTD) approach [JT03]. We employ a tree decomposition with a strictly bounded height. Constraint programming is used to enumerate consistent assignment of the root cluster. For each of these assignments, one subproblem is trivially obtained for each leaf cluster. Subproblems are then independently solved to optimality using ILP. This combination of CP and ILP yields interesting results and offers interesting improvement perspectives, notably by computing partial solutions during a preprocessing step in order to obtain bounds.

We then derive profit from the complementarity of our approaches by developing a *portfolio approach*. The resulting solver is able to choose one of the considered approaches automatically by relying on a number of features extracted from each instance. This approach gives encouraging results.

Résumé

L'objectif de cette thèse est, d'un point de vue général, de concevoir et d'évaluer des méthodes de décomposition applicables à des problèmes d'optimisation sous contraintes. Deux problèmes d'optimisation, en particulier, ont été considérés : le problème du plus grand sous-graphe commun, qui consiste à trouver la plus grande partie commune entre deux graphes, et le problème de la somme coloration, dans lequel un graphe doit être colorié d'une façon minimisant une somme de poids.

Le problème du plus grand sous-graphe commun est connu pour être particulièrement difficile à résoudre. Il survient dans de nombreux domaines applicatifs tels que la biologie, la chimie, ou encore le traitement d'images, où il est parfois nécessaire de mesurer le degré de similarité entre des objets structurés pouvant être représentés par des graphes.

La programmation par contraintes se montre compétitive pour résoudre ce problème, de même que des approches basées sur une reformulation du problème en instance du problème de la recherche d'une clique maximum. Cependant, la difficulté considérable du problème du plus grand sous-graphe commun rend fortement souhaitable d'employer une méthode de décomposition, couplée à une parallélisation de l'étape de résolution, sous peine de ne pas pouvoir traiter des instances d'une taille significative.

Une telle approche a été proposée par DEPOLLI et MCCREESH pour le problème de la clique maximum, tandis que RÉGIN et al. ont œuvré pour décomposer les problèmes de satisfaction de contraintes d'un point de vue plus général en subdivisant de manière répétée les domaines de certaines variables. Cette dernière approche est connue pour sa grande efficacité, mais se trouve être peu adaptée au problème du plus grand sous-graphe commun.

En effet, elle traite toutes les valeurs de la même manière, et le modèle de programmation par contraintes du problème qui nous intéresse comprend une valeur spéciale qui, lorsqu'elle est utilisée, marque le fait qu'un sommet ne figure pas dans le sous-graphe commun en cours de construction. Il en résulte des sous-problèmes de tailles très peu équilibrées lorsqu'une technique de division de domaines classique est employée.

Une manière alternative de décomposer un problème consiste à exploiter sa structure. Les problèmes de satisfaction de contraintes ou d'optimisation sous contraintes, en particulier, peuvent être traités au moyen d'une décomposition arborescente de leur graphe de contraintes. Cependant, dans de nombreux contextes dont celui du problème du plus grand sous-graphe commun, cette technique n'est pas applicable, car l'une des contraintes englobe l'intégralité des variables du problème. Dans de tels cas, toute tentative de décomposition arborescente ne donnera qu'un unique groupe de variables dont le traitement sera équivalent à la résolution du problème initial, ce qui rend une telle décomposition complètement inutile.

Afin de contourner ce problème et de permettre une certaine forme de décomposition dans ces cas problématiques, CHMEISS et al. ont proposé une approche alternative, qui agit à un plus bas niveau. Plus précisément, l'objet de la décomposition n'est plus le graphe de contraintes mais la microstructure du problème, plus grande et plus détaillée. De plus, cette méthode produit des sous-problèmes parfaitement indépendants, facilitant ainsi la parallélisation du processus de résolution.

Cette approche n'avait encore jamais été employée pour résoudre le problème du plus grand sous-graphe commun. Nous l'avons de ce fait évaluée dans ce contexte, en tentant de réduire la taille de l'espace de recherche global et de résoudre les sous-problèmes en parallèle. Nous avons également ajouté, après la phase de décomposition, une étape visant à réduire l'impact des redondances existant entre certains des sous-problèmes générés par cette méthode.

Notre évaluation confirme que cette approche est adaptée à ce problème, notamment lorsque croît la difficulté des instances considérées, car le temps dépensé pour décomposer le problème est alors mieux rentabilisé. Nous proposons également de décomposer récursivement les plus gros sous-problèmes afin d'obtenir des sous-problèmes encore plus équilibrés et ainsi de mieux répartir la charge de travail sur les unités de calcul disponibles. Nos résultats sont comparés à ceux pouvant être obtenus avec la méthode de division de domaines de RÉGIN et al. Il est apparu que notre approche rentabilisait mieux l'usage de nombreuses unités de calcul.

Le second problème sur lequel nous avons travaillé est celui de la somme coloration. Il s'agit d'une variante $\mathcal{NP}$-difficile du très classique problème de coloration de graphe. Comme dans la plupart des problèmes de coloration, il s'agit d'affecter des couleurs aux sommets d'un graphe en s'assurant que les sommets voisins ne partagent pas la même couleur. Dans le problème de somme coloration, cependant, chaque couleur est associée à un poids. L'objectif

n'est plus de minimiser le nombre total de couleurs employées mais d'obtenir la plus petite somme de poids possible sur l'ensemble des sommets. Il en résulte que, pour un même graphe, ce problème est généralement bien plus difficile à résoudre que le problème de coloration classique.

Ce problème trouve des échos dans des situations concrètes d'allocation de ressources et de planifications, dans lesquelles le voisinage des sommets dénote certaines incompatibilités et conflits, et où les poids associés aux couleurs introduisent une notion de coûts associés aux ressources.

Le problème de somme coloration n'a pas encore été largement étudié. En partie pour cette raison, la plupart des approches proposées jusqu'à maintenant sont des heuristiques, inaptes à assurer l'optimalité d'une solution dans le cas général. Parmi les rares approches complètes présentées dans la littérature, on notera notamment l'existence d'un modèle de programmation par contraintes et d'une approche homologue utilisant la programmation linéaire, ainsi que d'une méthode basée sur le principe du *branch and bound* (séparation et évaluation).

Le modèle de programmation par contraintes existant s'est avéré être assez élémentaire, ce qui menait à un certain manque de compétitivité. Nous avons entrepris de le rendre plus efficace tout en menant une évaluation assez poussée de ses facultés à résoudre le problème de somme coloration.

Nous avons pu prouver que bien que la programmation par contraintes semble désavantagée dans ce contexte, elle possède certains avantages, en particulier ses faibles besoins en mémoire, notamment par rapport à la programmation linéaire. Un solveur tel que `Gecode` peut ainsi mener une longue recherche et trouver des solutions de qualité, même sur de très grandes instances.

La compétitivité d'un tel solveur peut être fortement améliorée en procédant à un paramétrage minutieux, comme par exemple en instaurant une politique de *restarts* ou en déterminant quelles heuristiques sont les plus adaptées pour décider quelles valeurs et variables seront traitées en priorité. De plus, calculer des bornes de qualité peut permettre de supprimer des branches de l'arbre de recherche.

En complément de ces travaux, nous avons développé et évalué une méthode combinant la programmation par contraintes et la programmation linéaire, dans le but de concilier la robustesse et l'efficacité. Cette approche s'inspire du *backtracking bounded by tree decomposition* (BTD, retour sur trace borné par une décomposition arborescente) de JÉGOU et TERRIOUX, qui tire partie de l'indépendance de certaines parties du problème identifiées lors d'une phase de décomposition préalable.

Nous employons, de même, une décomposition arborescente, mais nous bornons fortement sa hauteur, afin de forcer chaque groupe de variables à être soit la racine de l'arbre, soit une de ses feuilles. Nous utilisons ensuite la programmation par contraintes pour énumérer toutes les affectations cohérentes de la racine. De chacune de ces affectations découle de manière triviale un

sous-problème par feuille de la décomposition. Ces sous-problèmes sont ensuite indépendamment résolus à l'optimum via un modèle de programmation linéaire.

Cette combinaison de méthodes donne des résultats plus satisfaisants que l'emploi systématique de la programmation par contraintes ou de la programmation linéaire, et surpasse également, dans ce contexte, l'application classique de BTD. De plus, elle offre d'intéressantes perspectives d'amélioration. Nous explorons en particulier la possibilité de calculer des solutions partielles sur les différentes parties désignées par la décomposition. Nous en tirons des bornes locales aidant à la résolution, puis nous combinons ces solutions partielles afin d'obtenir une première solution globale. Ces nouvelles données permettent de faciliter la résolution proprement dite.

Après avoir constaté une grande complémentarité parmi les principales méthodes étudiées pour cette thèse, nous avons entrepris d'en combiner certaines au sein d'une approche portfolio. Le solveur qui en résulte est capable, après une phase d'apprentissage, de choisir automatiquement l'une des méthodes du portfolio en fonction d'attributs extraits de chaque instance considérée.

Cette approche a donné des résultats encourageants, en obtenant des solutions d'une qualité comparable à celle offerte par les bornes fournies par la programmation par contraintes et en menant à bien autant de preuves d'optimalité qu'une approche basée sur la programmation linéaire, et ce sans rencontrer de problèmes dus à un manque de mémoire.

Contents

Introduction

MANY of the research problems considered by the artificial intelligence and operational research communities are $\mathcal{NP}$-hard optimisation problems. For most of them, the sheer number of possibilities that must be evaluated in order to find the optimal solution and to prove its optimality makes it mandatory to employ more elaborate methods in order to speed up the solution process.

Among the most common ways to alleviate this difficulty, decomposition methods are especially appealing. Following the classical concept of "divide and conquer", they allow to split a given problem into subproblems. Desirable properties for these subproblems include independence – so that they can be solved in parallel – and a somewhat lower difficulty with regards to the initial problem.

Several approaches following these ideas have been recently proposed in the context of constraint satisfaction and constrained optimisation problems. Two in particular are worth mentioning:

Backtracking bounded by tree decomposition (BTD) The BTD approach consists, firstly, in decomposing the constraint graph of the problem into a set of variable clusters organized in the shape of a tree. Its efficiency is strongly tied with the inherent structure of the problem: the more structured an instance is, the better the tree decomposition. The tree decomposition is then used to guide the search and to derive profit from the independence of certain parts of the problem, identified during the decomposition step.

Embarrassingly parallel search (EPS) EPS is a domain splitting technique developed for constraint satisfaction problems in general. It pre-assigns values to a subset of the problem's variables, each different assignment then being subject to constraint propagation and leading to a separate, independent subproblem. It can be employed to generate a very large number of subproblems, thus making it easier to balance the workload between processing units during a parallel resolution.

The objective of this thesis is, from a general standpoint, to design and evaluate decomposition methods for solving constrained optimisation problems. Two optimisation problems in particular have been considered: the *maximum common induced subgraph problem*, in which the largest common part between two objects is to be found, and the *sum colouring problem* [KS89], where a

graph must be coloured in a way that minimises a sum of weights. Indeed, the aforementioned existing decomposition methods appear to be rather inefficient for these problems.

The maximum common subgraph problem

The maximum common subgraph (MCIS) problem is notably difficult, with a strong applicability in domains such as biology, chemistry and image processing, where the need to measure the similarity between structured objects represented by graphs may arise.

Even though EPS is known to be highly efficient in general, it adapts poorly to the MCIS problem, due to one peculiarity of the constraint programming (CP) model commonly used for this problem: each variable can take a special value meaning that the corresponding vertex does not belong to the current common subgraph. EPS handles every value and subproblem in the same way, and this results in a very unsatisfying workload balance for the MCIS problem.

Moreover, BTD cannot even be applied to the MCIS problem to begin with. Indeed, in the constraint programming model designed to solve the MCIS problem, one of the constraints affects every single variable. In such occurrences, any tree decomposition attempt will only yield a decomposition comprised of a sole large cluster, meaning that the resulting problem will be completely equivalent to the initial one, rendering this decomposition method useless.

To enable the structural decomposition of such problems despite this peculiarity, CHMEISS et al. proposed an alternative approach acting at a lower level. It is based on the *microstructure* of the constraint satisfaction problem rather than on its constraint graph [CJK03]. The subproblems generated by this technique are perfectly independent, and can thus be solved in parallel without particular precautions.

This approach had yet to be applied to the MCIS problem. We evaluate it in this context, aiming at reducing the overall size of the search space that must be explored during the solution process, while also enabling parallelisation. Moreover, we added a post-decomposition step that focuses on alleviating redundancies between subproblems.

This work has been subject to the following publications:

- Workshop “Bridging the gap between theory and practice in constraint solvers” at CP 2014 [MN14];
- JFPC 2015 [MNS15B] (in French);
- ICTAI 2015 [MNS15A].

The sum colouring problem

The sum colouring problem is an $\mathcal{NP}$-hard variant of the widely known classical graph colouring problem. As in most colouring problems, it basically consists

in assigning colours to the vertices of a given graph while making sure no neighbour vertices use the same colour. In the sum colouring problem, however, each colour is associated with a weight. The objective is to minimise the sum of the weights of the colours used by every vertex. This leads to generally harder instances than the classical colouring problem, which simply requires to use as few colours as possible.

This problem is amply reminiscent of concrete situations occurring in domains such as resource allocation problems or scheduling. Colouring problems, in general, have a tight relationship with timetabling problems and the like. The weights introduced by the sum colouring problem add a notion of costs and preferences to such contexts.

The sum colouring problem has not been extensively studied yet. Moreover, most of the approaches suggested in the literature are heuristics and metaheuristics – methods that cannot guaranty the optimality of a solution in the general case. Only a few exact methods have been proposed. Among them stand notably a constraint programming model, a branch and bound approach [LEC+15A], as well as an integer linear programming (ILP) model [WAN+12].

The existing sum colouring CP model appeared to be a rather straightforward one, and its performances were not as convincing as one could have expected. Moreover, it had yet to be evaluated at a reasonable scale. We therefore lead an in-depth investigation of its capabilities to solve the sum colouring problem, while also looking into ways to make it more efficient.

BTD can be used to solve the sum colouring problem. However, since this is an optimisation problem, tedious enumerations must take place within each non-leaf cluster of the employed tree decomposition. This leads to significant losses of time and prevents BTD from being competitive for this problem.

We investigate in this thesis ways to make all these approaches more efficient to solve the sum colouring problem. In particular, we design a new decomposition approach, BFD (for "backtracking bounded by flower decomposition"), inspired by them. BFD employs a tree decomposition, just like BTD. However, the decomposition we use has a strictly bounded height, forcing every cluster of the tree to be either the root itself or a leaf cluster. During the solution process, constraint programming is used to enumerate every consistent assignment of the root cluster. For each of these assignments, one subproblem is trivially obtained for each leaf cluster. Subproblems are then independently solved to optimality using integer linear programming techniques. BFD thus conciliates the strong points of these highly complementary approaches and provides a middle ground between robustness and speed.

The set of approaches comprised of CP strategies, ILP ones and BFD exhibited great complementarity, with various strong points as well as distinct weaknesses. We derive profit from these aspects by developing a *portfolio approach*. The resulting solver is able to chose one of the considered approaches automatically by relying on a number of features extracted within reasonable time from each instance.

Our work on the sum colouring problem was published in the following conferences:

- JFPC 2016 [MNS16A] (in French);
- Doctoral program in CP 2016 [MNS16B];
- CPAIOR 2017 [MNS17].

Outline

This thesis will start with a part providing information on the context, with important definitions and concepts. Part II covers everything related to the first problem at hand, namely the maximum common subgraph problem, while Part III is concerned with the sum colouring problem. For each of these problems, existing approaches will be discussed, and improvements as well as new techniques will be presented. Lastly, a global conclusion will recapitulate the contents of this thesis.

ൿ Part I ൿ

Context

This part introduces all the notions that are needed to fully apprehend this thesis.

It begins with presenting the basics of graph theory in Chapter 1, since the two problems addressed in this thesis both involve graphs. We will of course also present more advanced concepts that are closely related to the problems that will be introduced later on.

The main formalisms, namely constraint programming (CP) and integer linear programming (ILP), that we used to solve the considered problems will be described in details through Chapters 2 and 3.

Lastly, in Chapter 4, the subject of portfolio approaches will be discussed, while giving a few necessary notions of classification.

Chapter 1

Graphs

Contents

Graphs are extensively used in both mathematics and computer sciences. They can be used to represent various structured objects, or networks of entities. Moreover, many properties hold on such objects and enable useful computations.

This chapter presents an array of definitions related with graph theory in order to make it easier for the reader to grasp the principles behind the two mains problems addressed in this thesis.

We start by giving a few basic definitions in Section 1.1. We then define hypergraphs in Section 1.2, triangulated graphs in Section 1.3, and finally tree decompositions in Section 1.4.

1.1 Basic definitions

Graphs are widely used mathematical objects with many applications. They can be employed to model structured entities such as, for example, molecules, documents, road networks... They basically correspond to binary relations (called *edges*) defined over a given set of objects (the *vertices*, or *nodes*).

Definition 1.1. An *undirected graph* $G = (V, E)$ is defined by a finite set of *vertices* (or *nodes*) V and a set of edges E. Each edge $\{x, y\} \in E$ is a set of two vertices of V, which defines an undirected couple of nodes.
In a *directed graph*, $E \subseteq V \times V$, and each edge $(x, y) \in E$ is a directed couple of nodes.

Figure 1.1 shows examples of graphs.

In the following definitions, we will only be considering undirected graphs. However, most of these notions may be applied to directed graphs as well.

An edge can theoretically use the same vertex as both of its extremities; it is then called a *loop*. However, in this thesis, the graphs we consider do not have loops. Moreover, we will only use *simple* graphs, meaning that there cannot be more than a single edge between two given vertices. Many of the presented notions, however, can rather easily be adapted to such cases.

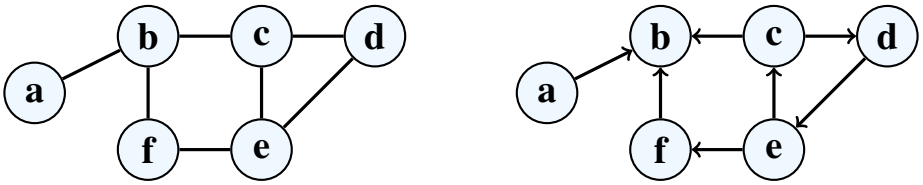


Figure 1.1 – On the left, an undirected graph $G = (V,E)$, with $V = \{a, b, c, d, e, f\}$ and $E = \big\{\{a,b\},\{b,c\},\{c,d\},\{d,e\},\{c,e\},\{e,f\},\{f,b\}\big\}$. On the right, a directed graph $G' = (V',E')$, with $V' = \{a,b,c,d,e,f\}$ and $E' = \big\{(a,b),(c,b),(c,d),(d,e),(e,c),(e,f),(f,b)\big\}$.

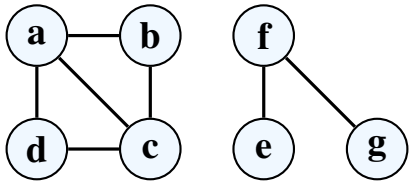


Figure 1.2 – A disconnected, simple, undirected graph.

Definition 1.2. The *density* of a graph is the ratio of the number of edges it actually contains and the maximal number of edges it could contain according to its number of vertices. In a simple undirected graph, it is given by the following formula:

$$\frac{2 \cdot |E|}{|V| \cdot (|V| - 1)}$$

Definition 1.3. A *path* is a sequence of edges connecting a sequence of vertices. The *length* of a path is the number of edges it contains.

In the left graph from Figure 1.1, (a,b,c,e) is a path. Its length amounts to 3. In the directed graph on the right of this same figure, (d,e,f,b) is a directed path.

Definition 1.4. A graph $G = (V,E)$ is *connected* if and only if for every pair $\{x,y\}$ of vertices of V, there is a path in G linking x to y.

For example, the graph on the left of Figure 1.1 is connected while the one in Figure 1.2 is not.

Definition 1.5. $G' = (V',E')$ is an *induced subgraph* of $G = (V,E)$ if $V' \subseteq V$ and $E' = E \cap \big\{\{x,y\} \;\big|\; x,y \in V'\big\}$. G' is called *the subgraph of G induced by* V', and will be denoted by $G_{\downarrow V'}$.

In other words, the induced subgraph $G' = G_{\downarrow V'}$ is obtained from G by removing all nodes of G which are not in V' and keeping only edges whose extremities are both in V'.

On the other hand, a *partial subgraph* is obtained by considering a subset V' of V and a subset of edges in G whose extremities are both in V'.

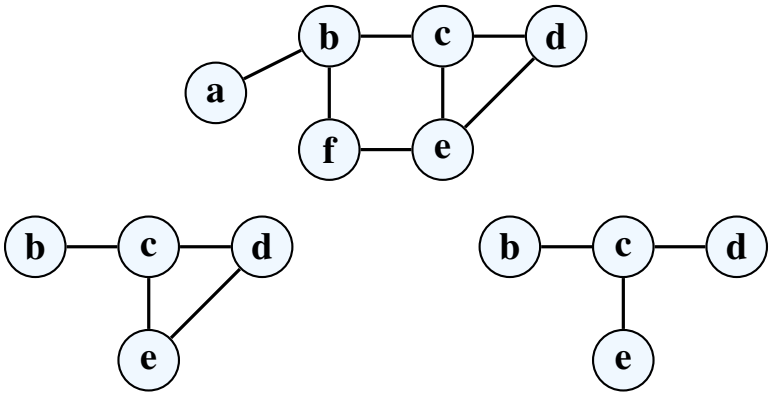


Figure 1.3 – A graph (on top), the subgraph induced by the set of vertices $\{b,c,d,e\}$ (bottom left graph), and a partial subgraph containing edges $\big\{\{b,c\},\{c,d\},\{c,e\}\big\}$ (bottom right graph).

Definition 1.6. $G' = (V',E')$ is a *partial subgraph* of $G = (V,E)$ if $V' \subseteq V$ and $E' \subseteq E \cap (V' \times V')$. A partial subgraph G' of G can be derived from any subset E' of E by defining G' as (V',E'), where $V' = \bigcup_{\{x,y\} \in E'} \{x,y\}$.

Figure 1.3 shows examples of induced and partial subgraphs.

Definition 1.7. A *connected component* G' of G is an induced subgraph of G that is connected and maximal (G' is a subgraph of no other connected induced subgraph of G).

The example graph from Figure 1.2 is composed of two connected components: one containing a, b, c and d, the other containing e, f and g.

Definition 1.8. A *cycle* is a path in which the first and last vertices are the same.

In the graph seen in Figure 1.2, (a,b,c,d,a) is a cycle.

Definition 1.9. A *chord* of a cycle C is an edge linking two non-consecutive vertices of C.

For example, in the graph from Figure 1.2, $\{a,c\}$ is a chord of the cycle (a,b,c,d,a).

Definition 1.10. A *tree* is a connected graph in which there is no cycle. Any vertex with a degree of 1 in a tree is called a *leaf*.

Alternative definitions exist. Saying that a graph $G = (V,E)$ is a tree is equivalent to stating that:

- it has no cycle and has exactly $|V| - 1$ edges;
- it is connected and has exactly $|V| - 1$ edges;
- it has no cycle and adding one edge would create exactly one;

- it is connected but would get disconnected if any edge were to be removed;
- for every pair $\{x,y\}$ of vertices of G, there is only one path linking x to y.

Also, note that a *forest* is a graph in which every connected component is a tree.

Definition 1.11. The *neighbourhood* $\mathrm{N}(v)$ of a vertex v is the set of vertices that are linked to v by edges:

$$\mathrm{N}(v) = \{x \in V \mid \{x,v\} \in E\}$$

The *degree* of a vertex v is its number of neighbours, $|\mathrm{N}(v)|$.

For example, in Figure 1.2, a has a degree of 3, while e only has a degree of 1.

Definition 1.12. A *clique* in a graph $G = (V,E)$ is a subset of V in which nodes are all linked pairwise (*i.e.*, it induces a complete subgraph).
A clique is *maximal* if it is not strictly included in any other clique, and it is *maximum* if it is the biggest clique of a given graph, with respect to the number of vertices.

For example, in the undirected graph on the left of Figure 1.1, the vertices $\{c,d,e\}$ induce a maximal and maximum clique, while $\{b,f\}$ induces a maximal clique that is not maximum due to its smaller number of vertices.

Computing a maximal clique is an easy problem that can be solved in polynomial time by a simple greedy algorithm. Computing a maximum clique, however, is $\mathcal{NP}$-hard [GJ02].

Definition 1.13. A *stable set* of vertices S within a graph (V,E) is a subset of V such that there does not exist an edge $\{x,y\}$ in E such that x and y are both in V.

In other words, it is a set of vertices within which no pair is linked by an edge.

The concept of *stable sets* is closely related to cliques, since a clique in a graph G corresponds to a stable set in the corresponding complementary graph, where two vertices are connected by an edge if and only if it is not the case in G.

1.2 Hypergraphs

Graphs have been generalized into the concept of *hypergraphs*, which are used to model relations of arbitrary arities.

Definition 1.14. A *hypergraph* $H = (V,C)$ is a couple holding a set V of vertices and a set C of *hyperedges*. Each hyperedge is a subset of V.

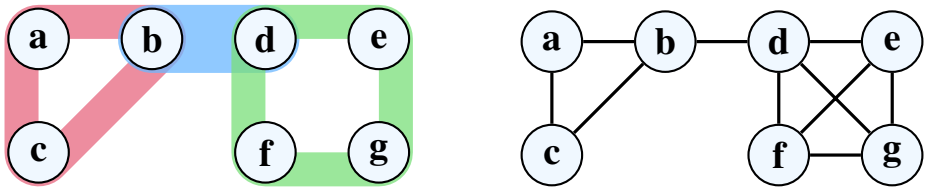


Figure 1.4 – A hypergraph and its 2-section. This hypergraph is defined by $H = \big(\{a,b,c,d,e,f,g\},\ \big\{\{a,b,c\},\{b,d\},\{d,e,f,g\}\big\}\big)$.

In other words, what makes a hyperedge different from an edge is its arity: it can link more than two vertices. Edges can be seen as a particular case of hyperedges.

In some cases, it is more convenient to represent a hypergraph as a graph, by replacing each hyperedge by a clique made of binary edges. This is called the 2-*section* of the hypergraph.

Definition 1.15. [BER73] The 2-*section* of a hypergraph $H = (V,C)$ is a graph $G = (V,E)$ such that E is the set of all edges $\{x,y\}$ such that at least one hyperedge in C contains both x and y:

$$E = \big\{\{x,y\} \,\big|\, \exists c \in C \,\big|\, \{x,y\} \subseteq c\big\}$$

Figure 1.4 shows a hypergraph example, along with its 2-section.

1.3 Triangulated graphs

Definition 1.16. A graph is *triangulated* (or *chordal*) if and only if every cycle of length 4 or more that can be found in it has a chord.

Triangulated graphs have several remarkable properties (see [GOL80] for a more complete overview) and were extensively studied. Several $\mathcal{NP}$-complete and $\mathcal{NP}$-hard problems can be solved with polynomial algorithms for triangulated graphs.

Definition 1.17. A vertex v of G is *simplicial* if the subgraph induced by its neighbourhood is complete, *i.e.*, if $\mathrm{N}(v)$ is a clique of G.

Theorem 1.1. [DIR61; LB62] *A triangulated graph always has at least one simplicial vertex.*

Theorem 1.2. *Removing a simplicial vertex from a triangulated graph yields another triangulated graph.* [GOL80]

FULKERSON and GROSS used Theorems 1.1 and 1.2 to design an efficient recognition method for triangulated graphs [F+65]. It simply consists in recursively looking for a simplicial vertex in the considered graph and to remove it. If

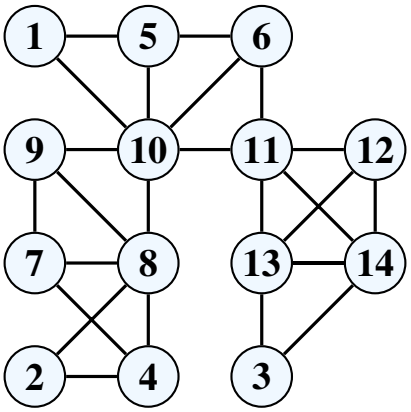


Figure 1.5 – A simplicial order (corresponding to the numeric order among the integer values associated with vertices) for the triangulated graph from Figure 1.6. The 8^{th} vertex, for example, has vertices $\{2,4,7,9,10\}$ as its neighbourhood. This set does not induce a clique. But the *ulterior* neighbourhood provided by the chosen order to this vertex is only comprised of the 9^{th} and 10^{th} vertices. These two vertices are linked by an edge and therefore constitute a clique.

the graph can be reduced to zero vertices by this procedure, it was triangulated; if at some point no simplicial vertex can be found, it was not. The order used to eliminate every vertex in such a way was named "perfect elimination scheme".

Definition 1.18. Let $<$ be a total order defined over the set of vertices of a graph. The *ulterior neighbourhood* $\mathrm{N}^+_<(v_i)$ of a vertex v_i according to $<$ is defined as follows:

$$\mathrm{N}^+_<(v_i) = \{v_j \in \mathrm{N}(v_i) \mid v_i < v_j\}$$

Definition 1.19. An order $<$ on the vertices of G is a *simplicial order* (or *perfect elimination scheme*) if, for each vertex v of G, the graph induced by $\mathrm{N}^+_<(v)$ is complete, *i.e.*, if $\mathrm{N}^+_<(v)$ is a clique of G. It is also equivalent to stating that v is simplicial in the graph induced by $\{v\} \cup \mathrm{N}^+_<(v)$.

Figure 1.5 shows an example of a simplicial order.

Simplicial orders were used to give an alternate definition for triangulated graphs.

Theorem 1.3. [F+65] *A graph is triangulated if and only if a simplicial order can be found on its vertices.*

Using these properties, along with others not mentioned here, triangulated graphs can be recognized in linear time [LUE74; RTL76], using for example the *maximum cardinality search* (MCS) algorithm.

In most domains, for a given graph, the probability that it is naturally triangulated is fairly low, since the conditions to belong to this class are strong. Most of the time, when an algorithm – no matter its goal – makes use of

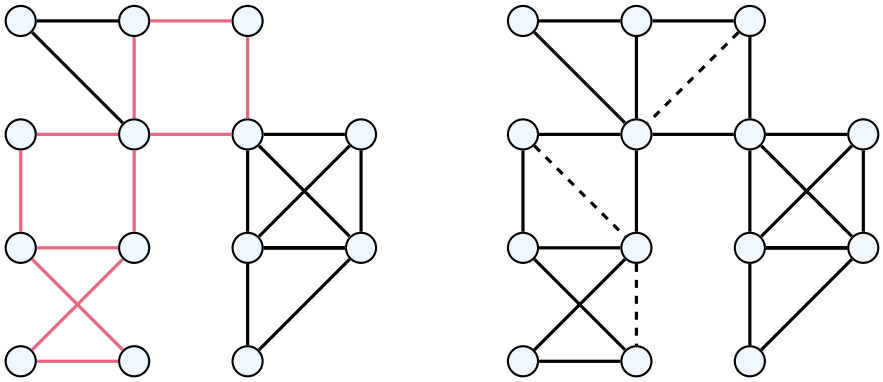


Figure 1.6 – A graph on the left, and a way to make it triangulated by adding three edges. The cycles of length 4 or more that prevent the original graph from being triangulated are highlighted in **red**. The suggestions of edge additions are shown in dashed lines.

triangulated graph properties, a given graph is actually forcefully turned into a triangulated graph in order to be able to derive profit from said properties.

The operation consisting in adding edges to a non-triangulated graph in order to turn it into a triangulated one will be referred to as *triangulation* from here onwards.

Definition 1.20. The term *triangulation* is used to refer to the process of adding a set of edges to a given graph in order to turn it into a triangulated graph. It also sometimes describe in the literature the set of these additional edges.
The additional edges themselves are called *fill edges*.

Figure 1.6 shows a graph along with a way to triangulate it.

Many approaches can be adopted to find an appropriate set of fill edges to perform a triangulation. In most cases, it is preferable to have as few fill edges as possible. However, finding the minimal set is $\mathcal{NP}$-hard. A good approximation can generally be found with the simple – and yet efficient and widely used – *MinFill* algorithm [KJA90].

MinFill is a greedy algorithm that consists in building a simplicial order $<$ by adding fill edges. This order is built incrementally from its first vertex to the last, adding one vertex at a time.

Each time a vertex v is selected to be the next one in the order $<$, its ulterior neighbourhood $\mathrm{N}^+_<(v)$ is turned into a clique by adding all the necessary edges. Note that $\mathrm{N}^+_<(v)$ is comprised of the neighbours of v that have yet to be placed in the order, since they will necessarily be placed *after* v (*i.e.*, unordered vertices are seen as ulterior to ordered ones). The vertices are selected in a greedy way, according to the number of fill edge additions that their selection would cause: priority is given to vertices that need the fewest fill edges to become simplicial. Once every vertex has been placed in the order $<$, the graph has become

triangulated, according to Theorem 1.3, and $<$ is ensured to be a simplicial order in the new graph.

1.4 Tree decompositions

1.4.1 Definitions

A tree decomposition is a hierarchical set of *clusters* of vertices. It has many uses, and can make the inner structure of a graph easier to grasp and to exploit.

Definition 1.21. [RS86] For a graph $G = (V, E)$, a *tree decomposition* of G is a couple $(\mathcal{C}, \mathcal{T})$, where:

- $\mathcal{T} = (V_\mathcal{T}, E_\mathcal{T})$ is a tree;
- $\mathcal{C}$ is a set holding one element $\mathcal{C}_v$ for each node v of $V_\mathcal{T}$;
- Each element of $\mathcal{C}$ is a subset of V; these subsets are called *clusters*;
- The union of the clusters of $\mathcal{C}$ equals V;
- For each edge $\{x, y\}$ of G, there exists a cluster $\mathcal{C}_v$ of $\mathcal{C}$ such that both x and y are in $\mathcal{C}_v$;
- For each $x, y, z \in V_\mathcal{T}$, if y is on a path linking x to z in $\mathcal{T}$, then $\mathcal{C}_x \cap \mathcal{C}_z \subseteq \mathcal{C}_y$.

Definition 1.22. The *width* of a tree decomposition $(\mathcal{C}, \mathcal{T})$ of a graph G is the value given by $\max_{\mathcal{C}_v \in \mathcal{C}} \{|\mathcal{C}_v|\} - 1$.

Intuitively, it corresponds to a length that no path would be able to exceed in G by remaining in a single cluster of the decomposition and using vertices only once, even if the clusters were cliques.

Definition 1.23. The *treewidth* w of a graph is the minimal width among its possible tree decompositions.

Definition 1.24. A *separator* between two adjacent clusters of a tree decomposition is the intersection of these clusters.

Figure 1.7 shows an example of tree decomposition.

1.4.2 Construction

For a given graph, many different tree decompositions may be computed, and some of them might be more helpful than others. The main goal is generally to allow a faster resolution by providing high quality decompositions to solvers that are able to use such decompositions. This quality is generally estimated

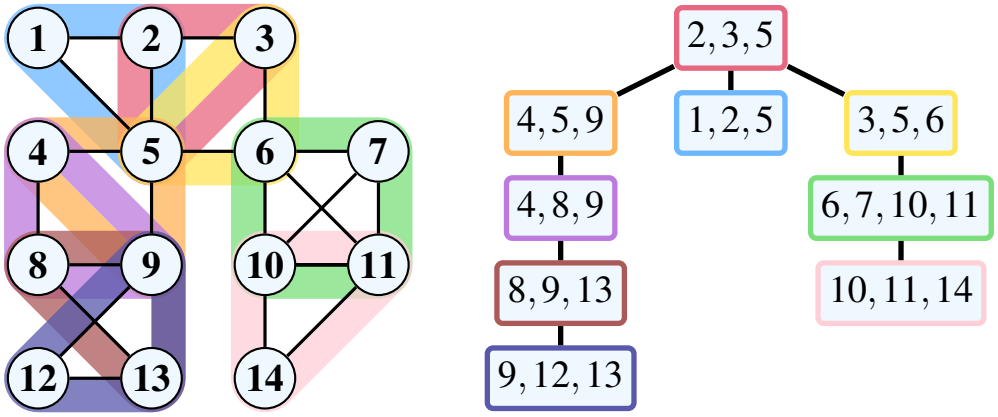


Figure 1.7 – A tree decomposition of width 3 for the initial graph from Figure 1.6. The separator between $\{2,3,5\}$ and $\{3,5,6\}$ is $\{3,5\}$.

beforehand using the width w of the decomposition, which has a significant impact on the theoretical complexity of numerous problems.

Computing a tree decomposition of minimal width, however, is an $\mathcal{NP}$-hard problem in itself. For this particular reason, the optimal width is very seldom sought in practice, and approximate solutions are heuristically computed instead [Bod96; HHR03].

Computing a tree decomposition commonly involves a triangulation step. Indeed, every triangulated graph can be used to obtain a tree decomposition by using the set of its maximal cliques as clusters. More precisely, one can follow a simplicial order $<$ of the triangulated graph and, for each vertex v in this order, use $\mathrm{N}^+_<(v) \cup \{v\}$ – which is a clique by definition – as one of the clusters of the tree decomposition.

In the context of tree decompositions, the main quality criterion for the triangulation is generally not be the number of fill edges but rather the maximal size of the cliques that it yields, since it is directly linked to the size of the generated clusters and thus to the theoretical complexity bound. Still, *MinFill* is widely used even in this context, due to its good performances and relatively short execution time.

The next step consists in checking intersections to find out which clusters can be made adjacent in the future tree decomposition. This is done by building the *clique graph* of the triangulated graph. In the clique graph, each vertex corresponds to one of the computed cliques. These vertices are linked by an edge if and only if the corresponding cliques intersect (*i.e.*, if they have at least one vertex in common). Moreover, these edges are weighted according to the size of these intersections.

The shape of the tree decomposition is eventually determined by Prim's algorithm ([Pri57]; originally developed by Vojtěch Jarník in 1930). Using as its input an undirected graph with weights associated to its edges, this algorithm builds a tree spanning each of these vertices, with a maximal sum of edge weights. It starts with a tree comprising a single, arbitrarily chosen

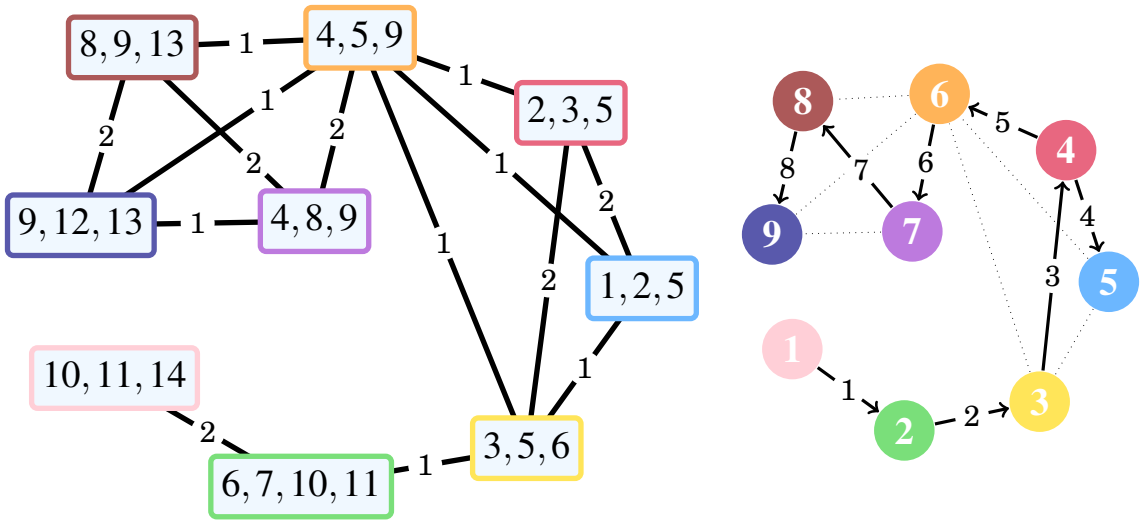


Figure 1.8 – On the left, the clique graph that can be computed from the triangulated graph of Figure 1.5. The numbers displayed on each edge correspond to the number of vertices comprised in each non-empty clique intersection. On the right, a tree spanning every vertex of this clique graph, built using Prim's algorithm by prioritizing edges with great weights. The numbers on vertices correspond to the order in which they were integrated to the tree; the same goes for the edges. Note that the arrows simply show which vertex was already in the tree and which was not when each edge was added to the current tree – the resulting tree is not a directed graph. Using this tree to build a tree decomposition would yield precisely the one shown in Figure 1.7.

vertex, and greedily adds edges, one at a time, choosing at each step an edge having the highest possible weight among edges that connect the current tree to vertices that are not yet in the tree.

Note that Prim's algorithm is usually used to produce trees with low weights, but in this context, choosing great weights (*i.e.*, large intersections between cliques) ensures that a defining trait of tree decompositions is respected: the intersection between two clusters must be included in every path existing between these clusters (as stated in Definition 1.21).

Figure 1.8 shows how this algorithm would proceed when confronted with the clusters suggested on the left-hand side of Figure 1.7.

Additionally, a noteworthy property of triangulated graphs makes them particularly interesting for the construction of tree decompositions, as stated by the next theorem.

Theorem 1.4. *A triangulated graph* (V,E) *cannot contain more than* $|V|$ *maximal cliques.*

In a triangulated graph, the number of maximal cliques is thus linear, whereas it can be exponential in other graphs. This makes it possible to build

tree decompositions with a reasonable number of clusters, in addition to the aforementioned fact that these maximal cliques are easy to find (by following a simplicial order) once the graph is triangulated.

Triangulation is often the most costly step in a tree decomposition algorithm. *MinFill*'s time complexity bound is $\mathcal{O}\big(|V| \cdot (|V| + |E|)\big)$, while implementations of Prim's algorithm in $\mathcal{O}\big(|E| \cdot \log(|V|)\big)$ and $\mathcal{O}\big(|E| + |V| \cdot \log(|V|)\big)$ are possible.

Note that the fill edges added during the triangulation step are merely temporary tools to build the tree decomposition more easily. As such, they get removed once the decomposition phase is over. What matters is that the decomposition should reflect the strong connections between some sets of variables and the independence of some other sets, and fill edges generally do not add too much noise in these aspects.

Chapter 2

Constraint programming

Contents

Creating an ad hoc solver for each new problem that surfaces is not always a viable solution, especially when time is of the essence or when numerous different experiments must be carried out for the purpose of testing. Therefore, various frameworks and formalisms have been proposed in the literature in order to easily define and solve instances of many problems.

This chapter begins by presenting the constraint programming framework in Section 2.1. Both satisfaction and optimisation problems will be addressed in this context. Tools and notions necessary to solve such problems will be listed through Section 2.2. The important topic of decomposition applied to such problems will then be discussed in Section 2.3. Finally, parallelisation methods will be described in Section 2.4.

2.1 Constraint satisfaction problems

2.1.1 Formalism

Constraint programming (CP) is a generic framework used to model and solve constraint satisfaction problems (CSPs). CSPs allow to easily express numerous problems, simply by stating variables (the unknowns of the problem) and constraints that link some of those variables together or prevent them from taking certain values.

Unknowns, which are concerned with choices that must be made to solve the problem, are represented by variables. A constraint is a relation defined over one or several variables that forbid the conjoint use of certain values for these variables.

Several CP modelling languages were developed during the past decades, and large numbers of constraints exist. When using CP, the user can focus on modelling the problem, leaving the solution process entirely to the solver, which remains amply generic. There is no necessity to tell the solver *how* to find solutions to the problem; one must simply *describe* the problem by stating as explicitly as possible what the variables and constraints are.

These fairly simple concepts appear to be sufficient to model a large array of real-life and academic problems, either for optimization or to determine the existence of a solution.

CSPs may be discrete or continuous, depending on whether variables may draw their values from discrete or continuous sets of possibilities. This thesis only addresses the case of discrete CSPs.

Definition 2.1. [MON74] A CSP $\mathcal{P} = (X, D, C)$ is defined by:

- A set of variables $X = \{x_1, \ldots, x_n\}$;
- A set of finite, discrete domains $D = \{D_{x_1}, \ldots, D_{x_n}\}$;
- A set of constraints $C = \{C_1, \ldots, C_m\}$, each constraint C_i being defined over a subset of X (the *scope* of the constraint); the size of the scope is the *arity* of the constraint.

A CSP is *binary* if none of its constraints have an arity greater than 2. It is *n-ary* otherwise.

Each constraint defines restrictions on the combinations of values for the variables of its scope. These restrictions can be expressed by a set of *relations* $R = \{R_1, \ldots, R_m\}$, where R_i is the set of combinations of values satisfying the constraint C_i. Thus, if C_i is defined over variables $\{y_1, \ldots, y_k\}$, R_i will be a subset of $D_{y_1} \times \ldots \times D_{y_k}$.

Constraints can be stated either in *intention* (by expressing the constraint with a formula) or in *extension* (by explicitly enumerating every allowed combination of values). Stating relations in extension usually uses up a consequent amount of memory, depending on the number of necessary tuples. Furthermore, checking whether an assignment is consistent with the constraints of

Table 2.1 – An example of a constraint given in extension. The considered constraint is $x_1 \neq x_2$. This relation states every acceptable combination of values for the variables in the constraint's scope.

x_1	1	1	1	2	2	2	3	3	3	4	4	4
x_2	2	3	4	1	3	4	1	2	4	1	2	3

the problem is often significantly faster when using formulas rather than lists of tuples.

A commonly given example of CSP is the n-queens problem, which consists in placing chess queens on a chess board of a given size (generally $n \times n$) in such a way that none threatens another. More formally, there should not be any pair of queens a and b such that a and b share a same diagonal, line or row.

The following example shows how to model the 4-queens problem on a board of size 4×4 as a simple CSP.

Example 2.1. Basic CSP for the 4-queens problem:

$$
\begin{aligned}
X &= \{x_1, x_2, x_3, x_4\} \\
&\quad \forall x \in X,\ D_x = \{1,2,3,4\} \\
C &= \left\{ \text{“}x_i \neq x_j\text{”} \;\middle|\; i,j \in \{1,2,3,4\} \wedge i < j \right\} && \text{(Columns)} \\
&\quad \cup \left\{ \text{“}|i-j| \neq |x_i - x_j|\text{”} \;\middle|\; i,j \in \{1,2,3,4\} \wedge i < j \right\} && \text{(Diagonals)}
\end{aligned}
$$

Giving the “$x_1 \neq x_2$” constraint in extension would mean building the relation given in Table 2.1.

Definition 2.2. Given a CSP (X, D, C), an *assignment* A of a set $X' \subseteq X$ of variables is a function defined on X' such that, for each $x \in X'$, $A(x) \in D_x$. It is *complete* for a given CSP if $X' = X$ (X' is then the full set of the variables of this CSP); otherwise, it is *partial*.

For convenience, we also use the notation $x \leftarrow v$ to state that x is assigned the value v.

Less formally, building an assignment consists in associating to variables values from their respective domains. In our 4-queens example, $\{x_1 \leftarrow 3,\ x_3 \leftarrow 2\}$ would be a partial assignment, while $\{x_1 \leftarrow 3,\ x_2 \leftarrow 1,\ x_3 \leftarrow 2,\ x_4 \leftarrow 1\}$ would be a complete one.

Definition 2.3. The *projection* on X' of an assignment of a set $X \supset X'$ of variables is a smaller assignment defined only on X' and using the same values.

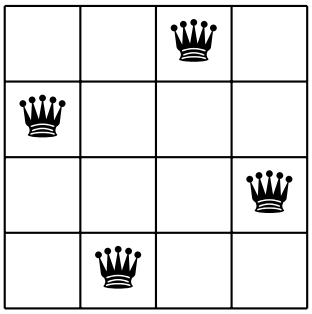


Figure 2.1 – A solution to the 4-queens problem.

Definition 2.4. Given a constraint c and an assignment A defined at least on all the variables of c's scope, c is:

- *satisfied* by A if the projection of A on c's scope belongs to the relation corresponding to c (*i.e.*, the combination of values imposed by A is allowed by c);
- *violated* otherwise.

An assignment that does not violate any constraint is said to be *consistent*. A *solution* to a CSP is a consistent complete assignment of its variables.

Depending on the context, the CSP may be solved in different ways. Among the most common tasks, we find:

- Checking for the **existence** of at least one solution;
- **Counting** existing solutions;
- Finding **every solution**;
- Finding the **best solution** with regards to a particular *objective function* defined over the variables.

These tasks are generally $\mathcal{NP}$-complete or $\mathcal{NP}$-hard [GJ02].

A solution to the 4-queens example could be $x_1 = 3$, $x_2 = 1$, $x_3 = 4$ and $x_4 = 2$. It corresponds to the layout described in Figure 2.1.

The 4-queens problem model proposed earlier uses the fact that there will necessarily be exactly one queen on each row to reduce the complexity of the model. Indeed, it makes the problem simpler to solve: only the columns on which queens are placed must be chosen, instead of two coordinates for each of them. As a matter of fact, most problems can be modelled with many different CSPs, leading to varying resolution speeds. Regarding the n-queens problem, one could even pre-compute permutations of $\{1,\ldots,n\}$ to make sure the queens do not share a column, and then use explicit constraints only for the diagonals. However, such an approach would not be scalable, since the number of permutations grows exponentially with the dimensions of the board.

Yet another way to model this problem is to rely on a set of binary variables associated with each square of the board. These variables are assigned a value of 1 if any queen sits on the corresponding square, and 0 if the square is vacant.

Constraints based on sums then allow to easily check whether multiple queens lie in a same row, column or diagonal.

Basic structure

The structure of a CSP can be roughly represented by using a *constraint hypergraph*.

> **Definition 2.5.** The *constraint hypergraph* of a CSP $\mathcal{P}$ is a hypergraph $G = (V, C)$ where V is comprised of one vertex v_{x_i} per variable x_i of $\mathcal{P}$, and C is a set of hyperedges holding one hyperedge for each constraint of the CSP. Each hyperedge links the variables that are in the scope of the corresponding constraint.

In a binary CSP, this constraint network can be represented by simply using a graph, each edge corresponding to a constraint.

This definition of constraint graphs was primarily designed for binary CSPs. When the CSP contains constraints that have an arity greater than 2, the 2-section of the constraint hypergraph might be useful. This 2-section is by definition a normal graph, in which there is an edge linking v_{x_a} and v_{x_b} if and only if there is a constraint in the CSP that is defined over both x_a and x_b.

2.1.2 Global constraints

Some possible alternative 4-queens problem models involve *global constraints*. Global constraints are an extension to constraint programming, consisting of new primitive constraints with heavier semantics and sometimes specific consistency algorithms which help speed up the solution process.

A common example of global constraint is the *allDifferent* constraint [RÉG94]. It arises naturally in many academic and real life problems, and will be extensively used in this thesis. It states that each variable in its scope must use a distinct value. Enforcing the constraint $allDifferent(a, b, c)$ is thus equivalent to the conjunction of the three constraints $a \neq b$, $b \neq c$ and $a \neq c$.

Several consistency algorithms were designed for this constraint, each with its own level of consistency and complexity [RÉG94]. These algorithms commonly use concepts such as *Hall intervals* and establish a connection with the maximum matching problem, using bipartite graphs in which variables must be associated with values [COS94].

Using the *allDifferent* constraint to model the 4-queens problem, one could replace the constraints pertaining to the columns in the previously described model with the constraint $allDifferent(x_1, x_2, x_3, x_4)$.

2.1.3 Optimisation problems

Constraint satisfaction problems, as defined previously, basically involve finding a solution that satisfies, by definition, all the constraints, or proving that none exists. However, in some contexts, given parameters have to be optimized, thus inducing preferences between existing solutions. This can be done for example by introducing an *objective function* into the model.

> **Definition 2.6.** A *constrained optimization problem* (COP) is a CSP that includes, as an additional element, an *objective function*, which is defined over some of the variables of the problem. The goal of a COP might be either to find the solution that maximizes the objective function or the one that minimizes it.

On the other hand, some problems do not have any solution that can satisfy every single constraint. Such problems can be turned into maximization problems, where the goal is to maximize constraint satisfaction. These techniques led to weighted CSPs.

A *weighted constraint satisfaction problem* (WCSP), also referred to as *cost function network*, is a generalisation of CSPs in which some constraints may be violated, given a certain cost. This allows to express preferences among solutions.

> **Definition 2.7.** A WCSP is defined like a COP in every respect, except for two aspects:
>
> - Each constraint is defined as a *cost function* associating a numeric cost with each possible assignment of the variables in the constraint's scope. Costs can be infinite.
> - The objective function is implicit and corresponds to the sum of the effective costs of every constraint. It must always be minimized.

> **Definition 2.8.** Constraints associating only infinite or null costs to assignments are described as *hard*, as opposed to *soft* constraints.

An assignment with an infinite cost cannot be part of a valid solution. Also, note that a CSP can be modelled using the WCSP formalism, simply by using only hard constraints.

2.2 Generic algorithms

Solving a CSP involves finding a complete consistent assignment. Since this problem is $\mathcal{NP}$-complete, a multitude of techniques have been developed over the years to widen the scope of instances that can be solved with reasonable material and temporal resources.

2.2.1 Chronological backtracking

A trivial way to solve a CSP is to enumerate every assignment. This is called the *chronological backtracking* method, since it goes back on its choices whenever it runs out of possibilities. This enumeration is achieved by exploring the search space in a *depth-first search* (DFS) fashion: starting from an empty assignment, we recursively extend it by choosing a non-assigned variable x and a value $v \in D_x$ and adding $x \leftarrow v$ to the current assignment. Whenever the resulting assignment becomes inconsistent, a *backtrack* is performed: the search goes back to the last choice point and an alternative value is used to extend the assignment. If every value for this choice point has already been tested, we backtrack a step further. On the other hand, if the current assignment happens to be a solution, the search may stop.

The worst case complexity of the backtracking algorithm is exponential with respect to the number of variables. A major drawback of this method is the omnipresence of redundancies encountered during the search. It can be improved in several ways, for example by using variable- and value-ordering heuristics [PUR83; DM94; MOS+01; BOU+04], or by backtracking in a non-chronological way [SS77; GAS79]. Such backtracking techniques can rely on a variety of things, such as conflicts [PRO93; BAK94] or the constraint graph [DEC90].

The enumeration process implied by a backtracking approach can be visualized as a tree, commonly referred to as the *search tree*, in which:

- The root node corresponds to an empty assignment, where no variable has a value yet. This is generally where the search starts from;
- Leaves are either consistent complete assignments (solutions) or inconsistent assignments that are not necessarily complete (failures);
- Internal nodes are consistent partial assignments.

Thus, the edges that link these nodes corresponds to the assignment of a value to a variable. A variant involves edges which semantically mean either $x = value$ or $x \neq value$, each non-leaf node thus having two children.

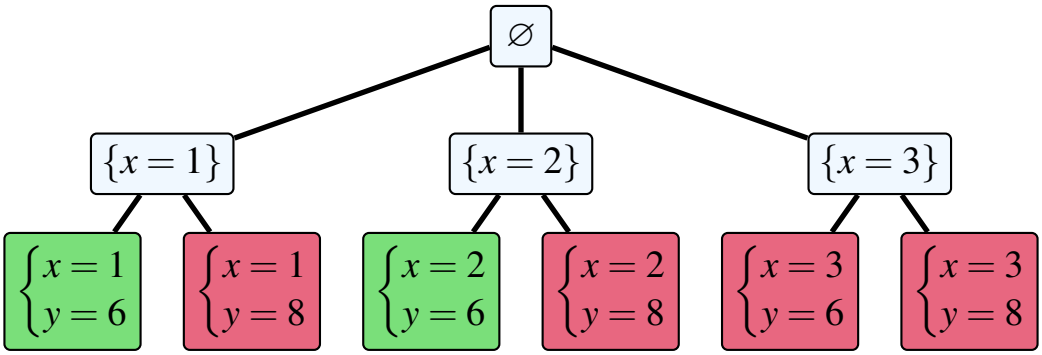
Figure 2.2 shows a simple example of a search tree.

Several different search trees can be produced for a same CSP model. The shape of the tree depends for example on the heuristics used to choose the next variable that will be assigned, and on the use of many other strategies, some of which will be detailed further in this section.

2.2.2 Local consistency

Constraint propagation

Local consistencies are properties that can be applied to a CSP or to some of its constraints in order to remove inconsistent values. This process, which can be seen as a kind of *filtering* method, reduces the size of the search tree,



$$D_x = \{1,\ 2,\ 3\}$$
$$D_y = \{6,\ 8\}$$
$$C = \{\text{“}x + y \leq 8\text{”}\}$$

Figure 2.2 – The concept of a search tree, graphically explained for a simple problem with two variables and one constraint. Leaves are marked either as **solutions** or **failures**.

thus making the problem easier to solve. Removing a value from a domain generally involves proving that the current assignment cannot be extended to a solution when this value is used. This ultimately eliminates branches that would otherwise have been explored.

Different degrees of filtering have been proposed, each with its own cost and aggression level with regards to the number of values it is likely to be able to remove. When the filtering is only made with regards to a subset of the variables of the problem, it is referred to as a *local consistency* method.

Constraint propagation consists in repeatedly removing every value that are proved to violate constraints. Since a removal can sometimes allow for further deductions, the process starts again with each value deletion, until a fixed point is reached (*i.e.*, until no more suppressions can be performed).

Most global constraints have their own consistency algorithms. This is one of the major appeals of these constraints, as these ad-hoc algorithms are generally more efficient than the general ones.

The act of exploring the search space by repeatedly assigning values to variables and propagating constraints by means of filtering is a method called *branch and propagate*.

Forward checking

The first and most straightforward filtering algorithm was *forward checking* (FC), which was defined for binary CSPs. It has a low computational cost due to the fact that it avoids performing several passes over the variables.

When a variable gets assigned, FC checks whether variables linked to it by constraints still have compatible values in their domains. In the meantime, it removes incompatible values from those domains.

FC can be used as a *look ahead technique*: when trying to evaluate an assignment $A : x \leftarrow v$, it starts by extending the current partial assignment with A; it then considers every variable x' such that x' is still unassigned and is a neighbour of x in the constraint graph, and checks whether its domain contains at least one value that is consistent with the extended partial assignment.

Though FC was primarily designed for binary CSPs, n-ary variants (nFC1, nFC2...) were eventually defined [LM98; BES+99].

Arc consistency

Arc consistency is the most commonly used type of local consistency, due to a good balance between its computational cost and filtering abilities.

Definition 2.9. Given a variable x, a value $v \in D_x$ and a constraint c involving x, an assignment A of the variables in c's scope is a *support* of v with regards to c if A does not violate c and if A assigns v to x.

Definition 2.10. A constraint c is *arc consistent* if and only if, for every variable x in its scope, for every value $v \in D_x$, v has a support with regards to c. [WAL75]

Definition 2.11. A CSP is said to be arc consistent when all its constraints are arc consistent. [WAL75]

Figure 2.3 shows the successive steps of an arc consistency enforcement.

Arc consistency was designed for binary constraints but can be generalized to n-ary cases. It is then called *generalized arc consistency* (GAC), or *hyper-arc consistency*.

Arc consistency can be used both as a preprocessing technique and during the search. Numerous arc consistency algorithms were proposed [MAC77; MH86; BES94; BR01; ZY01]. One of the most used ones, AC-3 [MAC77], has a worst-case time complexity of $\mathcal{O}(m \cdot d^3)$ and a space complexity of $\mathcal{O}(m)$, m being the number of edges in the constraint graph and d the maximal size among domains. It maintains a list of constraints that need to be checked again for consistency, thus reducing the computational cost implied by the repeated executions of the consistency procedure. Its main appeal, though, is its simplicity. Slightly more elaborate algorithms, such as AC-3.1 (also known as AC-2001) are even more widely used [BES+05].

To make the filtering step shorter, one can use *bound consistency* instead of the default *domain consistency*. Bound consistency consists in only scanning the bounds of domains (*i.e.*, the minimal and maximal values currently available) when checking whether every value has a support [PUG98].

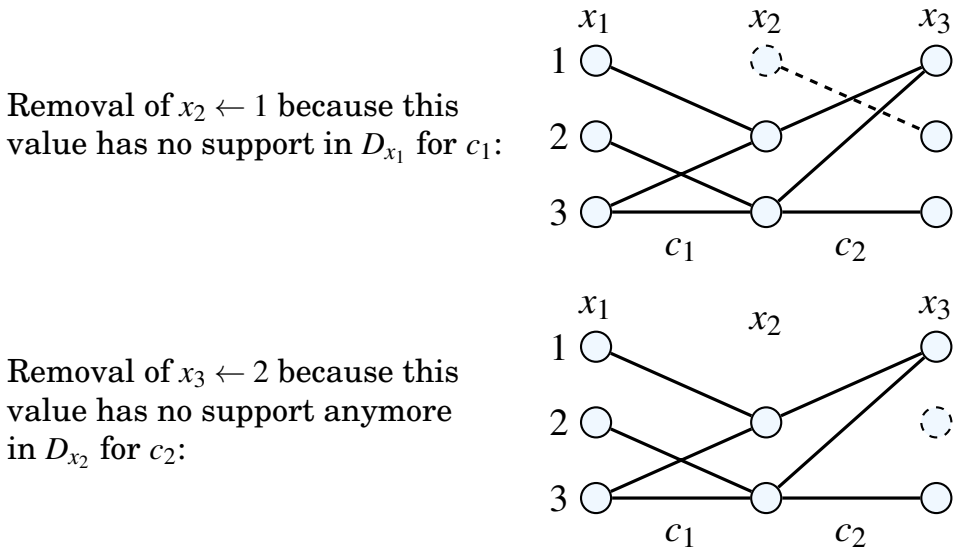


Figure 2.3 – The enforcement of arc consistency on a problem with three variables and two constraints. Edges represent supports (compatible values with regards to a constraint). The first removal makes c_2 non-arc consistent and occasions another removal.

The strategy consisting in making sure a CSP is arc consistent at all times is called *maintaining arc consistency* (MAC).

All these algorithms are widely used in practice [SF94; FRE95], but do not completely suppress the problems posed by the theoretically prohibitive complexity of backtracking algorithms.

2.2.3 Variable-ordering heuristics

Numerous heuristics can be employed in order to decide which variable should be assigned next at each step of the search. Here is a short presentation of some heuristics that had a significant impact on research.

Deg and Ddeg The *Deg* heuristic selects the variable that has the highest degree in the constraint graph [DM94]. Its dynamic version, *Ddeg*, only considers constraints that have unassigned variables in their scopes.

Dom The *Dom* heuristic selects the variable that has the fewest values in its current domain, thus creating less branching perspectives [HE80]. This heuristic has been extensively used since it was proposed in 1980.

Dom/Ddeg *Dom* and *Ddeg* can be combined by computing the ratio between the size of the current domain and the dynamic degree of a variable. Lowest

ratios are generally preferred over large ones. This heuristic is simply called *Dom / Ddeg* [BR96; SG97].

Wdeg Some heuristics are based on *conflicts*: each constraint is attributed a weight which is increased every time this constraint causes a domain to be completely emptied during propagation. Each variable x also has weights, called *weighted degrees*, that correspond to the sum of the weights of the constraints whose scopes contain x. Variables with highest weighted degrees are selected first by the *Wdeg* heuristic [BOU+04].

Dom / Wdeg The heuristic *Dom / Wdeg* is implemented in the same way as *Dom / Ddeg*: ratios are computed using both the *Dom* and *Wdeg* heuristics, and lowest ratios are favoured [BOU+04].

Activity Activity-based heuristics attach an *activity* value to each variable. This value is increased every time the variable is involved in a backtrack-inducing conflict. After such conflicts, the activity of every variable is *decayed* by means of a multiplication by a constant lower than 1. Generally, variables with a higher level of activity are favoured by the variable-ordering heuristics of this type. These approaches stem from SAT solvers development [MOS+01].

2.2.4 Nogoods

Definition 2.12. A *nogood* $\overline{g}$ is a partial assignment that cannot possibly be extended to obtain a solution: there exists no consistent complete assignment that includes all assignments from $\overline{g}$.

In the 4-queens problem shown in Figure 2.1, placing the first queen in the upper left corner of the board (*i.e.*, assigning 1 to x_1) is a nogood, since no solution can be found using this placement.

In a decision problem, when a branch of the search tree is found to contain no solution, it can be interesting to record the conflicting set of assignments that caused this absence of solutions [DEC90]. This set is by definition a nogood. A trivial one is made of the whole set of assignments that were performed up to the beginning of the fruitless branch. However, since the main goal of nogoods is to avoid exploring a part of the search space several times, it is generally better to compute a smaller set of assignments to obtain a more useful nogood, since it will then describe a situation that is more likely to occur again: it is more general and less restrictive.

For this reason, much effort can be directed at circumscribing the cause of failures. Each time such a failure occurs, constraints may be analysed in order to determine which of them is at the source of this failure, and variables may be removed from the nogood depending on their implication [SV94; PRO93; BAK94].

2.2.5 Solving constrained optimisation problems

Most of the aforementioned techniques can be extended or adapted to an optimisation context. A few remarks should be made on some techniques, however.

Branch and bound

When using a backtracking approach, each time a branch is considered for exploration, one can compute estimates (bounds) of the objective function's future values. If it can be proven that a branch does not contain any solution which is better than the best one found to this point, this branch can be pruned. This approach is called *branch and bound*.

Furthermore, lower bounds computed on subproblems can be used in order to know whether the current branch is worth exploring further. A branch and bound approach also generally involves adding a constraint to the problem whenever a new global solution is found. This constraint simply states that from this point on, only solutions that are of greater quality than the last one are to be sought.

Branch and bound is usually combined with constraint propagation.

Consistency for WCSPs

Consistency cannot always be enforced in a WCSP exactly as it is in a CSP. Indeed, in a WCSP, it is allowed to violate some constraints as long as the associated cost is acceptable, so removing every value that would induce a non-null cost is not an option. What must be checked instead is whether these values would prevent the search algorithm from finding a solution whose quality is higher than that of the current best solution. Therefore, consistency algorithms had to be adapted, and the concept of local consistency itself has been redefined [LAR02; LS03; DE +05].

2.2.6 Restarts

Motivation and principle

Researchers observed extremely high variability in the runtime needed to solve some problems [S+93; GW94]. This uncertainty can be exploited as a positive trait by starting the search anew after some time. More precisely, if the search does not end after a given number of backtracks (the *cutoff*), it is restarted with a new random seed, leading to different choices if the ordering heuristics in use comprise stochastic elements [G+98].

This technique bears the straightforward name of *restart*, and has been heavily studied. For this approach to be even more profitable, a few informations can be retained: the search generally does not restart exactly from scratch. These informations, such as upper and lower bounds, may contribute in pushing the search along more interesting branches than during previous tries. Nogoods can also be recorded to make the subsequent search sessions more fruitful [LEC+07A; LEC+07B].

Restart strategies were initially designed for decision problems and incomplete approaches, but has also been popular for a long time for complete approaches. One could intuitively think that restarting the search every now and then would not be a good idea when the search space must be checked in its entirety. However, experience showed that restarts are, in practice, very profitable in optimization and decision problems alike [G+98]. The same applies to proofs of inconsistency (to show that a problem does not have any solution).

When dynamic ordering heuristics such as *Wdeg* are used, any related information (weights attributed to variables and so on) may also be kept during restarts.

Strategies

The restart method can be declined into numerous strategies. These strategies mainly differ in the way the successive restarts are spaced and how this spacing evolves. More precisely, a *cutoff* is set. It can corresponds either to a number of backtracks or to a number of explored nodes in the search tree. When the cutoff is reached, a restart is performed, and the cutoff might then be modified according to the chosen strategy.

A *geometric* restart strategy, as its name implies, make the cutoff grow geometrically at each restart. It is thus parametrized by two values: an initial cutoff and a factor.

The *Luby* strategy, named after Michael LUBY, was designed to offer good performances when information on the estimated runtime necessary to solve a problem is lacking [LSZ93]. Is relies on a somewhat surprising sequence of cutoffs of the following form:

$$1,1,2,\ \ 1,1,2,4,\ \ 1,1,2,\ \ 1,1,2,4,8,\ \ 1,\ldots$$

The authors describe it more formally by stating that “all run lengths are powers of two, and that each time a pair of runs of a given length has been completed, a run of twice that length is immediately executed”. More precisely still, $\mathit{cutoffs} = (t_1, t_2, \ldots)$, where:

$$t_i = \begin{cases} 2^{k-1} & \text{if } \exists k \in \mathbb{N},\ \ i = 2^k - 1 \\ t_{i-2^{k-1}+1} & \text{if } \exists k \in \mathbb{N},\ \ 2^{k-1} \leq i < 2^k - 1 \end{cases}$$

In those definitions, a factor of 2 is used, but the authors explain that any other value can be used. This is the sole parameter of the Luby restart strategy.

2.3 Structural decomposition

Let us consider a constraint network with a set C of constraints. The theoretical time complexity of a simple backtrack approach visiting every possible node of the search tree is in $\mathcal{O}(a \cdot r \cdot |C| \cdot d^n)$, where:

- a is the maximal arity found among the constraints of C;
- r is the maximal size (number of tuples) among the relations associated with the constraints of C;
- d is the maximal domain size among the variables;
- n is the number of variables.

The necessity to improve these theoretical bounds to solve real-life CSPs quickly arose, and persisted even after filtering algorithms were introduced. Indeed, CP approaches sometimes lack scalability.

The basic idea of decomposition is to split the problem into several subproblems, that may even be independent depending on the applied decomposition method.

Many problems have an inherent structure, and being aware of it generally helps solving them. To this avail, structural decomposition methods were proposed to derive profit from the structural information contained in the constraint network. The main approaches use either a tree decomposition or a hypertree decomposition in order to divide the variables of the problem among several clusters, which can be defined as sets of variables.

2.3.1 Structure

Once a problem has been modelled as a CSP, its structure can be extracting rather easily using a few formalisms. As explained in Section 2.1.1, one way to represent the structure of a CSP is to compute its constraint graph. Depending on the situation, though, this might not be sufficient. Another method to capture this overall structure is offered by the notion of *microstructure*.

Definition 2.13. [JÉG93] The *microstructure* of a binary CSP $\mathcal{P} = (X, D, C, R)$ is a graph $\mu(\mathcal{P}) = (V, E)$ such that:

- $V = \{(x, a) \mid x \in X \text{ and } a \in D_x\}$;
- $E = \{\{(x_i, a),\ (x_j, b)\} \mid (a, b) \in R_{ij}\}$, where R_{ij} corresponds to the binary constraint linking x_i to x_j. If there is no such constraint, a relation allowing every tuple ("universal constraint") is considered.

Less formally, the microstructure comprises one vertex for each (*variable*, *value*) couple that is worth considering with regards to the domains of the variables. Edges depict compatibilities between the assignments corresponding to the vertices. This describes the problem at a much lower level than the constraint graph, at the cost of a greater size.

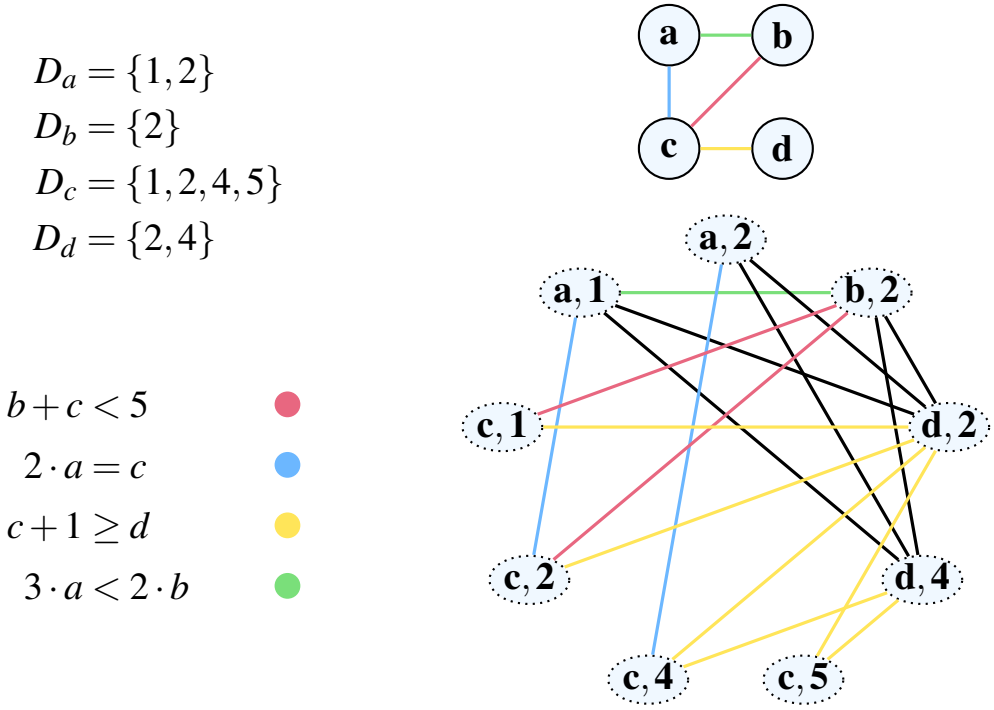


Figure 2.4 – The domains of four variables (from a to d), along with four example binary constraints that link some of them. The graph in the top right corner is the constraint graph, while the graph just below it is the microstructure of the problem. For the sake of readability, each constraint has been associated with a colour. In the microstructure, the colour edges mean that the constraint that links the considered variables does not forbid to simultaneously make the two corresponding assignments. The black edges stand where only the universal constraint holds. There is an edge between $(a,1)$ and $(c,2)$ because 2×1 equals 2, as required by the **blue** constraint. However, no edge stands between $(a,1)$ and $(c,4)$ since 2×1 is not equal to 4: assigning simultaneously 1 to a and 4 to c would violate the constraint $2 \cdot a = c$.

Since there is an edge between two vertices of the microstructure if and only if the corresponding assignments are compatible, each clique in this graph depicts a set of compatible assignments. Consequently, a clique having as many vertices that there are variables in the problem yields a solution to the CSP [JÉG93]. Note that it is actually impossible to find a clique with even more vertices, since there is no edge between vertices that represent assignments of a same variable.

Figure 2.4 shows a set of variables, domains and constraints, as well as the resulting constraint graph and microstructure.

2.3.2 Using a tree decomposition

To use a tree decomposition to decompose a CSP, the decomposition must be computed on the constraint graph (or, if some constraints are not binary, on the 2-section of the constraint hypergraph). Any fill edge added during the triangulation step must of course be removed at the end of the process, since they do not correspond to real constraints of the problem. Still, the clusters of decompositions obtained this way are generally heavily connected sets of vertices, *i.e.*, variables that are linked by numerous constraints.

Once such a decomposition is computed, it can be used in several ways:

- The order in which variables are assigned can be constrained by the decomposition. When a backtrack is performed, the assignments that get undone are generally more related to the failure than if the variables had been assigned in an order that was not constrained by a structural decomposition;
- Since a tree decomposition has, by definition, a tree structure, assigning all the variables of an internal cluster disconnects the graph, thus creating independent subproblems;
- Information can be recorded on the separators between clusters, in order to avoid exploring the same region of the search space twice.

One of the first methods that were proposed to derive profit from a tree decomposition computed on the constraint graph of a CSP was *tree clustering* [DP89]. It basically consists in solving the clusters independently and keeping all their solutions in memory as relations. This latter point unfortunately makes it impractical for most problems. Moreover, solving the clusters is no easy task as well. However, once this step is complete, the problem can be solved in polynomial time by combining the stored partial solutions.

As hinted before, a tree decomposition can be employed to guide the search, especially while using a backtracking approach. This combination is called *backtracking bounded by* (or *on*) *tree decomposition* (BTD) [JT03].

We designate as the *root* of the tree decomposition the cluster that is explored first during the search.

Given a tree decomposition $(\mathcal{C}, \mathcal{T})$ of a CSP (X, D, C) and a root node $r \in V_{\mathcal{T}}$, BTD identifies independent subproblems which are solved separately. More precisely, each subproblem only contains a subset of the initial set of variables.

We will now describe in more details this procedure, also outlined in Algorithm 2.1.

BTD first assigns the variables of the root cluster $\mathcal{C}_r$ (lines 1–2). If there is no assignment of these variables which satisfies the constraints, then the problem is declared inconsistent (line 14). Otherwise, if r has k children v_1, $\ldots$, v_k in the tree $\mathcal{T}$, BTD recursively solves the k corresponding independent subproblems.

Each subproblem is associated with a child v_i of r and contains all vari-

ables that occur in the clusters associated with the nodes of the subtree of $\mathcal{T}$ rooted in v_i (lines 5–6). These k subproblems are independent because, due to the definition of a tree decomposition, no constraint is shared by different subproblems (once the variables of the root cluster have been assigned).

For example, if the decomposition shown in Figure 1.7 were to be used for BTD, assigning the three variables of the cluster $\{2,3,5\}$ would create three independent subproblems, corresponding to the subtrees rooted respectively at the clusters $\{4,5,9\}$, $\{1,2,5\}$, and $\{3,5,6\}$.

The notion of nogoods that was discussed in Section 2.2.4 can be applied in an improved version when a tree decomposition is used. As before, the main goal is to avoid exploring some parts of the search space twice.

Definition 2.14. A *structural good* (respectively, *nogood*) is an assignment of the variables of a separator such that the subproblem associated with the underlying subtree is consistent (respectively, inconsistent).

Indeed, when looking for the source of a failure, the separators between clusters can be readily used as nogoods. Their counterpart, *structural goods*, are used to remember that a subproblem contains at least one consistent full assignment of its variables.

Tree decompositions can also serve as a framework for more complex methods, such as hybrid search strategies [ALL+15].

2.3.3 BTD and optimisation problems

To be able to use a resolution method such as BTD on an optimisation problem, it must first be demonstrated that the problem at hand has a *decomposable* objective function. This condition is met if and only if, when the root cluster C_r of the tree decomposition is fully assigned, it can be ensured that the optimal global solution for this partial assignment can be obtained simply by solving to optimality the subproblems stemming from the children of C_r. This is the case, for example, in weighted constraint satisfaction problems, but many optimisation problems do not meet this condition and thus cannot be solved using BTD.

In the context of optimization problems, the adaptation of BTD requires that we only record *valued goods* instead of structured goods. Valued goods give bounds (a lower bound as well as an upper one) for the subproblems according to the values assigned to the variables of the above separator [DSV06].

More precisely, in an optimization context (let us assume here, without loss of generality, that it is a minimisation problem), every subproblem has its own lower and upper bounds. Before solving a subproblem p, a maximal acceptable cost c for p may be computed by combining the lower bounds of sibling subproblems and subtracting them to the current global upper bound. Then, this cost can be used in a number of ways:

Algorithm 2.1: BTD$\big((X,D,C),\ (\mathcal{C},\mathcal{T}),\ r\big)$

Input: A CSP instance (X,D,C);
A tree decomposition $(\mathcal{C},\mathcal{T})$;
The root node r from $\mathcal{T}$.
Output: *success* if there exist consistent domains $D' \subseteq D$ which assign all variables in all clusters associated with nodes of $\mathcal{T}$;
failure otherwise.

```
1  Let P be the CSP (X,D,C) reduced to the subset of variables from C_r
2  foreach solution S of P do
3      foreach child i of r in T do
4          Let A_i be the tuple of values assigned to the separator C_r ∩ C_i in S
5      while ∄ a child j of r such that A_j is a nogood and ∃ a child i of r such
       that A_i is not a good do
6          Let i be a child of r such that A_i is not a good and T_i be the subtree
           of T rooted in i
7          Let newD be the current domains
8          if BTD((X,newD,C), (C,T_i), i) = success then
9              Record A_i as a good
10         else
11             Record A_i as a nogood
12     if for every child i of r, A_i is a good then
13         return success
14 return failure
```

1. Before even attempting to solve p, we check whether it has a valued good for the current assignment on its separator. If it does have one, the lower bound of this good is considered. If this bound is higher than the maximal cost c that we are ready to pay for p, it becomes obvious that solving p would be useless and a backtrack is performed.
2. During p's solution process, if it can be proved that there is no solution for p that has an objective value better than c, the resolution of p can be stopped altogether, and c yields a new lower bound for p.
3. Conversely, if the resolution of p is successful, both p's lower and upper bounds are updated by using the solution's objective value.

Thus, at least one of the two bounds is shifted each time a subproblem is considered. This ensures each of them is solved a finite number of times, since the lower bound and the upper one will eventually meet.

Figure 2.5 shows an example of these techniques on the previously described tree decomposition.

Another important point is that when solving a problem to optimality using

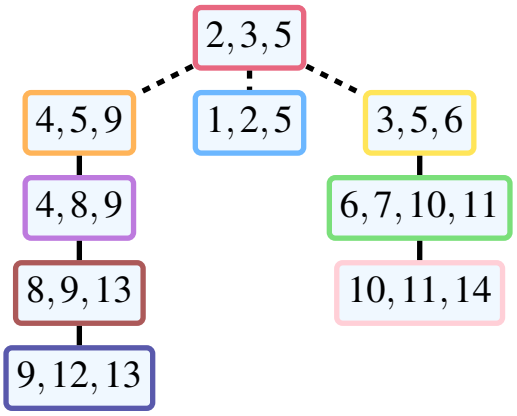


Figure 2.5 – An example on using a tree decomposition. Once the **red** cluster (comprised of x_2, x_3 and x_5, designated by their numbers), chosen as the root, is fully assigned, the three underlying subtrees become completely independent. Then, if we consider for example the first of these subtrees, rooted at the **orange** cluster (x_4, x_5, x_9), we notice that the separator between it and the original root is $\{x_5\}$. If, for example, x_5 has been assigned the value 1 and this subtree has already been explored in these conditions (x_5 being equal to 1), the solver will be able to simply reuse the previous results and immediately proceed to the next subtree.

BTD, consistent assignments of non-leaf clusters must be enumerated. For each of them, the underlying subtree must be recursively solved to optimality. This can prove to be very costly.

In decision problems, structural goods and nogoods bring the time complexity of BTD down to $\mathcal{O}\left(n \cdot m \cdot d^{w+1}\right)$, n being the number of variables, m the number of constraints, d the maximum domain size and w the width of the decomposition. The space complexity is $\mathcal{O}\left(n \cdot s \cdot d^{s}\right)$, s being the size of the largest separator.

When it comes to the optimisation extension of BTD, however, a same subproblem can be solved several times.

Since each new solution phase unavoidably betters the bounds associated with this subproblem, this number of resolutions is bounded by the maximal violation cost k for a WCSP model [DSV06]. This ensures that the number of nodes in the search tree is in $\mathcal{O}\left(k \cdot d^{w+1}\right)$.

An overview of the application of BTD to optimisation problems is offered by Algorithm 2.2.

Note that a valued good $(A_i, \mathrm{LB}, \mathrm{UB})$ comprised both a lower and an upper bound. These bounds are concerned with the optimal objective value for $P_{\mathrm{desc}(i)} - f(A_i)$, where $P_{\mathrm{desc}(i)}$ is the initial COP (X, D, C, f) reduced to the variables belonging to the subtree rooted in i. This ensures that bounds do not redundantly include the values assigned on the separator.

2.3.4 The hierarchy of structural decomposition methods

Tree decomposition makes interesting resolution methods available, with better theoretical complexities. BTD can solve CSPs in $\mathcal{O}(|P|^{w+1})$ time, where $|P|$ denotes the size of the problem and w the treewidth of the tree decomposition. On the other hand, hypertree decomposition (a kind of tree decomposition where each cluster has its own set of constraints [GLS99]) brings this bound to $\mathcal{O}(|P|^{h})$, where h is the width of the decomposition, given by the largest number of constraints in a cluster. Note that h is lower or equal to w.

A few other decomposition methods are worth mentioning for the current section:

Hinge decomposition is based on a tree structure with additional properties [GP82]. It was later combined with tree clustering [GJC94].

Biconnected components is a method using sets of vertices formed in such a way that the subgraph they induce in the constraint network cannot be disconnected by removing only one other vertex [FRE85].

Cycle cutset and hypercutset are methods relying on sets of vertices (or, respectively, hyperedges) whose removal breaks cycles [DEC92].

Figure 2.6 shows a theoretical hierarchy proposed in 2000. It sorts the main decomposition methods according to their respective time complexities [GLS00]. To fully understand this hierarchy, the following concepts are necessary:

Definition 2.15.

- A decomposition method D_1 is said to *generalize* a method D_2 when every problem that is tractable (*i.e.*, solvable in polynomial time) using D_2 is also tractable using D_1.
- A method D_1 is said to *beat* a method D_2 if there exists a class of problems that is tractable using D_1 but not using D_2.
- A method D_1 *strongly generalizes* another method D_2 if it generalizes *and* beats it.
- Two methods D_1 and D_2 are considered equivalent if they generalize each other.

This hierarchy places hypertree decomposition above the other methods, and tree decomposition alongside tree clustering. However, in practice, hypertree decomposition did not prove as efficient as tree decomposition. Among the explanations that have been put forward regarding this lack of efficiency, a notable point is that the constraints must be expressed by means of relations given in extension for such a decomposition to be employed. When applied to CSPs, this generally leads to prohibitively large tables. This is especially problematic when joins operation must be performed.

To obtain complexity bounds that fit observations better, JÉGOU et al. expressed the complexity of the main decomposition methods with respect to

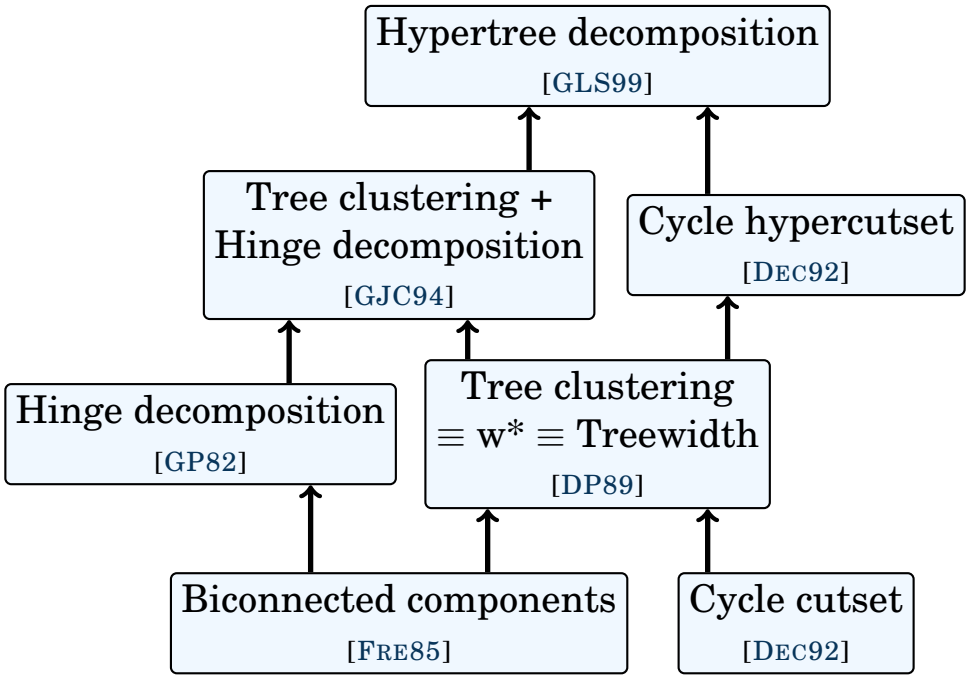


Figure 2.6 – The hierarchy proposed by GOTTLOB, LEONE and SCARCELLO [GLS00]. Arrows mean "is a strong generalisation of".

the size of the relations corresponding to the constraints of the problem [JNT08]. These new complexity bounds allowed them to revise the previously established hierarchy (see Figure 2.7), obtaining a new classification in which hypertree decomposition no longer dominates every other method. According to this new hierarchy, hypertree decomposition is equivalent (see Definition 2.15) to tree decomposition when it comes to solving constraint networks.

Actually, while hypertree decomposition tries hard to minimize, for each cluster, the number of constraints that must be taken into account, JÉGOU et al. proved theoretically as well as experimentally that it is more profitable to use every constraint whose scope contains at least one variable of the considered cluster, in order to filter more efficiently [JNT08]. As it happens, this is exactly what is done with tree decomposition. Since tree decomposition does not rely on join operations or relations given in extension, it can easily handle a larger set of constraints. Thus, tree decomposition is the most interesting decomposition method among those, when it comes to solving CSPs and COPs, and this thesis heavily relies on it.

2.3.5 Decomposing the microstructure

A different approach that still relies on the structure of problems consists in using the microstructure of a CSP as the base for the decomposition [JÉG93].

As explained in Section 2.3.1, since edges in the microstructure denote the fact that two assignments can be performed simultaneously without violating

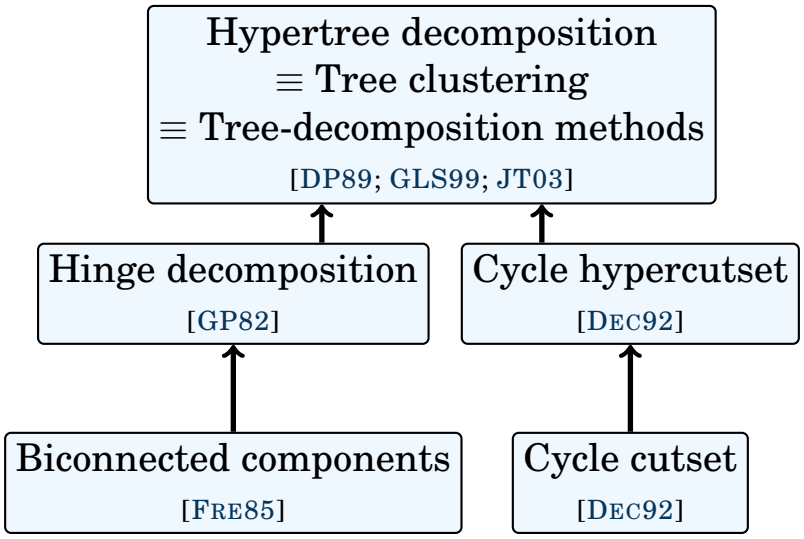


Figure 2.7 – The revised hierarchy, as proposed in [JNT08]. Arrows mean "is a strong generalisation of".

any constraint, a clique in this graph is by definition a valid assignment. It follows that when this assignment is complete, it corresponds to a solution to the problem.

As searching for cliques of a given size is an $\mathcal{NP}$-complete problem in the general case, dividing the microstructure into several graphs can be interesting. When doing so, however, one has to be careful not to split up an existing maximum clique (*i.e.*, a solution).

This approached will be explored in more details when we discuss a concrete case later on.

2.4 Parallelisation

A decomposition does not necessarily have to rely on the structure of the problem. Some techniques, for example, simply split the domains of variables into several subsets, using constraint propagation after each splitting step in order to remove trivially inconsistent subproblems. This allows simple parallelisation approaches to be implemented, as subproblems generated in such a way are independent. Furthermore, the problem can be split in a large number of subproblems, thus making it easier to balance the workload on the available computing units. An approach of this type is proposed, for example, in [MP14; DEP+13; MP13] for the maximum clique problem, or in [RRM13; RRM14] for constraint satisfaction problems in general.

The decomposition into subproblems is computed by selecting a subset of variables and by enumerating the combinations of values of these variables that are not detected as being inconsistent by the propagation mechanism of a CP solver. More precisely, a *depth-bounded depth-first search* (DBDFS) is used

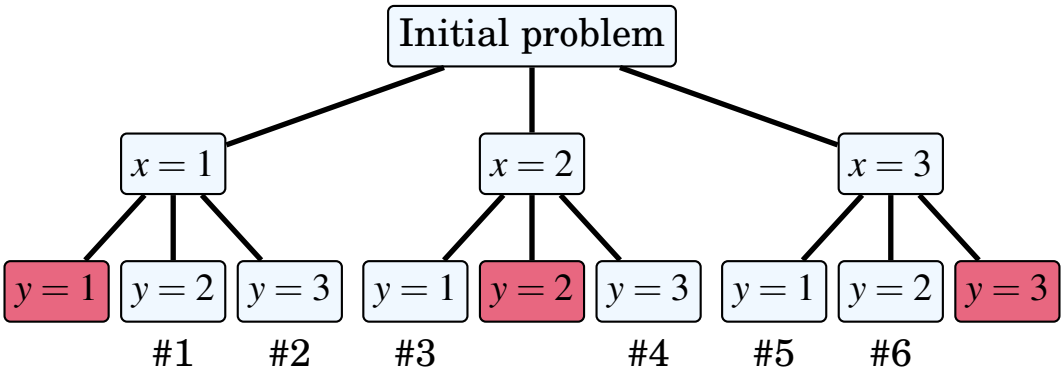


Figure 2.8 – An example of domain splitting for a simple problem. Every variable here has a domain equal to $\{1,2,3\}$, and an *allDifferent* constraint is applied to the full set of variables. Red nodes indicate an inconsistency and will not be visited by the DBDFS. The first generated subproblem comprises the two forced assignments $x \leftarrow 1$ and $y \leftarrow 2$, the second $x \leftarrow 1$ and $y \leftarrow 3$, and so on.

to compute subproblems.

A DBDFS is a depth-first search that never visits nodes located at a depth greater than a given value. First, we consider a static ordering of the variables (usually, by non decreasing domain sizes). Then, the main step of the algorithm is applied: a DBDFS is performed, with a chosen depth p as its limit. This search triggers the constraint propagation mechanism each time a modification occurs. For each leaf of the DBDFS search tree that is not a failure, the first p variables are assigned and so the subproblem defined by this assignment is consistent with the propagation. Thus the set of leaves defines a set S of subproblems. Next, if S is large enough, the decomposition is complete. Otherwise, the main step is applied again until the expected number of subproblems is reached.

Figure 2.8 shows an example of how subproblems can be generated via domain splitting.

In order to balance the workload on the available processing units, the *embarrassingly parallel search* technique (EPS, [RRM13; RRM14]) splits the initial problem as explained, aiming for a very large number of subproblems. The solution time of these subproblems can then be shared by workers: all subproblems are put in a queue and workers take a subproblem from this queue whenever they need work.

Experiments in [RRM13; RRM14] show us that a good decomposition is generally obtained, with EPS, by generating about 30 subproblems per worker. In [RRM14], the average speedup reported with EPS is close to $k/2$, where k is the number of workers, on a large benchmark of instances. This is much better than the speedup obtained with a work stealing approach, where subproblems are dynamically generated by splitting the subproblem currently solved by a worker, whenever another worker has finished its own work. Indeed, work stealing induces more communication between workers, and this can lead to a

significant slow-down. However, even when decomposing the initial problem into 30 subproblems per worker, it may happen that one subproblem is much more difficult than the others and becomes a bottleneck for the speedup. Denoting by t_{max} the time needed to solve the hardest subproblem and by t_0 the time needed to solve the initial problem, it can be stated that the speedup cannot possibly exceed t_0/t_{max}.

Algorithm 2.2: $\text{BTD}_{\text{optim}}\big((X,D,C,f),\ (\mathcal{C},\mathcal{T}),\ r,\ maxcost\big)$

Input: A COP instance (X,D,C,f);
A tree decomposition $(\mathcal{C},\mathcal{T})$;
The root node r from $\mathcal{T}$;
A maximum acceptable cost $maxcost$.
Output: The optimal value for (X,D,C,f) if it is at most $maxcost$;
$maxcost+1$ otherwise.

Let $\mathcal{P}$ be the CSP (X,D,C) reduced to the subset of variables from $\mathcal{C}_r$
$best \leftarrow maxcost+1$
Let Ch be the set of children of r in $\mathcal{T}$
foreach *solution S of $\mathcal{P}$* **do**
  foreach *child $i \in Ch$* **do**
    /* Initialize bounds with values (theoretical, etc.) computed during preprocessing. */
    $\text{LB}_i \leftarrow init\text{LB}_i$
    $\text{UB}_i \leftarrow init\text{UB}_i$
    Let A_i be the tuple of values assigned in S to the separator between $\mathcal{C}_r$ and $\mathcal{C}_i$
    if *a valued good $(A_i,\text{LB},\text{UB})$ exists for A_i* **then**
      $\text{LB}_i \leftarrow \text{LB}$ from $(A_i,\text{LB},\text{UB})$
      $\text{UB}_i \leftarrow \text{UB}$ from $(A_i,\text{LB},\text{UB})$
  if $f(S)+\sum_{i\in Ch}\text{LB}_i < best$ **then**
    while $f(S)+\sum_{i\in Ch}\text{LB}_i < best$ **and** *there is a child $i \in Ch$ such that no optimal valued good exists for the assignment A_i of $\mathcal{C}_i$'s separator in S* **do**
      /* Either A_i is not a valued good, either it is a valued good with $\text{LB} < \text{UB}$. */
      Let i be a child of r such that A_i is not an optimal valued good
      Let $\mathcal{T}_i$ be the subtree of $\mathcal{T}$ rooted in i
      Let $newD$ be the current domains
      $maxcost_i \leftarrow \min\{best-1-f(S)-\sum_{j\in Ch, i\neq j}\text{LB}_j,\ \text{UB}_i-1\}$
      $best_i \leftarrow \text{BTD}_{\text{optim}}\big((X,newD,C,f),\ (\mathcal{C},\mathcal{T}_i),\ i,\ maxcost_i\big)$
      if $best_i \leq maxcost_i$ **then**
        Record the valued good $(A_i,best_i,best_i)$
      else
        Record the valued good $(A_i,best_i,\text{UB}_i)$
    $best_r \leftarrow f(S)+\sum_{i\in Ch}\text{LB}_i$
    if $best_r < best$ **then**
      $best \leftarrow best_r$
return *best*

Chapter 3

Integer linear programming

Contents

3.1 Linear programming

Linear programming may be viewed as a subset of CP. It allows to define constrained optimisation problems but is restricted to linear relationships between variables when it comes to expressing constraints. Furthermore, the objective function must be linear as well. Terms like x^2 or $x \cdot y$, where x and y are variables, are for example prohibited in linear programs. These conditions ensure that a linear program can be represented in *canonical form*:

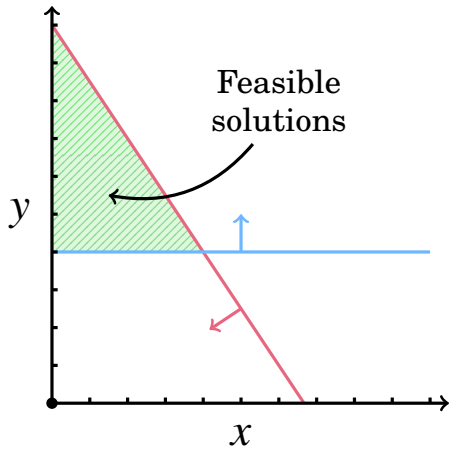
$$\begin{array}{ll} \text{Maximize} & c^T x \\ \text{Subject to} & Ax \leq b \\ \text{and} & x \geq 0 \end{array}$$

where x is a vector of variables, c and b are vectors of known coefficients and A a matrix of known coefficients.

Note that any equality constraint $a = b$ can be expressed by the conjunction of the two constraints $a \leq b$ and $b \leq a$. On the other hand, disequality constraints $a \neq b$ are prohibited as they would make the search space non-convex, thus preventing the use of linear programming's dedicated algorithms.

The linearity precondition is used to design efficient resolution algorithms. As a result, a linear program can be solved in polynomial time provided that the domains are continuous. This is achieved using, for example, some *interior point* methods [KAR84].

Linear programs that only have two variables can easily be represented on two-dimensional graphs, as shown in Figure 3.1. This figure also clearly demonstrates that linear constraints can be perceived as cuts applied to the



Maximize:

$$x + 2y$$

Subject to:

$$\begin{aligned} (c_1) \quad 3x + 2y &\leq 20 \\ (c_2) \quad 3y &\geq 12 \\ x,\ y &\geq 0 \end{aligned}$$

Figure 3.1 – A simple linear program and its graphical representation.

search space, each removing a part of it. The set of feasible solutions, in which the best values will have to be found, corresponds to the intersection of the spared parts of the search space.

On the downside, the fact that constraints have to be linear obviously prevents linear programming from being as expressive as constraint programming.

3.1.1 Simplex algorithm

The *simplex algorithm* has been widely used for decades to solve linear programs. It relies on a matrix representation (called a "simplex tableau") of the problem, using columns to hold the coefficients of variables and one line per constraint. Various operations ("pivots") are performed on the tableau in order to find the optimal solution by gradually eliminating the occurrences of variables.

Even though this algorithm generally performs well in practice, its worst-case complexity is exponential, and efficiency may be extremely poor on some families of linear programs [KM72].

3.1.2 Column generation

When a problem is too large for a solver to handle it (typically, when it has an exponential number of variables), the approach known as *column generation* may be more suitable than the traditional techniques. This notion of columns refers to the simplex tableau previously mentioned.

When using this approach, the initial problem is split into two problems, respectively labelled "master problem" and "subproblem". The master problem is a copy of the initial one, but with less variables: only a subset of them are

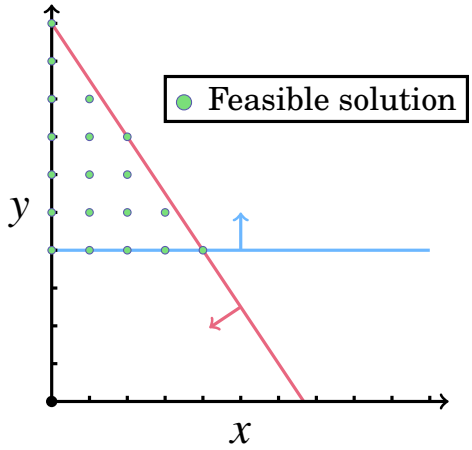


Figure 3.2 – An ILP representation of the problem from Figure 3.1. There are only nineteen valid solutions this time, instead of an infinity.

considered. The subproblem is entirely new and is used to identify variables that will be progressively added to the master problem.

3.2 Integer linear programming

Integer linear programming (ILP), often referred to in the literature simply as "integer programming", is a subset of linear programming in which every variable is required to use an integer value for a solution to be valid (Figure 3.2).

While a linear program with continuous domains may be solved in polynomial time, solving an ILP problem is $\mathcal{NP}$-hard.

3.2.1 Continuous relaxation

The *continuous relaxation* of an ILP problem is the corresponding linear program, with the same constraints, objective function and variables, but without the constraint stating that variables must take integer values. Relaxations are often used as a means to obtain an approximate solution and to deduce properties that can be applied to the problem. The key point is that every solution of the initial ILP problem is still a valid solution in its relaxed counterpart. This ensures that the optimal solution of the relaxed version is a valid upper bound for the initial problem when maximizing the objective function, and a lower bound when minimizing it.

An ILP problem is called *pure* when all coefficients in constraints and in the objective function are integers.

A problem where only a subset of variables must be integers falls within the larger scope of *mixed integer programming*. Another field of research only considers binary variables, which are actually integers with a domain equal

to $\{0, 1\}$.

3.2.2 Cutting plane algorithm

To solve a pure ILP problem, a common basic approach is to apply the following algorithm:

1. Solve the continuous relaxation of the problem.
2. If the optimal solution only uses integer values, stop: the problem has been solved.
3. Generate a *cut*: a constraint that is sure to be satisfied by every integer solution to the problem but that is violated by the optimal solution just found for the relaxation.
4. Add this constraint to the problem and start over from step 1.

When using this algorithm, convergence may be achieved at a more or less fast pace, depending on the cuts that are computed. With improper cuts, an infinite number of steps might even be needed. Fortunately, methods to ensure that this does not happen were found. The first one was described in the 1990s, several decades after GOMORY first proposed using cutting planes [GOM60]. This allowed integer programming to gain popularity by making it more efficient and appealing.

3.2.3 Branch and cut

The *branch and cut* method consists in a combination of branch and bound (see Section 2.2.5) and cutting planes. The branch and bound aspect of this approach consists in splitting the problem into several (usually two) versions by adding complementary inequality constraints. This is done at various points of the search. For example, we may split a problem into two subproblems by stating in the first that x must be strictly greater than 3 while stating in the other that x must be lower or equal to 3. The full extent of the initial problem is thus divided between the two subproblems, without any solution loss.

Furthermore, the relaxation of the ILP model is used to obtain bounds for the optimal value of the objective function.

Chapter 4
Portfolio approaches

Contents

4.1 Algorithm selection principle

When several resolution methods show complementary strengths – each method having a particular set of instances on which it performs well –, it can be interesting to combine them by devising an automatic method selection system. Such an approach falls within the scope of the *per-instance algorithm selection problem*.

More precisely, given a list of solvers (called a *portfolio*) [HLH97; GS01] and an instance i, the per-instance algorithm selection problem consists in picking from the portfolio the solver (or a subset of the portfolio) which can be expected to outperform the others on i [RIC76].

Algorithm selection systems usually build machine learning models to forecast which solver should be used in a particular context. Using the predictions, one or more solvers from the portfolio may be selected to be run sequentially or in parallel.

One of the most prominent and successful systems that employ this approach is SATzilla [XU+08], which defined the state of the art in SAT solving for a number of years. Other application areas include constraint solving [OMA+08], the travelling salesman problem [KOT+15], subgraph isomorphism [KMS16] and AI planning [SEI+12]. The reader is referred to a survey [KOT14] for additional information on algorithm selection.

The selection process generally comprises two steps: given an instance i to be solved, features are extracted from i in order to obtain a global idea of its characteristics; then, algorithm selection is run to choose a solver. The chosen solver can finally be run to try solving the instance i.

As briefly stated before, complementarity is a key point in portfolio approaches. Performances are greatly tied to the chosen set of solvers, as demonstrated by repeated experiments from the literature [KAD+10; XU+08; XHL10].

4.2 Supervised classification

Assigning a solver to each instance in the context of the algorithm selection problem is actually a supervised classification problem. Classification rules must be learned in order to be able to perform these choices.

Supervised classification is a very classical task of machine learning. It consists in associating given examples to classes. The list of existing classes is known beforehand.

Definition 4.1. Let I be the representation space of instances, and $C = \{c_1, \ldots, c_k\}$ a finite set of classes. An *example* is a couple $(i, c) \in I \times C$.

A *training set* allows a *learning algorithm* to build *classification rules* by using already labelled examples.

Definition 4.2. A *training set* is a set of examples $T \subseteq I \times C$ such that for every pair of examples $e = (i, c)$ and $e' = (i', c')$ taken from T, if $i = i'$, then $c = c'$. In other words, identical instances must belong to the same class.

A *validation set*, on the other hand, is employed to assess the results of a classification model by providing instances and comparing the predicted classes with the real ones, that are not known to the model beforehand.

Definition 4.3. A *validation set* is a set of examples $V \subseteq I \times C$ following the same rules as the training set. Training and validation sets must be disjoint (no example can be in both sets).

It is generally advisable to use a training set that is representative with regards to the validation set, especially when classes are not well balanced in terms of sizes.

Definition 4.4. A *learning algorithm* takes a learning set as its input and returns a classification rule.

Note that running a learning algorithm generally takes a significant amount of time.

Definition 4.5. A *classification rule* is an algorithm able to return a predicted class for any given element of the validation set.

The error rate of a classification rule, useful to assess its quality, corresponds to the proportion of examples (i, c) for which the predicted class differs from the actual class c.

Running a classification rule on a validation example is generally significantly faster than the learning step just mentioned.

In some domains or contexts, there are not many examples to work on to begin with. Among the techniques that were designed to enable a successful

learning process in such occurrences, the method known as *cross-validation* is especially noteworthy.

Cross-validation is a technique used to generate several couples made of a training set and a validation set, using a single example dataset.

More precisely, n-fold cross-validation, n being a natural integer, creates n such couples. It proceeds as follows:

1. The initial dataset D is randomly partitioned into n subsets $d_1, \ldots, d_n$. Since it is a *partition*, these subsets are disjoint and their union equals the full initial set.
2. For each subset d_k, a couple is created by using d_k as the training set and $D \setminus d_k$ as the validation set.

In this thesis, we will be using the *leave-one-out* approach. As its name implies, it consists in having a number of folds that exactly matches the number of examples in the initial dataset. Each validation set is thus a singleton. It helps having large training sets while guaranteeing the validity of experiments.

4.3 Applying supervised classification to algorithm selection

In the context of the per-instance algorithm selection problem, the examples are couples formed by at instance and a solver. The solvers correspond to classes, and the class to which a given example belongs is the one associated with the solver that gave the best results on the considered instance.

While in many classification tasks the results are heavily discretized (the example being either assigned to the correct class or to another one), this is not the case when performing automatic algorithm selection: choosing a solver that is not the best suited to solve a given instance might still allow the end-user to get satisfying results and to solve the instance. Choices can thus be evaluated according to degrees of quality. There might even be several criteria to take into account: the time needed to solve the instance, the quality of the best solution found within a given time limit, and so on.

The representation of instances may contain a large variety of features:

- Number of variables, size of domains, number and arity of constraints;
- Features derived from the constraint graph, such as its size and density;
- Results obtained with solvers run for a short amount of time to probe the instance;
- ...

Some numerical features can be enhanced and turned into several features by using statistical tools such as means, medians, minimal and maximal

values.

Tools have been developed specifically for algorithm portfolio and selection approaches, for example to simplify the task of implementing such approaches and evaluating different techniques. One of them is LLAMA, a "modular and extensible toolkit implemented as an R package" aiming at making it easier to explore a wide array of classification and regression techniques, among other things [KOT13]. LLAMA supports the most common algorithm selection approaches used in the literature.

Additionally, LLAMA can train a regression model instead of a classification one if asked to. This approach trains a model that predicts the performance difference between every pair of solvers in the portfolio, similarly to what is done in [XU+08]: if the first solver is better than the second, the difference is positive, otherwise negative. The solver with the highest cumulative performance difference (*i.e.*, the most positive difference over all other solvers) is chosen to be run.

Part II

The maximum common subgraph problem

In many applicative domains, graphs are used to model various structured objects. In such cases, comparing these objects and measuring the similarity between them often amounts to matching the corresponding graphs together. Among the different types of matching problems, a very classical and highly difficult one is the maximum common subgraph problem, where the goal is to identify the largest common part between two considered graphs. This is an $\mathcal{NP}$-hard problem, and complete approaches often struggle to solve instances in which the graphs have more than a few hundred vertices.

This problem will be defined in a more detailed fashion in Chapter 5, along with the existing solution methods.

In this part, we investigate the benefits offered by decomposing instances of this challenging problem into independent subproblems in order to speed up the solution process.

Chapter 6 presents our two new decomposition methods.

The first presented method (Section 6.1) relies on domain splitting. Its aim is to provide a better workload balance for this particular problem than more generic methods, but also to highlight some issues that domain splitting methods often run into when it comes to solving the MCIS problem.

The second method, described through Section 6.2, is a structural approach based on a decomposition of the microstructure of the problem. This provides greater per-instance adaptability faculties. Furthermore, this approach could theoretically be employed for other problems as well.

These new methods will be thoroughly evaluated and compared with the state of the art in Chapter 7. Finally, our contributions, observations and conclusions will be listed in Chapter 8, as a way to close this part relative to the MCIS problem.

Chapter 5

Background and definitions

Contents

In this chapter, we define the maximum common subgraph problem (Section 5.1), before describing two of the state-of-the-art complete approaches for this particular problem: Section 5.2 addresses a constraint programming model, while Section 5.3 explains how the maximum common subgraph problem can be reformulated into a maximum clique problem.

5.1 Graph comparisons

When trying to compare graphs, a common approach is to search for a large common part between these graphs. Graph isomorphisms can be used to define such parts.

Definition 5.1. Let $G = (V, E)$ and $G' = (V', E')$ be graphs. G is *isomorphic* to G' if there exists a bijective function (called *isomorphism*) $f : V \to V'$ which preserves edges, *i.e.*:

$$\forall (u, v) \in V \times V, \quad \{u, v\} \in E \Leftrightarrow \{f(u), f(v)\} \in E'$$

Figure 5.1 shows an example of a pair of isomorphic graphs.

Definition 5.2. A *common induced subgraph* (respectively, *partial subgraph*) of two graphs G and G' is a graph isomorphic to induced (respectively, partial) subgraphs of G and G' (see Definitions 1.5 and 1.6).

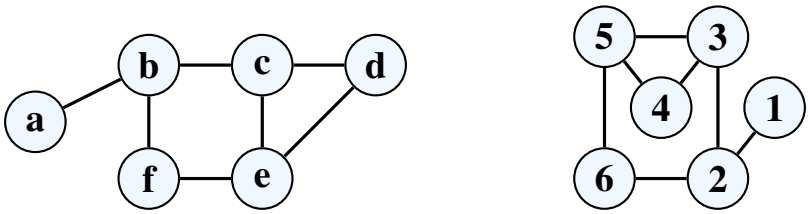


Figure 5.1 – Two isomorphic graphs. An isomorphism function can be defined as follows: $a \leftrightarrow 1$, $b \leftrightarrow 2$, $c \leftrightarrow 3$, $d \leftrightarrow 4$, $e \leftrightarrow 5$, $f \leftrightarrow 6$.

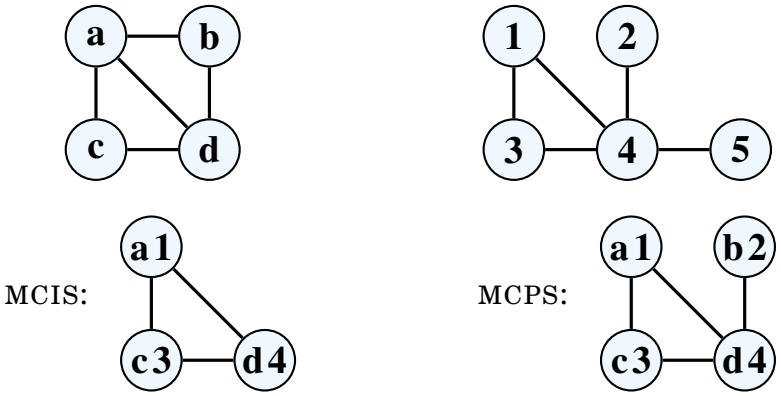


Figure 5.2 – Two graphs, along with a corresponding MCIS and an MCPS. The vertex b cannot be integrated into the MCIS since, in the other graph, there is no node that is linked to both 1 and 4 (vertices matched with neighbours of b).

In many applications, we are mostly interested in finding the largest possible common subgraphs. This can be used for example to define a similarity measure based on the size of these subgraphs (the larger the common subgraphs, the more similar the two graphs). This, however, deserves a discussion: while the size of an induced subgraph can easily be evaluated with its number of vertices, partial subgraphs cannot be considered in the same way. Indeed, given two graphs (V,E) and (V',E'), there always exists a common partial subgraph with $\min(|V|,|V'|)$ vertices (a trivial one having no edge). Therefore, when considering partial subgraphs, their number of edges is usually used to compute their size instead.

Definition 5.3. A *maximum common induced subgraph* (MCIS) is a common induced subgraph which has the maximal number of nodes among existing common induced subgraphs.

Definition 5.4. A *maximum common partial subgraph* (MCPS) is a common partial subgraph which has the maximal number of edges among existing common partial subgraphs.

Figure 5.2 shows examples of MCIS and MCPS.

The maximum common induced and partial subgraph problems consist in finding, for two given graphs, an MCIS or an MCPS, respectively.

Searching for a maximum common subgraph has many applications, for example, in chemoinformatics, bioinformatics, or image processing where it gives a measure of the similarity between objects represented by graphs [RGW02; RW02].

In addition to the fact that it is an $\mathcal{NP}$-hard problem, the number of subgraphs that must be considered is doubly exponential, since we must explore the two graphs simultaneously. In comparison, the similar problem called

subgraph isomorphism is far easier because the subgraph that must be looked for (the "pattern" graph) is known from the very beginning. Thus, the graphs that can be reasonably considered in maximum common subgraph instances rarely exceed a size of a hundred nodes. This limit can be pushed further if the instance uses labels on the nodes or edges, since this reduces the number of possibilities that it makes sense to consider. On the other hand, in the subgraph isomorphism problem, patterns of several hundreds of nodes can be searched for in target graphs of several thousands of nodes [KMS16]. Some other approaches are also very efficient but need certain hypothesis to hold on the instance. For example, when one of the two graphs of the considered pair is very close to the MCIS, dedicated algorithms can find the MCIS must faster than in the general case by proceeding in a way reminiscent of the subgraph isomorphism problem [HMR17].

There exist two main approaches for solving the MCIS problem:

- The first approach explores the search space by branch and bound. This may be achieved by using constraint programming;
- The second approach is based on a reformulation of the MCIS problem into a maximum clique problem.

Details will now be given on those two approaches.

5.2 CP model for the MCIS

One of the first approaches proposed to solve the MCIS problem was to apply branch and bound (see Section 2.2.5) [BB76; McG82]. Solutions are then built incrementally, by starting from a trivial common subgraph of a single vertex and making it grow one vertex at a time. Backtracks are performed whenever the common subgraph cannot be extended this way, and new branches are explored by cancelling any required number of recent choices.

The branch and bound approach can be enhanced by using CP as its framework: less branches will be explored thanks to constraint propagation.

A CSP model for solving MCIS was envisioned and introduced in 2011 [Vis11; NS11]. Given two graphs G and G', this CSP defines:

- A **variable** x_u for each node u of G;
- Equal **domains** for each variable, containing all nodes of G', with the addition of a special value, called $\perp$ ("bottom").

Assigning a value v to a variable x_u means matching vertices u and v. On the other hand, using the value $\perp$ for a variable x_u means that x_u is not used in the common subgraph currently being built: u is not matched with any node of G'.

Moreover, a set of edge **constraints** have to be introduced in order to ensure that variable assignments preserve edges between matched nodes. This

way, every consistent assignment defines a common induced subgraph.

The simplest way to do this is to make sure that, for every pair of vertices $\{u, v\}$ from G:

- Either one of them is not matched to any vertex of G' (then edge preservation is not a concern),
- or they are respectively matched to vertices u' and v' of G' and there is an edge between u and v if and only if there is also one between u' and v'.

This can be formalized by the following constraints:

$$\forall \{u,v\} \subseteq V_G, \;\; (x_u = \perp) \vee (x_v = \perp) \vee (\{u,v\} \in E_G \Leftrightarrow \{x_u, x_v\} \in E_{G'})$$

MCIS being an optimization problem aiming at maximizing the number of matched nodes, elements must be added to guide solvers towards this goal:

- A **variable** $x_\perp$ whose domain is $D(x_\perp) = \{\perp\}$ (it is forced to be assigned the value $\perp$);
- A **variable** *cost* which has its value fixed by the constraint presented in the following item. This will be essential to define the objective function of the CSP;
- A **soft constraint** $softAllDifferent(\{x_u, u \in V_G\} \cup \{x_\perp\}, cost)$. Generally speaking, the *softAllDifferent* constraint has to be given a set of variables as well as an additional variable x. It forces the x variable to be equal to the number of pairs of variables in the given set that use the same value. Therefore, the constraint $softAllDifferent(\{x_u, u \in V_G\} \cup \{x_\perp\}, cost)$ ensures that all x_u variables are assigned to values different from $\perp$ whenever it is possible, and that the *cost* variable is equal to the number of binary difference constraints that are violated within the given set of variables.

When a solution of cost c is found, it is always possible to derive from it a valid matching where c variables are assigned the value $\perp$. For example, the CSP might have a solution comprised of the following assignments: $x_1 \leftarrow 1$, $x_2 \leftarrow 1$, and $x_3 \leftarrow 2$. This does not correspond to a valid matching, since x_1 and x_2 use the same value. Still, a common subgraph can easily be deduced from such a solution, simply by choosing a single variable between these two conflicting variables and assigning it the value $\perp$. This could yield, for example, a solution made of the assignments $x_1 \leftarrow 1$, $x_2 \leftarrow \perp$, and $x_3 \leftarrow 2$. It can be noticed that the cost of this solution is the same as the original one ($cost = 1$, because $x_2 = x_\perp$) [NS11].

The objective of this CSP is to minimize the value of the *cost* variable. It follows from the explanations just given that this is equivalent to maximizing the size of the common subgraph.

The $x_\perp$ variable ensures that if every other variable can be assigned values different from $\perp$, this will necessarily happen. Indeed, since the *softAllDifferent* constraint counts the number of violated binary difference constraints within

its scope, if the $x_\perp$ variable was absent, it would be possible to obtain a null cost even with one variable using the value $\perp$.

Different constraint propagation techniques for this CSP model of the MCIS problem were experimentally evaluated [NS11]. The combination "MAC+*Bound*" generally obtains very good results and outperforms the branch and bound approach [McG82]: *maintaining arc consistency* (MAC) [SF94] is used to propagate hard constraints, while *Bound* checks whether it is possible to assign distinct values to enough x_u variables to surpass the best cost found so far. *Bound* is a weaker version of the generalized arc consistency for the *softAllDifferent* constraint [PRB01] which computes the maximal number of variables that can be assigned distinct values.

5.3 Reformulation of the MCIS problem as a maximum clique problem

We may solve the MCIS problem for two graphs G and G' by introducing their compatibility graph and searching for cliques in it [BY86; Dur+99; RGW02].

Definition 5.5. The compatibility graph of two graphs G and G' is an undirected graph G_C whose set of nodes is $N_{G_C} = V_G \times V_{G'}$ and whose set of edges is:

$$N_{G_E} = \left\{ \{(u,u'),\ (v,v')\} \subseteq N_{G_C} \;\middle|\; (u,u') \text{ and } (v,v') \text{ are compatible} \right\}$$

where two nodes (u,u') and (v,v') of N_{G_C} are compatible if $u \neq v$ and $u' \neq v'$, and if the corresponding assignments, when applied together, preserve edges (*i.e.* $\{u,v\} \in E_G \Leftrightarrow \{u',v'\} \in E_{G'}$).

Intuitively, a compatibility graph outlines, for each pair of variable assignments, whether these two assignments can be performed simultaneously without violating the constraints enforced by the nature of the MCIS problem.

As illustrated in Figure 5.3, a clique in G_C corresponds to a set of compatible matchings of nodes in G and G'. Therefore, such a clique corresponds to a common induced subgraph, and a maximum clique of G_C is a MCIS of G and G'. It follows that any method able to find a maximum clique in a graph can be used to solve the MCIS problem. This yields similar results than a branch and bound approach [McC+16], but with notable difference depending on whether labels are used.

In the case of the MCIS problem, the compatibility graph corresponds to the problem's microstructure when using the CP model previously introduced, provided that the special $\perp$ value as well as the *cost* and $x_\perp$ variables are ignored. Therefore, the observations made on the compatibility graph hold

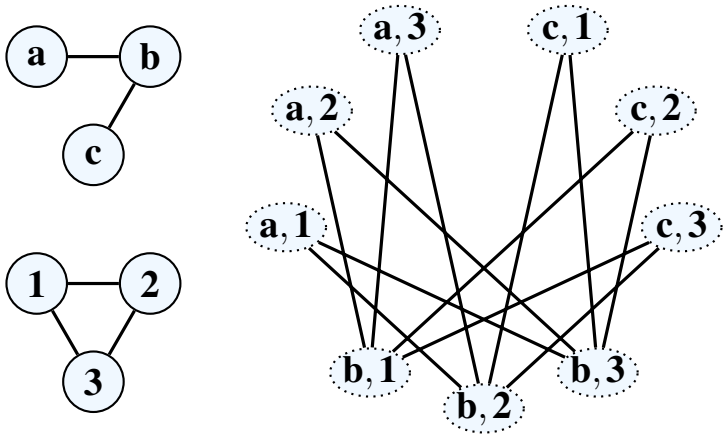


Figure 5.3 – Two graphs and their compatibility graph G_C. For example, since the edge $\{a,c\}$ is not present while $\{1,3\}$ exists, a cannot be matched to 1 when c is matched to 3. Thus, the edge $\{(a,1),\ (c,3)\}$ is not added to G_C. One of the many maximum cliques in G_C is $\{(a,3),\ (b,2)\}$ and it corresponds to an MCIS.

for the microstructure: each clique in the microstructure yields a consistent assignment. In the MCIS problem's context, this means that each clique can be used to build a common subgraph, and that the maximum cliques in the microstructure correspond to solutions of the MCIS problem.

This approach was primarily intended for decision problems. To solve them, one has to find a clique of a given size in the microstructure. In the context of the MCIS, however, the optimal solution corresponds to the *largest* clique in this graph. To find it, a possibility is to extract every maximal clique from the graph and keep the largest, since the maximum cliques of a graph are necessarily also maximal cliques.

Chapter 6
Decomposing the MCIS problem

Contents

The previous chapter introduced the maximum common subgraph problem and explained that it is an especially challenging $\mathcal{NP}$-hard problem. This new chapter aims at providing the reader with ways to speed up the solution process by decomposing instances of this problem.

Section 6.1 describes a domain-splitting strategy. It is derived from the EPS method from Section 2.4 but takes into account the peculiarities of the MCIS problem.

Section 6.2 discusses a structural decomposition method able to use the inherent structure of instances to create more balanced subproblems.

6.1 Binary domain decomposition

6.1.1 Principle

In Section 2.4, we described a decomposition approach allowing one to decompose a CSP into numerous independent CSPs [RRM13]. This decomposition being based on a splitting of the domains of variables, it will be referred to as DOM from here onwards.

The DOM decomposition approach creates subproblems by assigning some variables while removing inconsistent subproblems. When applying this decomposition method to the CP model of the MCIS problem (see Section 5.2), we obtain subproblems by assigning a subset of variables $X' \subseteq \{x_u, u \in N_G\}$ to nodes of G' or to $\perp$. First experiments have shown us that this decomposition leads to very unbalanced subproblems, and therefore very low speedups when solving these subproblems in parallel, even with numerous processing units.

Actually, assigning a variable x_u to a node $v \in N_{G'}$ often strongly reduces variable domains, as the propagation of edge constraints removes from the

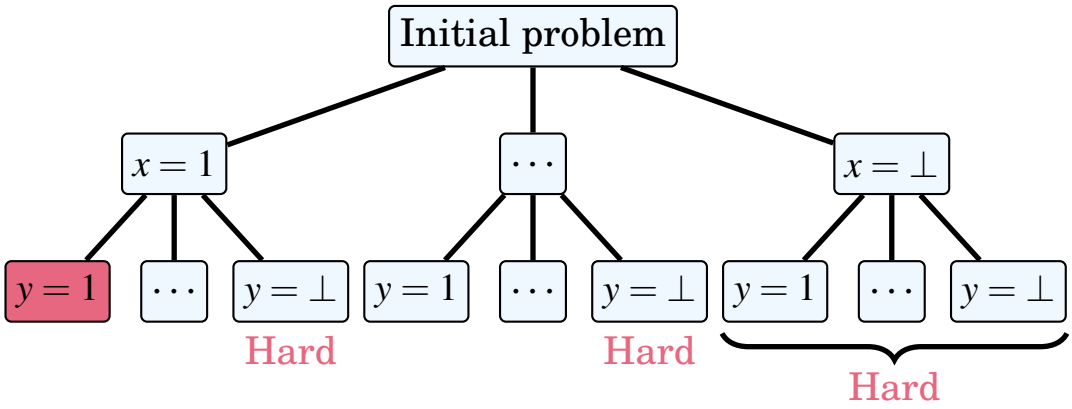


Figure 6.1 – The domain splitting method, applied to an MCIS instance. Subproblems in which variables get assigned the special value $\bot$ will generally be harder to solve due to poor domain-filtering opportunities.

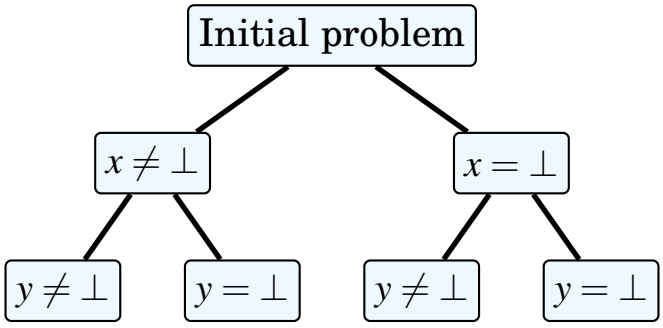


Figure 6.2 – The binary domain splitting method BIN for the MCIS problem.

domains of the variables associated with neighbours of u all nodes of $N_{G'}$ which are not neighbours of v. However, assigning a variable x_u to $\bot$ (*i.e.*, deciding that u will not be matched) never reduces the domains of the other variables. The only exception is when the number of variables assigned to $\bot$ becomes equal to a cost bound: $\bot$ is then removed from all other domains, in order to prevent the common subgraph built to be smaller that the best one found to this point. This behaviour explains why some subproblems are rather easy to solve whereas others (where $\bot$ is used) are much harder, even though they all have the same search space size. Figure 6.1 shows this issue in a more visual way.

In order to try to generate more balanced subproblems, we introduce another way of decomposing domains, called BIN hereafter. It is a straightforward adaptation of the decomposition of [RRM13]. We perform a DBDFS (see Section 2.4), but instead of creating $|N_{G'}|+1$ branches at each node (one for each possible value in the domain of the variable), we only create two branches: one where the variable is assigned $\bot$, and one where $\bot$ is removed from the variable's domain. This approach is outlined in Figure 6.2.

Additionally, Algorithm 6.1 shows in a more detailed manner how to carry out a BIN decomposition.

Algorithm 6.1: BIN decomposition

Input: A CSP model P for an MCIS instance with graphs of nbv vertices, as described in Section 5.2;
A target number $goal$ of subproblems;
A lower bound lb on the size of an MCIS for the considered instance.

Output: A set S of about $goal$ subproblems, with a global optimal solution corresponding to that of the initial problem.

1 $S \leftarrow P$
2 $depth \leftarrow 0$
3 **while** $\left|2 \cdot |S| - goal\right| < \left||S| - goal\right|$ **and** $depth < nbv - lb - 1$ **do**
4 $S_{\text{new}} \leftarrow \varnothing$
5 **foreach** $s \in S$ **do**
6 Let x be the unassigned variable in s with the smallest domain
7 Add to S_{new} two subproblems: one corresponding to s with the additional constraint "$x = \bot$" and one with "$x \neq \bot$
8 $S \leftarrow S_{\text{new}}$
9 $depth \leftarrow depth + 1$
10 **return** S

The condition of the outer loop, line 3, aims at finding the appropriate depth for the decomposition, *i.e.* the number of variables that should be subject to a preprocessing in order to obtain a number of subproblem close to the goal. This is done by making sure an additional splitting phase would not bring us further away from the goal than our current situation. Note that we also ensure that the number of variables assigned to $\bot$ in a subproblem never gets large enough to prevent us from building a subgraph that would beat the initial lower bound ($depth < nbv - lb - 1$).

The inner loop at lines 5–7 creates the required subproblems each time a splitting phase is deemed necessary, and replaces the old set of subproblems with the new ones.

6.1.2 Complexity of BIN and DOM

Let S be the set of subproblems computed by BIN. Computing all these subproblems is done in $\mathcal{O}\big(|S| \cdot \log(|S|)\big)$ time. The logarithmic part stems from the DBDFS algorithm, used in both DOM and BIN. Indeed, the number of steps needed to obtain $|S|$ subproblem is logarithmically bounded, since each time a variable is chosen to become subject to a split, every current subproblem gets divided. In BIN, the complexity of each of these logarithmically numbered steps

is bounded by $|S|$, since it corresponds to the largest number of subproblems that might be generated at a given time (the last step, actually).

Once the subproblems have been generated, each of them is solved using MAC+*Bound*. Since the subproblems define a partition of the search space, MAC+*Bound* explores the whole search space once while solving all subproblems.

All this considered, the overall time complexity of BIN is in $\mathcal{O}\big(|S| \cdot \log(|S|) + d^3 \cdot n^2 \cdot d^n)\big)$, where d is the size of the largest domain in the CSP and n its number of variables.

On the other hand, DOM uses a MAC+*Bound* bounded at depth p to compute its set of subproblems S in $\mathcal{O}\big(|S| \cdot d^3 \cdot n^2\big)$ while ensuring consistency. Subproblems are then solved using the standard MAC+*Bound*, bringing the overall time complexity bound of DOM to $\mathcal{O}\big(|S| \cdot d^3 \cdot n^2 + d^3 \cdot n^2 \cdot d^{n-p}\big)$. The $d^3 \cdot n^2$ factors come from the use of MAC. The $n - p$ exponent reflects the fact that descending deeper during the DBDFS reduces the number of variables that will have to be actually handled during the resolution of the subproblems.

6.2 Structural decomposition

6.2.1 TR-decomposition principle

We now introduce a new way to decompose the MCIS problem into independent subproblems while taking into account the structure of the problem. This method, called STR, is an adaptation of a decomposition devised in 1993 called TR-decomposition [JÉG93]. We apply it to the CP model proposed in 2011 for the maximum common subgraph problem [NS11].

TR-decomposition is a decomposition method based on the microstructure of the CSP. As explained in Section 5.3, the MCIS problem can be reformulated into an instance of the maximum clique problem when considering the microstructure. TR-decomposition builds up on this idea. To make it easier to find maximum cliques, the microstructure is triangulated, much like in the process described in Section 1.4.2 for building a tree decomposition. Once again, the main interests of this triangulation step are the linear number of maximal cliques that it yields and the fact that these cliques are easy to enumerate (using the simplicial order created by the triangulation algorithm *MinFill*).

TR-decomposition for the MCIS problem

While TR-decomposition was initially defined for binary CSP instances, the CP model of [NS11] is a *soft* CSP. Nevertheless, TR-decomposition can still be employed to solve the MCIS problem by considering the compatibility graph instead of the microstructure. Like JÉGOU ([JÉG93]), we propose to use the

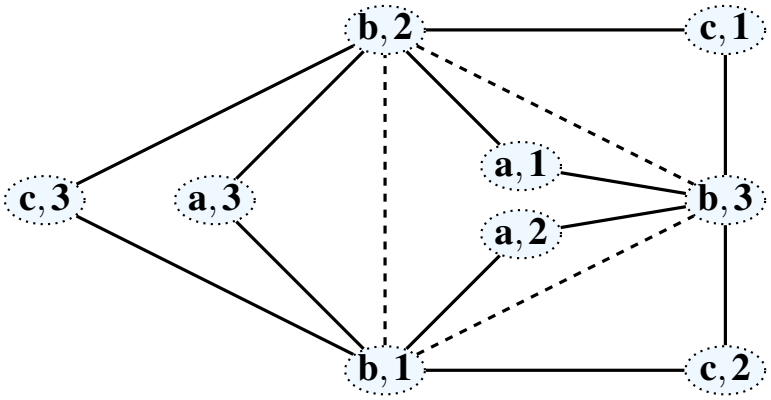


Figure 6.3 – A triangulated version of the compatibility graph of Figure 5.3. The edges that were added ("fill edges") are drawn with dashed lines.

class of triangulated graphs to take advantage of their property to have a few maximal cliques.

Let n be the number of variables in the CSP instance. JÉGOU proved that a solution of the problem corresponds to an n-clique in the microstructure and that, conversely, an n-clique in the microstructure gives a solution of the problem. Since finding an n-clique in a given graph is an $\mathcal{NP}$-complete problem, it is customary to use particular classes of graphs that are known to have only a limited, polynomial number of maximal cliques [JÉG93; CJK03].

More precisely, we triangulate the compatibility graph with the *MinFill* algorithm [KJA90]. As explained in Section 1.3, this algorithm adds edges, called *fill edges*. The fill edges add erroneous compatibilities. Therefore, a maximum clique in the triangulated version of the microstructure is not invariably the best solution anymore, but simply is a *subproblem* in which we might find solutions – or no solution at all, depending on the subproblem.

Figure 6.3 shows an example of a triangulated compatibility graph.

Actually, a maximum clique of the original compatibility graph is still a clique (though not necessarily maximum or even maximal) in the triangulated graph. Moreover, it is bound to appear in at least one maximal clique of the triangulated compatibility graph. This ensures that we cannot possibly miss a solution [JÉG93; CJK03].

Resulting subproblems

Each maximal clique of the triangulated graph defines a subproblem (actually an induced subgraph of the compatibility graph) in which we may find a maximum clique of the original compatibility graph. Such a subproblem can be seen as an MCIS instance, involving a pair of smaller graphs, that can be solved using the CP model of [NS11].

More precisely, given a maximal clique K of the triangulation of the compat-

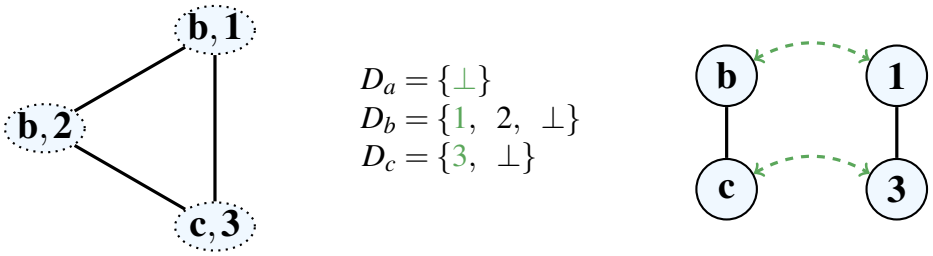


Figure 6.4 – A clique of the triangulated compatibility graph and a common subgraph that can be found in it by solving the corresponding CSP. Since no vertex in the clique involves the vertex a from the first of the two compared graphs, the domain for a in the corresponding CSP only contains $\bot$, and a therefore has no way to appear in this common subgraph since it cannot be matched with any node of the second graph involved in the comparison.

ibility graph associated with G and G', we define a subproblem which has the same variables and constraints as the initial problem. However, the domain of every variable x_u is restricted to:

$$D(x_u) = \{v \in N_{G'} \mid (u,v) \in K\} \cup \{\bot\}$$

The domains of $x_\bot$ and *cost* remain unchanged. For example, in Figure 6.3, the subset of nodes $\{(b,1),\ (b,3),\ (c,2)\}$ is a maximal clique of the triangulated graph. In the associated subproblem, $D(x_b) = \{1,3,\bot\}$, $D(x_c) = \{2,\bot\}$, and $D(x_a) = \{\bot\}$. Every solution of every subproblem corresponds to a maximal clique of the compatibility graph, and therefore to a common induced subgraph. In our example, the subproblem has two solutions of cost 1, namely $\{x_a \leftarrow \bot,\ x_b \leftarrow 1,\ x_c \leftarrow 2\}$ and $\{x_a \leftarrow \bot,\ x_b \leftarrow 3,\ x_c \leftarrow 2\}$. These solutions are optimal, and correspond to two maximal cliques of the compatibility graph: $\{(b,1),\ (c,2)\}$ and $\{(b,3),\ (c,2)\}$. The edge linking $(b,1)$ to $(b,3)$ cannot be used as it is a fill edge, added during the triangulation step.

Figure 6.4 shows an alternative concrete example of a constraint programming subproblem stemming from a TR-decomposition, along with a solution.

A final note on this topic should be made with regards to the order in which subproblems are considered. As pointed out in the literature, when solving an optimization problem such as the MCIS problem with a parallel approach, a key point is to solve the most promising subproblems first. They are likely to contain solutions offering a good objective value, and finding such solutions will allow to prune more branches of the search tree when solving the remaining subproblems [MP14].

6.2.2 Balancing the size of subproblems

STR may generate subproblems with very unbalanced sizes. In order to generate better balanced subproblems while also exerting some control over the

number of generated subproblems, we propose to recursively decompose the largest subproblems by running TR-decomposition on them again. This procedure is described in Algorithm 6.2:

1. We first triangulate the compatibility graph G_C and store all its maximal cliques in a set S (lines 1–3).
2. Then, while $|S|$ is lower than the required number n of subproblems, we remove from it its largest clique K, triangulate the subgraph of G_C induced by K and compute the set S_K of all its maximal cliques (lines 6–10).
3. If there is only one maximal clique in the triangulation of the subgraph of G_C induced by K, it means that K could not be decomposed any more. We store this information in order to avoid trying to decomposing K again (lines 11–12).
4. If the decomposition yielded several cliques (stored in S_K), we then consider them. Some of these cliques may actually be subsets of cliques that already are in S. These non-maximal cliques are not added to S; all other cliques of S_K, however, are stored in S alongside the others (lines 13–16).
5. Finally, once S contains enough cliques (or no decomposable clique remains in S), we build a subproblem for each clique, and return this set of subproblems (line 17).

Algorithm 6.2: STR decomposition

Input: Two graphs G and G';
The wanted number n of subproblems.
Output: A set of subproblems.

```
1  G_C ← compatibility graph for G and G'
2  G_T ← MinFill(G_C)
3  S ← set of all maximal cliques of G_T
4  Mark all cliques of S as "decomposable"
5  while |S| < n and S contains "decomposable" cliques do
6      K ← largest "decomposable" clique from S
7      Remove K from S
8      G_{C↓K} ← subgraph of G_C induced by K
9      G_{T↓K} ← MinFill(G_{C↓K})
10     S_K ← set of all maximal cliques of G_{T↓K}
11     if S_K = {K} then
12         Mark K as "non decomposable"
13     foreach clique K' ∈ S_K do
14         if ∀K'' ∈ S, K' ⊈ K'' then
15             Add K' to S
16             Mark K' as "decomposable"
17 return the set of all subproblems associated with cliques of S
```

A limit is set on the number of subproblems. Experiments showed that each of these multiple TR-decompositions creates a reasonable number of subproblems, thus enabling us to get very close to the chosen limit. Overall, our structural decomposition gives a satisfying control of the final number of subproblems.

On a side note, applying TR-decomposition on a subproblem corresponding to n nodes in the compatibility graph cannot lead to the creation of more than n subproblems, according to the properties already mentioned regarding the maximal cliques of a triangulated graph.

6.2.3 Mitigating redundancies

Due to the sheer number of generated subproblems and to the way STR decomposition was designed, numerous redundancies might appear within this set of subproblems. The resolution may be facilitated by sorting out these redundancies using a few simple techniques.

Filtering

If a given subproblem has enough variables with a domain containing only the value $\perp$, it can be proven that it does not contain any worthwhile solution. More precisely, one can compute a lower bound lb of the size of a maximum clique in the microstructure in order to determine a minimal number of node for the MCIS. Such bounds can be obtained in reasonable time, using heuristics such as ant colony optimisation algorithms [SF06]. Once this bound is found, subproblems in which less than $lb+1$ variables have a domain different from $\{\perp\}$ can be discarded, since there is no way for such a subproblem to allow a solver to build a common subgraph larger than the one provided by the approximate algorithm run beforehand.

Fusion

Some maximal cliques may have large intersections. In such cases, the corresponding subproblems have comparably large intersections, meaning that we might basically end up solving a same part of the problem several times. In order to reduce these redundancies, some subproblems can be fused, *i.e.* they are replaced by a new subproblem obtained by merging their variables' domains (see Figure 6.5 for an example). Note that any solution will still be present in the resulting subproblem, since domains can only grow bigger during this process.

To prevent subproblems from growing back to the size of the initial problem, we introduce the notion of *gain*, which expresses the evolution of the global problem size during a tentative fusion.

Subproblem S_a	Subproblem S_b	FUSION(S_a, S_b)
$\begin{cases} D_x = \{1,2,\perp\} \\ D_y = \{1,\perp\} \end{cases}$	$\begin{cases} D_x = \{1,2,3,\perp\} \\ D_y = \{2,\perp\} \end{cases}$	$\begin{cases} D_x = \{1,2,3,\perp\} \\ D_y = \{1,2,\perp\} \end{cases}$

Figure 6.5 – The fusion of two subproblems. Since available values are combined, every solution of both initial problems are preserved.

Definition 6.1. Let S_a and S_b be two distinct subproblems. The *gain* offered by the fusion of S_a with S_b is given by the following quotient:

$$\text{gain}(S_a, S_b) = \frac{\text{size}(S_a) + \text{size}(S_b)}{\text{size}(S_a \cup S_b)}$$

where $\text{size}(S_i)$ is the product of the sizes of variable domains of S_i, and $S_a \cup S_b$ represents the subproblem that would be created if S_a and S_b were to be merged.

We proceed to merge two subproblems only if the corresponding gain is greater than 1. When multiple suitable fusions can be found, priority is given to pairs offering the largest gains.

Chapter 7

Experimental evaluation

Contents

7.1 Experimental setup

Programs were written in C, compiled using GCC with `-O3` optimisation, and run on an Intel® Xeon® CPU E5-2670 at 2.6 GHz processor, with 20 480 KB of cache memory and 4 GB of RAM.

The subproblems, for every method, were solved using the CP model introduced in 2011 by NDIAYE and SOLNON for the MCIS [NS11], with the MAC+*Bound* consistency level.

In the experimental study that is to follow, one of our goals is to evaluate the capability of DOM, BIN and STR to generate balanced independent subproblems. As we want to avoid introducing biases, we did not use any particular ordering heuristics regarding the choices of the most promising subproblems. Furthermore, it is worth noting that heuristic algorithms are able to find optimal or near-optimal solutions to the MCIS problem very quickly. However, proving the optimality of such solutions with our baseline sequential approach (MAC+*Bound* [NS11]) remains challenging.

For each instance, we began by evaluating STR. This method aims to generate a number of subproblems close to a parameter k by means of iterative clique decompositions. In the following experiments, k has been set to 1 500. This set of cliques is then cut down to a new number of k' by fusing some subproblems, as explained in Section 6.2.3. To keep our study as fair as possible, BIN and DOM are then specifically asked to generate the same number k' of subproblems, thus solving about as many subproblems as STR. Once the number of subproblems is fixed in this way (with a possibly different number

for each instance), we try to solve those instances with different numbers of subproblems per workers by altering the number of available workers while keeping the total number of subproblems unchanged.

7.1.1 Initial lower bound

For each instance, the solution process begins with a run of an ant colony optimisation algorithm ([SF06]) on the microstructure of the problem in order to find a large maximal clique. This clique provides a lower bound on the size of the MCIS. This bound is then used both to speed up the search and to discard trivially uninteresting subproblems. For 90 % of the instances we deemed interesting in the initial benchmark, it found solutions that eventually appeared to be optimal. Therefore, the main task addressed here is proving the optimality of solutions rather than finding new ones.

7.2 Benchmark

7.2.1 Overview

Our experiments for the MCIS problem were run on 109 instances taken from the benchmark created by CONTE, FOGGIA and VENTO in 2007 [CFV07]. The full original benchmark is comprised of 81 400 instances, divided into a hundred qualitatively equivalent sets. We used several criteria to bring this huge number of instances down to a number that would allow more extensive experiments:

1. We picked four of those hundred sets (those bearing the numbers 2, 3, 4 and 5);
2. Since instances are grouped according to the relative sizes of their MCIS, we only considered those in which the common subgraph represented **from 10 to 30** % of the graphs to be compared;
3. Instances that could be solved in less than **a hundred seconds** by our baseline method (CP solver without decomposition) were deemed to easy and removed;
4. Conversely, we removed instance that could *not* be solved by the baseline in less than **three hours**.
5. Since the ant colony optimisation algorithm used to obtain an initial bound was very efficient, we decided to focus on performing proofs of optimality and selected the instances for which this heuristic was able to find the optimal solution.

Only 109 instances remained after this selection process. They are listed in Table 7.1.

Table 7.1 – Instances used for the maximum common subgraph problem. The name `mcs10_m3D_s60.04`, for example, means that the instance has an MCIS including ten per cent of the vertices of the graphs, is a three-dimensional mesh, has 60 vertices per graph, and belongs to the fourth series of the benchmark.

bvg	mesh	rand
mcs10_b03m_s40.02	mcs10_m2Dr4_s40.02	mcs10_r01_s50.02
mcs10_b03m_s40.03	mcs10_m2Dr4_s40.03	mcs10_r01_s50.03
mcs10_b03m_s40.04	mcs10_m2Dr4_s60.03	mcs10_r01_s50.05
mcs10_b03m_s40.05	mcs10_m2Dr6_s40.02	mcs10_r01_s60.04
mcs10_b03_s40.02	mcs10_m2Dr6_s50.05	mcs10_r01_s60.05
mcs10_b03_s40.03	mcs10_m2Dr6_s60.02	mcs10_r01_s70.02
mcs10_b03_s40.04	mcs10_m2Dr6_s60.05	mcs10_r01_s70.03
mcs10_b03_s40.05	mcs10_m2D_s40.02	mcs10_r01_s70.04
mcs30_b03m_s35.02	mcs10_m2D_s40.04	mcs10_r01_s70.05
mcs30_b03m_s40.02	mcs10_m2D_s40.05	mcs10_r01_s80.05
mcs30_b03m_s40.03	mcs30_m2Dr4_s50.03	mcs10_r02_s90.02
mcs30_b03m_s40.04	mcs30_m2Dr6_s50.03	mcs10_r02_s100.02
mcs30_b03m_s40.05	mcs30_m2Dr6_s60.03	mcs10_r02_s100.03
mcs30_b03_s30.03	mcs30_m2Dr6_s60.04	mcs10_r02_s100.04
mcs30_b03_s40.04	mcs10_m3D_s60.04	mcs10_r02_s100.05
mcs30_b06m_s50.02	mcs30_m3Dr2_s40.02	mcs10_r005_s30.05
mcs30_b06m_s50.03	mcs30_m3Dr2_s40.03	mcs10_r005_s35.05
mcs30_b06m_s50.05	mcs30_m3Dr2_s40.04	mcs10_r005_s40.02
mcs30_b06m_s60.02	mcs30_m3Dr4_s40.04	mcs10_r005_s40.04
mcs30_b06m_s60.03	mcs30_m3Dr4_s40.05	mcs10_r005_s60.05
mcs30_b06_s50.02	mcs30_m3Dr4_s70.04	mcs30_r01_s70.02
mcs30_b06_s50.03	mcs30_m3Dr6_s40.02	mcs30_r01_s70.03
mcs30_b06_s50.04	mcs30_m3Dr6_s70.03	mcs30_r01_s70.04
mcs30_b06_s60.02	mcs30_m4Dr2_s50.03	mcs30_r01_s70.05
mcs30_b06_s60.04	mcs30_m4Dr6_s60.02	mcs30_r01_s80.02
mcs30_b06_s60.05	mcs30_m4D_s50.03	mcs30_r01_s80.03
mcs30_b06_s70.05		mcs30_r01_s80.04
mcs30_b09m_s50.03		mcs30_r01_s80.05
mcs30_b09m_s50.04		mcs30_r01_s90.02
mcs30_b09m_s60.02		mcs30_r01_s90.03
mcs30_b09m_s60.03		mcs30_r01_s90.04
mcs30_b09m_s70.02		mcs30_r01_s90.05
mcs30_b09m_s70.04		mcs30_r01_s100.02
mcs30_b09m_s70.05		mcs30_r01_s100.03
mcs30_b09_s50.02		mcs30_r01_s100.04
mcs30_b09_s60.04		mcs30_r01_s100.05
mcs30_b09_s70.02		mcs30_r005_s30.03
mcs30_b09_s70.03		mcs30_r005_s40.02
mcs30_b09_s70.04		mcs30_r005_s40.04
mcs30_b09_s70.05		mcs30_r005_s60.03
mcs30_b09_s80.02		mcs30_r005_s60.04
mcs30_r005_s60.05		

7.2.2 Labels

This benchmark proposes optional labels for edges and vertices alike. Basically, a vertex having a certain label can only be matched with a vertex bearing the exact same label. As for edges, those that stand between matched pairs of vertices must have the same label in both graphs. Since these labels make instances easier to solve by providing straightforward domain-filtering opportunities, we applied these labels in order to be able to include instances of more varied sizes to our benchmark.

These labels can be parametrized with a percentage p. Given an instance where two graphs $G = (V,E)$ and $G' = (V',E')$ must be searched for an MCIS, the number of different label types is computed so as to be equal to $|V| \times p\%$. Note that, in this benchmark, the two graphs of any given instance always have the same number of nodes; thus, in our example, $|V| = |V'|$. Therefore, for a pair of graphs in which each have 50 vertices and a p value of 15%, there will be 7.5 (rounded down to 7) different kinds of labels for vertices and edges.

To ease comprehension, labels can be seen as colours or natural integers. As briefly explained before, a vertex with a label l can only be matched with vertices bearing the label l. Regarding edge labels, the basic idea is that for a node x to be matchable with a node x' when a node y is matched with y', the label on the edge $\{x,y\}$ must be the same as the one on the edge $\{x',y'\}$ [NS11].

Vertex labels provide a way to cut initial domains with little to no effort, while edge labels allow additional filtering to be performed every time new couples of linked vertices are matched, thus gradually speeding up the search.

7.2.3 Classes

Three different classes of graphs can be found in the benchmark, each divided into subclasses according to the value of a few parameters.

Bounded valence graphs (`bvg`)

In this class, every node has a degree lower than a given threshold, counting both inbound and outbound edges. This threshold is called *valence*. The authors used three different values for the valence: 3, 6, and 9.

This class also includes instances with introduced irregularities (*irregular bounded valence graphs*). These are created by first computing a standard bounded valence graph and then by moving some of its edges to other vertices, thus keeping the average valence unchanged by allowing vertices to go beyond the initially fixed limit.

Randomly connected graphs (`rand`)

Graphs from this class do not have a particular structure by design. This class has been introduced to model applications in which each entity can establish relations with any other entity, independently of their relative positions: the probability of an edge connecting two nodes is independent from the nodes themselves. Note that this is highly reminiscent of what is known as the Erdős–Rényi models.

In this benchmark, the edge-existence probability is fixed for the whole graph through a chosen edge density η. The authors considered three different values for η: 0.05, 0.1 and 0.2.

Meshes (`mesh`)

Some applications involve graphs with an especially regular structure (for example, in the lower levels of a vision task). Furthermore, it is generally agreed that graphs with a regular structure are difficult to solve using general graph matching algorithms, since many parts look alike while only leading to suboptimal solutions [ULL76].

This benchmark includes *mesh connected graphs*, in two, three and four dimensions. In two dimensions, each node (save for those standing at the edge of the mesh) is connected with its four neighbours. In three-dimensional meshes (respectively, four-dimensional), this number of neighbours rises up to six (respectively, eight).

Alongside these regular meshes, the benchmark contains *irregular mesh-connected graphs*, obtained by adding distortions to regular meshes. The authors added edges between randomly selected nodes according to a uniform distribution. For a pair of graphs in which each have N nodes, $\rho \times N$ edges are added, ρ being a constant greater than zero. The ρ values that have been used to generate instances for the initial benchmark are 0.2, 0.4 and 0.6.

7.3 Results

7.3.1 Decomposition time and reduction of the search space

When the initial instance gets decomposed into subproblems, some inconsistent values are filtered out, either by propagating constraints during the DBDFS (for DOM and BIN) or when generating subproblems with new domains from the maximal cliques of the STR decomposition method.

To be beneficial, our decomposition should be performed in a reasonable amount of time while still providing a significant reduction of the search space. Table 7.2 show that these conditions are generally met by STR, which results

Table 7.2 – Reduction of the search space on average on the benchmark using STR decomposition, along with the decomposition time given relatively to the baseline's resolution time without decomposition. s_0 is the size of the instance (the product of the domains' sizes), s_{all} is the sum of the sizes of every subproblem for a given instance, t_{dec} is the time needed to decompose the instance, and t_0 is the time used by the baseline to solve the instance without decomposition.

	bvg		mesh		rand		All	
	$\frac{s_0}{s_{\text{all}}}$	$\frac{t_{\text{dec}}}{t_0}$	$\frac{s_0}{s_{\text{all}}}$	$\frac{t_{\text{dec}}}{t_0}$	$\frac{s_0}{s_{\text{all}}}$	$\frac{t_{\text{dec}}}{t_0}$	$\frac{s_0}{s_{\text{all}}}$	$\frac{t_{\text{dec}}}{t_0}$
DOM	$3e^{2}$	$2e^{-4}$	$1e^{2}$	$1e^{-4}$	$1e^{3}$	$6e^{-4}$	$6e^{2}$	$4e^{-4}$
BIN	1	$3e^{-5}$	1	$3e^{-5}$	1	$1e^{-4}$	1	$7e^{-5}$
STR	$5e^{8}$	$2e^{-2}$	$6e^{5}$	$3e^{-2}$	$1e^{6}$	$8e^{-2}$	$2e^{8}$	$5e^{-2}$

in a stronger reduction of the search space than with DOM, but at a higher cost. BIN, on the other hand, cannot provide any search space reduction due to the way it was defined, but its computational cost is null.

DOM reduces the search space by a factor of 600 on average for all instances in all classes, while STR boasts a factor of 200 millions in the same context.

This factor varies depending on the instance class considered. In particular, for DOM, the search space reduction is lower on `bvg` and `mesh` instances, whereas it is higher on `rand` instances. A similar phenomenon can be observed regarding STR, which provides a lesser reduction on `mesh` and `rand` instances, with better performances mostly confined to the `bvg` instance class.

Search space reduction should obviously be considered together with the time t_{dec} actually spent to perform the considered decomposition. The decomposition times registered for DOM and BIN appear to be very similar, representing less than 0.04 % of t_0 (on average for the whole benchmark) for DOM and even 0.007 % for BIN. Not surprisingly, decomposition times of STR tend to be significantly higher, resulting in an average time of 5 % of t_0.

7.3.2 Speedup bounds

Ignoring decomposition times

We will first examine the resolution times by ignoring the time needed to perform any decomposition step involved in the solution process. More precisely, we compare the speedups of the different methods.

The ratio $t_0/t_{\max}$ between the time needed to solve the initial problem and the time needed to solve the hardest subproblem provides a first upper bound on the speedup.

Table 7.3 – The minimal, average and maximal speedups obtained on each class of instance by each method, by ignoring the decomposition time.

	bvg			mesh			rand			All
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Avg
DOM	1.6	4.9	45	1.4	4.9	15.2	1.3	2.9	15.5	4.1
BIN	1.1	4	8	0.7	4.6	10.5	0.5	3.7	11.6	4
STR	1.5	11.1	54.3	1.5	9.1	25.1	2.4	5.5	30.2	8.4

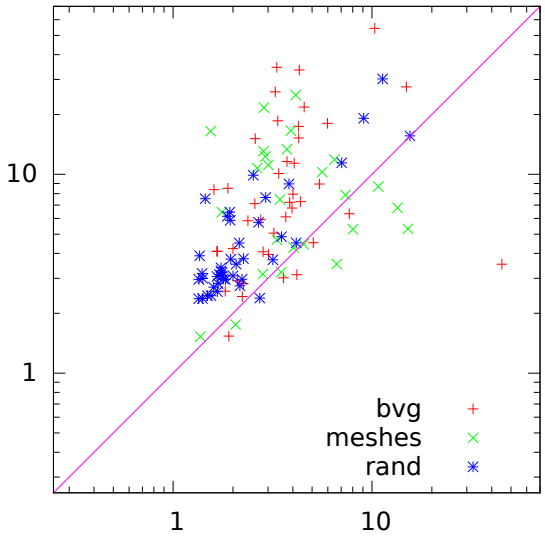


Figure 7.1 – Theoretical speedups of DOM (horizontal axis) and STR (vertical axis), each with 30 subproblems generated per available worker, with ignored decomposition times. Each mark corresponds to an instance.

Table 7.3 shows the resulting theoretical speedups through minimal, maximal and average values for the three considered methods.

On average for instances of all classes, the highest speedup bound is obtained with STR (8.4), the second highest by DOM (4.1), and the lowest by BIN (4). STR clearly outperforms the other two methods in this context.

Regardless, all these speedup bounds remain rather low, considering the fact that each instance has been decomposed into 470 subproblems in average, with a minimum of 192 subproblems and a maximum topping at 1 053.

Once more, we observe differences in the considered data depending on the instance classes. Both DOM and STR obtain speedup bounds higher than their respective means on `bvg` and `mesh` instances, while offering lower bounds on the `rand` class.

The plots in Figures 7.1, 7.2 and 7.3 show these theoretical speedup bounds in a more detailed manner, from a per-instance point of view.

For a majority of instances, the speedup bound is more appealing with STR

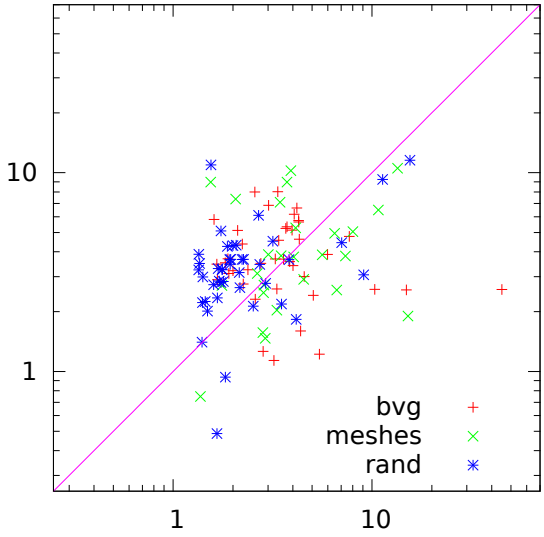


Figure 7.2 – Theoretical speedups of DOM (horizontal axis) and BIN (vertical axis), each with 30 subproblems generated per available worker, with ignored decomposition times. Each mark corresponds to an instance.

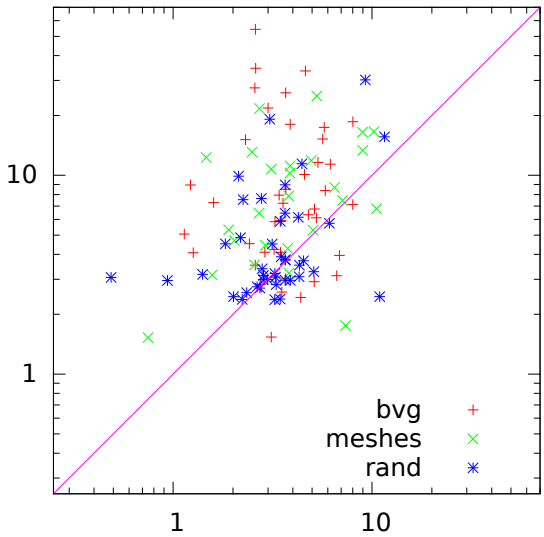


Figure 7.3 – Theoretical speedups of BIN (horizontal axis) and STR (vertical axis), each with 30 subproblems generated per available worker, with ignored decomposition times. Each mark corresponds to an instance.

Table 7.4 – Speedup values observed when taking into account the time needed to decompose the instance.

	bvg			mesh			rand			All
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Avg
DOM	1.6	4.8	44.6	1.4	4.9	15.2	1.3	2.9	15.4	4.1
BIN	1.1	4	8	0.7	4.6	10.5	0.5	3.7	11.5	4
STR	1.5	8.6	28.9	1.4	8.2	24.1	1.2	4	13.1	6.7

than with DOM. However, on some instances (mainly standing in the bvg and mesh classes), this trend is reversed. This remark holds for the two other plots of this section (DOM/BIN and BIN/STR) too. BIN, though not very impressive, still has better theoretical bounds than DOM on 67 instances out of the 109 that were considered.

With decomposition times

Though the time t_{dec} spent completing the decomposition step is rather small for DOM and BIN, it cannot be overlooked when it comes to our STR method. Consequently, a tighter upper bound on the speedup is worth considering if we are to provide the reader with a fair experimental evaluation.

We compute these new speedup bounds by dividing t_0 by the sum of the decomposition time t_{dec} and the largest subproblem solution time $t_{\max}$. This sum reflects the shortest actual solution time that can reasonably be expected even when using numerous processing units.

Speedups thus obtained are listed in Table 7.4.

Obviously, taking into account the decomposition times disadvantages STR, whose results are brought down in a significant way, while DOM and BIN's results remain practically unchanged since their decomposition process has a negligible cost. Still, STR's results remain attractive despite this change: it still obtains the largest average speedup bounds, though DOM gets close results.

Additionally, Figures 7.4, 7.5 and 7.6 display these results from a per-instance point of view.

We observe that DOM strongly dominates STR on rand instances, obtaining better speedups on all but two of the 42 instances of this class. The results are tighter on other classes: 21 against 20 on bvg instances in favour of DOM, and 16 against 10 on meshes. We notice that STR fares generally better when speedups are greater. In other words, it performs well on instances that offer a better potential for parallelisation.

In practice, BIN appears to be rather inefficient, and is clearly outperformed by DOM and STR alike. This is rather disappointing considering the promising theoretical speedup bounds it offered and its short decomposition time.

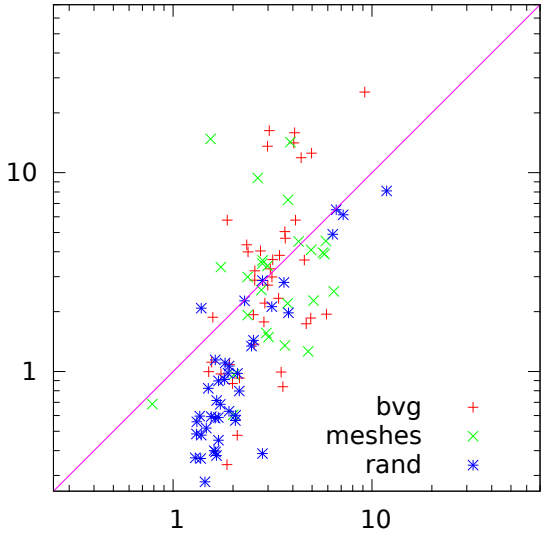


Figure 7.4 – Speedups of DOM (horizontal axis) and STR (vertical axis), each with 30 subproblems generated per available worker. Each mark corresponds to an instance.

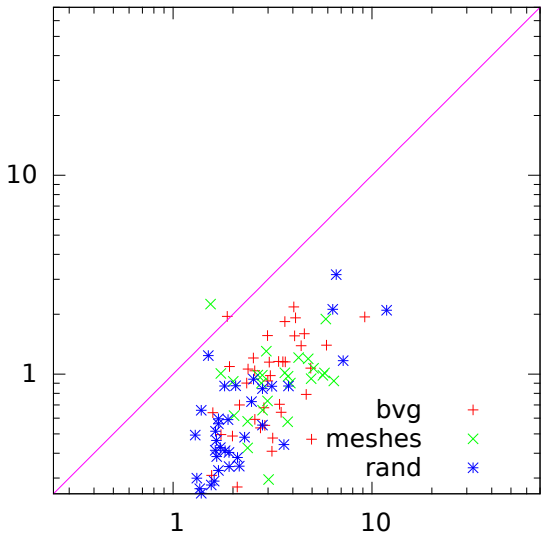


Figure 7.5 – Speedups of DOM (horizontal axis) and BIN (vertical axis), each with 30 subproblems generated per available worker. Each mark corresponds to an instance.

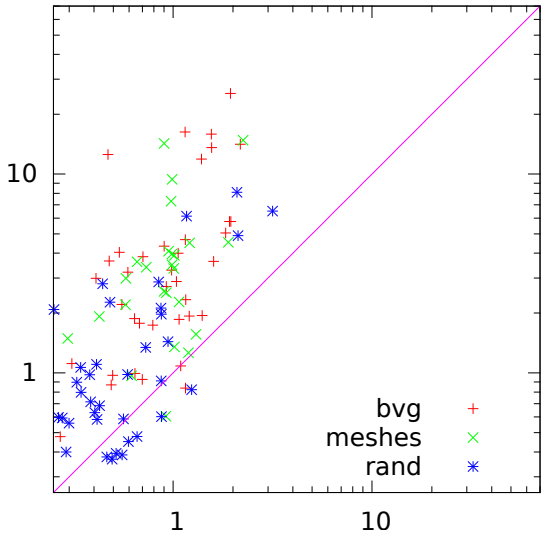


Figure 7.6 – Speedups of BIN (horizontal axis) and STR (vertical axis), each with 30 subproblems generated per available worker. Each mark corresponds to an instance.

Time needed to solve subproblems

As explained previously, a major bottleneck in the currently studied context lies in the time needed to solve the hardest subproblem. However, the hardness of the rest of the subproblems cannot be overlooked, as they evidently play a significant part in the speedup eventually obtained by the different methods.

To obtain some insight over this hardness, we considered:

- The solution time ratio between the **easiest subproblem** and the initial instance: $t_{\min}/t_0$;
- The **average** solution time ratio between a subproblem and the initial instance: $t_{\text{all}}/(k' \times t_0)$;
- The solution time ratio between the **hardest subproblem** and the initial instance: $t_{\max}/t_0$.

Note that these are computed on average for all instances of each class. Furthermore, the "easiest" and "hardest" subproblems are designated once the solution process is complete, according to the time it took to solve each subproblem.

The corresponding results are displayed in Table 7.5.

It can be observed that, on average, STR generates harder subproblems than DOM:

- On `bvg`, they appear to take four times as much time to be solved;
- On `mesh`, this factor is lower but still amounts to two;
- Lastly, on `rand`, subproblems are ten times harder.

On the other hand, the hardest subproblem for each instance is easier to

Table 7.5 – Subproblem solving time: minimal ($t_{\min}/t_0$), average ($t_{\text{all}}/(k' \times t_0)$) and maximal ($t_{\max}/t_0$) values.

	bvg			mesh			rand		
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
DOM	$0.05\,e^{-3}$	$4\,e^{-3}$	$320\,e^{-3}$	$0.1\,e^{-3}$	$7\,e^{-3}$	$293\,e^{-3}$	$0.1\,e^{-3}$	$4\,e^{-3}$	$491\,e^{-3}$
BIN	$6\ e^{-3}$	$48\,e^{-3}$	$273\,e^{-3}$	$2\ e^{-3}$	$44\,e^{-3}$	$317\,e^{-3}$	$14\ e^{-3}$	$78\,e^{-3}$	$306\,e^{-3}$
STR	$0.1\ e^{-3}$	$15\,e^{-3}$	$175\,e^{-3}$	$0.1\,e^{-3}$	$12\,e^{-3}$	$179\,e^{-3}$	$0.3\,e^{-3}$	$38\,e^{-3}$	$265\,e^{-3}$

solve when decomposing with STR than when decomposing with DOM. This significantly helps broadening the aforementioned speedup bottleneck, while also making workload balancing slightly easier.

These observations match our expectancies regarding the STR approach: the difficulty is more balanced between the generated subproblems, whereas DOM creates numerous trivial subproblem and a few very hard ones.

Speedup with 30 subproblems per worker

To observe the behaviour of the three considered methods when confronted with a number of subprobems of about 30 times the number of workers, we first ran STR on every instance i, recording the resulting numbers of subproblems s_i, and then ran DOM and BIN with $s_i/30$ workers, aiming for a number of s_i subproblems.

Table 7.6 displays the speedup observed with 30 subproblems per worker, for each decomposition method, and each class of instances (minimum, average and maximum speedups).

This particular number of subproblems per worker was used in these experiments because it was described as generally efficient in the literature when DOM was first described and evaluated [RRM14]. Note, however, that each worker might not actually get to solve exactly 30 subproblems; we simply generate a number of subproblems corresponding to 30 times the number of available workers. What subsequently happens along the solution process depends on actual resolution times.

Let t_{real} be the time elapsed until the last worker is done with its last subproblem. The resulting speedup is then given by the ratio $t_0/(t_{\text{dec}} + t_{\text{real}})$.

We note that, overall, BIN offers very low speedups. On these grounds, STR outperforms DOM on both bvg and mesh instance classes, whereas it results in lower speedups on rand instances. These rankings essentially correspond to the observations we made earlier on theoretical speedup bounds for DOM and STR. Additionally, actual speedups appear to be closer to their theoretical bounds for STR than for the two other considered approaches.

Let us finally note that these speedups are quite low, even for the best decomposition method STR. This can be emphasized by comparing those results with those found in the literature for other problems. In particular, RÉGIN et

Table 7.6 – Speedups obtained when generating a number of subproblems corresponding to 30 times the number of available workers.

	bvg			mesh			rand			All
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Avg
DOM	1.5	3.2	9.2	0.8	3.5	6.4	1.3	2.5	11.9	3
BIN	0.2	1	2.2	0.1	0.9	2.2	0.1	0.6	3.2	0.8
STR	0.3	4.8	25.4	0.6	4	14.8	0.3	1.5	8.1	3.3

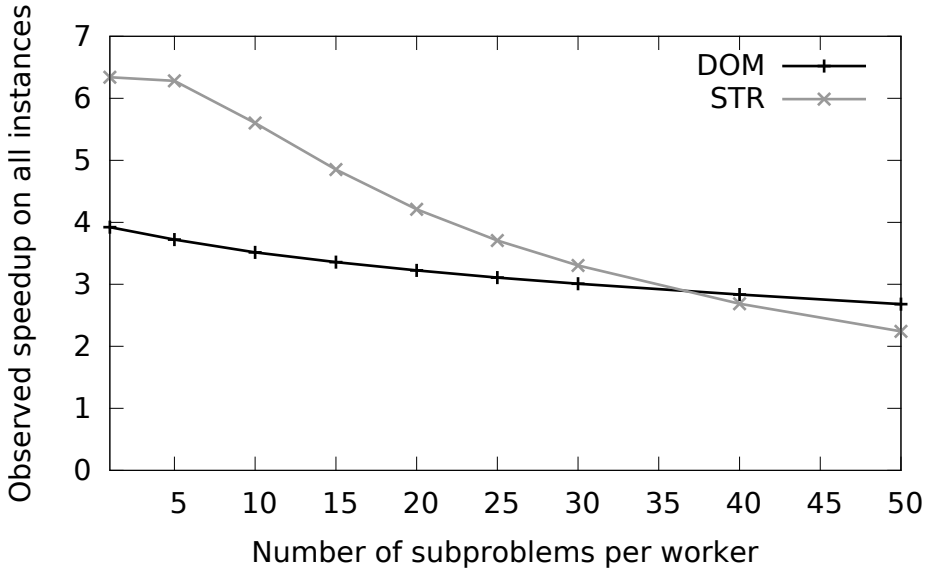


Figure 7.7 – Evolution of the speedups obtained – in average on the whole benchmark – by DOM and STR according to the ratio between the number of generated subproblems and the number of workers.

al. reported positive results on 20 CSPs, with many different instances for each of those CSPs. With 40 workers and 30 subproblems per worker (for a total of 1 200 subproblems per instance), the average speedup value was 21.3, with the lowest value, 8.6, still being rather satisfying [RRM13].

Using different numbers of workers

We investigated the effects of a change in the number of available workers on the speedups of both DOM and STR. The corresponding results are outlined in Figure 7.7.

This figure shows quite clearly that STR offers greater speedups when the number of workers is sufficiently high for them to have less than 35 subproblems each to solve in average. It can thus be said that STR makes better use of spare workers than DOM, which, on the other hand, performs better than STR

when there are more subproblems per worker.

Chapter 8

Discussion

In the maximum common subgraph problem, a subgraph of the largest possible size must be found in two compared graphs. Common exact solution methods include:

- Branch and bound, building subgraphs incrementally;
- Constraint programming, possibly with generic domain decomposition;
- Reformulation as a maximum clique problem by exploring the micro-structure of the corresponding CSP.

In this part, we investigated decomposition methods aiming at facilitating the resolution of the maximum common subgraph problem. To this aim, two new approaches, BIN and STR, were described and evaluated against an already existing method designed for generic CSP solving, DOM.

The conclusions of our observations, thoughts and experiments can be summarized as follows.

Reduction of the search space

Obviously, a more costly decomposition method can offer better theoretical advantages by reducing the size of the search space that the subsequently used solver will have to go through. STR, by means of successive decompositions that take into account the structure of the problem, followed by a step of fusions aiming at reducing redundancies, offers a significant reduction of this space. However, such a decomposition needs far more time to complete than a simple domain splitting that takes less information into account and proceeds in a more systematic fashion.

Issues encountered with BIN

Our earliest experiments quickly showed that BIN was, in practice, tremendously hindered by numerous hard subproblems. In addition to making it harder to balance the workload, they obviously made the overall solution process longer.

However, it is worth noting that BIN's hardest subproblem for each instance tends to be lower that DOM's on two of the three considered instance classes. Therefore, the subproblems generated by BIN have a well-balanced difficulty, but they are challenging overall.

After conducting more in-depth evaluations, we had to conclude that BIN was not competitive for this problem, especially when compared with STR and DOM.

Efficiency according to the number of subproblems per worker

When trying to make the number of subproblems per worker vary, it appeared that every method did not fare as well in each situation. On our benchmark, when generating less than 35 subproblems per available worker, STR showed a higher efficiency in terms of effective speedup. On the other hand, when using a larger number of subproblems per worker, DOM will generally appear to be better suited.

Overall impressions on efficiency

What eventually transpired through all these experiments was that, overall, all those methods were still insufficiently adapted and efficient for the MCIS problem. Speedups remained rather low and results were quite disappointing. This serves to prove once more how exceptionally hard the MCIS problem is, as the instances we considered were still generally small (no more than a hundred vertices, which is actually a limit in the benchmark they were taken from).

Part III

The sum colouring problem

This part is concerned with the sum colouring problem, a variant of the classical graph colouring problem. We will begin by defining this problem in Chapter 9, while also presenting the existing approaches.

The sum colouring problem is outstandingly difficult to solve to optimality. A few complete approaches have been proposed, but overall results remain disappointing. This part looks into ways to cope with this difficulty, to a certain extent.

Firstly, we investigate ways to improve existing methods: most of them had only been proposed recently, and the models, notably, were still quite simple. Moreover, partly due to the nonexistence of dedicated benchmarks and relative lack of research directed towards this problem, some methods were not intensively evaluated or compared. The improvements and alterations we proposed are described within Chapter 10.

Secondly, we capitalize on the knowledge offered by these existing methods and our improvements in order to devise a new approach, whose workings are given in Chapter 11. This approach employs a tree decomposition in order to derive profit from any structural information the instances may contain. It also has the peculiarity of combining two different models: one using constraint programming and a second one based on integer linear programming. These different models obviously have to be handled using different solvers, but combining them holds certain advantages.

As in many optimization problems, bounds are particularly useful to a solver. The sum colouring problem being a minimisation problem, upper bounds are especially interesting to reduce the scope of the search, since they allow the solver to discard any part of the search space that cannot possibly hold a solution beating this bound. A way to obtain such bounds in the context of our main new approach is investigated in Chapter 12.

After introducing all these elements, we research ways to combine a set of methods. This is done by means of a portfolio approach, which automatically chooses one of the available resolution method for each instance, according to a set of features that get extracted from it. This idea is detailed in Chapter 13.

Finally, Chapter 14 will provide a discussion on the contents of this part in order to suitably bring it to a close.

Chapter 9

The sum colouring problem

Contents

This chapter discusses the sum colouring problem in details, with the first necessary definitions presented in Section 9.1, and ways to compute various bounds compiled in Section 9.2. Existing approaches are then listed in Section 9.3, starting with the incomplete ones. We then describe the constraint programming and integer linear programming models, with a few words on boolean satisfiability models.

9.1 Definitions

9.1.1 Graph colouring

Before defining the sum colouring problem, we must make sure to be familiar with the notion of colouring and the classical graph colouring problem, from which sum colouring originated.

Definition 9.1. Given a positive integer k and a graph $G = (V, E)$, a *colouring* (or, more precisely, k-colouring) of G is a function $c : V \to \{1, 2, \ldots, k\}$. k is the number of colours of c.

Definition 9.2. A colouring c of a graph $G = (V, E)$ is said to be *valid* (or *proper*) if and only if for every edge $\{x, y\} \in E$, $c(x) \neq c(y)$.

Intuitively, it means that neighbour vertices do not share the same colour.

Definition 9.3. For a given graph G, the smallest number k such that a proper k-colouring of G exists is called the *chromatic number* $\chi(G)$ of G.

The classical graph colouring problem consists in finding a colouring of a graph G with $\chi(G)$ colours. Such problems find various concrete applications, notably in timetabling processes, where colours represent time slots, vertices stand for activities that must be matched with such slots, and edges denote conflicts between activities (human resources, objects that cannot be in two places at the same time...). In such a case, finding a valid colouring with few colours will yield a usable timetable spanning fewer time slots.

9.1.2 Sum colouring

In some contexts, there might be additional notions of costs or preferences between some tasks or time slots. The sum colouring problem aims at providing a more complex framework for such occurrences [KUB04].

Definition 9.4. The *sum colouring* for a given colouring c of a graph $G = (V,E)$ is the number $\sum(c) = \sum_{x \in V} c(x)$.

Definition 9.5. For a graph G, the number

$$\min\{\sum(c') \mid c' \text{ is a valid colouring of } G\}$$

is the *chromatic sum* of G, denoted by $\sum(G)$.

The minimum sum colouring (MSC) problem consists in finding, for a given graph G, a valid colouring c minimizing $\sum(c)$ (finding the value of $\sum(G)$ in the process, since $\sum(c) = \sum(G)$). It is an $\mathcal{NP}$-hard problem [KS89].

This problem holds a close relationship with the classical *colouring problem*, which consists in finding a valid colouring with the minimal number of colours and is also $\mathcal{NP}$-hard [GJ79]. However, it is important to note that for a given graph G, the optimal solutions generally differ. An example of such a divergence is shown in Figure 9.1.

One of the numerous reasons why the MSC problem is harder to solve than the colouring problem is that simply finding a solution that uses k colours does not allow the solver to start ignoring solutions that use k or more colours. The search space thus generally shrinks significantly slower.

Major colourings

As in many graph problems, symmetries can be found in the MSC problem's search space. The notion of major colourings can help breaking some of them.

A colouring c of G can be seen as an ordered partition of the vertices of G into stable sets. Indeed, for each colour k, the set of all vertices x_i such that $c(x_i) = k$ constitutes a stable set.

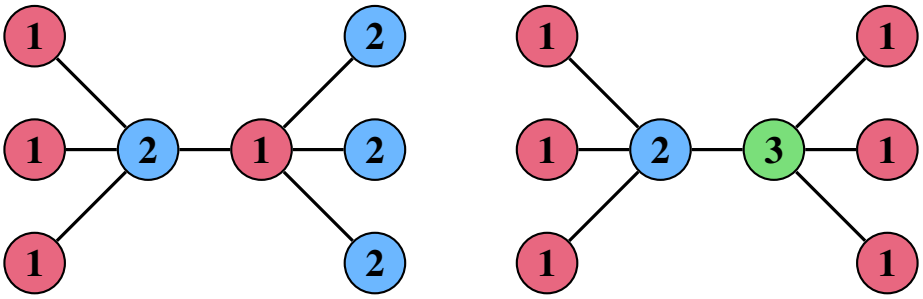


Figure 9.1 – A graph with, on the left, an optimal solution for the classical colouring problem ($\chi(G) = 2$), and, on the right, an optimal solution for the MSC problem ($\sum(G) = 11$). Introducing a third colour allows us to obtain a lower sum of weights (11 instead of 12).

Intuitively, the order of these sets (which are more specifically referred to as *colour classes*) can be changed without impairing the validity of the considered colouring. We can define a new valid k-colouring c' from an initial k-colouring c by means of a permutation p defined on the set $\{1,\ldots,k\}$ simply by stating $c'(x) \mapsto p(c(x))$. Applying such a substitution to a colouring is equivalent to swapping the colours around without altering the contents of the stable sets themselves.

The set of colourings that can be obtained by such exchanges from a given initial colouring are symmetrical and form an equivalence class. Within such a class, all colourings share the same number of colours. However, the sum of weights they induce can vary greatly. To extract the colourings that yield the best sums from these equivalence classes, the notion of *major colourings* has been introduced [LLL16].

Definition 9.6. A *major* (or *dominant*) k-colouring c is a colouring for which $|c_1| \geq |c_2| \geq \ldots \geq |c_k|$, where c_i is the set of vertices that use the colour i in c.

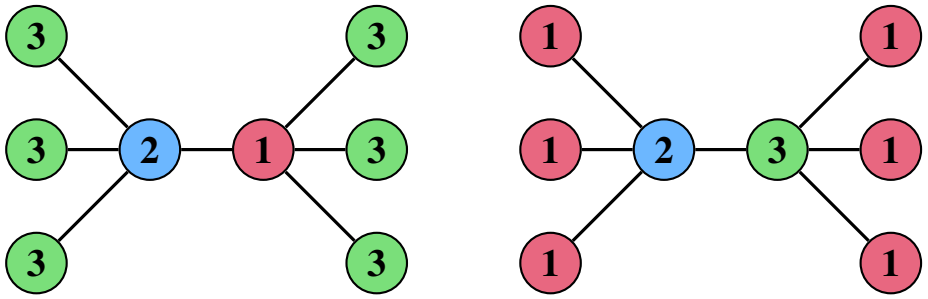
In other words, in such a colouring, the cheapest colours are used on the largest stable sets, while the ones with larger weights are preserved for sets that do not hold many vertices.

Figure 9.2 shows a dominated (*i.e.*, non-major) colouring and a corresponding major colouring.

A direct consequence of the definition of a major colouring is that such a colouring c has a lower sum colouring than every colouring that belongs to the same equivalence class as c. It naturally follows that every optimal solution of an instance of the MSC problem is a major colouring, and that only these particular colourings are worth considering when solving this problem.

From here onwards, the technique consisting in reordering the colour classes by size in order to tentatively improve each solution found during a search will be referred to as *colour swapping*.

It is important to note that solvers can still be allowed to look for a colouring that is not a major one in order to find upper bounds more easily. Such a



$\sum(c) = 1+2+6\times 3 = 21$ $\sum(c') = 6\times 1+2+3 = 11$

Figure 9.2 – A 3-colouring c (on the left), and a dominant 3-colouring c' (on the right) that uses the same sets of vertices as its colour classes, but reordered so as to obtain a lower sum of weights.

colouring might then be used to build a major one and further improve the bound, since the appropriate colour permutation is trivial to compute and is only based on the sizes of the colour classes.

9.2 Existing bounds

9.2.1 Bounds for the chromatic sum

Several theoretical bounds have been found for the MSC problem.

A lower and an upper bound for the chromatic sum can be derived from the number $|E|$ of edges in the considered graph $G = (V,E)$.

Theorem 9.1. $\left\lceil \sqrt{8|E|} \right\rceil \leq \sum(G) \leq \left\lfloor \frac{3(|E|+1)}{2} \right\rfloor$ [THO+89]

In addition, another upper bound is the sum of the numbers of vertices and edges.

Theorem 9.2. $\sum(G) \leq |V| + |E|$ [KUB04]

From here onwards, the maximal degree found among the vertices of a graph G will be denoted by $\Delta(G)$.

Theorem 9.3. *An optimal sum colouring of a graph G will never employ more than $\Delta(G)+1$ colours.* [KUB04]

Using cliques

Given a valid partial colouring c of the considered graph, a lower bound on the cost of full colourings that can be obtained by extending c can be computed in order to get better insights on the usefulness of the branch currently being

explored. A recently proposed method, primarily designed in a branch and bound context, computes such a bound by relying on clique partitions built from the set of vertices that have yet to be coloured [LEC+15B].

Definition 9.7. A clique partition of a graph G is a set $\mathcal{C}$ of cliques of G such that each vertex of G appears in exactly one clique of $\mathcal{C}$.

Using cliques partitions to compute bounds for the chromatic sum is a rather common thing. It has been demonstrated for example that given a clique decomposition $\mathcal{C}$ of G, we have:

$$\sum(G) \geq \sum_{C_i \in \mathcal{C}} \frac{|C_i| \cdot (|C_i| + 1)}{2}$$

as all vertices within each clique must use different colours [MOU+10].

Such a lower bound basically corresponds to a relaxation of the problem: every edge standing between two different cliques is ignored, thus allowing vertices to use cheaper colours than normally possible.

9.2.2 Bounds for the chromatic strength

The number of colours that should be considered is not an actual concern in the classical colouring problem: there are as many allowed colours as there are vertices, and the amount that will effectively be in use gets naturally minimised through the solution process itself, since it is the sole objective.

This ceases from being true for the sum colouring problem, however. Indeed, a solution a can be less profitable than a solution b even when b needs more colours than a. We thus need be more permissive, and cannot for example disregard colourings that use n colours on the grounds that one such colouring has already been found. The formal conclusion of this observation is that the initial sets of colours made available to each vertex, which were of no real importance in the classical colouring problem, must be defined explicitly and with care when modelling the sum colouring problem. Using a trivial model that acknowledges as many colours as there are vertices would lead to poor performances due to a tremendously increased search space size.

Considering a graph $G = (V, E)$, the most naive way of defining the initial sets of available colours consists in providing every vertex with the possibility to use any colour from 1 to $|V|$: in the worst case, a graph will need one new colour for each vertex in order to be coloured in a valid way. However, this case is only met if the instance is a complete graph, which is both very unlikely and very easy to check before attempting any kind of resolution. We thus need more precise ways of estimating the number of colours that will be needed to solve the instance. Alternatively, a different set of available colours could of course be defined for each vertex instead of allowing the exact same colours for every vertex.

A slightly better bound that can be applied to every vertex is $\Delta(G)+1$, where $\Delta(G)$ is the largest degree that can be found among the vertices of G [KUB04]. It can be rather easily explained using a few notions and intuitions. In an optimal sum colouring solution, a given vertex always uses the lowest available colour, otherwise it would be possible to lower the global sum and the solution would not be optimal. Furthermore, for a colour to be unavailable to a vertex v, it has to be used by a neighbour of v. Therefore, the largest colour that might be useful in the graph cannot exceed $\Delta(G)+1$: one colour per neighbour and one for the vertices having a degree of $\Delta(G)$ themselves.

A notion that can help reducing the number of allowed colours is the *chromatic strength* of the considered graph.

Definition 9.8. The minimal number of colours found among optimal solutions of the MSC problem for a graph G is called the *chromatic strength* (or simply *strength*) of G, and denoted by $s(G)$.

From this definition, it naturally follows that obtaining an upper bound of $s(G)$ allows us to globally reduce the number of colours that are worth considering while looking for an optimal sum colouring of G, and to cut the sets of allowed colours for all variables accordingly.

To obtain bounds for the strength of a graph, it has been proposed to explore an abstraction of the set of solutions of the MSC problem [LLL16; LLL17]. This abstraction is based on the concept of *motifs* ("patterns") [BV09; BV14].

Definition 9.9. A *motif* is a representation of a major colouring c (see Section 9.1.2) by means of a non-increasing sequence of positive integers $p = (|c_1|,\ldots,|c_k|)$, where c_i is the set of vertices that use the colour i when c is applied.
The i-th integer of p is denoted by $p[i]$, which is equal to $|c_i|$.
The *length* $|p|$ of p corresponds to the number k of colours used by c.

Motifs thus bind together sets of colourings that result in the same sum of weight. Moreover, they only represent major colourings and help reducing the scope of the problem, since optimal sum colourings are necessarily major colourings.

Definition 9.10. The set $\phi(n)$ contains all the motifs that can be defined on a graph with n vertices:

$$\phi(n) = \{p \mid \sum_{i=1}^{|p|} p[i] = n\}$$

Definition 9.11. The set $\phi(n,k)$ is defined for each strictly positive integer k as the subset of $\phi(n)$ containing only motifs that use k colours:

$$\phi(n,k) = \{p \in \phi(n) \mid |p| = k\}$$

It follows from these definitions that the motif of a k-colouring c contains sufficient data to compute the sum colouring of c:

$$\sum(c) = \sum(p) = \sum_{i=1}^{k}(i \times p[i])$$

Several different colourings can share the same motif. Consequently, considering motifs instead of major colourings reduces the search space.

By introducing a notion of *dominance* between motifs, the authors depicted ways to remove some motifs from the search space whenever a valid colouring is found, while obtaining bounds on the chromatic strength of the graph.

Definition 9.12. A motif p is said to *dominate* another motif q, if and only if the following holds:

$$\forall t \text{ such that } 1 \leq t \leq \min\{|p|,|q|\},\ \sum_{i=1}^{t} p[i] \geq \sum_{i=1}^{t} q[i]$$

This relation is denoted by $p \succeq q$.

This notion is interesting as a motif p dominating a motif q will necessarily correspond to a colouring offering a lower sum of weights than q.

Motifs are used to bound $\mathrm{s}(G)$ using Algorithm 9.1 [LLL16]. Note that the function BUILDMAJORMOTIF$(\lambda,\ k)$ computes a *major motif* p from $\phi(n,k)$ such that $p[1] = \lambda$. Such a motif is defined as follows:

Definition 9.13. Let n and k be positive integers representing, respectively, a number of vertices and a number of colours. Let λ be an integer such that $\lceil \frac{n}{k} \rceil \leq \lambda \leq n-k+1$. Let β be a shorthand for the value $\left\lfloor \frac{n-k}{\lambda-1} \right\rfloor$. A major motif p in $\phi(n,k)$ is formed as follows:

$$p[i] = \begin{cases} \lambda & \text{if } 1 \leq i \leq \beta \\ n - \beta \times \lambda - (k-\beta-1) & \text{if } i = \beta+1 \\ 1 & \text{if } \beta+1 < i \leq k \end{cases}$$

Strictly speaking, in a major motif, the β first colour classes hold λ vertices each, the $k-\beta-1$ last classes contain only one vertex each, and the intermediate class (at the index $\beta+1$) receives the remaining vertices.

Major motifs have several interesting properties.

Theorem 9.4. *Let p and q be two motifs of $\phi(n,k)$. Then:*

- *If p is major and $p[1] = q[1]$, then $p \succeq q$;*
- *If both p and q are major and $p[1] > q[1]$, then $p \succeq q$.*

Algorithm 9.1: Computational upper bound for $s(G)$

Input: The number of vertices n of G;
The cardinality of a maximum stable set $\alpha(G)$;
A valid colouring c.
Output: An upper bound of $s(G)$.

```
1 k ← |c|
2 λ ← α(G)
3 do
4 |   k ← k + 1
5 |   λ = min{λ, n − k + 1}
6 |   p ← BUILDMAJORMOTIF(λ, k)
7 while ∑(p) ≤ ∑(c)
8 return k − 1
```

Whenever a valid colouring c is found, Algorithm 9.1 can be used to find the smallest k such that the motif obtained from $\phi(n,k)$ with the properties described in Definition 9.13 has a sum colouring that exceeds that of c. It was proved that no colouring using k or more colours can be better than c with regards to the MSC problem [LLL16]. Therefore, every solution using these numbers of colours can be eliminated from the search space.

This approach brought a significant improvement over the existing bounds for the chromatic strength [LLL16] and was subsequently employed to compute bounds directly for the MSC problem itself [LLL17].

9.3 Existing approaches

9.3.1 Incomplete approaches

Most of the work directed towards the MSC problem consists in approximate methods. A review of most of these approaches may be found in [JHH16]. It classifies main contributions in three classes:

- Greedy algorithms [WH12; WH13];
- Local search heuristics [BH12; HC11];
- Evolutionary algorithms [JHH14; MOU+; JH16].

Most of these algorithms provide upper bounds, while some also yield lower bounds. As one would expect though, none of them are able to reach all best known bounds on the commonly considered instances. The percentage of instances on which the best known upper bound is reached ranges from 46 % ([WH12; WH13]) to 90 % ([JH16]) on tested graphs, depending on the heuristic.

Such approaches can still prove the optimality of a solution if the lowest upper bound happens to reach the highest lower bound. However, such proofs can only be achieved on 21 instances out of 94, even when combining all the bounds found by the methods mentioned in [JHH16].

9.3.2 Constraint programming

The first CP model proposed in the literature for the sum colouring problem is rather straightforward [LEC+15B]:

- The **variables** simply correspond to the vertices of the considered graph;
- The **values** that these variables may take correspond to the available colours (1, 2, ...);
- A difference **constraint** $x_a \neq x_b$ is added for each edge $\{a,b\}$ of the graph;
- The **objective function** asks for the minimization of the sum of all variables.

More formally, for a graph $G = (V,E)$:

$$
\begin{aligned}
& X = \{x_1, \ldots, x_{|V|}\} \\
& \forall i \in \{1, \ldots, |V|\},\ D(x_i) = \{1, \ldots, \Delta(G)+1\} \\
& C = \bigcup_{\{a,b\} \in E} \{x_a \neq x_b\} \\
& \text{Minimize} \sum_{x_i \in X} x_i
\end{aligned}
$$

In the literature, this model was shown to be rather inefficient, especially when it comes to making proofs of optimality [LEC+15B].

9.3.3 Integer linear programming

The MSC problem can be modelled through ILP as follows [WAN+12]:

- For each vertex of the considered graph, a boolean **variable** is defined for each colour that this vertex is allowed to use. This number of colours can be bounded, for example, by $\Delta(G)+1$.
  Intuitively, if a variable x_{uk} is set to *true*, it serves to represent the fact that the vertex u is using the colour k.
- Two different kinds of **constraints** are used:
  1. For a solution to be valid, each vertex must only use a single colour;
  2. For each edge e of the graph and each colour c, at most one extremity of e can use c at any time. This effectively prevents neighbour vertices from using the same colour.

- The **objective** is to minimize a weighted sum, in which each variable occurs once, with a weight corresponding to the colour it represents.

The formal representation of this model is as follows:

Minimize:

$$f(x) = \sum_{u=1}^{|V|} \sum_{k=1}^{K} k \cdot x_{uk}$$

Under the constraints:

$$\sum_{k=1}^{K} x_{uk} = 1, \quad u \in \{1, \ldots, |V|\} \tag{1}$$

$$x_{uk} + x_{vk} \leq 1, \quad \forall (u,v) \in E, \quad \forall k \in \{1, \ldots, K\} \tag{2}$$

Where $K = \Delta(G) + 1$

The main problem that arose with this model is the extensive memory needs inherent to most ILP approaches. These needs quickly become prohibitive when the instances grow larger [WAN+12].

9.3.4 Boolean satisfiability

The MSC problem has been modelled as several boolean satisfiability (SAT) problems [LEC+15B]. This consists in encoding the instance into an equivalent propositional formula. Both weighted partial MinSAT and MaxSAT encodings were proposed.

A set of hard clauses is defined, as well as soft clauses. Hard clauses, as their name implies, have to be satisfied by every solutions. On the other hand, soft clauses, much like the soft constraints of WCSPs, can be left unsatisfied, resulting in a cost given by a specific weight associated to each clause. The goal is either to minimize ("MinSAT") or to maximize ("MaxSAT") the sum of these costs.

In a way similar to what is done in the ILP model presented in Section 9.3.3, a boolean variable is instantiated for each pair formed by a vertex of the graph and a colour that it might need to use.

The hard clauses stem directly from the classical graph colouring problem, and ensure that:

- Each vertex should be assigned at least one colour;
- Each vertex should be assigned at most one colour;
- Adjacent vertices use different colours.

Each soft clause of these models is comprised of a single literal (a variable or its negation), with a weight determined by the colour corresponding to the

involved variable.

The best results were obtained with a dual encoding of the MaxSAT model, using the ISAC solver [KAD+10].

The results were rather encouraging. However, due to the overall similarity between these SAT encodings and the ILP model and to a lack of time, this thesis focuses on CP and ILP approaches. Moreover, the results shown in the literature demonstrated that SAT had for example issues solving instances like queen5_5, which is almost instantly solved to optimality using CPLEX with an ILP model.

Chapter 10

Improving the existing models

Contents

This chapter presents several improvement for the existing approaches previously described.

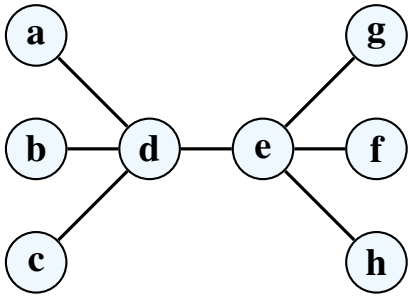
The first two improvements, detailed in Sections 10.1 and 10.2, can be applied to both constraint programming and integer linear programming. On the other hand, the latter improvements, described in Sections 10.3 to 10.6, were specifically used in the constraint programming context.

Lastly, Section 10.7 will present the results obtained with those improvements.

10.1 Reduction of initial domains

In Section 9.2.2, we stressed the importance of considering less colours, and recalled a few ways to do so in a global fashion.

We saw that a trivial global bound can be defined by $\Delta(G)+1$ for all vertices. This reasoning can be pushed a little further when there is no particular need to provide the same initial set of allowed colours to every variable: instead of using the maximal degree found in the graph, the degree of each vertex can be considered individually. Indeed, the rules mentioned in the $\Delta(G)$ approach still hold when considering each vertex separately: if the maximal degree is 5, there is still no need for a vertex of degree 3 to have more than $3+1=4$ available colours, since its neighbours will never be able to use every colour in the



($\Delta(G)+1=5$)

Without reduction:
CP: 8 variables with 5 colours
$\Rightarrow 5^8 = 390\,625$
ILP: 8×5 binary variables
$\Rightarrow 2^{40} \simeq 1.1 \times 10^{12}$

With reduction ($\deg(v)+1$)**:**
CP: 6 variables with 2 colours and 2 with 5
$\Rightarrow 2^6 \times 5^2 = 1\,600$
ILP: 6×2 + 2×5 binary variables
$\Rightarrow 2^{6\times2+2\times5} \simeq 4\,194\,304$

Figure 10.1 – The impact of a reduction of the number of considered colours when considering each vertex independently. The size of the search space for both CP and ILP models is given. Note that in ILP's case, there would also be less constraints. For example, in the reduced version, there is no need to prevent a and d from using the colour 3 simultaneously, since a cannot even use this colour anymore.

set $\{1,2,3,4\}$, and the colour 5 is of no use whatsoever as long as there is a lower colour available. This leads us to the following simple theorem:

Theorem 10.1. *For any vertex v of a graph (V,E), the initial set of available colours can be reduced to $\{1,\ldots,\deg(v)+1\}$ without changing the resulting set of optimal sum colourings.*

Proof. To prove this property, let us suppose that it does not hold for a given optimal colouring c of a graph (V,E). It follows that there exists a vertex v in V such that $c(v) > \deg(v)+1$. In such a case, there has to exist a colour $x \in \{1,\ldots,\deg(v)+1\}$ such that every neighbour of v has a colour different from x (since v only has $\deg(v)$ neighbours). As a consequence, a colouring yielding a lower sum than c can be obtained by colouring v with x instead of $c(v)$. Therefore, c is not optimal, which contradicts our initial claim. □

When using the ILP model outlined in the first part of this thesis, reducing the number of considered colours for a vertex has a slightly different effect: since additional variables and constraints have to be introduced for each considered colour, these entities become unnecessary and can be removed from the model altogether.

Figure 10.1 shows the effects of reducing the number of considered colours for both CP and ILP models.

10.2 Adding *allDifferent* constraints

The *allDifferent* global constraint forces the variables $\{x_1,\ldots,x_k\}$ in its scope to use different values. It is thus semantically equivalent to a set of dis-

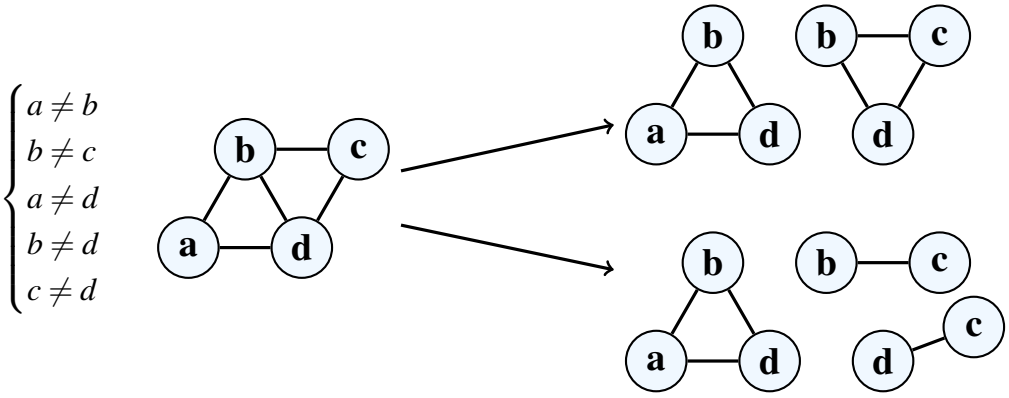


Figure 10.2 – Different ways to introduce *allDifferent* constraints in the sum colouring CP model for a given graph (on the left). The first version (in the uppermost right corner) only uses *allDifferent* constraints and enforces the disequality of b and d twice, while the second (just below) still comprises two binary disequality constraints ("$b \neq c$" and "$d \neq c$") and has no redundancies.

equality constraints linking all these variables pairwise: $\{x_1 \neq x_2,\ x_1 \neq x_3,\ \ldots,\ x_{k-1} \neq x_k\}$. Modelling a problem using *allDifferent* constraints instead of such sets of binary constraints, however, is generally beneficial. As explained in Section 2.1.2, global constraints allow solvers to get a better grasp of the problem and use more specific tools. A CP solver, for example, will employ a dedicated propagator in order to eliminate more values from the domains of the involved variables. The time complexity can be improved as well: *allDifferent*'s propagators may for example enforce global consistency in polynomial time.

For these reasons, we decided to introduce *allDifferent* constraints in sum colouring problem models. Of course, this cannot be done in a systematic way as in the n-queens problem: the number of disequality constraints and their scopes are determined by each instance individually, since they stem directly from the graph's structure.

In the sum colouring context, a set of disequality constraints can be safely replaced with an *allDifferent* constraint only if they form a clique. Note that this clique can be found either in the constraint graph or in the graph that must be coloured, since they are equivalent for this particular problem.

Since the set of solutions of a CP model remains unchanged whether we state once or n times that two variables must take different values, the scopes of *allDifferent* constraints can safely intersect, leading to redundancies that can be useful to a certain extent, especially when propagators specific to *allDifferent* constraints are used. It follows that there are numerous ways to apply *allDifferent* constraints to a same given graph. The only imperative is to cover the entirety of the original disequality constraints. This is shown by Figure 10.2.

To fully cover the graph with constraints (with a combination of *allDifferent*

constraints and, for remaining edges, binary disequality constraints), one must compute a set of cliques $\{C_1, \ldots, C_k\}$ such that every edge of the graph is found in at least one of these cliques. Note that binary disequality constraints correspond to cliques of two vertices in this set.

Such a clique set can be computed in many different ways, and many of them are not that costly. We investigated several ways to build them, with different levels of redundancies.

Low redundancy version

A way to avoid introducing too many redundancies into the model while replacing sets of disequality constraints by *allDifferent* constraints is to proceed as follows:

1. Initialize a graph G as a copy of the graph that must be coloured for the considered sum colouring instance. Create an empty set of cliques S.
2. Find the vertex with the highest degree in G. This vertex will be the first of a new clique C.
3. Add to C the vertex that has the highest degree in the subgraph of G induced by vertices that are linked to every single vertex of C. Repeat until the clique becomes maximal.
4. Remove from G every edge that links vertices of C, and store C in S.
5. Repeat steps 2–4 until every edge of G has been removed.

Note that all along this procedure, ties are randomly broken.

At the end of this algorithm, each clique of S can be used as the scope of an *allDifferent* constraint. However, cliques of only two vertices should obviously be used to create basic disequality constraints rather than *allDifferent* constraints to avoid unnecessary computations during constraint propagation.

Note that using this method, cliques are not necessarily maximal with respect to the whole initial graph, since edge removals may prevent some of them to grow to their full initial potential.

Random version

We developed an alternate version, which comprises stochastic aspects and induces an average level of redundancies between the cliques.

1. We start by building a clique of one vertex for each vertex of the graph.
2. Then, each of these cliques is made maximal by iteratively adding to it as many vertices as possible.

The main peculiarity of this approach is that at each step during the maximisation phase of each clique, the additional vertex is selected completely randomly.

Once all the cliques have been maximized in this fashion, some may have become equal. The simplest possible example is a graph of two vertices a and b, with an edge between them. A first clique would be created for a, and b would be added to it, while another clique would start from b and would be extended with a. The two resulting cliques would then both be comprised of the vertices $\{a,b\}$. Since stating several times the exact same *allDifferent* constraint cannot possibly speed-up the solution process, duplicates are discarded. Note that since these cliques get maximized, there is no inclusion possible between cliques; only equalities may happen.

Finally, we must make sure every initial disequality constraint is represented in at least one of the computed cliques. Missing constraints are added (in the form of binary disequality constraints) to make sure that the resulting problem accepts the same solutions as the initial one.

Greedy version

Finally, we employed a greedy version, with a higher level of redundancies. It proceeds just as the random version does, but instead of maximising the cliques in a random fashion, we choose, at each step, the vertex with the highest degree in the subgraph induced by the set of vertices that are linked to every vertex already in the clique.

This is actually the maximisation heuristic used in the low redundancy version. The difference here is that edges belonging to the built cliques are not removed from the graph, thus allowing redundancies to appear between the cliques.

To spread redundancies in a more balanced way, we break ties by selecting vertices that appear in the fewest cliques.

Observations

Early experiments showed us quite clearly that there was no significant difference between the improvements brought in results by the *allDifferent* constraints built using these three methods. We eventually settled for the greedy version, which was marginally better and more consistent in the provided results.

Adaptation to the ILP model

allDifferent constraints do not actually exist as such in integer linear programming. Furthermore, the ILP model we use for the sum colouring problem contains several variables for each vertex of the graph (one per colour). Still, *allDifferent* constraints can be simulated rather easily in this context.

For each computed clique C and for every colour *col* allowed for at least two vertices of the considered graph:

1. We generate the set *Vars* of binary variables of the original ILP model that correspond to a vertex of C and to the colour *col*.
2. Then, if the *Vars* set holds at least two variables, we add to the model the constraint $\sum_{x \in Vars} x \leq 1$. In other words, we make sure no more than a single vertex in C uses the colour *col* at any given time.
3. Once this constraint has been introduced, we can remove those (of the initial model) that state $x + y \leq 1$ for any pair of variables x and y of *Vars*.

10.3 Deriving a cheap lower bound from a clique partition

The contents of this section are concerned with improving the CP approaches. The described tools were not applied to our ILP approaches.

As explained in Section 9.2.1, lower bounds can be derived at any node of the search tree from a clique partition built from the set of uncoloured vertices. However, we noticed that the cost of building such partitions is far from negligible and that the abusive use of such a method generally has a negative impact on the resolution time. Since the idea of such bounds is still an interesting point, we aimed at finding a better balance between the time allotted to the computation of these lower bounds and their quality.

Instead of computing a new clique partition at each node of the search tree, we suggest to cut down the associated cost by computing such a partition only once, with an acceptable loss in the precision of bounds. This is done before the actual beginning of the search. In such a case, the partition is obviously computed on the whole graph, since "the set of uncoloured vertices" is actually the full set of vertices.

When deriving lower bounds from cliques, the most straightforward method consists in using the trivial fact that a clique of n vertices will need at least n colours to be coloured in a valid way, since all its vertices are linked pairwise. This method can be improved by taking into account the current domains of the variables corresponding to the clique's vertices. These domains get progressively reduced during the search by the propagation of inter-clique difference constraints, and such reductions will necessarily make the sum of weights rise within the cliques by forcing some vertices to use more costly colours.

For a clique of n vertices, the naive lower bound of $n \cdot (n+1)/2$ can be raised to the smallest possible sum of n distinct values from the n variables' domains. However, updating this bound with an exact value after each domain reduction would be too costly. Therefore, we introduce an approximate, easier to compute version of this bound.

For each clique C, we maintain the union u of the domains of the variables corresponding to the vertices of C. Then, we use the sum of the $|C|$ smaller

$$
\begin{aligned}
D_x &= \{1, \quad\quad\quad 4 \quad\quad\}\\
D_y &= \{\quad 2, \quad\quad\quad 5\}\\
D_z &= \{\quad 2 \quad\quad\quad\quad\}\\
u &= \{1, \quad 2, \quad 4, \quad 5\}
\end{aligned}
$$

Naive bound: $1+2+3=6$
Three minimal values: 1, 2, 4
Resulting bound: $1+2+4=7$
Optimal lower bound: $1+2+5=8$

Figure 10.3 – A lower bound example for a clique of three vertices. In this case, two values used to compute the bound (1 and 4) can only be found in the same variable's domain: x. Therefore, this does not correspond to a real solution, since x obviously cannot take a second value in another variable's stead. This shows how merging domains and forgetting about the origin of each value can be used as a means to obtain a better balance between the computational cost and the quality of lower bounds.

values found in u. This indeed corresponds to a lower bound, since even in optimal conditions no lower sum could be achieved. Actual sums encountered when building valid solutions will generally be higher, because among the values employed to obtain such a lower bound, several might be only found in the domain of a same single variable (see Figure 10.3 for an example).

The complexities of the operations necessary to obtain these bounds appear to be reasonable:

- Every time a value *val* is removed from the domain of a variable in a clique C, a linear time is needed to check whether at least one variable of C still holds *val* in its domain;
- The union u of the domains of the clique's variables is updated accordingly;
- Then, if *val* is not available in this clique anymore and was one of the $|C|$ lowest values in u, the bound is updated by subtracting *val* from it and adding the new $|C|$-th lowest value in u.

The global complexity of such an update is thus linear with respect to the size of the considered clique.

Usage restrictions

Such a bound is generally not very useful when only a few vertices are coloured: the bound might not be precise enough to prune the current branch (which is the only goal of such a lower bound). Conversely, when almost all vertices have been coloured, updating a lower bound might take longer than completing the exploration of the current branch of the search tree. For these reasons, we decided to add two parameters to this method.

The *gap* parameter This first parameter aims at preventing lower bounds from being computed when not enough information on the current partial colouring is available. To be able to prune a branch, we need the lower bound to reach the current global upper bound (which means that the current partial solution cannot be extended into an interesting solution). These bounds are more likely to meet when the sum of assigned variables gets closer to the upper bound. Therefore, we only compute this lower bound if the sum of assigned variables is at a distance of at most $gap\,\%$ of the current upper bound (which generally corresponds to the best known solution). Our experiments showed that a value of $20\,\%$ is typically a good choice.

The *unc* parameter To prevent spurious computations, we stop using this lower bound when only *unc* or less vertices are still uncoloured. After a few experiments, we settled for a value of 5.

10.4 Combining *sum* and *allDifferent* constraints

This section is concerned with constraint programming, and more precisely with the interactions between constraints.

In 2012, BELDICEANU et al. published a paper on the interactions between a constraint consisting in bounding the sum of a set of variables (*sum*) and an *allDifferent* constraint enforced on the same variables [BEL+12]. Capturing interesting properties of constraint associations allows to develop specific propagators that do less computations or provide more information and prune more values from the domains of variables.

More formally, the *sum*+*allDifferent*$(C, cost)$ constraint ensures that all variables in C use different values, while also preventing their sum from exceeding the given *cost* value.

When using a CP model to solve the sum colouring problem, instead of just using a clique partition of a graph to obtain lower bounds and prune branches (as explained in Section 10.3), we can see such bounds as a *sum* constraint. Indeed, we must make sure the sum of every variables does not exceed the current global upper bound. Since, in the sum colouring context, we can define *allDifferent* constraints on the same cliques without changing the set of solutions of the problem, this conjunction of constraints deserved our interest. As a matter of fact, by using *sum*+*allDifferent* constraints whose scopes define a partition of the whole graph, the sum of those partial lower bounds will give a global lower bound.

We propose to investigate in further details this conjunction's relevance in the context of the sum colouring problem.

10.4.1 Description

Usage

For any given clique C in the graph that must be coloured, applying the *sum+allDifferent* consistency algorithm allows to tighten the domains of the variables corresponding to the vertices of C. In the process, a lower bound LB_C can be obtained for the sum of these variables.

At the beginning of the solution process, a set of cliques $\{C_1, \ldots, C_k\}$ must be computed in such a way that they define a partition of the graph: every vertex must appear in exactly one of those cliques. This ensures that the sum $LB_{C_1} + \ldots + LB_{C_k}$ yields a valid lower bound for the whole graph.

Using *allDifferent* (or *sum+allDifferent*) constraints only on cliques that define a partition of the graph's vertices is rather restrictive, but nothing prevents the model from containing regular *allDifferent* constraints alongside the *sum+allDifferent* constraints. This way, large cliques that might be somewhat neglected by the *sum+allDifferent* constraints may benefit from the regular *allDifferent* consistency algorithm.

The *sum+allDifferent* consistency algorithm relies on a few additional elements that will now be presented.

Minimal cost matchings

The lower bound for the sum of the variables in the scope of an *sum+allDifferent* constraint is obtained by means of a matching computation. This matching, or more precisely *minimal cost matching*, associates a value with each variable involved in C. The minimal cost matching is a bipartite graph, which means that it contains two distinct types of vertices:

- Some vertices (set *Variables*) represent every variable of C;
- The others (set *Values*) represent the values that appear in the union of these variables' domains.

The minimal cost matching observes the following conditions:

- By definition of a bipartite graph, edges necessarily have one extremity in each on these sets;
- Each vertex of the *Variable* set is linked to a *Value* vertex;
- For each variable v in C, the corresponding vertex from the *Variables* set must be linked to a *Values* vertex that corresponds to a value belonging to v's domain;
- Each value in the matching can only be used once, *i.e.* vertices of the *Values* set cannot be linked to more than a single vertex.

All these properties ensure that a minimal cost matching for a set C of variables describes an assignment of C in which every variable uses a different value picked from its own domain.

The paper that introduced this approach ([BEL+12]) provides an algorithm to compute a minimal cost matching for a given *sum+allDifferent* constraint while ensuring that the sum of all values used in this matching is minimized. An imperative condition for this algorithm to be applicable, however, is that the variables' domains must not contain *holes*. In other words, if two values a and b such that $a < b$ are in a variable v's domain, any value i such that $a < i < b$ must be in this domain too.

Additionally, the order in which edges are incorporated into the matching matters:

- The available values are considered one at a time, starting with the lowest ones;
- Priority is then given to variables whose maximal values (from their respective domains) are lower.

This ensures that they are assigned a value if possible before all the values of their domains are used by other variables. Figure 10.4 shows an example of a minimal cost matching computed in such a way:

1. The first considered value is 1 (the lowest in the union of domains); it is assigned to a because the maximal value in D_a (4) is smaller than those of D_b and D_c (5);
2. Then, 2 is considered, and assigned to the only unassigned variable that has 2 in its domain: c;
3. Lastly, 3 is assigned to b following the same rules.

The computation of this matching yields a lower bound through the sum of the values linked to variables by its edges. It is important to understand that these values have nothing to do with the actual CP model solution process: this matching is only a temporary hypothetical assignment used as a tool to obtain this bound (and, as we will see later, to tighten domains).

Variable blocks

Even with an efficient implementation (the original paper proposes one in $\mathcal{O}(n \log n)$ time), computing a minimal cost matching is a process which is too costly to be run at each node of the search tree, or even when the search conducted by the CP solver leads to the actual assignment of a variable belonging to the scope of the considered *sum+allDifferent* constraint. To prevent unnecessary computations, BELDICEANU et al. designed a convenient workaround, based of groups of variables called *blocks*.

For each *sum+allDifferent*$(C, cost)$ constraint, blocks of variables must be defined. A block is a subset of the variables corresponding to the vertices of C. Furthermore, the blocks of C define a partition of the variables of C: every variable involved in C appears in exactly one block. The main idea behind this approach is that variables of a same block are considered as playing the same

$$X_C = \{a, b, c\}$$

$$D_a = \{1,\ 2,\ 3,\ 4\ \}$$
$$D_b = \{\ \ \ 3,\ 4,\ 5\}$$
$$D_c = \{\ \ 2,\ 3,\ 4,\ 5\}$$

$$D_a \cup D_b \cup D_c = \{1,2,3,4,5\}$$

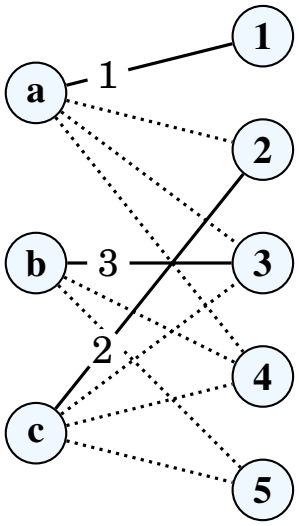


Figure 10.4 – A simple minimal cost matching example for a clique of three vertices. The edges are numbered according to the order in which they were added to the matching. a was given a value first because its domain's maximal value, 1, is the lowest. The sum of used values – and thus the lower bound – is 6. Dotted edges are those that could have been added to the matching with regards to the variables' domains but were not selected.

role in the computed minimal cost matching and are thus interchangeable.

A block is more formally defined as a set of $(variable, value)$ edges from the previously computed minimal cost matching. Within any given block, the edges must have been added consecutively to the matching. Blocks are built in such a way that certain properties hold:

- Variables corresponding to every edge past the end of the block have a domain with a minimal value which is strictly greater than the value corresponding to the last edge of the block;
- There must *not* be an index j greater than that of the beginning of the block such that for every index i between j and the end of the block, the minimal value of the variable of the i-th edge is greater or equal than the used value of the j-th edge.

The first of those properties simply ensures that blocks are maximal with regards to the number of edges they contain. The second one prevents a block from being included into another [BEL+12].

To list blocks, one should start with the first one, which has the very first edge of the matching as its starting point, and then look for an end point that satisfies the aforementioned conditions. Candidate end points are considered starting with the farthest one (the last edge of the matching), towards the beginning of the matching, until a suitable one is found. The next block can then be built in much the same way, starting with the edge of the matching

placed just after the last edge of the previously computed block. This process is repeated until a block reaching the last edge of the matching is found.

Basically, blocks form groups of variables that have similar domains and that can be indifferently swapped when computing the lower bound associated with the considered clique. This provides a way to quickly update the previously computed bound depending on the assignments that were actually performed by the solver. This particular point will now be addressed.

Updating the matching's bound

For a variable v in a block b, if v actually gets assigned a value n by the CP solver:

- If, in the computed matching, n is linked to a variable v' of b, then the previously computed lower bound is still valid: v and v' are considered swappable and this assignment has no particular effect on the perspectives offered by the current partial assignment. This, of course, also applies if $v = v'$;
- If, on the other hand, n is outside the boundaries of b according to the values that the variables of b use in the computed matching, v is expelled from the block b and constitutes a new block on its own. Furthermore, the lower bound has to be updated by substituting a new value for the one that used to be linked to v in the matching. This new value, which can be found in linear time, is the smallest unmatched value greater than or equal to n.

Therefore, overheads only occur when the lower bound actually needs to be updated, and even so, these computations are fast compared with a new matching computation.

Bound consistency

Apart from providing a lower bound for the sum of the variables in the scope of a *sum+allDifferent* constraint, the consistency algorithm of this conjunction tighten the domains of variables: it is a *bound consistency* condition, where only the extreme values of the domains are considered for deletion.

This filtering algorithm works by identifying, for each block b of variables (starting with the last), the lowest value v such that it can be proven that if a variable of the block b is assigned the value v, the minimal cost of the matching will exceed the maximal acceptable cost for this *sum+allDifferent* constraint.

10.4.2 Adaptation to sum colouring

Obtaining a global lower bound

To obtain a lower bound on the sum of every variable of the problem using *sum+allDifferent* constraints, we would need an instance of such a constraint spanning all the variables. Obviously, this is not the case: we cannot force every single variable to use a different value and must respect the initial structure of the problem.

A convenient workaround is to define a partition into cliques of the vertex of the considered graph. Then, by applying separate *sum+allDifferent* constraints to the set of variables corresponding to these cliques, we can obtain local lower bounds whose sum yields a global lower bound for the sum of every variable of the problem.

The validity of this technique is ensured by the fact that the *sum* constraint is decomposable: the sum of lower bounds computed on parts of the problem corresponds to a valid global lower bound.

On a side note: when using such a partition, the maximal acceptable cost for a *sum+allDifferent* constraint required for its bound consistency filtering algorithm can be easily obtained in our context by subtracting the sum of the lower bounds of the other *sum+allDifferent* constraints from the global upper bound.

Matchings and holes

As stated before, a minimal cost matching can only be computed safely when the domains of the considered variables do not have holes. In sum colouring instances, domains often have such holes, especially when considered partway through the search led by the CP solver. A simple workaround is to behave as if holes did not exist when computing the matchings. However, this will of course degrade the quality of the resulting lower bounds, since we allow the algorithm to use values that are not actually available.

Usage restrictions

As explained in Section 10.3, using bounds in inappropriate contexts may lead to unfruitful repetitive computations. We thus provided this method with the same two parameters used by the previously presented bound algorithm. However, experiments showed us that the optimal values were different. For the *gap* parameter, the value 50 % appeared to be a good choice for the *sum+allDifferent* bound computation algorithm, while *unc* was brought to 0.

Blocks and sum colouring

As briefly explained earlier, the blocks computed from the matchings used by this approach are closely related to the domains of variables: variables with similar domains will tend to be gathered into a block.

In the sum colouring problem's context, most variables have very similar initial domains. In particular, most domains have the same minimal value. We observed during early experiments that, for almost every instance of our benchmark, each clique of the computed partition gave birth to a single monolithic block. This appeared to lower the quality of the resulting lower bounds: too many abstractions were being made, since every variable within the scope of each *allDifferent* constraint was considered to play the exact same role.

Matching delay

Since the initial domains' nature seemed to be the cause of this lack of efficiency in our context, we considered the possibility of waiting before computing the blocs and matchings.

We therefore wait until the condition *gap* is reached to compute the matchings and corresponding blocks. Here, however, the main idea is not to prevent unnecessary computations, but rather to build the matchings – and their associated sets of blocks – later, when the variables' domains have already been subject to some filtering. We expected this to allow more blocks to coexist within each clique than when building them at the beginning of the solution process.

It is important to observe, though, that since these blocks become dependant of a certain state during the search, they must be scrapped when we backtrack above the level at which they were created. For example, if matchings and blocks are computed at a node of the search tree where a variable x is assigned the value 1 and y the value 3, these matchings and blocks become invalid as soon as a backtrack undoes one of those assignments. Later during the search, if the activation conditions are met again, new matchings and blocks will be computed. These might be discarded again, and so on. Therefore, the cost to pay is significantly higher than with the standard use of this approach.

Observations regarding this thesis Due to a lack of time, we could not fine-tune the approach consisting in computing the matching after a certain delay. It appeared during the few experiments regarding this technique that most of the time, most domains fell into one of the two following categories: some domains were almost unreduced and thus looked very much alike each other, and the others were reduced to a single value (which is equivalent to an assignment). The overall results thus obtained were indistinguishable from those stemming from a regular use of the *sum*+*allDifferent* conjunction (by com-

puting matchings once, at the very beginning of the search, and without ever scrapping them). This does not mean, however, that this delaying technique is of no value whatsoever. Further research would be needed in order to be able to pronounce a definitive conclusion on this matter.

Issues regarding filtering

When using the bound consistency filtering algorithm of the *sum+allDifferent* constraint, the trick consisting in acting as if the holes in domains did not exist cannot be used anymore. More precisely, this filtering algorithm cannot be run on a block in which at least one variable has a hole in its domain. Indeed, the value chosen as a new maximum to cut the higher part of domains is computed on the assumption that the minimal cost matchings can indeed be realized by the CP solver. Since we fill holes in domains in order to compute this matching, it often remains purely fictional, with values that could not be used in the actual search process.

This, however, does not render the filtering algorithm completely unexploitable in the sum colouring context. We simply have to check whether a block has holes in one of its variables' domain before attempting to tighten these domains. If holes are indeed detected, we can only compute the lower bound discussed earlier. This is somewhat problematic and makes this part of the approach less beneficial for the sum colouring problem. Our experiments showed us that virtually every domain that does not have holes during the solution of a sum colouring instance is actually a singleton (meaning that the variable was already assigned).

10.5 Heuristic choices

In CP approaches, choosing appropriate heuristics to order values and variables is a very important point.

As the goal of the sum colouring problem is to minimize the sum of the variables, we kept the trivial value ordering heuristic choosing the smallest value available.

We have designed and compared different variable ordering heuristics, including well-known ones such as *Activity* and *Dom / Wdeg* (see Section 2.2.3). The most interesting results generally involve what we called the *MinElim* heuristic. *MinElim* chooses the variable that has the smallest minimal value in its domain, and break ties by choosing the variable for which this value would be removed from the fewest domains.

Example 10.1. Let us consider three variables x, y and z with the following respective current domains:

$$D_x = \{1,2\} \qquad D_y = \{1,3\} \qquad D_z = \{2,3\}$$

A tie would occur between x and y, who both have 1 as their lowest available value. Let n_x be the number of neighbours of x that currently have 1 in their domains and that would therefore lose it were x to be chosen, and n_y the number of neighbours of y that would be in a similar situation in the event of an assignment of 1 to y. Eventually, x would be chosen if the n_x value were lower than n_y, and vice versa.

10.6 Hybrid strategies

Thanks to the flexibility of CP solvers, it is possible to bring changes to the search strategy partway through the search. This can be done, for example, to intensify researches after an initial exploration phase, thus making it easier to prove the optimality of a solution of high quality.

10.6.1 Restarts

During our experiments, we noticed that some strategies were more suited for exploration (improvement of the global upper bound) while others made proofs of optimality more easy to perform. However, trying to prove the optimality of the current upper bound right from the very beginning of the solution process is generally an inefficient option, as true optimal bounds tend to be discovered later. This duality within the search process is actually a common concern in constraint programming.

In an endeavour to get CP solvers to perform more proofs of optimality, we decided to combine restarts strategies. More specifically, starting with an exploratory strategy before switching to a proof-oriented strategy after a while may intuitively lead to better results with regards to proofs of optimality. The first strategy is Luby [LSZ93], while the second is a geometric one.

When appending strategies in such a manner, an additional thing to decide is when to switch from the first to the second strategy. A time limit is an obvious solution, but would not take into account the actual progress of the search, which could vary. Indeed, it is more profitable to wait until a good solution (that has a high probability of being the optimal one) is found before switching strategies.

Considering this, we opted for another approach, consisting in setting a limit *nbRest* representing a number of restarts. If a total of *nbRest* successive restarts are performed with, in the meantime, no global upper bound improvement whatsoever, the solver will assume that the probability that the current solution is the optimal one is high enough for a change of strategy to be carried out in order to favour proof-making.

This combination hence needs four parameters:

- The **base** value for the **geometric** phase. 2 is generally a good choice;

- The corresponding **factor** for this phase. 100 leads to satisfying results;
- A second **factor** reserved to the **Luby** phase. Using our usual value of 500 is advisable;
- The number of **successive unsuccessful restarts** that must be reached for the strategy to change. 50 appeared to be a good value.

10.6.2 Variable-ordering heuristics

Variable-ordering heuristics, just like restart strategies, can favour different aspects of the search, especially diversification (to find good solutions) and intensification (to tentatively prove the optimality of the current best solution). Therefore, alongside the scheduled change of restart strategy, we allow the CP solver to go from *MinElim* to *Dom*/*Ddeg* when this change occurs. While *MinElim* is better suited for diversification, *Dom*/*Ddeg* offers better capabilities when it comes to proving optimality.

10.7 Results

10.7.1 Experimental setup and benchmark

Programs were written in C/C++, compiled using GCC with `-O3` optimisation, and run on an Intel® Xeon® CPU E5-2670 at 2.6 GHz processor, with 20 480 KB of cache memory and 4 GB of RAM.

We considered 126 instances which are classically used for sum colouring [WAN+12; JHH16]. Some are from COLOR02/03/04,[1] but most of them are DIMACS instances designed for the classical colouring problem.[2] The size of these instances is detailed in Tables 10.1 and 10.2.

The time limit was set to 24 hours. Note that this still proves to be insufficient for some of the hardest instances, regardless of the employed resolution method.

Each instance has an associated *reference solution*, which is the best known upper bound, either available in the literature (notably [WAN+12; JHH16]), or computed by one of our approaches prior to experiments shown in this thesis. Actually, many instances in our benchmark did not have any bound in the literature. These reference bounds give a good overview of the state of the art for our benchmark.

Some tables include *distances* to reference bounds. The distance between an upper bound b and a reference solution r is a percentage given by the ratio $\frac{b-r}{r}$, meaning that finding the reference solution leads to a null distance while a distance of 100 % means that b is twice as large as r.

[1] http://mat.gsia.cmu.edu/COLOR02

[2] ftp://dimacs.rutgers.edu/pub/challenge/graph/benchmarks/color/

Table 10.1 – (1/2) For each instance: its numbers of vertices, edges, and the resulting density ("D.").

Inst.	$\|V\|$	$\|E\|$	D.
1-FullIns_3	30	100	23.0
1-FullIns_4	93	593	13.9
1-FullIns_5	282	3247	8.2
1-Insertions_4	67	232	10.5
1-Insertions_5	202	1227	6.0
1-Insertions_6	607	6337	3.4
2-FullIns_3	52	201	15.2
2-FullIns_4	212	1621	7.2
2-FullIns_5	852	12201	3.4
2-Insertions_3	37	72	10.8
2-Insertions_4	149	541	4.9
2-Insertions_5	597	3936	2.2
3-FullIns_3	80	346	10.9
3-FullIns_4	405	3524	4.3
3-FullIns_5	2030	33751	1.6
3-Insertions_3	56	110	7.1
3-Insertions_4	281	1046	2.7
3-Insertions_5	1406	9695	1.0
4-FullIns_3	114	541	8.4
4-FullIns_4	690	6650	2.8
4-FullIns_5	4146	77305	0.9
4-Insertions_3	79	156	5.1
4-Insertions_4	475	1795	1.6
5-FullIns_3	154	792	6.7
5-FullIns_4	1085	11395	1.9
DSJC125.1	125	736	9.5
DSJC125.5	125	3891	50.2
DSJC125.9	125	6961	89.8
DSJC250.1	250	3218	10.3
DSJC250.5	250	15668	50.3
DSJC250.9	250	27897	89.6
DSJC500.1	500	12458	10.0

Inst.	$\|V\|$	$\|E\|$	D.
DSJC500.5	500	62624	50.2
DSJC500.9	500	112437	90.1
DSJC1000.1	1000	49629	9.9
DSJC1000.5	1000	249826	50.0
DSJC1000.9	1000	449449	90.0
DSJR500.1	500	3555	2.8
DSJR500.1c	500	121275	97.2
DSJR500.5	500	58862	47.2
anna	138	493	5.2
ash331GPIA	662	4181	1.9
ash608GPIA	1216	7844	1.1
ash958GPIA	1916	12506	0.7
david	87	406	10.9
flat300_20_0	300	21375	47.7
flat300_26_0	300	21633	48.2
flat300_28_0	300	21695	48.4
flat1000_50_0	1000	245000	49.0
flat1000_60_0	1000	245830	49.2
flat1000_76_0	1000	246708	49.4
fpsol2.i.1	496	11654	9.5
fpsol2.i.2	451	8691	8.6
fpsol2.i.3	425	8688	9.6
games120	120	638	8.9
homer	561	1628	1.0
huck	74	301	11.1
inithx.i.1	864	18707	5.0
inithx.i.2	645	13979	6.7
inithx.i.3	621	13969	7.3
jean	80	254	8.0
latin_square_10	900	307350	76.0
le450_5a	450	5714	5.7
le450_5b	450	5734	5.7

Table 10.2 – (2/2) For each instance: its numbers of vertices, edges, and the resulting density ("D.").

Inst.	$\|V\|$	$\|E\|$	D.
le450_5c	450	9803	9.7
le450_5d	450	9757	9.7
le450_15a	450	8168	8.1
le450_15b	450	8169	8.1
le450_15c	450	16680	16.5
le450_15d	450	16750	16.6
le450_25a	450	8260	8.2
le450_25b	450	8263	8.2
le450_25c	450	17343	17.2
le450_25d	450	17425	17.2
miles250	128	387	4.8
miles500	128	1170	14.4
miles750	128	2113	26.0
miles1000	128	3216	39.6
miles1500	128	5198	64.0
mug88_1	88	146	3.8
mug88_25	88	146	3.8
mug100_1	100	166	3.4
mug100_25	100	166	3.4
mulsol.i.1	197	3925	20.3
mulsol.i.2	188	3885	22.1
mulsol.i.3	184	3916	23.3
mulsol.i.4	185	3946	23.2
mulsol.i.5	186	3973	23.1
myciel3	11	20	36.4
myciel4	23	71	28.1
myciel5	47	236	21.8
myciel6	95	755	16.9
myciel7	191	2360	13.0
qg.order30	900	26100	6.5
qg.order40	1600	62400	4.9

Inst.	$\|V\|$	$\|E\|$	D.
qg.order60	3600	212400	3.3
queen5_5	25	160	53.3
queen6_6	36	290	46.0
queen7_7	49	476	40.5
queen8_8	64	728	36.1
queen8_12	96	1368	30.0
queen9_9	81	1056	32.6
queen10_10	100	1470	29.7
queen11_11	121	1980	27.3
queen12_12	144	2596	25.2
queen13_13	169	3328	23.4
queen14_14	196	4186	21.9
queen15_15	225	5180	20.6
queen16_16	256	6320	19.4
r125.1	125	209	2.7
r125.1c	125	7501	96.8
r125.5	125	3838	49.5
r250.1	250	867	2.8
r250.1c	250	30227	97.1
r250.5	250	14849	47.7
r1000.1	1000	14378	2.9
school1	385	19095	25.8
school1_nsh	352	14612	23.7
wap05a	905	43081	10.5
wap06a	947	43571	9.7
wap07a	1809	103368	6.3
wap08a	1870	104176	6.0
will199GPIA	701	6772	2.8
zeroin.i.1	211	4100	18.5
zeroin.i.2	211	3541	16.0
zeroin.i.3	206	3540	16.8

10.7.2 Comparison of CP models

Implementation

We implemented CP models using Gecode (version 4.2.1) [TEA08].

We cannot reasonably report results for all the possible combinations of the different improvements described in the previous sections. Instead, we chose a few configurations highlighting the changes brought about in the results by each improvement.

Base The basic model, only using **binary disequality** constraints. It uses a basic lower bound defined as the **sum of the smallest values** from each domain. **Bound consistency** is ensured during the search.

AllDiff+Bound A model that uses *allDifferent* **constraints**, as defined in Section 10.2. A lower bound is defined by using a **clique partition** and computing for each clique C the sum of its $|C|$ smallest values in the union of its variables' domains. **Bound consistency** is ensured. The parameters *gap* and *unc* governing the activation and deactivation of the lower bound computation algorithm are set to 20 % and 5, respectively.

AllDiff+Bound+Swap Same as *AllDiff+Bound*, but with **colour swapping** (see Section 9.1.2).

AllDiff+Bound+Swap+Dom Same as *AllDiff+Bound+Swap*, but with **domain consistency** instead of bound consistency.

AllDiff+SumBound+Swap Same as *AllDiff+Bound+Swap*, but lower bound computation is carried out using the bound consistency algorithm described in Section 10.4 ([BEL+12]), using **blocks of variables**. The parameters *gap* and *unc* are respectively set to 50 % and 0 (the best settings for this configuration).

A few other parameters are worth listing:

- For all these five configurations, the branch and bound (BAB) search engine was selected;
- When used, *allDifferent* constraints were represented by Gecode's "*distinct*" constraint;
- The consistency level used for domain consistency was GAC ("`ICL_DOM`"), while bound consistency corresponds to "`ICL_BND`";
- Gecode uses restarts by default. We chose the Luby policy, with a scale of 500.

Results

Table 10.3 compares the selected CP models on ten representative instances, while Table 10.4 gives a summary of global results obtained on the whole benchmark.

AllDiff+Bound outperforms *Base* on six instances out of these ten. Adding colour swapping (+*Swap*) allows an overall improvement of bounds, and proofs are generally performed faster. Replacing bound consistency (in *AllDiff+Bound+Swap*) with domain consistency (*AllDiff+Bound+Swap+Dom*) pays off on some instances, but hinders the solution process on some others.

Finally, using the *softAllDifferent* conjunction of constraints improves the solution process on a few instances, but also often degrades it. As explained in the dedicated section, we noticed that, in many cases, all variables within the scope of each *allDifferent* constraint used to compute bounds have very similar domains. Therefore, the bound computed and applied to their sum is very close to the sum of the smallest values found in the union of the domains.

As a conclusion, none of the proposed CP models appear to be competitive with state-of-the-art incomplete approaches, as even the best model (*AllDiff+Bound+Swap*) is able to reach the reference solution for only 49 instances.

Making proofs with CP None of the considered configurations appeared to be suitable to make proofs of optimality, as proofs were only made for eleven instances. The variable ordering heuristic, *MinElim*, is most probably at fault. It aims in priority at quickly finding good solutions, extensively exploring the search space. No focus is given whatsoever on demonstrating that a previously found solution is the optimal one.

To make our study more complete in this regard, we conducted experiments with a new CP configuration, designed to prioritize proof-making. It employs the hybrid restart policy hinted at back in Section 10.6, coupled with the aforementioned scheduled heuristic change.

This configuration uses the same settings as *AllDiff+Bound+Swap+Dom*, except that, as explained in the dedicated section, we change the variable ordering heuristic partway through the search: as soon as the search endured 50 consecutive restarts (the *nbRest* parameter) without having improved the global upper bound, *MinElim* is replaced by a heuristic that aims at proving optimality, namely *Dom/Ddeg*. When the variable ordering heuristic is changed, we also begin using a geometric restart policy, with a scale value of 100 and a base of 2.

Using this configuration, we are able to prove optimality for 15 instances instead of 11. Even though this makes up for an improvement of 36 %, this is still far from the state of the art. For reference, using all known bounds computed with heuristic approaches from the literature, optimality was proven for 21 of the 94 considered instances (some of these instances are also used in the present study) [JHH16].

10.7.3 Comparison of ILP models

We evaluated the impact of the two improvements that can be adapted to ILP in addition to being usable in CP solvers: the reduction of the set of colours

Table 10.3 – Detailed results of CP approaches on ten instances chosen to highlight the peculiarities of our methods. The improvements have been added gradually to outline the associated differences in results. For each method, from left to right, the distance to the reference bound is given, as well as the time needed to find the best upper bound returned by the method, and the time needed to prove its optimality if such an event occurred. Times are given in seconds.

	Base			AllDiff+Bound			AllDiff+Bound +Swap			AllDiff+Bound +Swap+Dom			AllDiff+SumBound +Swap		
Instance	Dist.	t_{UB}	t_{proof}	Dist.	t_{UB}	t_{proof}	Dist.	t_{UB}	t_{proof}	Dist.	t_{UB}	t_{proof}	Dist.	t_{UB}	t_{proof}
DSJC250.5	12.1	24 823		11.5	16 179		11.9	840		10.3	51 076		11.4	9 505	
DSJC1000.1	14.8	84 001		15	29 570		14.9	3 340		14.7	66 581		14.5	61 921	
ash331GPIA	0.3	14 439		0.6	55 543		0.4	30 122		0.3	6 755		0.7	14 258	
le450_5b	12.4	46 685		11.9	63 361		11.8	8 819		10.1	6 439		7.9	3 827	
3-Insert._3	0	0		0	0	18 650	0	0	17 838	0	0	19 542	0	0	
qg.order60	0	404		0	592	593	0	203	203	0	138	139	0	218	218
r125.1	0.4	4		0	1 472	4 193	0	1 561	4 797	0	1 570	4 713	0.4	0	
inithx.i.3	0.1	17 571		0.1	84 043		0	142		0	463		0.1	4	
school1	36.3	77 581		36.3	52 114		32	75 519		36.3	48 434		36.4	1 072	
school1_nsh	28.2	1 329		31.2	42 737		26.7	1 047		26.7	2 604		25.1	13 581	

Table 10.4 – A summary of the global results obtained by our CP approaches. The columns represent, from left to right: the average distance to reference bounds (Dist.); the number of times the reference bound was reached (Ref.); the number of proofs of optimality performed (Proofs); the number of times a memory out occurred (Mem.).

	Dist.	Ref.	Proofs	Mem.
Base	5.57	43	5	0
AllDiff+Bound	5.58	45	11	0
AllDiff+Bound+Swap	5.32	49	11	0
AllDiff+Bound+Swap+Dom	5.44	48	11	0
AllDiff+SumBound+Swap	5.34	45	8	0

initially considered for each vertex initial domain (represented by a decrease in the number of variables and constraints), and the introduction of *allDifferent* constraints (using constraints involving larger sums of variables, as explained in Section 10.2).

Implementation

We used ILOG's CPLEX solver (version 12.6.2) [CPL05]. To help CPLEX to avoid running out of memory, the two following adjustments were made:

- Depth-first search (DFS) was forced as a node selection strategy;
- The cuts factor was set to 1.5.

Previous experiments showed us that these parameters did not significantly lessen CPLEX's ability to solve the instances we use.

Results

Table 10.5 compares results of the initial ILP model as proposed in [WAN+12], simply denoted ILP, with a version that includes the two improvements, ILP+.

The overall amelioration observed in the results can be attributed, for the most part, to domain reduction, since reducing domains for ILP also removes a significant number of variables and constraints.

Fine-tuning the ILP approach allowed us to increase the number of optimality proofs from 61 to 65, and the number of times the reference solution has been found from 66 to 73. Besides, the number of memory outs goes down from 28 to 23, still being the weakest point of this approach, but to a lesser extent.

When comparing ILP+ with CP, we note that they perform very differently. For four of the highlighted instances (the `DSJC*` and `school*` classes), ILP+ ran out of memory. Therefore, the best solution if could find remains far from the

Table 10.5 – Detailed results of ILP approaches on ten instances chosen to highlight the peculiarities of our methods. For each method, from left to right, the distance to the reference bound is given (Dist.), as well as the time t_{UB} needed to find the best upper bound returned by the method, and the time t_{proof} needed to prove its optimality if such an event occurred. A "#M#" mark means that the search was aborted due to a lack of memory. Times are given in seconds.

	ILP			ILP+		
Instance	Dist.	t_{UB}	t_{proof}	Dist.	t_{UB}	t_{proof}
DSJC250.5	42.9	795		42.9	2457	#M#
DSJC1000.1	43.4	1	#M#	43.4	8138	#M#
ash331GPIA	1.8	7066		0	29870	
3-Insert._3	0	0	1	0	0	0
le450_5b	7.4	53963		3.6	22554	
qg.order60	96.7	0	#M#	6.1	86393	
r125.1	0	0	0	0	0	0
inithx.i.3	27	1	#M#	0	9	20
school1	114	7962	#M#	115.7	1	#M#
school1_nsh	111.9	3573		108.2	7990	#M#

reference solution, as well as from the best solution found with CP models. For qg.order60, the best solution found by ILP is far from optimality, whereas every single CP model was able to reach the optimum, with two of them even proving optimality. However, for the five remaining instances, ILP either finds better solutions (ash331GPIA, le450_5b), proves optimality quicker (3-Insertions_3, r125.1), or proves optimality while CP cannot (inithx.i.3).

Global results (on the whole benchmark) for ILP models are presented in Table 10.6. When considering these, it appears quite clearly that ILP+ is able to find reference solutions and to prove optimality much more often than CP, but the average distance to reference solutions is much larger, mostly because of the times it ran out of memory.

More generally, ILP+ ran out of memory for 23 instances, and for these instances it found solutions very far from reference solutions (and from the solutions found with CP). Over the 103 remaining instances, ILP found the reference solution 73 times, and proved optimality for 65 instances – more than one half of the benchmark.

Table 10.6 – A summary of the global results obtained by the basic and improved ILP approaches. The columns represent, from left to right: the average distance to reference bounds (Dist.); the number of times the reference bound was reached (Ref.); the number of proofs of optimality performed (Proofs); the number of times a memory out occurred (Mem.).

	Dist.	Ref.	Proofs	Mem.
ILP	73.36	66	61	28
ILP+	63.89	73	65	23

Chapter 11

Backtracking bounded by flower decomposition

Contents

The set of improvements described in Chapter 10 allowed CP and ILP to perform better. However, results are still disappointing, for CP (which is not as fast as one could expect) as well as for ILP (which uses extensive amounts of memory). This led us to research into ways to decompose sum colouring instances into subproblems in order to make these approaches more competitive still. A way to perform such a decomposition is to use backtracking bounded by tree decomposition.

As explained in the first part, BTD can be used on optimisation problems, provided that the objective function is decomposable: the global optimal assignment for a given assignment on the root must be obtainable by combining the optimal solutions of the subproblems. As a matter of fact, the sum colouring problem's CP model has a decomposable objective function: lowering the sum of the weights on a part of the graph always goes towards the global goal consisting in bringing the total sum down. Note that, for the classical colouring problem, a few adjustments would have to be made: the valued goods should be associated with equivalence classes rather than with precise assignments, and the colours should be arranged into a definite order so that the number of used colours can be deduced from the highest value found in an assignment.

We decided to take advantage of the straightforward decomposability of the sum colouring problem in order to design a BTD-inspired structural decomposition well suited to solve this problem.

11.1 Motivation and principle

When using BTD to solve an optimisation problem, a major drawback is that consistent assignments must be enumerated on each cluster that is not a leaf of the tree decomposition, as explained in Section 2.3.3. These enumerations occur recursively at each level of the tree and induce a large computational overhead, mostly to compute uninteresting partial solutions.

To alleviate this issue, we propose to significantly reduce the number of non-leaf clusters, since enumeration only takes place on these. More precisely, we use a tree decomposition having only one non-leaf cluster, corresponding to the root, for a resulting height of 1.

The idea of the *backtracking bounded by flower decomposition* (BFD) approach is to better exploit the structure of the instance. Besides, it is worth recalling that in the context of the sum colouring problem, the constraint graph corresponds to the graph that must be coloured.

More precisely, we aim at splitting $V \setminus C_r$ into k subsets $C_1, \ldots, C_k$, so that, for each colouring of C_r, the problems associated with these subsets may be solved independently. Furthermore, this offers a great opportunity to derive profit from the complementarity of CP and ILP approaches for this problem: CP is well suited to enumerate consistent assignments on the root (CP solvers like Gecode can provide one solution at a time on demand), and ILP can quickly solve subproblems to optimality.

During the search, lower and upper bounds can be recorded for the different leaf clusters: before attempting to solve a subproblem, if we know that we will have to pay at least a certain cost on the following subproblems, we can easily deduce a maximal acceptable cost for the cluster we are about to handle.

Moreover, we used a slight improvement in our implementation: if the maximal acceptable cost c computed for a subproblem p happens to be higher than $\mathrm{UB}_p - 1$, where UB_p is the upper bound computed for p, then we updated c to the value $\mathrm{UB}_p - 1$. It follows that in such conditions, no matter what happens during the solution of p, the upper and lower bounds of p will meet, and the optimal cost will be found.

11.2 Building a flower decomposition

This new method consists in computing a tree decomposition while bounding its height to 1. This basically means that apart from the root cluster C_r, every other cluster belongs to a set of leaves which are all children of C_r, as shown in Figure 11.1.

We build flower decompositions as follows:

1. Build a tree decomposition $\big(\mathcal{C},\ \mathcal{T} = (V_\mathcal{T}, E_\mathcal{T})\big)$ of the graph $G = (V, E)$, using for example *MinFill*;

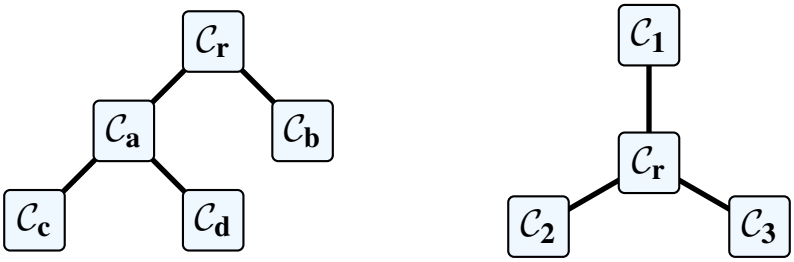


Figure 11.1 – Example shapes of classical tree decomposition (on the left) and flower decomposition (on the right).

2. Gather all separators of the decomposition into a set S:

$$S = \{\mathcal{C}_x \cap \mathcal{C}_y \mid x, y \in V_\mathcal{T} \ \wedge \ \mathcal{C}_x \cap \mathcal{C}_y \neq \varnothing\}$$

3. Select a subset S' of S to define a flower decomposition whose root is the cluster $\mathcal{C}'_r = \bigcup S'$;
4. Compute the subgraph of G induced by $V \setminus \mathcal{C}'_r$ and find its connected components $\mathcal{C}'_1, \ldots, \mathcal{C}'_k$. The connected components of this subgraph will serve as a base to define the leaves of the flower decomposition;
5. Extend each leaf cluster $\mathcal{C}'_i$ by adding to it any vertex of $\mathcal{C}'_r$ that is adjacent to at least one vertex of $\mathcal{C}'_i$ in G. Note that since the fill edges of the initial tree decomposition cease to exist as soon as this initial decomposition is complete (*i.e.*, at the end of the first step), they do not interfere here;
6. The obtained flower decomposition is $\big(\mathcal{C}', \ \mathcal{T}' = (V_{\mathcal{T}'}, E_{\mathcal{T}'})\big)$, where:
   - $\mathcal{C}' = \{\mathcal{C}'_r, \ \mathcal{C}'_1, \ \ldots, \ \mathcal{C}'_k\}$
   - $V_{\mathcal{T}'} = \{v_r, \ v_1, \ \ldots, \ v_k\}$
   - $E_{\mathcal{T}'} = \big\{\{v_r, v\} \mid v \in \{v_1, \ldots, v_k\}\big\}$

An example of this procedure is shown in Figure 11.2.

This process is, overall, similar to a method recently employed by JÉGOU, KANSO and TERRIOUX [JKT15], who also demonstrated that it yields valid tree decompositions.

Handling poorly structured instances

For instances that are poorly structured, tree decomposition might result in the creation of a single monolithic cluster holding all the variables of the problem. In such a case, the decomposition process itself goes to waste and the problem does not get any easier to solve.

Building on BFD's idea though, we can handle such cases in a slightly better way, forcefully decomposing them to a certain extent.

When a tree decomposition only contains one cluster, we may decompose its set V of vertices into two subsets: $\mathcal{C}_r$ and $V \setminus \mathcal{C}_r$. We then use CP to enumerate

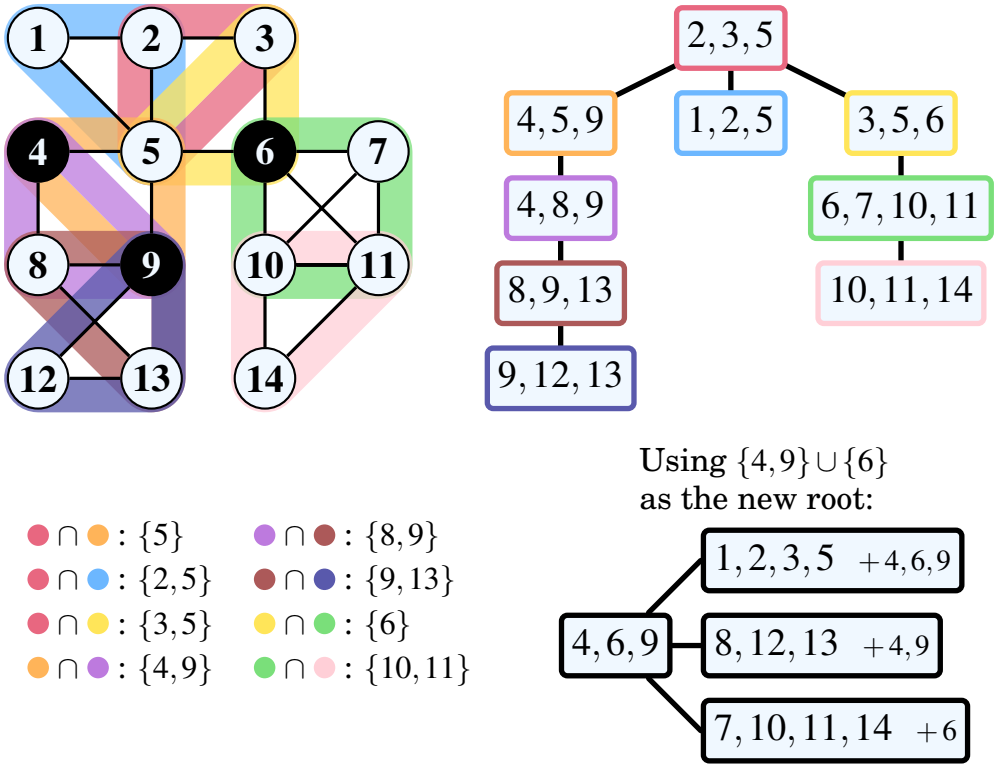


Figure 11.2 – The tree decomposition of Figure 1.7, the corresponding set of separators, and a flower decomposition resulting from the selection of an example subset of these separators to build the new root cluster. The vertices that form this new root are highlighted in black in the initial graph. In the final decomposition, the vertices listed after the "+" sign are those that were subsequently introduced into the different leaf clusters in order to form new separators: they are vertices from the root that are linked to at least one vertex of the considered leaf cluster.

all proper colouring of $\mathcal{C}_r$. For each of these colourings, we use ILP to find the optimal sum colouring of $V \setminus \mathcal{C}_r$, taking into account the assignments performed on $\mathcal{C}_r$. Each solution is thus extended to the whole problem in an optimal fashion. It follows that the best global solution found during this enumeration and optimisation process is the optimal solution for the initial considered problem.

11.3 Aiming for a good flower decomposition

In the building process outlined in the previous section, no information was given on the way the subset of separators has to be chosen at step 3. We will now cover this particular topic.

11.3.1 Size of the separators

In BTD, a limit is also enforced on the size of the separators between the clusters of the tree decomposition. This is usually done by merging clusters whose separators contain too many variables.

Regarding BFD, a similar approach is used. In BFD's case, however, we keep the clusters untouched. Instead, if a separator s exceeds the size limit, the only effect is that no valued good will be recorded on s during the search, since doing so would consume a large amount of memory. Moreover, if a separator happens to correspond to the full root cluster, there is no need to record any valued good on it, since assignments on such a separator cannot occur twice.

11.3.2 Size of the leaves

To obtain an advantageous flower decomposition, the key point is to build $\mathcal{C}_r$ in such a way that the generated leaves are small enough to be properly solved to optimality with ILP. To this end, we introduce a parameter l, that enforces a limit on the size of the leaves in terms of a percentage of the total number of vertices. More precisely, for each leaf cluster $\mathcal{C}_i$, we ensure that $|\mathcal{C}_i \setminus \mathcal{C}_r| \leq l \times |V|$. Besides, we also carefully favour decompositions offering small root clusters, since we have to enumerate every proper colouring on this cluster.

Our goal is to find the subset S' such that the resulting flower decomposition satisfies the size limit l on the leaf clusters while keeping the size of $\mathcal{C}'_r$ at a minimum. We use `Gecode` to look for a subset of separators satisfying those criteria. As it is an $\mathcal{NP}$-hard problem, we only allot limited time to this optimisation phase, and use the best flower decomposition computed within this limit. More details on this topic follow.

11.3.3 CP model to choose separators

To choose a suitable set of separators from the original tree decomposition in order to form the root of the flower decomposition, a simple constrained optimisation model can be built:

- A **variable** is created for each existing separator. The **domains** of these variables are all equal to $\{0,1\}$: a separator will be used as part of the new root if and only if the considered solution assigns 1 to the corresponding variable;
- A single major **constraint** is defined. It ensures that removing from the graph every separator whose variable is set to 1 does not create connected components with a size exceeding the limit fixed through the l parameter of BFD. Naturally, CP solvers do not provide any propagator for such a constraint, and it has to be implemented by the user. Furthermore, it is strongly advised to run this propagator only once every variable is assigned. Indeed, aside from the fact that this propagator is rather costly to run, we must be aware that at the beginning of the search, since no separator is marked for removal, there would be only a single, large connected component.
  For a set of variables X, a graph G and a limit l, this constraint can be written as $\mathit{LimLeaf}(X,\ G,\ l)$.
- The **objective** function aims at minimizing the size of the future root, *i.e.* the number of distinct vertices in the union of the separator whose variables are set to 1.

The heuristics we used are also rather straightforward:

- The **variables** prioritized for assignment are those corresponding to the largest available separators;
- The **value** 1 is favoured over 0.

We chose to run the CP solver `Gecode` for 15 minutes on this model and use the best solution found within this time.

If this CSP turns out to be inconsistent (or if no solution could be found fast enough, although this is rather unlikely), we built the decomposition using a greedy algorithm instead as a fallback. This algorithm iteratively selects individual vertices, starting with those having the highest degree. It stops when the removal of the set of selected vertices from the considered graph generates connected components observing the limit stemming from BFD's usual l parameter.

To alleviate the computations throughout the run of this greedy algorithm, several vertices are selected at each step before checking whether the resulting connected components fit the requirements. The algorithm starts with one vertex, and proceeds by adding *nbSelect* new ones at each step. This *nbSelect* value is computed through the ratio $\mathit{maxSizeRoot}/\mathit{maxNbSteps}$, where:

- *maxSizeRoot* is simply the maximal number of vertices that can possibly be in the final root: it is computed by considering a case in which the decomposition has only two clusters and where the sole leaf cluster has the maximal number of vertices allowed by the l parameter. It is thus equal to $|V| \times (1-l)$ for a graph (V,E);
- *maxNbSteps* is an additional parameter aiming at making sure the total number of steps performed by this algorithm does not exceed a certain number. Intuitively, it corresponds to the number of steps that will be needed in the worst case, leading to a trivial decomposition of only two clusters.

After some experiments, we decided to set *maxNbSteps* to a value of 50.

Computing the connected components resulting from the removal of a given set of vertices has a time complexity of $\mathcal{O}\big(|V|+|E|\big)$. Therefore, the complexity of the whole greedy algorithm, which performs such computations a parameter-bounded number of times, is $\mathcal{O}\big(|V|+|E|\big)$ as well.

11.4 BFD summary

Putting together all the information given so far on BFD, we can summarize the way we use this approach as written in Algorithm 11.1. This algorithm is largely reminiscent of the Algorithm 2.2, which was provided earlier for the optimization case of BTD. The main difference is, obviously, that the recursive call has been replaced with a conversion of the subproblem into an ILP model, followed by a call to the CPLEX ILP solver.

Solutions for the root cluster are enumerated by the outer loop (line 4). For each of these consistent assignments, bounds are computed for the corresponding subproblems (lines 5–11). Then, any subproblem that is not trivially useless is solved as an ILP model (lines 14–19). The bounds associated with the considered subproblem are then updated according to the ILP solver's results (lines 20–23). Throughout the algorithm, values goods are represented by tuples containing an assignment on the relevant separator, a lower bound, as well as an upper bound for the underlying subproblem.

11.5 Experimental results

This experimental section aims at comparing the standard BTD approach with the new BFD decomposition method in the context of sum colouring.

11.5.1 Configurations

The classical approach will be referred to simply as BTD. A few precisions on this configuration ought to be given:

- The tree decomposition is built using the *MinFill* algorithm;
- CP is used to solve subproblems;
- Leaf clusters are solved with the Gecode configuration *AllDiff+Bound+Swap*, that uses restarts;
- On the other hand, non-leaf clusters are solved with the same configuration, but without restarts, as we need to enumerate all solutions.

We also consider two configurations that employ the newly presented flower decomposition. These methods will be referred to as BFD l, where l will be replaced by the value of the parameter setting a limit on the size of leaf clusters, as seen in the previous sections. Here are a few configuration details:

- Constraint programming (*AllDiff+Bound+Swap* without restarts) is used to enumerate the solutions of the non-leaf clusters (in BFD's context, this only concerns the root);
- In BFD, ILP+ is used to solve the subproblems induced by the leaves for each assignment of the separators, once a complete consistent assignment of the root cluster has been found;
- The maximal size for the separators is set to 30;
- Two different values are used for the l parameter: 75% and 90%. The resulting methods will thus be referred to as BFD 75 and BFD 90, respectively.

11.5.2 Results

Table 11.1 reports experimental results of BTD, BFD 90 and BFD 75 on the ten highlighted instances from previous experimental sections of this part.

When looking into these detailed results on our ten representative instances, one can notice that they have complementary results:

- BTD is better than BFD 90 and BFD 75 on `DSJC250.5` and `school1`;
- BFD 90 is better on `ash331GPIA`, `3-Insertions_3`, `inithx.i.3`, `school1_nsh` and `r125.1`;
- BFD 75 obtains the best results on `DSJC1000.1`, `le450_5b`, `qg.order60` and `r125.1`.

Moreover, for two of these instances (namely, `r125.1` and `school1_nsh`), the best results, over all the approaches considered in this section, are actually obtained by BFD 90.

Table 11.2 shows a summary of the global results for these same methods. When looking at those results, we note that BTD is able to find the reference solution for only 17 instances (instead of 46 and 48 for BFD 90 and BFD 75 respectively), and proves optimality for only 13 instances, when BFD 90 and BFD 75 perform 39 and 26 proofs respectively. Actually, most of the considered instances do not show any particular structure, or only a very poor one, and BTD is generally outperformed on them by the CP approaches that were evaluated

in Section 10.7.2.

BFD 90 and BFD 75 are able to find reference solutions and to prove optimality for much more instances than BTD. Actually, on 40 of the 126 instances, we noticed that the graph's structure is so poor that there is only one single cluster in the resulting tree decomposition. In such situations, BFD is still able to build a flower decomposition by creating a root cluster that contains $|V| \times (1-l)$ nodes along with a leaf cluster comprised of the remaining $|V| \times l$ variables. This often allows BFD to behave much better than BTD.

Still, BFD suffers from a relatively high average distance to reference solutions. A partial explanation for this issue is that, on some instances, BFD spends a lot of time enumerating valid colourings of the root cluster, while many of these partial colourings cannot be extended to obtain good solutions. BFD has no trivial way of noticing such occurrences and wastes a lot of time solving to optimality useless subproblems.

Comparing BFD 90 and BFD 75 proves that allowing larger leaf clusters to be created increases the memory needs, but also eases the computation of upper bounds. Indeed, it makes the root cluster smaller (alleviating the enumeration steps) and gives ILP a more global view of the problem, preventing it in some cases to spend too much time solving a useless subproblem to optimality.

When comparing BFD with the CP approaches of Section 10.7.2, we note an increase in the number of proofs (39 and 26 instead of 11), but the average distance to reference solutions is significantly larger.

Similarly, BFD reaches the reference bound less often than ILP+, and performs fewer optimality proofs. However, BFD's lower memory needs allowed it to have its solution process aborted more rarely than ILP+.

Putting these observations together, one can conclude that BFD and ILP+ have complementary performance. BFD 90 performs strictly better than CPLEX on 21 instances, and strictly worse on 91 instances. BFD 75 offers marginally better results, beating CPLEX on 28 instances while performing worse on 85 instances.

Algorithm 11.1: $\text{BFD}_{\text{CPLEX}}\big((X,D,C,f),\ (\mathcal{C},\mathcal{T}),\ r,\ maxcost\big)$

Input: A COP instance (X,D,C,f);
A tree decomposition $(\mathcal{C},\mathcal{T})$;
The root node r from $\mathcal{T}$;
A maximum acceptable cost $maxcost$.
Output: The optimal value for (X,D,C,f) if it is at most $maxcost$;
$maxcost+1$ otherwise.

Let $\mathcal{P}$ be the CSP (X,D,C) reduced to the subset of variables from $\mathcal{C}_r$
$best \leftarrow maxcost+1$
Let Ch be the set of children of r in $\mathcal{T}$
foreach *solution S of $\mathcal{P}$ provided by* `Gecode` **do**
  foreach *child $i \in Ch$* **do**
    /* Initialize bounds with values (theoretical, etc.) computed during preprocessing. */
    $\text{LB}_i \leftarrow init\text{LB}_i$
    $\text{UB}_i \leftarrow init\text{UB}_i$
    Let A_i be the tuple of values assigned in S to the separator between $\mathcal{C}_r$ and $\mathcal{C}_i$
    if *a valued good $(A_i,\text{LB},\text{UB})$ exists for A_i* **then**
      $\text{LB}_i \leftarrow$ LB from $(A_i,\text{LB},\text{UB})$
      $\text{UB}_i \leftarrow$ UB from $(A_i,\text{LB},\text{UB})$
  if $f(S)+\sum_{i\in Ch}\text{LB}_i < best$ **then**
    while $f(S)+\sum_{i\in Ch}\text{LB}_i < best$ **and** *there is a child $i \in Ch$ such that no optimal valued good exists for the assignment A_i of $\mathcal{C}_i$'s separator in S* **do**
      /* Either A_i is not a valued good, either it is a valued good with $\text{LB} < \text{UB}$. */
      Let i be a child of r such that A_i is not an optimal valued good
      Let $\mathcal{T}_i$ be the subtree of $\mathcal{T}$ rooted in i
      Let $newD$ be the current domains
      /* Convert to ILP the subproblem rooted in i, assigning the separator's variables according to A_i. */
      $(X',C',f') \leftarrow \text{CONVERTCPTOILP}\big((X,newD,C,f),\ i\big)$
      $maxcost_i \leftarrow \min\{best-1-f(S)-\sum_{j\in Ch, i\neq j}\text{LB}_j,\ \text{UB}_i-1\}$
      $best_i \leftarrow \text{CPLEX}\big((X',C',f'),\ maxcost_i\big)$
      if $best_i \leq maxcost_i$ **then**
        Record the valued good $(A_i,best_i,best_i)$
      else
        Record the valued good $(A_i,best_i,\text{UB}_i)$
    $best_r \leftarrow f(S)+\sum_{i\in Ch}\text{LB}_i$
    if $best_r < best$ **then**
      $best \leftarrow best_r$
return *best*

Table 11.1 – Detailed results of BTD and our two selected BFD configurations on the usual ten highlighted instances. For each method, from left to right, the distance to the reference bound is given, as well as the time needed to find the best upper bound returned by the method, and the time needed to prove its optimality if such an event occurred. A "#M#" mark means that the search was aborted due to a lack of memory. Times are given in seconds.

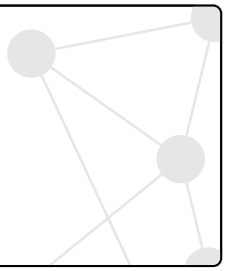
	BTD			BFD 90			BFD 75		
Instance	Dist.	t_{UB}	t_{proof}	Dist.	t_{UB}	t_{proof}	Dist.	t_{UB}	t_{proof}
DSJC250.5	36.4	1		51.2	86400		395.9	0	#M#
DSJC1000.1	43.2	62		463.1	0	#M#	38.7	86400	
ash331GPIA	23.4	13		1.1	86400		6.7	1894	
3-Insert._3	0	25	1796	0	261	331	0	857	1007
le450_5b	65	16237		41.8	86400		39.5	86400	
qg.order60	0.6	4193		5	86400		0.4	86400	
r125.1	0	0	1	0	0	0	0	0	0
inithx.i.3	1.2	5		0	106	686	230.3	10907	
school1	107.8	82466		628.5	0	#M#	628.5	0	#M#
school1_nsh	98.2	10747		6.1	86400		67.1	86400	

Table 11.2 – A summary of the global results of our decomposition approaches, with BTD for comparison purposes. The columns represent, from left to right: the average distance to reference bounds (Dist.); the number of times the reference bound was reached (Ref.); the number of proofs of optimality performed (Proofs); the number of times a memory out occurred (Mem.).

	Dist.	Ref.	Proofs	Mem.
BTD	24.42	17	13	6
BFD 90	83.4	46	39	18
BFD 75	85.05	48	26	16

Chapter 12

Computing local and global bounds using partial solutions



Contents

An issue frequently encountered during our experiments with BFD is that it might take a long time to find the first good solutions, which are essential to prune branches during the search and to avoid spending too much time solving uninteresting subproblems.

As explained in Section 11.1, lower bounds computed on the different clusters can be used to detect trivially useless subproblems. However, this proved to be insufficient when using only simple theoretical lower bounds, due to the overall low quality of such bounds. We thus investigated ways to obtain better bounds at an affordable cost.

Similar goals were pursued in different contexts in the literature, for example when applying a BTD approach to a WCSP [DSV06; SAN+09].

12.1 Local bounds

To improve our approach in this regard, we designed an improvement that can be used as a preprocessing step for BFD, plugged between the computation of the flower decomposition and the actual beginning of the resolution. We simply use ILP models and CPLEX to find good sum colourings on the clusters of the decomposition, taken independently, including the root cluster.

To be able to combine and use the resulting bounds more easily, we make sure every vertex of the graph is considered only once. On account of this, when we compute bounds for the leaf clusters, the separator is not included. On the other hand, the bound computed on the root cluster does indeed take the underlying separators into consideration. This ensures that the sum of the lower bounds computed for each cluster provide a valid global lower bound.

The ILP solver is thus run on each of these parts of the graph, with a given time limit.

- If, for a given cluster, the time imparted to compute the partial colouring proves to be enough to obtain the **optimal local solution**, a bound is directly given by the sum s associated with this colouring. Indeed, we know that, even without the constraints coming from an already coloured separator, it would still cost s to fully colour this cluster.
- If, on the other hand, **a timeout occurs** when computing this local bound, we cannot use the current upper bound of the ILP solver as a lower bound for the cluster, since it might be too high to be a correct bound. In such a case, we use instead the current lower bound of the ILP solver and pass it on as a lower bound for the cluster.

These lower bounds can be employed during the solution process to avoid solving some subproblems: since they give a minimal cost that will necessarily be paid to colour a part of the graph, the algorithm can tell whether the subproblem is worth solving depending on the cost already paid on the rest of the instance. Of course, this "cost already paid" may involve other lower bounds from the next subproblems. Details on how bounds are used can be found in Chapter 11.

12.2 First solution and global bound

These ILP solver runs do not only provide lower bounds, but also partial colourings (colourings computed independently on the different parts, namely each leaf cluster without its separator, and the root cluster). Some might be optimal with regards to the subgraph on which they were computed; some might simply correspond to the best solution the ILP solver could find before a timeout occurred.

Theoretically, it could happen that the ILP solver does not find any solution fast enough on at least one of the considered subgraphs. However, it is worth remarking that this never actually occurred during our many experiments, except on instances where ILP ran out of memory, thus compromising not only the bound computations, but also the solution process itself.

If at least one valid colouring has been found on each of the considered subgraphs, these partial solutions can easily be stitched together to form a global solution. We simply reintroduce the constraints that stand between clusters and that were ignored when the partial colourings were computed. Once these constraints are put back in effect though, conflicts appear: some vertices using the same colour in distinct partial solutions become neighbours. We solve these conflicts by first unassigning the variables corresponding to conflicting vertices (*i.e.*, we uncolour these vertices).

More formally, for each edge u, v of the graph, if the variables corresponding to u and v have the same value, these variables are unassigned. Note that in our current implementation the edges are considered in an arbitrary order during this phase.

Once there are no conflicts left, we build a problem P whose objective amounts to reassigning every variable that has just been unassigned. The values of other variables will remain unchanged during this recolouring step – actually, these variables are not even included in the problem P.

More precisely, P is defined as follows:

- The improved ILP **model** from previous sections (see, in particular, Sections 9.3.3 and 10) is used;
- The considered **vertices** (those that will be represented by variables of P) are those that were uncoloured during the conflict-resolution phase;
- The considered **colours** for each involved vertex v are those from the set $\{1,\dots,\deg(v)+1\}$, from which we remove every colour that is used by at least one coloured neighbour of v.

Since the number of considered vertices and colours is fairly reduced, the number of constraints needed by the model is also lower than in most ILP sum colouring models.

This recolouring problem could actually be solved using any kind of model and solver. We chose to use an ILP model and to solve it using CPLEX, mostly because these subproblems very seldom grow to sizes that CPLEX cannot handle. We also performed experiments using `Gecode` instead, but the results were marginally less interesting.

Moreover, regarding the conflict-resolution phase, we considered unassigning only one of the two conflicting variables for each problematic edge, instead of the variables corresponding to both extremities. However, it appeared that giving slightly more liberty to whichever solver has to recolour the vertices afterwards was generally beneficial (to a minor extent).

By using the combination of partial solutions and overriding the colours of conflicting vertices with the solution of the recolouring problem, we obtain a global solution, which offers an upper bound which is generally significantly better than theoretical ones.

Figure 12.1 depicts the whole process described in this section, using a basic example.

A few details have to be checked upon before actually using the constructed global solution:

1. Firstly, the repairing process might have introduced suboptimalities in the colour choices of certain nodes. It can occur that a variable x eventually uses a colour c while actually having the opportunity of using a cheaper colour c' ($c' < c$) unused by the neighbouring variables. To cope with this, we simply scan every variable of the problem and lower their values whenever it is possible to do so without generating any spurious conflict. The computational cost of this operation is, of course, negligible.
2. Finally, we check that no colour swap can improve the obtained solution (see Section 9.1.2). If the global solution appears to be a dominated colouring, we apply the necessary colour swaps in order to lower the

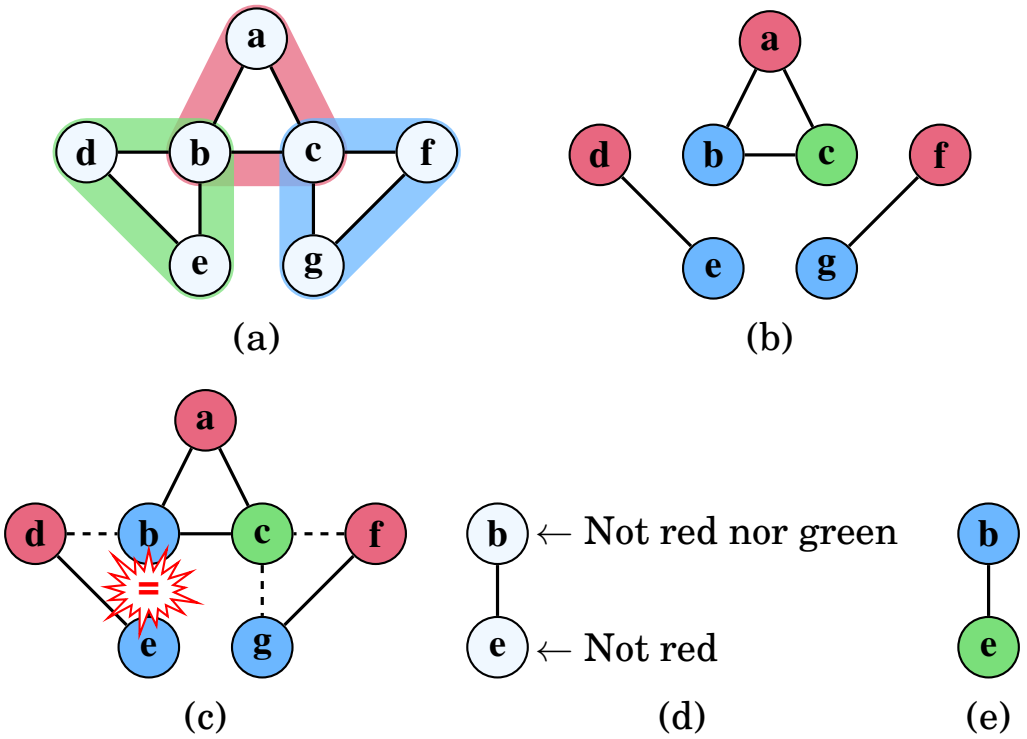


Figure 12.1 – A flower decomposition with three clusters (a); the resulting subproblems coloured to obtain initial bounds (b); the tentative combination of the partial solutions obtained, with a conflict between b and e (c); the subproblem defined to resolve this conflicting situation, in which initial domains are reduced by removing values already used by the respective neighbours of the involved vertices (d); and finally, the chosen colours for these two vertices (e).

global cost.

12.3 Allotted time for bounds

The partial colourings necessary for those bounds are built while observing a given time limit t_{LB}. Therefore, they might be suboptimal in some cases. The limit t_{LB} is computed as a fraction of the global time limit imposed on the full solution process. Quick experiments showed us that allotting 5 % of our total time of 24 hours was enough to obtain good results. Going beyond this fraction of time allowed for no significant improvements, either in the quality of the resulting local colourings (which remained practically unchanged) or in the global results themselves.

Since there are several bounds (one per cluster) to compute, the allotted time must be shared between them. The approach we adopted consists in sorting the clusters according to their number of variables and starting with the smallest ones. A reference time limit of t_{LB}/nbc is used, nbc being the total number of clusters. If, for a given cluster, the optimal local sum colouring is found in a time t lower than t_{LB}/nbc, the remaining time $t_{\mathrm{LB}}/nbc - t$ is equally shared between the clusters that have not been considered yet. Therefore, if the smaller clusters are easy to handle, more time will be available to tackle the more complex cases, towards the end of the process.

A simple way to perceive and to implement this method is to consider a common time pool, initialized at t_{LB}. Each time a new cluster has to be processed, we allot it a portion of $1/(nbc-n)$ of the pool, where n is the number of clusters already processed. If the optimal solution is found, the remaining time is stored in the pool.

On the other hand, the step consisting in repairing the resulting global solution by uncolouring a few vertices and recolouring them can generally be completed within a reasonable amount of time. Even in cases where it takes more time, it can be considered an acceptable cost since it is still necessarily easier than solving the whole problem and it yields a first solution. Therefore, we did not set any particular time limit for this step and always seek the optimal way to recolour the set of uncoloured vertices.

12.4 Results

The general setup for these experiments was the same as for the previously described ones (see Section 10.7.1).

Table 12.1 details the experiments we conducted in order to settle for a time portion to allot to the lower bound computation phase. It shows that spending too much time on this particular step prevents the solver from performing one of the proofs of optimality made in the traditional context. Furthermore, the

Table 12.1 – A summary of the results we observed when trying to figure out which portion of our 24 h time limit to allot to initial bound computations. We show the minimal, average and maximal distance to reference upper bounds, followed by the number of proofs of optimality, of errors due to a lack a memory. "Found best" denotes the number of times the reference bound was reached, and "Times best" the number of times the considered approach was the best among those presented in this table.

		3 %	5 %	7 %	10 %
Dist	Min	−0.1	−0.1	0	−0.1
	Avg	80.6	80.6	76.6	80.5
	Max	1 119.5	1 119.5	1 119.5	1 119.5
Proofs		43	43	42	42
Mem. out		18	18	17	18
Found best		57	58	57	58
Times best		62	49	44	46

upper bound value at the end of the solving process does not appear to get significantly lower when allotting more time to the initial computations. We thus settled for a portion of 5 %, which seemed to bring a good balance. On a side note, three versions out of the considered four could beat a reference bound, hence the negative minimal distances showcased in the table.

Tables 12.2 to 12.5 show in details the bounds obtained on most instances and compare them with the reference and theoretical bounds. These bounds correspond to the approach consisting of allotting 5 % of the total time of 24 h to bound computations, with a flower decomposition where leaves are limited to 90 % of the problem's variables. Only the 105 instances comprised of a single connected component are shown in order to ensure that the comparisons – especially with the reference bounds – are relevant.

We note that the overall quality of these bounds is satisfying. The cost of the first solution often corresponds to less than half of the theoretical upper bound, and the distance to reference bounds is often brought below 10 %. On 24 of these 105 instances, the first solution even reaches the reference bound. It can also be noticed that the improvement is generally stronger when a significant amount of time was needed to find the first solution. This proves that spending five percent of the global time limit on this phase can be worthwhile. Still, we cannot overlook the 13 instances on which the search was aborted due to a lack of memory. It can be noted, however, that this occurs – as one could expect – on rather difficult instances.

Table 12.6 shows a comparison of the standard version of BFD with versions using either just the new lower bounds, or this as well as the global solution

Table 12.2 – Details of the bounds obtained by combining partial colourings to obtain a first global solution (part 1 of 4). "Ref." is the reference upper bound, "LB" and "UB" are the theoretical bounds, "FS" the cost of the first solution, "Imp." the improvement, in percentage, between UB and FS, "Dist." the distance, in percentage, remaining between FS and the reference bound, and "t" the number of seconds used to compute the partial colourings and get the first solution. A "#M#" mark means that CPLEX ran out of memory when computing the bounds. Note that for the sake of simplicity, only instances with only one connected component are shown (105 out of 126).

Instance	Ref.	LB	UB	FS	Imp.	Dist.	t
1-FullIns_3	54	45	130	54	58	0	0
1-FullIns_4	166	135	686	166	76	0	11
1-FullIns_5	499	389	3529	500	86	0	316
1-Insertions_4	119	92	299	119	60	0	0
1-Insertions_5	357	271	1429	361	75	1	50
1-Insertions_6	1068	801	6944	1096	84	3	2165
2-FullIns_3	93	79	253	94	63	1	1
2-FullIns_4	363	298	1833	374	80	3	32
2-FullIns_5	1433	1151	13053	1475	89	3	2164
2-Insertions_3	62	52	109	62	43	0	0
2-Insertions_4	249	200	690	259	62	4	7
2-Insertions_5	996	790	4533	996	78	0	996
3-FullIns_3	145	124	426	145	66	0	1
3-FullIns_4	683	572	3929	696	82	2	224
3-FullIns_5	3335	2771	35781	4032	89	17	2306
3-Insertions_3	92	78	166	93	44	1	1
3-Insertions_4	459	383	1327	470	65	2	46
3-Insertions_5	2289	1884	11101	2320	79	1	2321
4-FullIns_3	205	184	655	205	69	0	0
4-FullIns_4	1138	977	7340	1140	84	0	2162
4-FullIns_5	6679	5667	81451	#M#			
4-Insertions_3	127	111	235	130	45	2	2
4-Insertions_4	761	654	2270	768	66	1	561
5-FullIns_3	280	247	946	283	70	1	1
5-FullIns_4	1776	1535	12480	1948	84	9	2207
DSJC125.1	326	208	861	349	59	7	2161
DSJC125.5	1012	412	4016	1404	65	28	2165

Table 12.3 – Details of the bounds obtained by combining partial colourings to obtain a first global solution (part 2 of 4). "Ref." is the reference upper bound, "LB" and "UB" are the theoretical bounds, "FS" the cost of the first solution, "Imp." the improvement, in percentage, between UB and FS, "Dist." the distance, in percentage, remaining between FS and the reference bound, and "t" the number of seconds used to compute the partial colourings and get the first solution. A "#M#" mark means that CPLEX ran out of memory when computing the bounds. Note that for the sake of simplicity, only instances with only one connected component are shown (105 out of 126).

Instance	Ref.	LB	UB	FS	Imp.	Dist.	t
DSJC125.9	2503	1405	7086	2974	58	16	2167
DSJC250.1	970	474	3468	1208	65	20	2164
DSJC250.5	3210	930	15918	4309	73	26	2187
DSJC250.9	8277	3171	28147	#M#			
DSJC500.1	2836	987	12958	3871	70	27	2167
DSJC500.5	10886	2111	63124	#M#			
DSJC500.9	29862	8082	112937	#M#			
DSJC1000.1	8991	2116	50629	12773	75	30	2964
DSJC1000.5	37575	4642	250826	#M#			
DSJC1000.9	103445	19221	450449	#M#			
DSJR500.1	2156	1822	4055	2187	46	1	2166
DSJR500.1c	16286	13441	121775	#M#			
DSJR500.5	25440	19202	59362	30427	49	16	2185
anna	276	270	631	276	56	0	0
ash331GPIA	1432	1061	4843	1467	70	2	2178
ash608GPIA	2600	1942	9060	2767	69	6	3068
ash958GPIA	4172	3090	14422	4478	69	7	3074
david	237	226	493	237	52	0	0
flat300_20_0	3150	1126	21675	5770	73	45	2183
flat300_26_0	3966	1166	21933	6107	72	35	2179
flat300_28_0	4238	1145	21995	5717	74	26	2177
flat1000_50_0	25500	4564	246000	#M#			
flat1000_60_0	30100	4603	246830	#M#			
flat1000_76_0	37164	4680	247708	#M#			
games120	443	412	758	448	41	1	3
latin_square_10	41444	35547	308250	#M#			

Table 12.4 – Details of the bounds obtained by combining partial colourings to obtain a first global solution (part 3 of 4). "Ref." is the reference upper bound, "LB" and "UB" are the theoretical bounds, "FS" the cost of the first solution, "Imp." the improvement, in percentage, between UB and FS, "Dist." the distance, in percentage, remaining between FS and the reference bound, and "t" the number of seconds used to compute the partial colourings and get the first solution. A "#M#" mark means that CPLEX ran out of memory when computing the bounds. Note that for the sake of simplicity, only instances with only one connected component are shown (105 out of 126).

Instance	Ref.	LB	UB	FS	Imp.	Dist.	t
le450_5a	1350	928	6164	1403	77	4	2165
le450_5b	1350	952	6184	1787	71	24	2164
le450_5c	1350	992	10253	1350	87	0	102
le450_5d	1350	987	10207	1350	87	0	2162
le450_15a	2632	1818	8618	2895	66	9	2167
le450_15b	2632	1871	8619	2945	66	11	2162
le450_15c	3487	1809	17130	5053	71	31	2235
le450_15d	3505	1821	17200	5261	69	33	2389
le450_25a	3153	2622	8710	3332	62	5	2169
le450_25b	3365	2836	8713	3533	59	5	2165
le450_25c	4515	2627	17793	5628	68	20	2896
le450_25d	4544	2431	17875	5748	68	21	2522
miles500	705	579	1298	709	45	1	16
miles750	1173	1025	2241	1186	47	1	515
miles1000	1666	1399	3344	1703	49	2	90
miles1500	3354	3151	5326	3362	37	0	11
mug88_1	178	156	220	179	19	1	0
mug88_25	178	150	220	178	19	0	0
mug100_1	202	175	250	202	19	0	1
mug100_25	202	175	250	202	19	0	1
myciel3	21	15	31	22	29	5	0
myciel4	45	31	94	46	51	2	0
myciel5	93	61	283	93	67	0	2
myciel6	189	122	850	189	78	0	10
myciel7	381	246	2551	382	85	0	760
qg.order30	13950	13950	27000	13953	48	0	305

Table 12.5 – Details of the bounds obtained by combining partial colourings to obtain a first global solution (part 4 of 4). "Ref." is the reference upper bound, "LB" and "UB" are the theoretical bounds, "FS" the cost of the first solution, "Imp." the improvement, in percentage, between UB and FS, "Dist." the distance, in percentage, remaining between FS and the reference bound, and "t" the number of seconds used to compute the partial colourings and get the first solution. A "#M#" mark means that CPLEX ran out of memory when computing the bounds. Note that for the sake of simplicity, only instances with only one connected component are shown (105 out of 126).

Instance	Ref.	LB	UB	FS	Imp.	Dist.	t
qg.order40	32 800	32 800	64 000	33 110	48	1	2 279
qg.order60	109 800	109 800	216 000	#M#			
queen5_5	75	69	185	75	59	0	0
queen6_6	138	107	326	142	56	3	3
queen7_7	196	138	525	201	62	2	49
queen8_8	291	194	792	292	63	0	2 160
queen8_12	624	624	1 464	632	57	1	1
queen9_9	409	346	1 137	423	63	3	2 162
queen10_10	553	384	1 570	595	62	7	2 161
queen11_11	733	476	2 101	802	62	9	2 162
queen12_12	943	699	2 740	1 021	63	8	2 164
queen13_13	1 191	715	3 497	1 343	62	11	2 161
queen14_14	1 482	935	4 382	1 655	62	10	2 160
queen15_15	1 814	1 370	5 405	2 026	63	10	2 161
queen16_16	2 193	1 446	6 576	2 426	63	10	2 167
r125.1c	2 249	2 108	7 626	2 475	68	9	2 193
r125.5	1 825	1 275	3 963	1 925	51	5	341
r250.1	704	637	1 117	709	37	1	1
r250.1c	5 951	5 339	30 477	#M#			
r250.5	6 712	5 391	15 099	7 333	51	8	2 161
r1000.1	7 204	5 470	15 378	8 198	47	12	2 528
wap05a	13 656	10 128	43 986	16 422	63	17	2 189
wap06a	13 773	10 442	44 518	16 572	63	17	2 235
wap07a	28 617	18 716	105 177	35 785	66	20	3 135
wap08a	28 885	19 395	106 046	35 664	66	19	3 127
will199GPIA	1 940	1 583	7 473	1 943	74	0	2 222

Table 12.6 – The original BFD version, along with a version that computes lower bounds on the clusters (BLB, for "Better Lower Bounds"), and, finally, a version that instead of just computing lower bounds also combine the associated partial colourings to obtain a first solution (FS). Each is presented with several limits set for the size of leaf clusters. The columns represent, from left to right: the average distance to reference bounds (Dist.); the number of times the reference bound was reached (Ref.); the number of proofs of optimality performed (Proofs); the number of times a memory out occurred (Mem.).

	Dist.	Ref.	Proofs	Mem.
BFD 90	83.4	46	39	18
BLB 90	83.3	49	42	18
FS 90	80.6	58	43	18
BFD 75	85.1	48	26	16
BLB 75	80.4	31	29	15
FS 75	65.4	42	29	14
BLB 50	80.1	16	16	13
FS 50	65.5	34	14	13

computation technique previously described. 5 % of the total time of 24 h were used to compute partial sum colourings on the clusters of the flower decomposition.

We note that the improvements brought to BFD allow it to perform a few additional proofs of optimality. This is not exactly true, however, for the versions using a limit of 50 % for the size of the leaf clusters. An hypothesis is that smaller leaves lessen the impact of bounds computed on them.

Overall, the average quality of the upper bounds returned at the end of the solution process is increased, with an even more notable gap between BLB and FS, proving how important it can be to quickly obtain a good global solution.

An important point is that the number of memory out remained very stable. This shows us that instances that cause a memory issue during lower bound computations generally cause the very same issues during the classical search process; BLB and FS thus cannot really be blamed for running into such issues themselves. Actually, in the case of FS 75, we can even see that starting the actual search with a better global upper bound may allow to avoid a few memory issues, as FS 75 was terminated 14 times instead of 16 for BFD 75.

Chapter 13

Portfolio approach

Contents

As explained in Chapter 4, a portfolio approach can be used to combine several resolution methods by building a selector that automatically chooses the most fitting approach on a per-instance basis.

The approaches already described in this thesis for the sum colouring problem appear to be highly complementary: CP models offer a robust way to find bounds on many instances, almost without any regard to their sizes; ILP models are fast and efficient, but cannot be used safely on larger instances due to their memory requirements; BFD provides methods that fit in between the pure CP and ILP approaches, by decomposing the problem, even when it is not particularly well structured.

We thus thought that automatically choosing a resolution method among these subsets of approaches would make for an interesting meta-solver. After considering seven approaches to build a portfolio, we brought this number down to five, removing some that were dominated by others and did not allow any significant improvement (or any improvement at all) in the final results.

13.1 Methods

We initially considered a total of seven methods, stemming from the experiments detailed in the previous sections:

AllDiff+SumBound+Swap `Gecode` with the conjunction of the constraints *sum* and *allDifferent*, with the same parameters as in previous experiments (see Section 10.7.2).

AllDiff+Bound+Swap Improved `Gecode` with bound consistency.

AllDiff+Bound+Swap+Dom Improved `Gecode` with domain consistency.

Gecode hybrid Improved Gecode with the hybrid restart policy and heuristic change described in Section 10.6, with domain consistency.

ILP+ Improved ILP model, as used previously.

BFD 90 BFD with a limit of 90 % on the size of leaf clusters, as used before.

BFD 75 Same as above, with a limit of 75 %.

It subsequently appeared that, in the context of such a portfolio, `Gecode` *hybrid* and BFD 75 were virtually always dominated by at least one other member of the portfolio, and did not bring any significant improvement to the overall results. Running the learning process with or without them gave the same results. These two methods were therefore removed from the portfolio, leading to a new total of five methods.

13.2 Feature extraction

Given a graph $G = (V, E)$ for which we consider looking for the chromatic sum, we compute the following features (a "(+)" mark denoting the use of the minimum value, maximum value, mean and standard deviation):

- Number of nodes $|V|$ and edges $|E|$;
- Degrees of the vertices in V; (+)
- Number of connected components in G;
- Size of the connected components in G; (+)
- Number of constraints and variables in the ILP+ model;
- Number of *allDifferent* constraints (with an arity greater than 2) in the *AllDiff+Bound* CP model;
- Arity of these *allDifferent* constraints; (+)
- Features computed from the largest connected component G' of G:
  - Density;
  - Theoretical upper and lower bounds of $\sum(G')$;
  - Number of clusters in the tree decomposition computed with *MinFill*.

Moreover, the tree decomposition just mentioned is used to compute a flower decomposition of the largest connected component (with a limit of 90 % on the sizes of leaves, as it is done for BFD 90), which gives a large array of additional features:

- Size of the root cluster;
- Cartesian product of the sizes of the domains in the root cluster;
- Number of clusters;
- Distance between the theoretical upper and lower bounds of the root cluster;
- Density of the root cluster;
- Density of leaf clusters; (+)
- Number of proper variables in clusters; (+)
- Separator density; (+)
- Separator sizes; (+)
- Distance between theoretical upper and lower bounds on leaf clusters; (+)

- Number of binary variables and constraints in the ILP+ models associated with leaf clusters. (+)

This set of features is computed in 5.4 minutes in average, with 110 of our instances actually requiring less than 5 minutes to be processed. Some instances, such as `latin_square_10`, `DSJC1000.9` or `flat1000_50_0` make this feature-extraction phase prohibitively long, mostly because of the decompositions needed.

13.3 Selection model

We use LLAMA [KOT13] to build our solver selection model. We performed a set of preliminary experiments to determine the approach that works best here, *i.e.*, a pairwise regression approach with random forest regression (see Section 4.3).

Since there are few instances, we chose to perform the learning step in a *leave-one-out* fashion (see Section 4.2): for each instance i, a new selector is trained on the 125 remaining instances in order to classify i. The definition of the selector is internally also done using the leave-one-out cross-validation approach, by performing 125 rounds. In each of these rounds, the set of the 125 remaining instances is partitioned by creating a testing set of a single instance and a training set consisting of the 124 others.

As this approach already performs well, we did not tune the parameters of the random forest machine learning algorithm. Opportunities to improve it by doing so may very well exist, and we make no claims that the particular solver selection approach we use here is the best possible.

13.4 Results

The general setup for these experiments was the same as for the previously described ones (see Section 10.7.1).

Tables 13.1 to 13.4 give detailed results for both the portfolio approach and the *virtual best solver* (VBS). The VBS is an imaginary selector always choosing the best approach for each instance, with a null decision time as well as no time requirements to compute features. Note that the "best results" are designated following these rules: making the proof of optimality is the most important aspect, then the upper bound is used to break ties, followed by the time needed to obtain it.

In these detailed tables, the five resolution methods comprising the portfolio are designated through the following abbreviated names:

G_s `Gecode` with the *sum+allDifferent* conjunction of constraints.

G_b `Gecode` with bounds consistency.

G_d Gecode with domain consistency.

C CPLEX.

B BFD 90.

These detailed results highlight the fact that the choices made by the selector are generally satisfying. Even when the chosen method differs from that of the VBS, the loss in efficiency is almost always negligible: the chosen method is not utterly inapt at providing interesting results on the considered instance.

An overview of these results is offered in Table 13.5, allowing to confirm the previously made observations.

The portfolio approach performs 65 proofs of optimality (more than one half of the benchmark), out of a total of 66 possible proofs for the considered portfolio, as demonstrated by the VBS. This number rivals with the proofs that can be made by using ILP+ (65 as well) as a systematic solver on this benchmark.

The average distance to reference solutions amounts to 5.49 for the selector. This value is strongly reminiscent of the results obtained with our CP approaches (from 5.32 to 5.44), which proved to be the best suited when trying to obtain solutions of great quality in almost every circumstances, even when instances grow larger.

Regarding the number of times the reference upper bound is reached, the selector approach displays a satisfying 77 – a number higher than the values offered by any of the five solvers comprising the portfolio (the best one among them being CPLEX, with 73 bounds reached).

When it comes to the number of times a lack of memory cut the search short, both the VBS and our selector stayed completely clear of such occurrences.

Overall, the results obtained by using our selector approach are very interesting: it successfully combines the proof-making abilities of ILP+ with the general bound quality of CP approaches, while avoiding memory problems by detecting instances that could cause such issues. BFD 90 can be employed by the portfolio as an in-between approach or to derive profit from an apparent structure overseen in an instance.

As the attentive reader may notice, these experiments do not involve the "first solution" (FS) version of our BFD approach. Actually, we developed FS later than the first portfolio approach of this thesis. We then tried, however, to integrate FS to such portfolios, but it appeared that the learning process gave FS slightly too much credit, leading to selectors that tend to use FS on very large instances. Because of this, these new portfolio approaches tend to suffer from a lack of memory on a few instances, as shown in Table 13.6. Further work would be needed to obtain better results using FS within a portfolio approach.

Table 13.1 – Detailed results of the portfolio approach, compared with the virtual best solver (part 1 of 4). "Ref." is the reference upper bound. Then, for both the VBS and the selector, "A" is the chosen algorithm, UB the upper bound found, t_{UB} the time that was needed to find it, and t_{proof} the time needed to prove its optimality if such a thing occurred. Finally, t_{fea} is the time imparted to feature computation. Times are given in seconds. Yellow cells indicate that the reference bound was equalled, while green ones highlight bounds that beat the reference ones.

		VBS				Selector				
Instance	Ref.	A	UB	t_{UB}	t_{proof}	t_{fea}	A	UB	t_{UB}	t_{proof}
DSJC125.1	326	G_s	337	29 041		0	C	347	40 376	
DSJC125.5	1 012	G_s	1 064	9 649		0	C	1 342	65 906	
DSJC125.9	2 503	G_s	2 559	35 482		1	G_d	2 566	63 964	
DSJC250.1	970	G_b	1 046	33 564		1	C	1 112	53 546	
DSJC250.5	3 210	G_d	3 540	51 076		6	G_s	3 577	9 062	
DSJC250.9	8 277	G_b	8 762	18 986		28	G_d	8 992	2 162	
DSJC500.1	2 836	G_d	3 136	11 108		10	C	4 025	42 546	
DSJC500.5	10 886	G_b	12 525	23 074		136	G_s	12 589	8 665	
DSJC500.9	29 862	G_s	32 455	26 271		563	G_s	32 455	26 835	
DSJC1000.1	8 991	G_s	10 295	66 842		126	G_s	10 295	66 968	
DSJC1000.5	37 575	G_b	44 431	47 807		2 165	G_b	44 431	49 972	
DSJC1000.9	103 445	G_b	119 347	14		8 223	G_s	119 347	8 237	
DSJR500.1	2 156	C	2 142	49 745		4	C	2 142	49 749	
DSJR500.1c	16 286	G_b	16 927	15 720		942	G_d	17 218	1 907	
DSJR500.5	25 440	G_b	28 179	2 876		34	G_d	28 315	6 830	
flat300_20_0	3 150	G_b	4 304	6 407		12	G_b	4 304	6 419	
flat300_26_0	3 966	G_b	4 843	3 536		12	G_b	4 843	3 549	
flat300_28_0	4 238	G_d	4 808	540		13	G_d	4 808	552	
flat1000_50_0	25 500	G_s	43 378	7 542		2 209	G_s	43 378	9 752	
flat1000_60_0	30 100	G_s	43 723	22 197		2 084	G_b	43 751	45 186	
flat1000_76_0	37 164	G_d	43 990	37 747		2 057	G_b	44 149	3 254	
1-FullIns_3	54	G_d	54	0	0	0	B	54	0	2
1-FullIns_4	166	C	166	3	4	0	C	166	3	4
1-FullIns_5	499	C	499	110	704	0	C	499	110	704
2-FullIns_3	93	C	93	0	0	0	C	93	0	0
2-FullIns_4	363	C	363	25	26	0	C	363	25	26
2-FullIns_5	1 433	G_s	1 433	4		15	C	1 450	85 644	
3-FullIns_3	145	C	145	1	1	0	C	145	1	1
3-FullIns_4	683	C	683	665	686	1	C	683	666	687
3-FullIns_5	3 335	G_d	3 335	49		279	G_b	3 335	334	
4-FullIns_3	205	C	205	0	0	0	C	205	0	0

Table 13.2 – Detailed results of the portfolio approach, compared with the virtual best solver (part 2 of 4). "Ref." is the reference upper bound. Then, for both the VBS and the selector, "A" is the chosen algorithm, UB the upper bound found, t_{UB} the time that was needed to find it, and t_{proof} the time needed to prove its optimality if such a thing occurred. Finally, t_{fea} is the time imparted to feature computation. Times are given in seconds. Yellow cells indicate that the reference bound was equalled, while green ones highlight bounds that beat the reference ones.

		VBS				Selector				
Instance	Ref.	A	UB	t_{UB}	t_{proof}	t_{fea}	A	UB	t_{UB}	t_{proof}
4-FullIns_4	1138	C	1138	3636	3717	5	C	1138	3641	3722
4-FullIns_5	6679	G_s	6679	1112		2079	G_s	6679	3191	
5-FullIns_3	280	C	280	2	3	0	C	280	2	3
5-FullIns_4	1776	C	1776	1218	32972	62	C	1776	1280	33034
games120	443	C	443	1	5	0	C	443	1	5
ash331GPIA	1432	C	1432	29870		20	C	1432	29890	
ash608GPIA	2600	C	2600	83844		917	C	2600	84761	
ash958GPIA	4172	C	4172	71761		961	C	4172	72722	
will199GPIA	1940	C	1940	19670		69	C	1940	19738	
1-Insertions_4	119	C	119	1	1	0	C	119	1	1
1-Insertions_5	357	C	357	88	318	0	C	357	88	318
1-Insertions_6	1068	G_d	1068	1		4	C	1070	44493	
2-Insertions_3	62	C	62	0	0	0	B	62	5	8
2-Insertions_4	249	C	249	4	8	0	C	249	4	8
2-Insertions_5	996	C	996	12096	24673	2	C	996	12098	24676
3-Insertions_3	92	C	92	0	0	0	C	92	0	0
3-Insertions_4	459	C	459	48	49	0	C	459	48	50
3-Insertions_5	2289	G_d	2289	8		274	C	2293	13682	
4-Insertions_3	127	C	127	0	1	0	C	127	0	1
4-Insertions_4	761	C	761	368	411	1	C	761	369	412
latin_square_10	41444	G_b	44641	4095		8639	G_s	44863	9389	
le450_5a	1350	C	1380	37342		3	C	1380	37345	
le450_5b	1350	C	1398	22554		4	C	1398	22558	
le450_5c	1350	C	1350	3362	14719	5	C	1350	3367	14724
le450_5d	1350	C	1350	1568	9165	5	C	1350	1573	9170
le450_15a	2632	G_b	2852	3448		5	C	2894	6181	
le450_15b	2632	C	2841	5172		5	C	2841	5177	
le450_15c	3487	G_b	4387	18734		11	G_d	4404	9861	
le450_15d	3505	G_s	4377	57066		10	G_s	4377	57076	
le450_25a	3153	C	3157	58795		10	C	3157	58805	
le450_25b	3365	C	3349	56318		6	C	3349	56324	
le450_25c	4515	G_s	4992	7640		9	C	5237	77900	

Table 13.3 – Detailed results of the portfolio approach, compared with the virtual best solver (part 3 of 4). "Ref." is the reference upper bound. Then, for both the VBS and the selector, "A" is the chosen algorithm, UB the upper bound found, t_{UB} the time that was needed to find it, and t_{proof} the time needed to prove its optimality if such a thing occurred. Finally, t_{fea} is the time imparted to feature computation. Times are given in seconds. Yellow cells indicate that the reference bound was equalled, while green ones highlight bounds that beat the reference ones.

		VBS				Selector			
Instance	Ref.	A	UB	t_{UB}	t_{proof}	t_{fea}	A	UB	t_{UB}	t_{proof}
le450_25d	4544	G_s	5042	4837		10	G_s	5042	4847	
mug88_1	178	B	178	0	3	0	C	178	0	3
mug88_25	178	C	178	0	1	0	C	178	0	1
mug100_1	202	C	202	0	3	0	C	202	0	3
mug100_25	202	C	202	0	4	0	B	202	0	5
myciel3	21	G_d	21	0	0	0	B	21	0	0
myciel4	45	G_b	45	0	0	0	C	45	0	0
myciel5	93	C	93	1	2	0	C	93	1	2
myciel6	189	C	189	18	81	0	C	189	18	81
myciel7	381	C	381	9538	12698	0	C	381	9538	12698
qg.order30	13950	G_s	13950	1	1	42	G_d	13950	44	44
qg.order40	32800	G_d	32800	5	5	314	G_d	32800	319	319
qg.order60	109800	G_d	109800	138	139	4866	G_b	109800	5070	5070
r125.1	257	B	257	0	0	0	B	257		0
r125.1c	2249	G_b	2249	1058		2	G_b	2249	1060	
r125.5	1825	C	1825	41150		0	C	1825	41150	
r250.1	704	C	704	1	1	0	C	704	1	1
r250.1c	5951	G_b	5951	314		75	G_d	5951	4314	
r250.5	6712	B	6712	86411		3	G_s	7389	760	
r1000.1	7204	B	7204	86543		464	C	7391	77866	
fpsol2.i.1	3403	B	3403	9	9	3	B	3403		11
fpsol2.i.2	1668	B	1668	8	8	2	C	1668	7	11
fpsol2.i.3	1636	B	1636	8	8	2	C	1636	6	10
inithx.i.1	3676	C	3676	13	29	10	B	3676		42
inithx.i.2	2050	C	2050	7	19	8	B	2050		29
inithx.i.3	1986	C	1986	9	20	8	B	1986	114	694
mulsol.i.1	1957	C	1957	1	1	0	C	1957	1	2
mulsol.i.2	1191	C	1191	1	1	0	C	1191	1	2
mulsol.i.3	1187	C	1187	1	1	0	C	1187	1	2
mulsol.i.4	1189	C	1189	1	1	0	C	1189	1	2
mulsol.i.5	1160	C	1160	1	1	0	C	1160	1	2

Table 13.4 – Detailed results of the portfolio approach, compared with the virtual best solver (part 4 of 4). "Ref." is the reference upper bound. Then, for both the VBS and the selector, "A" is the chosen algorithm, UB the upper bound found, t_{UB} the time that was needed to find it, and t_{proof} the time needed to prove its optimality if such a thing occurred. Finally, t_{fea} is the time imparted to feature computation. Times are given in seconds. Yellow cells indicate that the reference bound was equalled, while green ones highlight bounds that beat the reference ones.

		VBS					Selector			
Instance	Ref.	A	UB	t_{UB}	t_{proof}	t_{fea}	A	UB	t_{UB}	t_{proof}
zeroin.i.1	1822	B	1822	2	2	0	B	1822		2
zeroin.i.2	1004	C	1004	1	1	0	C	1004	1	2
zeroin.i.3	998	C	998	1	1	0	C	998	1	2
school1	2674	G_b	3531	75519		18	G_s	3648	1149	
school1_nsh	2392	B	2539	86908		10	G_s	2992	14166	
anna	276	C	276	0	0	0	C	276	0	0
david	237	C	237	0	0	0	C	237	0	0
homer	1150	C	1150	38	42	1	C	1150	39	44
huck	243	C	243	0	0	0	C	243	0	0
jean	217	C	217	0	0	0	C	217	0	0
miles250	325	B	325	0	0	0	C	325	0	0
miles500	705	C	705	5	28	0	C	705	6	29
miles750	1173	C	1173	223	303	0	C	1173	223	304
miles1000	1666	C	1666	261	262	0	C	1666	261	262
miles1500	3354	C	3354	12	13	0	B	3492	79912	
queen5_5	75	C	75	0	0	0	B	75	0	15
queen6_6	138	C	138	48	683	0	C	138	48	683
queen7_7	196	C	196	1	1	0	C	196	1	1
queen8_8	291	C	291	919	34382	0	C	291	919	34382
queen8_12	624	C	624	2	2	0	C	624	2	2
queen9_9	409	C	409	39273		0	C	409	39273	
queen10_10	553	G_b	575	4560		0	C	582	59050	
queen11_11	733	G_s	761	1035		0	C	767	14377	
queen12_12	943	G_d	986	7710		0	C	1010	81235	
queen13_13	1191	G_b	1232	65736		0	C	1270	68347	
queen14_14	1482	G_b	1533	21390		1	C	1609	67092	
queen15_15	1814	G_d	1883	3415		1	C	1932	50692	
queen16_16	2193	G_b	2269	1204		1	C	2394	36443	
wap05a	13656	C	14153	85513		93	G_b	14834	49579	
wap06a	13773	C	14494	74629		130	G_s	14923	29235	
wap07a	28617	G_s	31801	36955		1470	G_d	31902	17938	
wap08a	28885	G_d	32082	47505		1504	G_s	32094	4792	

Table 13.5 – A summary of the results obtained individually by the five methods of the portfolio, as well as by the virtual best solver (VBS) and the selector itself. From left to right, the columns give values for the average distance to reference upper bounds, the number of times these reference bounds were reached, the number of proofs of optimality, and the number of times a memory out forced the search to abort. Additionally, we give the number of times each method was employed by the VBS and by the selector.

	Dist.	Ref.	Proofs	Mem.	∈ VBS	∈ Sel.
Gecode Sum	5.34	45	8	0	15	15
Gecode Bnd	5.32	49	11	0	20	9
Gecode Dom	5.44	48	11	0	14	10
CPLEX	63.89	73	65	23	67	80
BFD 90	83.4	46	39	18	10	12
VBS	4.25	83	66	0		
Selector	5.49	77	65	0		

Table 13.6 – Results for multiple algorithm portfolios. From left to right: the previously presented portfolio that does not involve FS (solvers computing a first global solution from partial colourings); a portfolio where BFD 90 is replaced with three FS versions with different limits for the size of the leaf clusters; a similar portfolio that is just missing FS 50; and finally, a portfolio where BFD 90 is simply replaced with FS 90. We show the minimal, average and maximal distance to reference upper bounds, followed by the number of proofs of optimality, of errors due to a lack a memory. "Found best" denotes the number of times the reference bound was reached, and "Times best" the number of times the considered approach was the best among those presented in this table.

		No FS	FS 50, 75, 90	FS 75, 90	FS 90
Dist	Min	0	0	0	0
	Avg	5.5	15.3	13.7	10.8
	Max	70.1	628.5	628.5	628.5
Proofs		65	65	65	65
Mem. out		0	3	3	2
Found best		77	76	77	77
Times best		60	62	66	71

Chapter 14

Discussion

The classical colouring problem consists in finding a colouring of a given graph such that neighbour vertices do not share the same colour. In the sum colouring variant, each colour has a weight and the sum of the weights of all vertices must be minimized.

The sum colouring problem has not been extensively studied yet, and the existing complete approaches are few in number. Still, several ways to quickly model instances of this problem have been proposed:

- A constraint programming model, using variables to represent the vertices of the considered graph and values for the available colours;
- An integer linear programming model, with binary variables for each pair formed by a vertex of the graph and a colour that it might need to use;
- Boolean satisfiability models, similar in essence to the ILP model mentioned above.

Alternatively, a branch and bound approach can be employed directly on the problem.

This part was mostly concerned with the existing approaches aiming at solving the sum colouring problem to optimality. We described several ways to improve these approaches and developed a new one, which in turn gets improved as well.

A summary of our work on these topics lies hereafter, together with thoughts on the perspectives it offers.

Results

We put forward several improvements to help solving the sum colouring problem with CP and ILP, and demonstrated that they have complementary advantages:

- ILP is very efficient on small instances, but fails to solve larger instances due to its large memory needs;
- CP never runs out of memory on our benchmark, even on the largest instances, but it is seldom able to compute optimal solutions as fast as ILP approaches do on smaller instances.

Alternatively, we proposed a combination of CP and ILP (BFD) that may serve as a compromise between the approaches consisting in using only one of those two techniques. Besides, this combination employs a tree decomposition,

which makes it more suited than CP or ILP used on their own to solve numerous well-structured instances. This new approach gave encouraging results.

BFD appeared to suffer from an inability to quickly detect situations where it is led to solve unpromising subproblems to optimality. Therefore, we focused on this aspect of the method, and designed ways to compute bounds helping with this task, since a good way to detect useless subproblems is to obtain better bounds, either on the different clusters or on the problem itself.

In this regard, we investigated ways to obtain such bounds early during the search. We implemented a preprocessing step consisting in computing local solutions that give local bounds on the different clusters. These partial colourings can then easily be combined to form a first global solution, yielding an upper bound clearly surpassing the simple theoretical ones. All these informations allowed us to obtain improved results for approaches based on BFD.

We combined several of the approaches presented in this part of the thesis into a portfolio approach comprised of five methods. The resulting selector obtains results close to those of the virtual best solver for this array of algorithms. It has been able to prove optimality for more than one half of the considered instances. In the course of our experiments, it has also been able to improve the best known upper bounds for two instances.

Perspectives

Sum colouring studies being still quite rare, there is still ample room for improvements. The use of a dedicated decomposition algorithm, notably, might be an interesting research subject. Building a flower decomposition from scratch rather than resorting to an initial tree decomposition would make it more straightforward to obtain a balanced decomposition. Being able to automatically fine-tune BFD's parameters (the size limit enforced on leaf clusters, as well as the maximal size of separators) could also prove useful.

A major drawback of BFD is that some subproblems are solved to optimality even if they are rendered utterly useless by poor assignments in the root making it impossible to find interesting solutions in the current context. Preventing the ILP solver from spending more that a set amount of time on a leaf cluster could alleviate the associated loss of performances. Should such a time limit be reached, the solver would ask for a new assignment on the root. It can be seen as another form of restarts, as seen in [ALL+15].

Due to a lack of time, we could only perform a few preliminary experiment regarding the chromatic strength and its capability to tighten the domains of variables. It appeared that the domain reductions brought by the currently existing tools often had a negligible impact due to the fact that the search process generally naturally induce comparable reductions. However, we still think that chromatic strength might hold some potential to ease the solution of sum colouring instances.

Conclusion

With their high declarative capabilities and relative ease of use, constraint programming and integer linear programming, among other approaches, hold promise in the realm of challenging $\mathcal{NP}$-hard optimisation problems. However, even when they are competitive with other state-of-the-art solution methods, they still often fail to provide truly satisfying results on a systematic basis for some problems.

Such particularly challenging problems include the maximum common subgraph problem as well as the sum colouring problem. To try to alleviate this difficulty to some extent, one can invest some computational time into a decomposition method in order to reduce the size of the search space, possibly also enabling a resolution in parallel. Additionally, a thoughtfully enacted decomposition can provide a better grasp of the inherent structure of an instance, thus allowing a more enlightened exploration of the search space while also deriving profit from the independence of some parts of the problems.

In this context, this thesis aimed at providing clues to help solving both the maximum common induced subgraph problem and the sum colouring problem, mainly by using decomposition approaches, but also by honing existing tools such as constraint programming and integer linear programming in order to make them more efficient and attractive.

The maximum common subgraph problem

To make solving the MCIS problem easier, we designed a decomposition method, denoted throughout this thesis by STR. It is based on the structure of the instance and stems from the method known in the literature as TR-decomposition.

When we evaluated STR, it appeared quite plainly that, as one could expect, allotting more computational time to the decomposition step can lead to a greater reduction of the size of the search space. Indeed, the reduction offered by STR is orders of magnitude more important than that provided by the state-of-the-art decomposition method called embarrassingly parallel search (EPS). However, the time used for an STR decomposition amounts on average to 5 % of the total solution time, as opposed to the proportions of far less than 0.1 % used by EPS.

In the course of our study, we investigated the impact of a variation of the number of processing units available to solve the subproblems generated by

the considered decomposition methods. It appeared that every method did not fare as well in each situation. On our benchmark, when generating less than 35 subproblems per available worker, STR showed a higher efficiency in terms of speedup. On the other hand, when using a larger number of subproblems per worker, EPS will generally be better suited.

To our mild disappointment, the overall conclusion of this part was that, even with high levels of fine-tuning, the speedups offered by every considered method remained rather low. Further work will be needed for instances of more than a hundred vertices to be systematically solved within a reasonable time.

The sum colouring problem

The sum colouring problem is a more complex variant of the classical colouring problem. Only a few complete approaches have been proposed to date, and the results they offer are generally unsatisfying.

Basic constraint programming and integer linear programming models have been suggested in the literature. We evaluated them, and researched into ways to make them more practical: reducing the number of considered colours, computing bounds and using global constraints, for example, proved to be interesting options.

Additionally, our experiments revealed quite clearly that CP and ILP had complementary strong points when confronted with sum colouring instances. While ILP solvers are fast to find good solutions and to prove their optimality, they often suffer from a lack of memory, especially on larger instances. Conversely, CP approaches allowed to find interesting solutions even on the largest instances of our benchmark without issues, but generally need more time on smaller instances to find solutions on a par with those obtained by ILP solvers.

Alternatively, we proposed a combination of CP and ILP, called BFD, that may serve as a compromise between the approaches consisting in using only one of those two techniques. Besides, this combination employs a tree decomposition, which makes it more suited than CP or ILP used on their own to solve numerous well-structured instances.

We combined CP and ILP approaches, as well as BFD, in a portfolio approach. The resulting selector obtains results close to those of the virtual best solver for this array of algorithms. It has been able to prove optimality for more than half of the considered instances, while keeping the average quality of solutions very high.

In the course of our experiments, our approaches could improve the best known upper bounds for two instances. Moreover, both BFD and the improved ILP model were able to perform a few proofs of optimality that had not been made in the literature, for example on `2-Insertions_3` or `mug100_25`.

BFD appears to suffer from an inability to quickly detect situations where it is led to solve unpromising subproblems to optimality. Therefore, we focused

on this aspect of the method, and designed ways to compute bounds helping with this task, since a good way to detect useless subproblems is to obtain better bounds, either on the different clusters or on the problem itself.

In this regard, we investigated ways to obtain such bounds early during the search. We implemented a preprocessing step consisting in computing partial solutions that give local bounds on the different clusters. These partial colourings can then easily be combined to form a first global solution, yielding an upper bound clearly surpassing the simple theoretical ones. All these informations allowed us to obtain improved results for approaches based on BFD.

Since sum colouring studies are still infrequent, it seems clear that there are still many ways to improve the different methods or to create brand new ones. As for BFD itself, we assuredly think that it has still a lot to offer. The use of a dedicated decomposition algorithm, in particular, might be an interesting research subject: directly building a flower decomposition instead of using an intermediate tree decomposition would surely help obtaining more balanced leaf clusters. Being able to automatically choose a value for the limit set on the size of leaf clusters according to features extracted from the instance could also prove useful.

As stated previously, a major drawback of BFD is that some subproblems are solved to optimality even if they are rendered utterly useless by poor assignments in the root making it impossible to find interesting solutions in the current context. Preventing the ILP solver from spending more that a set amount of time on a leaf cluster could alleviate the associated loss of performances. Should such a time limit be reached, the solver would ask for a new assignment on the root. It can be seen as another form of restarts, as seen in [ALL+15].

Due to a lack of time, we could only perform a few preliminary experiments regarding the chromatic strength and its capability to tighten the domains of variables. Interesting works have been recently published on this particular topic, and the resulting tools might lead to significant reductions of this problem's search space if used properly, both for CP and ILP models as well as for approaches such as BFD.

Bibliography

[All+15] David Allouche et al. 'Anytime Hybrid Best-First Search with Tree Decomposition for Weighted CSP'. In: *Proceedings of the 21st International Conference on Principles and Practice of Constraint Programming - Volume 9255*. 2015, pp. 12–29.

[Bak94] Andrew B Baker. 'The hazards of fancy backtracking'. In: *AAAI*. Vol. 94. 1994, p. 288.

[BB76] H. G. Barrow and R. M. Burstall. 'Subgraph isomorphism, matching relational structures and maximal cliques'. In: *Information Processing Letters* 4.4 (1976), pp. 83–84.

[Bel+12] Nicolas Beldiceanu et al. 'An O (nlog n) Bound Consistency Algorithm for the Conjunction of an alldifferent and an Inequality between a Sum of Variables and a Constant, and its Generalization.' In: *ECAI*. Vol. 12. 2012, pp. 145–150.

[Ber73] Claude Berge. 'Graphes et hypergraphes'. In: (1973).

[Bes+05] Christian Bessière et al. 'An optimal coarse-grained arc consistency algorithm'. In: *Artificial Intelligence* 165.2 (2005), pp. 165–185.

[Bes+99] Christian Bessière et al. 'On forward checking for non-binary constraint satisfaction'. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 1999, pp. 88–102.

[Bes94] Christian Bessiere. 'Arc-consistency and arc-consistency again'. In: *Artificial intelligence* 65.1 (1994), pp. 179–190.

[BH12] Una Benlic and Jin-Kao Hao. 'A study of breakout local search for the minimum sum coloring problem'. In: *Simulated Evolution and Learning*. Springer, 2012, pp. 128–137.

[Bod96] Hans L Bodlaender. 'A linear-time algorithm for finding tree-decompositions of small treewidth'. In: *SIAM Journal on computing* 25.6 (1996), pp. 1305–1317.

[Bou+04] Frédéric Boussemart et al. 'Boosting systematic search by weighting constraints'. In: *ECAI*. Vol. 16. 2004, p. 146.

[BR01] Christian Bessière and Jean-Charles Régin. 'Refining the Basic Constraint Propagation Algorithm.' In: *IJCAI*. Vol. 1. 2001, pp. 309–315.

[BR96] Christian Bessiere and Jean-Charles Régin. 'MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems'. In: *Principles and Practice of Constraint Programming—CP96*. Springer. 1996, pp. 61–75.

[BV09] Flavia Bonomo and Mario Valencia-Pabon. 'Minimum Sum Coloring of P4-sparse graphs'. In: *Electronic Notes in Discrete Mathematics* 35 (2009), pp. 293–298.

[BV14] Flavia Bonomo and Mario Valencia-Pabon. 'On the minimum sum coloring of p 4-sparse graphs'. In: *Graphs and Combinatorics* 30.2 (2014), pp. 303–314.

[BY86] Egon Balas and Chang Sung Yu. 'Finding a maximum clique in an arbitrary graph'. In: *SIAM Journal on Computing* 15.4 (1986), pp. 1054–1068.

[CFV07] Donatello Conte, Pasquale Foggia and Mario Vento. 'Challenging Complexity of Maximum Common Subgraph Detection Algorithms: A Performance Analysis of Three Algorithms on a Wide Database of Graphs.' In: *J. Graph Algorithms Appl.* 11.1 (2007), pp. 99–143.

[CJK03] Assef Chmeiss, Philippe Jégou and Lamia Keddar. 'On a generalization of triangulated graphs for domains decomposition of CSPs'. In: *IJCAI*. Citeseer. 2003, pp. 203–208.

[Cos94] Marie-Christine Costa. 'Persistency in maximum cardinality bipartite matchings'. In: *Operations Research Letters* 15.3 (1994), pp. 143–149.

[CPL05] ILOG CPLEX. *High-performance software for mathematical programming and optimization*. 2005.

[De +05] Simon De Givry et al. 'Existential arc consistency: Getting closer to full arc consistency in weighted CSPs'. In: *IJCAI*. Vol. 5. 2005, pp. 84–89.

[Dec90] Rina Dechter. 'Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition'. In: *Artificial Intelligence* 41.3 (1990), pp. 273–312.

[Dec92] Rina Dechter. *Constraint networks*. Information and Computer Science, University of California, Irvine, 1992.

[Dep+13] Matjaz Depolli et al. 'Exact parallel maximum clique algorithm for general and protein graphs'. In: *Journal of chemical information and modeling* 53.9 (2013), pp. 2217–2228.

[Dir61] Gabriel Andrew Dirac. 'On rigid circuit graphs'. In: *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*. Vol. 25. 1. Springer. 1961, pp. 71–76.

[DM94] Rina Dechter and Itay Meiri. 'Experimental evaluation of preprocessing algorithms for constraint satisfaction problems'. In: *Artificial Intelligence* 68.2 (1994), pp. 211–241.

[DP89] Rina Dechter and Judea Pearl. 'Tree clustering for constraint networks'. In: *Artificial Intelligence* 38.3 (1989), pp. 353–366.

[DSV06] Simon De Givry, Thomas Schiex and Gerard Verfaillie. 'Exploiting tree decomposition and soft local consistency in weighted CSP'. In: *AAAI*. Vol. 6. 2006, pp. 1–6.

[Dur+99] Paul J Durand et al. 'An efficient algorithm for similarity analysis of molecules'. In: *Internet Journal of Chemistry* 2.17 (1999), pp. 1–16.

[F+65] Delbert R Fulkerson, Oliver A Gross et al. 'Incidence matrices and interval graphs'. In: *Pacific J. Math* 15.3 (1965), pp. 835–855.

[Fre85] Eugene C Freuder. 'A sufficient condition for backtrack-bounded search'. In: *Journal of the ACM (JACM)* 32.4 (1985), pp. 755–761.

[Fre95] Eugene C Freuder. 'Using inference to reduce arc consistency computation'. In: *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI'95, pages 592–598. Morgan Kaufmann Publishers, Inc, 1995. 23, 28, 33*. 1995.

[G+98] Carla P Gomes, Bart Selman, Henry Kautz et al. 'Boosting combinatorial search through randomization'. In: *AAAI/IAAI* 98 (1998), pp. 431–437.

[Gas79] John Gaschig. *Performance measurement and analysis of certain search algorithms*. Tech. rep. CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, 1979.

[GJ02] Michael R Garey and David S Johnson. *Computers and intractability*. Vol. 29. wh freeman New York, 2002.

[GJ79] Michael R Garey and David S Johnson. 'A Guide to the Theory of NP-Completeness'. In: *WH Freemann, New York* 70 (1979).

[GJC94] Marc Gyssens, Peter G Jeavons and David A Cohen. 'Decomposing constraint satisfaction problems using database techniques'. In: *Artificial intelligence* 66.1 (1994), pp. 57–89.

[GLS00] G. Gottlob, N. Leone and F. Scarcello. 'A Comparison of Structural CSP Decomposition Methods'. In: *Artificial Intelligence* 124 (2000), pp. 343–282.

[GLS99] Georg Gottlob, Nicola Leone and Francesco Scarcello. 'Hypertree decompositions and tractable queries'. In: *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM. 1999, pp. 21–32.

[Gol80] M.C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press. New-York, 1980.

[Gom60] Ralph Gomory. *An algorithm for the mixed integer problem*. Tech. rep. RAND CORP SANTA MONICA CA, 1960.

[GP82] Marc Gyssens and Jan Paredaens. 'A Decomposition Methodology for Cyclic Databases.' In: *Advances in data base theory* 2 (1982), pp. 85–122.

[GS01] Carla P. Gomes and Bart Selman. 'Algorithm Portfolios'. In: *Artificial Intelligence* 126.1-2 (2001), pp. 43–62.

[GW94] Ian P Gent and Toby Walsh. 'Easy problems are sometimes hard'. In: *Artificial Intelligence* 70.1-2 (1994), pp. 335–345.

[HC11] Anders Helmar and Marco Chiarandini. 'A local search heuristic for chromatic sum'. In: *Proceedings of the 9th metaheuristics international conference*. Vol. 1101. 2011, pp. 161–170.

[HE80] Robert M Haralick and Gordon L Elliott. 'Increasing tree search efficiency for constraint satisfaction problems'. In: *Artificial intelligence* 14.3 (1980), pp. 263–313.

[HHR03] Chris Harrelson, Kirsten Hildrum and Satish Rao. 'A polynomial-time tree decomposition to minimize congestion'. In: *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*. ACM. 2003, pp. 34–43.

[HLH97] Bernardo A. Huberman, Rajan M. Lukose and Tad Hogg. 'An Economics Approach to Hard Computational Problems'. In: *Science* 275.5296 (1997), pp. 51–54.

[HMR17] Ruth Hoffmann, Ciaran McCreesh and Craig Reilly. 'Between Subgraph Isomorphism and Maximum Common Subgraph'. In: (2017).

[Jég93] Philippe Jégou. 'Decomposition of domains based on the micro-structure of finite constraint-satisfaction problems'. In: *AAAI*. Vol. 93. 1993, pp. 731–736.

[JH16] Yan Jin and Jin-Kao Hao. 'Hybrid evolutionary search for the minimum sum coloring problem of graphs'. In: *Information Sciences* 352 (2016), pp. 15–34.

[JHH14] Yan Jin, Jin-Kao Hao and Jean-Philippe Hamiez. 'A memetic algorithm for the minimum sum coloring problem'. In: *Computers & Operations Research* 43 (2014), pp. 318–327.

[JHH16] Yan Jin, Jean-Philippe Hamiez and Jin-Kao Hao. 'Algorithms for the minimum sum coloring problem: a review'. In: *Artificial Intelligence Review* (2016), pp. 1–28.

[JKT15] Philippe Jégou, Hanan Kanso and Cyril Terrioux. 'An Algorithmic Framework for Decomposing Constraint Networks'. In: *Tools with Artificial Intelligence (ICTAI), 2015 IEEE 27th International Conference on*. IEEE. 2015, pp. 1–8.

[JNT08] Philippe Jégou, Samba Ndojh Ndiaye and Cyril Terrioux. 'A new Evaluation of Forward Checking and its Consequences on Efficiency of Tools for Decomposition of CSPs'. In: *Tools with Artificial Intelligence, 2008. ICTAI'08. 20th IEEE International Conference on*. Vol. 1. IEEE. 2008, pp. 486–490.

[JT03] Philippe Jégou and Cyril Terrioux. 'Hybrid backtracking bounded by tree-decomposition of constraint networks'. In: *Artificial Intelligence* 146.1 (2003), pp. 43–75.

[Kad+10] Serdar Kadioglu et al. 'ISAC-Instance-Specific Algorithm Configuration.' In: *ECAI*. Vol. 215. 2010, pp. 751–756.

[Kar84] Narendra Karmarkar. 'A new polynomial-time algorithm for linear programming'. In: *Proceedings of the sixteenth annual ACM symposium on Theory of computing*. ACM. 1984, pp. 302–311.

[Kja90] Uffe Kjaerulff. *Triangulation of Graphs - Algorithms Giving Small Total State Space*. Tech. rep. Judex R.R. Aalborg., Denmark, 1990.

[KM72] V Klee and GJ Minty. *How good is the simplex algorithm? Inequalities III.(0. SHISHA, Ed.) pp. 159-175*. 1972.

[KMS16] Lars Kotthoff, Ciaran Mccreesh and Christine Solnon. 'Portfolios of Subgraph Isomorphism Algorithms'. In: *Learning and Intelligent OptimizatioN Conference (LION 10)*. Springer. 2016.

[Kot+15] Lars Kotthoff et al. 'Improving the State of the Art in Inexact TSP Solving using Per-Instance Algorithm Selection'. In: *LION 9*. 2015.

[Kot13] Lars Kotthoff. 'LLAMA: Leveraging Learning to Automatically Manage Algorithms'. In: *CoRR* abs/1306.1031 (2013). URL: http://arxiv.org/abs/1306.1031.

[Kot14] Lars Kotthoff. 'Algorithm Selection for Combinatorial Search Problems: A Survey'. In: *AI Magazine* 35.3 (2014), pp. 48–60.

[KS89] Ewa Kubicka and Allen J Schwenk. 'An introduction to chromatic sums'. In: *Proceedings of the 17th conference on ACM Annual Computer Science Conference*. ACM. 1989, pp. 39–45.

[Kub04] Marek Kubale. *Graph colorings*. Vol. 352. American Mathematical Soc., 2004.

[Lar02] Javier Larrosa. 'Node and arc consistency in weighted CSP'. In: *AAAI/IAAI*. 2002, pp. 48–53.

[LB62] C Lekkeikerker and J Boland. 'Representation of a finite graph by a set of intervals on the real line'. In: *Fundamenta Mathematicae* 51.1 (1962), pp. 45–64.

[Lec+07a] Christophe Lecoutre et al. 'Nogood Recording from Restarts.' In: *IJCAI*. Vol. 7. 2007, pp. 131–136.

[Lec+07b] Christophe Lecoutre et al. 'Recording and minimizing nogoods from restarts'. In: *Journal on Satisfiability, Boolean Modeling and Computation* 1 (2007), pp. 147–167.

[Lec+15a] Clément Lecat et al. 'Comparaison de méthodes de résolution pour le probleme de somme coloration'. In: *JFPC'15: Journées Francophones de Programmation par Contraintes*. 2015.

[Lec+15b] Clément Lecat et al. 'Exact methods for the minimum sum coloring problem'. In: *DPCP-2015*. Cork, Ireland, Iran, 2015, pp. 61–69. URL: https://hal.archives-ouvertes.fr/hal-01323741.

[LLL16] Clément Lecat, Corinne Lucet and Chu-Min Li. 'Sum Coloring: New upper bounds for the chromatic strength'. In: (2016).

[LLL17] Clément Lecat, Corinne Lucet and Chu-Min Li. 'New Lower Bound for the Minimum Sum Coloring Problem.' In: *AAAI*. 2017, pp. 853–859.

[LM98] Javier Larrosa and Pedro Meseguer. 'Adding constraint projections in n-ary csp'. In: *Proceedings of the ECAI*. Vol. 98. 1998, pp. 41–48.

[LS03] Javier Larrosa and Thomas Schiex. 'In the quest of the best form of local consistency for weighted CSP'. In: *IJCAI*. Vol. 3. 2003, pp. 239–244.

[LSZ93] Michael Luby, Alistair Sinclair and David Zuckerman. 'Optimal speedup of Las Vegas algorithms'. In: *Information Processing Letters* 47.4 (1993), pp. 173–180.

[Lue74] GS Lueker. 'Structured breadth first search and chordal graphs'. In: *Princeton Univ. Tech. Rep. TR-158* (1974).

[Mac77] Alan K Mackworth. 'Consistency in networks of relations'. In: *Artificial intelligence* 8.1 (1977), pp. 99–118.

[McC+16] Ciaran McCreesh et al. 'Clique and constraint models for maximum common (connected) subgraph problems'. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2016, pp. 350–368.

[McG82] James J McGregor. 'Backtrack search algorithms and the maximal common subgraph problem'. In: *Software: Practice and Experience* 12.1 (1982), pp. 23–34.

[MH86] Roger Mohr and Thomas C Henderson. 'Arc and path consistency revisited'. In: *Artificial intelligence* 28.2 (1986), pp. 225–233.

[MN14] Maël Minot and Samba Ndojh Ndiaye. 'Searching for a maximum common induced subgraph by decomposing the compatibility graph'. en. In: *Bridging the Gap Between Theory and Practice in Constraint Solvers, CP2014-Workshop*. Sept. 2014, pp. 1–17. URL: http://liris.cnrs.fr/publis/?id=6911.

[MNS15a] Maël Minot, Samba Ndojh Ndiaye and Christine Solnon. 'A Comparison of Decomposition Methods for the Maximum Common Subgraph Problem'. In: *Tools with Artificial Intelligence (ICTAI), 2015 IEEE 27th International Conference on*. IEEE. 2015, pp. 461–468.

[MNS15b] Maël Minot, Samba Ndojh Ndiaye and Christine Solnon. 'Recherche d'un plus grand sous-graphe commun par décomposition du graphe de compatibilité'. In: *Onzièmes Journées Francophones de Programmation par Contraintes (JFPC)*. 2015.

[MNS16a] Maël Minot, Samba Ndojh Ndiaye and Christine Solnon. 'An evaluation of complete approaches for the sum colouring problem'. In: *Douzièmes Journées Francophones de Programmation par Contraintes (JFPC 2016)*. Montpellier, France, June 2016. URL: https://hal.archives-ouvertes.fr/hal-01309350.

[MNS16b] Maël Minot, Samba Ndojh Ndiaye and Christine Solnon. 'Using CP and ILP with tree decomposition to solve the sum colouring problem'. In: *Doctoral program of CP 2016*. Toulouse, France, Sept. 2016. URL: https://hal.archives-ouvertes.fr/hal-01366291.

[MNS17] Maël Minot, Samba Ndojh Ndiaye and Christine Solnon. 'Combining CP and ILP in a tree decomposition of bounded height for the sum colouring problem'. In: *CPAIOR 2017*. Padova, Italy, June 2017. URL: https://hal.archives-ouvertes.fr/hal-01447818.

[Mon74] Ugo Montanari. 'Networks of constraints: Fundamental properties and applications to picture processing'. In: *Information sciences* 7 (1974), pp. 95–132.

[Mos+01] Matthew W Moskewicz et al. 'Chaff: Engineering an efficient SAT solver'. In: *Proceedings of the 38th annual Design Automation Conference*. ACM. 2001, pp. 530–535.

[Mou+] A Moukrim et al. *Upper and lower bounds for the minimum sum coloring problem, submitted for publication*.

[Mou+10] Aziz Moukrim et al. 'Lower bounds for the minimal sum coloring problem'. In: *Electronic Notes in Discrete Mathematics* 36 (2010), pp. 663–670.

[MP13] Ciaran McCreesh and Patrick Prosser. 'Multi-threading a state-of-the-art maximum clique algorithm'. In: *Algorithms* 6.4 (2013), pp. 618–635.

[MP14] Ciaran McCreesh and Patrick Prosser. 'The Shape of the Search Tree for the Maximum Clique Problem, and the Implications for Parallel Branch and Bound'. In: *arXiv preprint arXiv:1401.5921* (2014).

[NS11] Samba Ndojh Ndiaye and Christine Solnon. 'CP models for maximum common subgraph problems'. In: *Principles and Practice of Constraint Programming–CP 2011*. Springer, 2011, pp. 637–644.

[OMa+08] Eoin O'Mahony et al. 'Using Case-based Reasoning in an Algorithm Portfolio for Constraint Solving'. In: *Proceedings of the 19th Irish Conference on Artificial Intelligence and Cognitive Science*. Jan. 2008.

[PRB01] Thierry Petit, Jean-Charles Régin and Christian Bessière. 'Specific filtering algorithms for over-constrained problems'. In: *Principles and Practice of Constraint Programming – CP 2001*. Springer. 2001, pp. 451–463.

[Pri57] Robert Clay Prim. 'Shortest connection networks and some generalizations'. In: *Bell Labs Technical Journal* 36.6 (1957), pp. 1389–1401.

[Pro93] Patrick Prosser. 'Hybrid algorithms for the constraint satisfaction problem'. In: *Computational intelligence* 9.3 (1993), pp. 268–299.

[Pug98] Jean-Francois Puget. 'A fast algorithm for the bound consistency of alldiff constraints'. In: *Aaai/Iaai*. 1998, pp. 359–366.

[Pur83] Paul Walton Purdom. 'Search rearrangement backtracking and polynomial average time'. In: *Artificial intelligence* 21.1-2 (1983), pp. 117–133.

[Rég94] Jean-Charles Régin. 'A filtering algorithm for constraints of difference in CSPs'. In: *AAAI*. Vol. 94. 1994, pp. 362–367.

[RGW02] J W Raymond, E J Gardiner and P Willett. 'RASCAL: calculation of graph similarity using maximum common edge subgraphs'. In: *The Computer Journal* 45.6 (2002), pp. 631–644.

[Ric76] John R. Rice. 'The Algorithm Selection Problem'. In: *Advances in Computers* 15 (1976), pp. 65–118.

[RRM13] Jean-Charles Régin, Mohamed Rezgui and Arnaud Malapert. 'Embarrassingly Parallel Search'. In: *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*. 2013, pp. 596–610. URL: http://dx.doi.org/10.1007/978-3-642-40627-0_45.

[RRM14] Jean-Charles Régin, Mohamed Rezgui and Arnaud Malapert. 'Improvement of the Embarrassingly Parallel Search for Data Centers'. In: *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*. 2014, pp. 622–635. URL: http://dx.doi.org/10.1007/978-3-319-10428-7_45.

[RS86] N. Robertson and P.D. Seymour. 'Graph minors II: Algorithmic aspects of tree-width'. In: *Algorithms* 7 (1986), pp. 309–322.

[RTL76] D. Rose, R. Tarjan and G. Lueker. 'Algorithmic Aspects of Vertex Elimination on Graphs.' In: *SIAM J. Comput.* 5.2 (1976), pp. 266–283.

[RW02] John W Raymond and Peter Willett. 'Maximum common subgraph isomorphism algorithms for the matching of chemical structures'. In: *Journal of computer-aided molecular design* 16.7 (2002), pp. 521–533.

[S+93] Bart Selman, Henry A Kautz, Bram Cohen et al. 'Local search strategies for satisfiability testing.' In: *Cliques, coloring, and satisfiability* 26 (1993), pp. 521–532.

[San+09] Marti Sanchez et al. 'Russian Doll Search with Tree Decomposition.' In: *IJCAI*. 2009, pp. 603–608.

[Sei+12] Jendrik Seipp et al. 'Learning Portfolios of Automatically Tuned Planners'. In: *ICAPS*. 2012.

[SF06] Christine Solnon and Serge Fenet. 'A study of ACO capabilities for solving the maximum clique problem'. In: *J. Heuristics* 12.3 (2006), pp. 155–180. URL: http://dx.doi.org/10.1007/s10732-006-4295-8.

[SF94] Daniel Sabin and Eugene C Freuder. 'Contradicting conventional wisdom in constraint satisfaction'. In: *Principles and Practice of Constraint Programming*. Springer. 1994, pp. 10–20.

[SG97] Barbara M Smith and Stuart A Grant. 'Trying harder to fail first'. In: *Research Report Series-University of Leeds School of Computer Studies Lu Scs Rr* (1997).

[SS77] Richard M Stallman and Gerald J Sussman. 'Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis'. In: *Artificial intelligence* 9.2 (1977), pp. 135–196.

[SV94] Thomas Schiex and Gérard Verfaillie. 'Nogood recording for static and dynamic constraint satisfaction problems'. In: *International Journal on Artificial Intelligence Tools* 3.02 (1994), pp. 187–207.

[Tea08] Gecode Team. *Gecode: Generic constraint development environment, 2006*. 2008.

[Tho+89] Carsten Thomassen et al. 'Tight bounds on the chromatic sum of a connected graph'. In: *Journal of Graph Theory* 13.3 (1989), pp. 353–357.

[Ull76] Julian R Ullmann. 'An algorithm for subgraph isomorphism'. In: *Journal of the ACM (JACM)* 23.1 (1976), pp. 31–42.

[Vis11] Philippe Vismara. 'Programmation par contraintes pour les problemes de plus grand sous-graphe commun'. In: *JFPC'11: Journées Francophones de Programmation par Contraintes*. 2011, pp. 327–335.

[Wal75] David Waltz. *Understanding Line Drawings of Scenes with Shadows." The Psychology of Computer Vision. Patrick Henry Winston, ed*. 1975.

[Wan+12] Yang Wang et al. 'Solving the minimum sum coloring problem via binary quadratic programming'. In: *Optimization* (2012).

[WH12] Qinghua Wu and Jin-Kao Hao. 'An effective heuristic algorithm for sum coloring of graphs'. In: *Computers & Operations Research* 39.7 (2012), pp. 1593–1600.

[WH13] Qinghua Wu and Jin-Kao Hao. 'Improved lower bounds for sum coloring via clique decomposition'. In: *arXiv preprint 1303.6761* (2013).

[XHL10] Lin Xu, Holger Hoos and Kevin Leyton-Brown. 'Hydra: Automatically Configuring Algorithms for Portfolio-Based Selection.' In: *AAAI*. Vol. 10. 2010, pp. 210–216.

[Xu+08] Lin Xu et al. 'SATzilla: Portfolio-based Algorithm Selection for SAT'. In: *J. Artif. Intell. Res. (JAIR)* 32 (2008), pp. 565–606.

[ZY01] Yuanlin Zhang and Roland HC Yap. 'Making AC-3 an optimal algorithm'. In: *IJCAI*. Vol. 1. 2001, pp. 316–321.

FOLIO ADMINISTRATIF

THÈSE DE L'UNIVERSITÉ DE LYON OPÉRÉE AU SEIN DE L'INSA LYON

NOM : Minot **DATE DE SOUTENANCE :** 19 décembre 2017
PRÉNOM : Maël

TITRE : Investigating decomposition methods for the maximum common subgraph and sum colouring problems

NATURE : Doctorat **NUMÉRO D'ORDRE :** 2017LYSEI120
ÉCOLE DOCTORALE : InfoMaths
SPÉCIALITÉ : Informatique

RÉSUMÉ :

Cette thèse vise, d'un point de vue général, à concevoir et évaluer des méthodes de décomposition applicables à des problèmes d'optimisation sous contraintes, ainsi qu'à rendre ces méthodes le plus compétitives possible. Deux problèmes d'optimisation connus pour leur difficulté on été considérés en particulier : le problème du plus grand sous-graphe commun, qui consiste à trouver la plus grande partie commune entre deux objets, et le problème de la somme coloration, dans lequel un graphe doit être colorié d'une façon minimisant une somme de poids. Nous employons notamment, en plus de la programmation linéaire, des techniques de décomposition arborescentes et la programmation par contrainte, et abordons également la sélection automatique d'algorithmes.

MOTS-CLEFS : Décomposition · Graphes · Isomorphisme · Coloration · Programmation par contraintes · Programmation linéaire · Optimisation combinatoire · Portfolio

LABORATOIRE DE RECHERCHE : LIRIS

DIRECTRICE DE THÈSE : Christine SOLNON
PRÉSIDENT DE JURY : Christophe LECOUTRE
COMPOSITION DU JURY :
Sambda Ndojh NDIAYE
Simon DE GIVRY
Chu Min LI