Exploitation de structures de graphe en programmation par contraintes

Jean-Guillaume Fages

▶ **To cite this version:**

Jean-Guillaume Fages. Exploitation de structures de graphe en programmation par contraintes. Algorithme et structure de données [cs.DS]. Ecole des Mines de Nantes, 2014. Français. NNT : 2014EMNA0190 . tel-01085253

HAL Id: tel-01085253

https://theses.hal.science/tel-01085253

Submitted on 21 Nov 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

l'université nantes angers le mans
PÔLE DE RECHERCHE ET D'ENSEIGNEMENT SUPÉRIEUR

MINES Nantes
Institut Mines-Télécom

Thèse de Doctorat

Jean-Guillaume FAGES

Mémoire présenté en vue de l'obtention du
grade de Docteur de l'École nationale supérieure des mines de Nantes
sous le label de l'Université de Nantes Angers Le Mans

École doctorale : Sciences et technologies de l'information, et mathématiques

Discipline : Informatique et applications, section CNU 27
Unité de recherche : Laboratoire d'informatique de Nantes-Atlantique (LINA)

Soutenue le 23 octobre 2014
Thèse n° : 2014EMNA0190

Exploitation de structures de graphe en programmation par contraintes

JURY

Président :	**M. Narendra JUSSIEN**, Directeur, Télécom Lille
Rapporteurs :	**Mme Christine SOLNON**, Professeur, Institut National des Sciences Appliquées de Lyon
	M. Jean-Charles RÉGIN, Professeur, Université de Nice-Sophia Antipolis
Examinateurs :	**M. Christophe LECOUTRE**, Professeur, Université d'Artois
	M. Benoît ROTTEMBOURG, Directeur Associé, EURODECISION
Directeur de thèse :	**M. Nicolas BELDICEANU**, Professeur, Ecole des Mines de Nantes
Co-directeur de thèse :	**M. Xavier LORCA**, Maître-Assistant, Ecole des Mines de Nantes

A Nougatine,

Table des matières

1 Remerciements **9**

2 Introduction **11**

I Etat de l'art 15

3 Théorie des Graphes **17**
- 3.1 Définitions et notations de base 17
- 3.2 Concepts avancés 20

4 Programmation par contraintes **25**
- 4.1 Filtrage 26
- 4.2 Exploration 27
- 4.3 Utilisation de graphes en programmation par contraintes 28
  - 4.3.1 Etude et décomposition de CSP 28
  - 4.3.2 Filtrage de contraintes globales 28
- 4.4 Modélisation d'un graphe par des variables 30
  - 4.4.1 Un modèle basé sur des variables entières 30
  - 4.4.2 Un modèle booléen 32
  - 4.4.3 Un modèle ensembliste 32
  - 4.4.4 Un modèle dédié 33

II Des contraintes sur les graphes 35

5 Filtrage des contraintes d'arbres et de chemins orientés **37**
- 5.1 Définition des contraintes de graphe 38
  - 5.1.1 La contrainte TREE 38
  - 5.1.2 La contrainte ANTIARBO 39
  - 5.1.3 La contrainte ARBO 39
  - 5.1.4 La contrainte PATH 40
  - 5.1.5 Relations entre ces contraintes 40
- 5.2 Filtrage structurel 42
  - 5.2.1 Amélioration de la contrainte TREE 42
  - 5.2.2 Filtrage de la contrainte ARBO 43
  - 5.2.3 Filtrage de la contrainte PATH 44
- 5.3 Du chemin au circuit 46
  - 5.3.1 La contrainte CIRCUIT 46
  - 5.3.2 Filtrer la contrainte CIRCUIT à partir de la contrainte PATH 46
- 5.4 Evaluation empirique 47
  - 5.4.1 Passage à l'échelle pour la contrainte tree 48
  - 5.4.2 Recherche d'un circuit Hamiltonien 50
- 5.5 Conclusion du chapitre 54

6 Résolution du problème du voyageur de commerce en contraintes **55**
6.1 Modélisation . . . 56
6.1.1 Modèle mathématique . . . 56
6.1.2 Justification d'une approche non-orientée . . . 56
6.1.3 Modèle en contraintes . . . 57
6.1.4 Description du propagateur Lagrangien . . . 58
6.2 Etude du branchement . . . 62
6.2.1 Quelques heuristiques génériques sur les graphes . . . 63
6.2.2 Adaptation de *Last Conflict* à une variable graphe . . . 64
6.3 Etude expérimentale . . . 66
6.3.1 Impact de l'orientation sur la recherche d'un cycle Hamiltonien . . . 66
6.3.2 Evaluation empirique des heuristiques de branchement . . . 69
6.4 Conclusion du chapitre . . . 72

III Des graphes pour les contraintes **75**

7 Cliques et allocation de ressources **77**
7.1 Description formelle du problème . . . 78
7.1.1 Un modèle mathématique . . . 78
7.1.2 Un modèle en contraintes . . . 79
7.2 Reformulation par la contrainte de graphe NCLIQUE . . . 79
7.2.1 Filtrage de la contrainte NCLIQUE . . . 80
7.2.2 Prise en compte naturelle des contraintes de différences . . . 81
7.2.3 Inconvénient de la méthode . . . 81
7.3 Utilisation d'un graphe pour filtrer ATMOSTNVALUE . . . 82
7.3.1 Diversification du filtrage . . . 83
7.3.2 Intégration d'inégalités . . . 84
7.4 Etude expérimentale sur un problème de planification de personnel . . . 86
7.4.1 Un branchement basé sur la réification de contraintes globales . . . 87
7.4.2 Description des instances étudiées . . . 88
7.4.3 Analyse empirique du modèle . . . 90
7.4.4 Comparaison aux approches existantes . . . 93
7.5 Conclusion du chapitre . . . 97
7.5.1 Perspectives . . . 97

8 Auto-décomposition de contraintes globales **101**
8.1 Fondements théoriques . . . 102
8.1.1 f-monotonie de contrainte . . . 103
8.1.2 f-décomposition de contrainte . . . 103
8.1.3 f-préservation de cohérence . . . 104
8.1.4 Quatre exemples génériques d'auto-décomposition . . . 104
8.2 Implémentation générique d'une $f_{vn}^{\cap}$-décomposition . . . 109
8.2.1 Grandes lignes . . . 109
8.2.2 Aspects avancés . . . 110
8.3 Etude de la contrainte CUMULATIVE . . . 111
8.3.1 $f_{vn}^{\cap}$-décomposition de la contrainte . . . 111
8.3.2 Autres auto-décompositions de la contrainte . . . 113
8.3.3 Intérêt pratique . . . 113
8.4 Conclusion du chapitre . . . 116

9 Conclusion **119**

Liste des tableaux

3.1 Précision sur la terminologie liée à la notion de sous-graphe. 19

5.1 Comparaison d'une approche par décomposition, de la contrainte globale TREE [BFL05] et de sa version revisitée [FL11] pour partitionner un graphe orienté en anti-arborescences. 49
5.2 Apport du filtrage structurel pour filtrer CIRCUIT. 50
5.3 Apport de l'aléatoire pour passer de PATH à CIRCUIT. 52
5.4 Positionnement de notre approche par rapport à l'état de l'art, avec une heuristique de branchement lexicographique. 53
5.5 Positionnement de notre approche par rapport à l'état de l'art, avec une heuristique de branchement générique et adaptative (VSIDS [MMZ+01], ABS [MH12] et MinDom* [HE79, LSTV06]). 53

6.1 Filtrage structurel : propagateurs de la contrainte CYCLE($\mathcal{G}$) 58
6.2 Filtrage simple basé sur les coûts 58
6.3 Recherche d'un cycle Hamiltonien dans un graphe éparse de grande taille. 67

7.1 Filtrage de la relation d'équivalence sur une variable graphe $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. 80

8.1 Etude de passage à l'échelle pour la contrainte CUMULATIVE sur des instances aléatoires. Comparaison de sa version classique (FULL), propagée sur toutes les variables, à sa $f^{\cap}_{vn}$-décomposition (AUTO). La résolution est soumise à une limite de cinq minutes. . 114
8.2 Résolution d'un problème de jobshop. Comparaison de la valeur des meilleures solutions trouvées par FULL et AUTO, dans les cinq minutes allouées à la résolution de chaque instance. 116

Table des figures

3.1 Exemple de graphes. 18
3.2 Illustration des concepts de chaîne, chemin, cycle et circuits. 18
3.3 Union et intersection de graphes. 19
3.4 Exemple de graphes complets, cliques et ensembles indépendant. 20
3.5 Illustration de graphes complémentaires et inverses. 21
3.6 Un graphe non-orienté composé de deux composantes connexes, induites par les ensembles de noeuds $\{1, 2, 3, 4\}$ et $\{5, 6\}$. Le noeud 4 est un point d'articulation. 21
3.7 Connexité forte. 22
3.8 Un graphe de flot enraciné en 2. Le noeud 3 domine les noeuds 4 et 5. 22
3.9 Illustration des différents concepts d'arbre. 23

4.1 Exemple de CSP. 27
4.2 Apport de la contrainte ALLDIFFERENT sur le CSP de la figure 4.1. 27
4.3 Illustration de l'exploration arborescente d'un espace de recherche. Les numéros indiquent l'ordre de créations des noeuds de l'arbre de recherche. Le premier noeud, en vert, est appelé racine. Les noeuds roses indiquent des échecs. Chaque arc épais (resp. fin) représente l'application (resp. la réfutation) d'une décision. 28
4.4 Exemple de graphe G^* à rechercher, tel que $\underline{G} \subseteq G^* \subseteq \overline{G}$. 30
4.5 Utilisation de la représentation *successeurs* sur l'exemple de la figure 4.4. 31
4.6 Quelques exemples de graphes ne pouvant pas être correctement modélisés par la représentation *successeurs*. 31
4.7 Représentation booléenne du graphe recherché en figure 4.4. 32
4.8 Modèle ensembliste de représentation du graphe de la figure 4.4. 33
4.9 Modélisation de l'exemple de la figure 4.4 avec une variable de graphe. 34

5.1 Illustration de la contrainte TREE($\mathcal{G}$,N). 38
5.2 Illustration de la contrainte ANTIARBO($\mathcal{G}, 3$). 39
5.3 Illustration de la contrainte ARBO($\mathcal{G}, 3$). 40
5.4 Illustration de la contrainte PATH($\mathcal{G}, 1, 5$). 40
5.5 Passage de la contrainte TREE à la contrainte ANTIARBO. 41
5.6 Illustration du filtrage de ARBO($\mathcal{G}, 3$) basé sur la notion de dominance. 44
5.7 Illustration du filtrage de REDUCEDPATH($\mathcal{G}$, 1, 5), basé sur le graphe réduit de $\overline{\mathcal{G}}$. Les arcs en pointillés rouges sont éliminés par filtrage. 45
5.8 Illustration de la contrainte CIRCUIT. 46
5.9 Passage de la contrainte CIRCUIT à la contrainte PATH. 47
5.10 Etude du passage à l'échelle de la contrainte TREE. 49

6.1 Impact de l'orientation sur l'énumération des cycles/circuits d'un graphe complet. 57
6.2 Deux manières d'appliquer une relaxation par ARPM. 59
6.3 Illustration du calcul des coûts marginaux et de remplacement. 60
6.4 Illustration d'une variable graphe $\mathcal{G}$ et d'une relaxation du TSP. 64

6.5 Illustration de l'exploration d'un arbre de recherche en utilisant LC_FIRST conjointement à MAX_COST sur l'exemple de la figure 6.4. Après un échec, la prochaine décision doit inclure une arête incidente au premier noeud de la dernière décision calculée (ici le noeud A). 66
6.6 Caption for LOF 68
6.7 Résolution du problème du cavalier d'Euler avec trois modèles différents. l, n et m indiquent respectivement la largeur de l'échiquier, le nombre de noeuds et le nombre d'arêtes de l'instance traitée. 68
6.8 Evaluation quantitative des différentes configurations de branchement sur les 42 instances les plus simples de la TSPLIB, avec une limite de temps d'une minute. 70
6.9 Comparaison du temps de résolution (en secondes) de SOTA [BVHR+12] et de CHOCO avec l'heuristique MAX_COST et LC_FIRST. Le facteur d'accélération est donné par le ratio du temps de résolution de SOTA sur celui de CHOCO. 71
6.10 Comparaison de l'approche par contraintes de l'état de l'art avec notre approche, sur les instances difficiles de la TSPLIB et avec une limite de temps de 30000 secondes. 72

7.1 Illustration du SMPTSP, avec 5 ressources et 5 tâches 78
7.2 Illustration des propagateurs de type AMNV. Les noeuds de l'ensemble indépendant calculé par MD sont entourés. Les valeurs filtrées sont barrées. Seul un appel à l'algorithme de filtrage d'AMNV est représenté ici ; la propagation d'autres contraintes n'est pas prise en compte afin de ne pas bruiter notre analyse. 86
7.3 Impact de la réification de la contrainte $\textsc{Max}(\mathcal{X}) = N$ par la variable b_{max}. 88
7.4 Comparaison des caractéristiques des instances de Data_137 et Data_100. 89
7.5 Hétérogénéité des ressources (mesurée en quartiles), selon les instances triées par nombre de tâche croissant. d_{H} : distance de Hamming normalisée. 89
7.6 Ratio $\frac{|E_{\mathcal{CI}}|}{|E_{\mathcal{I}}|}$ des instances triées par nombre de tâche croissant. 90
7.7 Comparaisons des bornes inférieures ($\underline{N}$) calculées par $\underline{N}^{\neq}$, $\mathtt{MD}(G_{\mathcal{I}})$ et $\mathtt{MD}(G_{\mathcal{CI}})$ au noeud racine. Les instances sont triées par valeur croissante de $\underline{N}^{\neq}$. 91
7.8 Nombre d'instances résolues à l'optimum par $\uparrow$AMNV$\langle G_{\mathcal{CI}}|\mathtt{MD}, \mathrm{R}^k|\mathcal{R}_1\rangle$ en moins de 5 minutes, en fonction de k. 91
7.9 Nombre d'instances résolues à l'optimum en moins de 5 minutes, en fonction de k, selon différentes configurations d'AMNV. 92
7.10 Nombre d'instances résolues à l'optimum par $\uparrow$AMNV$\langle G_{\mathcal{CI}}|\mathtt{MD}, \mathrm{R}^k|\mathcal{R}_{1,3}\rangle$ en moins de 5 minutes, selon différentes valeurs de k, avec et sans l'ajout de la variable b_{max} pour réifier la contrainte $\textsc{Max}(\mathcal{X}) = N$. 93
7.11 Ecart relatif entre $\underline{N}$ et $\overline{N}$, selon le modèle utilisé, sur Data_137 et Data_100, après 6 minutes de résolution. Les instances sont triées par nombre de tâches croissant. 94
7.12 Nombre d'instances résolues à l'optimum par chaque modèle en fonction du temps d'exécution, sur Data_137. 95
7.13 Comparaison de la borne inférieure calculée par le modèle programmation par contraintes au noeud racine avec $k = 40$ ($\underline{N}_{R^{40}}$) et $k = 1000$ ($\underline{N}_{R^{1000}}$) à la relaxation de [KEB12] ($\underline{N}_{\mathcal{L}}$), sur les instances de Data_137 triées par nombre croissant de tâches. 96

8.1 Exemple portant sur un ensemble de variables $\mathcal{X} = \{x_1, x_2, x_3, x_4, x_5, x_6\}$. 104
8.2 Illustration d'une $f_{cc}^{\cap}$-décomposition. 105
8.3 Illustration de la $f_{vn}^{\cap}$-décomposition. 106
8.4 Illustration de $f^{\mathcal{D}}$ et $f_{\mathcal{P}_*}^{\mathcal{D}}$. 108
8.5 Propagation incrémentale pour CUMULATIVE basée sur la $f_{vn}^{\cap}$-décomposition. 112
8.6 Impact de l'auto-décomposition de CUMULATIVE sur un problème d'ordonnancement industriel. Chaque point représente la résolution d'une instance. 115

9.1 Les graphes sont partout. 121

1

Remerciements

Je tiens tout d'abord à remercier les personnes qui m'ont donné l'envie de faire cette thèse, à savoir les enseignants-chercheurs de l'Ecole des Mines de Nantes, et plus particulièrement Romuald Debruyne, Narendra Jussien, Xavier Lorca, Philippe David, Thierry Petit et Fabien Candelier, qui m'ont fait découvrir l'informatique et m'ont donné goût à l'algorithmique.

Ensuite, je tiens à remercier les personnes qui m'ont offert l'opportunité de faire cette thèse, à savoir Nicolas Beldiceanu et Xavier Lorca, mais aussi le CNRS et le conseil régional des Pays de la Loire qui m'ont financé. Je remercie chaleureusement mes rapporteurs, Christine Solnon et Jean-Charles Régin, ainsi que mes examinateurs, Christophe Lecoutre, Benoit Rottembourg et Narendra Jussien, qui m'ont fait l'honneur d'accepter d'être dans mon jury de thèse. Je les remercie également pour les discussions que nous avons pu avoir et qui m'ont enrichi.

Je remercie maintenant les personnes grâce auxquelles cette thèse s'est bien déroulée. Je remercie d'abord mon encadrant, Xavier Lorca, pour tout ce qu'il m'a apporté : son temps et ses conseils, mais également sa confiance et sa patience. Je remercie bien sûr très chaleureusement les autres personnes avec qui j'ai eu la chance de travailler : Tanguy Lapègue, Charles Prud'Homme, Gilles Chabert et Thierry Petit. Je remercie particulièrement Charles Prud'Homme et Renaud Masson pour avoir partagé avec moi toutes leurs connaissances dès le démarrage de ma thèse. Ils m'ont été d'une grande aide. Je remercie aussi Alban Derrien, Damien Prot, Odile Morineau et Agnès Leroux pour leur sympathie et les échanges passionnants que nous avons pu avoir ensemble.

Je remercie toute ma famille de m'avoir offert un environnement favorable à ce travail et de continuer à me soutenir dans mes projets. Je remercie tout particulièrement ma femme, Sophie, pour son amour et son soutien inconditionnel dans les moments parfois difficiles de la thèse.

Enfin, je remercie mon chat, Nougatine, qui m'a tenu compagnie lors de nombreuses nuits blanches. D'ailleurs, le voilà qui arrive...

Introduction

Les systèmes d'information modernes ont permis de simplifier la gestion et d'augmenter la productivité de nombreuses entreprises. Ces systèmes sont généralement constitués de grandes bases de données et de différents programmes de traitement de ces données. De plus en plus d'entre eux incluent également des briques logicielles permettant de résoudre au mieux des problèmes combinatoires complexes, afin que le système, ou son administrateur, puisse prendre les meilleures décisions [Ben04]. Dans un contexte économique en crise et un environnement de plus en plus complexe (normes, technologie, usages, *etc.*), ces composants représentent un enjeu majeur et sont en pleine croissance. Les problèmes combinatoires concernés peuvent par exemple porter sur l'allocation de machines virtuelles dans les centres de calculs [HLM13], l'optimisation de tournées de véhicules [Sha98], la planification de personnel [DKS+94, DKMS97, KEB12], l'optimisation de plans de production industrielle [FMT87, ABC+09, ARU13]. La plupart de ces problèmes sont NP-difficiles [GJ79], c'est-à-dire qu'il n'existe actuellement pas d'algorithme pour les résoudre en temps polynomial. De plus, à cause de la nature même des processus en jeu, ils sont généralement complexes à exprimer et contiennent souvent un grand nombre de contraintes métiers à respecter. Enfin, à l'ère du *Big Data*, une nouvelle difficulté apparaît : la capacité de passage à l'échelle de ces programmes. En effet, les systèmes d'information sont parvenus à emmagasiner une telle quantité d'information, que les exploiter est devenu un défi, même sur des problèmes polynomiaux. Résoudre des problèmes combinatoires de grande taille est donc à la fois un enjeu important pour le monde socio-économique et un réel défi scientifique.

Face à la diversité et au nombre grandissant de problèmes traités, des objets mathématiques permettant un bon niveau d'abstraction (comme les graphes ou automates) sont devenus incontournables. La théorie des graphes permet de simplifier un problème donné pour mieux en étudier les relations liant certains de ses éléments. Un graphe est défini par un ensemble d'éléments appelés noeuds, un noeud étant une abstraction sur un objet (e.g., ville, personne ou machine), et un ensemble de liens entre ces noeuds. Ces liens, appelés arcs ou arêtes, sont aussi abstraits (e.g., connexions ferroviaires, rencontres amoureuses, réseaux). Les graphes sont des objets mathématiques formels pour lesquels de nombreuses propriétés ont été caractérisées [Tar72, GJ79, GM95]. Cette abstraction permet de mieux capitaliser sur les avancées de la littérature et facilite grandement les échanges entre personnes travaillant dans des domaines applicatifs différents. Enfin, bien plus qu'un objet mathématique, un graphe est un catalyseur d'idées. En se ramenant à des concepts basiques, cette montée en abstraction permet d'avoir

facilement des intuitions. Des algorithmes extrêmement efficaces ont ainsi pu être conçus, pour résoudre des problèmes impliquant des milliers, voire des millions, de noeuds [Hab13]. Néanmoins, si la théorie des graphes permet de formaliser une problématique spécifique, elle n'est en général pas suffisante pour exprimer et résoudre un problème industriel dans son intégralité.

De nombreuses méthodes ont été mises au point et perfectionnées au fil des ans pour résoudre au mieux des problèmes industriels combinatoires. Le courant le plus répandu est la programmation mathématique, un cadre générique de résolution exacte qui se décompose en sous-catégories : programmation linéaire, programmation linéaire en nombres entiers, programmation quadratique, programmation semi-définie, programmation non-linéaire, pour ne citer que les plus connues. Malheureusement, ces approches sont limitées par la nature et la taille des instances traitées. Concernant les problèmes de graphes, il est parfois difficile de dépasser les quelques centaines ou milliers de noeuds. Face aux limites de ces méthodes exactes, des méthodes approchées (e.g., recherche locale, méta-heuristiques [DDCG99]) ont été développées pour proposer de meilleurs compromis entre temps de calcul et qualité des solutions (respect des contraintes et valeur d'une éventuelle fonction objectif). Ces méthodes permettent de résoudre des problèmes définis sur des graphes ayant des millions de noeuds, mais sans possibilité de prouver l'optimalité de la solution. [Hel00, RGJM07].

Un autre paradigme est la programmation par contraintes [Mon74, Lau78, VH89, RvBW06], une discipline à la croisée de la recherche opérationnelle et de l'intelligence artificielle. Elle offre à l'utilisateur un langage déclaratif, expressif et compositionnel qui facilite la modélisation de problèmes complexes. Au delà de l'aspect modélisation, la programmation par contraintes propose une technique de résolution de problèmes basée sur l'algorithme du *backtrack* [HE79]. Pour faire simple, un solveur de contraintes commence par considérer un large ensemble des choix possibles pour résoudre un problème. Ensuite, le processus de résolution consiste à appliquer des algorithmes de filtrage (on parle de propagation de contraintes), pour éliminer des choix ne pouvant mener à solution, et émettre des hypothèses (on parle d'exploration), pour essayer de construire une solution. La force principale de la programmation par contraintes réside donc en la capacité de ses contraintes à identifier les combinaisons de valeurs (on parle de tuples) n'aboutissant à aucune solution. Enfin, la programmation par contraintes a également une forte composante relevant du génie logiciel. Elle offre de grandes possibilités d'hybridation [Ben04]. Un solveur de contraintes peut être vu comme une grande plateforme d'intégration d'algorithmes permettant de combiner facilement différentes techniques issues de la recherche opérationnelle et de l'intelligence artificielle : relaxations [FLM02, Sel04], recherche locale [Sha98], évolution de population [KAS10], apprentissage [Jus03], *etc.* L'utilisation de graphes en programmation par contraintes a déjà été réalisée avec succès dans de nombreux cas. On notera l'étude théorique de la structure d'un réseau de contraintes [EMJT14], la décomposition arborescente [GLS00, JT03, JT14], l'utilisation d'algorithmes de graphe pour filtrer des contraintes globales [Rég94, Rég96, Rég11], la reformulation de contraintes globales [BCRT05, BPR06, BCDP07] ou encore la résolution de problèmes définis sur des graphes. Cette dernière utilisation des graphes se fait essentiellement sous deux formes : soit les graphes sont des variables du problème [CL97, LPPRS02, DDD05, Doo06, QVRDC06, BFL08], soit ils en sont une donnée d'entrée [Rég03, Sol10, NS11, GM12, GFM+14]. Malheureusement, de telles approches ont souvent du mal à passer à l'échelle et sont généralement limitées à quelques dizaines voir quelques centaines de noeuds. Cette difficulté s'explique par la nature même de la programmation par contraintes, qui implique l'exécution d'algorithmes de filtrage relativement lourds à chaque étape du processus de résolution, soit potentiellement des millions de fois, pour aboutir à une solution. En dépit de ce fait, il a déjà été montré que les contraintes pouvaient former une approche efficace pour résoudre certains problèmes combinatoires de grande taille [Rég03, Sol10, LCB13], encourageant ainsi nos travaux.

Dans ce contexte, nous nous fixons pour objectif d'améliorer le passage à l'échelle des approches par contraintes utilisant des graphes. Dans un premier temps, nous repartirons de récents travaux portant sur la résolution de problèmes de graphes [BFL05, BVHR+12, FS14], afin de chercher à en améliorer les résultats. Pour cela, nous étudierons comment exploiter au mieux les structures de graphe sur des questions de modélisation et de filtrage. Nous nous concentrerons sur des problèmes impliquant la recherche d'un circuit Hamiltonien. Nous étudierons également diverses heuristiques de branchement sur un graphe, afin de gagner en généralité. Par la suite, nous chercherons à nous extraire progressivement des problèmes classiquement définis sur les graphes pour exploiter ce concept sur des problèmes définis sur les entiers, voire les réels. Nous verrons ainsi comment la théorie des graphes aide à résoudre un problème d'allocation de ressources en contraintes. Enfin, nous proposerons d'auto-décomposer des contraintes globales, en modifiant le domaine de définition de leurs algorithmes de filtrage à l'aide de propriétés remarquables sur des familles de graphes particulières. Le fil conducteur de ces travaux sera l'exploitation de concepts de graphe dans le but d'augmenter la taille des problèmes pouvant être résolus par une approche en contraintes, avec une volonté croissante de gagner en généralité.

Le plan de ce document est le suivant : la partie I fournit le bagage technique ainsi que les notations nécessaires à la lecture du manuscrit. Le chapitre I.3 donne les notions de théorie des graphes qui seront exploitées. Le chapitre I.4 introduit brièvement la programmation par contraintes et présente l'utilisation de graphes dans ce contexte. La partie II traite des problèmes de recherche d'un arbre, chemin, ou cycle dans un graphe. Le chapitre II.5 est dédié à l'étude du filtrage de telles contraintes, définies sur des graphes orientés. La contribution majeure de ce chapitre, publiée à *CP'11* [FL11], est la réduction de la complexité théorique et la simplification des règles de filtrage de la contrainte TREE. Le chapitre II.6 traite de la résolution du problème du cycle Hamiltonien et du problème du voyageur de commerce, définis sur un graphe non-orienté. Il propose une vaste étude expérimentale sur les heuristiques de branchement, acceptée pour publication dans *Constraints* [FLR14], et démontre l'excellence de la programmation par contraintes pour résoudre le problème du cycle Hamiltonien. La partie III porte sur l'utilisation de graphes pour le filtrage de contraintes globales n'étant initialement pas définies sur des graphes. Le chapitre III.7 se focalise sur le filtrage de la contrainte NVALUE basé sur une propriété de graphe. Il propose également une approche originale pour l'exploration de l'espace de recherche. Ce travail a donné lieu à un prix *best student paper* à *CP'13* [FL13] et a par la suite été étendu par une version journal, publiée dans *Artificial Intelligence* [FL14]. Le chapitre III.8 exploite des familles de graphes pour l'auto-décomposition de contraintes globales. Ce travail a été publié à *ECAI'14* [FLP14]. Enfin, ce document s'achève sur un bilan synthétique du travail réalisé et sur un rappel des principales perspectives de recherche qui en découlent.

Etat de l'art

3

Théorie des Graphes

Sommaire

3.1 Définitions et notations de base 17

3.2 Concepts avancés 20

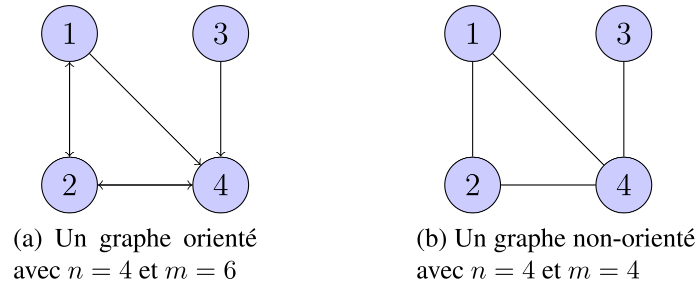
Dans ce chapitre, nous introduisons les définitions et notations relatives aux graphes que nous utiliserons par la suite. Ces notions de base sont décrites dans [Tar72, GJ79, GM95].

3.1 Définitions et notations de base

Un *graphe* est une abstraction mathématique servant à étudier les propriétés discrètes d'un réseau d'objets abstraits. Ces objets sont appelés *noeuds* ou *sommets* et les liens qui les relient sont appelés *arêtes* ou *arcs*. Formellement, un graphe est une paire ordonnée $G = (V, A)$ comprenant un ensemble de noeuds V (de l'anglais *Vertices*) et un ensemble d'arcs $A \subseteq V^2$ induisant une relation binaire sur V. Le graphe G est dit *orienté* ou *non-orienté* selon que la relation induite par A est respectivement asymétrique ou symétrique. Une illustration de ce concept est donnée par la figure 3.1. La non-orientation d'un graphe est une propriété forte qui simplifie nombre de concepts et algorithmes. Pour la mettre en avant, nous emploierons des termes et notations spécifiques. Par convention, dans le cas non-orienté, nous parlerons d'*arêtes* (*Edges* en anglais) et non d'arcs. De plus, cet ensemble sera noté E et non A. Ainsi, dans la suite du document, un graphe noté $G = (V, A)$ sera implicitement orienté alors qu'un graphe noté $G = (V, E)$ sera implicitement non-orienté.

Un graphe est dit *simple* s'il contient au plus une arête (ou un arc) entre toute paire (ordonnée) de noeuds et s'il ne contient pas de boucle (arête ou arc dont les extrémités coïncident). Les définitions suivantes sont toutes basées sur des graphes simples. Par la suite, les graphes étudiés dans les chapitres suivant n'auront pas plus d'une arête (ou arc) entre deux mêmes sommets. En revanche, sans perte de généralité, certains contiendront des boucles.

Le nombre de noeuds contenus dans un graphe G est appelé *ordre* de G. Par convention, il est noté n. De même, le nombre d'arcs (ou d'arêtes, selon l'orientation du graphe) contenues dans un graphe G est noté m. La *taille* d'un graphe G est $|G| = n + m$. Ainsi, on dira d'un algorithme dont la complexité temporelle vaut $O(|G|)$ qu'il s'exécute en temps linéaire.



(a) Un graphe orienté avec $n = 4$ et $m = 6$

(b) Un graphe non-orienté avec $n = 4$ et $m = 4$

FIGURE 3.1 – Exemple de graphes.

chaine, cycle, chemin et circuit

Dans un graphe $G = (V, E)$, une *chaîne* allant d'un noeud $i \in V$ à un noeud $j \in V$ est une séquence d'arêtes consécutives ayant les noeuds i et j pour extrémités : $\langle(i, k), (k, l), ...(p, j)\rangle$. Un *cycle* est une chaîne dont les sommets extrémaux coïncident, *i.e.,* une séquence d'arêtes de la forme $\langle(i, k), (k, l), ...(p, i)\rangle$. Dans un graphe orienté $G = (V, A)$, un *chemin* allant d'un noeud $i \in V$ à un noeud $j \in V$ est une séquence d'arcs consécutifs ayant les noeuds i et j pour extrémités : $\langle(i, k), (k, l), ...(p, j)\rangle$. Un *circuit* est un chemin dont les sommets d'origine et de fin coïncident, *i.e.,* une séquence d'arcs de la forme $\langle(i, k), (k, l), ...(p, i)\rangle$.

Un chemin ou une chaîne est dit *élémentaire* s'il ne passe pas deux fois par le même sommet. Par défaut, tous les chemins et chaînes considérés seront implicitement élémentaires.

Comme l'illustre la figure 3.2, on peut utiliser les concepts de chaîne et de cycle dans les graphes orientés en négligeant l'orientation des arcs. Un chemin (resp. un circuit) est une chaine (resp. un cycle), mais la réciproque n'est pas toujours vraie.

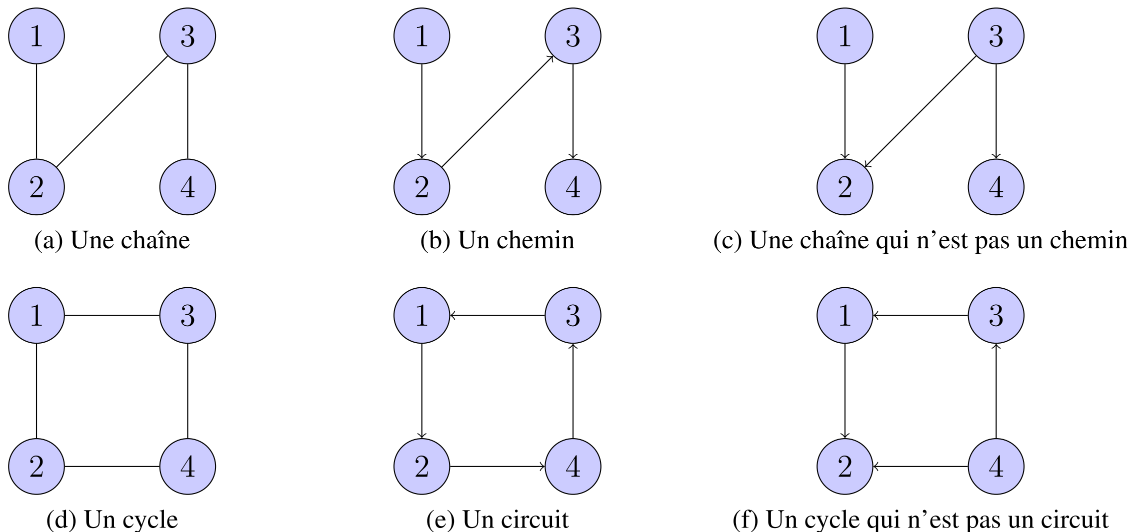


FIGURE 3.2 – Illustration des concepts de chaîne, chemin, cycle et circuits.

Union, intersection et inclusion de graphes

Considérons deux graphes $G_1 = (V_1, A_1)$ et $G_2 = (V_2, A_2)$. Leur *union*, notée $G_1 \cup G_2$, est donnée par le graphe $(V_1 \cup V_2, A_1 \cup A_2)$. Leur *intersection*, notée $G_1 \cap G_2$, est donnée par le graphe $(V_1 \cap V_2, A_1 \cap A_2)$. Une illustration est donnée en figure 3.3. L'*inclusion* de G_1 dans G_2, notée $G_1 \subseteq G_2$, signifie que $V_1 \subseteq V_2 \wedge A_1 \subseteq A_2$. Si $G_1 \subseteq G_2$, alors G_1 est un *sous-graphe*

de G_2 et G_2 est un *super-graphe* de G_1. Si $V_1 \subseteq V_2$ et $A_1 = A_2 \cap V_1^2$, alors G_1 est un sous-graphe de G_2 *induit* par V_1. Si $V_1 = V_2$ et $A_1 \subseteq A_2$, alors G_1 est un sous-graphe *recouvrant* de G_2. Cette terminologie est la plus répandue dans le monde (en anglais). En revanche, comme l'indique la table 3.1, il existe une autre terminologie qui est assez répandue en français. Nous justifions notre choix par la volonté de considérer d'emblée le problème le plus général, à savoir celui de la recherche de deux graphes G_1 et G_2 tels que $V_1 \subseteq V_2 \wedge A_1 \subseteq A_2$. Tout éventuel cas particulier est alors vu comme une propriété supplémentaire à satisfaire.

Restrictions	Terminologie de la thèse	Terminologie répandue en Français
Aucune	G_1 est un sous-graphe de G_2	G_1 est un sous-graphe partiel de G_2
$V_1 = V_2$	G_1 est un sous-graphe recouvrant de G_2	G_1 est un sous-graphe de G_2
$A_1 = A_2 \cap V_1^2$	G_1 est un sous-graphe de G_2 induit par V_1	G_1 est un graphe partiel de G_2

TABLE 3.1 – Précision sur la terminologie liée à la notion de sous-graphe.

Une partition d'un graphe G est un ensemble $\{G_1, G_2, ..., G_p\}$ de graphes tel que $\forall i \in [1,p], \forall j \in [1,p], i \neq j, G_i \cap G_j = \emptyset$ et $G = \bigcup_{i \in [1,p]} G_i$. On appelle p la cardinalité de la partition.

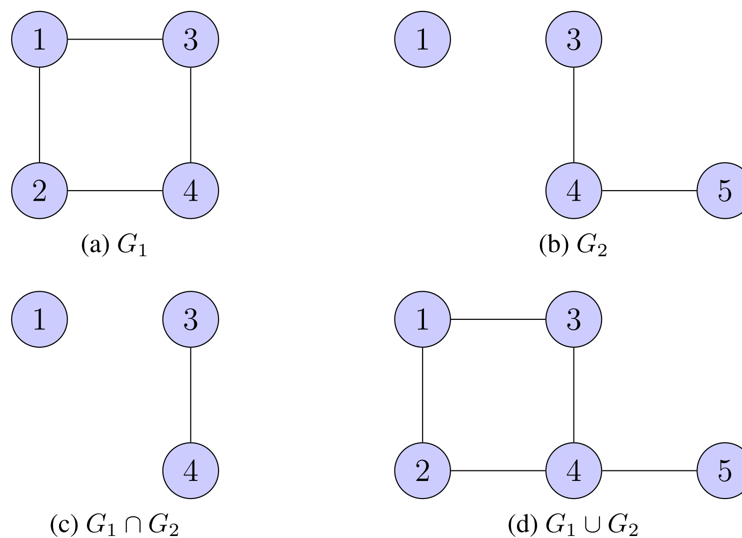


FIGURE 3.3 – Union et intersection de graphes.

Par soucis de simplicité, l'ajout et le retrait d'un noeud i dans un graphe G seront respectivement notés $G \cup \{i\}$ et $G \setminus \{i\}$. De même, l'ajout et le retrait d'une arête (i,j) dans un graphe G seront respectivement notés $G \cup \{(i,j)\}$ et $G \setminus \{(i,j)\}$.

Voisinage

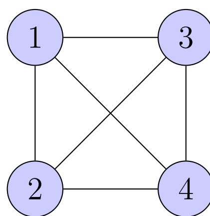
Etant donné un graphe non-orienté G et un noeud $i \in V$, les *voisins* de i dans G sont les noeuds $\{j \in V \mid (i,j) \in E\}$. Si le graphe est orienté, les *successeurs* (resp. *prédécesseurs*) de i dans G sont les noeuds $\{j \in V \mid (i,j) \in A\}$ (resp. $\{j \in V \mid (j,i) \in A\}$). Nous utiliserons les notations suivantes :

- V_G : dénote l'ensemble des noeuds du graphe G
- $V_G^{(i,i)}$: dénote l'ensemble des noeuds ayant une boucle dans le graphe G
- $\delta_G^+(i)$: dénote l'ensemble des successeurs du noeud i dans le graphe orienté G
- $\delta_G^-(i)$: dénote l'ensemble des prédécesseurs du noeud i dans le graphe orienté G
- $\delta_G(i)$: dénote l'ensemble des voisins du noeud i dans le graphe non-orienté G

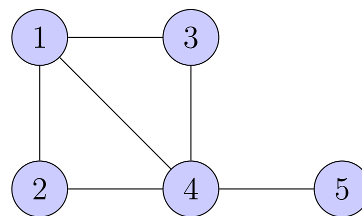
3.2 Concepts avancés

Graphe complet, clique et ensemble indépendant

Un graphe non-orienté $G = (V,E)$ est *complet* si et seulement si chaque paire de noeuds est connectée par une arête, *i.e.,* $|E| = \frac{|V|.(|V|-1)}{2}$. On note K_n le graphe complet d'ordre n. Une *clique* dans G est un sous-ensemble de V induisant un graphe complet. Un *ensemble indépendant* dans G est un sous-ensemble de ses noeuds, $I \subseteq V$, tel qu'aucune paire de noeuds n'est reliée dans G, *i.e.,* $\forall i,j \in I, (i,j) \notin E$. Un ensemble indépendant dans G est *maximum* s'il n'en existe pas de plus grand dans G. Sa cardinalité est notée $\alpha(G)$. Un ensemble indépendant I de G est *maximal au sens de l'inclusion* s'il n'existe pas d'ensemble indépendant I_2 tel que $I \subset I_2$. Un ensemble indépendant maximum est donc maximal au sens de l'inclusion, mais la réciproque est fausse. Une illustration est fournie en figure 3.4. Calculer un ensemble indépendant maximum dans un graphe quelconque est NP-difficile [GJ79].



(a) Le graphe complet K_4. Les ensembles de noeuds $\{1,2,3,4\}$ et $\{1\}$ forment respectivement une clique et un ensemble indépendant maximum.



(b) Un graphe quelconque : $\{1,2,4\}$ est une clique ; $\{2,3,5\}$ est un ensemble indépendant maximum ; $\{1,5\}$ est un ensemble indépendant maximal au sens de l'inclusion qui n'est pas maximum.

FIGURE 3.4 – Exemple de graphes complets, cliques et ensembles indépendant.

Inverse et complémentaire

Etant donné un graphe orienté $G = (V, A)$, son graphe inverse $G^{-1} = (V, A^{-1})$ est obtenu en inversant l'orientation de tous les arcs : $(i, j) \in A \Leftrightarrow (j, i) \in A^{-1}$. Son graphe complémentaire est $G^\circ = (V, V^2 \setminus A)$. De manière analogue, le complémentaire d'un graphe non-orienté $G = (V, E)$ indique les arêtes à compléter pour obtenir un graphe complet, *i.e.,* $G^\circ = K_{|V|} \setminus E$. Ces concepts sont illustrés sur la figure 3.5. On remarquera que si un ensemble C est une clique dans un graphe non-orienté G, alors C est un ensemble indépendant dans G°.

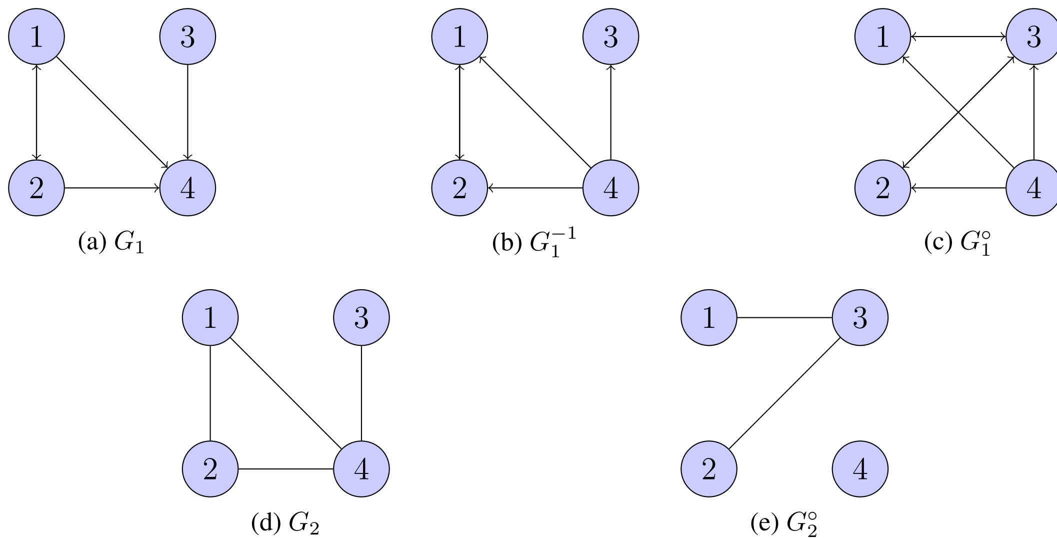


FIGURE 3.5 – Illustration de graphes complémentaires et inverses.

Connexité

Un graphe est *connexe* si et seulement si il existe une chaîne entre toute paire de noeuds. Une *composante connexe* dans un graphe non-orienté G est un sous-graphe connexe de G, maximal au sens de l'inclusion. Dans un graphe, chaque noeud appartient à exactement une composante connexe. Un *point d'articulation* de G est un noeud dont la suppression augmente le nombre de composantes connexes de G (figure 3.6). On peut calculer l'ensemble des composantes connexes et points d'articulations d'un graphe en temps linéaire [Tar72].

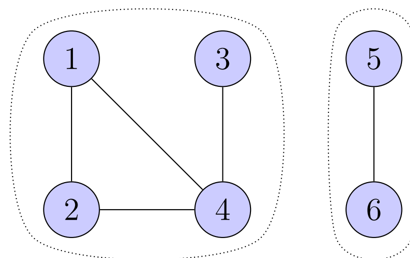
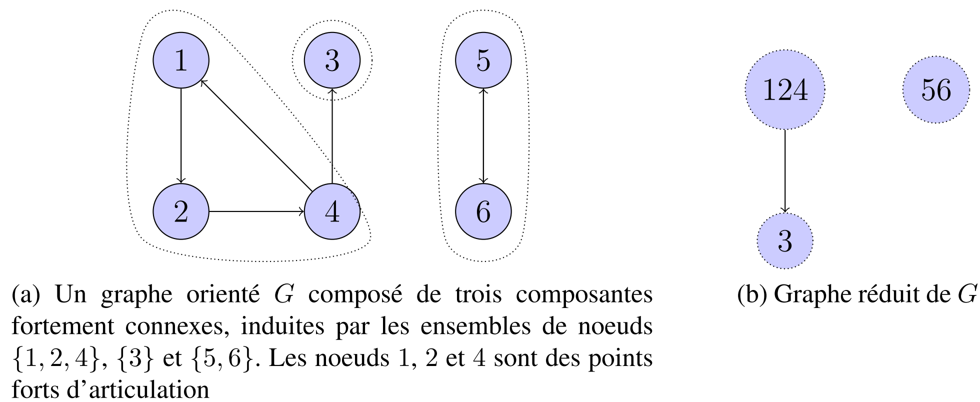


FIGURE 3.6 – Un graphe non-orienté composé de deux composantes connexes, induites par les ensembles de noeuds $\{1, 2, 3, 4\}$ et $\{5, 6\}$. Le noeud 4 est un point d'articulation.

Ces notions sont par essence non-orientées. Cependant, moyennant une simple montée en abstraction, on peut également les appliquer à des graphes orientés, en ignorant l'orientation des arcs, *i.e.,* en considérant chaque arc comme une arête. Il existe également une version orientée de ces concepts.

Un graphe orienté est *fortement connexe* si et seulement si il existe un chemin entre toute paire de noeuds. Une *composante fortement connexe* dans un graphe orienté G est un sous-graphe fortement connexe de G, maximal au sens de l'inclusion. Dans un graphe, chaque noeud appartient à exactement une composante fortement connexe. Un *point fort d'articulation* de G est un noeud dont la suppression augmente le nombre de composantes fortement connexes de G. Le *graphe réduit* de G est le graphe obtenu en fusionnant les noeuds appartenant à une même composante fortement connexe (figure 3.7). On peut calculer l'ensemble des composantes fortement connexes et points forts d'articulations d'un graphe en temps linéaire [Tar72, ILS10].

Notez que la définition usuelle n'impose pas qu'une composante (fortement) connexe soit maximale au sens de l'inclusion. Cependant, seules ces dernières présentent un intérêt. C'est pourquoi nous incluons directement cette restriction dans notre définition.



(a) Un graphe orienté G composé de trois composantes fortement connexes, induites par les ensembles de noeuds $\{1, 2, 4\}$, $\{3\}$ et $\{5, 6\}$. Les noeuds 1, 2 et 4 sont des points forts d'articulation

(b) Graphe réduit de G

FIGURE 3.7 – Connexité forte.

Dominance

Etant donnés un graphe orienté $G = (V, A)$ et un noeud $r \in V$, si pour chaque noeud $v \in V$ il existe un chemin allant de r à v, alors on dit que $G(r)$ est un *graphe de flot enraciné en* r. De plus, étant donnés deux noeuds i et j dans V, on dit que i *domine* j dans $G(r)$ si et seulement si tous les chemins allant de r à j passent pas i. On dit que i *domine immédiatement* j dans $G(r)$, si i domine j dans $G(r)$ et si il n'existe pas de noeud $k \in V$, $k \neq i$, $k \neq j$, tel que i domine k et k domine j dans $G(r)$. En d'autre termes, le dominant immédiat d'un noeud $j \in V$ est le noeud dominant de j le plus bas dans tout chemin allant de r à j. Mis à part r, chaque noeud a exactement un dominant immédiat. Un exemple est fourni sur la figure 3.8. L'ensemble des noeuds dominant d'un graphe de flot enraciné peut être calculé en temps linéaire [BKRW98, AHLT99].

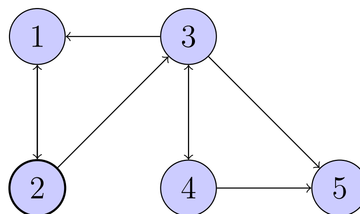
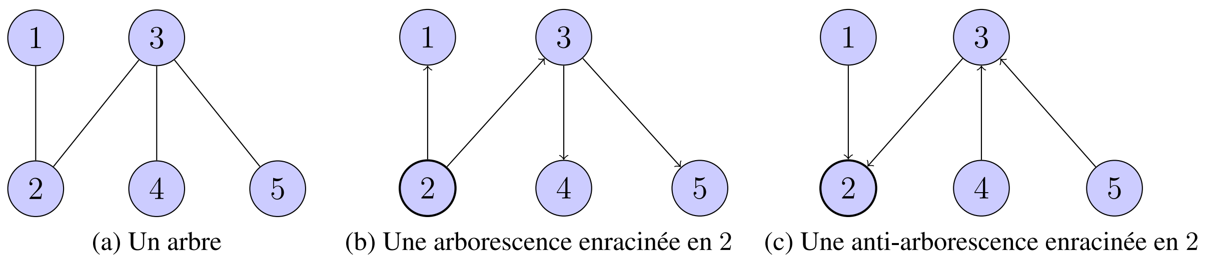


FIGURE 3.8 – Un graphe de flot enraciné en 2. Le noeud 3 domine les noeuds 4 et 5.

Arbre et arborescences

Un *arbre* est un graphe non orienté $G = (V, E)$ connexe et sans cycle. Il satisfait l'égalité suivante : $|E| = |V| - 1$. Une *arborescence enracinée en* r est un graphe de flot $G(r)$ connexe et sans cycle. Par conséquent, chaque noeud de $V \setminus \{r\}$ a exactement un prédécesseur et r n'a aucun prédécesseur. On a donc $|A| = |V| - 1$. Une *anti-arborescence enracinée en* r est un graphe orienté G tel que G^{-1} est une arborescence enracinée en r. Une illustration est donnée sur la figure 3.9.



(a) Un arbre (b) Une arborescence enracinée en 2 (c) Une anti-arborescence enracinée en 2

FIGURE 3.9 – Illustration des différents concepts d'arbre.

4

Programmation par contraintes

Sommaire

4.1 Filtrage **26**
4.2 Exploration **27**
4.3 Utilisation de graphes en programmation par contraintes **28**
4.3.1 Etude et décomposition de CSP 28
4.3.2 Filtrage de contraintes globales 28
4.4 Modélisation d'un graphe par des variables **30**
4.4.1 Un modèle basé sur des variables entières 30
4.4.2 Un modèle booléen 32
4.4.3 Un modèle ensembliste 32
4.4.4 Un modèle dédié 33

La programmation par contraintes [Mon74, Lau78, VH89, RvBW06] est un paradigme de programmation déclaratif permettant de modéliser et résoudre efficacement des problèmes de satisfaction de contraintes, appelés CSP (de l'anglais *Constraint Satisfaction Problem*). Une caractéristique importante de la programmation par contraintes est qu'elle capitalise ses contributions pour les réutiliser dans d'autres contextes. Ainsi, les contraintes, qui représente la majeure partie de l'intelligence d'un modèle, peuvent se combiner à souhait. Formellement, un CSP est défini par un ensemble de variables, leurs domaines respectifs et un ensemble de contraintes portant sur ces variables. Une *variable* est une inconnue du problème. Le *domaine* d'une variable est l'ensemble des valeurs pouvant être prises par cette variable. Une *contrainte* est une relation portant sur un ensemble de variables qui définit une restriction sur les combinaisons de valeurs autorisées pour les variables impliquées. Une solution est une affectation, de chaque variable à une valeur de son domaine, satisfaisant l'ensemble des contraintes du CSP. Le processus de résolution implique deux mécanismes : la propagation de contraintes, qui retire des valeurs incohérentes des domaines des variables à l'aide d'algorithmes de filtrage ; l'exploration de l'espace de recherche, qui permet d'énumérer les combinaisons de valeurs admissibles lorsque le raisonnement apporté par la propagation n'est pas suffisant.

4.1 Filtrage

L'application d'un algorithme de filtrage pour une contrainte donnée permet de retirer des domaines de ses variables des valeurs n'appartenant à aucune solution. Il arrive que les domaines filtrés respectent une certaine propriété vis-à-vis de la contrainte, on parle alors de niveaux de cohérence. La plus connue est la *domaine-cohérence* [VHSD95] (définition 1). On parle également d'*arc-cohérence* (AC) et d'*arc-cohérence généralisée* (GAC).

Définition 1 *Une contrainte est* domaine-cohérente *(DC) si, pour chacune de ses variables x et pour chaque valeur $v \in dom(x)$, il existe une affectation associant chaque variable à une valeur de son domaine, satisfaisant la contrainte et telle que $x = v$.*

La domaine-cohérence n'est pas le seul niveau de cohérence utilisé. Pour des raisons pratiques, il est fréquent d'employer des niveaux plus faibles de cohérence, comme la cohérence aux bornes (définition 2). Cette cohérence est souvent utilisée pour les variables ayant de très grands domaines munis d'une relation d'ordre, comme les variables entières liées à des mesures géométriques ou temporelles et les variables ensemblistes.

Définition 2 *Une contrainte est* borne-cohérente *(BC) si, pour chacune de ses variables x et pour chaque valeur v aux bornes du domaine de x, i.e., $v \in \{min(dom(x)), max(dom(x))\}$, il existe une affectation associant chaque variable à une valeur comprise entre les bornes de son domaine, satisfaisant la contrainte et telle que $x = v$.*

La propagation de contraintes consiste à appliquer les algorithmes de filtrage de chaque contrainte jusqu'à ce que plus aucune inférence ne soit possible. Cet état s'appelle atteindre un point fixe. Il existe différentes politiques de propagation, menant à différent résultats en termes de performance [PLDJ14]. Malheureusement, propager les contraintes indépendamment les unes des autres conduit dans de nombreux cas à un filtrage relativement faible. Pour bien comprendre les enjeux liés au filtrage, il est fondamental de bien comprendre qu'établir la DC sur une contrainte c_1 et sur une contrainte c_2 n'établit pas la DC sur la contrainte $c_3 = c_1 \wedge c_2$. Un contre-exemple pour s'en convaincre est un CSP constitué de deux variables binaires, x_1 et x_2 (prenant donc chacune leur valeur dans $\{0, 1\}$), et de deux contraintes, $x_1 = x_2$ et $x_1 \neq x_2$. Prises séparément, ces contraintes ne permettent aucune déduction alors que, prises simultanément, elles permettent de détecter immédiatement l'insatisfiabilité du CSP. On a donc parfois intérêt à ajouter des contraintes associées à des conjonctions de contraintes. Elles sont donc redondante d'un point de vue sémantique, mais permettent parfois d'améliorer le filtrage. Plus généralement, différentes techniques ont été proposées, comme les cohérences fortes [DB01, MDL14], les contraintes globales [Rég94, BVH03, Rég11, BCDP07] et les explications [Jus03]. Ce manuscrit exploite la notion de contrainte globale.

Il n'existe pas de définition formelle et universelle pour caractériser une contrainte globale [BVH03]. Nous disons qu'une contrainte est globale lorsqu'elle représente une conjonction de contraintes élémentaires et qu'elle représente un sous-problème susceptible d'être retrouvé dans de nombreux problèmes. D'ailleurs, de nombreux algorithmes exacts de recherche opérationnelle peuvent être transposés en algorithmes de filtrage pour des contraintes globales dédiées (e.g., [FLM02, Rég08, BVHR+12]). En pratique, l'utilisation d'une seule contrainte globale plutôt que d'un réseau de contraintes plus élémentaires offre souvent la possibilité d'obtenir un filtrage plus puissant et/ou un gain sur la complexité algorithmique du filtrage. Les contraintes globales sont donc des armes très puissantes qui font la force de la programmation par contraintes. Cependant, elles sont aussi à double tranchant : on peut vite perdre de vue la généralité de l'approche et chercher à proposer une contrainte globale dédiée à chaque

problème rencontré. Ce genre d'excès conduit à fabriquer de véritables usines à gaz qu'il est très difficile de maintenir dans le temps.

Variables :	**Contraintes :**
$\mathcal{X} = \{x_1, ..., x_5\}$	$\forall i, j \in [1, 5], i \neq j, x_i \neq x_j$
	$x_1 + x_3 = x_2$

Domaines initiaux :	**Domaines après filtrage :**	**Solution :**
$dom(x_1) = \{1\}$	$dom(x_1) = \{1\}$	$dom(x_1) = \{1\}$
$dom(x_2) = \{1, 3, 5\}$	$dom(x_2) = \{\not{1}, 3, 5\}$	$dom(x_2) = \{3\}$
$dom(x_3) = \{2, 4\}$	$dom(x_3) = \{2, 4\}$	$dom(x_3) = \{2\}$
$dom(x_4) = \{2, 4\}$	$dom(x_4) = \{2, 4\}$	$dom(x_4) = \{4\}$
$dom(x_5) = \{2, 4, 5\}$	$dom(x_5) = \{2, 4, 5\}$	$dom(x_5) = \{5\}$

FIGURE 4.1 – Exemple de CSP.

Un exemple de CSP est donné par la figure 4.1. On y retrouve un ensemble de cinq variables entières, devant prendre des valeurs distinctes telles que $x_1 + x_3 = x_2$. La propagation de la contrainte binaire $x_1 \neq x_2$ permet de retirer la valeur 1 du domaine de x_2, car la variable x_1 est déjà instanciée à cette valeur. On ne peut pas réduire davantage les domaines des variables en propageant chaque contrainte une à une. Il faut ensuite créer des hypothèses afin d'explorer l'espace de recherche et trouver une solution à ce problème.

On peut améliorer le modèle de la figure 4.1 en remplaçant les contraintes binaires de différences par une seule contrainte globale ALLDIFFERENT [Rég94] (figure 4.2). Cette contrainte utilise un algorithme de graphe pour trouver une affectation satisfaisant la contrainte et filtrer toutes les valeurs incohérentes des domaines des variables. Sur cet exemple, la phase de propagation des contraintes est suffisante pour supprimer toutes les valeurs des domaines qui n'appartient à aucune solution, au point de trouver immédiatement l'unique solution à ce CSP.

Domaines initiaux :	**Domaines après filtrage :**
$dom(x_1) = \{1\}$	$dom(x_1) = \{1\}$
$dom(x_2) = \{1, 3, 5\}$	$dom(x_2) = \{\not{1}, 3, \not{5}\}$
$dom(x_3) = \{2, 4\}$	$dom(x_3) = \{2, \not{4}\}$
$dom(x_4) = \{2, 4\}$	$dom(x_4) = \{\not{2}, 4\}$
$dom(x_5) = \{2, 4, 5\}$	$dom(x_5) = \{\not{2}, \not{4}, 5\}$

FIGURE 4.2 – Apport de la contrainte ALLDIFFERENT sur le CSP de la figure 4.1.

4.2 Exploration

Lorsque le raisonnement fourni par la propagation des contraintes n'est pas suffisant pour résoudre le problème traité, il est alors nécessaire d'explorer un espace de recherche. Cette étape constitue un élément essentiel en programmation par contraintes. Il existe une grande variété de stratégies, ou heuristiques, pour guider cette exploration. Classiquement, l'espace de recherche est exploré de manière arborescente à l'aide d'un parcours en profondeur d'abord, où à chaque noeud de branchement, une stratégie sélectionne une variable du problème et réduit son domaine. Généralement, il s'agit d'une réduction à une seule valeur, mais on peut par exemple décider de couper son domaine en deux parts égales. Si cette décision aboutit sur un échec, l'algorithme du backtrack remonte en arrière, réfute cette décision et en calcule une nouvelle,

jusqu'à trouver une solution ou prouver qu'il n'en existe aucune. La figure 4.3 en présente une illustration.

De nombreux travaux ont été menés pour faciliter la création de stratégies dédiées [STW+13, MFMS14] et proposer des heuristiques par défaut qui soient efficaces sur une multitude de problèmes [HE79, BHLS04, Ref04, MH12]. Au delà des heuristiques précises de la littérature, des concepts plus généraux ont été proposés, tels que l'utilisation de redémarrages, la recherche à voisinage large [Sha98], la réutilisation du dernier conflit [LSTV06] ou encore l'exploration parallèle [RRM13]. Dans l'ensemble, une bonne stratégie permet d'améliorer significativement les temps de résolution d'un modèle.

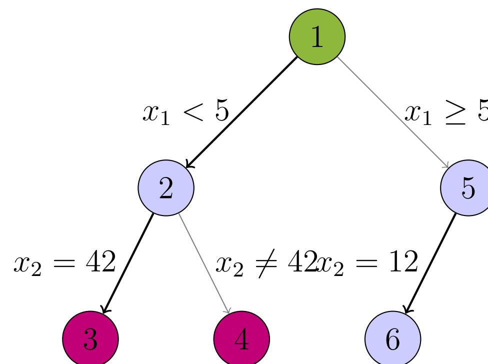


FIGURE 4.3 – Illustration de l'exploration arborescente d'un espace de recherche. Les numéros indiquent l'ordre de créations des noeuds de l'arbre de recherche. Le premier noeud, en vert, est appelé racine. Les noeuds roses indiquent des échecs. Chaque arc épais (resp. fin) représente l'application (resp. la réfutation) d'une décision.

4.3 Utilisation de graphes en programmation par contraintes

L'objectif de cette section est, d'une part, de rappeler quelques travaux de la littérature portant à la fois sur la programmation par contraintes et la théorie des graphes, afin de contextualiser cette thèse et, d'autre part, de montrer différentes manières de représenter un graphe par des variables.

4.3.1 Etude et décomposition de CSP

La structure de graphe est utilisée depuis de nombreuses années pour étudier la (micro) structure des CSP, avec des objectifs aussi bien théoriques que pratiques. Ces travaux exploitent fortement les structures de graphes : arborescences [JT03], composantes connexes [JT14] et cliques [EMJT14]. Une des applications principales de ces recherches concerne la conception de décompositions automatiques de problèmes, afin d'en réduire la complexité [GLS00].

4.3.2 Filtrage de contraintes globales

La notion de contrainte globale est née avec l'utilisation de graphes au coeur des algorithmes de filtrage. La plus célèbre d'entre elles, ALLDIFFERENT, est basée sur le problème du couplage maximum dans un graphe biparti [Rég94]. Tout comme le flot généralise le couplage, la contrainte ALLDIFFERENT a été généralisée vers la contrainte GLOBALCARDINALITY [Rég96]. Par la suite, des généralisations considérant une fonction de coût ont également été introduites [FLM99, PRB01, FLM02, Rég02].

Bourreau [Bou99] a proposé une famille de contraintes globales permettant de partitionner un graphe en circuits disjoints, sous contraintes multiples (temps, coûts, capacité de chargement etc.), pour réaliser des tournées de véhicules. Cette contrainte repose sur une représentation interne d'un graphe dynamique, dans lequel les noeuds consécutifs sont fusionnés.

Beldiceanu *et al.* ont proposé des reformulations de contraintes globales par des propriétés de graphe [Bel00, BPR06, BCDP07]. Ces travaux offrent la possibilité de modéliser un large ensemble de contraintes globales à l'aide d'un vocabulaire réduit, à l'instar des reformulations à bases de contraintes binaires ou d'automates. Leur contribution réside davantage en la capacité originale à exprimer des contraintes avec des concepts de graphe que dans l'efficacité du filtrage qui en résulte.

Lorca *et al.* ont introduit une contrainte de partitionnement d'un graphe en anti-arborescences [BFL05] afin de résoudre des problèmes de phylogénie grâce à la notion de connexité forte dans un graphe orienté. Ils ont ensuite étendu leurs travaux pour proposer une contrainte de partitionnement en chemins disjoints [BL07] et à la prise en compte de contraintes annexes [BFL08]. Ils ont clairement identifié des invariants, qu'ils ont ensuite traduits sous la forme d'algorithmes *ad hoc* de filtrage. La critique principale que l'on peut adresser à leur approche est la proportion avec laquelle ils ont fait passer des considérations algorithmiques devant la recherche de simplicité. D'un point de vue théorique, regrouper toutes les contraintes d'un problème dans une contrainte globale *ad hoc* va un peu à l'encontre du caractère compositionnel de la programmation par contraintes. En effet, peut-on encore vanter la généricité de la programmation par contraintes si chaque réseau de contraintes est représenté par une contrainte globale *ad hoc* ? D'un point de vue purement pragmatique, l'implémentation de cette contrainte, à savoir plus de cent mille lignes de code, n'a pas survécu à l'épreuve du temps et n'est maintenue dans aucun solveur.

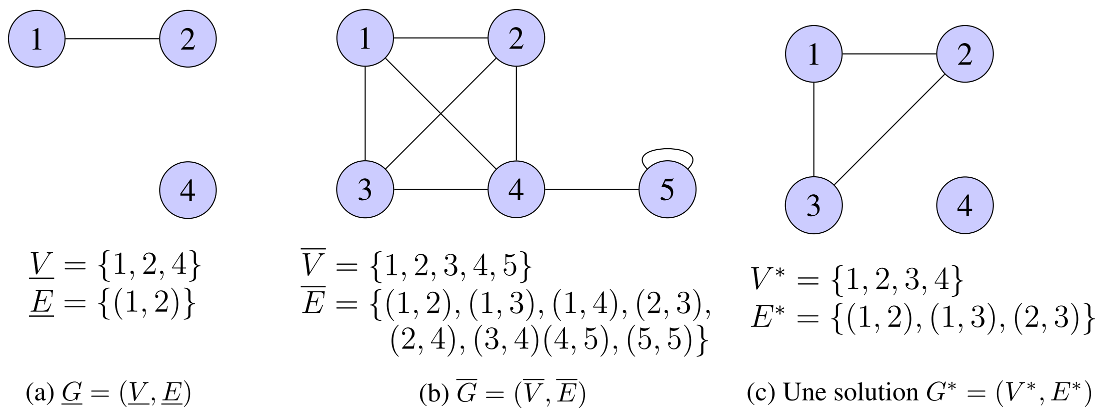
A l'inverse, Quesada *et al.* proposent une contrainte globale faisant le lien entre un graphe, son arbre de dominance et sa fermeture transitive [QVRDC06, Que06]. Plutôt que d'utiliser un algorithme de filtrage *ad hoc* pour chaque propriété qu'ils cherchent à établir, leur approche consiste à utiliser une variable de graphe par propriété (la dominance et la transitivité pour notre cas d'étude) et utiliser ensuite des contraintes plus simples pour lier ces variables entre elles. Bien qu'un peu lourde à court terme, cette approche possède une certaine élégance en ce qu'elle permet, sur un plus long terme, une meilleure capitalisation du travail effectué (l'arbre de dominance et la fermeture transitive pouvant alors être directement utilisés dans d'autres contraintes, voir pour guider l'heuristique de branchement). On retrouve sans surprise cette caractéristique dans les travaux de Dooms *et al.* [DDD05, Doo06], sur lesquels Quesada *et al.* se sont basés. On notera également que, à l'opposé des travaux de Lorca *et al.*, les aspects algorithmiques ont cette fois-ci été quelque peu délaissés : s'ils ont réussi à faire un lien entre la notion de dominance et la contrainte TREE, ils n'ont pas su exploiter cette propriété en temps linaire. Enfin, leurs études expérimentales sont relativement peu nombreuses et ne mettent pas vraiment en avant les intérêts algorithmiques et pratiques offerts par les variables de graphe. Par exemple, pour le problème du cavalier d'Euler, notre approche résout une instance à 40000 noeuds (voir figure 6.7) 5 fois plus vite que la leur sur une instance à seulement 1600 noeuds [Doo06].

Face à ce constat, nous allons essayer de concilier au mieux gains algorithmiques, qualité d'implémentation, et généralité de l'approche.

4.4 Modélisation d'un graphe par des variables

De nombreux problèmes impliquent de trouver un graphe G^* qui soit compris entre deux graphes donnés $\underline{G}$ et $\overline{G}$. Plus précisément, on cherche un graphe G^* tel que $\underline{G} \subseteq G^* \subseteq \overline{G}$. Bien souvent, d'autres contraintes viennent s'ajouter à ce problème pour restreindre l'ensemble des graphes solutions (e.g., trouver un chemin, un graphe complet), d'où l'intérêt d'une approche par contraintes pour résoudre ce type de problèmes.

L'exemple de la figure 4.4 illustre ce problème. la figure 4.4a indique que la solution G^* doit contenir l'ensemble de noeuds $\{1, 2, 4\}$ et l'arête $(1, 2)$. Ensuite, G^* peut également inclure les noeuds 3 et 5, ainsi que les arêtes $(1, 3)$, $(1, 4)$, $(2, 3)$, $(2, 4)$, $(3, 4)$, $(4, 5)$ et $(5, 5)$, comme l'indique la figure 4.4b. Parmi les nombreuses solutions de ce problème, nous en affichons une sur la figure 4.4c.



(a) $\underline{G} = (\underline{V}, \underline{E})$ (b) $\overline{G} = (\overline{V}, \overline{E})$ (c) Une solution $G^* = (V^*, E^*)$

FIGURE 4.4 – Exemple de graphe G^* à rechercher, tel que $\underline{G} \subseteq G^* \subseteq \overline{G}$.

Nous allons maintenant nous intéresser aux diverses manières de modéliser la recherche de ce graphe en utilisant le paradigme de la programmation par contraintes.

4.4.1 Un modèle basé sur des variables entières

Dans certains cas, un graphe peut être modélisé en associant chaque noeud possible $i \in \overline{V}$ du graphe avec une variable entière $succ_i$, représentant son unique successeur. Ainsi $succ_i = j$ indique que l'arc allant de i à j appartient au graphe solution. Par convention, on utilise généralement une boucle ($succ_i = i$) pour indiquer qu'un noeud $i \in \overline{V}$ n'appartient pas à la solution. Il s'agit donc d'une représentation orientée. Cette modélisation est communément appelée *successeurs*. Une illustration de la représentation *successeurs* sur notre exemple courant est donné sur la figure 4.5. On peut voir que forcer l'arête $(1, 2)$ à appartenir à une solution revient à poser une disjonction de contraintes, dont la propagation sera vraisemblablement faible. Dans ce modèle, la solution non-orientée attendue conduit à deux solutions symétriques.

Domaines initiaux :
$dom(succ_1) = \{-1, 1, 2, 3, 4, 6\}$
$dom(succ_2) = \{-1, 1, 2, 3, 4, 6\}$
$dom(succ_3) = \{-1, 1, 2, 3, 4, 6\}$
$dom(succ_4) = \{-1, 1, 2, 3, 4, 5, 6\}$
$dom(succ_5) = \{-1, 4, 5, 6\}$

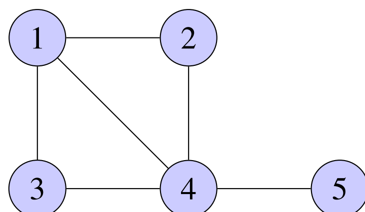
Contraintes :
$succ_1 = 2 \vee succ_2 = 1$

Solution 1 :
$succ_1 = 3, \quad succ_2 = 1$
$succ_3 = 2, \quad succ_4 = 6$
$succ_5 = -1$

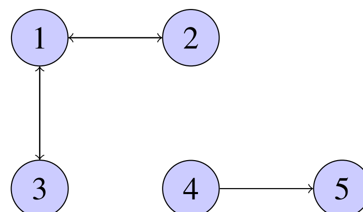
Solution 2 :
$succ_1 = 2, \quad succ_2 = 3$
$succ_3 = 1, \quad succ_4 = 6$
$succ_5 = -1$

FIGURE 4.5 – Utilisation de la représentation *successeurs* sur l'exemple de la figure 4.4.

Si cette modélisation convient bien à certaines contraintes, telles que CIRCUIT et SUBCIRCUIT [Lau78, Bou99, NIC, BCDP07], cette modélisation n'est pas la plus appropriée dans le cas général. Tout d'abord, si le graphe orienté est non orienté, forcer ainsi à raisonner sur un graphe orienté, bien que parfaitement correct, ajoute une nouvelle dimension au problème, le rendant plus difficile à traiter. De plus, les contraintes portant sur le voisinage des noeuds seront plus complexes à formuler et à implémenter, car il faudra considérer les successeurs et les prédécesseurs des noeuds. Deuxièmement, comme l'indique la figure 4.6, la modélisation *successeurs* a un champ d'application relativement restreint. Puisque $|\overline{V}|$ variables entières sont utilisées, cette représentation ne permet pas de considérer des graphes ayant plus de $|V|$ arcs (see figure 4.6a). Plus généralement, sans même considérer le nombre d'arcs, de nombreux graphes n'ont pas de représentations *successeurs* (figure 4.6b). Dans le cas où le graphe solution est supposé contenir moins de $|\overline{V}|$ arcs, on doit pouvoir caractériser les affectations de valeurs ne représentant pas un arc. Dans le cas de la contrainte TREE, des boucles étaient utilisées dans ce but. Le problème est qu'il y a alors un conflit entre les boucles supposées représenter un noeud que l'on supprime et celles supposées représenter un noeud n'ayant simplement pas de successeur. Le choix d'utiliser des valeurs n'étant pas associées à des noeuds du graphe (e.g., $succ_i = |\overline{V}| + 1$ indique que i n'a pas de successeur), utilisé pour la contrainte PATH dans [FS14], semble plus pertinent sur ce point. D'ailleurs, si l'on veut pouvoir représenter des noeuds qui possèdent explicitement une boucle, alors il faut que les noeuds supprimés ne soient plus représentés par une boucle mais par une deuxième valeur joker, e.g.,-1.



(a) Plus d'arêtes que de noeuds.



(b) Trop de successeurs pour le noeuds 1.

FIGURE 4.6 – Quelques exemples de graphes ne pouvant pas être correctement modélisés par la représentation *successeurs*.

En conclusion, la modélisation *successeurs* est limitée à une petite classe de graphes et devient vite source de confusion dans le cas de sous-graphes et de noeuds sans successeurs.

4.4.2 Un modèle booléen

N'importe quel graphe, orienté ou non, peut être représenté à l'aide de variables booléennes. On associe alors chaque noeud $i \in \overline{V}$ (resp. chaque arête $(i,j) \in \overline{E}$) à une variable booléenne b_i (resp. $b_{i,j}$) pour indiquer s'il est présent ou non dans le graphe solution. Des contraintes logiques permettent d'assurer la cohérence de la représentation : une variable représentant une arête (i,j) implique les variables représentant les noeuds i et j. La figure 4.7 illustre le modèle booléen sur l'exemple courant.

Domaines initiaux :
$dom(b_1) = \{\texttt{true}\}$
$dom(b_2) = \{\texttt{true}\}$
$dom(b_3) = \{\texttt{false}, \texttt{true}\}$
$dom(b_4) = \{\texttt{true}\}$
$dom(b_5) = \{\texttt{false}, \texttt{true}\}$
$\forall (i,j) \in \overline{E} \setminus \{(1,2)\},$
$\quad dom(b_{i,j}) = \{\texttt{false}, \texttt{true}\}$
$dom(b_{1,2}) = \{\texttt{true}\}$

Contraintes :
$\forall (i,j) \in \overline{E},$
$\quad b_{i,j} \Rightarrow b_i \wedge b_j$

Solution :
$b_1 = \texttt{true},\ \ b_2 = \texttt{true},\ \ b_3 = \texttt{true},\ \ b_4 = \texttt{true},\ \ b_5 = \texttt{false}$

$b_{1,2} = \texttt{true},\ \ b_{1,3} = \texttt{true},\ \ b_{1,4} = \texttt{false},\ \ b_{2,3} = \texttt{true},$
$b_{2,4} = \texttt{false},\ \ b_{3,4} = \texttt{false},\ \ b_{4,5} = \texttt{false},\ \ b_{5,5} = \texttt{false}$

FIGURE 4.7 – Représentation booléenne du graphe recherché en figure 4.4.

4.4.3 Un modèle ensembliste

Un plus haut niveau d'abstraction peut être atteint en utilisant des variables ensemblistes. Une variable ensembliste $\mathcal{S}$ représente un ensemble. son domaine est défini par un intervalle d'ensembles $dom(\mathcal{S}) = \left[\underline{\mathcal{S}}, \overline{\mathcal{S}}\right]$. Une valeur $\mathcal{S}^*$ est un ensemble satisfaisant la relation $\underline{\mathcal{S}} \subseteq \mathcal{S}^* \subseteq \overline{\mathcal{S}}$.

Un graphe étant une paire ordonnée d'un ensemble de noeuds et d'un ensemble d'arêtes, il peut être modélisé à l'aide de deux variables ensemblistes (une pour les noeuds, une autre pour les arêtes). Cependant, la plupart des algorithmes requièrent d'itérer sur les voisins de certains noeuds, ce qui est très coûteux en temps si toutes les arêtes sont mises dans le même ensemble. Aussi, il est préférable d'utiliser une variable ensembliste $\mathcal{S}^n$ pour désigner les noeuds appartenant à la solution et un ensemble de variables ensemblistes $\mathcal{S}^v$ tel que pour chaque noeud i, la variable ensembliste $\mathcal{S}^v_i$ indique l'ensemble des voisins de i dans le graphe solution. La contrainte INVERSESET [BCDP07], assurant la symétrie de la relation de voisinage, ainsi que des contraintes logiques assurent alors la cohérence de la solution. Si le graphe est orienté, alors on n'utilise plus une variable ensembliste mais deux pour chaque noeud, pour représenter leurs successeurs et prédécesseurs. La figure 4.8 illustre l'utilisation de variables ensemblistes sur notre exemple.

Domaines initiaux :
$Dom(\mathcal{S}^n) = [\underline{V}, \overline{V}]$
$Dom(\mathcal{S}^v_1) = [\{2\}, \{2,3,4\}]$
$Dom(\mathcal{S}^v_2) = [\{1\}, \{1,3,4\}]$
$Dom(\mathcal{S}^v_3) = [\{\}, \{1,2,4\}]$
$Dom(\mathcal{S}^v_4) = [\{\}, \{1,2,3,5\}]$
$Dom(\mathcal{S}^v_5) = [\{\}, \{4,5\}]$

Contraintes :
INVERSESET($\mathcal{S}^v$,$\mathcal{S}^v$)
$\forall (i,j) \in \overline{E}, j \in \mathcal{S}^v_i \Rightarrow i \in \mathcal{S}^n \wedge j \in \mathcal{S}^n$

Solution :
$\mathcal{S}^n = V^*$
$\mathcal{S}^v_1 = \{2,3\}$
$\mathcal{S}^v_2 = \{1,3\}$
$\mathcal{S}^v_3 = \{1,2\}$
$\mathcal{S}^v_4 = \{\}$
$\mathcal{S}^v_5 = \{\}$

FIGURE 4.8 – Modèle ensembliste de représentation du graphe de la figure 4.4.

Cette montée en abstraction offre deux avantages pratiques conséquents : tout d'abord, cela permet d'utiliser une vaste collection de contraintes définies sur les variables ensemblistes et déjà implémentées dans les solveurs. Ensuite, cela permet de meilleures structures de données. Etant donné un noeud $i \in V$, l'ensemble des voisins potentiels et l'ensemble des voisins obligatoires de i peuvent chacun être retrouvé en temps constant. Cela améliore significativement la complexité temporelle des algorithmes de filtrage et des heuristiques de branchement par rapport à l'approche booléenne.

Malgré ces améliorations, la représentation ensembliste possède une redondance d'information : une arête (i,j) présente dans une solution implique $i \in \mathcal{S}^v_j$ et $j \in \mathcal{S}^v_i$. Cela peut complexifier l'implémentation d'algorithmes de filtrage. Dans le cas d'un algorithme incrémental à grain fin, si l'évènement $i \in \mathcal{S}^v_j$ a déjà été propagé, alors l'évènement $j \in \mathcal{S}^v_i$ doit être ignoré, car l'information a déjà été traitée. Même dans le cas d'algorithmes non-incrémentaux, l'implémentation reste délicate car, si la contrainte `INVERSE_SET` n'a pas encore été propagée, alors on peut avoir un noeud i dans le voisinage d'un autre noeud j sans que j ne figure dans le voisinage de i. Enfin, l'utilisateur et le développeur doivent faire eux-même une dernière montée en abstraction, des variables ensemblistes vers le graphe recherché, ce qui peut être un frein et conduire à des erreurs.

4.4.4 Un modèle dédié

Des variables dédiées ont été conçues pour modéliser ce type de problèmes directement. Une variable de graphe $\mathcal{G}$ a un domaine défini par l'intervalle de graphes $dom(\mathcal{G}) = [\underline{\mathcal{G}}, \overline{\mathcal{G}}]$, qui forme un treillis de graphes. En effet, une représentation en extension serait de taille exponentielle. Une valeur $\mathcal{G}^*$ est un graphe satisfaisant la relation $\underline{\mathcal{G}} \subseteq \mathcal{G}^* \subseteq \overline{\mathcal{G}}$. Le processus de résolution consiste alors à retirer de $\overline{\mathcal{G}}$ certains noeuds et arêtes, ainsi qu'à ajouter à $\underline{\mathcal{G}}$ des noeuds et arêtes présents dans $\overline{\mathcal{G}}$. Ces étapes s'achèvent lorsque la variable graphe est instanciée, *i.e.,* lorsque $\underline{\mathcal{G}} = \overline{\mathcal{G}}$. Puisque le domaine est un intervalle, nous parlerons de borne cohérence (BC) lorsqu'un algorithme de filtrage d'une contrainte ajoute à $\underline{\mathcal{G}}$ tous les éléments (noeuds et arêtes) appartenant à toutes les solutions et retire de $\overline{\mathcal{G}}$ tous les éléments n'appartenant à aucune solution de la contrainte. Cette cohérence est équivalente à la domaine cohérence dans le cas où le graphe est représenté par des variables entières ou booléennes.

Notons qu'une variable graphe est une représentation cohérente d'un graphe. Par exemple, lorsqu'un noeud est supprimé, l'ensemble de ses voisins l'est également. Une variable graphe peut être définie comme étant soit orientée soit non-orientée. Son ensemble de noeuds est noté $\mathcal{V}$, son ensemble d'arêtes (resp. d'arcs) est noté $\mathcal{E}$ (resp. $\mathcal{A}$). Nous utilisons ainsi les notations $G = (V, E)$ et $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ pour se référer respectivement à un graphe donné et à une variable graphe. Comme l'indique la figure 4.9, le modèle par variable de graphe est très naturel.

Domaines initiaux :	**Contraintes :**	**Solution :**
$dom(\mathcal{G}) = [\underline{G}, \overline{G}]$	aucune	$\mathcal{G} = G^*$

FIGURE 4.9 – Modélisation de l'exemple de la figure 4.4 avec une variable de graphe.

L'origine des variables de graphes remonte à la résolution d'un problème de conception de réseaux [LPPRS02], pour lequel il était nécessaire de représenter des chemins. Ce qui est surprenant, c'est que la recherche d'un chemin est l'un des rares cas où la représentation *successeurs* fonctionne bien. L'intérêt de la variable de graphe n'est donc pas limité aux restrictions de la modélisation par des entiers. En fait, les auteurs justifient l'introduction de telles variables par une plus grande facilité pour implémenter des algorithmes de filtrage (basés sur des graphes) et par une volonté de monter en généralité. Plus précisément, les variables utilisées dans [LPPRS02] étaient des chemins. Aussi, Régin a-t-il proposé [Rég04] de généraliser cette approche en utilisant des variables de graphe et en voyant la notion de chemin comme une propriété à satisfaire, *i.e.,* une contrainte. Ce concept a ensuite été développé [DDD05, Doo06] et exploité [QVRDC06, Que06, Lor11]. Un module permettant de manipuler de telles variables a été proposé pour le solveur `Gecode` [DDD05] et, plus tard, dans le solveur `Choco`.

Par la suite, nous considérerons la variable graphe comme la représentation par défaut pour modéliser un graphe. Il s'agit d'une abstraction dont l'implémentation concrète peut bien sûr prendre différentes formes. Nous considérons que les arcs et arêtes sont indexés selon leurs extrémités afin de les parcourir efficacement. De plus, les variables graphes disposent de leur propre système évènementiel permettant de propager des contraintes de graphe de manière incrémentale. Notons que des mécanismes de vues [ST06] et des contraintes de jointures peuvent permettre de bénéficier simultanément de différentes représentations d'un même graphe. Nous utiliserons les termes de graphe *obligatoire* et *potentiel* pour se référer à respectivement $\underline{\mathcal{G}}$ et $\overline{\mathcal{G}}$. Puisque $\underline{\mathcal{G}} \subseteq \overline{\mathcal{G}}$, et par soucis de simplicité, nous utiliserons les notations n et m pour se référer à respectivement $|\overline{\mathcal{V}}|$ et $|\overline{\mathcal{E}}|$ dans nos études de complexité.

Des contraintes sur les graphes

5

Filtrage des contraintes d’arbres et de chemins orientés

Sommaire

5.1 Définition des contraintes de graphe 38

5.1.1 La contrainte TREE 38

5.1.2 La contrainte ANTIARBO 39

5.1.3 La contrainte ARBO 39

5.1.4 La contrainte PATH 40

5.1.5 Relations entre ces contraintes 40

5.2 Filtrage structurel 42

5.2.1 Amélioration de la contrainte TREE 42

5.2.2 Filtrage de la contrainte ARBO 43

5.2.3 Filtrage de la contrainte PATH 44

5.3 Du chemin au circuit 46

5.3.1 La contrainte CIRCUIT 46

5.3.2 Filtrer la contrainte CIRCUIT à partir de la contrainte PATH 46

5.4 Evaluation empirique 47

5.4.1 Passage à l’échelle pour la contrainte tree 48

5.4.2 Recherche d’un circuit Hamiltonien 50

5.5 Conclusion du chapitre 54

De nombreux problèmes combinatoires impliquent de trouver une arborescence ou un chemin dans un graphe orienté. Typiquement, cette préoccupation se retrouve dans des applications informatiques destinées à la logistique, la conception de réseaux, la production industrielle, le séquençage d’ADN ou encore la phylogénie. Dans ce chapitre, nous allons étudier comment exploiter certaines propriétés propres aux graphes dans le but de modéliser efficacement des

problèmes d'arborescences et de chemins. Ce chapitre s'inscrit dans la suite de la thèse de Xavier Lorca [Lor11], répondant à certaines questions restées ouvertes jusqu'alors. En particulier, nous introduisons un algorithme de filtrage linéaire pour la contrainte TREE [FL11].

Dans ce chapitre, nous considérons une unique variable graphe, modélisant un graphe orienté, notée $\mathcal{G}$. Son domaine est défini par $dom(\mathcal{G}) = [\underline{\mathcal{G}}, \overline{\mathcal{G}}]$. Par convention, nous notons également $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ où $\mathcal{V}$ et $\mathcal{A}$ représentent respectivement l'ensemble des noeuds et l'ensemble des arcs du graphe solution. Ils ont pour domaines respectifs $dom(\mathcal{V}) = [\underline{\mathcal{V}}, \overline{\mathcal{V}}]$ et $dom(\mathcal{A}) = [\underline{\mathcal{A}}, \overline{\mathcal{A}}]$ (section 4.4.4). Dans ce chapitre, nous considérons le cas où l'ensemble des noeuds est fixé, *i.e.,* $\underline{\mathcal{V}} = \overline{\mathcal{V}}$. Instancier $\mathcal{G}$ reviendra alors à trouver un ensemble d'arcs satisfaisant les contraintes du modèle.

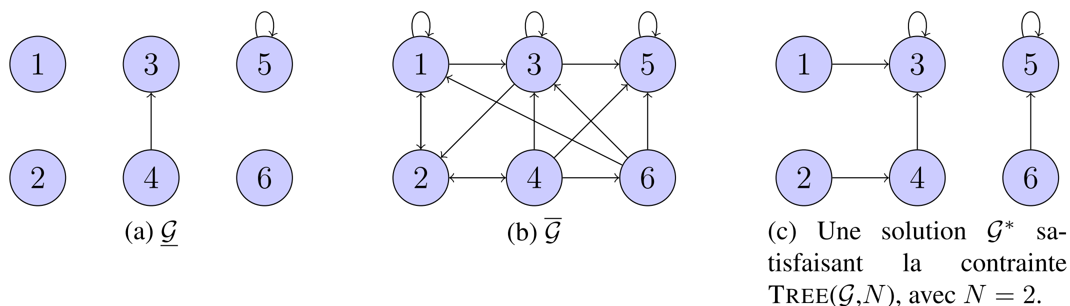
5.1 Définition des contraintes de graphe

Nous allons maintenant définir les contraintes orientées TREE, ANTIARBO, ARBO et PATH. Chaque contrainte sera illustrée par un exemple de solution.

5.1.1 La contrainte TREE

La contrainte TREE$(\mathcal{G}, N)$, introduite dans [BFL05], force une variable graphe orientée $\mathcal{G}$ à former un ensemble de N anti-arborescences. Les noeuds à la racine d'une anti-arborescence sont représentés avec une boucle. Cette contrainte interdit la présence de circuit de taille strictement supérieure à un, mais les boucles sont autorisées puisqu'elles sont utilisées à des fins de modélisation. Elle est illustrée sur la figure 5.1.

Il est intéressant de noter à quel point la définition de cette contrainte a été influencée par le choix de modélisation d'un graphe par sa représentation *successeurs* (section 4.4.1), basée sur des variables entières. Tout d'abord, cela explique le fait que, contrairement à ce que son nom indique, TREE représente des anti-arborescences et non un arbre (concept non-orienté qui n'est pas modélisable par la représentation *successeurs*). On comprend mieux le choix du sens d'orientation retenu, qui n'est pas le plus naturel, ainsi que la présence de boucles, qui est pourtant en contradiction avec le caractère acyclique d'un arbre. En effet, dans une arborescence, chaque noeud a zéro ou plusieurs successeurs, alors que dans une anti-arborescence, un noeud (excepté la racine qui est un cas à part) a exactement un seul successeur. Une variable entière n'ayant qu'une seule valeur, la représentation *successeurs* ne permet que de modéliser les anti-arborescences. De plus, la variable associée à la racine, supposée n'avoir aucun successeur, doit bien prendre une valeur. Pour cette raison, des boucles ont été introduites pour modéliser les racines. Ainsi, chaque valeur d'une variable entière est bien associée à un arc du graphe qu'il représente.

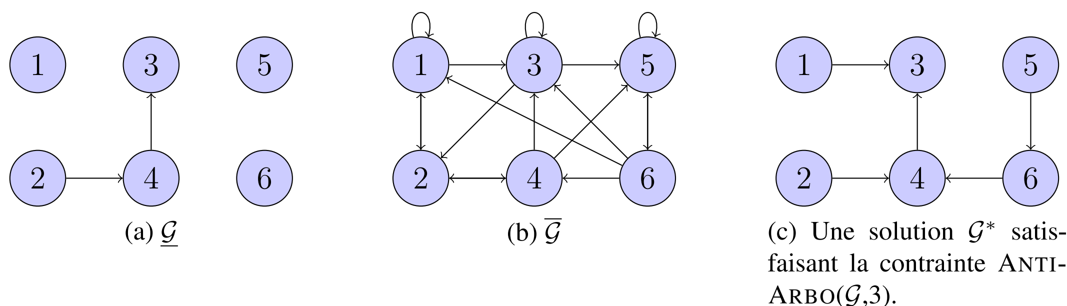


(a) $\underline{\mathcal{G}}$ (b) $\overline{\mathcal{G}}$ (c) Une solution $\mathcal{G}^*$ satisfaisant la contrainte TREE($\mathcal{G}$,N), avec $N = 2$.

FIGURE 5.1 – Illustration de la contrainte TREE($\mathcal{G}$,N).

5.1.2 La contrainte ANTIARBO

La contrainte ANTIARBO$(\mathcal{G}, R)$ force une variable graphe orientée $\mathcal{G}$ à former une anti-arborescence enracinée au noeud R. C'est une variante simplifiée de TREE dans laquelle la cardinalité de la partition est égale à un et le noeud racine, indiqué par une constante, n'a pas d'arc sortant (il n'a donc pas de boucle). Cette contrainte est illustrée sur la figure 5.2.



(a) $\underline{\mathcal{G}}$ (b) $\overline{\mathcal{G}}$ (c) Une solution $\mathcal{G}^*$ satisfaisant la contrainte ANTIARBO($\mathcal{G}$,3).

FIGURE 5.2 – Illustration de la contrainte ANTIARBO$(\mathcal{G}, 3)$.

Cette contrainte présente deux avantages. D'une part, elle s'abstrait des défauts liés à la modélisation de la contrainte TREE. D'autre part, elle met de côté l'aspect partitionnement pour se concentrer sur les propriétés structurelles liées à la notion d'(anti)-arborescence.

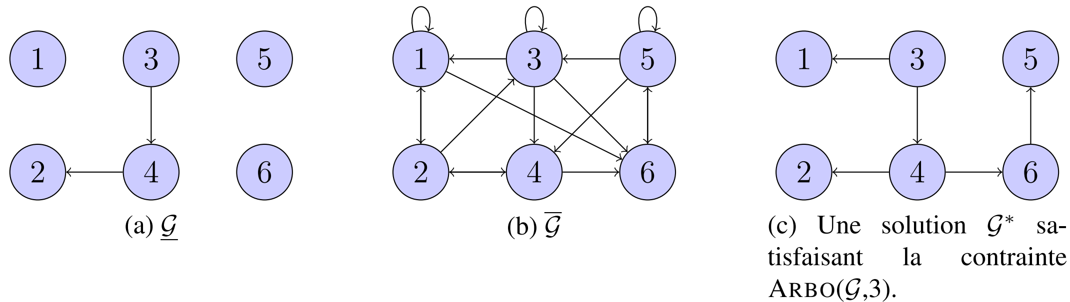
Il est bon de remarquer que le choix d'une constante, au lieu d'une variable, pour désigner la racine n'engendre aucune perte de généralité. En effet, supposons que l'on souhaite que la variable graphe $\mathcal{G}$ forme une anti-arborescence sans pour autant connaitre précisément sa racine, représentée alors par une variable nommée R_{var}. Alors, nous pouvons introduire une variable graphe $\mathcal{G}_2$, identique à $\mathcal{G}$ mais à laquelle on introduit un noeud fictif supplémentaire, dont l'unique prédécesseur correspond à la racine de $\mathcal{G}$. Cela se traduit par la formule suivante :

$$\begin{aligned} \text{ANTIARBO_WITHROOTVAR}(\mathcal{G}, R_{var}) \quad \Leftrightarrow \quad & \text{ANTIARBO}(\mathcal{G}_2, n+1) \\ & \wedge V_{\mathcal{G}_2} = V_{\mathcal{G}} \cup \{n+1\} \\ & \wedge \delta^-_{\mathcal{G}_2}(n+1) = \{R_{var}\} \\ & \textstyle\bigwedge_{i \in [1,n]} \delta^-_{\mathcal{G}_2}(i) = \delta^-_{\mathcal{G}}(i) \end{aligned}$$

A l'évidence, il est possible d'encapsuler cette reformulation dans une méthode pour que l'utilisateur puisse manipuler une signature de ANTIARBO ayant une variable entière pour désigner la racine.

5.1.3 La contrainte ARBO

La contrainte ARBO$(\mathcal{G}, R)$ force une variable graphe orientée $\mathcal{G}$ à former une arborescence enracinée au noeud R. C'est une variante de la contrainte ANTIARBO dans laquelle l'orientation des arcs est inversée (les arcs sont orientés de la racine vers les autres noeuds du graphe). Il en découle que la racine a potentiellement plusieurs arcs sortants mais les feuilles n'en ont pas. Cette orientation est plus naturelle. Cette contrainte est illustrée sur la figure 5.3.

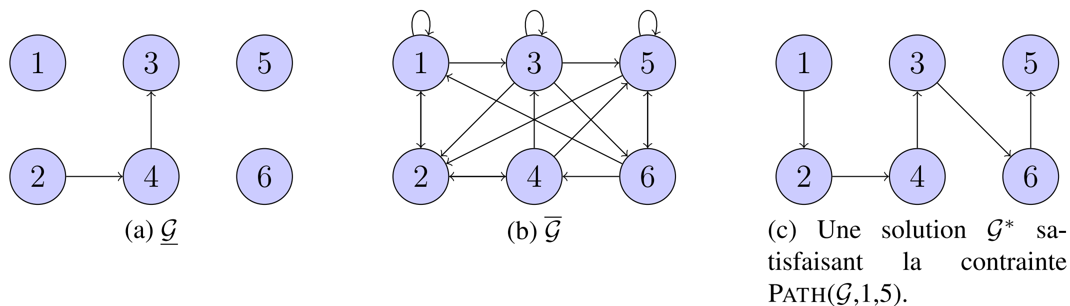


(a) $\underline{\mathcal{G}}$ (b) $\overline{\mathcal{G}}$ (c) Une solution $\mathcal{G}^*$ satisfaisant la contrainte ARBO($\mathcal{G}$,3).

FIGURE 5.3 – Illustration de la contrainte ARBO$(\mathcal{G}, 3)$.

5.1.4 La contrainte PATH

La contrainte PATH$(\mathcal{G}, O, F)$ force une variable graphe orientée $\mathcal{G}$ à former un chemin allant du noeud O (pour origine) au noeud F (pour fin), avec O et F deux constantes telles que $O \neq F$ [CB04]. Cette contrainte est illustrée sur la figure 5.4.



(a) $\underline{\mathcal{G}}$ (b) $\overline{\mathcal{G}}$ (c) Une solution $\mathcal{G}^*$ satisfaisant la contrainte PATH($\mathcal{G}$,1,5).

FIGURE 5.4 – Illustration de la contrainte PATH$(\mathcal{G}, 1, 5)$.

De même que pour la contrainte ANTIARBO, l'utilisation d'entiers (fixés) pour représenter les noeuds de départ et d'arrivée n'engendre aucune perte de généralité.

5.1.5 Relations entre ces contraintes

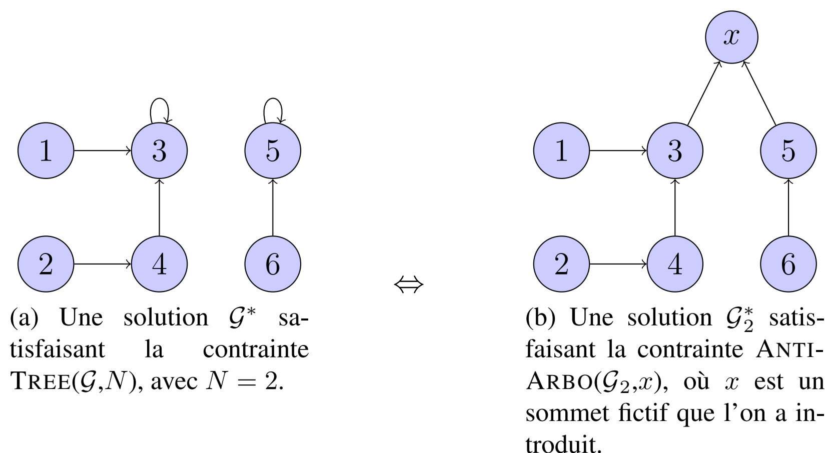
Cette section se consacre à expliquer les relations entre les contraintes TREE, ANTIARBO, ARBO et PATH. Nous allons voir comment passer d'une représentation à l'autre, ce qui présente un double intérêt : améliorer ses connaissances des différentes modélisations possibles et savoir renforcer le filtrage de certaines contraintes à l'aide des algorithmes de filtrage des autres.

Pour passer de TREE à ANTIARBO, il suffit de modifier légèrement le graphe d'entrée de telle sorte que les racines ne soient plus des boucles mais pointent vers un noeud fictif que l'on aura introduit. Pour cela, on introduit une deuxième variable graphe $\mathcal{G}_2$. Le nombre de racines dans $\mathcal{G}$ est alors le nombre de prédécesseurs du noeud fictif dans $\mathcal{G}_2$. Le lien entre TREE et ANTIARBO, illustré sur la figure 5.5, est donné par la relation suivante :

$$
\begin{aligned}
\text{TREE}(\mathcal{G}, N) \quad \Leftrightarrow \quad & \text{ANTIARBO}(\mathcal{G}_2, n+1) && (1)\\
& \wedge\, |\delta^-_{\mathcal{G}_2}(n+1)| = N && (2)\\
& \wedge\, V_{\mathcal{G}_2} = V_{\mathcal{G}} \cup \{n+1\} && (3)\\
& \wedge\, \delta^-_{\mathcal{G}_2}(n+1) = V_{\mathcal{G}}^{(i,i)} && (4)\\
& \textstyle\bigwedge_{i \in [1,n]} \delta^-_{\mathcal{G}_2}(i) = \delta^-_{\mathcal{G}}(i) \setminus \{i\} && (5)
\end{aligned}
$$

La formule est un peu longue, mais finalement assez simple. On peut la décomposer comme suit : filtrage structurel (1), filtrage de cardinalité (2) et jonction entre $\mathcal{G}$ et $\mathcal{G}_2$ $(3, 4, 5)$.

Remarquons que, si cette reformulation préserve bien la sémantique et la cohérence du filtrage structurel de la contrainte TREE, elle peut dégrader son filtrage de cardinalité. Plus précisément, l'évaluation du nombre minimum de racines est ici assez naïf. La décomposition proposée borne inférieurement N par le nombre de boucles obligatoires dans $\mathcal{G}$, *i.e.,* elle impose $N \geq |\{(i, n+1) \in \underline{\mathcal{G}_2}\}|$, avec $|\{(i, n+1) \in \underline{\mathcal{G}_2}\}| = |\{(i, i) \in \underline{\mathcal{G}}\}|$, alors que TREE emploie un algorithme global basé sur la notion de composante puits pour calculer exactement la cardinalité minimum de la partition [BFL05]. Cet algorithme peut bien entendu être trivialement adapté à $\mathcal{G}_2$ et intégré dans un propagateur *ad hoc* supplémentaire.



(a) Une solution $\mathcal{G}^*$ satisfaisant la contrainte TREE($\mathcal{G}$,N), avec $N = 2$.

(b) Une solution $\mathcal{G}_2^*$ satisfaisant la contrainte ANTIARBO($\mathcal{G}_2$,x), où x est un sommet fictif que l'on a introduit.

FIGURE 5.5 – Passage de la contrainte TREE à la contrainte ANTIARBO.

Le passage de ARBO à ANTIARBO, assez évident, est donné par :

$$\text{ARBO}(\mathcal{G}, R) \Leftrightarrow \text{ANTIARBO}(\mathcal{G}^{-1}, R)$$

Le passage d'une contrainte à l'autre n'engendre aucune perte de filtrage.

Les contraintes ARBO et ANTIARBO peuvent être utilisées pour filtrer la contrainte PATH :

$$\text{PATH}(\mathcal{G}, O, F) \Leftrightarrow \text{ARBO}(\mathcal{G}, O) \wedge \text{ANTIARBO}(\mathcal{G}, F)$$

En résumé, les contraintes TREE, ANTIARBO, ARBO et PATH sont fortement corrélées. Moyennant les transformations précédemment introduites, il est possible d'exploiter les algorithmes de filtrage des unes pour filtrer les autres. Ainsi, nous concentrerons dans un premier temps nos efforts sur le filtrage de la contrainte ARBO, qui capture le filtrage structurel de la contrainte TREE. Par la suite, nous exploiterons les propriétés additionnelles relatives à la contrainte PATH afin d'en affiner le filtrage. Enfin, nous étudierons l'exploitation directe de ces résultats pour filtrer la contrainte CIRCUIT.

5.2 Filtrage structurel

Nous allons maintenant étudier les algorithmes de filtrage associés à ces contraintes. Nous tâcherons de caractériser finement les niveaux de cohérence atteints ainsi que la complexité temporelle, au pire cas, des algorithmes de filtrage utilisés.

5.2.1 Amélioration de la contrainte TREE

La contrainte TREE décrite dans [BFL05] adopte une représentation *successeurs* pour représenter le graphe à partitionner. Elle établit ainsi la GAC sur son ensemble de variables (entières). Or, établir la GAC sur une représentation *successeurs* est équivalent à établir la BC sur une variable graphe représentant le même graphe (section 4.4.4). De plus, concernant la variable N, le filtrage aux bornes et le filtrage sur un domaine énuméré sont équivalents [BFL05]. Le niveau de cohérence atteint par le filtrage associé à TREE dans [BFL05] correspond donc à la BC si l'on considère une variable graphe $\mathcal{G}$, au lieu d'un ensemble de variables entières, pour représenter le graphe à partitionner. Si la qualité du filtrage existant était déjà amplement satisfaisante, la complexité de l'algorithme de propagation de la contrainte présentait une marge de progression. En effet, un des axes d'amélioration de la contrainte TREE, identifié dès la première publication de cette contrainte [BFL05], était la complexité temporelle de son filtrage, nécessitant $O(nm)$ opérations dans le pire cas. Ce filtrage peut se décomposer sous la forme de deux propagateurs :

- TREESTRUCT($\mathcal{G}$) assure le filtrage structurel de la contrainte, *i.e.,* le fait que le graphe corresponde bien à une partition en anti-arborescences.

- TREECARD($\mathcal{G}, N$) assure le filtrage de cardinalité de la contrainte, *i.e.,* le fait que cette partition soit de taille N.

Alors que l'algorithme originel assure bien la BC sur TREECARD($\mathcal{G}, N$) (filtrage aux bornes de $\mathcal{G}$ par rapport aux bornes de N, et inversement) en $O(n+m)$, il établit la BC sur TREESTRUCT($\mathcal{G}$) en $O(nm)$, expliquant ainsi la complexité élevée de la contrainte. Ce problème était imputé à l'impossibilité de calculer efficacement l'ensemble des points forts d'articulation d'un graphe orienté [BFL05]. En réponse à leur article, Italiano et. al. ont proposé un schéma de calcul linéaire, *i.e.,* en $O(n + m)$, basé sur la notion de dominance dans un graphe de flot [ILS10]. Au premier abord, on pourrait donc penser qu'utiliser l'algorithme d'Italiano et. al. permette de réduire la complexité du filtrage de TREE à $O(n + m)$. Il n'en est rien. Quand bien même l'ensemble des points forts d'articulation pourrait être calculé en temps contant, la complexité de l'algorithme existant demeurerait $O(nm)$, car il y a $O(n)$ points forts d'articulation, chacun donnant lieu à un traitement en $O(n+m)$ [FL11]. Néanmoins, inspiré par les travaux d'Italiano et. al., nous avons été en mesure de proposer une nouvelle règle de filtrage, bien plus simple que la précédente, qui applique le même niveau de cohérence (à savoir la BC avec une représentation par variable graphe) sur TREESTRUCT($\mathcal{G}$) en $O(n + m)$. La nouvelle règle s'affranchit de la notion de point fort d'articulation, pour ne plus utiliser que celle de noeud dominant. Il est bon de noter que, moyennant la simple transformation décrite en section 5.1.5, TREESTRUCT correspond à ANTIARBO et donc, à l'orientation près, à ARBO, qui est plus naturelle. Aussi, le nouvel algorithme de filtrage structurel pour TREE découle naturellement de l'algorithme de filtrage de la contrainte ARBO, que nous allons introduire. Ces travaux s'inscrivent donc comme une suite naturelle aux travaux de Lorca *et al.* [Lor11].

5.2.2 Filtrage de la contrainte ARBO

Nous nous intéressons maintenant à la contrainte ARBO$(\mathcal{G}, R)$. On peut vérifier que cette contrainte est satisfiable, c'est-à-dire le fait qu'il existe encore au moins une solution, grâce à la proposition 1. En pratique, cela se fait à l'aide d'un parcours sur les noeuds pour vérifier leurs degrés respectifs et d'un parcours sur les arcs pour vérifier que les noeuds soient bien atteignables depuis R. Cette étape se fait donc aisément en $O(n+m)$ opérations.

Proposition 1 *Il existe une solution à la contrainte* ARBO$(\mathcal{G}, R)$ *si et seulement si les conditions suivantes sont vraies :*

- *R n'a pas de prédécesseur :* $|\{(i,R) \in \underline{\mathcal{G}}\}| = 0$
- $\forall j \in \overline{\mathcal{V}} \setminus \{R\}$,
  - *j a un prédécesseur :* $|\{(i,j) \in \underline{\mathcal{G}}\}| \leq 1 \leq |\{(i,j) \in \overline{\mathcal{G}}\}|$
  - *Il existe un chemin de R à j dans $\overline{\mathcal{G}}$*

Preuve : Définition même d'une anti-arborescence (section 3.2). □

Si la contrainte ARBO$(\mathcal{G}, R)$ est bien satisfiable, alors on peut s'intéresser à filtrer la variable $\mathcal{G}$. Filtrer les arcs impossibles se fait grâce au théorème 1. La première condition de filtrage peut être assurée en postant la contrainte de degré $|\delta^-_{\mathcal{G}}(i)| = 1$, pour tout noeud $i \in \overline{\mathcal{V}} \setminus \{R\}$. La seconde nécessite un propagateur dédié, qui fonctionne en deux étapes (voir figure 5.6) :

1. Calculer les relations de dominance immédiate dans le graphe de flot $\overline{\mathcal{G}}(R)$.
2. Eliminer tout arc $(i,j) \in \overline{\mathcal{G}}$ tel que j domine i dans $\overline{\mathcal{G}}(R)$.

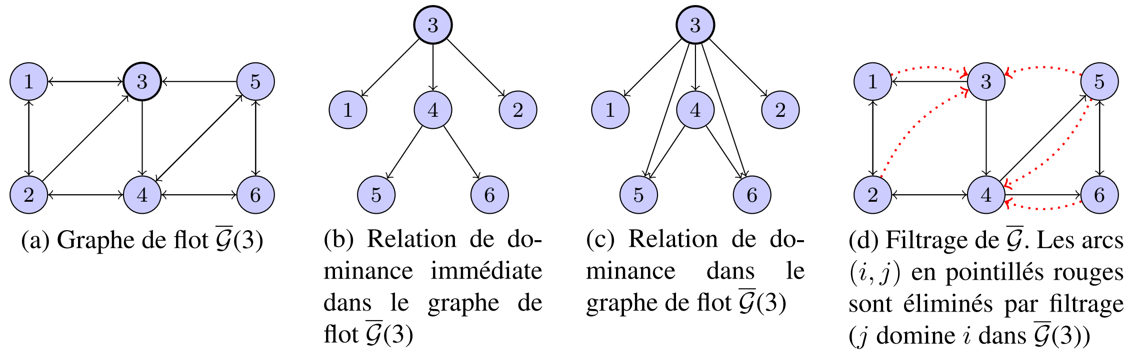
Puisque chaque noeud est dominé par R, tous les arcs de la forme $(i,R) \in \overline{\mathcal{G}}$ seront immédiatement retirés. De même, un noeud se dominant lui-même, les boucles seront également filtrées. Dès que tous les arcs interdits ont été filtrés de $\overline{\mathcal{G}}$, les arcs obligatoires sont automatiquement identifiés et ajoutés à $\underline{\mathcal{G}}$ grâce à la propagation des contraintes sur les degrés entrants des noeuds. En effet, si un arc $(i,j) \in \overline{\mathcal{G}}$ est obligatoire, c'est que tous les arcs $(k,j) \in \overline{\mathcal{G}}$, $k \neq i$, sont interdits (proposition 2). On peut établir la borne cohérence sur ARBO$(\mathcal{G}, R)$ en temps linéaire (théorème 2).

Théorème 1 *Un arc $(i,j) \in \overline{\mathcal{G}}$ n'appartient à aucune solution de* ARBO$(\mathcal{G}, R)$ *si et seulement si l'une des conditions suivantes est vraie :*

- *Il existe un noeud $k \neq i$ tel que $(k,j) \in \underline{\mathcal{G}}$.*
- *j domine i dans le graphe de flot $\overline{\mathcal{G}}(R)$.*

Preuve : Supposons qu'il existe un arc $(k,j) \in \underline{\mathcal{G}}$. Par définition d'une arborescence (section 3.2), chaque noeud a au plus un prédécesseur, donc tout arc $(i,j) \in \overline{\mathcal{G}}$ tel que $i \neq k$ n'appartient à aucune solution de la contrainte. Supposons maintenant qu'il n'existe pas de prédécesseur de j dans $\underline{\mathcal{G}}$. Soit l'arc $(i,j) \in \overline{\mathcal{G}}$ tels que tous les chemins de R à i passent par j. Si $i = j$, alors (i,j) est impossible car les boucles, formes de cycles, sont interdites (section 3.2). Sinon, par la proposition 1, il existera dans chaque solution un chemin allant de R à i. Chacun de ces chemin passera donc par le noeud j avant d'atteindre i. Utiliser l'arc (i,j) créera donc un circuit, ce qui est prohibé.

Supposons maintenant que les deux conditions du théorème soient fausses : j n'a aucun prédécesseur dans $\underline{\mathcal{G}}$ et il existe au moins un chemin allant de R à i sans passer par j dans $\overline{\mathcal{G}}$. Forcer l'arc (i,j) respecte la proposition 1, donc il existe au moins une solution contenant l'arc (i,j). Le théorème caractérise donc exactement l'ensemble des arcs impossibles. □



(a) Graphe de flot $\overline{\mathcal{G}}(3)$

(b) Relation de dominance immédiate dans le graphe de flot $\overline{\mathcal{G}}(3)$

(c) Relation de dominance dans le graphe de flot $\overline{\mathcal{G}}(3)$

(d) Filtrage de $\overline{\mathcal{G}}$. Les arcs (i, j) en pointillés rouges sont éliminés par filtrage (j domine i dans $\overline{\mathcal{G}}(3)$)

FIGURE 5.6 – Illustration du filtrage de ARBO$(\mathcal{G}, 3)$ basé sur la notion de dominance.

Proposition 2 *Si un arc* $(i, j) \in \overline{\mathcal{G}}$ *appartient à toutes les solutions de* ARBO$(\mathcal{G}, R)$ *alors, nécéssairement,* $\forall k \in \overline{\mathcal{V}} \setminus \{j\}$*,* (k, j) *n'appartient à aucune solution d'*ARBO$(\mathcal{G}, R)$*.*

Preuve : Par définition d'une arborescence (section 3.2), chaque noeud a au plus un prédécesseur. □

Théorème 2 *On peut établir la borne cohérence sur* ARBO$(\mathcal{G}, R)$ *en* $O(n + m)$ *opérations.*

Preuve : Les contraintes de degrés sont propagée en $O(n)$. Le propagateur basé sur la notion de dominance s'exécute en temps linéaire : la première phase s'effectue en $O(n + m)$ [BKRW98, AHLT99] ; la seconde, numérote chaque noeud de façon *pre-ordre* et *post-ordre*. Cela nécessite un pré-calcul en $O(n + m)$. Ensuite, elle effectue un test de dominance pour chaque arc de $\overline{\mathcal{G}}$ (simple comparaison des numérotations). Ceci requiert donc $O(m)$. Dans l'ensemble, la BC est établie sur ARBO$(\mathcal{G}, R)$ en $O(n + m)$ opérations [FL11]. □

5.2.3 Filtrage de la contrainte PATH

Une des caractéristiques d'un chemin élémentaire est qu'il ne contient pas de circuit. Pour filtrer en fonction de cette propriété, Caseau et Laburthe ont proposé un algorithme incrémental très efficace, propageant la découverte de chaque arc obligatoire en temps constant [CL97]. De plus, la contrainte PATH implique que les arcs du graphe aient tous des origines et des destinations différentes. Le modèle classique pour filtrer la contrainte PATH est donc la combinaison d'une contrainte ALLDIFFERENT [Rég94] et d'une contrainte d'élimination de circuit [CL97], portant sur les successeurs des noeuds de $\mathcal{G}$ (sauf F qui n'a pas de successeur).

Dominance et accessibilité

Comme indiqué en section 5.1.5, la contrainte PATH$(\mathcal{G}, O, F)$ est équivalente à la conjonction des contraintes ARBO$(\mathcal{G}, O)$ et ANTIARBO$(\mathcal{G}, F)$. Ainsi, appliquer ces dernières est suffisant pour garantir le respect de la contrainte PATH. De plus, le filtrage aux bornes de ces contraintes assure les propriétés suivantes :

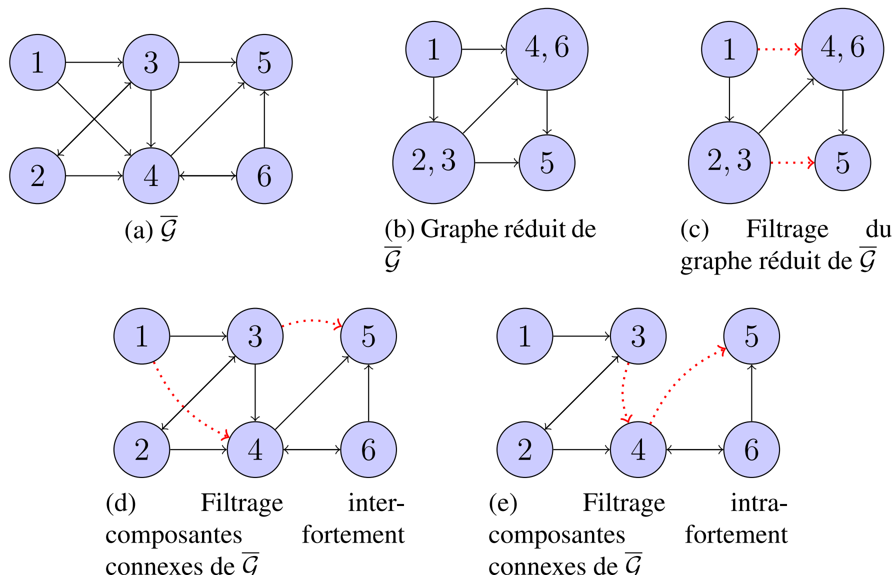
- Chaque noeud dans $\overline{\mathcal{V}} \setminus \{O\}$ a au moins un prédécesseur dans $\overline{\mathcal{G}}$.
- Chaque noeud dans $\overline{\mathcal{V}} \setminus \{F\}$ a au moins un successeur dans $\overline{\mathcal{G}}$.
- Il existe un chemin dans $\overline{\mathcal{G}}$ allant de O à chaque noeud de $\overline{\mathcal{V}} \setminus \{O\}$.
- Il existe un chemin dans $\overline{\mathcal{G}}$ allant de chaque noeud de $\overline{\mathcal{V}} \setminus \{F\}$ à F.

- Pour tout arc $(i, j) \in \overline{\mathcal{A}}$, il existe au moins chemin allant de O à i qui ne passe pas par j.
- Pour tout arc $(i, j) \in \overline{\mathcal{A}}$, il existe au moins chemin allant de j à F qui ne passe pas par i.
- $\underline{\mathcal{G}}$ ne contient pas de cycle, et ajouter un arc de $\overline{\mathcal{A}}$ dans $\underline{\mathcal{A}}$ n'en créé pas.

Il est bon de noter que ce filtrage s'effectue en $O(n + m)$, contrairement à l'utilisation de la contrainte DOMREACHABILITY qui nécessite $O(nm)$ [QVRDC06] à cause de son maintien explicite de la fermeture transitive de $\mathcal{G}$. De plus, ce filtrage est incomparable avec celui de la contrainte ALLDIFFERENT, on peut donc les combiner. En revanche, l'élimination des cycles par un propagateur dédié n'est plus nécessaire.

Composantes fortement connexes et graphe réduit

Afin de filtrer la contrainte PATH($\mathcal{G}$, O, F), Cambazard et Bourreau ont introduit le propagateur REDUCEDPATH($\mathcal{G}$, O, F) qui filtre $\mathcal{G}$ en assurant la propriété que le graphe réduit des composantes fortement connexes de $\mathcal{G}$ forme un chemin Hamiltonien allant de O à F [CB04]. En d'autres termes, REDUCEDPATH transpose PATH sur le graphe réduit de $\mathcal{G}$. Le filtrage se décompose en deux étape : le filtrage inter-composantes retire les arcs entre deux composantes qui ne sont pas successives dans l'unique chemin Hamiltonien du graphe réduit ; le filtrage intra-composantes filtre les arcs qui ne permettent pas d'entrer dans la composante, parcourir tous ses noeuds et quitter la composante. Alors que le filtrage inter-composante est linéaire (problème polynomial car le graphe réduit ne contient pas de circuit), le filtrage intra-composante est NP-complet. On doit donc se contenter de règles *ad hoc*. Par exemple, si une composante d'au moins deux noeuds n'a qu'un point d'entrée, alors les successeurs de ce noeud doivent rester dans la composante (sinon on ne visiterait pas les autres noeuds de la composante). D'autres règles sont données dans [FL12]. Une illustration du filtrage est fournie sur la figure 5.7.



(a) $\overline{\mathcal{G}}$ (b) Graphe réduit de $\overline{\mathcal{G}}$ (c) Filtrage du graphe réduit de $\overline{\mathcal{G}}$ (d) Filtrage inter-composantes fortement connexes de $\overline{\mathcal{G}}$ (e) Filtrage intra-composantes fortement connexes de $\overline{\mathcal{G}}$

FIGURE 5.7 – Illustration du filtrage de REDUCEDPATH($\mathcal{G}$, 1, 5), basé sur le graphe réduit de $\overline{\mathcal{G}}$. Les arcs en pointillés rouges sont éliminés par filtrage.

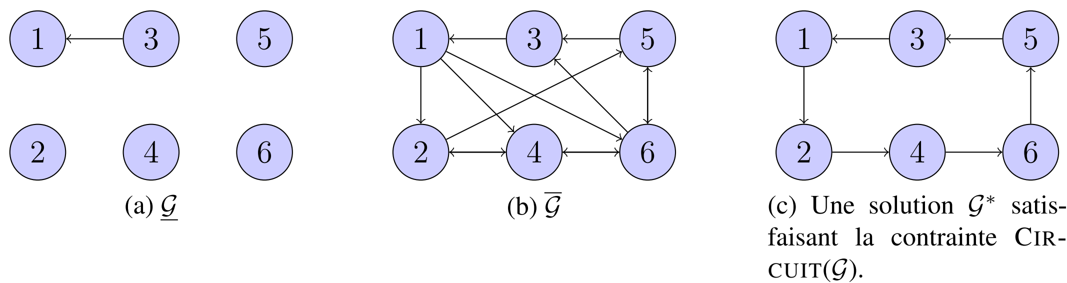
En résumé, on peut filtrer PATH à l'aide des contraintes ALLDIFFERENT, ARBO, ANTI-ARBO et REDUCEDPATH, qui filtrent $\mathcal{G}$ en fonction de propriétés bien différentes, respectivement basées sur la notion de couplage, (anti-)arborescences et connexité forte.

5.3 Du chemin au circuit

Cette section précise les liens entre une contrainte de chemin et une contrainte de circuit. Nous montrons comment utiliser les algorithmes de filtrage de PATH pour filtrer CIRCUIT. Plus précisément, nous montrons que certains algorithmes *ad hoc* utilisés pour filtrer la contrainte CIRCUIT proviennent de propriétés utilisées pour PATH.

5.3.1 La contrainte CIRCUIT

La contrainte CIRCUIT$(\mathcal{G})$ force une variable graphe orientée $\mathcal{G}$ à former un circuit. Cette contrainte est illustrée sur la figure 5.8. Son algorithme de filtrage classique consiste à appliquer conjointement la contrainte ALLDIFFERENT sur les successeurs des noeuds et l'élimination des sous-circuits de [CL97]. Voyons comment améliorer ce filtrage.



(a) $\underline{\mathcal{G}}$ (b) $\overline{\mathcal{G}}$ (c) Une solution $\mathcal{G}^*$ satisfaisant la contrainte CIRCUIT$(\mathcal{G})$.

FIGURE 5.8 – Illustration de la contrainte CIRCUIT.

5.3.2 Filtrer la contrainte CIRCUIT à partir de la contrainte PATH

Le passage de PATH à CIRCUIT est donné par la relation :

$$\text{PATH}(\mathcal{G}, O, F) \Leftrightarrow \text{CIRCUIT}(\mathcal{G} \cup \{(F, O)\})$$

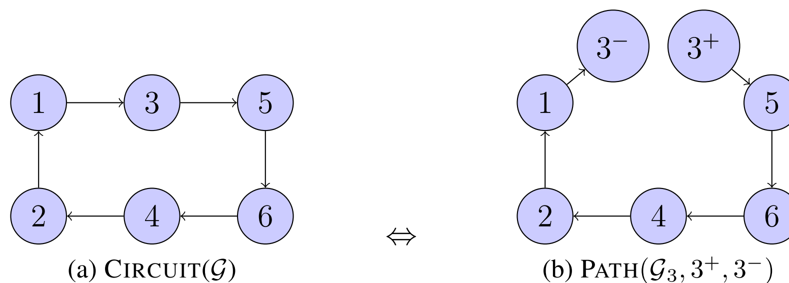
En conséquence, on peut filtrer la contrainte PATH en appliquant la contrainte CIRCUIT sur un autre graphe, $\mathcal{G}_2$, défini par la contrainte $\mathcal{G}_2 = \mathcal{G} \cup \{(F, O)\}$. Réciproquement, dès lors qu'il existe un arc obligatoire $(i, j) \in \underline{\mathcal{A}}$, la contrainte CIRCUIT$(\mathcal{G})$ peut être propagée en appliquant les algorithmes de la contrainte PATH comme suit :

$$\text{CIRCUIT}(\mathcal{G}) \Rightarrow \bigwedge_{(i,j) \in \underline{\mathcal{A}}} \text{PATH}(\mathcal{G} \setminus \{i, j\}, j, i)$$

Cette règle de filtrage peut être généralisée au cas où $\underline{\mathcal{G}}$ ne contient pas d'arc obligatoire. Etant donnée la variable graphe $\mathcal{G}$ et un sommet $i \in \underline{\mathcal{V}}$, notons $\mathcal{G}_i$ le graphe obtenu en dupliquant le sommet i en i^- et i^+, respectivement associés aux arcs entrant et sortant de i dans $\mathcal{G}$ (voir figure 5.9). Dans ce cas, on obtient la relation suivante :

$$\textbf{CIRCUIT}(\mathcal{G}) \Rightarrow \bigwedge_{i \in \underline{\mathcal{V}}} \textbf{PATH}(\mathcal{G}_i, i^+, i^-)$$

Etant donné qu'un arc ne peut pas être obligatoire sans que ses extrémités ne soient des sommets obligatoires, la formule ci-dessus domine celle qui la précède. Nous la retenons donc pour appliquer les algorithmes de filtrages de la contrainte PATH sur la contrainte CIRCUIT.



(a) CIRCUIT($\mathcal{G}$)

(b) PATH($\mathcal{G}_3, 3^+, 3^-$)

FIGURE 5.9 – Passage de la contrainte CIRCUIT à la contrainte PATH.

Choix du noeud à dupliquer : Si nous disposions d'un algorithme de filtrage complet, nous pourrions nous contenter d'un seul noeud pris aléatoirement pour effectuer cette transformation. Malheureusement, pour propager ces contraintes dans un temps raisonnable, nous devrons nous contenter d'un filtrage partiel. Dès lors, le choix du noeud à dupliquer va avoir un impact sur le filtrage réalisé, si bien que l'on peut appliquer cette transformation sur chacun des noeuds du graphe pour obtenir un maximum de filtrage. Vraisemblablement, utiliser n contraintes PATH (une pour chaque noeud) engendrerait un trop lourd surcoût algorithmique pour que le filtrage soit rentable. Aussi proposons-nous de n'appliquer les algorithmes de filtrage de PATH que sur un seul noeud (le noeud 3 sur la figure 5.9). Nous retenons deux manières d'appliquer cette transformation :

Statique : Opérer statiquement la transformation à la pose du modèle, en créant une deuxième variable graphe $\mathcal{G}_i$, image de $\mathcal{G}$ après duplication d'un noeud $i \in \underline{\mathcal{V}}$, et en posant une contrainte PATH($\mathcal{G}_i, i^+, i^-$).

Dynamique : Opérer dynamiquement la transformation dans un propagateur dédié à la contrainte CIRCUIT, qui applique directement les algorithmes de filtrage de PATH. Ce propagateur effectue la transformation sur un graphe interne, qui n'est pas une variable.

Dans les deux cas, la transformation est fondamentalement la même. Néanmoins, l'implémentation varie. La première est sans nul doute la plus simple, mais on peut lui reprocher l'introduction d'une nouvelle variable. Un autre avantage de l'approche dynamique est l'opportunité de diversifier le filtrage, en changeant à chaque fois le noeud sur lequel effectuer la transformation. On diminue donc le risque d'avoir mal choisi le noeud de séparation.

L'intérêt de pouvoir changer de noeud séparateur, en le choisissant aléatoirement, a été mis en lumière par Schulte, qui a illustré la notion de monotonie d'un propagateur à l'aide d'un algorithme *ad hoc* pour filtrer la contrainte CIRCUIT [ST09]. Par la suite, Glenn Francis et Stuckey ont amélioré, et expliqué, cet algorithme [FS14]. Cependant, leur algorithme est en fait dominé par la conjonction de REDUCEDPATH, ARBO et ANTIARBO.

5.4 Evaluation empirique

Cette section est dédiée à l'évaluation empirique de différents algorithmes de filtrage vus dans ce chapitre. Dans un premier temps, nous évaluons notre contribution sur la contrainte TREE, à savoir l'algorithme linéaire basé sur les noeuds dominants [FL11] (section 5.2.2). Le but de cette étape est de valider que ce gain sur la complexité théorique de l'algorithme améliore bien les résultats en pratique. Dans un second temps, nous nous intéresserons à la contrainte CIRCUIT, qui est très largement utilisée. L'objectif de cette étude est de déterminer le meilleur

schéma pour filtrer cette contrainte. Pour cela, nous étudierons l'impact des propagateurs de PATH (ARBO, ANTIARBO et REDUCEDPATH) sur les résultats obtenus. Nous comparerons également le cas où la transformation de CIRCUIT vers PATH est statique avec le cas dynamique (voir section 5.3.2). Enfin, nous nous comparons aux résultats de l'état de l'art, basé sur des explications [FS14], afin d'en fournir une étude critique.

Nos implémentations sont basées sur le solveur `Choco`, en Java. Les tests ont été exécutés sur une machine Mac mini 4.1, avec un processeur Intel core 2 duo à 2.4 Ghz.

5.4.1 Passage à l'échelle pour la contrainte tree

Nous commençons notre étude expérimentale en évaluant empiriquement l'intérêt du nouvel algorithme de filtrage, basé sur la notion de dominance dans un graphe de flot, pour les contraintes ARBO, ANTIARBO et TREE. Nous retenons pour notre étude la contrainte TREE. Nous étudions donc le problème de partitionner un graphe orienté en anti-arborescences. Pour cela, nous comparons sur la table 5.1 trois manières de propager la contrainte TREE : la décomposition usuelle de la contrainte TREE (rappelée dans la suite de cette section) ; la contrainte globale originelle [BFL05], basée sur la notion de connexité forte ; et la contrainte globale revisitée [FL11], qui est basée sur la notion de dominance dans un graphe de flot. Pour les trois approches, l'heuristique de branchement sélectionnera aléatoirement un arc dans $\overline{\mathcal{A}} \setminus \underline{\mathcal{A}}$ pour l'ajouter à $\underline{\mathcal{A}}$. En cas d'échec, cet arc sera donc retiré de $\overline{\mathcal{A}}$.

Nous générons aléatoirement un ensemble de graphes orientés ayant de 50 à 300 noeuds ainsi que différentes densités en nombre d'arc. Les arcs sont ajoutés aléatoirement, avec une distribution uniforme, jusqu'à ce que la densité souhaitée soit atteinte. Chacun de ces graphes représente une instance. Pour chaque instance, seuls certains noeuds (ceux qui ont une boucle dans le graphe d'entrée) peuvent former la racine d'une arborescence. De plus, puisque nous nous intéressons ici au filtrage structurel, le domaine de N (indiquant le nombre d'anti-arborescences) est initialisé à $[1, n]$. Bien que le problème soit polynomial, l'approche décomposée peut peiner à la résoudre, car elle ne dispose pas d'un filtrage puissant. A l'inverse, les approches basées sur une contrainte globale pourront le résoudre sans rencontrer le moindre échec dans l'arbre de recherche, mais cela aura un coût algorithmique. Le but de cette expérience est de voir ce qui est le plus rentable en pratique.

L'approche décomposée utilise, pour chaque noeud i du graphe, trois variables $b_i \in \{false, true\}$, $v_i \in [0, n-1]$ et $r_i \in [0, n-1]$ pour représenter respectivement si le noeud i est une racine, le successeur de i et la distance du noeud i à la racine de l'anti-arborescence à laquelle i appartient. On décompose alors TREE($\mathcal{G}$,N) par le CSP suivant :

$$\forall i \in [1;n],\ v_i = j \land i \neq j \Rightarrow r_i > r_j \quad (1)$$
$$\forall i \in [1;n],\ b_i \Leftrightarrow v_i = i \quad (2)$$
$$N = \sum_{i=1}^{n} b_i \quad (3)$$

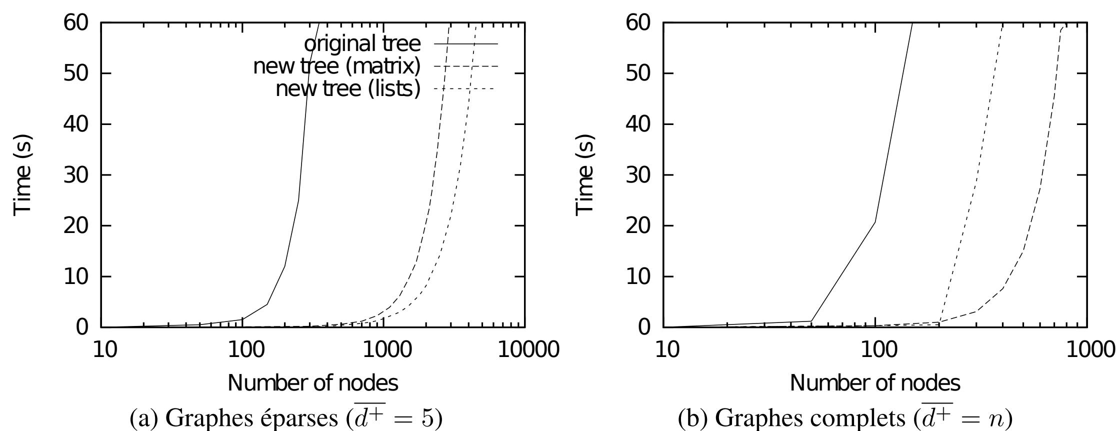
Regardons maintenant attentivement les résultats donnés en table 5.1. La table indique le temps moyen de résolution et le nombre d'instances résolues, sur un total de 100, pour chaque configuration : n indique le nombre de noeuds et $d+$ indique le degré sortant moyen des noeuds dans le graphe d'entrée. Chaque exécution est soumise à une limite de temps d'une minute. Premièrement, un grand nombre d'instances, même de petite taille, n'a pas pu être résolu dans le temps imparti par la décomposition. On notera en revanche que, lorsqu'elle parvient à résoudre l'instance considérée, le temps de résolution est assez faible en moyenne. La décomposition est donc une approche avec une forte variabilité en performance. A l'opposé, les contraintes globales offrent des performances plus stables. En ce sens, elles sont plus fiables. Deuxièmement, on observe pour l'algorithme originel une vraie difficulté à résoudre les instances de grande

taille. Il ne lui est pas possible de partitionner un graphe complet de 150 noeuds, à cause de son coût algorithmique et non de la combinatoire de l'instance. Enfin, on remarque que le nouvel algorithme améliore très nettement son prédécesseur, permettant alors de traiter des graphes complets de 300 noeuds en quelques secondes.

Instances		Décomposition		TREE [BFL05]		TREE [FL11]	
n	$d+$	temps (s)	nb. résolues	temps (s)	nb. résolues	temps (s)	nb. résolues
50	5	1.1	80	0.5	**100**	**0.0**	**100**
	20	0.1	100	1.3	**100**	**0.0**	**100**
	50	0.1	100	1.2	**100**	**0.0**	**100**
150	5	2.6	60	4.5	**100**	**0.0**	**100**
	20	3.1	80	11.3	**100**	**0.0**	**100**
	50	0.6	87	25.3	**100**	**0.1**	**100**
	150	2.8	100	–	0	**0.3**	**100**
300	5	**0.1**	20	51.6	**100**	**0.1**	**100**
	20	0.4	47	–	0	**0.2**	**100**
	50	1.7	53	–	0	**0.5**	**100**
	300	17.7	77	–	0	**2.4**	**100**

TABLE 5.1 – Comparaison d'une approche par décomposition, de la contrainte globale TREE [BFL05] et de sa version revisitée [FL11] pour partitionner un graphe orienté en anti-arborescences.

Afin d'appréhender les nouvelles limites de la contrainte TREE, nous repoussons encore un peu l'échelle des problèmes traités. Nous considérons dans cette étude deux implémentations d'une variable graphe. La première est basée sur une matrice d'adjacence (utilisant des `BitSet`). La seconde est basée sur une liste d'adjacence. Nous précisons qu'en utilisant des structures de données théoriquement optimales (voir [LSSL13]), nous obtenons en pratique un compromis entre ces deux tendances. Comme le montre la figure 5.10, le nouvel algorithme améliore significativement la capacité de la contrainte TREE à passer à l'échelle. Il permet de traiter des instances ayant de 800 à 5000 noeuds en moins d'une minute. L'influence des structures de données utilisées sur les performances témoigne de l'intérêt de la montée en abstraction offerte pas la variable graphe, permettant ainsi de passer simplement d'une implémentation à une autre.



(a) Graphes éparses ($\overline{d^+} = 5$) (b) Graphes complets ($\overline{d^+} = n$)

FIGURE 5.10 – Etude du passage à l'échelle de la contrainte TREE.

5.4.2 Recherche d'un circuit Hamiltonien

Pour illustrer les règles de filtrage structurel vues en sections 5.2 et 5.3.2, nous nous intéressons au problème de la recherche d'un circuit Hamiltonien dans un graphe orienté, tel que le coût de son arc le plus cher est minimal. Ce problème a été récemment exploité pour mettre en avant l'utilisation d'explications pour la contrainte CIRCUIT [FS14]. Nous reprenons un échantillon représentatif des instances proposées, à savoir $1,500$ instances ayant de 15 à 60 noeuds chacune. Nous reprenons leur protocole de test ; *i.e.,* chaque instance n'est jouée qu'une seule fois. Puisque nous partons d'instances au format MiniZinc, le circuit n'est pas représenté par une variable de graphe mais par n variables entières. Le but de notre évaluation empirique est multiple :

1. Mesurer l'apport du filtrage de ARBO, ANTIARBO et REDUCEDPATH.
2. Mesurer l'apport de la transformation dynamique de PATH vers CIRCUIT proposée en section 5.3.2.
3. Identifier la meilleure manière de propager la contrainte CIRCUIT.
4. Comparer notre approche à l'état de l'art [FS14], basé sur des explications.

Apport du filtrage structurel

Pour filtrer la contrainte CIRCUIT, nous utilisons une contrainte ALLDIFFERENT (GAC) et le propagateur incrémental d'élimination des sous-tours de [CL97]. Cette configuration simple mais efficace est notée `Basic`. Nous considérons ensuite l'ajout des propagateurs de la contrainte PATH, via la transformation dynamique décrite en section 5.3.2 :

- `AR` : ajout de ARBO et ANTIARBO, en dupliquant aléatoirement un noeud du graphe.
- `RPR` : ajout de REDUCEDPATH, en dupliquant aléatoirement un noeud du graphe.
- `ARPR` : ajout de `AR` et `RPR`
- `ARPA` : ajout de ARBO, ANTIARBO et REDUCEDPATH, en dupliquant chaque noeud du graphe.

Nous considérons une heuristique statique lexicographique pour explorer l'arbre de recherche et une limite de temps de cinq minutes.

	Modèle	Taille d'instance 15	30	60
Nombre d'instances résolues à l'optimum	`Basic`	500	500	87
	`AR`	500	500	**477**
	`RPR`	500	500	214
	`ARPR`	500	500	476
	`ARPA`	500	500	460
Nombre moyen d'échecs	`Basic`	62.6	6125.2	3125393.5
	`AR`	9.6	39.1	96601.0
	`RPR`	20.1	386.0	1366790.7
	`ARPR`	9.5	38.2	83541.4
	`ARPA`	9.3	37.5	**8895.0**
Temps moyen de résolution (s)	`Basic`	0.09	0.78	253.55
	`AR`	0.06	0.14	**18.34**
	`RPR`	0.06	0.26	183.44
	`ARPR`	0.06	0.15	18.51
	`ARPA`	0.11	0.32	31.79

TABLE 5.2 – Apport du filtrage structurel pour filtrer CIRCUIT.

Les résultats de notre étude sont donnés sur la table 5.2. On observe une réelle difficulté à résoudre les instances de taille 60, alors que les autres sont résolues quelle que soit l'approche retenue. On remarque également des différences sur les nombres d'échecs rencontrés ainsi que sur les temps de résolution. Tout d'abord, l'approche simpliste (`Basic`) rencontre en moyenne significativement plus d'échecs que les autres approches. Le propagateur REDUCEDPATH (RPR) permet de réduire le nombre d'échecs, mais il est nettement plus intéressant d'utiliser ARBO et ANTIARBO (AR). La conjonction de ces propagateurs (ARPR) n'engendre qu'une légère diminution du nombre d'échecs rencontrés, sans grande incidence sur le temps de calcul. En revanche, le renforcement du filtrage basé sur la duplication de chaque noeud du graphe (ARPA), au lieu d'un seul pris aléatoirement, n'est pas rentable. Bien qu'il diminue significativement le nombre d'échecs rencontré sur les instances de taille 60, le surcoût algorithmique qu'il engendre est trop important. Puisqu'il correspond à l'ajout de n propagateurs ARBO, ANTIARBO et REDUCEDPATH, ce résultat n'est pas surprenant. Au contraire, on aurait pu s'attendre à des performances encore moins bonnes.

En conclusion, nous recommandons la configuration AR, basée sur la contrainte TREE [FL11], qui améliore significativement le filtrage de la contrainte CIRCUIT. Ce résultat valide donc la pertinence de la première contribution de cette thèse. Nous retenons également la configuration ARPR qui permet d'établir une propriété supplémentaire (REDUCEDPATH) sur le domaine de $\mathcal{G}$ pour un surcoût algorithmique quasiment négligeable.

Remise en question de l'impact du noeud de séparation

Il a été montré que choisir aléatoirement le noeud à dupliquer pour passer de la contrainte PATH à la contrainte CIRCUIT apportait des améliorations significatives sur leur algorithme de filtrage *ad hoc* [ST09, FS14], qui est une forme dégradée du filtrage structurel étudié ici. Nous allons voir si ce résultat reste valide pour la conjonction des propagateurs ARBO, ANTIARBO et REDUCEDPATH, qui domine strictement leur propagateur, où si l'aléatoire ne venait que pallier un défaut dans leur algorithme. Nous comparons donc nos deux meilleures configurations, à savoir AR et ARPR avec deux nouvelles variantes ne faisant plus appel à l'aléatoire :

- AF : ajout de ARBO et ANTIARBO, en dupliquant toujours le premier noeud du graphe.
- ARPF : ajout de ARBO, ANTIARBO et REDUCEDPATH, en dupliquant toujours le premier noeud du graphe.

Là encore, nous considérons une heuristique statique lexicographique pour explorer l'arbre de recherche et une limite de temps de cinq minutes.

	Modèle	Taille d'instance 15	30	60
Nombre d'instances résolues à l'optimum	ARPR	500	500	476
	ARPF	500	500	475
	AR	500	500	**477**
	AF	500	500	470
Nombre moyen d'échecs	ARPR	9.5	38.2	**83541.4**
	ARPF	9.4	38.0	88835.3
	AR	9.6	39.1	96601.0
	AF	10.6	43.3	114828.4
Temps moyen de résolution (s)	ARPR	0.06	0.15	18.5
	ARPF	0.06	0.15	19.5
	AR	0.06	0.14	**18.3**
	AF	0.06	0.14	22.7

TABLE 5.3 – Apport de l'aléatoire pour passer de PATH à CIRCUIT.

Les résultats de notre comparaison sont donnés sur la table 5.3. On s'aperçoit alors que le choix du noeud opérant la transformation de PATH vers CIRCUIT ne change pas significativement les résultats entre les configurations ARPR et ARPF. Les nombres d'instances résolues à l'optimum, les nombres moyens d'échecs et les temps moyens de résolution sont équivalents. En revanche, un observe une légère perte de AR vers AF. Précisons que cette baisse est nettement inférieure aux écarts indiqués dans [ST09, FS14]. On en déduit que les gains obtenus par cette littérature sont dus à la nature incomplète de leurs propagateurs, dont le filtrage est fortement lié au choix du noeud dupliqué. En utilisant de meilleurs algorithmes de filtrage, ce choix importe assez peu. En conclusion, plus le filtrage est puissant, moins le choix du noeud à dupliquer est important. Ce résultat est important, car il signifie que l'on peut utiliser une transformation statique de la contrainte PATH vers la contrainte CIRCUIT.

Comparaison à l'état de l'art

Nous comparons nos résultats (Choco, muni de la configuration AR qui offrait les meilleures performances) avec ceux de la dernière publication du domaine [FS14] : l'état de l'art basé sur les explications [FS14] (Chuffed) et les résultats du solveur Gecode (version 4.0.0) qui leur a servi de référence. Dans les trois cas, l'heuristique de branchement est lexicographique et chaque résolution est soumise à une limite de dix minutes de résolution. Les résultats sont affichés sur la table 5.4. On remarque que Choco se positionne bien loin devant Gecode, avec des performances comparables à celles de Chuffed. Pour les petites instances ($n < 60$), Choco est plus performant que Chuffed, que ce soit en termes de nombre d'échecs ou en termes de temps de calcul. En revanche, sur les instances de taille 60, il résout 2% d'instances en moins que l'approche basée sur des explications. Ces instances non résolues par Choco engendrent une augmentation des moyennes sur le nombre d'échecs rencontrés et sur le temps de résolution.

Dans l'ensemble, Choco, qui dispose d'un filtrage puissant, domine Gecode, dont les niveaux de cohérence sont plus faibles, et est relativement compétitif avec Chuffed, qui repose sur l'explication d'un filtrage léger. Les performances de Choco viennent ainsi tempérer les résultats de [FS14], qui ne s'étaient pas comparé au filtrage état de l'art [FL11].

	Modèle	Taille d'instance 15	30	60
Nombre d'instances résolues à l'optimum	Gecode	500	425	130
	Chuffed	500	**500**	**487**
	Choco	500	**500**	478
Nombre moyen d'échecs (en milliers)	Gecode	3.97	5362.00	17821.00
	Chuffed	0.04	1.80	**80.50**
	Choco	**0.01**	**0.04**	168.97
Temps moyen de résolution (s)	Gecode	0.09	111.10	465.40
	Chuffed	0.12	0.60	**19.10**
	Choco	**0.05**	**0.14**	31.96

TABLE 5.4 – Positionnement de notre approche par rapport à l'état de l'art, avec une heuristique de branchement lexicographique.

Il est mentionné dans [FS14] que l'heuristique VSIDS [MMZ+01] améliore significativement les résultats obtenus par Chuffed. Attestant ainsi qu'heuristique et explications se combinent bien. Par analogie, nous considérons l'heuristique basée sur l'activité des variables (ABS) de [MH12] et l'heuristique choisissant la variable de plus petit domaine [HE79], combinée avec l'exploitation du dernier échec de [LSTV06] (MinDom*), pour notre modèle Choco. Les résultats sont affichés sur la table 5.5. Là encore, les performances sont très similaires. Les trois approches résolvent l'ensemble des $1,500$ instances. Dans les trois cas, le nombre d'échecs et le temps de calcul sont très faibles. On remarque un léger surcoût induit par ABS, mais cette approche reste très compétitive. Dans l'ensemble, avec une bonne heuristique, filtrer plus ou expliquer son filtrage se valent.

	Modèle	Taille d'instance 15	30	60
Nombre d'instances résolues à l'optimum	Chuffed_VSIDS	500	500	500
	Choco_ABS	500	500	500
	Choco_MinDom*	500	500	500
Nombre moyen d'échecs (en milliers)	Chuffed_VSIDS	0.03	0.10	0.10
	Choco_ABS	**0.01**	0.07	**0.05**
	Choco_MinDom*	**0.01**	**0.02**	0.39
Temps moyen de résolution (s)	Chuffed_VSIDS	**0.04**	**0.10**	**0.30**
	Choco_ABS	0.16	0.36	1.58
	Choco_MinDom*	0.06	0.13	0.47

TABLE 5.5 – Positionnement de notre approche par rapport à l'état de l'art, avec une heuristique de branchement générique et adaptative (VSIDS [MMZ+01], ABS [MH12] et MinDom* [HE79, LSTV06]).

5.5 Conclusion du chapitre

Ce chapitre, consacré au filtrage de certaines contraintes de graphes, a mis en évidence deux résultats : Tout d'abord, en terme de modélisation, nous avons montré comment passer d'une contrainte d'arbre à une contrainte de chemin puis d'une contrainte de chemin à une contrainte de circuit. Nous avons ainsi montré comment un algorithme de filtrage, associé à une propriété bien établie sur la contrainte PATH, pouvait être utilisé pour la contrainte CIRCUIT. Cela se distingue de récents algorithmes *ad hoc* pour filtrer la contrainte CIRCUIT. De plus, nous avons proposé un nouvel algorithme pour établir la BC (ou GAC si variables entières) sur ARBO, ANTIARBO et TREE en temps $O(n + m)$ [FL11], au lieu de $O(nm)$. Cet algorithme, basé sur la notion de dominance dans un graphe de flot, est également plus simple dans sa formulation. En pratique, notre évaluation expérimentale confirme que ce nouvel algorithme se traduit par des gains très significatifs.

Il ressort de nos travaux une nette amélioration des performances pour résoudre des problèmes de partitionnement en arborescences et de recherche d'un circuit Hamiltonien en programmation par contraintes. Nous avons multiplié par 10 la taille des instances que la contrainte TREE pouvait traiter dans un temps donné. L'influence des structures de données sur le passage à l'échelle a également été mise en avant, justifiant l'utilisation d'une variable abstraite pour représenter un graphe. Concernant la contrainte CIRCUIT, nous montrons que notre schéma de filtrage permet d'égaler une approche par explications [FS14].

On pourrait étendre cette étude en cherchant de nouvelles propriétés à dériver en algorithmes de filtrage. Notre évaluation expérimentale indiquant que certaines instances à 60 noeuds ne sont pas résolues avec une heuristique lexicographique montre bien que le filtrage a encore une marge de progression. De plus, il serait intéressant de comparer notre modèle, état de l'art en ce qui concerne le filtrage, avec son équivalent expliqué.

Dans le chapitre suivant, nous étudierons la recherche d'un cycle Hamiltonien non-orienté. Nous nous intéresserons particulièrement au cas où chaque arête est associée à un coût, dont la somme doit être minimisée. Nous verrons comment combiner structures de graphe et coûts de manière à disposer d'un filtrage puissant. De plus, nous étudierons différentes stratégies de branchement génériques portant sur une variable de graphe.

6

Résolution du problème du voyageur de commerce en contraintes

Sommaire

6.1 Modélisation **56**
- 6.1.1 Modèle mathématique 56
- 6.1.2 Justification d'une approche non-orientée 56
- 6.1.3 Modèle en contraintes 57
- 6.1.4 Description du propagateur Lagrangien 58

6.2 Etude du branchement **62**
- 6.2.1 Quelques heuristiques génériques sur les graphes 63
- 6.2.2 Adaptation de *Last Conflict* à une variable graphe 64

6.3 Etude expérimentale **66**
- 6.3.1 Impact de l'orientation sur la recherche d'un cycle Hamiltonien . . . 66
- 6.3.2 Evaluation empirique des heuristiques de branchement 69

6.4 Conclusion du chapitre **72**

Etant donné un graphe complet non-orienté $G = (V, E)$ muni d'une fonction de coûts sur les arêtes $c : E \to \mathbb{R}$, le problème du voyageur de commerce (TSP) consiste à trouver un sous-graphe $G' = (V, E')$ de G qui forme un cycle Hamiltonien de coût minimum [ABCC07]. En d'autres termes, il faut trouver un sous-graphe connexe de G, recouvrant tous ses noeuds, tel que chaque noeud a exactement 2 voisins et dont la somme des coûts des arêtes est minimale. Ce problème NP-difficile est probablement l'un des plus étudié en recherche opérationnelle. Il possède de nombreuses applications pratiques, telles que les problèmes de tournées de véhicules, l'optimisation de production industrielle ou encore l'ordonnancement.

Ce chapitre porte sur des questions liées à la modélisation du problème du voyageur de commerce en contraintes. Tout d'abord, nous traiterons de la question du choix d'orientation du modèle de graphe, à savoir : faut-il mieux utiliser un graphe orienté, plus général et capable de prendre en compte des coûts non symétriques, ou au contraire privilégier un graphe non-orienté, plus proche de l'énoncé du problème du voyageur de commerce ? Nous rappellerons ensuite les

algorithmes de filtrage de l'état de l'art en programmation par contraintes [BVHR+12]. Puis, nous étudierons la conception d'une bonne heuristique de recherche pour ce problème, sujet relativement peu approfondi dans la littérature qui s'est davantage focalisée sur des procédures de filtrage. Nous supporterons notre propos par une étude expérimentale montrant un gain significatif sur l'approche de l'état de l'art.

6.1 Modélisation

6.1.1 Modèle mathématique

Le problème du voyageur de commerce se modélise en programmation linéaire en nombre entiers, en associant à chaque arête $(i, j) \in E$ une variable binaire x_{ij} désignant si l'arête appartient ou non à la solution calculée. Le coût de cette arête est noté c_{ij}. On obtient alors le modèle suivant :

$$\begin{array}{llll}
\text{Minimiser} & \sum_{(i,j)\in E} c_{ij}x_{ij} & & (1) \\
\text{Avec} & \sum_{j\in\delta_G(i)} x_{ij} = 2 & \forall i \in V & (2) \\
& \sum_{i,j\in S, i<j} x_{ij} \leq |S| - 1 & \forall S \subset V, |S| > 2 & (3) \\
& x_{ij} \in \{0, 1\} & \forall (i, j) \in E & (4)
\end{array}$$

La fonction objectif (1) est exprimée sous la forme d'un produit scalaire entre un vecteur de réels, c, et un vecteur de variables binaires, x. Ce modèle est également composé de contraintes de degré (2), qui assurent que chaque noeud a bien deux voisins, et de contraintes d'élimination de sous-tours (3), qui éliminent tout éventuel cycle qui ne serait pas Hamiltonien. Il est bon de noter qu'il y a potentiellement une quantité exponentielle de contraintes de sous-tours. Aussi sont-elles ajoutées dynamiquement. Conjointement, les contraintes (2) et (3) assurent implicitement que le graphe est connexe et contient exactement n arêtes. Enfin, les variables du problème sont binaires (4).

Avant d'introduire un modèle en programmation par contraintes pour ce problème, nous devons définir si nous souhaitons ou non opter pour une représentation orientée du problème.

6.1.2 Justification d'une approche non-orientée

Bien que le problème du voyageur de commerce soit symétrique, la plupart des approches par contraintes (e.g., [CL97, FLM02]) en suggèrent une modélisation orientée : à chaque noeud est associé exactement un successeur, de telle sorte que le cycle se transforme en circuit. On utilise alors les contraintes orientées vues au chapitre précédent. Cette modélisation offre l'avantage d'être plus générale : elle permet de considérer directement des coûts non symétriques, ce qui est fréquent dans de nombreuses applications. Cependant, ce gain en généralité a un coût. Il introduit une dimension supplémentaire au problème : le sens de parcours. Ce changement augmente naturellement la combinatoire par rapport à des instances symétriques. Concrètement, chaque cycle de n noeuds engendre 2 circuits et $2^n - 2$ non-circuits. L'augmentation de la combinatoire induite par le modèle orienté est illustrée sur la figure 6.1. Si le modèle orienté à deux fois plus de solutions, l'espace de recherche associé est exponentiellement plus grand. Le problème du voyageur de commerce étant déjà suffisamment difficile en soit, nul besoin de le complexifier davantage.

	non-orienté	orienté
nombre d'arêtes/arcs	$\frac{n(n-1)}{2}$	$n(n-1)$
nombre de cycles/circuits Hamiltoniens	$\frac{(n-1)!}{2}$	$(n-1)!$
densité de cylces/circuits Hamiltoniens	$\frac{(n-1)!}{2*\binom{\frac{n(n-1)}{2}}{n}}$	$\frac{(n-1)!}{\binom{n(n-1)}{n}}$

(a) Evaluation de la combinatoire.

	$n = 10$	$n = 50$	$n = 100$
$\frac{\text{densité de cycles}}{\text{densité de circuits}}$	10^3	10^{15}	10^{30}

(b) Impact chiffré sur la densité de solutions.

FIGURE 6.1 – Impact de l'orientation sur l'énumération des cycles/circuits d'un graphe complet.

Face à ce constat, et dans un souci d'efficacité, nous optons donc pour une modélisation non-orientée du problème du voyageur de commerce.

6.1.3 Modèle en contraintes

Le modèle que nous retenons pour modéliser le problème du voyageur de commerce en contraintes est le suivant :

$$
\begin{array}{rll}
\text{Minimiser} & z & (1) \\
\text{Avec} & \textsc{WeightedCycle}(\mathcal{G}, z, c) & (2) \\
 & \mathcal{G} \in [(V, \emptyset), (V, E)] & (3) \\
 & z \in [-\infty, \texttt{LKH}(G, c)] & (4)
\end{array}
$$

Nous considérons une variable graphe non-orienté $\mathcal{G}$, ayant pour domaine $[\underline{\mathcal{G}}, \overline{\mathcal{G}}]$ avec initialement $\underline{\mathcal{G}} = (V, \emptyset)$ et $\overline{\mathcal{G}} = (V, E)$ (3). Pour rappel, $G = (V, E)$ désigne le graphe d'entrée, supposé complet. Nous introduisons également une variable réelle z, de domaine $[\underline{z}, \overline{z}]$, pour modéliser la fonction objectif (1). Initialement (4), $\underline{z} = -\infty$ et $\overline{z} = \texttt{LKH}(G, c)$, où LKH désigne l'heuristique de [RGJM07] basée sur l'algorithme k-opt, qui résout très efficacement le TSP de manière approchée en $O(n^{2.2})$. Les variables $\mathcal{G}$ et z sont ensuite sujettes à une contrainte de cycle pondéré WEIGHTEDCYCLE($\mathcal{G}$,z,c) [BVHR+12], équivalente d'un point de vue sémantique à la conjonction de CYCLE($\mathcal{G}$) et de $z = \sum_{(i,j)\in\mathcal{G}} c_{ij}$.

Les propagateurs de la contrainte CYCLE sont donnés dans la table 6.1. Chaque propagateur est appliqué incrémentalement en temps constant. On peut ensuite ajouter les propagateurs de la table 6.2 pour filtrer la variable objectif en temps linéaire. Néanmoins, ce filtrage demeure relativement faible pour résoudre à l'optimum le problème du voyageur de commerce. Aussi renforçons-nous le filtrage en conciliant coûts et structure de graphe par l'ajout d'un propagateur

basé sur une relaxation Lagrangienne des contraintes de degré [BVHR+12]. Ce propagateur est décrit en section suivante. Une autre relaxation du problème, relâchant cette fois la connexité de la solution, est la relaxation d'affectation minimum [FLM99, FLM02]. En revanche, comme affirmé dans [BVHR+12], elle est surtout intéressante dans le cas orienté. Pour cette raison, nous ne la considérerons pas ici.

Propriété	Complexité
pas de cycle de taille $< n$	$O(1)$ par ajout d'arête [CL97]
$\forall i \in V, \lvert\delta_{\mathcal{G}}(i)\rvert \leq 2$	$O(1)$ par ajout d'arête
$\forall i \in V, \lvert\delta_{\mathcal{G}}(i)\rvert \geq 2$	$O(1)$ par retrait d'arête[1]

TABLE 6.1 – Filtrage structurel : propagateurs de la contrainte CYCLE($\mathcal{G}$)

Propriété	Astuce	Complexité du filtrage
$z = \sum_{(i,j)\in\mathcal{G}} c_{ij}$		$O(n+m)$
$z = \sum_{(i,j)\in\mathcal{G}} c_{ij}$ $\wedge \lvert\mathcal{E}\rvert = n$	Trier les arêtes par coûts	$O(n+m)$
$z = \sum_{(i,j)\in\mathcal{G}} c_{ij}$ $\bigwedge_{i\in V} \lvert\delta_{\mathcal{G}}(i)\rvert = 2$ (approximation)	Maintenir les deux arêtes les moins chères incidentes à chaque noeud. Faire la somme des coûts et diviser le total par deux.	$O(n+m)$

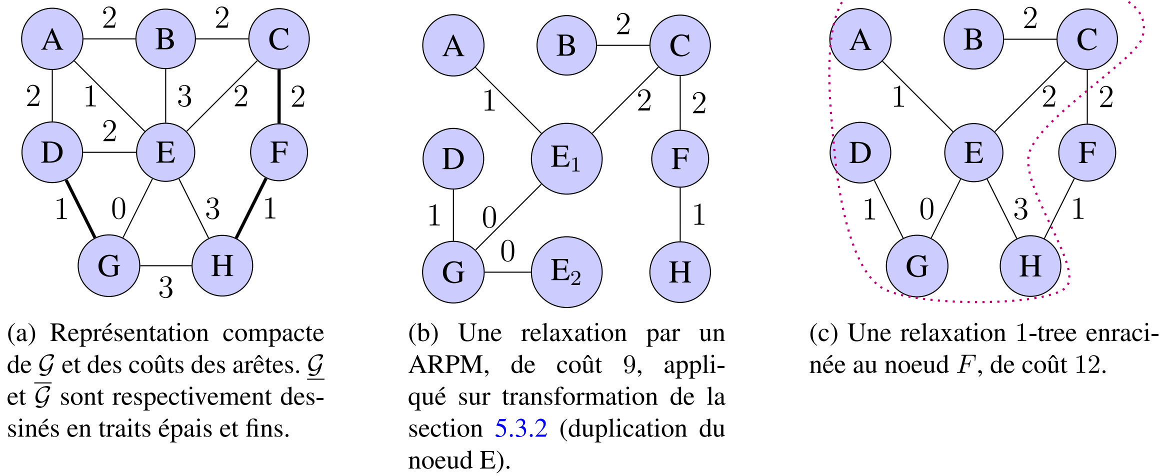
TABLE 6.2 – Filtrage simple basé sur les coûts

6.1.4 Description du propagateur Lagrangien

Relaxation par un 1-tree

Comme vu au chapitre II.5, arbres et cycles sont étroitement liés. On peut donc utiliser la transformation décrite en section 5.3.2, *i.e.,* dupliquer un noeud et calculer un arbre recouvrant de poids minimum (ARPM) sur ce graphe, pour avoir une borne inférieure sur le problème du cycle Hamiltonien de coût minimum. D'ailleurs, Held et Karp ont raffiné cette borne en introduisant la relaxation 1-tree [HK71], qui consiste à isoler le premier sommet $1 \in V$ (on pourrait bien sûr choisir n'importe quel sommet), calculer un ARPM sur $G \setminus \{1\}$ et ajouter ensuite les deux arêtes incidentes au noeud 1 les moins chères. La différence par rapport à la transformation classique décrite en section 5.3.2, est que dans un 1-tree, le noeud mis à l'écart possède exactement deux voisins (distincts) dans la relaxation. Il s'agit donc d'une amélioration assez subtile et dont l'effet se lisse avec la taille du problème. Ces deux approches sont illustrées par la figure 6.2. Dans les deux cas, l'essentiel de la relaxation réside dans la capacité à recouvrir le graphe par un arbre de coût minimum. Nous allons donc rappeler les grandes lignes de cette contrainte.

[1] En pratique, le nombre d'arêtes supprimées étant très élevé, on préférera utiliser une version non-incrémentale propageant un ensemble de retraits d'arêtes en $O(n)$.



(a) Représentation compacte de $\mathcal{G}$ et des coûts des arêtes. $\underline{\mathcal{G}}$ et $\overline{\mathcal{G}}$ sont respectivement dessinés en traits épais et fins.

(b) Une relaxation par un ARPM, de coût 9, appliqué sur transformation de la section 5.3.2 (duplication du noeud E).

(c) Une relaxation 1-tree enracinée au noeud F, de coût 12.

FIGURE 6.2 – Deux manières d'appliquer une relaxation par ARPM.

Filtrage d'une contrainte d'arbre recouvrant pondéré

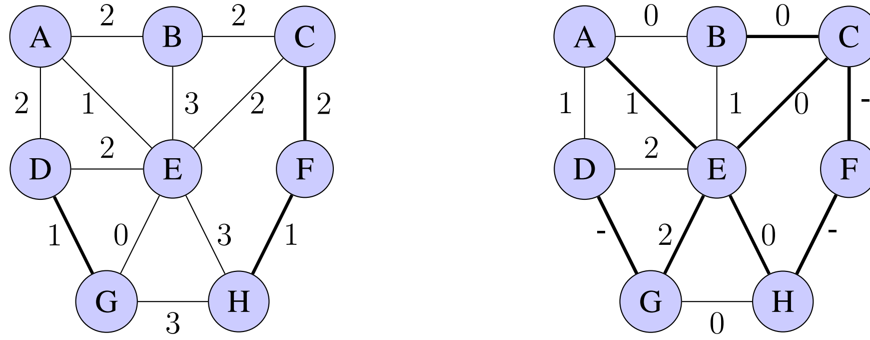
Nous rappelons ici le fonctionnement de la contrainte imposant qu'une variable graphe $\mathcal{G}$ forme un arbre recouvrant de coût inférieur ou égal à z [Rég08]. Appelons $T(\mathcal{G}, c)$ un ARPM dans $\overline{\mathcal{G}}$ contenant les arêtes obligatoires de $\underline{\mathcal{G}}$. Notons $z_{T(\mathcal{G},c)} = \sum_{(i,j)\in T} c_{ij}$ son coût. Puisque l'ARPM est une relaxation, nous pouvons aisément filtrer le domaine de z en appliquant la contrainte :

$$z \geq z_{T(\mathcal{G},c)}$$

Si l'on souhaite maintenant filtrer le domaine de $\mathcal{G}$, il faut connaitre, pour chaque arête dans T, le coût d'un ARPM n'utilisant pas cette arête et, pour chaque arête n'appartenant pas à T, le coût d'un ARPM utilisant forcément cette arête :

$$\begin{aligned} \forall (i,j) \in \overline{\mathcal{G}} \setminus T, \quad & z_{T([\overline{\mathcal{G}}, \underline{\mathcal{G}} \cup \{(i,j)\}],c)} \geq \overline{z} \quad \Rightarrow \quad (i,j) \text{ est impossible} \\ \forall (i,j) \in T, \quad & z_{T([\overline{\mathcal{G}} \setminus \{(i,j)\}, \underline{\mathcal{G}}],c)} \geq \overline{z} \quad \Rightarrow \quad (i,j) \text{ est obligatoire} \end{aligned}$$

Naïvement, il faudrait donc calculer $O(n)$ ARPM. Heureusement, à partir d'un seul ARPM de $\mathcal{G}$, il est possible de calculer en temps constant le coût des autres. Ainsi, étant donné une arête $(i,j) \in \overline{\mathcal{G}}$, $z_{T([\overline{\mathcal{G}}, \underline{\mathcal{G}} \cup \{(i,j)\}],c)} = z_{T(\mathcal{G},c)} + c_{ij} - r^i_{ij}$, où r^i_{ij} désigne le coût de l'arête de $\overline{\mathcal{G}} \setminus \underline{\mathcal{G}}$ la plus chère sur un chemin allant de i à j dans T. De même, étant donné une arête $(i,j) \in T$, $z_{T([\overline{\mathcal{G}} \setminus \{(i,j)\}, \underline{\mathcal{G}}],c)} = z_{T(\mathcal{G},c)} + r^o_{ij} - c_{ij}$, où r^o_{ij} désigne le coût de l'arête de $\overline{\mathcal{G}}$ la moins chère reliant les composantes connexes de i et j dans $T \setminus \{(i,j)\}$. $c_{ij} - r^i_{ij}$ et $r^o_{ij} - c_{ij}$ sont respectivement appelés coût marginal et coût de remplacement de l'arête (i,j). Les coûts marginaux et de remplacement sont aussi appelées coûts réduits. Le calcul de ces coûts est illustré sur la figure 6.3. Par exemple, le coût marginal de l'arête (A,D) est de 1, car (A,D) et l'arête la plus chère liant A à D dans T, à savoir (A,E), valent respectivement 2 et 1. De même, le coût de remplacement de (E,H) vaut 0, car on peut remplacer cette dernière par (G,H), qui a le même coût.



(a) Représentation compacte de $\mathcal{G}$ et des coûts des arêtes. $\underline{\mathcal{G}}$ et $\overline{\mathcal{G}}$ sont respectivement dessinés en traits épais et fins.

(b) Coûts marginaux et de remplacement des arêtes dans une relaxation 1-tree, enracinée en F. Les arêtes dessinées en traits épais forment le support de la relaxation.

FIGURE 6.3 – Illustration du calcul des coûts marginaux et de remplacement.

Dans l'ensemble, la contrainte établit la BC en $O(m\alpha(n,m))$ opérations [RRRVH10]. La fonction inverse d'Ackermann, α, valant 3 pour $n = 10^{80}$ (estimation du nombre d'atomes dans l'univers), on pourrait ne pas la mentionner et annoncer une complexité linéaire. En pratique, derrière cette belle complexité théorique se cachent des constantes, la rendant souvent comparable à un logarithme.

Relaxation Lagrangienne

Vision mathématique

Le modèle mathématique de la relaxation 1-tree est donné ci-dessous. L'unique différence par rapport au modèle mathématique du problème du voyageur de commerce (section 6.1) est que les contraintes de degré associées aux noeuds de $V \setminus \{1\}$ ont été relâchées. Sans ces contraintes de degré, la contrainte $|\mathcal{E}| = n$ n'est plus implicitement établie. Elle est donc ajoutée explicitement au modèle.

$$
\begin{array}{llrr}
\text{Minimiser} & \sum\limits_{(i,j)\in E} c_{ij}x_{ij} & & (1)\\
\text{Avec} & \sum\limits_{j\in\delta_G(1)} x_{1j} = 2 & & (2)\\
 & \sum_{(i,j)\in E} x_{ij} = n & & (3)\\
 & \sum\limits_{i,j\in S, i<j} x_{ij} \leq |S| - 1 & \forall S \subset V, |S| > 2 & (4)\\
 & x_{ij} \in \{0,1\} & \forall (i,j) \in E & (5)
\end{array}
$$

Néanmoins, retirer $n-1$ contraintes est un peu excessif, si bien que la borne qui résulte de cette relaxation risque fort bien de s'avérer trop faible améliorer significativement le filtrage. Aussi, plutôt que de complètement relâcher ces contraintes, Held et Karp [HK71] on proposé de les dualiser, *i.e.,* les inclure dans la fonction objectif de manière à ce que chaque violation soit pénalisée. Chaque contrainte relâchée, portant sur le degré du noeud $i \in V \setminus \{1\}$, est alors associée à un coefficient multiplicateur $\lambda_i \in \mathbb{R}$. Le modèle mathématique du sous-problème Lagrangien est :

$$
\begin{aligned}
\text{Minimiser} \quad & \sum_{(i,j)\in E} c_{ij}x_{ij} + \sum_{i\in V\setminus\{1\}} \lambda_i(2 - \sum_{j\in\delta_G(i)} x_{ij}) && && (1)\\
\text{Avec} \quad & \sum_{j\in\delta_G(1)} x_{1j} = 2 && && (2)\\
& \textstyle\sum_{(i,j)\in E} x_{ij} = n && && (3)\\
& \sum_{i,j\in S, i<j} x_{ij} \leq |S| - 1 && \forall S \subset V, |S| > 2 && (4)\\
& x_{ij} \in \{0,1\} && \forall (i,j) \in E && (5)
\end{aligned}
$$

Quelles que soient les valeurs des λ_i, ce modèle donne une borne inférieure sur le problème du voyageur de commerce. La preuve en est que l'optimum du problème initial est une solution réalisable de même coût dans le sous-problème Lagrangien. En effet, le terme $\sum_{i\in V\setminus\{1\}} \lambda_i(2 - \sum_{j\in\delta_G(i)} x_{ij})$ vaut alors 0 car les contraintes de degré ne sont pas violées. Néanmoins, selon les valeurs des coefficients multiplicateurs, la borne du sous-problème Lagrangien est plus ou moins intéressante. Plus précisément, puisque nous calculons ici une borne inférieure, résoudre cette relaxation Lagrangienne consiste à calculer les valeurs des λ_i qui maximisent la borne donnée par la résolution du sous-problème. Notre borne inférieure sera ainsi au plus près de l'optimum. Pour trouver des multiplicateurs Lagrangiens optimaux, nous mettons en oeuvre l'algorithme d'optimisation de sous-gradient de Held et Karp [HK71]. Cette méthode itérative part d'un vecteur nul, pour les multiplicateurs, qu'elle modifie en fonction des degrés de violation des contraintes. Il a été prouvé que la valeur de cette relaxation était égale à la valeur de la relaxation linéaire du problème [KV12]. L'intérêt de cette relaxation est double : d'une part, elle ne nécessite pas de solveur de programmation linéaire, d'autre part, elle converge bien plus vite vers une bonne borne. En revanche, la terminaison de la convergence est très lente, si bien que l'on interrompt souvent ce processus de manière heuristique.

Vision pragmatique

D'un point de vue plus pragmatique, la relaxation Lagrangienne des contraintes de degré n'est autre qu'une succession de relaxations 1-tree entre lesquelles la fonction de coût est à chaque fois modifiée (par l'algorithme de sous-gradient). On notera que, puisque la fonction de coût varie entre chaque appel, on ne peut pas utiliser d'algorithme incrémental pour calculer le 1-tree [Rég08].

Paramétrage du propagateur

Il est crucial de garder à l'esprit qu'intégrer la relaxation Lagrangienne des contraintes de degré dans un propagateur, signifie que la méthode sera appelée à chaque noeud de l'arbre de recherche. Ainsi, il est important de limiter son temps d'exécution et de l'interrompre dès lors qu'elle ne produit plus d'amélioration significative sur l'estimation inférieure à l'optimum. La propagation des autres contraintes et le branchement permettra de mieux avancer dans la résolution.

De plus, il est inutile de repartir du vecteur nul à chaque nouvelle propagation. Au contraire, il vaut mieux repartir du vecteur précédent, supposé de bonne qualité. On pourra alors se permettre de diminuer encore le nombre d'itérations de la méthode. En ce sens, le propagateur dispose d'une forme d'incrémentalité. Cela permet de réduire drastiquement le temps passé dans la contrainte.

La politique idéale de rétro-propagation (filtrage sur $\mathcal{G}$) reste une question ouverte. En effet, chaque itération de la relaxation lagrangienne peut donner lieu à une rétro-propagation et, plus $\overline{\mathcal{G}}$ est éparse et $\underline{\mathcal{G}}$ est dense, plus la résolution de chaque sous-problème est rapide. De plus, le

filtrage à une itération k est incomparable au filtrage à l'itération $k+1$ [Sel04]. Filtrer durant la convergence peut donc conduire à plus de filtrage sur $\mathcal{G}$ que filtrer à la fin de la convergence (là où la borne inférieure est pourtant optimale). On aurait donc envie de filtrer $\mathcal{G}$ à chaque itération de la relaxation Lagrangienne. Cependant, filtrer durant la convergence perturbe l'algorithme de sous-gradient, retirant les garanties de convergence [Sel04]. De plus, la procédure de filtrage rajoute un surcoût à chaque itération. Aussi, nous préconisons de n'appliquer la rétro-propagation que par intermittence (e.g., toutes les 30 itérations), de manière à trouver un bon compromis sur la fréquence du filtrage. D'ailleurs, le filtrage basé sur un 1-tree proposé dans [RRRVH10] repose sur l'algorithme de Kruskal et nécessite un tri des arêtes. En pratique, si l'on ne souhaite pas filtrer $\mathcal{G}$ à une itération donnée, un simple appel à l'algorithme de Prim est plus rapide. On peut donc changer dynamiquement la méthode de résolution du sous-problème, selon que l'on souhaite ou non filtrer $\mathcal{G}$. On améliore ainsi les performance de l'approche.

Enfin, dans le cas particulier où la recherche de solution est soumise à une méthode de redémarrage (*restart*), il est pertinent de mémoriser la meilleure paire borne et coût réduit de chaque arête, calculés aux noeuds racines de l'arbre de recherche. Ainsi, lorsqu'un redémarrage est déclenché, les relaxations des noeuds racines précédents offrent une condition de filtrage immédiate, avant même qu'une nouvelle relaxation ait été calculée. Nous parlons de noeuds racines au pluriel car nous considérons que chaque redémarrage donne lieu à un nouveau noeud racine. De manière analogue, lors d'une optimisation par le bas, on peut réutiliser les paires borne et coûts réduits du noeud racine après chaque branchement sur la variable objectif.

Perspectives d'adaptation de techniques d'exploration à l'optimisation par sous-gradient

Puisque le filtrage dépend du processus de convergence, alors lancer en parallèle plusieurs relaxations Lagrangiennes, avec différents vecteurs initiaux et/ou différents algorithmes de sous-gradient peut, en théorie, permettre d'aboutir à davantage de filtrage. Typiquement, certains algorithmes de sous-gradient qui ne convergent pas sont parfois plus efficaces en pratique [Cam14]. On aurait donc envie d'avoir un portfolio d'algorithmes de sous-gradient. Le temps passé dans une relaxation Lagrangienne étant, en général, largement supérieur au temps passé à recopier les domaines des variables, on peut paralléliser ce processus.

De même, à l'instar du principe de redémarrage utilisé en recherche arborescente (*random-restart*), on peut dynamiser une relaxation Lagrangienne qui aurait du mal à améliorer sa borne, en modifiant arbitrairement ses multiplicateurs et en relançant une convergence. Cette technique permet d'espérer filtrer davantage, mais induit en revanche un surcoût algorithmique certain.

Ce type de raisonnements ne s'applique pas uniquement aux contraintes embarquant une relaxation Lagrangienne. Il peut être étendu aux contraintes basées sur un diagramme de décision multi-valué [BCS+14]. En effet, le filtrage de telles contraintes dépend en partie de l'ordre dans lequel les variables ont été choisies. Des choix différents amèneraient à des diagrammes différents et donc, potentiellement, à plus de filtrage. Ces méthodes étant algorithmiquement coûteuses, il ferait également sens de les paralléliser.

6.2 Etude du branchement

Pour améliorer un modèle, il est coutume de dire que travailler sur les structures de données, l'heuristique de recherche et le filtrage permet d'espérer accélérer la résolution par des facteurs respectifs de 10, 10^3 et 10^6, avec une chance de succès de respectivement 95%, 1% et 0.001% [Rég13]. Dans notre cas, l'utilisation d'une variable graphe assure le premier facteur 10. De plus, la procédure de filtrage basée sur une relaxation Lagrangienne de [BVHR+12] a déjà apporté un facteur 10^6. Obtenir un facteur 10^6 supplémentaire semble très ambitieux. Aussi

allons-nous nous consacrer au branchement, qui n'a pas suscité beaucoup d'attention dans l'état de l'art, pour espérer obtenir un facteur 10^3 sur le temps de calcul. Cette section décrit l'application de diverses heuristiques de branchement conjointement à l'exploitation du dernier conflit introduite par Lecoutre *et al.* [LSTV06, LSTV09] et adaptée au cas des variables graphes.

6.2.1 Quelques heuristiques génériques sur les graphes

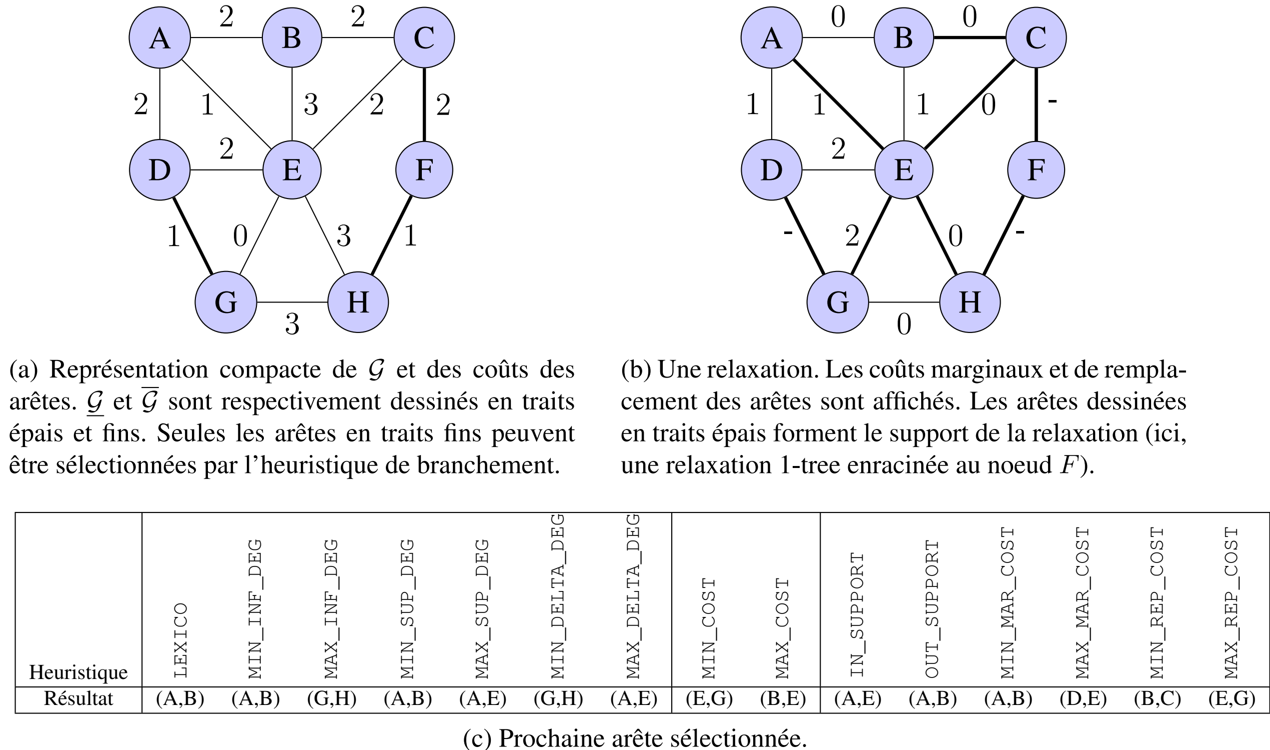
Nous présentons ici un ensemble varié d'heuristiques permettant de brancher sur une variable graphe. Plus précisément, nous nous intéressons à des heuristiques de choix d'arêtes non déjà fixées, *i.e.,* dans $\overline{\mathcal{E}} \setminus \underline{\mathcal{E}}$. Afin de gagner en généralité, nous prenons volontairement des heuristiques relativement simples, chacune définie par un seul critère. Les éventuelles égalités sont tranchées par ordre lexicographique.

- Heuristiques génériques basées uniquement sur la variable graphe concernée :
  - `LEXICO` : sélectionne la première arête de $\overline{\mathcal{E}} \setminus \underline{\mathcal{E}}$.
  - `MIN_INF_DEG` (resp. `MAX_INF_DEG`) : sélectionne la première arête pour laquelle la somme des degrés de ses extrémités dans $\underline{\mathcal{G}}$ est minimale (resp. maximale).
  - `MIN_SUP_DEG` (resp. `MAX_SUP_DEG`) : sélectionne la première arête pour laquelle la somme des degrés de ses extrémités dans $\overline{\mathcal{G}}$ est minimale (resp. maximale).
  - `MIN_DELTA_DEG` (resp. `MAX_DELTA_DEG`) : sélectionne la première arête pour laquelle la somme des degrés de ses extrémités dans $\overline{\mathcal{G}}$ moins la somme des degrés de ses extrémités dans $\underline{\mathcal{G}}$ est minimale (resp. maximale).
- Heuristiques basées sur une fonction de coût :
  - `MIN_COST` (resp. `MAX_COST`) : sélectionne la première arête de coût minimal (resp. maximal).
- Heuristiques basées sur une relaxation :
  - `IN_SUPPORT` (resp. `OUT_SUPPORT`) : sélectionne la première arête inclue dans le (resp. exclue du) support de la relaxation.
  - `MIN_MAR_COST` (resp. `MAX_MAR_COST`) : sélectionne la première arête dont le coût marginal est minimal (resp. maximal), *i.e.,* l'arête exclue du support de la relaxation dont l'ajout à $\underline{\mathcal{E}}$ engendre la plus petite (resp. grande) augmentation de la valeur de la relaxation.
  - `MIN_REP_COST` (resp. `MAX_REP_COST`) : sélectionne la première arête dont le coût de remplacement est minimal (resp. maximal), *i.e.,* l'arête du support de la relaxation dont le retrait de $\overline{\mathcal{E}}$ engendre la plus petite (resp. grande) augmentation de la valeur de la relaxation.

Ces heuristiques sont illustrées sur la table de la figure 6.4c. Etant donné le domaine de $\mathcal{G}$ décrit sur la figure 6.4a et la relaxation de la figure 6.4b, la table 6.4c indique l'arête qui serait sélectionnée par chaque heuristique pour calculer la prochaine décision de branchement.

Il est bon de mentionner que l'heuristique employée dans l'approche par contraintes de l'état de l'art [BVHR+12] est `MAX_REP_COST`. Cependant, ils utilisent un ordre de branchement inverse (ils commencent par retirer l'arête sélectionnée de $\overline{\mathcal{E}}$ puis, en cas d'échec, backtrackent pour l'ajouter à $\underline{\mathcal{E}}$). Nous avons trouvé que l'ordre de branchement n'avait pas un impact très fort sur les performances, c'est pourquoi nous considérons dans notre étude le cas où l'arête

est d'abord forcée puis, éventuellement, filtrée. Cet ordre est plus naturel, par analogie aux variables entières pour lesquelles les décisions sont généralement des affectations, et non des retraits de valeurs.



(a) Représentation compacte de $\mathcal{G}$ et des coûts des arêtes. $\underline{\mathcal{G}}$ et $\overline{\mathcal{G}}$ sont respectivement dessinés en traits épais et fins. Seules les arêtes en traits fins peuvent être sélectionnées par l'heuristique de branchement.

(b) Une relaxation. Les coûts marginaux et de remplacement des arêtes sont affichés. Les arêtes dessinées en traits épais forment le support de la relaxation (ici, une relaxation 1-tree enracinée au noeud F).

Heuristique	LEXICO	MIN_INF_DEG	MAX_INF_DEG	MIN_SUP_DEG	MAX_SUP_DEG	MIN_DELTA_DEG	MAX_DELTA_DEG	MIN_COST	MAX_COST	IN_SUPPORT	OUT_SUPPORT	MIN_MAR_COST	MAX_MAR_COST	MIN_REP_COST	MAX_REP_COST
Résultat	(A,B)	(A,B)	(G,H)	(A,B)	(A,E)	(G,H)	(A,E)	(E,G)	(B,E)	(A,E)	(A,B)	(A,B)	(D,E)	(B,C)	(E,G)

(c) Prochaine arête sélectionnée.

FIGURE 6.4 – Illustration d'une variable graphe $\mathcal{G}$ et d'une relaxation du TSP.

6.2.2 Adaptation de *Last Conflict* à une variable graphe

Last Conflict [LSTV06] est une heuristique générique composite, qui vient se greffer sur une heuristique principale pour modifier son comportement après un échec, d'où le terme de patron de branchement. Pour la résumer, cette règle impose que si une décision portant sur la variable x conduit à un échec, alors il faut continuer à brancher sur x. Ceci, dans le but d'échouer à nouveau et ainsi sortir d'un espace de recherche présumé non réalisable. Pour cette raison, on peut voir ce choix heuristique comme une forme très naïve d'apprentissage, n'engendrant aucun surcoût algorithmique. Cette heuristique suit le principe *Fail First* [HE79, SG98, BPW04, BPW05], qui recommande de prendre des décisions de branchement qui sont susceptibles de déclencher des échecs, dans le but de sortir le plus rapidement possible des régions non réalisables

Revenons à notre étude du problème du voyageur de commerce en contraintes. Il existe en recherche opérationnelle de très bonnes méta-heuristiques permettant de trouver d'excellentes solutions à ce problème [Hel00, RGJM07]. Si nous en utilisons une en pré-calcul pour initialiser la borne supérieure de la variable objectif (à une valeur proche de l'optimum), on peut vraisemblablement supposer qu'une grande partie de la recherche se fera dans des régions non réalisables de l'espace de recherche. Typiquement, une fois une solution optimale trouvée, prouver son optimalité nécessite d'explorer une région non-réalisable et prouver que cet espace ne contient pas de solution. On aurait donc *a priori* intérêt à utiliser la méthode *Last Conflict*. Cependant, on ne peut pas l'utiliser telle quelle, puisque nous ne branchons de toute manière que sur une seule variable, à savoir $\mathcal{G}$. Nous allons donc proposer une interprétation de *Last Conflict* pour les variables graphes. Concrètement, plutôt que de mémoriser la variable, nous proposons

de mémoriser un de ses noeuds impliqués dans la dernière décision de branchement. Cette règle est illustrée sur la figure 6.5 : lorsqu'une décision portant sur une arête $(i, j) \in \overline{\mathcal{E}}$ est appliquée, une de ses deux extrémités, par exemple i, est mémorisée. Si la propagation de cette décision engendre un échec, alors notre règle stipule que la prochaine décision doit porter sur une arête incidente à ce noeud i. Ceci ne s'applique évidemment que lorsque l'ensemble des arêtes de $\overline{\mathcal{E}} \setminus \underline{\mathcal{E}}$ qui sont incidentes à i est non nul. En d'autres termes, lorsqu'une décision engendre un échec, on continue à brancher sur la même région du graphe, dans le but de découvrir un nouvel échec le plus tôt possible.

Nous étudions trois manières assez naturelles de sélectionner le noeud à réutiliser après qu'une décision portant sur l'arête $(u, v) \in E$ ait conduit à un échec :

- `LC_FIRST` : on ne mémorise que le premier noeud, à savoir u.
- `LC_RANDOM` : on mémorise aléatoirement une des extrémités.
- `LC_BEST` : on mémorise les deux noeuds et on sélectionne celui qui maximise le critère de l'heuristique de branchement principale. Par exemple, si `MIN_COST` est utilisée, alors nous allons réutiliser le noeud ayant l'arête non fixée incidente la moins chère.

Une manière simple d'implémenter `LC_FIRST` est proposée par l'algorithme 1. Lorsqu'un échec est levé, cette règle modifie l'heuristique principale en réduisant l'ensemble des arêtes candidates à être sélectionnées dans la prochaine décision à l'ensemble des arêtes non fixées et incidentes au noeud sélectionné (ligne 7).

Algorithm 1 Implémentation de `LC_FIRST` comme une heuristique de graphe composite

```
global int fails_stamp
global GraphHeuristic h
global Vertex last_vertex

 1: function NEXTEDGE(GraphVar g)
 2:     Edge next_edge
 3:     if (fails_stamp = nbFails() ∨ (|g.getNeighbors(last_vertex)| =
        |ḡ.getNeighbors(last_vertex)|)) then
 4:         next_edge ← h.NEXTEDGE(g)
 5:         last_vertex ← next_edge.getFirstVertex()
 6:     else // Ajuste l'heuristique h en imposant de brancher sur une arête incidente au noeud
        last_vertex
 7:         next_edge ← h.NEXTEDGEINCIDENTTO(last_vertex, g)
 8:     end if
 9:     fails_stamp ← nbFails()
10:     return next_edge
11: end function
12:
13: function NEXTEDGEINCIDENTTO(Vertex i, GraphVar g)
14:     return h.NEXTEDGEINCIDENTTO(i,g)
15: end function
```

Une illustration de `LC_FIRST` appliqué à `MAX_COST` est donnée sur la figure 6.5. `MAX_COST` sélectionne séquenciellement les arêtes (B, E), (G, H), et (A, B). La propagation de contraintes lève un échec, donc le solveur backtrack et supprime l'arête (A, B). Cela mène à un nouvel

échec, tout comme le retrait de l'arête (G, H), donc le solveur continue de backtracker et déduit que l'arête (B, E) doit être retirée. La propagation de ce retrait n'engendre pas d'échec, donc une nouvelle décision doit alors être calculée pour poursuivre la résolution. `LC_FIRST` modifie alors `MAX_COST` de telle sorte qu'elle sélectionne une arête incidente au noeud A, car A figurait dans la dernière décision calculée.

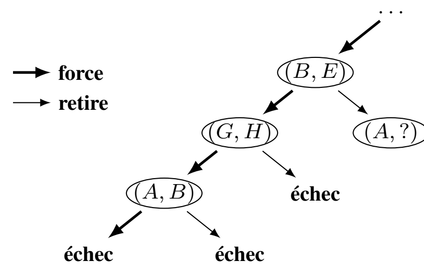


FIGURE 6.5 – Illustration de l'exploration d'un arbre de recherche en utilisant `LC_FIRST` conjointement à `MAX_COST` sur l'exemple de la figure 6.4. Après un échec, la prochaine décision doit inclure une arête incidente au premier noeud de la dernière décision calculée (ici le noeud A).

6.3 Etude expérimentale

Dans cette section, nous cherchons à évaluer empiriquement l'impact de différents aspects importants liés à la modélisation d'un problème. Tout d'abord, nous évaluons l'impact d'utiliser un modèle orienté pour résoudre un problème non-orienté. Afin de simplifier cette étude, et pour traiter des instances de grandes tailles, nous étudions le problème du cycle Hamiltonien (un TSP sans coûts). Ensuite, nous évaluons les différentes heuristiques vues en section 6.2 sur la résolution du TSP. Nous comparons alors la meilleure configuration trouvée avec l'état de l'art [BVHR+12], dans le but de mesurer le gain obtenu en travaillant uniquement l'heuristique de branchement.

Notre approche a été implémentée avec le solveur de contraintes `Choco-3`. Les tests ont été exécutés sur un `Mac Pro` pourvu de six coeurs `Intel Xeon` à 2.93 Ghz, sous `MacOS 10.6.8` et `Java 1.7`. Chaque exécution disposait d'un seul coeur.

6.3.1 Impact de l'orientation sur la recherche d'un cycle Hamiltonien

Dans cette section, nous comparons les capacités respectives des modèles orientés et non-orientés à résoudre un problème de recherche d'un cycle Hamiltonien dans un graphe (table 6.3). Pour le modèle non-orienté (UNDIR), nous retenons le filtrage structurel de la section 6.1. Puisque l'orientation augmente la complexité du problème, nous considérons deux modèles orientés : le premier (DIR_1) est exactement analogue au modèle non-orienté ; le deuxième (DIR_2) est renforcé par une contrainte ALLDIFFERENT assurant la GAC. Nous considérons les instances HCP de Erbacci de la TSPLIB. Ces instances impliquent des graphes ayant de $1,000$ à $5,000$ noeuds, et environ deux fois plus d'arêtes qu'il y a de noeuds. Ce sont donc des graphes très éparses, ce qui rend le problème d'autant plus difficile. Comme heuristique de recherche, nous sélectionnons arbitrairement un des noeuds ayant le plus petit ensemble de successeurs (ou voisins) et forçons arbitrairement un de ses arcs (ou arêtes) incidents. Nous considérons deux modes de résolution : le premier, classique, sans redémarrage de la recherche ; le deuxième relance la recherche tous les 100 échecs. Le processus de résolution est limité à 10 secondes.

Commençons par analyser les résultats obtenus sans redémarrage (table 6.3a). L'approche non-orientée est clairement plus efficace que les deux modèles orientés. Plus précisément, concernant le modèle orienté, la contrainte ALLDIFFERENT apporte un gain significatif sur le nombre de noeuds explorés, sans que cela ne soit rentable en terme de temps. On observe également que la capacité à résoudre une instance est assez binaire : soit l'instance est résolue très rapidement, soit elle n'est pas résolue dans le temps accordé. Dans ce type de situation, cela peut être du à un mauvais choix de branchement assez tôt dans la recherche, et il est généralement conseillé de redémarrer cette dernière pour sortir d'une mauvaise branche.

Nous étudions alors le cas où la recherche est relancée tous les 100 échecs (table 6.3b). La tendance est encore plus marquée : alors que le modèle non-orienté bénéficie de cet ajout (il résout toutes les instances en moins de trois secondes), le modèle orienté est gravement détérioré. Seule la première instance a pu être résolue dans le temps imparti. Pour conclure, non seulement une modélisation non-orientés offre de meilleurs résultats, mais elle permet également d'être plus robuste à des modifications de l'exploration de l'arbre de recherche.

	DIR_1		DIR_2		UNDIR	
instance	|échecs|	temps (s)	|échecs|	temps (s)	|échecs|	temps (s)
alb1000.hcp	4377	0.6	**182**	0.3	392	**0.1**
alb2000.hcp	1653	0.2	1109	1.6	**38**	**0.0**
alb3000a.hcp	-	-	-	-	**872**	**0.2**
alb3000b.hcp	38052	4.6	-	-	**4165**	**0.4**
alb3000c.hcp	-	-	-	-	-	-
alb3000d.hcp	15399	1.4	943	2.9	**7**	**0.1**
alb3000e.hcp	9319	1.1	438	2.2	**19**	**0.1**
alb4000.hcp	-	-	-	-	**18197**	**0.2**
alb5000.hcp	-	-	-	-	-	-

(a) Exploration sans redémarrage.

	DIR_1		DIR_2		UNDIR	
instance	|échecs|	temps (s)	|échecs|	temps (s)	|échecs|	temps (s)
alb1000.hcp	10154	0.9	1638	2.3	**618**	**0.4**
alb2000.hcp	-	-	-	-	**38**	**0.0**
alb3000a.hcp	-	-	-	-	**1300**	**0.6**
alb3000b.hcp	-	-	-	-	**1803**	**0.8**
alb3000c.hcp	-	-	-	-	**2903**	**1.3**
alb3000d.hcp	-	-	-	-	**7**	**0.1**
alb3000e.hcp	-	-	-	-	**19**	**0.1**
alb4000.hcp	-	-	-	-	**607**	**0.5**
alb5000.hcp	-	-	-	-	**1901**	**2.2**

(b) Exploration avec redémarrage tous les 100 échecs.

TABLE 6.3 – Recherche d'un cycle Hamiltonien dans un graphe éparse de grande taille.

Pour aller encore plus loin en ce qui concerne le passage à l'échelle, nous considérons maintenant le problème du Cavalier d'Euler. Considérons un damier carré de longueur l. Le problème du cavalier d'Euler consiste à parcourir exactement une fois chacune des cases du damier en se déplaçant selon la règle du cavalier aux échecs, *i.e.,* un pas droit et un pas en diagonale. Une illustration de ce problème est donnée sur la figure 6.6. On numérote arbitrairement chacune des cases de 1 à l^2. Considérons le graphe non-orienté $G = (V, E)$ où chaque noeud de $i \in V$ est associé à la case numéro i et où il existe une arête $(i, j) \in E$ si et seulement si la règle de

déplacement du cavalier autorise le déplacement entre la case i et la case j. Alors, le problème du cavalier d'Euler se ramène au problème de recherche d'un cycle Hamiltonien dans G. Nous étudions alors la capacité de différents modèles à résoudre ce problème.

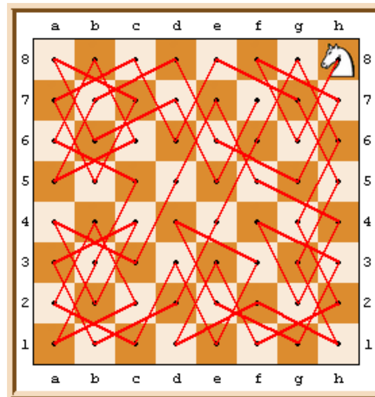


FIGURE 6.6 – Illustration[2] du problème du cavalier d'Euler sur un damier classique de longueur 8. La dernière arête fermant le cycle n'est pas représentée.

Nous considérons notre modèle basé sur une variable de graphe non-orienté et le filtrage structurel de la section 6.1. Nous considérons également une autre version de ce modèle mais encodé avec des variables booléennes, une variable étant associée à chaque arête pour indiquer son appartenance à la solution. Ces deux premiers modèles sont équipés des mêmes algorithmes de filtrage. La seule différence réside dans les améliorations d'implémentation apportées par les variables graphes (e.g., un accès direct aux voisins des noeuds, une plus faible consommation mémoire). Nous considérons aussi un troisième modèle basé sur des variables entières et la contrainte CIRCUIT. Ce modèle utilise des algorithmes de filtrage plus lourds (ALLDIFFERENT, ARBO *etc.*) pour compenser l'ajout des symétries qu'il engendre. Puisqu'il est basé sur des variables entières, il s'agit du modèle le plus utilisé en programmation par contraintes.

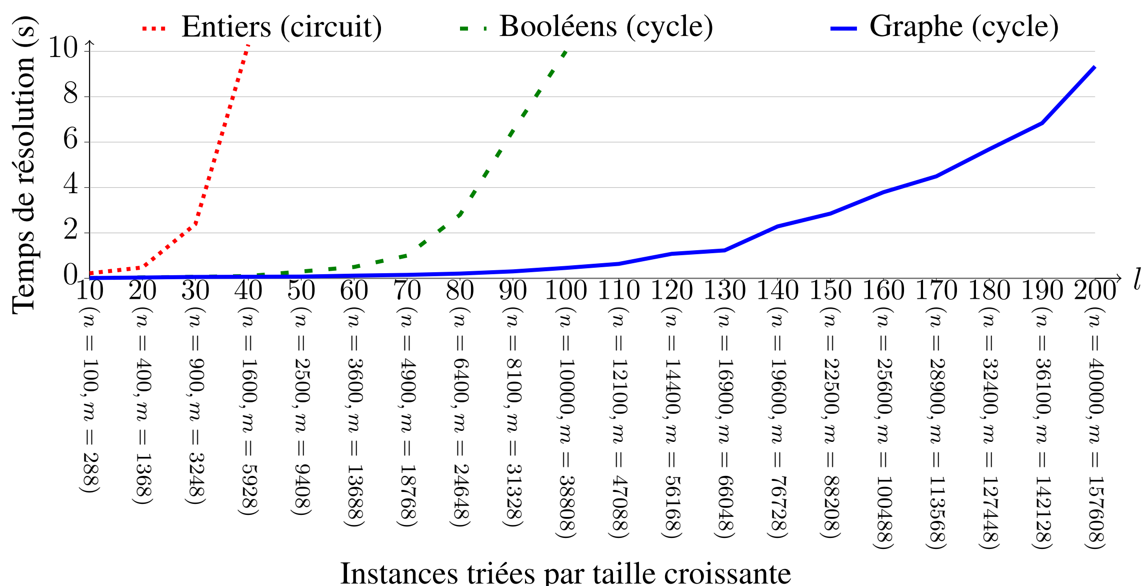


FIGURE 6.7 – Résolution du problème du cavalier d'Euler avec trois modèles différents. l, n et m indiquent respectivement la largeur de l'échiquier, le nombre de noeuds et le nombre d'arêtes de l'instance traitée.

[2] Source de l'image : http ://commons.wikimedia.org/wiki/File :Marche_du_cavalier_selon_al-Adli_ar-Rumi.png ?uselang=fr.

Nous évaluons ces trois modèles sur le problème du cavalier d'Euler sur des instances de taille croissante et avec une limite de temps de résolution de dix secondes. La figure 6.7 représente les résultats obtenus. Tout d'abord, le modèle utilisant des variables entières est assez décevant, puisqu'il échoue dès 1600 noeuds ($l = 40$). Ceci est dû au coût algorithmiques des propagateurs utilisés. Nous précisons qu'en appliquant un filtrage moins fort (mais plus rapide), les résultats sont pires. Le modèle booléen se débrouille bien mieux, bloquant à partir de 10000 noeuds ($l = 100$). Avec un variable de graphe, la limite est repoussée jusqu'à 40000 noeuds ($l = 200$). Dans l'ensemble, cette expérience met en relief deux résultats importants :

- Il vaut mieux avoir le bon modèle (non-orienté) et des algorithmes de filtrage légers qu'un mauvais modèle muni d'algorithmes beaucoup plus sophistiqués (modèle *successeurs*).
- Non seulement une variable de graphe facilite l'étude théorique et l'implémentation d'un algorithme de filtrage, mais elle en améliore également la capacité de passage à l'échelle.

6.3.2 Evaluation empirique des heuristiques de branchement

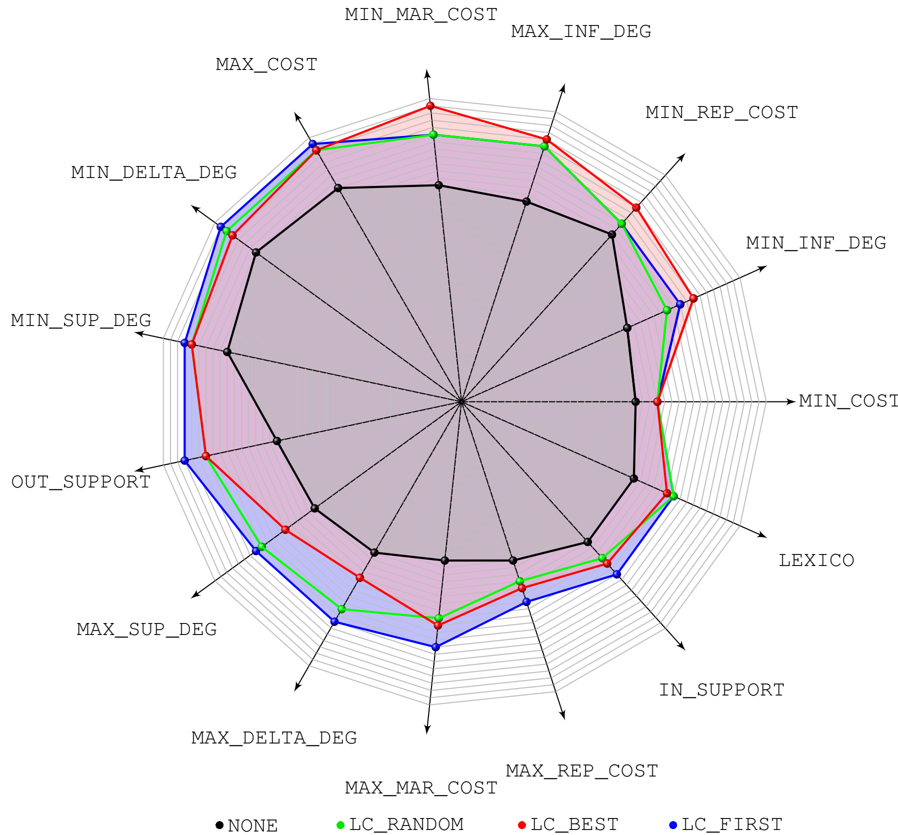
Nous considérons ici les instances de la librairie TSPLIB, qui est le benchmark de référence en recherche opérationnelle. Cette librairie est une compilation d'instances venant de différents contributeurs. Nous considérons celles qui ont jusqu'à 300 noeuds, ce qui semble être la nouvelle limite d'une approche par contraintes sur ce benchmark hautement combinatoire. Nous concentrons notre étude sur la capacité à trouver une solution optimale et à prouver son optimalité. Aussi, nous utiliserons l'heuristique de [Hel00] en pré-calcul pour obtenir une borne supérieure initiale. Sur les instances étudiées, cette borne est optimale.

Notre évaluation empirique est divisée en deux parties. Dans un premier temps, nous comparons les heuristiques introduites en section 6.2 conjointement aux différentes politiques de *Last Conflict* que nous avons proposés. Nous considérons également le cas où aucune forme de *Last Conflict* n'est utilisé. Ceci permet de mettre en relief les bonnes et les mauvaises directions lors de la conception d'une heuristique de branchement pour ce type de problèmes. Dans un deuxième temps, nous comparons les performances de l'approche état de l'art avec celles de notre approche, en retenant la meilleure heuristique. Cela permet de mesurer le gain que l'on peut espérer en ne travaillant que sur l'heuristique de branchement.

Analyse empirique

Nous comparons maintenant les performances des diverses configurations pour l'heuristique de branchement qui ont été introduites précédemment. Nous considérons ici les 42 instances de la TSPLIB les plus simples et une limite de temps d'une minute. La figure 6.8 restitue les résultats obtenus pour chaque heuristique et chaque politique de *Last Conflict*. La ligne `NONE` correspond au cas où *Last Conflict* n'est pas utilisé du tout. Nous indiquons le nombre d'instances résolues à l'optimum dans le temps imparti. Les meilleurs résultats sont obtenus avec les heuristiques `MAX_COST` et `MIN_DELTA_DEG`. A l'inverse, `MIN_COST` donne de mauvais résultats. En d'autres termes, pour trouver (et en faire la preuve) une solution de coût minimal, il vaut mieux chercher à utiliser les arêtes les plus chères : cela confirme bien l'intuition du principe *Fail-First* sur lequel nous nous sommes basés. Ce résultat intéressant montre au passage que ces heuristiques très simples offrent de meilleurs résultats que celles, plus complexes, qui sont basées sur une relaxation du problème. Etonnamment, `MAX_REP_COST`, qui est employé dans l'approche de l'état de l'art, ne donne pas de très bons résultats. Cela confirme qu'il y avait un vide à combler dans les travaux portant sur la résolution de ce problème en programmation par contraintes. De plus, quelqu'en soit la politique, *Last Conflict* améliore chaque heuristique qui a été étudiée.

Cela confirme sont intérêt pratique dans le cas général. Le mode le plus performant est aussi le plus simple, LC_FIRST, qui permet en moyenne de résoudre 27% d'instances en plus dans le temps alloué. Dans l'ensemble, la combinaison de LC_FIRST avec MIN_DELTA_DEG ou MAX_COST permet de résoudre 41 des 42 instances considérées en moins d'une minute chacune. En conclusion, on peut affirmer que l'adaptation à une variable graphe de *Last Conflict* mérite d'être considérée car elle améliore significativement chaque heuristique de notre modèle.



(a) Représentation graphique du nombre d'instances résolues à l'optimum en moins d'une minute.

	LEXICO	MIN_INF_DEG	MAX_INF_DEG	MIN_SUP_DEG	MAX_SUP_DEG	**MIN_DELTA_DEG**	MAX_DELTA_DEG	MIN_COST	**MAX_COST**	IN_SUPPORT	OUT_SUPPORT	MIN_MAR_COST	MAX_MAR_COST	MIN_REP_COST	MAX_REP_COST	Minimum	Average	Maximum
NONE	26	33	25	25	29	35	24	24	34	26	26	30	22	31	23	22	27.5	35
LC_FIRST	32	39	35	33	37	41	35	27	41	32	39	37	34	33	29	**27**	**34.9**	**41**
LC_BEST	31	38	30	35	38	39	28	27	40	30	36	41	31	36	27	**27**	33.8	**41**
LC_RANDOM	32	38	34	31	37	40	33	27	40	29	36	37	30	33	26	26	33.5	40
Minimum	26	33	25	25	29	**35**	24	24	34	26	26	30	22	31	23			
Average	30.3	37.0	31.0	31.0	35.3	**38.8**	30.0	26.3	**38.8**	29.3	34.3	36.3	29.3	33.3	26.3			
Maximum	32	39	35	35	38	**41**	35	27	**41**	32	39	**41**	34	36	29			

(b) Représentation sous forme de table du nombre d'instances résolues à l'optimum en moins d'une minute. Les minima, moyennes et maxima sont également affichés pour une lecture globale des résultats.

FIGURE 6.8 – Evaluation quantitative des différentes configurations de branchement sur les 42 instances les plus simples de la TSPLIB, avec une limite de temps d'une minute.

Comparaison à l'état de l'art

Nous comparons maintenant notre modèle avec celui de l'état de l'art, proposé par Benchimol et d'autres [BVHR$^+$12]. Nous le notons `SOTA`. Il s'agit d'un solveur dédié implémenté en `C++` et compilé avec `gcc` en mode `-O3`. Pour cette étude, nous retenons l'heuristique `MAX_COST` avec `LC_FIRST`, une des meilleures configurations trouvées.

Afin de bien mesurer l'impact de l'heuristique sur les performances, nous employons maintenant une limite de temps de $30,000$ secondes. Nous considérons 34 instances ayant entre 100 et 300 noeuds, ce qui correspond à des instances très difficiles, mais dont la résolution reste envisageable. La figure 6.10 donne les résultats de notre approche et de `SOTA`. L'axe horizontal représente les instances. Le premier axe vertical montre le facteur d'accélération réalisé par rapport à `SOTA` (courbe noire), avec une échelle logarithmique. Le second axe vertical affiche le temps de résolution de `CHOCO` (gris foncé) et de `SOTA` (gris clair), en secondes. Les temps précis de résolution sont également donnés sur une table. Nous affichons également, les résultats obtenus avec `MIN_DELTA_DEG` au lieu de `MAX_COST`. Les résultats sont assez similaires.

La figure 6.10 montre clairement que notre schéma de branchement améliore significativement une approche par contraintes. En moyenne, on observe un facteur 511 sur le temps de résolution. Ce gain augmente avec la difficulté des instances traitées, jusqu'à quatre ordres de grandeurs (sur `bier127`). Le gain diminue ensuite artificiellement à cause de la limite de temps imposée. Nous avons donc réussi à décaler la barrière exponentielle du problème, permettant ainsi de résoudre des instances 50% plus grandes, jusqu'à un maximum d'environ 300 noeuds sur ces instances difficiles. Cette amélioration de passage à l'échelle est donc significative. Ces résultats sont compétitifs avec ceux obtenus par un solveur de programmation linéaire et un investissement raisonnable pour identifier des coupes. Néanmoins, il reste encore une belle marge de progression avant d'espérer être compétitif avec le solveur Concorde, dédié à la résolution du TSP, qui est capable de clore des instances ayant des milliers de noeuds.

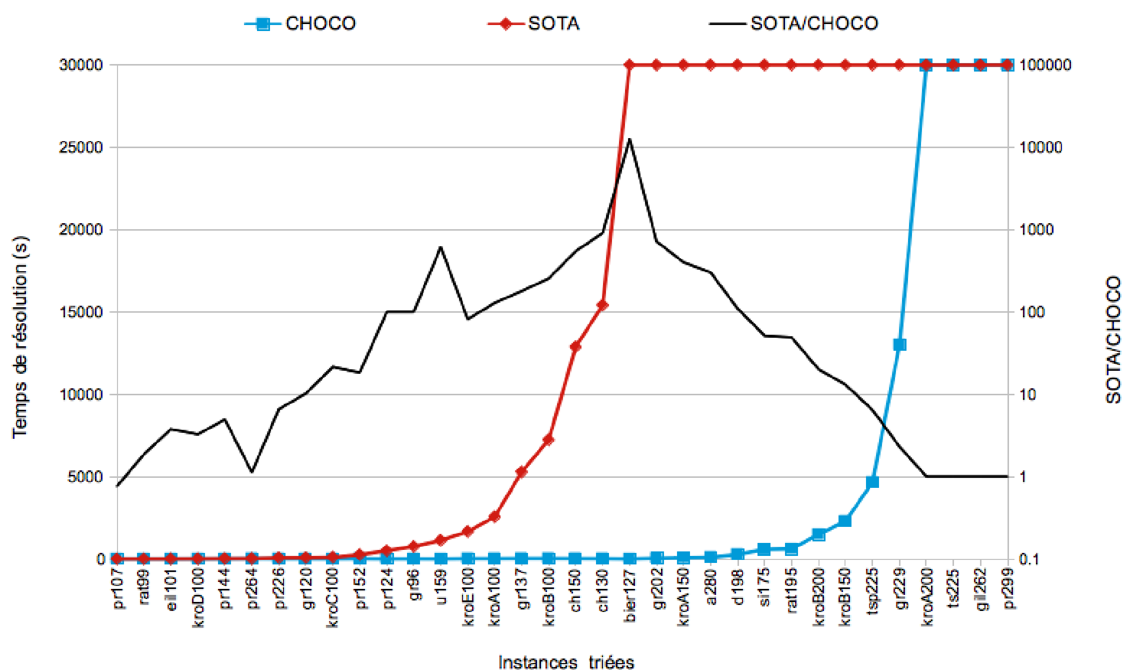


FIGURE 6.9 – Comparaison du temps de résolution (en secondes) de `SOTA` [BVHR$^+$12] et de `CHOCO` avec l'heuristique `MAX_COST` et `LC_FIRST`. Le facteur d'accélération est donné par le ratio du temps de résolution de `SOTA` sur celui de `CHOCO`.

Instance	CHOCO LC_FIRST + MAX_COST	CHOCO LC_FIRST + MIN_DELTA_DEG	SOTA [BVHR+12] (MAX_REP_COST)
pr107	1.5	1.5	**1.1**
rat99	**0.8**	**0.8**	1.5
eil101	**0.8**	1.0	3.0
kroD100	3.1	**0.9**	10.1
pr144	**4.8**	12.9	23.5
pr264	**21.0**	21.6	23.8
pr226	10.3	**6.4**	68.0
gr120	7.4	**3.3**	76.1
kroC100	**4.7**	5.9	101.9
pr152	**14.0**	89.2	258.1
pr124	**5.0**	12.7	503.0
gr96	7.5	**3.3**	754.2
u159	**1.8**	7.3	1126.5
kroE100	**20.3**	69.2	1661.1
kroA100	**19.9**	20.5	2556.4
gr137	**29.5**	33.7	5283.4
kroB100	28.4	**17.0**	7236.0
ch150	**23.6**	28.0	12875.7
ch130	**17.0**	72.0	15411.5
bier127	2.4	**1.8**	30000.0
gr202	41.9	**24.4**	30000.0
kroA150	**75.0**	119.9	30000.0
a280	**99.4**	806.6	30000.0
d198	273.1	**76.1**	30000.0
si175	**581.6**	4544.1	30000.0
rat195	610.0	**330.5**	30000.0
kroB200	**1490.2**	2218.9	30000.0
kroB150	2295.5	**1609.0**	30000.0
tsp225	4659.7	**4171.4**	30000.0
gr229	**12999.6**	14025.4	30000.0
kroA200	30000.0	30000.0	30000.0
ts225	30000.0	30000.0	30000.0
gil262	30000.0	30000.0	30000.0
pr299	30000.0	30000.0	30000.0

(a) Résultats bruts (le temps de résolution est donné en secondes).

FIGURE 6.10 – Comparaison de l'approche par contraintes de l'état de l'art avec notre approche, sur les instances difficiles de la TSPLIB et avec une limite de temps de 30000 secondes.

6.4 Conclusion du chapitre

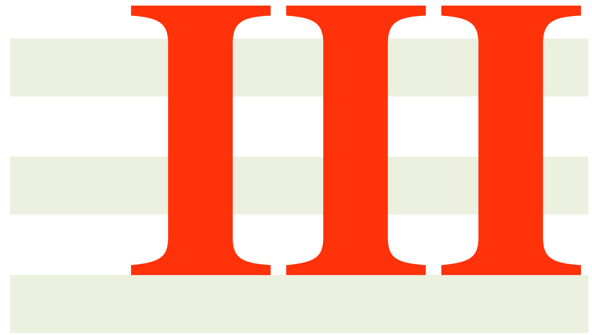
Nous avons montré l'intérêt d'utiliser une modélisation non orientée pour la résolution du problème du cycle Hamiltonien. Les variables de graphe permettent une telle représentation, avec en plus des structures de données très efficaces. On peut ainsi résoudre des instances du problème du cycle Hamiltonien ayant des milliers de noeuds (40000 dans le cas du problème du cavalier d'Euler) en quelques secondes à peine. Ces résultats améliorent par un facteur 10 environ la taille des instances pouvant être traitées en programmation par contraintes. Nous avons ainsi montré empiriquement que la programmation par contraintes était une méthode très efficace pour traiter le problème du cycle Hamiltonien, y compris sur des instances à large échelle. Plus généralement, ceci démontre l'intérêt des variables graphes, que ce soit en terme de modélisation, de combinatoire ou d'implémentation.

Nous avons également décrit une large gamme d'heuristiques pour brancher sur une variable de graphe ainsi que diverses manières de réutiliser le dernier conflit durant la recherche. Nous les avons comparés sur des instances du problème du voyageur de commerce. Il en résulte que des heuristiques très simples, basées sur le domaine de la variable graphe ou sur la fonction de coûts sont très efficaces, voire même meilleures que d'autres heuristiques basées sur des relaxations. Plus généralement, la réutilisation du dernier conflit permet d'améliorer drastiquement les résultats, quelle que soit l'heuristique de branchement retenue. Ce dernier point est sans doute le plus important. Dans l'ensemble, il résulte de nos travaux une amélioration significative des

meilleurs résultats obtenus en programmation par contraintes. Le schéma de branchement proposé, pourtant très simple, permet de résoudre des instances jusqu'à 50% plus grandes et apporte un facteur allant jusqu'à quatre ordres de grandeurs sur le temps de résolution du problème du voyageur de commerce.

Aller plus loin dans la résolution du TSP n'est pas une chose simple. L'idéal serait d'améliorer la puissance de filtrage du modèle utilisé, en trouvant un nouvel algorithme de filtrage. La qualité de nos résultats sur le problème du cycle Hamiltonien indique qu'il est nécessaire que cet algorithme intègre les coûts dans son raisonnement : la programmation par contraintes sait déjà très bien résoudre le problème sans coût. Peut-on encore améliorer la relaxation Lagrangienne des contraintes de degrés ? Peut-on trouver une autre relaxation qui soit encore plus puissante, sans pour autant avoir recours à la programmation linéaire ? Un autre axe d'amélioration du modèle en contraintes concerne la recherche d'une bonne solution. Dans notre étude, nous avons repris le modèle de [BVHR+12], qui utilise la meilleure méthode de recherche locale [RGJM07] pour initialiser le domaine de la variable modélisant l'objectif. Il serait bon de pouvoir s'abstraire de ce pré-calcul dédié et de reposer sur notre modèle en contraintes pour trouver une bonne borne. Cela ne nécessiterait pas de filtrage trop lourd, mais une bonne heuristique, voire une méta-heuristique (e.g., recherche à voisinage large [Sha98]). Enfin, après le filtrage et la recherche, l'explication des conflits forme une piste prometteuse. Le gain engendré par la réutilisation du dernier conflit dans la recherche, qui peut être vu comme une forme dégradée d'explication, nous encourage dans cette voie. S'il ne semble pas très compliqué d'expliquer le modèle orienté classique du TSP en contraintes, expliquer une contrainte globale basée sur une relaxation, telle que la relaxation 1-tree [RRRVH10, BVHR+12] nous parait ambitieux. La question est alors la suivante : sera-t-il rentable de combiner explications et contraintes globales sur les coûts ?

Enfin, nous constatons que pour être compétitif, il est très difficile de se passer de contraintes globales dédiées, souvent coûteuses à implémenter et à maintenir. Or, si ces contraintes sont une richesse et une force pour la programmation par contraintes, elles introduisent également des difficultés d'utilisation et des surcoûts algorithmiques. De plus, la puissance de filtrage d'une contrainte globale dépend fortement des contraintes qui l'entourent. Dans le prochain chapitre, nous allons nous intéresser à la contrainte globale NVALUE, qui compte le nombre de valeurs prises par un ensemble de variables. Nous verrons comment modifier simplement son algorithme de filtrage, basé sur un graphe, pour y inclure d'éventuelles contraintes de différences du modèle, offrant ainsi un filtrage plus puissant.



Des graphes pour les contraintes

7

Cliques et allocation de ressources

Sommaire

7.1 Description formelle du problème **78**

7.1.1 Un modèle mathématique 78

7.1.2 Un modèle en contraintes 79

7.2 Reformulation par la contrainte de graphe NCLIQUE **79**

7.2.1 Filtrage de la contrainte NCLIQUE 80

7.2.2 Prise en compte naturelle des contraintes de différences 81

7.2.3 Inconvénient de la méthode 81

7.3 Utilisation d'un graphe pour filtrer ATMOSTNVALUE **82**

7.3.1 Diversification du filtrage 83

7.3.2 Intégration d'inégalités 84

7.4 Etude expérimentale sur un problème de planification de personnel . . . **86**

7.4.1 Un branchement basé sur la réification de contraintes globales 87

7.4.2 Description des instances étudiées 88

7.4.3 Analyse empirique du modèle 90

7.4.4 Comparaison aux approches existantes 93

7.5 Conclusion du chapitre **97**

7.5.1 Perspectives 97

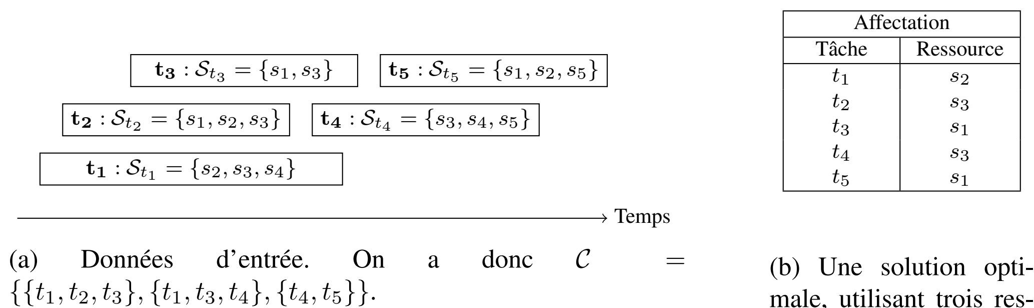
De nombreuses applications nécessitent d'optimiser l'affectation de ressources à des tâches. Par exemple, dans les transports, des équipages sont affectés à des trajets [LFM11]. Dans les aéroports, des équipes au sol sont assignées à des opérations de maintenance [DKS+94, DKMS97]. Dans les usines, des tâches sont affectées à des machines [FMT87, KSVW94]. Tous ces domaines sont sujets à une montée de la demande en parallèle de restrictions budgétaires. Dans ce contexte, minimiser l'utilisation des ressources, alors précieuses, est crucial. Nous allons voir dans cette section comment répondre efficacement à ce type de problématiques par une approche exacte en programmation par contraintes basée sur les graphes. Contrairement

au chapitre précédent, l'utilisation des graphes n'est ici pas directement issue de la nature du problème traité, dont la description ne parle pas de graphes, mais est ajoutée artificiellement pour offrir une vue d'ensemble sur un réseau de contraintes binaires implicites.

Dans un premier temps, nous introduirons formellement le problème étudié en section 7.1. Nous en donnerons un modèle mathématique ainsi qu'un modèle en programmation par contraintes, basé notamment sur la contrainte globale NVALUE[BC01, BHH+06]. Dans un second temps, nous étudierons en section 7.2 la reformulation de cette contrainte avec la contrainte de graphe NCLIQUE que nous introduirons. Puis, nous verrons en section 7.3 comment, toujours à partir de concepts liés aux graphes, il est possible d'améliorer directement le filtrage de la contrainte NVALUE. Ensuite, nous proposerons en section 7.4 une étude expérimentale pour valider notre propos. Nous décrirons un schéma de branchement original faisant appel à la réification de contrainte pour résoudre efficacement ce problème. Nous évaluerons chacune de nos contributions et nous montrerons que notre approche par contraintes permet d'égaler les meilleurs résultats connus de la recherche opérationnelle. Enfin, nous donnerons en section 7.5 une conclusion succincte de ce travail et détaillerons un ensemble de perspectives qui en émergent.

7.1 Description formelle du problème

Soit $\mathcal{T} \subseteq \mathbb{N}$ un ensemble de tâches à effectuer, fixées dans le temps, et soit $\mathcal{S} \subseteq \mathbb{N}$ un ensemble de ressources disponibles. Considérons le problème d'affectation de chaque tâche $t \in \mathcal{T}$ à une ressource $s \in \mathcal{S}$, utilisant un nombre minimum de ressources et respectant les contraintes suivantes : chaque tâche $t \in \mathcal{T}$ ne peut être réalisée que par un sous-ensemble des ressources qualifiées $\mathcal{S}_t \subseteq \mathcal{S}$; deux tâches effectuées en même temps doivent être affectées à des ressources différentes. Ce problème a été introduit dans un contexte de planification de personnel par Krishnamoorthy et d'autres [KEB12], sous le nom de SMPTSP (de l'anglais *Shift Minimisation Personnel Task Scheduling Problem*). Il a été prouvé NP-difficile [KEB12]. Nous notons également $\mathcal{C} \subseteq \mathcal{P}(\mathcal{T})$ l'ensemble des sous-ensembles, maximaux au sens de l'inclusion, des tâches en intersection temporelle. Puisque cela revient à calculer toutes les cliques, maximales au sens de l'inclusion, d'un graphe d'intervalle (donc triangulé), calculer $\mathcal{C}$ est polynomial. Un petit exemple introductif du SMPTSP est donné par la figure 7.1.



Affectation	
Tâche	Ressource
t_1	s_2
t_2	s_3
t_3	s_1
t_4	s_3
t_5	s_1

(a) Données d'entrée. On a donc $\mathcal{C} = \{\{t_1, t_2, t_3\}, \{t_1, t_3, t_4\}, \{t_4, t_5\}\}$.

(b) Une solution optimale, utilisant trois ressources.

FIGURE 7.1 – Illustration du SMPTSP, avec 5 ressources et 5 tâches

7.1.1 Un modèle mathématique

Le SMPTSP peut être modélisé en Programmation Linéaire en Nombres Entiers (PLNE) en utilisant des variables binaires $x_{t,s}$ et y_s, pour représenter respectivement si la tâche t est affectée

à la ressource s et si la ressource s est utilisée ou non. A partir de ces variables, on obtient le modèle de l'état de l'art [KEB12] en PLNE :

$$\begin{array}{rlll} \textit{Minimiser} & \sum_{s\in\mathcal{S}} y_s & & (1)\\ \textit{En respectant :} & \sum_{s\in\mathcal{S}_t} x_{t,s} = 1 & ,\forall t \in \mathcal{T} & (2)\\ & \sum_{t\in C} x_{t,s} \leq y_s & ,\forall s \in \mathcal{S}, \forall C \in \mathcal{C} & (3)\\ & x_{t,s} \in \{0,1\} & ,\forall t \in \mathcal{T}, \forall s \in \mathcal{S}_t & (4)\\ & y_s \in \{0,1\} & ,\forall s \in \mathcal{S} & (5) \end{array}$$

Le nombre de ressources utilisées est donné par (1). L'affectation des tâches à des ressources compétentes est assuré par (2). La contrainte (3) assure deux propriétés : Tout d'abord, elle interdit à toute ressource d'effectuer plus d'une tâche à la fois. Ensuite, elle assure que les ressources affectées à des tâches sont bien comptées comme étant utilisées. Ce modèle est connu pour être très efficace sur de petites et moyennes instances, mais il rencontre de sérieuses difficultés à résoudre des instances à plusieurs centaines de tâches. Aussi, nous allons maintenant investiguer ce que donne une approche en programmation par contraintes sur ce problème.

7.1.2 Un modèle en contraintes

La programmation par contraintes permet d'adresser ce problème avec un modèle déclaratif concis. On peut associer chaque tâche $t \in \mathcal{T}$ à une variable entière $x_t \in \mathcal{X}$ indiquant à quelle ressource affecter t et introduire une variable entière N pour compter le nombre de valeurs prises dans $\mathcal{X}$. On peut ainsi établir le modèle suivant :

$$\begin{array}{llll} \textit{Minimiser :} & N & & \\ \textit{En respectant :} & \text{NVALUE}(\mathcal{X}, N) & & (1)\\ & \text{ALLDIFFERENT}(\{x_i \mid i \in C\}) & \forall C \in \mathcal{C} & (2)\\ & \max_{C\in\mathcal{C}} |C| \leq N \leq |\mathcal{S}| & & (3)\\ & x_t \in S_t & \forall t \in \mathcal{T} & (4) \end{array}$$

Ce modèle emploie une contrainte NVALUE [BHH+06] pour compter le nombre de valeurs prises dans $\mathcal{X}$ (1) et plusieurs contraintes ALLDIFFERENT [Rég94] afin d'imposer efficacement que deux tâches en intersection dans le temps soient affectées à des ressources différentes (2). Les contraintes (3) et (4) indiquent quant à elles comment initialiser les domaines des variables.

Nous allons maintenant voir deux façons, fortement corrélées, de filtrer la contrainte NVALUE efficacement dans ce contexte. La première reposera sur une reformulation basée sur une variable graphe à partitionner en cliques. La deuxième portera sur une amélioration, simple mais significative, de l'algorithme de l'état de l'art.

7.2 Reformulation par la contrainte de graphe NCLIQUE

Compter le nombre de valeurs distinctes prises par $\mathcal{X}$ revient à répartir les variables en classes d'équivalence et à compter le nombre de ces classes. La théorie des graphes propose une montée en abstraction pour résoudre ce problème, grâce au partitionnement en cliques. En effet, on peut créer une variable graphe non-orientée $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ pour représenter la relation d'équivalence sur $\mathcal{X}$. La variable graphe sert ici à réifier un réseau de contraintes binaires, pour en obtenir une vue d'ensemble et y appliquer des propriétés (de graphe) globales. On parle alors de graphe de réification. Plus précisément, chaque variable $x \in \mathcal{X}$ est associée à un noeud $v_x \in \mathcal{V}$, avec $\overline{\mathcal{V}} = \underline{\mathcal{V}}$ et $|\overline{\mathcal{V}}| = |\mathcal{X}|$, et chaque arête $(i, j) \in \mathcal{E}$ réifie la contrainte $x_i = x_j$, *i.e.*, $(i, j) \in \mathcal{E} \Leftrightarrow x_i = x_j$. $\mathcal{G}$ doit ensuite former un ensemble d'au plus N cliques :

$$\text{NVALUE}(\mathcal{X}, N) \Leftrightarrow \text{GRAPHREIF}(\mathcal{G}, \mathcal{X}, =) \wedge \text{NCLIQUE}(\mathcal{G}, N)$$

Nous introduisons les contraintes GRAPHREIF (définition 3) et NCLIQUE (définition 4), pour respectivement lier les variables entières $\mathcal{X}$ et la variable graphe $\mathcal{G}$, en réifiant un réseau de contraintes binaires (des égalités) portant sur $\mathcal{X}$, et contraindre $\mathcal{G}$ a constituer N cliques. Nous ne détaillons pas le fonctionnement de GRAPHREIF puisqu'il est parfaitement analogue à la réification d'une contrainte d'égalité sur chaque paire de variables (les variables booléennes de réification remplaçant les arêtes de $\mathcal{G}$). Nous allons en revanche détailler un algorithme de filtrage efficace pour la contrainte NCLIQUE.

Définition 3 *Etant donné une variable graphe $\mathcal{G}$, un ensemble de variables $\mathcal{X}$ et une relation binaire R, la contrainte* GRAPHREIF$(\mathcal{G}, \mathcal{X}, R)$ *est satisfaite si et seulement si $(i, j) \in \mathcal{G} \Leftrightarrow x_i R x_j$, pour tout $x_i, x_j \in \mathcal{X}$*

Définition 4 *Etant donné une variable graphe $\mathcal{G}$ et une variable entière N, la contrainte* NCLIQUE$(\mathcal{G}, N)$ *est satisfaite si et seulement si $\mathcal{G}$ forme un ensemble de N cliques.*

7.2.1 Filtrage de la contrainte NCLIQUE

Filtrer la contrainte NCLIQUE se décompose en deux étapes : le filtrage structurel assure que la variable graphe $\mathcal{G}$ forme effectivement une partition en cliques ; le filtrage de cardinalité contrôle la cardinalité de cette partition.

Filtrage structurel : faire des cliques

Pour partitionner $\mathcal{G}$ en cliques, il faut filtrer $\mathcal{G}$ en fonction des propriétés intrinsèques à la relation d'équivalence comme indiqué dans la table 7.1.

Propriété	Règle de filtrage
Réflexivité	$\forall i \in \overline{\mathcal{V}}, i \in \underline{\mathcal{V}} \Rightarrow (i, i) \in \underline{\mathcal{E}}$
Symétrie	$\mathcal{G}$ est non-orienté donc cette propriété est implicitement assurée
Transitivité	$\forall \{i, j, k\} \in \overline{\mathcal{V}}^3, (i, j) \in \underline{\mathcal{E}} \wedge (j, k) \in \underline{\mathcal{E}} \Rightarrow (i, k) \in \underline{\mathcal{E}}$

TABLE 7.1 – Filtrage de la relation d'équivalence sur une variable graphe $\mathcal{G} = (\mathcal{V}, \mathcal{E})$.

Assurer la réflexivité se fait très simplement. Puisqu'ici $\overline{\mathcal{V}} = \underline{\mathcal{V}}$, il suffit d'ajouter une boucle à chaque noeud dans $\underline{\mathcal{G}}$. Cela se fait donc en une seule passe, avant même de démarrer le processus de recherche. La symétrie est assurée implicitement par le caractère non-orienté de $\mathcal{G}$. En revanche, la transitivité peut être coûteuse à propager. Naïvement, puisqu'elle implique des triplets de noeuds, une étape de propagation nécessite $O(n^3)$ opérations. De plus, en cas de filtrage, la règle doit se rappeler elle-même pour converger à son point fixe et faire toutes les déductions possibles. En pratique, on peut améliorer la propagation de cette règle avec une implémentation incrémentale. Chaque ajout ou retrait d'arête est alors propagé en $O(n)$, en itérant sur les voisins de ses extrémités.

Filtrage de cardinalité

Dans des travaux antérieurs, Beldiceanu et d'autres [BPR06] proposent d'utiliser une contrainte sur le nombre de composantes (fortement) connexes de $\mathcal{G}$ afin de contraindre le nombre de cliques.

Estimation du nombre maximum de cliques : Puisque chaque noeud n'ayant aucun voisin dans $\underline{\mathcal{G}}$ peut former une clique singleton, le nombre maximum de cliques est maintenu par la règle suivante :

$$N \leq \overline{\mathcal{V}} - \underline{\mathcal{V}} + |ConnectedComponents(\underline{\mathcal{G}})|$$

Calculer cette borne requiert $O(n+|\underline{\mathcal{G}}|)$, ce qui est inférieur ou égal à $O(n+m)$. On peut ensuite rétro-propager cette borne dans le cas où N est instanciée à $\overline{\mathcal{V}}-\underline{\mathcal{V}}+|ConnectedComponents(\underline{\mathcal{G}})|$, en ajoutant tous les noeuds de $\overline{\mathcal{V}}$ à $\underline{\mathcal{V}}$ (et donc en ajoutant à $\underline{\mathcal{E}}$ les boucles qui y sont associées) et en retirant ensuite toutes les arêtes de $\overline{\mathcal{E}} \setminus \underline{\mathcal{E}}$.

Estimation du nombre minimal de cliques : Contrairement au cas précédent, estimer le nombre minimal de cliques dans un graphe quelconque est NP-difficile [GJ79]. Il convient donc d'employer une approche heuristique. L'approximation du nombre minimal de cliques par le nombre minimal de composantes connexes ne donne pas de bons résultats en pratiques. La meilleure approche actuelle consiste à calculer un ensemble de noeuds indépendants aussi grand que possible dans $\overline{\mathcal{G}}\cap\underline{\mathcal{V}}$, à l'aide d'une heuristique. Notons f la fonction renvoyant un ensemble indépendant du graphe donné en argument. On obtient alors une borne inférieure sur le nombre de cliques :

$$N \geq |f(\overline{\mathcal{G}} \cap \underline{\mathcal{V}})|$$

En cas d'égalité, *i.e.,* si N est instanciée à la taille de l'ensemble indépendant calculé, on peut alors rétro-propager cette borne sur le domaine de $\mathcal{G}$. Soit $I = f(\overline{\mathcal{G}} \cap \underline{\mathcal{V}})$ cet ensemble indépendant, on peut ainsi filtrer la contrainte suivante : $\forall i \in \mathcal{V}$, $|\delta_{\mathcal{G}}(i) \cap I| > 0$. Ainsi, les noeuds adjacents à aucun élément de l'ensemble indépendant peuvent être filtrés de $\overline{\mathcal{V}}$ et, si un noeud $i \in \underline{\mathcal{V}}$ est adjacent à un unique élément j de I, alors l'arête (i, j) peut être ajoutée à $\underline{\mathcal{E}}$. Ce simple ajout d'arête représente la découverte d'une contrainte d'égalité entre les deux variables. On notera ainsi que la variable graphe simplifie grandement l'apprentissage de ces contraintes.

7.2.2 Prise en compte naturelle des contraintes de différences

Il est important de noter que l'utilisation d'une contrainte de réification GRAPHREIF$(\mathcal{G}, \mathcal{X}, =)$ met en évidence la négation de ces égalités, et donc l'importance des éventuelles contraintes de différences du modèle. Ainsi, si une contrainte $x_i \neq x_j$ est posée, alors, nécessairement, la contrainte $x_i = x_j$ sera fausse dans chacune des solutions. Ce raisonnement très simple permet donc de détecter, dès le noeud racine de l'arbre de recherche, un grand nombre de contraintes réifiées d'égalité qui ne pourront jamais être satisfaites. Puisque cette inférence se matérialise par un filtrage des arêtes de $\mathcal{G}$, elle se propage à la contrainte NCLIQUE.

7.2.3 Inconvénient de la méthode

L'inconvénient principal lié à la reformulation de NVALUE avec NCLIQUE est que la contrainte NCLIQUE n'a pas accès aux variables (entières) d'affectation. Les ajouts et suppressions d'arêtes sont propagés sur ces variables via la réification du réseau d'égalités binaires qui les lient, mais une partie du filtrage est perdue. Par exemple, la relaxation par un ensemble indépendant pourrait être rétro-propagée plus finement si ces variables étaient directement accessibles. Bien sûr, on pourrait concevoir une grosse contrainte globale embarquant à la fois la variable graphe et les variables de décision, mais on perdrait alors toute l'élégance et la flexibilité de la reformulation. C'est l'une des principales raisons du choix de modélisation qui a été retenu dans les articles [FL13, FL14].

7.3 Utilisation d'un graphe pour filtrer ATMOSTNVALUE

La contrainte NVALUE a été introduite par [PR99] et développée par [BC01, BHH+06]. Cette contrainte permet de contrôler le nombre de valeurs distinctes prises par un ensemble de variables. Elle peut naturellement se décomposer en ATLEASTNVALUE et ATMOSTNVALUE qui contrôlent respectivement le nombre minimum et le nombre maximum de valeurs distinctes. Alors qu'assurer la GAC sur ATLEASTNVALUE peut se faire en temps polynomial [PRB01], assurer la GAC sur ATMOSTNVALUE se ramène à résoudre un problème NP-difficile [BHH+06]. Puisque nous cherchons à minimiser le nombre de valeurs, nous nous focaliserons sur la contrainte ATMOSTNVALUE.

Nous rappelons ici que le graphe d'intersection des domaines d'un ensemble de variables $\mathcal{X}$ est noté $G_{\mathcal{I}}(\mathcal{X})$. Nous donnons également une illustration du graphe d'intersection des variables de notre exemple sur la figure 7.2b. Par souci de concision, et puisqu'il n'y a ici qu'un seul ensemble de variables, nous le noterons simplement $G_{\mathcal{I}}$.

L'algorithme de filtrage proposé par Bessière *et al.* [BHH+06] pour la contrainte ATMOSTNVALUE est basé sur le fait que la taille d'un ensemble indépendant maximum dans $G_{\mathcal{I}}$ est inférieure à la cardinalité minimum d'une partition en clique de $G_{\mathcal{I}}$. Soit I un ensemble indépendant de $G_{\mathcal{I}}$, il peut alors être utilisé pour filtrer la variable comptant le nombre de valeurs, selon la règle suivante :

$$\mathcal{R}_1 : N \geq |I|$$

En cas d'égalité, il est possible de rétro-propager cette borne sur les domaines des variables de décision à l'aide de la règle $\mathcal{R}_2$, qui indique que l'ensemble des valeurs admissibles est induit par I. Les variables n'appartenant pas à l'ensemble indépendant I doivent prendre leurs valeurs dans les domaines des variables de I :

$$\mathcal{R}_2 : N = |I| \Rightarrow \forall i \in \mathcal{V} \setminus I,\ x_i \in \bigcup_{j \in I} dom(x_j)$$

Complexité : Notons l l'ensemble des valeurs. Nous supposons que l'intersection et l'union de deux domaines de variables entières peuvent être calculées en $O(\log(l))$, à l'aide d'opérations basées sur des vecteurs de bits. L'ensemble de valeurs $\bigcup_{j \in I} dom(x_j)$ peut être calculé une fois pour toutes, en effectuant $O(n)$ unions de domaines, soit une complexité de $O(n \log(l))$. Ensuite, le filtrage d'une variable peut s'effectuer avec une opération d'intersection de deux domaines, soit une complexité de $O(\log(l))$. Il y a n variables à filtrer, donc la complexité totale pour filtrer la règle $\mathcal{R}_2$ est de $O(n \log(l))$ dans le pire cas.

On pourrait calculer tous les ensembles indépendants maxima avec l'algorithme de Bron-Kerbosch [BK73] afin de filtrer le plus possible. Cependant, ce ne serait pas réaliste, car appliquer un algorithme exponentiel à chaque propagation de la contrainte pénaliserait sérieusement la résolution et réduirait à néant toute ambition de passage à l'échelle. En pratique, le propagateur calcule un ensemble indépendant, aussi grand que possible, de manière gloutonne. Concrètement, la méthode employée est l'heuristique des degrés minimaux (MD), qui sélectionne successivement les noeuds de plus faibles degrés [HR94] afin de créer un ensemble indépendant (algorithme 2). L'algorithme peut être codé de manière simple et efficace en $O(n^2)$. Avec une implémentation plus avancée, basée sur une structure de tas binaire pour récupérer le noeud de plus petit degré, on peut descendre la complexité temporelle à $O(m \log(n))$. On aura en effet $O(n)$ requêtes de retrait du plus petit élément et $O(m)$ mises à jour de la valeur d'un élément (lorsque l'on retire un noeud du graphe, le degré de ses voisins diminue).

Algorithm 2 MD : une heuristique pour calculer un ensemble indépendant dans un graphe

```
1:  function MD(Graph G)
2:      G' ← G.clone()                                  ▷ Copie le graphe d'entrée G
3:      I ← ∅                      ▷ Créé un ensemble indépendant I, initialement vide
4:      while (G' ≠ ∅) do
5:          x ← smallestDegreeVertex(G')   ▷ Sélectionne un noeud x parmi ceux de plus
    faible degré dans G'
6:          I ← I ∪ {x}                              ▷ Ajoute ce noeud x à l'ensemble I
7:          G' ← G' \ {y | x ≠ y ∧ (x, y) ∈ G'}       ▷ Retire de G' tous les voisins de x
8:          G' ← G' \ {x}                                          ▷ Retire x de G'
9:      end while
10: return I                   ▷ Renvoie un ensemble indépendant du graphe d'entrée G
11: end function
```

En montant en abstraction, on peut dire que l'algorithme de filtrage pour la contrainte ATMOSTNVALUE crée un graphe G, appelle une fonction f qui calcule un ou plusieurs ensembles indépendants sur G pour filtrer les domaines de ses variables à l'aide d'un ensemble de règles $\mathcal{R}$. Nous introduisons la notation AMNV$\langle G|f|\mathcal{R}\rangle$ pour définir cette classe de propagateurs (AMNV servant d'acronyme pour ATMOSTNVALUE). En conséquence, le propagateur de [BHH+06] est défini par AMNV$\langle G_{\mathcal{I}}|\texttt{MD}|\mathcal{R}_{1,2}\rangle$. Par la suite, nous allons modifier G, f et $\mathcal{R}$ pour aboutir à un nouveau propagateur, plus performant et capable de prendre en compte un ensemble de contraintes de différences.

7.3.1 Diversification du filtrage

Une première amélioration possible concerne la méthode de calcul de l'ensemble indépendant. En effet, aussi bonne soit-elle, l'heuristique MD souffre d'un défaut important : elle manque de diversification. Tout d'abord, elle peut être mal adaptée à certaines instances pathologiques. Ensuite, même si sa borne est bonne, elle risque de calculer des ensembles indépendants très similaires d'une propagation à l'autre, ce qui restreint la rétro-propagation.

Pour palier cet inconvénient, nous introduisons la méthode $\texttt{R}^k$ (voir l'algorithme 3). Cette heuristique calcule aléatoirement k ensembles indépendant, chacun étant maximal au sens de l'inclusion. La méthode de calcul d'un ensemble indépendant I diffère de MD par sa fonction de sélection d'un noeud à ajouter à I : MD choisi un des noeuds dont le degré est minimal ; $\texttt{R}^k$ choisi aléatoirement un noeud. Cette méthode offre donc une grande diversification. En revanche, rien ne garantit que les k ensembles calculés soient tous différents. De plus, à cause de son caractère aléatoire, un propagateur basé sur $\texttt{R}^k$ ne sera ni idempotent, ni monotone [ST09].

D'un point de vue théorique, cette méthode offre quelques propriétés intéressantes. D'abord, elle propose naturellement un certain contrôle sur le compromis entre la qualité et le temps accordé au filtrage. Ensuite, calculer des ensembles aléatoirement permet de filtrer les variables de décision de manière homogène. Il est bon de remarquer que lorsque k tend vers l'infini, alors la méthode tend à énumérer tous les ensembles indépendants de cardinalité maximale. Puisque l'application de chaque règle de filtrage de AMNV dépend de l'ensemble indépendant considéré, faire tendre k vers l'infini engendre un filtrage maximal pour AMNV.

D'un point de vue pratique, on a intérêt à limiter le nombre k afin que le temps passé dans l'algorithme soit raisonnable. Fixer $k = 30$ semble un bon paramétrage par défaut. De plus, afin d'avoir de bonnes garanties de fonctionnement, il est préférable d'appliquer MD puis $\texttt{R}^k$, menant ainsi à $k + 1$ ensembles indépendants. En effet, les deux méthodes sont complémentaires : MD

ne donne jamais de très mauvais résultats mais reste néanmoins limitée sur certains types de graphes, alors que $\texttt{R}^k$ est capable du pire comme du meilleur, quelque soit le graphe considéré.

Algorithm 3 $\texttt{R}^k$: une heuristique pour calculer k ensembles indépendants dans un graphe

1: **function** MD(Graph G)
2: $\mathcal{I} \leftarrow \emptyset$ ▷ Créé un ensemble d'ensembles indépendants, initialement vide
3: **for** $(i \in [1, k])$ **do**
4: $G' \leftarrow G.clone()$ ▷ Copie le graphe d'entrée G
5: $I \leftarrow \emptyset$ ▷ Créé un ensemble indépendant I, initialement vide
6: **while** $(G' \neq \emptyset)$ **do**
7: $x \leftarrow randomNode(G')$ ▷ Sélectionne aléatoirement un noeud x dans G'
8: $I \leftarrow I \cup \{x\}$ ▷ Ajoute ce noeud x à l'ensemble I
9: $G' \leftarrow G' \setminus \{y \mid x \neq y \wedge (x, y) \in G'\}$ ▷ Retire de G' tous les voisins de x
10: $G' \leftarrow G' \setminus \{x\}$ ▷ Retire x de G'
11: **end while**
12: $\mathcal{I} \leftarrow \mathcal{I} \cup \{I\}$ ▷ Ajoute l'ensemble indépendant I à $\mathcal{I}$
13: **end for**
14: **return** $\mathcal{I}$ ▷ Renvoie k ensembles indépendants du graphe d'entrée G
15: **end function**

7.3.2 Intégration d'inégalités

Nous allons maintenant montrer une manière simple et élégante d'inclure des contraintes de différences dans notre raisonnement. Sans perte de généralité, nous considérons ici des contraintes de différence binaires, qu'elles soient explicites ou induites par d'autres contraintes (e.g., ALL-DIFFERENT, ou encore des contraintes *ad hoc* impliquant des différences entre certaines variables). Pour cela, nous introduisons la définition de graphe d'intersection contraint (voir la définition 5). Cette définition est illustrée par la figure 7.2c sur notre exemple courant. De même que pour $G_{\mathcal{I}}$, nous noterons le graphe d'intersection contraint $G_{\mathcal{CI}}$.

Définition 5 *Le graphe d'intersection contraint $G_{\mathcal{CI}}(\mathcal{X}, \mathcal{D})$, d'un ensemble de variables $\mathcal{X}$ et d'un ensemble de contraintes de différences $\mathcal{D}$, est la généralisation du graphe d'intersection $G_{\mathcal{I}}(\mathcal{X})$ à la prise en compte de contraintes $\mathcal{D}$ interdisant des intersections de domaines. Il est défini par $G_{\mathcal{CI}}(\mathcal{X}, \mathcal{D}) = G_{\mathcal{I}}(\mathcal{X}) \setminus \{(i, j) | (x_i \neq x_j) \in \mathcal{D}\}$.*

Nous allons alors modifier notre propagateur en remplaçant $G_{\mathcal{I}}$ par $G_{\mathcal{CI}}$: conduisant ainsi à un nouveau propagateur pour la contrainte ATMOSTNVALUE. L'intérêt de $G_{\mathcal{CI}}$ par rapport à $G_{\mathcal{I}}$ est illustré sur les figures 7.2d et 7.2e.

Proposition 3 *Soit une fonction f qui calcule des ensembles indépendants dans un graphe,* AMNV$\langle G_{\mathcal{CI}} | f | \mathcal{R}_{1,2} \rangle$ *filtre la contrainte* ATMOSTNVALUE.

Preuve : Chaque ensemble indépendant dans $G_{\mathcal{CI}}$ correspond à un ensemble de variables qui prendront des valeurs distinctes, soit parce que leurs domaines sont déjà disjoints, soit parce que des contraintes de différences l'imposent. La taille de cet ensemble est donc une borne valide sur le nombre de valeurs distinctes qui seront prises dans $\mathcal{X}$. □

Comme indiqué dans la proposition 4, $G_{\mathcal{CI}}$ offre une meilleure borne inférieure au nombre de valeurs. Il conduit donc à un filtrage plus fort. Pour rappel, la fonction α donne la taille du plus grand ensemble indépendant dans le graphe considéré en paramètre.

Proposition 4 $\alpha(G_{\mathcal{I}}) \leq \alpha(G_{\mathcal{CI}})$

Preuve : D'après la définition 5, $G_{\mathcal{CI}} \subseteq G_{\mathcal{I}}$, donc $\alpha(G_{\mathcal{I}}) \leq \alpha(G_{\mathcal{CI}})$. □

On peut maintenant raffiner la rétro-propagation en introduisant une nouvelle règle de filtrage ($\mathcal{R}_3$), en utilisant la notation $I_i = \{j \in I \mid (i,j) \in G_{\mathcal{CI}}\}$, avec $i \in \mathcal{V} \setminus I$:

$$\mathcal{R}_3 : N = |I| \Rightarrow \forall i \in \mathcal{V} \setminus I, \begin{cases} x_i \in \bigcup\limits_{j \in I_i} dom(x_j) \\ I_i = \{j\} \Rightarrow x_i = x_j \end{cases}$$

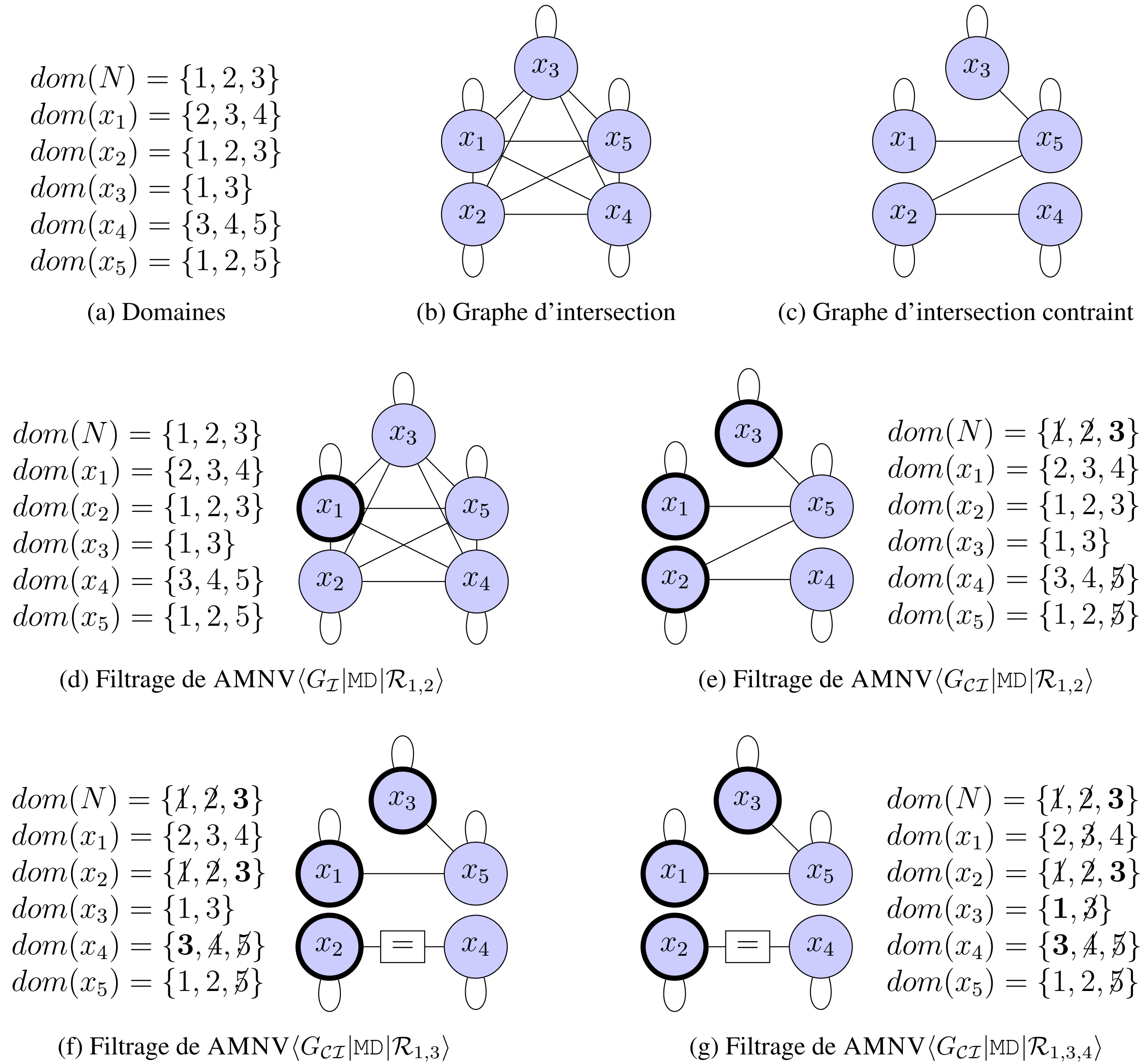
La règle $\mathcal{R}_3$ est un raffinement de $\mathcal{R}_2$ qui permet une rétro-propagation prenant elle aussi en compte les contraintes de différences du modèle. Elle vient donc remplacer $\mathcal{R}_2$ pour augmenter la puissance du filtrage. Dans certains cas, elle permet même d'apprendre des contraintes d'égalités devant être respectées. En effet, si un ensemble indépendant I est de taille N, et s'il existe un noeud $i \in \mathcal{V} \setminus I$ n'ayant qu'un seul voisin $j \in I$ dans $G_{\mathcal{CI}}$, alors les variables i et j doivent prendre la même valeur. Autrement, les variables prendraient au moins $|I| + 1$ valeurs, ce qui est strictement plus grand que N et donc impossible. Propager l'égalité alors apprise permet de réduire les domaines de i et de j. De plus, c'est une information forte qui pourrait améliorer d'autres raisonnements globaux. Une illustration de $\mathcal{R}_3$ est donnée par la figure 7.2f.

Enfin, dans le cas où les contraintes de différences sont exprimées sous formes de simples contraintes binaires (et non via des contraintes globales comme ALLDIFFERENT ou SOMEDIFFERENT), la règle de filtrage $\mathcal{R}_4$ permet d'avoir une vue d'ensemble de ce réseau de différences pour en tirer un filtrage plus fort :

$$\mathcal{R}_4 : \text{ALLDIFFERENT}(\{x_i \mid i \in I\})$$

L'application de cette règle de filtrage est illustrée sur la figure 7.2g.

Le niveau de cohérence du filtrage ainsi obtenu est compris entre le niveau de cohérence de la propagation du réseau de différences binaires et le niveau de cohérence obtenu si toutes les cliques maximales, au sens de l'inclusion, de différences étaient propagées de manière globale, chacune avec une contrainte ALLDIFFERENT. Bien entendu, si toutes ces cliques sont déjà connues et modélisées par des contraintes ALLDIFFERENT, alors la règle $\mathcal{R}_4$ est inutile. De plus, si ce n'est pas le cas, il peut être préférable d'effectuer une phase de reformulation du modèle, dans un pré-traitement, pour poser de telles contraintes. Cette approche statique est notamment employée par Gualandi et Malucelli [GM12] pour résoudre un problème de coloration de graphe. Plus généralement, la reformulation automatique de cliques de différences en contraintes ALLDIFFERENT, avec l'algorithme de Bron-Kerbosch [BK73] par exemple, est une pratique connue. Néanmoins, si ces contraintes binaires sont créées dynamiquement durant la résolution, alors ce pré-traitement statique n'est plus pertinent et $\mathcal{R}_4$ prend alors tout son sens.



(a) Domaines (b) Graphe d'intersection (c) Graphe d'intersection contraint

(d) Filtrage de AMNV$\langle G_{\mathcal{I}}|\texttt{MD}|\mathcal{R}_{1,2}\rangle$ (e) Filtrage de AMNV$\langle G_{\mathcal{CI}}|\texttt{MD}|\mathcal{R}_{1,2}\rangle$

(f) Filtrage de AMNV$\langle G_{\mathcal{CI}}|\texttt{MD}|\mathcal{R}_{1,3}\rangle$ (g) Filtrage de AMNV$\langle G_{\mathcal{CI}}|\texttt{MD}|\mathcal{R}_{1,3,4}\rangle$

FIGURE 7.2 – Illustration des propagateurs de type AMNV. Les noeuds de l'ensemble indépendant calculé par MD sont entourés. Les valeurs filtrées sont barrées. Seul un appel à l'algorithme de filtrage d'AMNV est représenté ici ; la propagation d'autres contraintes n'est pas prise en compte afin de ne pas bruiter notre analyse.

7.4 Etude expérimentale sur un problème de planification de personnel

Pour illustrer notre propos, nous appliquons notre méthode à la résolution du SMPTSP. Pour rappel, il s'agit d'un problème d'allocation de ressources humaines dont le modèle en programmation par contraintes est donné en section 7.1. Nous allons dans un premier temps proposer une heuristique de branchement adaptée à la résolution de ce problème. Ensuite, nous présenterons une analyse détaillée des instances traitées. Enfin, nous donnerons les résultats d'une vaste étude expérimentale permettant, d'une part, de bien comprendre le fonctionnement du modèle à contraintes et, d'autres part, de le positionner aux approches existantes en recherche opérationnelle.

7.4.1 Un branchement basé sur la réification de contraintes globales

Filtrage et branchement sont étroitement liés, si bien qu'on ne peut pas chercher à améliorer un modèle de programmation par contraintes en occultant la phase de recherche. Puisqu'ici la rétro-propagation nécessite une borne supérieure de N qui soit très basse, nous allons mettre en place une optimisation par le bas (↑). Cela se fait naturellement via la stratégie de branchement qui commencera par instancier N à sa borne inférieure courante. Ensuite, les variables de $\mathcal{X}$ seront instanciées une à une en sélectionnant à chaque noeud de branchement la variable de plus petit domaine et en lui affectant une valeur déjà prise par d'autres variables (quand cela est possible). Ainsi, la première solution trouvée est nécessairement optimale.

Remarquons que ce branchement n'est efficace que si l'on dispose d'un filtrage puissant offrant une très bonne borne inférieure. Sans cela, il faut privilégier une optimisation par le haut, comme cela se fait par défaut dans les outils de résolution de contraintes.

Nous allons maintenant détailler une spécificité de notre heuristique de branchement, liée à la réification de contraintes.

La chance sourit aux audacieux

Il est bon de noter que, si les ressources étaient toutes identiques (si les valeurs dans $\mathcal{X}$ étaient toutes interchangeables) alors, par rapprochement avec le problème de coloration d'un graphe triangulé, le problème serait polynomial. Soit N^* la valeur de l'optimum, nous pourrions utiliser n'importe quel ensemble de ressources de taille N^*. En outre, les N^* premières ressources, associées à l'ensemble de valeur $[1, N^*]$, conviendraient à coup sûr. Il serait donc pertinent d'ajouter au modèle la contrainte $\text{MAX}(\mathcal{X}) = N$ afin de casser ces symétries de valeur. Malheureusement, les ressources de notre problème ne sont pas identiques, car elles représentent des employés ayant des compétences et horaires de travail différents. Dès lors, un sous-ensemble quelconque de N^* valeurs n'appartient pas nécessairement à une solution optimale. Ceci vaut pour l'intervalle $[1, N^*]$. Il n'est donc pas correct de poster la contrainte $\text{MAX}(\mathcal{X}) = N$.

Cela étant dit, il se pourrait quand même que cette contrainte soit vérifiée dans une solution optimale de l'instance traitée. Ce phénomène peut en effet se produire lorsque la répartition des compatibilités tâche-ressource est homogène : alors que, localement, les ressources diffèrent toutes, globalement, aucune ne se démarque particulièrement des autres, au point que les N^* premières ressources sont suffisamment diversifiées pour couvrir l'ensemble des besoins. Pour cette raison, nous allons quand même inclure la contrainte $\text{MAX}(\mathcal{X}) = N$ à notre modèle, mais en la réifiant. Pour cela, nous introduisons la variable binaire b_{max}, qui prend pour valeur 1 ou 0 selon que la contrainte est satisfaite ou violée. Nous changeons également légèrement notre heuristique de recherche pour que, une fois la variable objectif fixée, le solveur commence par fixer b_{max} à 1 pour propager la contrainte qui y est associée et ainsi réduire drastiquement l'espace de recherche. S'il n'existe pas de solution, alors le mécanisme de backtrack va naturellement remonter l'arbre de recherche et fixer b_{max} à 0, signifiant alors que $\text{MAX}(\mathcal{X}) \neq N$. La réification de contrainte nous permet donc de faire des paris osés lors de l'exploration de notre arbre de recherche, tout en préservant l'exactitude de notre approche.

Illustrons maintenant sur la figure 7.3 l'utilisation de b_{max} sur notre exemple. Fixer cette variable booléenne à 1 permet de supprimer les valeurs 4 et 5 des domaines de toutes les variables de décision. Cela permettra de vite trouver une solution optimale ($\{x_1 = 2, x_2 = 3, x_3 = 1, x_4 = 3, x_5 = 1\}$ ou $\{x_1 = 2, x_2 = 3, x_3 = 1, x_4 = 3, x_5 = 2\}$). Si l'on cherche à énumérer toutes les solutions optimales, alors le mécanisme de backtrack permettra de remettre en cause le choix effectué sur b_{max} et ainsi poser $\text{MAX}(\mathcal{X}) \neq N$, permettant alors de trouver la solution $\{x_1 = 4, x_2 = 3, x_3 = 1, x_4 = 3, x_5 = 1\}$.

$$
\begin{aligned}
dom(N) &= \{1,2,3\}\\
dom(b_{max}) &= \{0,1\}\\
dom(x_1) &= \{2,3,4\}\\
dom(x_2) &= \{1,2,3\}\\
dom(x_3) &= \{1,3\}\\
dom(x_4) &= \{3,4,5\}\\
dom(x_5) &= \{1,2,5\}
\end{aligned}
$$

(a) Domaines initiaux

$$
\begin{aligned}
dom(N) &= \{1,2,3\}\\
dom(b_{max}) &= \{\cancel{0},\mathbf{1}\}\\
dom(x_1) &= \{2,3,\cancel{4}\}\\
dom(x_2) &= \{1,2,3\}\\
dom(x_3) &= \{1,3\}\\
dom(x_4) &= \{\mathbf{3},\cancel{4},\cancel{5}\}\\
dom(x_5) &= \{1,2,\cancel{5}\}
\end{aligned}
$$

(b) Application de la contrainte $\text{MAX}(\mathcal{X}) = N$

$$
\begin{aligned}
dom(N) &= \{\cancel{1},\cancel{2},\mathbf{3}\}\\
dom(b_{max}) &= \{\cancel{0},\mathbf{1}\}\\
dom(x_1) &= \{\mathbf{2},\cancel{3},\cancel{4}\}\\
dom(x_2) &= \{\cancel{1},\cancel{2},\mathbf{3}\}\\
dom(x_3) &= \{\mathbf{1},\cancel{3}\}\\
dom(x_4) &= \{\mathbf{3},\cancel{4},\cancel{5}\}\\
dom(x_5) &= \{1,2,\cancel{5}\}
\end{aligned}
$$

(c) Propagation des contraintes

FIGURE 7.3 – Impact de la réification de la contrainte $\text{MAX}(\mathcal{X}) = N$ par la variable b_{max}.

7.4.2 Description des instances étudiées

Les instances de la littérature pour le SMPTSP ont été introduites par Krishnamoorthy et d'autres [KEB12]. Ils ont ainsi proposé un répertoire de 137 instances générées à partir de considérations réelles, avec entre 70 et 1600 tâches. Nous appellerons ce répertoire d'instances noté ici Data_137. Malheureusement, elles présentent toutes un biais très fort : $N^* = \max_{C \in \mathcal{C}} |C|$. Autrement dit, la notion de qualification des ressources est très faible et n'influence pas la valeur de l'optimum, lequel peut être calculé en temps polynomial. Ceci réduit grandement l'intérêt académique de ces instances. Pour pallier ce biais, nous avons donc proposé notre propre jeux d'instances, plus combinatoires, ayant entre 60 et 950 tâches. Pour les générer, nous sommes partis des résultats empiriques de [KEB12, SWMB14] pour générer des ensembles de tâches menant à des instances difficiles. Cependant, nous avons davantage structuré les compatibilités tâche-ressource en découpant l'horizon de temps et en imposant une fenêtre de temps à la disponibilité de chaque ressource (en plus de compétences aléatoires). Fondamentalement, chaque fenêtre de temps peut être assimilée à une compétence, donc nous n'avons pas changé la définition du problème. En revanche, ce mode de génération abouti sur un partitionnement structuré des ressources, mettant bien en relief les différences entre les ressources qui rendent le problème NP-difficile. La procédure complète de génération des instances est détaillée dans [LFPBM13].

Pour mieux comprendre les différences entre les deux jeux d'instances, nous proposons dans un premier temps d'étudier les ratios $\frac{|\mathcal{S}|}{|\mathcal{T}|}$ et $\frac{N^*}{|\mathcal{S}|}$ de chaque instance. Les valeurs de $|\mathcal{S}|$ et $|\mathcal{T}|$ sont des données d'entrée, alors que N^* n'est connu qu'après résolution. On peut cependant assez vite en avoir une bonne approximation. On étudie donc des caractéristiques liées à la fois aux données d'entrées et à la résolution de ces instances. Comme le montre la figure 7.4, les deux jeux d'instances forment deux groupes bien distincts. La première différentiation se fait avec le ratio $\frac{N^*}{|\mathcal{S}|}$, qui vaut respectivement 39% et 83% en moyenne sur les instances de Data_100 et Data_137. De plus, le ratio $\frac{|\mathcal{S}|}{|\mathcal{T}|}$ est en moyenne deux fois plus grand sur Data_100 que Data_137. En résumé, étant donné un nombre fixé de tâches, les instances de Data_100 ont initialement plus de ressources disponibles que dans Data_137, mais elles en utilisent un plus petit pourcentage dans leurs solutions optimales. Il est donc *a priori* plus difficile de trouver un sous-ensemble de ressources appartenant à une solution optimale.

On peut également remarquer que les instances de Data_137 sont partitionnées en sous-groupes (dont des lignes obliques très nettes), révélant un patron régulier dans la génération de ces instances.

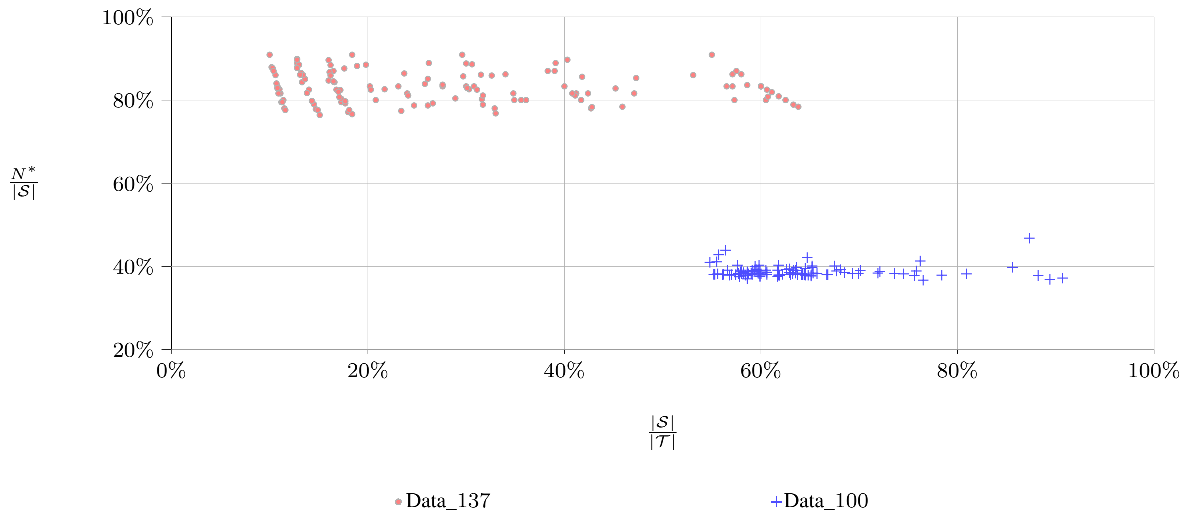


FIGURE 7.4 – Comparaison des caractéristiques des instances de Data_137 et Data_100.

Dans un second temps, nous étudions l'hétérogénéité des ressources. Pour l'évaluer, nous utilisons, sur chaque instance, la distance normalisée de Hamming définie sur les ressources comme suit : étant données deux ressources s_1 et s_2, $d_{\mathrm{H}}(s_1, s_2)$ vaut le nombre de tâches ne pouvant être effectuées que par l'une des deux ressources, divisé par $|\mathcal{T}|$. Nous calculons cette distance pour toute paire de ressources. Nous restituons ces résultats sous la forme de quartiles dans la figure 7.5. On retrouve bien le fait que les différences entre les ressources de Data_137 sont homogènes, ce qui est cohérent avec une génération purement aléatoire (et avec une distribution uniforme) des compétences. Enfin, on remarque que plus le nombre de tâches augmente, plus la distance de Hamming est lissée et converge vers une moyenne de 44%. Ce phénomène, du à la loi des grands nombres, est aussi la marque d'une génération aléatoire non structurée. A l'opposé, les résultats montrent que les instances Data_100 sont relativement structurées, car la variance de la distance de Hamming est bien plus élevée : le premier quartile avoisine les 16% alors que le troisième avoisine les 53%. Ceci est principalement du à la génération des ressources basées sur des fenêtres de temps. Les ressources disponibles durant la même période de temps sont naturellement plus proches les unes des autres, et à l'opposé, les ressources dont la disponibilité est en disjonction sont très différentes.

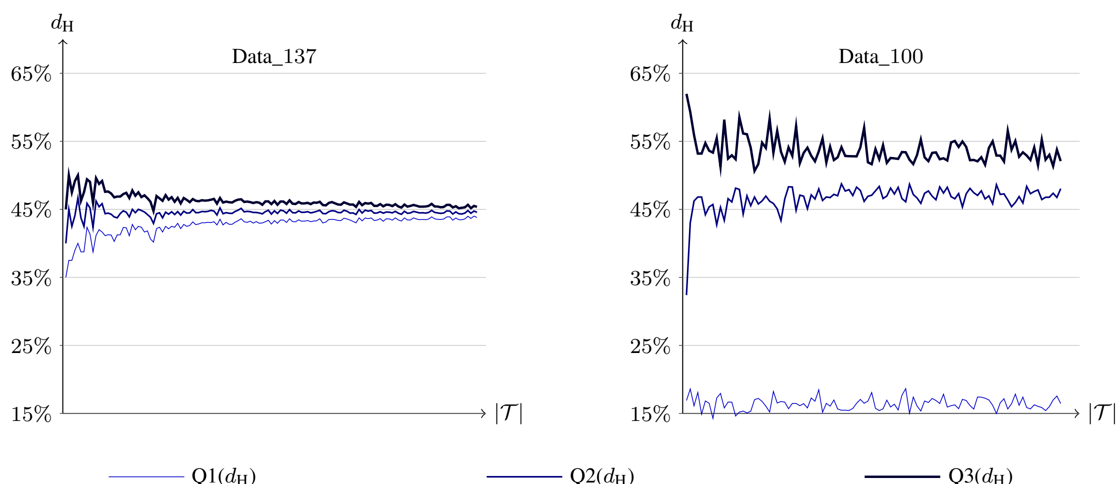


FIGURE 7.5 – Hétérogénéité des ressources (mesurée en quartiles), selon les instances triées par nombre de tâche croissant. d_{H} : distance de Hamming normalisée.

7.4.3 Analyse empirique du modèle

Impact de $G_{\mathcal{CI}}$ sur le noeud racine

L'intérêt de remplacer $G_{\mathcal{I}}$ par $G_{\mathcal{CI}}$ vient de la volonté d'augmenter la borne donnée en calculant un ensemble indépendant maximum. Cela est rendu possible lorsque $G_{\mathcal{CI}}$ est moins dense que $G_{\mathcal{I}}$, *i.e.,* lorsqu'il y a beaucoup de contraintes de différences. C'est donc tout naturellement que nous allons comparer le nombre d'arêtes de $G_{\mathcal{I}}$ et $G_{\mathcal{CI}}$, respectivement notés $E_{\mathcal{I}}$ et $E_{\mathcal{CI}}$. La figure 7.6 représente ainsi le ratio $\frac{E_{\mathcal{CI}}}{E_{\mathcal{I}}}$. Celui-ci varie de 35% à 85% (avec une moyenne de 65%) et de 67% à 75% (avec une moyenne de 71%) sur respectivement Data_137 et Data_100. Cette fois-ci, c'est Data_100 qui présente une structure plus régulière, venant d'une répartition assez homogène des tâches dans le temps. Dans l'ensemble, $G_{\mathcal{CI}}$ a significativement moins d'arêtes que $G_{\mathcal{I}}$.

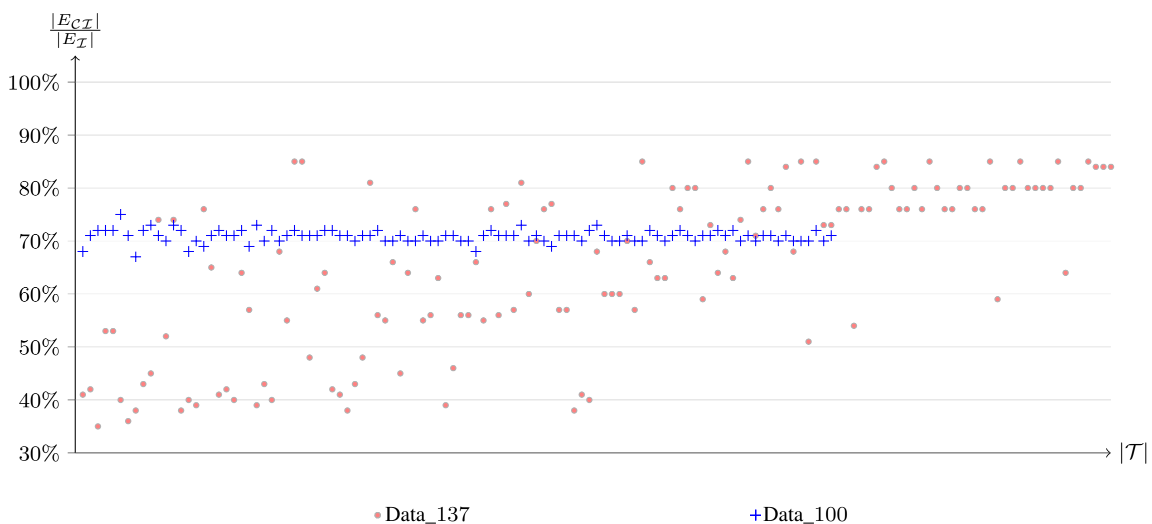


FIGURE 7.6 – Ratio $\frac{|E_{\mathcal{CI}}|}{|E_{\mathcal{I}}|}$ des instances triées par nombre de tâche croissant.

Nous étudions maintenant la borne inférieure calculée au noeud racine. Comme vu en début de chapitre, il existe une borne inférieure triviale à calculer égale au plus grand nombre de tâches en intersection sur l'axe du temps (devant donc être affectées à des ressources différentes). Nous noterons cette borne $\underline{N^{\neq}} = \max_{C \in \mathcal{C}} |C|$. Nous comparons maintenant sur la figure 7.7 la borne calculée par $\texttt{MD}$ sur $G_{\mathcal{I}}$ à celle calculée par $\texttt{MD}$ sur $G_{\mathcal{CI}}$, ce qui correspond au filtrage apporté par respectivement AMNV$\langle G_{\mathcal{I}}|\texttt{MD}|\mathcal{R}_1\rangle$ et AMNV$\langle G_{\mathcal{CI}}|\texttt{MD}|\mathcal{R}_1\rangle$. Nous affichons également $\underline{N^{\neq}}$ comme point de référence pour juger de la qualité d'une borne inférieure. Rappelons que cette borne triviale vaut l'optimum sur toutes les instances de Data_137. Les résultats sont clairs : $G_{\mathcal{CI}}$ améliore drastiquement la borne donnée par un ensemble indépendant. Sur Data_137, $\texttt{MD}(G_{\mathcal{CI}})$ est à environ 6% de $\underline{N^{\neq}}$ (l'optimum), tandis que sur Data_100, elle offre une borne deux fois supérieure à cette dernière. A l'inverse, $\texttt{MD}(G_{\mathcal{I}})$ ne décolle pas et n'apporte donc aucun filtrage. Ces résultats illustrent bien à quel point l'ajout de contraintes annexes (ALLDIFFERENT) peut mettre en défaut la qualité du filtrage d'une contrainte globale (NVALUE). Heureusement, nous montrons ici une simple modification permettant de retrouver une vision globale sur le problème, et ainsi fournir un filtrage puissant.

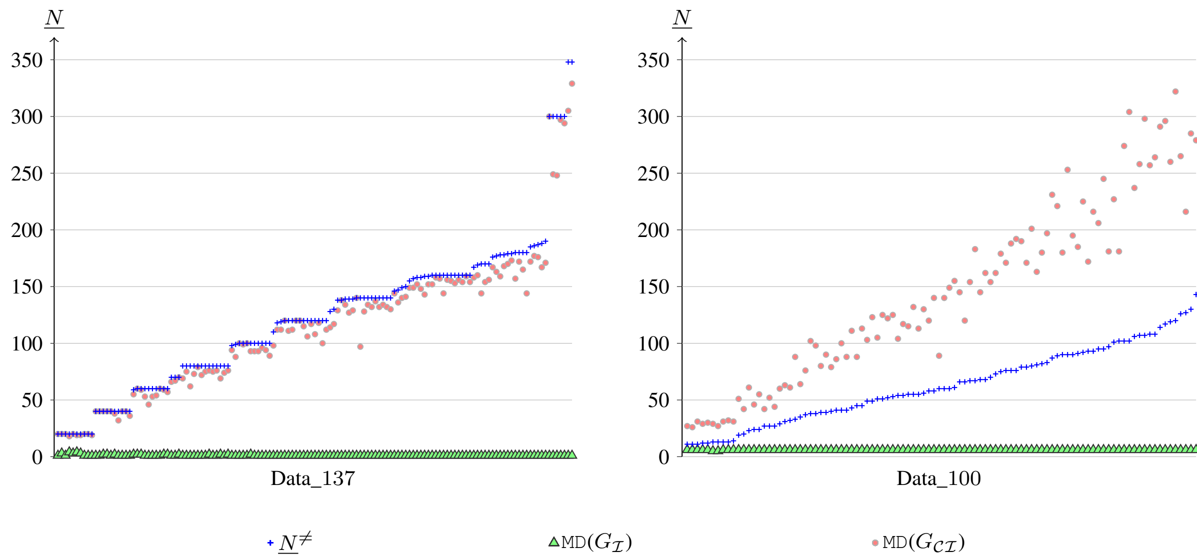


FIGURE 7.7 – Comparaisons des bornes inférieures ($\underline{N}$) calculées par $\underline{N}^{\neq}$, MD($G_{\mathcal{I}}$) et MD($G_{\mathcal{CI}}$) au noeud racine. Les instances sont triées par valeur croissante de $\underline{N}^{\neq}$.

Impact du paramètre k

Pour évaluer la pertinence de la méthode $\mathtt{R}^k$ qui permet de calculer des ensembles indépendants, nous comparons le nombre d'instances résolues à l'optimum par AMNV$\langle G_{\mathcal{CI}}|\mathtt{MD}, \mathtt{R}^k|\mathcal{R}_1\rangle$, selon différentes valeurs de k. Notons que $k = 0$ revient à appliquer uniquement MD. Les résultats sont affichés sur la figure 7.8.

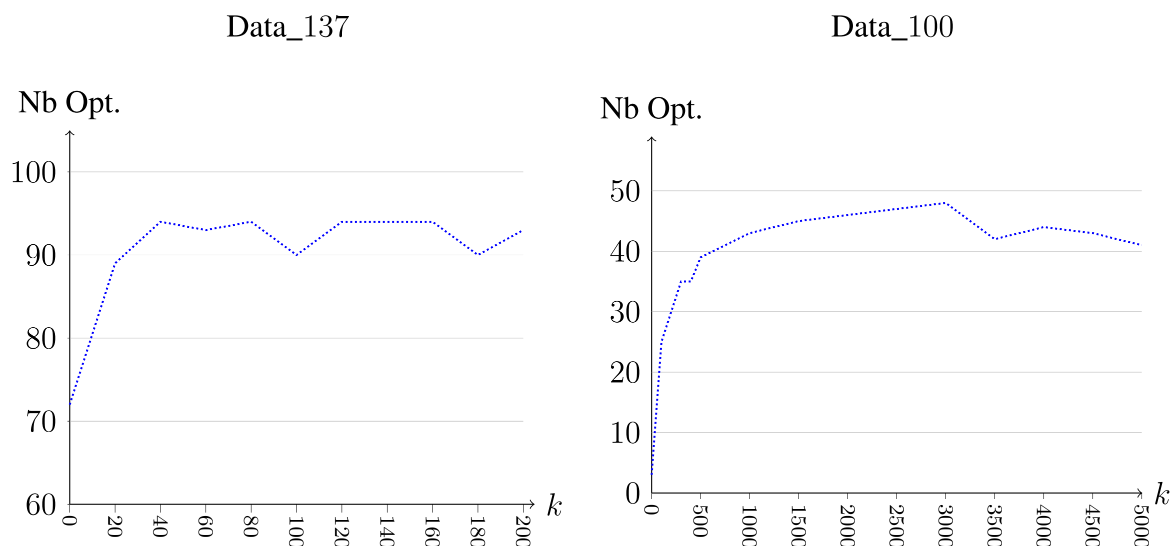


FIGURE 7.8 – Nombre d'instances résolues à l'optimum par ↑AMNV$\langle G_{\mathcal{CI}}|\mathtt{MD}, \mathtt{R}^k|\mathcal{R}_1\rangle$ en moins de 5 minutes, en fonction de k.

$\mathtt{R}^k$ améliore grandement les performances du modèle : $\mathtt{R}^k$ permet de résoudre respectivement une vingtaine et une quarantaine d'instances supplémentaires sur Data_137 et Data_100. Cependant, la meilleure valeur de k varie d'un jeu à l'autre. En effet, alors que quelques dizaines d'itérations donnent de très bons résultats sur Data_137, il vaut mieux en faire des milliers sur les instances de Data_100. Ceci est très intéressant pour deux raisons : d'une part, cela confirme

l'intérêt pratique d'avoir un levier sur le compromis à faire entre temps de calcul et puissance de filtrage, et ainsi mieux s'adapter aux instances traitées ; d'autre part, le fait que cinq mille itérations (qui engendrent nécessairement un lourd surcoût algorithmique) valent mieux qu'une montre bien que notre modèle manque encore de puissance de filtrage et nous incite à travailler sur davantage de filtrage pour résoudre ces instances.

Impact de la rétro-propagation

Pour évaluer l'importance de la rétro-propagation d'AMNV (donnée par $\mathcal{R}_2$ et $\mathcal{R}_3$), nous comparons le nombre d'optima trouvé en 5 minutes par notre modèles avec les propagateurs AMNV$\langle G_{\mathcal{CI}}|\texttt{MD}, \texttt{R}^k|\mathcal{R}_1\rangle$, AMNV$\langle G_{\mathcal{CI}}|\texttt{MD}, \texttt{R}^k|\mathcal{R}_{1,2}\rangle$ et AMNV$\langle G_{\mathcal{CI}}|\texttt{MD}, \texttt{R}^k|\mathcal{R}_{1,3}\rangle$ sur la figure 7.9. Concernant Data_137, on remarque que $\mathcal{R}_1$ est généralement plus performant que $\mathcal{R}_{1,2}$. Cela signifie que le filtrage engendré par $\mathcal{R}_2$ n'est pas suffisant pour amortir son léger surcoût algorithmique. A l'opposé, $\mathcal{R}_3$, qui est plus coûteux à propager que $\mathcal{R}_2$ améliore significativement les résultats, cette règle apporte donc une bonne qualité de filtrage. Il est bon de remarquer que la force de la rétro-propagation croît avec le paramètre k, témoignant de l'importance de diversifier le filtrage. Passé une certaine limite, les performances diminuent pour des raisons algorithmiques. Il faut donc trouver un bon compromis pour avoir un filtrage qui soit le plus rentable possible. Sur Data_100, les résultats restent cohérent, bien que nettement moins marqués. Dans l'ensemble, $\texttt{R}^k$ permet non seulement de mieux filtrer la variable objectif que $\texttt{MD}$ ($\mathcal{R}_1$), mais aussi de mieux filtrer l'ensemble des variables de décision ($\mathcal{R}_3$), grâce à la diversité des ensembles indépendants calculés.

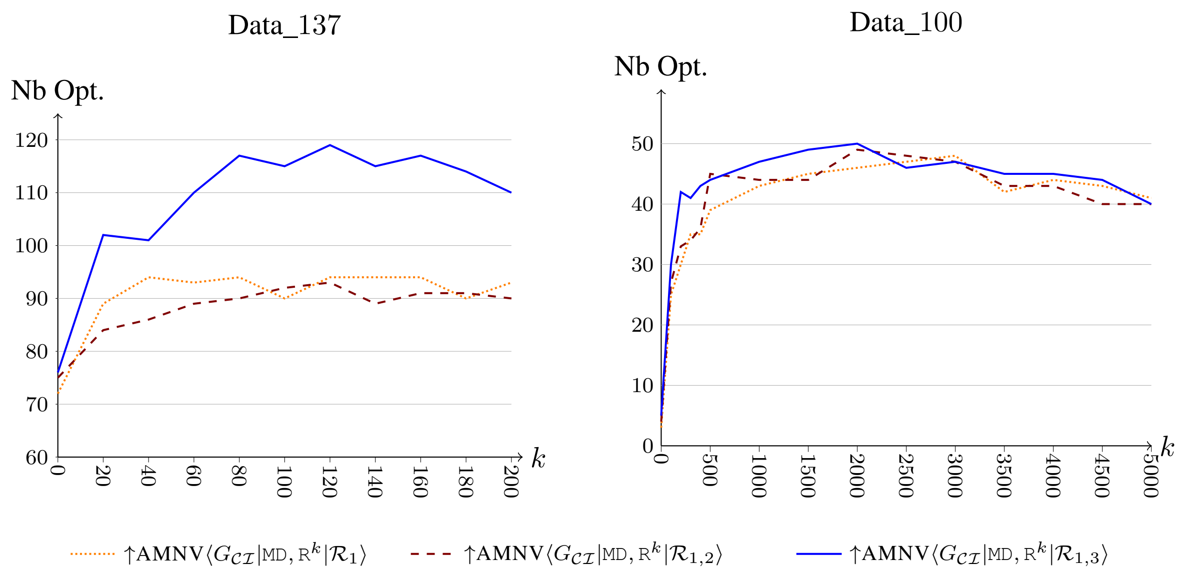


FIGURE 7.9 – Nombre d'instances résolues à l'optimum en moins de 5 minutes, en fonction de k, selon différentes configurations d'AMNV.

Réification de la contrainte MAX$(\mathcal{X}) = N$: un pari gagnant

Nous évaluons maintenant l'impact d'ajouter la contrainte MAX$(\mathcal{X}) = N$, réifiée par la variable b_{max}. Comme le montre la figure 7.10, b_{max} améliore grandement la résolution des instances de Data_137 : 136 des 137 instances sont résolue en moins de 5 minutes, contre un peu moins de 120 sans cette réification. Ceci corrobore une fois de plus le fait que le dimensionnement des ressources et la génération uniforme de leurs compétences rendent assez simple la recherche d'un

ensemble de ressources menant à une solution optimale. Le choix heuristique fait en branchant sur b_{max} (et donc en propageant la contrainte associée) permet de plonger rapidement vers une solution optimale. Malheureusement, cela n'aide pas la résolution des instances de Data_100, pour lesquelles on note une dégradation mineure. Dans l'ensemble, si on met en regard le gain obtenu sur Data_137 avec la légère perte engendrée sur Data_100, le pari est gagnant.

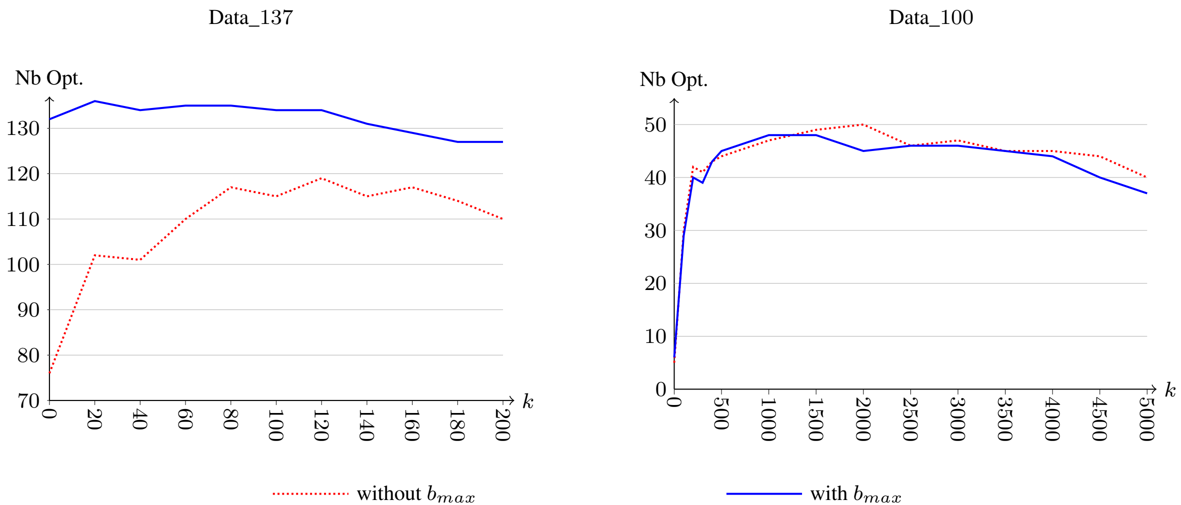


FIGURE 7.10 – Nombre d'instances résolues à l'optimum par $\uparrow$AMNV$\langle G_{\mathcal{CI}}|\texttt{MD}, \texttt{R}^k|\mathcal{R}_{1,3}\rangle$ en moins de 5 minutes, selon différentes valeurs de k, avec et sans l'ajout de la variable b_{max} pour réifier la contrainte $\textsc{Max}(\mathcal{X}) = N$.

7.4.4 Comparaison aux approches existantes

Passage à l'échelle

Nous évaluons maintenant la capacité de notre approche à rapidement fournir de bonnes bornes sur de grandes instances. Nous comparons ainsi l'écart relatif $(\overline{N} - \underline{N})/\overline{N}$ de notre modèle avec celui obtenu par le modèle PLNE de l'état de l'art avec le solveur `CPLEX-12.4`. Concernant notre modèle, nous employons une optimisation par le bas ($\uparrow$) pendant une durée de trois minutes afin d'améliorer la borne inférieure. Si l'optimum n'a pas été trouvé durant ce temps, nous employons alors les trois minutes restantes en optimisation par le haut ($\downarrow$) afin d'être sûr de trouver une solution réalisable et améliorer la borne supérieure. Nous prenons 0 et $|\mathcal{S}|$ comme valeurs par défaut pour respectivement $\underline{N}$ et $\overline{N}$, *i.e.*, $\underline{N}^{\neq}$ n'est pas utilisé ici pour mieux évaluer chaque approche. Par conséquent, une exécution n'étant pas capable de produire une solution réalisable aura une borne supérieure à $|\mathcal{S}|$. Un écart relative égal à 100% témoigne d'une exécution n'ayant produit ni solution réalisable ni borne inférieure qui soit strictement supérieure à 0. Durant la phase d'optimisation par le haut, nous souhaitons explorer rapidement des solutions, nous posons donc $b_{max} = 0$ car la contrainte $\textsc{Max}(\mathcal{X}) = N$ retirerait nombre de solutions réalisables. De plus, puisque cette phase est destinée à l'amélioration de la borne supérieure et non la preuve d'optimalité, il est préférable d'aller vite plutôt que de filtrer beaucoup. Pour cela, nous posons $k = 0$.

Les résultats obtenus sont affichés sur la figure 7.11. Nous pouvons y voir que le modèle PLNE ne parvient pas à traiter environ la moitié des instances sur chaque jeu de données. Sur ces instances, le `CPLEX` n'arrive pas à produire de solution réalisable. Pire encore, sa relaxation linéaire du noeud racine n'arrive pas à produire une borne inférieure qui soit strictement supérieure à 0 dans le temps alloué (écart de 100%). Sur Data_137, il parvient à produire une borne inférieure positive, mais pas de solution réalisable, sur 12 instances (écart de 70% à

95%). On notera en revanche une bonne capacité à trouver des optima sur les petites instances. Nous précisons qu'après utilisation des outils de paramétrage automatique de CPLEX, les résultats ne s'améliorent pas. Comparativement à la PLNE, la programmation par contraintes s'en sort brillamment. Elle permet de trouver une bonne borne inférieure et une bonne solution sur chaque instance (écart relatif moyen de $0,1\%$ et $1,4\%$ pour respectivement Data_137 et Data_100). Sur les instances les plus difficiles, on voit qu'il reste une marge de progression, mais il est clair qu'embarquer une relaxation linéaire dans une contrainte globale n'aidera pas à résoudre les grandes instances.

Dans l'ensemble, ces résultats témoignent bien d'une difficulté de la PLNE à passer à l'échelle pour ce problème. Ce n'est pas nouveau, mais il est intéressant de constater que la programmation par contraintes, souvent critiquée sur des questions de passage à l'échelle, n'a pas la même difficulté du tout. Bien au contraire, la programmation par contraintes s'avère être une technologie à la fois efficace et fiable pour produire de bonnes bornes en des temps acceptables.

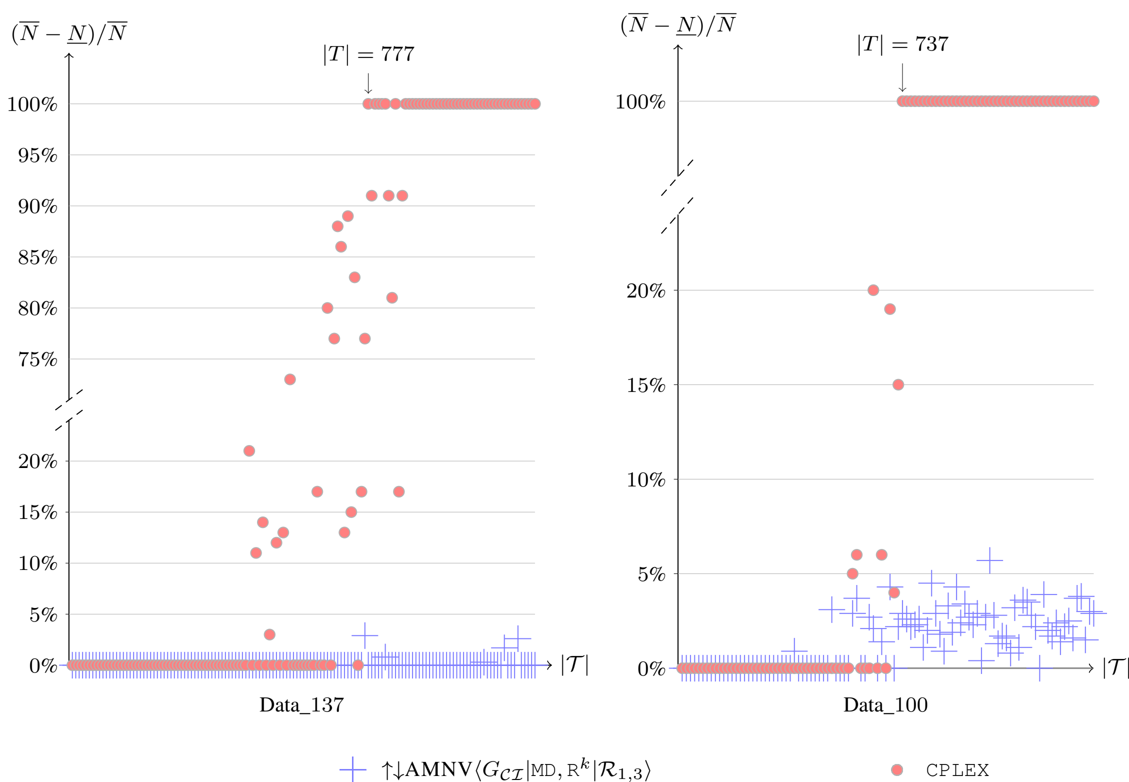


FIGURE 7.11 – Ecart relatif entre $\underline{N}$ et $\overline{N}$, selon le modèle utilisé, sur Data_137 et Data_100, après 6 minutes de résolution. Les instances sont triées par nombre de tâches croissant.

Une approche très compétitive

Nous allons maintenant comparer notre modèle programmation par contraintes aux meilleurs approches RO dédiées à la résolution du SMPTSP. Les deux premières approchent viennent de Krishnamoorthy et d'autres [KEB12] qui ont introduit un modèle PLNE ainsi qu'une méta-heuristique dédiée dotée d'une relaxation Lagrangienne pour aussi fournir une borne inférieure. Suite à ces travaux, Smet et d'autres [SWMB14] ont proposé une math-heuristique (méta-heuristique basée sur la PLNE) permettant de trouver rapidement de bonnes solutions, mais pas de preuve d'optimalité. Enfin, Lin et Ying [LY14] ont proposé une approche en trois phases : un

algorithme glouton produit une solution ; une méta-heuristique l'améliore ; le modèle PLNE de l'état de l'art résout le problème en initialisant sa base avec la meilleure solution ainsi trouvée.

Comme le montre la figure 7.12, notre modèle programmation par contraintes, qui est une approche exacte, est compétitif avec la meilleure méta-heuristique connue [SWMB14]. Ces deux approches dominent largement les autres. Ce résultat est important car, s'il est admit qu'elle soit adaptée aux problèmes fortement contraints, la programmation par contraintes est souvent mise en défaut sur des problèmes d'optimisation combinatoire ayant relativement peu de contraintes (ce qui est le cas du SMPTSP). En effet, la PLNE arrive à trouver de bonnes bornes inférieures à l'aide d'un raisonnement global appliqué au modèle tout entier (sa relaxation linéaire, éventuellement augmentée de coupes). En revanche, la programmation par contraintes traite en général chaque contrainte séparément. Ce manque de vision globale se traduit souvent par une mauvaise capacité à borner l'objectif. Ici, nous sommes parvenu à intégrer les contraintes de différences dans la contrainte NVALUE, offrant ainsi une vision d'ensemble et donc une vraie capacité à borner l'objectif.

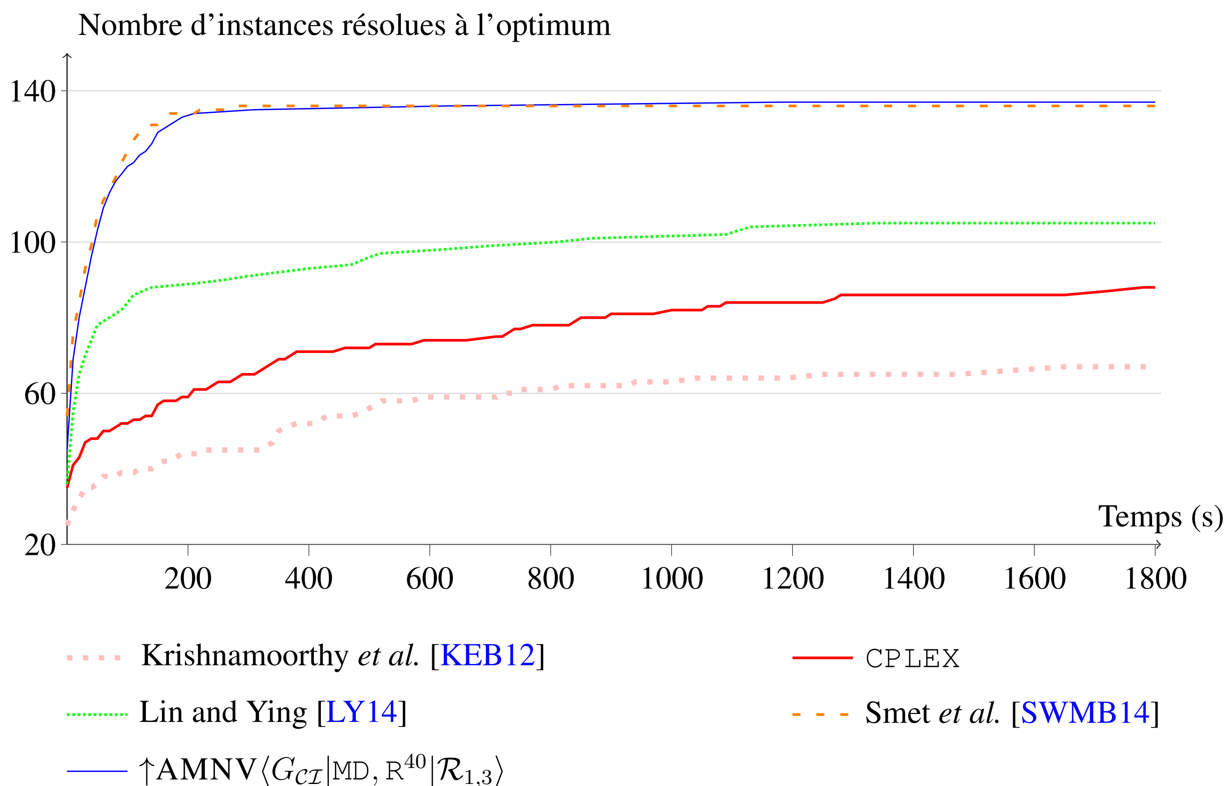
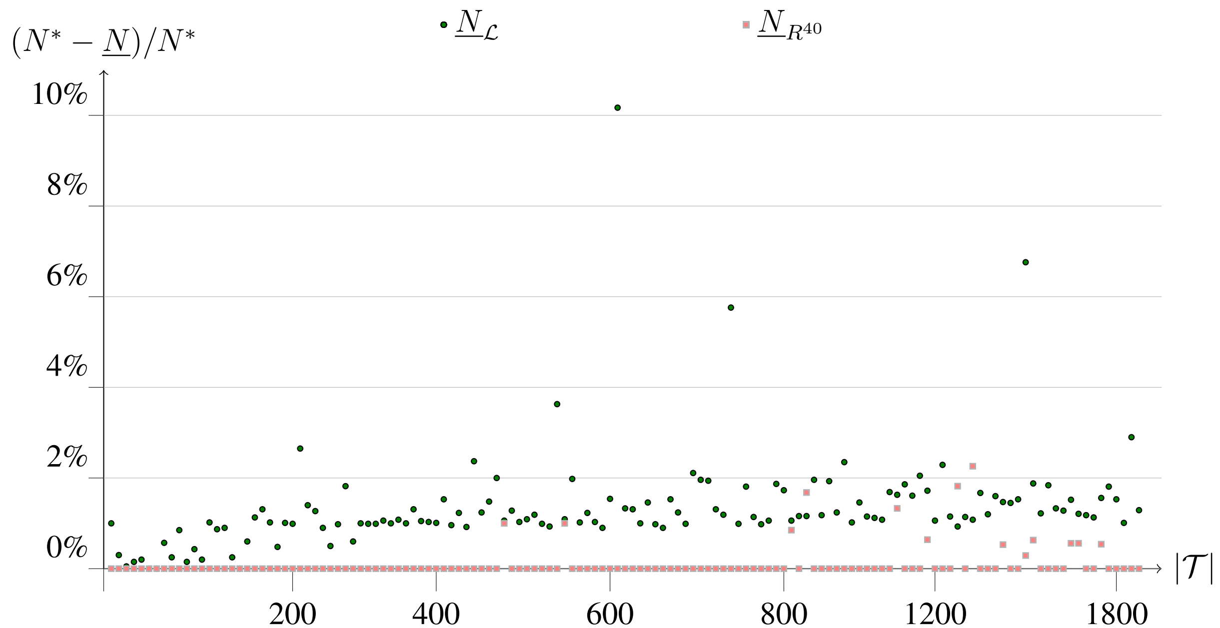


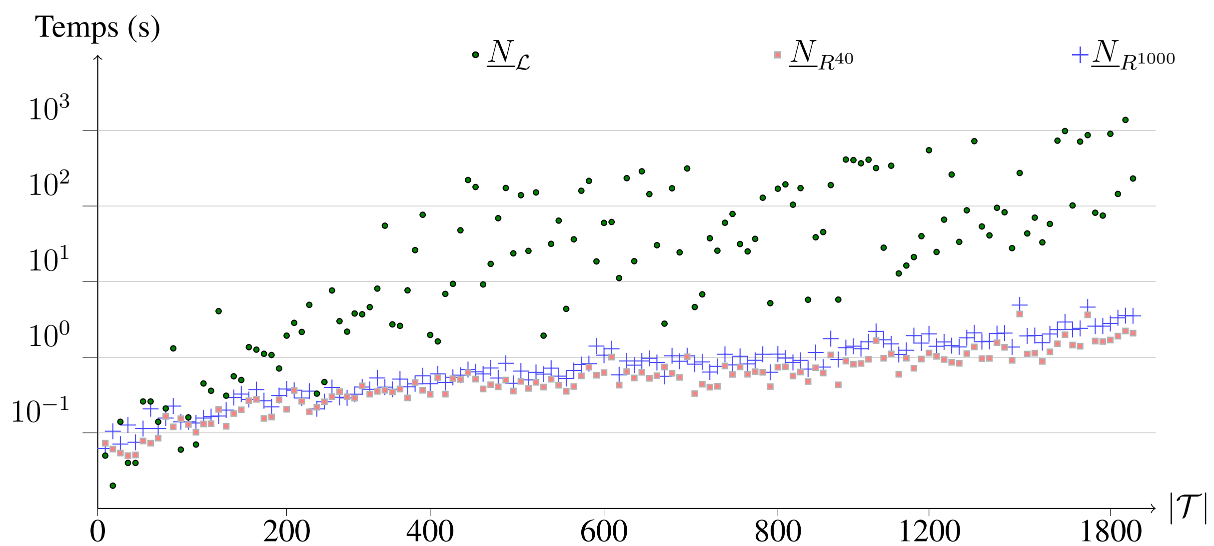
FIGURE 7.12 – Nombre d'instances résolues à l'optimum par chaque modèle en fonction du temps d'exécution, sur Data_137.

Au premier abord, on pourrait s'étonner des mauvaises performances de l'approche dédiée de Krishnamoorthy et d'autres [KEB12], puisqu'elle trouve moins de solutions optimales que le modèle PLNE. En fait, ils ont également constaté l'incapacité du modèle PLNE à passer à l'échelle et c'est pourquoi ils ont proposé cette approche, plus légère, qui est plus à même de calculer des bornes inférieures sur les grandes instances. Nous allons maintenant comparer leur borne, notée $\underline{N}_{\mathcal{L}}$ à la borne inférieure calculée dès le noeud racine du modèle de programmation par contraintes. Pour apprécier les différences de ces méthodes, nous comparons à la fois la valeur de la borne retournée et le temps d'exécution de la méthode (figure 7.13). Concernant le modèle de programmation par contraintes, nous considérons deux valeurs pour k : 40 et 1000, menant respectivement aux bornes $\underline{N}_{R^{40}}$ et $\underline{N}_{R^{1000}}$. Comme le montre la figure 7.13a, $\underline{N}_{R^{40}}$

est nettement meilleure que $\underline{N}_{\mathcal{L}}$. De plus, $\underline{N}_{R^{1000}}$ est encore meilleure puisque cette borne vaut l'optimum sur chacune des instances calculées. Concernant les temps de calculs, la figure 7.13c montre que, là encore, la programmation par contraintes est gagnante. Le calcul de $\underline{N}_{\mathcal{L}}$ est plus de 100 fois plus lent que $\underline{N}_{R^{40}}$ et $\underline{N}_{R^{1000}}$. On remarque aussi que le surcoût engendré par $\underline{N}_{R^{1000}}$ par rapport à $\underline{N}_{R^{40}}$ est raisonnable (un facteur légèrement inférieur à 2) pour un calcul de borne. Pour une résolution entière, où la méthode serait appelée à chaque noeud de l'arbre de recherche, ce surcoût serait en revanche trop cher. Ceci confirme une fois de plus la qualité de la borne calculée par le modèle en contraintes et la pertinence du paramètre k.



(a) Ecart relatif entre $\underline{N}$ et N^*, selon le modèle employé. $\underline{N}_{R^{1000}}$ n'est pas affiché car il vaut toujours N^*.



(c) Temps (en secondes) pour calculer $\underline{N}$, selon le modèle utilisé

FIGURE 7.13 – Comparaison de la borne inférieure calculée par le modèle programmation par contraintes au noeud racine avec $k = 40$ ($\underline{N}_{R^{40}}$) et $k = 1000$ ($\underline{N}_{R^{1000}}$) à la relaxation de [KEB12] ($\underline{N}_{\mathcal{L}}$), sur les instances de Data_137 triées par nombre croissant de tâches.

7.5 Conclusion du chapitre

Dans ce chapitre nous avons vu deux manières de filtrer la contrainte NVALUE : une approche reformulée explicitement sous forme de graphe ; une approche directe intégrant un algorithme de graphe dans une contrainte globale. Fondamentalement, ces deux méthodes sont très proches. Dans les deux cas, nous avons vu comment intégrer simplement des contraintes de différence pour améliorer le filtrage. Ces améliorations sont faciles à mettre en oeuvre et n'engendrent aucune difficulté d'implémentation ou de maintenance. Malgré leur simplicité nous avons montré qu'elles pouvaient apporter des gains considérables. De plus, nos études expérimentales nous ont amené à proposer un schéma de branchement innovant basé sur la réification de contraintes. Répandre l'utilisation de ce type de stratégies de recherche nous semble très prometteur.

Nous avons mené une longue étude expérimentale, permettant d'apprécier l'apport de chaque règle de filtrage et de l'heuristique de branchement. Nous avons comparé notre approche à l'état de l'art, que ce soit en méthodes exactes ou approchées. Il en ressort que notre modèle est actuellement la meilleure approche exacte permettant de traiter des instances impliquant 800 à 2000 activités. Plus précisément, alors que les solveurs de programmation linéaire ne franchissent pas la barre des 800 activités, notre modèle est capable de fournir à la fois une solution réalisable et une borne inférieure de bonne qualité en un temps très court. De plus, concernant les temps d'exécution, notre méthode offre des performances équivalentes à la meilleure méta-heuristique de la littérature [SWMB14]. Dans l'ensemble, nous avons bâti un modèle capable de résoudre efficacement de grandes instances du SMPTSP, plaçant la programmation par contraintes au dessus des approches existantes.

7.5.1 Perspectives

Réification d'un réseau de contraintes par une variable graphe

Nous avons vu comment utiliser une variable graphe et la contrainte NCLIQUE pour réifier un réseau de contraintes d'égalités sur les entiers et assurer une propriété globale relative au nombre de classes d'équivalences ainsi formées. Notre travail peut être repris pour partitionner (on parle de *clustering*) des points dans l'espace, où des contraintes de différences sont posées pour contraindre les noeuds jugés trop distants à appartenir à des clusters différents [DDV13a, DDV13b]. Cette capacité à prendre en compte des contraintes annexes connait un intérêt grandissant en fouille de données, car elle permet de renforcer les méthodes d'apprentissage avec des connaissances expertes [DR09, WQD14]. On notera également que, si le nombre de clusters à rechercher est généralement un paramètre fixé dans les méthodes usuelles, la programmation par contraintes permet de le représenter par une variable, offrant une plus grande richesse de modélisation. Il serait donc très intéressant d'utiliser la programmation par contraintes pour résoudre des applications classiques de fouilles de données.

Nous pouvons naturellement généraliser ce raisonnement à des variables ensemblistes aussi bien discrètes que continues. On peut alors directement traiter les problèmes hybrides pour lesquels les variables à partitionner sont des ensembles continus, mais la propriété globale sur le nombre de classes d'équivalences est discrète. C'est notamment le cas de la contrainte globale NVECTOR [CJL09], qui contrôle le nombre de valeurs prises par un ensemble de boites (variables continues multidimensionnelles). Cette généralisation au cas continus et multidimensionnel conserve le fait que, si deux noeuds sont adjacents dans une solution, alors leurs variables associées sont égales. D'ailleurs, parallèlement à nos travaux, Jaulin a redécouvert le concept de variable de graphe pour résoudre ce problème hybride dans une application fondamentale en robotique [Jau14].

Enfin, on peut chercher à réifier d'autres contraintes binaires. Par exemple, il est possible de

passer d'un ensemble de contraintes d'égalités (e.g., $x_i = x_j$) à un ensemble de contraintes de distances (e.g., $|x_i - x_j| \leq \epsilon$, avec $\epsilon \in \mathbb{R}^+$). De même, on peut réifier un ensemble d'inégalités strictes, dont le réseau forme un graphe acyclique.

Utilisation de l'aléatoire dans les algorithmes de filtrage

Nous avons également vu une manière simple de faire appel à l'aléatoire pour améliorer un algorithme de filtrage et tendre vers un plus haut niveau de cohérence. On peut chercher à étendre ce type d'usages vers d'autres contraintes. Par exemple, il est possible d'atteindre facilement un compromis entre BC et GAC pour la contrainte ALLDIFFERENT en appliquant plusieurs fois l'algorithme de BC et en renumérotant aléatoirement les valeurs entre chaque appel. On obtient ainsi des intervalles différents, donc potentiellement plus de filtrage. De manière plus générale, l'aléatoire ayant démontré sa force pour l'exploration de l'espace de recherche, il serait intéressant d'investiguer davantage son intérêt pour le filtrage.

Des heuristiques audacieuses, basées sur la réification de contraintes

Notre approche heuristique basée sur la réification de la contrainte MAX$(\mathcal{X}) = N$ (section 7.4.1) est assez simple. En effet, seule une contrainte est réifiée, donc sa variable associée peut être créée dès la pose du modèle. Cependant, on pourrait tout à fait la généraliser à un nouveau concept d'heuristique qui, à chaque décision, introduit une nouvelle contrainte, la réifie avec une nouvelle variable binaire et branche sur cette dernière pour propager la contrainte ainsi créée et réduire un ensemble de domaines d'un coup. On aboutirait alors sur de nouvelles générations d'heuristiques, aussi bien dédiées que génériques. Concevoir une telle stratégie qui fonctionne comme une boite-noire et qui soit compétitive avec les heuristiques génériques actuelles [BHLS04, MH12, Ref04] sur de nombreux problèmes semble très ambitieux. Notons tout de même que certains travaux en détection de symétries [FPS+09], en apprentissage de contraintes [BS11] et en exploitation de la réification pour la recherche [MFMS14] ont sûrement déjà apporté quelques pierres fondatrices à ce beau défi d'intelligence artificielle. En outre, de très récents travaux portant sur les cohérences fortes pour les contraintes de tables [MDL14] forment une première porte d'entrée vers de telles recherches. Pour chaque contrainte de table, ils introduisent une variable où chaque valeur représente un tuple : en branchant sur ces variables, on obtient naturellement une heuristique audacieuse. Enfin, si l'on ne cherche pas à développer de méthode générique, alors, sur certaines classes d'instances, il est déjà possible de réaliser des stratégies audacieuses dédiées qui soient très efficaces. En fait, cela a déjà été fait, d'une certaine manière, avec l'introduction d'algorithmes gloutons dans des contraintes globales [ABC+09, LCB13].

En travaillant sur des contraintes de placement géométrique et d'ordonnancement de tâches, Ågren *et al.* [ABC+09] et Letort *et al.* [LCB13] ont introduit des fonctionnements gloutons dans certaines contraintes globales. Le fonctionnement d'une contrainte en mode glouton est le suivant : l'algorithme glouton va chercher un support validant la contrainte. S'il y parvient alors le propagateur va fixer toutes ses variables selon le support qu'il a trouvé. Sinon, il applique normalement son filtrage et tentera à nouveau sa chance lors de la prochaine propagation. Cette technique offre des gains drastiques en termes de passage à l'échelle sur les problèmes simples. Cependant, l'algorithme glouton ne peut pas être appliqué comme n'importe quel algorithme de filtrage, durant la phase de propagation des contraintes, car on perdrait alors le caractère compositionnel et la correction du modèle. Par exemple, il est possible que la contrainte gloutonne trouve un support valide dès le noeud racine et fixe ainsi toutes ses variables, mais rien ne garantit que ce tuple satisfasse toutes les autres contraintes du modèle. Si une contrainte

est violée, alors le solveur répondra que le problème n'admet pas de solution, ce qui est potentiellement faux. Au fait de cet inconvénient, les auteurs suggèrent d'appliquer l'algorithme glouton dans une procédure dédiée qui succède la phase de propagation des contraintes : une fois les contraintes propagées (sans algorithme glouton), la contrainte gloutonne crée un nouveau noeud dans l'arbre de recherche, applique son algorithme glouton et propage l'ensemble des contraintes du modèle. Si une solution est trouvée, alors le programme s'arrête. Sinon, les variables étant fixées, un échec est créé pour déclencher un bracktrack. On se retrouve alors dans l'état précédant l'application de l'algorithme glouton et le processus de recherche peut continuer. Cette approche préserve donc la correction du modèle.

Nous émettons néanmoins quelques critiques à cette approche : premièrement, modifier l'arbre de recherche devrait être du ressort des heuristiques de recherche et non des contraintes. D'ailleurs, il peut arriver que plusieurs contraintes proposent des algorithmes gloutons, auquel cas il serait bon de développer différentes stratégies de sélection, rappelant encore une fois la notion d'heuristique de recherche. Deuxièmement, en cas d'échec de l'algorithme glouton, aucune information n'est gagnée. Enfin, la mise en place d'un tel schéma dépend beaucoup de l'architecture même du solveur utilisé et peut s'avérer complexe, voire intrusive. Néanmoins, ceci relève peut-être du détail et ce schéma reste proche de la notion d'heuristique audacieuse. L'approche par heuristique audacieuse, plus générique, consiste à décaler l'application de l'algorithme glouton dans l'heuristique. Considérons un appel à une heuristique de branchement, plutôt que de simplement choisir une variable de décision et une valeur à lui affecter, notre heuristique va appliquer un algorithme glouton sur une sous-partie de notre modèle pour ainsi trouver un tuple *probablement valide* (à interpréter au sens large). L'algorithme glouton peut être soit donné par l'utilisateur, soit donné par les contraintes du modèle. Si plusieurs résultats (tuples) sont fournis, alors on pourra établir des critères (e.g., taille, coût) pour les ordonner. Une fois le tuple sélectionné, l'heuristique va créer une contrainte de table avec cet unique tuple, créer une variable binaire réifiant cette contrainte et enfin brancher sur cette variable en la fixant à 1. La propagation de contraintes va répercuter l'instanciation à ce tuple sur le reste du modèle. Si tout va bien, une solution sera très vite trouvée. Sinon, si la propagation mène à un échec, un backtrack permettra de réfuter le tuple (on a ainsi un gain en information, bien que relativement faible) et relancer l'heuristique pour calculer un autre glouton, respectant si possible cette fois les autres contraintes du modèle.

Enfin, nous précisons que l'adjectif "audacieuse" est volontairement choisit pour véhiculer le principe d'effectuer des paris osés, c'est-à-dire ayant une perspective de gain élevée mais une faible probabilité de succès, durant la phase de branchement. On pourra ensuite parler de branchement basé sur la génération de tuples, l'ajout de contraintes, des algorithmes *ad hoc*, *etc.* selon l'approche mise en place techniquement.

Mis-à-part le travail portant sur les heuristiques, que l'on peut réutiliser sur des problèmes très différents, nous avons jusqu'à présent essentiellement exploité des notions de graphe pour améliorer le filtrage de contraintes globales particulières. En ce sens, ce travail était relativement dédié. Nous allons maintenant étudier comment utiliser des structures de graphes dans le but d'accélérer la propagation d'une grande famille de contraintes globales : les contraintes auto-décomposables.

8

Des graphes génériques pour l'auto-décomposition de contraintes globales

Sommaire

8.1 Fondements théoriques . . . 102

8.1.1 f-monotonie de contrainte . . . 103

8.1.2 f-décomposition de contrainte . . . 103

8.1.3 f-préservation de cohérence . . . 104

8.1.4 Quatre exemples génériques d'auto-décomposition . . . 104

8.2 Implémentation générique d'une $f^{\cap}_{vn}$-décomposition . . . 109

8.2.1 Grandes lignes . . . 109

8.2.2 Aspects avancés . . . 110

8.3 Etude de la contrainte CUMULATIVE . . . 111

8.3.1 $f^{\cap}_{vn}$-décomposition de la contrainte . . . 111

8.3.2 Autres auto-décompositions de la contrainte . . . 113

8.3.3 Intérêt pratique . . . 113

8.4 Conclusion du chapitre . . . 116

Dans ce chapitre, nous allons étudier une façon originale d'utiliser des graphes pour améliorer la propagation de certaines contraintes globales. Nous nous intéresserons particulièrement à la problématique de propagation de contraintes impliquant un très grand nombre de variables. En effet, à l'époque du *Big Data*, de plus en plus d'applications de programmation par contraintes impliquent de traiter un grand volume de variables. Pour accélérer la propagation de telles contraintes, de nombreux travaux ont été menés pour réduire au maximum la complexité temporelle des algorithmes de filtrage qui, en général, est fonction du nombre de variables n. Néanmoins, comme l'a dit Toby Walsh à la conférence internationale CP 2012, traiter les problèmes de grandes tailles est un réel défi pour la programmation par contraintes et

diminuer la complexité des algorithmes de filtrage ne suffira pas à y répondre. En effet, au bout d'une certaine valeur de n, même une complexité de $O(n \log n)$ peut être trop lourde, car les contraintes sont propagées à chaque noeud de l'arbre de recherche. Notre contribution fait suite à ce constat : si même une complexité aussi faible que $O(n \log n)$ devient trop chère payée, alors il faut envisager de diminuer ce n d'une manière ou d'une autre. Pour cela, nous suggérons de ne pas considérer l'ensemble des variables lors du filtrage, quitte à relâcher la cohérence de ce dernier. Ce travail a été publié dans [FLP14].

L'idée de filtrer uniquement sur des sous-ensembles de variables n'est pas complètement nouvelle en soi. Par exemple, l'une des améliorations majeures de la contrainte SOMEDIFFERENT est basée sur cette idée [AER+10]. Néanmoins, les travaux antérieurs portant sur ce sujet ont été réalisés de manière dédiée. Nous proposons une étude portant sur une famille générique de contraintes dont la satisfaction est assurée en appliquant leur propre algorithme de filtrage, sans modification de ce dernier, à des sous-ensembles des variables de la contrainte. Nous les appelons les contraintes auto-décomposables. Concrètement, nous formalisons deux types de décompositions existantes, la $f^{\cap}_{cc}$-décomposition et la $f^{\mathcal{D}}$-décomposition, et nous les généralisons dans le but d'obtenir de nouvelles possibilités en terme de compromis entre filtrage et temps de calcul, débouchant alors sur la $f^{\cap}_{vn}$-décomposition et la famille des $f^{\mathcal{D}}_{\mathcal{P}_*}$-décomposition. Notre paradigme est indépendant de l'algorithme de filtrage concerné et ne requiert aucune modification de ce dernier. Il peut donc être embarqué facilement dans un solveur de contraintes pour en améliorer les performances sur des problèmes de grande taille.

Les concepts théoriques liés à l'auto-décomposition de contraintes globales sont introduits en section 8.1. Une implémentation générique est proposée en section 8.2. Une illustration sur la contrainte CUMULATIVE, appuyée par des évaluations empiriques, est fournie en section 8.3. Enfin, une conclusion et des perspectives de recherche sont données en section 8.4.

8.1 Fondements théoriques

Cette section est organisée comme suit. Tout d'abord, les conditions nécessaires à ce que le filtrage sur des sous-ensembles de variables ne supprime pas de solutions sont introduites. Ensuite, nous étudions le cas où ce filtrage est suffisant pour satisfaire la contrainte sur son ensemble de définition. Puis, nous étudions le cas où la décomposition offre des garanties de cohérence. Enfin, nous illustrons notre propos à l'aide de quatre exemples génériques d'auto-décomposition de contraintes globales.

Dans un soucis de simplicité, et sans perte de généralité, nous considérons dans cette section qu'une contrainte a un seul algorithme de filtrage, embarqué dans un unique propagateur. Nous utilisons la notation $c(\mathcal{X})$ pour désigner une contrainte c définie sur un ensemble de variables $\mathcal{X}$. Nous utilisons également la notation $\mathcal{P}(\mathcal{X})$ pour représenter toutes les parties de $\mathcal{X}$.

Puisque nous cherchons à appliquer l'algorithme de filtrage d'une contrainte sur seulement quelques parties de ses variables, il est primordial de vérifier qu'un tel filtrage ne va pas retirer de solutions à la contrainte. Pour cela, nous reformulons légèrement la définition de la monotonie d'un contrainte proposée par Barták [Bar03] et exploitée par Maher [Mah09]. Il est intéressant de noter que ce concept a été introduit pour étudier les contraintes dont l'ensemble de définition peut grossir durant la recherche (les *open constraints*). Nous faisons ici l'inverse, d'où la reformulation proposée (définition 6).

Définition 6 *Une contrainte $c(\mathcal{X})$ est **monotone** si et seulement si, pour tout sous-ensemble de variables $\mathcal{Y} \in \mathcal{P}(\mathcal{X})$, si $c(\mathcal{X})$ est satisfiable, alors sa projection sur $\mathcal{Y}$ est également satisfiable.*

En d'autres termes, étant donnés deux ensembles de variables $\mathcal{X}$ et $\mathcal{Y}$, tels que $\mathcal{Y} \subseteq \mathcal{X}$. Si la contrainte c est monotone, alors la contrainte $c(\mathcal{X})$ implique la contrainte $c(\mathcal{Y})$.

Nous précisons ici que cette définition n'a rien à voir avec la définition de monotonie d'une contrainte proposée dans [VHDT92] ou bien la monotonie d'un propagateur [ST09]. En revanche, elle est très proche de la notion de *contractibilité* de [Mah09]. De plus, cette définition pourrait bien être liée à celle d'un circuit booléen monotone, qui a déjà été utilisée pour étudier des décompositions de contraintes globales [BKNW09].

8.1.1 f-monotonie de contrainte

Puisque les contraintes monotones s'appliquent de manière implicite à chacun de leurs sous-ensembles de variables, on peut appliquer leur algorithme de filtrage sur toutes les parties de leurs variables, sans que ce filtrage n'engendre de perte de solution (proposition 5).

Proposition 5 *Si $c(\mathcal{X})$ est monotone, alors appliquer son algorithme de filtrage à chaque partie de $\mathcal{X}$ ne supprime aucune solution.*

Preuve : Supposons qu'une solution s de $c(\mathcal{X})$ est perdue suite à l'application de l'algorithme de filtrage de c sur le sous-ensemble de variables $\mathcal{Y} \subseteq \mathcal{X}$. Considérons maintenant que nous réduisons le domaine de chaque variable $x_i \in \mathcal{X}$ à sa valeur dans s. Par reconstruction, $c(\mathcal{X})$ est satisfaite alors que $c(\mathcal{Y})$ est violée. Ceci est en contradiction avec la définition 6. □

Cependant, la définition de la monotonie est vraiment très restrictive, si bien que très peu de contraintes sont monotones (comme déjà mentionné par Barták [Bar03]). Cependant, en le relâchant un peu, se concept s'applique à de nombreuses contraintes. Pour cela, nous introduisons une multi-fonction f qui énumère seulement *certaines* parties de son ensemble de définition, *i.e.,* $f(\mathcal{X}) \subseteq \mathcal{P}(\mathcal{X})$. Nous généralisons ainsi la monotonie vers une f-monotonie (définition 7).

Définition 7 *Une contrainte $c(\mathcal{X})$ est **f-monotone** si et seulement si elle s'applique sur chacune des parties de $\mathcal{X}$ induites par f, i.e., $\forall \mathcal{Y} \in f(\mathcal{X})$, $c(\mathcal{X})$ implique $c(\mathcal{Y})$.*

Proposition 6 *Si $c(\mathcal{X})$ est f-monotone, alors appliquer son algorithme de filtrage à n'importe quelle partie $\mathcal{Y} \in f(\mathcal{X})$ ne supprime aucune solution.*

Preuve : Analogue à la proposition 5. □

Par exemple, si l'on note $f_\forall$ la multi-fonction qui énumère toutes les parties, alors monotonie et $f_\forall$-monotonie sont équivalentes. De plus, si l'on note f_I la multi-fonction identité, *i.e.,* $\forall \mathcal{X}$, $f(\mathcal{X}) = \{\mathcal{X}\}$, alors n'importe quelle contrainte est f_I-monotone.

8.1.2 f-décomposition de contrainte

Nous avons caractérisé les contraintes pouvant être propagées sur des sous-ensembles de variables sans perte de solution. Cependant, le filtrage ainsi obtenu pourrait être trop faible, au point de laisser passer des non-solutions. Nous allons donc maintenant étudier le cas où filtrer une contrainte sur $f(\mathcal{X})$ est suffisant pour assurer sa satisfaction et donc ne produire aucune non-solution. Nous introduisons alors le concept de f-décomposition (définition 8).

Définition 8 *Une contrainte $c(\mathcal{X})$ est **f-décomposable** si et seulement si l'ensemble des solutions de $c(\mathcal{X})$ est égal à l'ensemble des solutions de $\bigwedge_{\mathcal{Y} \in f(\mathcal{X})} c(\mathcal{Y})$.*

Remarquons que si une contrainte est f-décomposable, alors elle est également f-monotone. Cependant, la réciproque n'est pas vraie.

Plus généralement, on parlera de contraintes auto-décomposables (définition 9). Ces contraintes peuvent être propagées en appliquant à chacune leur propre algorithme de filtrage, sans aucune modification de ce dernier, sur des parties de leur ensemble de variables.

Définition 9 *Une contrainte est **auto-décomposable** si et seulement si il existe une multi-fonction f non triviale i.e., $f \neq f_I$, pour laquelle la contrainte est f-décomposable.*

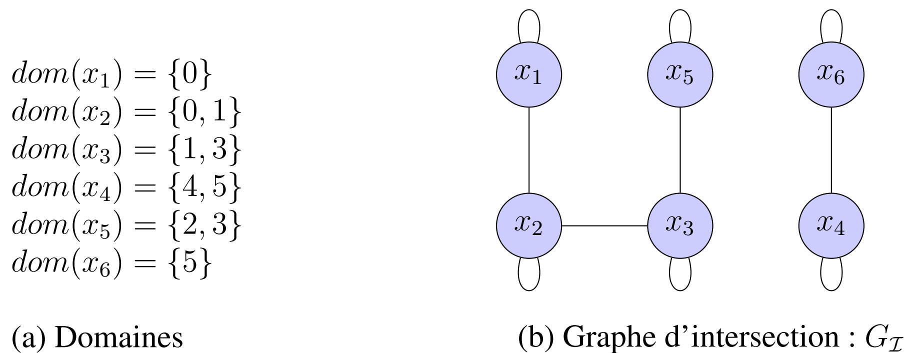
8.1.3 f-préservation de cohérence

Nous savons comment dynamiquement décomposer le filtrage d'une contrainte globale sur différentes parties de ses variables. Etudions maintenant le cas où cette décomposition assure un niveau de cohérence donné. Pour cela, nous introduisons la définition de λ-f-préservation (définition 10) signifiant qu'un niveau de cohérence λ est assuré en ne filtrant que sur les parties de $\mathcal{X}$ induites par f.

Définition 10 *Etant donné un niveau de cohérence λ, une contrainte f-décomposable $c(\mathcal{X})$ est **f-λ-préservative** si et seulement si établir la λ cohérence sur chacune des parties de $f(\mathcal{X})$ assure la λ cohérence sur l'ensemble $\mathcal{X}$.*

8.1.4 Quatre exemples génériques d'auto-décomposition

Nous allons maintenant introduire quelques multi-fonctions d'énumérations remarquables. Pour cela, nous utiliserons à nouveau le concept de graphe d'intersection. Un petit exemple illustratif, impliquant un ensemble de variables $\mathcal{X}$, est donné en figure 8.1. Le graphe d'intersection de $\mathcal{X}$ est noté $G_{\mathcal{I}}$. Nous supposerons ici qu'une contrainte ALLDIFFERENT est appliquée sur $\mathcal{X}$.



(a) Domaines (b) Graphe d'intersection : $G_{\mathcal{I}}$

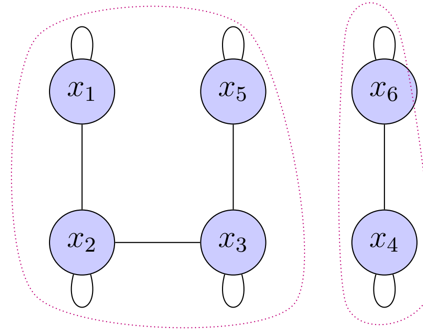
FIGURE 8.1 – Exemple portant sur un ensemble de variables $\mathcal{X} = \{x_1, x_2, x_3, x_4, x_5, x_6\}$.

Décomposition en composantes connexes

Notons $f_{cc}^{\cap}$ la multi-fonction qui énumère les parties de $\mathcal{X}$ associées à chaque composante connexe de $G_{\mathcal{I}}$ (définition 11). Il est connu que la contrainte ALLDIFFERENT est $f_{cc}^{\cap}$-décomposable, car deux composantes connexes forment deux problèmes indépendants de couplage. Sa généralisation, SOMEDIFFERENT l'est également. D'ailleurs, une sorte de $f_{cc}^{\cap}$-décomposition est décrite comme étant une amélioration significative de l'algorithme originel [AER+10]. Sur notre exemple, $f_{cc}^{\cap}(\mathcal{X}) = \{\{x_1, x_2, x_3, x_5\}, \{x_4, x_6\}\}$ (figure 8.2). La $f_{cc}^{\cap}$-décomposition de ALLDIFFERENT($\mathcal{X}$) signifie donc de filtrer les contraintes implicites ALLDIFFERENT($\{x_1, x_2, x_3, x_5\}$)

et ALLDIFFERENT($\{x_4, x_6\}\}$). On remarquera qu'une $f_{cc}^{\cap}$-décomposition génère 1 à $|\mathcal{X}|$ parties de $\mathcal{X}$.

Définition 11 $f_{cc}^{\cap} = \{\{x_i | i \in C\} | C \in \mathit{ConnectedComponentsOf}\,(G_{\mathcal{I}})\}$



$$f_{cc}^{\cap}(\mathcal{X}) = \{\{x_1, x_2, x_3, x_5\}, \{x_4, x_6\}\}$$

FIGURE 8.2 – Illustration d'une $f_{cc}^{\cap}$-décomposition.

La $f_{cc}^{\cap}$-décomposition est remarquable en ce qu'elle préserve la qualité du filtrage. Plus précisément, appliquer un algorithme de GAC de manière $f_{cc}^{\cap}$-décomposée établit la GAC sur l'ensemble des variables (théorème 3). De la même manière, on peut montrer que la $f_{cc}^{\cap}$-décomposition préserve la BC.

Théorème 3 *Toute contrainte $f_{cc}^{\cap}$-décomposable est $f_{cc}^{\cap}$-GAC-préservative.*

Preuve : Considérons une contrainte c qui soit $f_{cc}^{\cap}$-décomposable mais pas $f_{cc}^{\cap}$-GAC-préservative. Il existe donc une partie $\mathcal{Y} \in f_{cc}^{\cap}(\mathcal{X})$ pour laquelle il existe une valeur v dans le domaine d'une variable $x \in \mathcal{Y}$ qui n'est pas filtrée par $c(\mathcal{Y})$ mais qui est filtrée par $c(\mathcal{X})$. Puisque le niveau de cohérence de l'algorithme considéré est la GAC, il existe une solution de $c(\mathcal{Y})$ pour laquelle $x = v$. Or, la $f_{cc}^{\cap}$-décomposition implique de préserver l'ensemble des solutions et les composantes connexes partitionnent $\mathcal{X}$, donc cela n'est pas possible. □

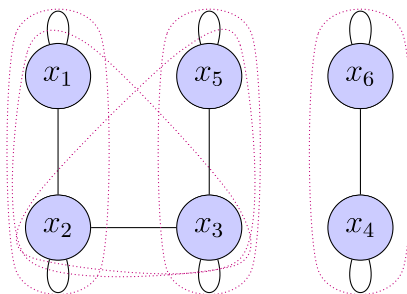
Selon la complexité de l'algorithme de filtrage, la $f_{cc}^{\cap}$-décomposition peut apporter des améliorations en temps de calcul. Cela se produit lorsque $G_{\mathcal{I}}$ est composé de nombreuses composantes connexes, plutôt homogènes en taille. Malheureusement, ce cadre est assez restrictif. Nous allons donc proposer une version relâchée de cette décomposition, dans le but d'obtenir de plus petits sous-ensembles de variables à propager.

Décomposition par voisinage

Notons $f_{vn}^{\cap}$ la multi-fonction qui énumère les parties de $\mathcal{X}$ associées au voisinage de chaque noeud dans $G_{\mathcal{I}}$ (définition 12 et figure 8.3). Sur notre exemple, effectuer la $f_{vn}^{\cap}$-décomposition de la contrainte ALLDIFFERENT($\mathcal{X}$) implique de filtrer ALLDIFFERENT sur les sous-ensembles de variables $\{x_1, x_2\}$, $\{x_1, x_2, x_3\}$, ..., $\{x_4, x_6\}$. Il y a donc plus de parties de $\mathcal{X}$ à propager (il y en a exactement $|\mathcal{X}|$), mais ces parties sont, *a priori*, plus petites. Remarquons que deux noeuds peuvent avoir le même voisinage, donc $f_{vn}^{\cap}(\mathcal{X})$ peut contenir des doublons.

Définition 12 $f_{vn}^{\cap} = \{\{x_j | j \in \delta_{G_{\mathcal{I}}}(i)\} | x_i \in \mathcal{X}\}$

L'intérêt pratique de la $f_{vn}^{\cap}$-décomposition est qu'elle permet une propagation plus légère des contraintes $f_{cc}^{\cap}$-décomposables (théorème 4), tout en garantissant le respect de son ensemble de solutions. Plus $G_{\mathcal{I}}$ est éparse, plus les voisinages des noeuds seront petits et donc plus la propagation $f_{vn}^{\cap}$-décomposée sera rapide. De plus, la $f_{vn}^{\cap}$-décomposition permet une propagation incrémentale des contraintes, en ne propageant que les parties de $\mathcal{X}$ dont les variables ont été modifiées. Malheureusement, ce gain potentiel a un prix : le filtrage $f_{vn}^{\cap}$-décomposé ne préserve pas toujours le niveau de cohérence de l'algorithme employé (théorème 5). De plus, chaque variable pouvant appartenir à plusieurs voisinages, la propagation sur une partie de $\mathcal{X}$ peut en déclencher une autre. En d'autre termes, la phase de filtrage peut dans certains cas mettre plus de temps à converger.



$$f_{vn}^{\cap}(\mathcal{X}) = \{\{x_1, x_2\}, \{x_1, x_2, x_3\},$$
$$\{x_2, x_3, x_5\}, \{x_4, x_6\}, \{x_3, x_5\}, \{x_4, x_6\}\}$$

FIGURE 8.3 – Illustration de la $f_{vn}^{\cap}$-décomposition.

Théorème 4 *Toute contrainte $f_{cc}^{\cap}$-décomposable est $f_{vn}^{\cap}$-décomposable, et réciproquement.*

Preuve : Pour chaque solution, composantes connexes et voisinages dans $G_{\mathcal{I}}$ sont confondus. □

Théorème 5 *Dans le cas général, la $f_{vn}^{\cap}$-décomposition ne préserve pas la GAC.*

Preuve : Considérons un ensemble de variables $\mathcal{X} = \{x_1, ..., x_7\}$ avec pour domaines respectifs $\{\{1,2\}, \{2,3\}, \{3,4\}, \{4,5\}, \{5,6\}, \{5,6\}, \{2,5\}\}$. Considérons l'ensemble de contraintes $\mathcal{C} = \{x_1 \neq x_2, x_2 \neq x_3, x_3 \neq x_4, x_4 \neq x_5, x_5 \neq x_6, x_6 \neq x_7, x_3 \neq x_6, x_7 \neq x_1\}$. Nous pouvons utiliser la contrainte SOMEDIFFERENT($\mathcal{X}, \mathcal{C}$) pour avoir une vue d'ensemble sur $\mathcal{C}$ et donc une propagation plus forte. L'algorithme GAC de SOMEDIFFERENT($\mathcal{X}, \mathcal{C}$) supprime la valeur 2 du domaine de x_1, alors que sa $f_{vn}^{\cap}$-décomposition en est incapable. En conséquence, la $f_{vn}^{\cap}$-décomposition ne préserve pas toujours la GAC. □

Décompositions basées sur les valeurs

Notons $\mathcal{D}$ l'union des domaines des variables. Nous introduisons alors la multi-fonction $f^{\mathcal{D}}$ pour fournir les parties de $\mathcal{X}$ associées à chaque valeur de $\mathcal{D}$ (définition 13). On a $|f^{\mathcal{D}}(\mathcal{X})| = |\mathcal{D}|$. Sur notre exemple (figure 8.4a), la $f^{\mathcal{D}}$-décomposition implique de filtrer ALLDIFFERENT sur $\{x_1, x_2\}, \{x_2, x_3\}, ..., \{x_4, x_6\}$.

Définition 13 $f^{\mathcal{D}} = \{\{x_i | v \in dom(x_i)\} | v \in \mathcal{D}\}$ *avec* $\mathcal{D} = \bigcup_{x \in \mathcal{X}} dom(x)$

Cette décomposition est assez naturelle pour les contraintes restreignant les affectations de variables à une même valeur, comme GLOBALCARDINALITY [Rég96] et BINPACKING. Pour rappel, GLOBALCARDINALITY contraint le nombre d'occurrence de chaque valeur dans $\mathcal{X}$. BINPACKING en est une généralisation pondérée : chaque variable de $\mathcal{X}$ est associée à un poids, la contrainte restreint la somme des poids des variables associées à la même valeur (boîte). Lorsqu'il n'y a pas de restriction sur le nombre d'occurrences (resp. l'occupation) minimum des valeurs (resp. boîtes), alors GLOBALCARDINALITY (resp. BINPACKING) est $f^{\mathcal{D}}$-décomposable. Il est bon de remarquer que les $f^{\mathcal{D}}$-décompositions de GLOBALCARDINALITY et BINPACKING correspondent en fait à leurs décompositions usuelles (statiques), basées respectivement sur une contrainte OCCURRENCE par valeur et une contrainte SCALAR par boîte. Le théorème 6 indique que ces contraintes sont également $f^{\cap}_{vn}$-décomposables.

Théorème 6 *Toute contrainte $f^{\mathcal{D}}$-décomposable est $f^{\cap}_{vn}$-décomposable, et réciproquement.*

Preuve : Pour chaque solution, deux noeuds de $G_{\mathcal{I}}$ sont voisins si et seulement si leurs variables sont affectées à la même valeur. □

La $f^{\mathcal{D}}$-décomposition ne préserve pas non plus la GAC dans le cas général (proposition 7). De plus, remarquons que chaque partie de $\mathcal{X}$ associée à une valeur de $\mathcal{D}$ est incluse dans au moins un voisinage de $G_{\mathcal{I}}$. On peut donc voir $f^{\cap}_{vn}$ comme un compromis entre $f^{\cap}_{cc}$ et $f^{\mathcal{D}}$.

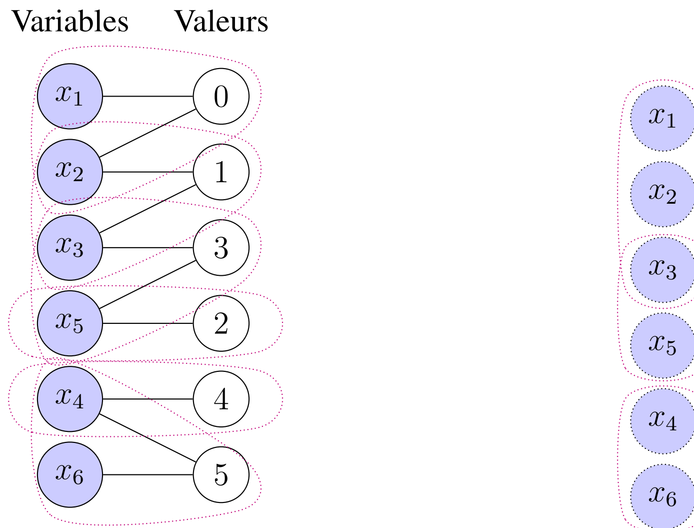
Proposition 7 *Dans le cas général, la $f^{\mathcal{D}}$-décomposition ne préserve pas la GAC.*

Preuve : La contrainte GLOBALCARDINALITY donne un contre exemple : sa $f^{\mathcal{D}}$-décomposition (équivalente à une reformulation par contraintes OCCURRENCE) est bien connue pour ne pas établir la GAC [Rég96]. □

Dans l'ensemble, on peut dire que la $f^{\mathcal{D}}$-décomposition est assez naïve. Nous allons donc l'étendre en proposant de composer les éléments de $f^{\mathcal{D}}(\mathcal{X})$.

Nous introduisons maintenant la famille de multi-fonctions $f^{\mathcal{D}}_{\mathcal{P}_*}$, pour énumérer les parties de $\mathcal{X}$ associées à *certaines* combinaisons des valeurs de $\mathcal{D}$ (définition 14). De telles décompositions offrent de nouvelles perspectives pour établir de meilleurs compromis entre puissance du filtrage et temps de calcul. Une illustration sur notre exemple est donnée en figure 8.4. Sur cet exemple, nous avons arbitrairement décidé de regrouper les valeurs en trois paires de valeurs consécutives : $\mathcal{P}_* = \{\{0, 1\}, \{2, 3\}, \{4, 5\}\}$, menant à la décomposition de $\mathcal{X}$ en $\{x_1, x_2, x_3\}$, $\{x_3, x_5\}$, et $\{x_4, x_6\}$.

Définition 14 $f^{\mathcal{D}}_{\mathcal{P}_*} = \{\{x_i | dom(x_i) \cap \mathcal{E} \neq \emptyset\} | \mathcal{E} \in \mathcal{P}_*\}$ *avec* $\mathcal{P}_* \subseteq \mathcal{P}(\mathcal{D})$



(a) $f^{\mathcal{D}} = \{\{x_1, x_2\}, \{x_2, x_3\}, \{x_5\}, \{x_3, x_5\}, \{x_4\}, \{x_4, x_6\}\}$.

(b) Exemple de multi-fonction $f^{\mathcal{D}}_{\mathcal{P}_*}(\mathcal{X}) = \{\{x_1, x_2, x_3\}, \{x_3, x_5\}, \{x_4, x_6\}\}$, avec $\mathcal{P}_* = \{\{0, 1\}, \{2, 3\}, \{4, 5\}\}$.

FIGURE 8.4 – Illustration de $f^{\mathcal{D}}$ et $f^{\mathcal{D}}_{\mathcal{P}_*}$.

Reprenons l'exemple de la contrainte BINPACKING. Supposons que l'on ait quelques centaines de boîtes. On peut appliquer un algorithme simple sur l'ensemble des variables, et un algorithme plus lourd, $f^{\mathcal{D}}_{\mathcal{P}_*}$-décomposé sur des groupes de quelques boîtes.

8.2 Implémentation générique d'une $f_{vn}^{\cap}$-décomposition

Nous proposons maintenant une structure permettant d'appliquer efficacement une $f_{vn}^{\cap}$-décomposition. Pour cela, nous introduisons un propagateur générique capable de filtrer selon la $f_{vn}^{\cap}$-décomposition de n'importe quelle contrainte $f_{vn}^{\cap}$-décomposable, dans le but d'accélérer le processus de propagation.

8.2.1 Grandes lignes

Algorithm 4 $f_{vn}^{\cap}$-décomposition

```
// Variables locales à renseigner
global Variable[] vars
global FilteringAlgorithm filter
// Structures de données internes
global int n
global Graph graph
global Set toCompute // Ensemble des variables dont le domaine a été modifié

    // Méthode appelée par le solveur une seule fois, à l'initialisation de la résolution
 1: method INITIALIZATION( )
 2:     // Initialisation du graphe d'intersection
 3:     n ← |vars|
 4:     graph ← ([1, n], ∅)
 5:     for (int i ∈ [1, n]) do
 6:         for (int j ∈ [i, n]) do
 7:             if (dom(vars[i]) ∩ dom(vars[j]) ≠ ∅) then
 8:                 graph ← graph ∪ {(i, j)} // Ajout de l'arête (i, j)
 9:             end if
10:         end for
11:     end for
12:     GLOBALFILTERING( )
13: end method

    // Méthode appelée par le solveur après une ou plusieurs modification de domaines
14: method PROPAGATION( )
15:     while (toCompute ≠ ∅) do
16:         int i ← toCompute.POP( )
17:         // Maintient du graphe d'intersection
18:         for (int j|(i, j) ∈ graph) do
19:             if (dom(vars[i]) ∩ dom(vars[j]) = ∅) then
20:                 graph ← graph \ {(i, j)} // Retrait de l'arête (i, j)
21:             end if
22:         end for
23:         // Applique l'algorithme de filtrage localement, sur le voisinage du noeud i
24:         filter.APPLYON({vars[j]|(i, j) ∈ graph})
25:     end while
26: end method

    // Méthode appelée par le solveur à chaque fois qu'un domaine de variable est modifié
27: method ONVARIABLEMODIFICATION(int i)
28:     toCompute ← toCompute ∪ {i}
29: end method

30: method GLOBALFILTERING( )
31:     filter.APPLYON(vars)
32:     toCompute ← ∅
33: end method
```

L'algorithme 4 fournit les grandes lignes pour implementer une $f_{vn}^{\cap}$-décomposition. L'algorithme se compose de deux grandes étapes : Tout d'abord, une phase d'initialisation (INITIALIZATION, lignes 1 à 13) permet de construire $G_{\mathcal{I}}$ et de propager les domaines initiaux de toutes les variables en une passe (GLOBALFILTERING, lignes 30 à 33). Ensuite, une propagation incrémentale appelle l'algorithme de filtrage ($filter$) de la contrainte décomposée, sur les sous-ensembles de variables associés aux voisinages des variables dans $G_{\mathcal{I}}$, des variables modifiées

(PROPAGATION, lignes 14 à 26). Cette propagation est déclenchée automatiquement par le solveur lorsque des variables ont été filtrées. L'ensemble des variables modifiées est mis à jour via le patron de conception *Observer/Observable* (ONVARIABLEMODIFICATION, lignes 27 à 29). Pour chaque variable $x \in \mathcal{X}$ qui a été modifiée, on applique $filter$ sur son voisinage, *i.e.,* sur la partie maximale de $\mathcal{X}$ telle que chaque variable a au moins une valeur en commun avec x.

Le but de cette structure est d'accélérer la propagation lorsque son point fixe ne nécessite vraisemblablement pas de considérer l'ensemble des variables de la contrainte. Par exemple, dans le cas d'un algorithme cubique, il peut être préférable d'effectuer quelques itérations sur des sous-ensembles de quelques dizaines de variables plutôt qu'une seule passe sur un millier de variables.

Maintenant que nous avons vu l'essence de cette structure, nous allons voir comment la raffiner pour la rendre plus performante. Nous allons étudier quelques améliorations génériques mais, à l'évidence, il en existe bien d'autres.

8.2.2 Aspects avancés

Cette section fournit des améliorations simples mais significatives de l'algorithme 4.

Algorithm 5 Réduction des comportements pathologiques

global final double $\alpha \leftarrow 2$ // valeur arbitraire dans $[0, n]$

1: **method** PROPAGATION( )
2: **int** $total \leftarrow 0$
3: **for** (**int** $i \in toCompute$) **do**
4: $total \leftarrow total + |graph.neighbors[i]|$
5: **end for**
6: // Critère heuristique pour limiter les comportements pathologiques
7: **if** ($total > \alpha.n$) **then**
8: // La $f_{vn}^{\cap}$-décomposition ne semble pas pertinente
9: GLOBALFILTERING( )
10: **else**
11: // Même instructions que dans la méthode PROPAGATION de l'algorithme 4
12: **end if**
13: **end method**

Algorithm 6 Propagation conditionnelle

1: **method** ONVARIABLEMODIFICATION(`int` i)
2: // Critère heuristique pour ignorer les propagation qui n'apporteront pas d'inférence
3: // La fonction booléenne CONDITION(i) peut être implémentée de nombreuses manières
4: // (e.g., $|dom(vars[i])| < 10$, voire même un test aléatoire)
5: **if** ($vars[i]$.INSTANTIATED( ) $\vee$ CONDITION(i)) **then**
6: $toCompute \leftarrow toCompute \cup \{i\}$
7: **end if**
8: **end method**

Premièrement, la phase d'initialisation de $G_{\mathcal{I}}$ peut être améliorée à l'aide d'un pré-calcul basé sur un algorithme d'intersection de droites [BO79], communément appelé *sweep*. L'idée est de d'abord calculer rapidement les paires de noeuds pour lesquelles l'approximation de leurs domaines à leurs bornes est en intersection. Si les domaines contiennent des trous, alors

une deuxième passe sur ces arêtes potentielles vérifiera qu'il existe bien une valeur commune à ces variables.

Deuxièmement, on observe que lorsque trop de variables ont été modifiées, alors il est souvent préférable de débrancher l'auto-décomposition et propager normalement la contrainte une fois pour toutes sur l'ensemble de ses variables. Un exemple pathologique est le cas où $G_{\mathcal{I}}$ forme un graphe complet où chaque variable a été modifiée : l'auto-décomposition naïve déclencherait n fois l'algorithme de filtrage sur l'ensemble des variables, alors que seule la première passe suffit. En conséquence, afin de réduire ce type de comportement pathologique, nous ajoutons une condition au déclenchement de l'auto-décomposition (algorithme 5). Dans notre exemple, nous proposons un peu arbitrairement comme critère le fait que la somme des tailles des voisinages à propager soit inférieure à $2n$. Bien que cette condition soit générale et assez pertinente en moyenne, elle gagnerait à être spécialisée d'une contrainte à une autre, selon la complexité des algorithmes de filtrages impliqués.

Enfin, dans de nombreux problèmes de grande taille, il n'est pas rentable de rechercher de hauts niveaux de cohérence [Per11]. Plus précisément, il n'est pas toujours intéressant d'atteindre le point fixe usuel que l'on obtient en propageant toutes les modifications de domaines. A l'inverse, on peut gagner du temps en ne propageant pas la modification d'une variable qui, vraisemblablement, n'aura pas un impact très fort sur les domaines des autres variables. Pour cette raison, nous suggérons d'ajouter une condition avant de propager la modification d'une variable (algorithme 6). Cette condition peut être basée sur le domaine de la variable qui vient d'être modifiée, comme par exemple : si la variable est instanciée ; pour une variable ensembliste, qu'elle ait une borne inférieure non vide ; pour une variable de tâche (voir section 8.3), qu'elle ait une partie obligatoire non vide (ce qui revient au cas précédent). Plus généralement, la condition au filtrage peut être n'importe quel test booléen (même aléatoire), du moment que toute variable instanciée est propagée, afin de garantir la préservation de l'ensemble de solutions de la contrainte. On peut donc utiliser l'algorithme 6 afin de réduire le nombre d'appels à $filter$, tout en préservant un vérificateur de solutions. Il est bon de remarquer que cette amélioration est aussi valable sans auto-décomposition. Si notre approche est ici assez empirique, des travaux théoriques ont déjà été menés pour évaluer la probabilité que la propagation d'une modification de domaine soit intéressante [DBGLT13].

8.3 Etude de la contrainte CUMULATIVE

Cette section illustre le concept d'auto-décomposition de contrainte globale sur la contrainte CUMULATIVE. Dans un premier temps, nous expliquons comment rendre cette contrainte $f_{vn}^{\cap}$-décomposable. Puis, nous proposons une étude expérimentale validant notre approche.

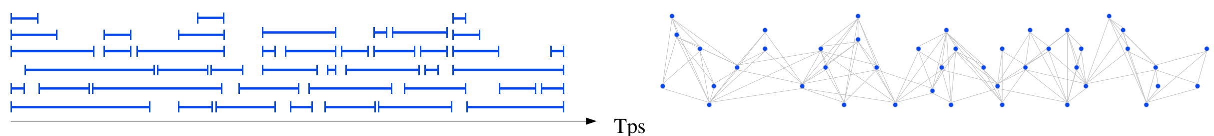
8.3.1 $f_{vn}^{\cap}$-décomposition de la contrainte

Pour éviter toute confusion possible, nous commençons par rappeler la définition de la contrainte CUMULATIVE. La contrainte CUMULATIVE assure que la consommation en ressource d'un ensemble de tâches non-préemptives ne dépasse jamais une capacité fixe donnée. Nous précisons que dans le cas où la capacité est une variable, il est toujours possible de considérer une capacité fixe élevée et d'ajouter une tâche fictive recouvrant tout l'horizon de planification pour simuler l'aspect variable de la capacité. Une tâche est définie par une date de début, une date de fin et une consommation (positive ou nulle) en ressource. Classiquement, une tâche est donc encodée sous la forme de plusieurs variables entières. Dans ce cas, CUMULATIVE n'est pas $f_{vn}^{\cap}$-décomposable. D'ailleurs, il ne fait aucun sens de calculer le graphe d'intersection de va-

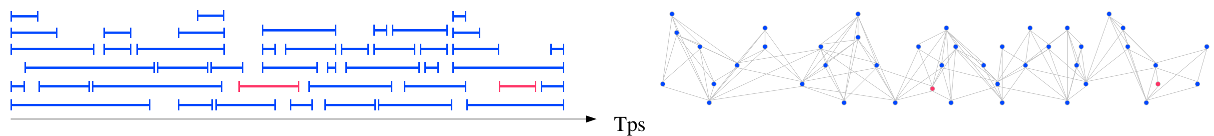
riables hétérogènes. Cependant, en montant légèrement en abstraction, la $f^{\cap}_{vn}$-décomposition de CUMULATIVE devient possible.

Nous considérons maintenant un ensemble de variables abstraites $\mathcal{X}$ pour représenter les tâches. Nous les appelons donc des variables de tâche. Nous utilisons alors la signature de contrainte suivante : CUMULATIVE`(TaskVariable[]` $\mathcal{X}$`, Int Capacity)`. Grâce à cette abstraction conceptuelle, les variables sont désormais toutes homogènes. Nous définissons l'intersection de deux domaines de variables de tâche comme suit : deux variables de tâche ont une intersection de domaines non nulle si et seulement si leurs enveloppes se recoupent et leurs consommations en ressource sont toutes les deux strictement positives. Nous précisons que le terme enveloppe désigne la fenêtre de temps obtenue en prenant la date de début au plus tôt et la date de fin au plus tard de la tâche. Ainsi, $G_{\mathcal{I}}$ est principalement structuré par le temps. De plus, il exclut naturellement les tâches ne consommant pas de ressource. Puisque la propriété assurée par la contrainte est que la capacité de la ressource ne doit être dépassée en aucun point de temps, et puisque les tâches pouvant se chevaucher dans le temps sont voisines dans $G_{\mathcal{I}}$, CUMULATIVE est $f^{\cap}_{vn}$-décomposable. Cela peut être naturellement généralisé au cas multi-ressources. Plus généralement encore, cette méthodologie basée sur une montée en abstraction peut être employée pour auto-décomposer les contraintes DISJUNCTIVE, DIFFN et GEOST.

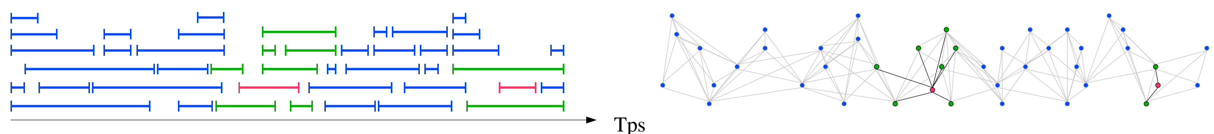
La $f^{\cap}_{vn}$-décomposition permet alors une propagation incrémentale de la contrainte CUMULATIVE : Lorsque deux activités sont modifiées, on propage séquentiellement CUMULATIVE sur le voisinage directe de chacune de ces deux activités. Il s'agit donc d'une forme d'incrémentalité très locale, basée sur une graphe d'intersection (voir figure 8.5). Cela la distingue de l'approche géométrique de [BC01], qui propagerait une seule fois la contrainte sur l'ensemble des activités inclues dans l'intervalle maximum englobant ces deux activités. Typiquement, si l'activité démarrant le plus tôt et celle finissant le plus tard sont modifiées, leur méthode propagera la contrainte CUMULATIVE sur l'ensemble de toutes les activités.



(a) Ensemble d'activités à ordonnancer, représentées par des intervalles de temps, et son graphe d'intersection.



(b) Le domaine des activité en rouge vient d'être modifié.



(c) Propagation incrémentale de la contrainte CUMULATIVE sur les activités modifiées (en rouge) et leurs voisins (en vert). Les autres activités (en bleu) sont exclues de la propagation.

FIGURE 8.5 – Propagation incrémentale pour CUMULATIVE basée sur la $f^{\cap}_{vn}$-décomposition.

8.3.2 Autres auto-décompositions de la contrainte

$f^{\cap}_{cc}$-décomposition

Puisque la contrainte est $f^{\cap}_{vn}$-décomposable, elle est naturellement $f^{\cap}_{cc}$-décomposable. Cette décomposition nécessite donc des groupes de tâches qui soient disjoints dans le temps. Si cela peut paraître trop restrictif dans le cas général, cette décomposition peut néanmoins faire sens pour des applications d'ordonnancement où la production est interrompue régulièrement (e.g., le soir ou le weekend).

$f^{\mathcal{D}}$-décomposition

La $f^{\mathcal{D}}$-décomposition signifie d'avoir, pour chaque point de temps t, l'ensemble des tâches dont l'enveloppe contient t. Cette décomposition n'a, *a priori*, aucune utilité pratique.

$f^{\mathcal{D}}_{\mathcal{P}_*}$-décomposition

En découpant l'horizon de planification en intervalles disjoints, on peut créer différentes $f^{\mathcal{D}}_{\mathcal{P}_*}$-décompositions qui soient pertinentes pour la contrainte CUMULATIVE. Ce découpage peut soit être régulier, soit avoir une structure particulière, liée à la densité des intervalles en nombre de tâches par exemple. On peut ainsi découper un horizon de planification d'un an en 52 semaines et ne propager la contrainte que semaine par semaine, afin de gagner un facteur d'échelle. Une telle auto-décomposition semble particulièrement pertinente si elle est utilisée dans le cadre d'une recherche à voisinage large (LNS, de l'anglais *Large Neighborhood Search*) [Sha98] basée sur ce même schéma, optimisant une solution semaine par semaine. On évite ainsi de perdre un temps considérable en traitant les tâches fixées qui ne sont d'aucune utilité pour la réparation de la semaine considérée.

8.3.3 Intérêt pratique

Améliorer la capacité de la programmation par contraintes à résoudre des problèmes d'ordonnancement et de placement de grande taille est reconnu comme étant un des enjeux majeurs pour la communauté [Fut11]. L'auto-décomposition de contraintes globales apporte un élément de réponse à cette vaste problématique. En effet, il est réaliste d'avoir de nombreuses variables, mais un graphe d'intersection relativement éparse. Cela se produit lorsque l'on planifie sur un horizon de temps important et que des contraintes (e.g., précédences et dates de livraison) structurent les tâches entre elles. Même si les tâches ne sont pas précisément fixées dans le temps, il arrive qu'elles aient une fenêtre de temps relativement restreinte. Par exemple, on peut connaitre le jour, la semaine, le mois *etc.* où la tâche sera réalisée. On peut donc s'attendre à ce que seule une petite partie des tâches ait des domaines en intersection. Nous allons maintenant le montrer empiriquement.

Pour illustrer l'impact de la $f^{\cap}_{vn}$-décomposition de CUMULATIVE, nous comparons son implémentation classique (`FULL`), se propageant sur l'ensemble des variables, avec sa $f^{\cap}_{vn}$-décomposition (`AUTO`), implémentée avec les astuces décrites en section 8.2.2. Les tests ont été réalisés sur un `Mac Pro` ayant 6-core `Intel Xeon` à 2.93 Ghz, avec `MacOS 10.6.8`, et `Java 1.7`. Chaque exécution disposait d'un coeur et d'une limite de temps de cinq minutes.

Nous commençons notre étude expérimentale par un problème cumulatif simple que nous avons généré. Nous cherchons à trouver un ordonnancement pour un ensemble de tâches satisfaisant une contrainte CUMULATIVE dont la capacité est variable et doit être minimisée. Ce problème est NP-difficile, d'où l'intérêt de la programmation par contraintes par rapport à un simple glouton. Pour mettre en relief l'impact de l'auto-décomposition sur le passage à

l'échelle, nous avons généré aléatoirement 250 instances ayant de $1,000$ à $20,000$ tâches. Nous notons n le nombre de tâches. Pour chaque tâche, le domaine initial pour sa date de début, sa durée (fixe) et sa consommation (fixe) a été pris dans les intervalles respectifs $[1, n]$, $[1, 50]$ et $[1, 5]$. Ce générateur permet donc d'obtenir des instances structurées dans le temps. Le but de cette démarche est de mesurer le gain de la $f_{vn}^{\cap}$-décomposition de CUMULATIVE dans un cas *favorable*. Pour filtrer la contrainte CUMULATIVE, nous utilisons l'algorithme dit de *sweep* de l'état de l'art [LCB13]. La capacité initiale n'étant pas bornée, il est trivial de trouver une première solution avec un glouton. Néanmoins, une approche en programmation par contraintes peut peiner à y arriver sur de grandes instances car la contrainte CUMULATIVE doit être propagée à chaque noeud de l'arbre de recherche, engendrant un surcoût important. En conséquence, nous étudions d'abord la capacité à trouver une première solution et ensuite la capacité à trouver une bonne solution. Les résultats figurent sur la table 8.1. En ce qui concerne la recherche d'une solution réalisable, AUTO permet d'en trouver pour chaque instance, alors que FULL ne parvient à résoudre aucune instance ayant de $15,000$ à $20,000$ tâches. La $f_{vn}^{\cap}$-décomposition apporte un facteur d'accélération allant de 2.8 à 16.6. Elle permet de résoudre les instances à $20,000$ tâches en moins d'une minute en moyenne alors qu'il faut trois minutes pour résoudre une instance à $10,000$ noeuds avec l'approche traditionnelle. AUTO a donc le temps d'explorer une plus grande partie de l'espace de recherche. Plus précisément, on observe en moyenne un facteur supérieur à 10 sur le nombre de noeuds explorés dans le temps imparti. Cela permet ainsi à l'approche décomposée de trouver de meilleures solutions. En moyenne AUTO permet de trouver des solutions 28.4% et 26.1% meilleures pour les instances ayant respectivement $5,000$ et $10,000$ tâches. Dans l'ensemble, la $f_{vn}^{\cap}$-décomposition améliore significativement la contrainte CUMULATIVE sur ces instances générées aléatoirement mais de manière structurée.

		Nombre de tâches				
		$1,000$	$5,000$	$10,000$	$15,000$	$20,000$
Nombre d'instances résolues*	FULL	50	50	50	0	0
	AUTO	50	50	50	**50**	**50**
Temps de résolution moyen (s) pour trouver la première solution	FULL	1.4	36.2	179.9	-	-
	AUTO	**0.5**	**3.0**	**10.8**	**27.9**	**57.1**
	$\frac{\text{FULL}}{\text{AUTO}}$	**2.8**	**12.3**	**16.6**	-	-
Nombre de noeuds moyen	FULL	164k	42k	15k	9k	6k
	AUTO	**638k**	**392k**	**301k**	**165k**	**100k**
	$\frac{\text{AUTO}}{\text{FULL}}$	**3.9**	**9.3**	**19.9**	**18.6**	**16.6**
Valeur moyenne de la meilleure solution	FULL	85	124	142	-	-
	AUTO	85	**89**	**105**	**129**	**137**
	$\frac{\text{FULL}-\text{AUTO}}{\text{FULL}}$	**0.1%**	**28.4%**	**26.1%**	-	-

* : nombre d'instances pour lesquelles au moins une solution réalisable a été trouvée.

TABLE 8.1 – Etude de passage à l'échelle pour la contrainte CUMULATIVE sur des instances aléatoires. Comparaison de sa version classique (FULL), propagée sur toutes les variables, à sa $f_{vn}^{\cap}$-décomposition (AUTO). La résolution est soumise à une limite de cinq minutes.

Nous étudions ensuite l'auto-décomposition de CUMULATIVE sur un problème réel, dont les instances nous ont été gentiment fournies par Arnaud Letort et Helmut Simonis. Ce problème

consiste à trouver un ordonnancement pour un ensemble de tâches qui satisfasse des contraintes CUMULATIVE (jusqu'à huit) et des contraintes de précédence. Le nombre de tâche varie de $6,000$ à $16,000$. Contrairement à l'exemple précédent généré aléatoirement par nos soins, cet exemple n'inclut pas de fenêtre de temps qui soit restrictive. La structuration des tâches s'effectue par le biais des contraintes de précédence. Les résultats sont donnés sur la figure 8.6. Clairement, AUTO améliore significativement FULL. En moyenne, la $f_{vn}^{\cap}$-décomposition de CUMULATIVE permet de gagner un facteur 5 sur le temps de résolution. En conclusion, l'intérêt pratique de l'auto-décomposition de contraintes globales ne se limite pas uniquement aux instances générées artificiellement, mais concerne également des instances réelles.

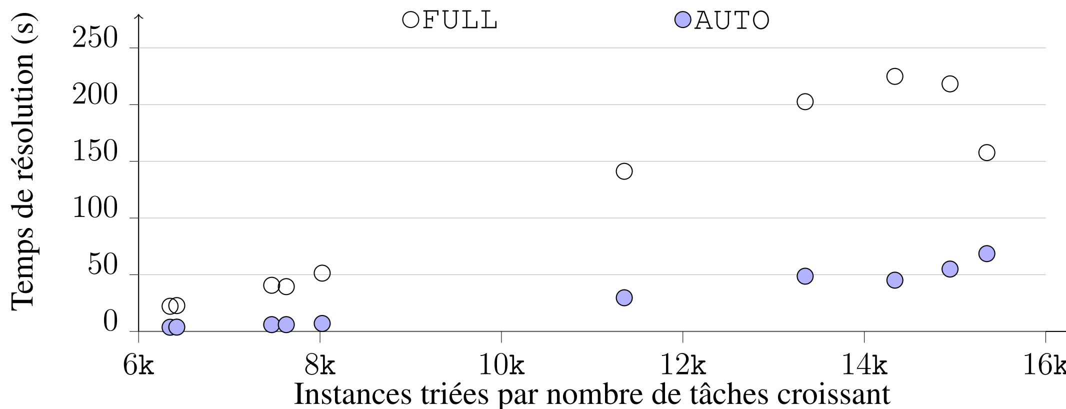


FIGURE 8.6 – Impact de l'auto-décomposition de CUMULATIVE sur un problème d'ordonnancement industriel. Chaque point représente la résolution d'une instance.

Le gain en passage à l'échelle n'est pas le seul intérêt de l'auto-décomposition de contraintes. Dans certains cas, elle permet même de filtrer plus, en tirant parti de tendances locales comme des périodes de pic d'activité ou au contraire des périodes creuses. Cela se produit par exemple sur un problème de planification d'atelier nommé *jobshop*[1]. Ce problème n'implique directement aucune fenêtre de temps restrictive pour les tâches mais il contient plusieurs contraintes CUMULATIVE ainsi que des contraintes de précédence entre les tâches. L'objectif du problème est de minimiser la durée du planning. Pour résoudre ce problème, nous utilisons la configuration par défaut de Choco-3.2, qui inclut le raisonnement classique appelé *time-table* que l'on retrouve dans l'algorithme de *sweep*, mais également un raisonnement énergétique partiel *ad hoc* qui améliore significativement le filtrage de la contrainte sans introduire de surcoût algorithme trop lourd. Comme on peut le voir sur la figure 8.2, l'approche décomposée est bien plus efficace que la version classique. Il améliore la résolution de quatre des huit instances traitées. Par exemple, pour jobshop_mt06.fzn, la version décomposée est capable de prouver l'optimalité de la solution en moins d'une seconde alors que la version classique n'y parvient pas dans les cinq minutes allouées. En augmentant la limite de temps à trente minutes, elle parcourt des millions de noeuds sans pour autant parvenir à fermer l'instance. Le gain n'est donc clairement pas lié à la vitesse de propagation mais à la puissance de celle-ci. En fait, la décomposition permet au raisonnement énergétique partiel de tirer parti des périodes de pic d'activité pour filtrer et détecter des incohérences très tôt. A l'inverse, lorsqu'il est appliqué sur l'ensemble des variables, cet algorithme est moins puissant car les périodes de pics sont compensées par les périodes creuses. En conclusion, l'auto-décomposition permet parfois de combler les lacunes d'algorithmes de filtrage partiels utilisés lorsqu'il est trop coûteux d'assurer un filtrage complet.

[1] https ://github.com/MiniZinc/minizinc-benchmarks/blob/master/jobshop2

Instance	Valeur de la meilleure solution trouvée		Temps de résolution (s)	
	FULL	AUTO	FULL	AUTO
jobshop_abz6.fzn	985	985	300	300
jobshop_la01.fzn	816	816	300	300
jobshop_la02.fzn	764	**733**	300	300
jobshop_la03.fzn	758	**714**	300	300
jobshop_la04.fzn	682	682	300	300
jobshop_la05.fzn	593*	593*	4.0	**3.4**
jobshop_mt06.fzn	55	**55***	300	**0.4**
jobshop_mt10.fzn	1092	**1084**	300	300

* : solution prouvée optimale

TABLE 8.2 – Résolution d'un problème de jobshop. Comparaison de la valeur des meilleures solutions trouvées par FULL et AUTO, dans les cinq minutes allouées à la résolution de chaque instance.

8.4 Conclusion du chapitre

Nous avons introduit la notion de contraintes auto-décomposables, une large famille de contraintes pouvant être filtrées sur des parties de leurs variables uniquement, tout en préservant leurs ensembles de solutions. Cette famille inclut les contraintes globales ALLDIFFERENT, SOMEDIFFERENT, DISJUNCTIVE, DIFFN, CUMULATIVE, GEOST et, en ce qui concerne les restrictions sur les max, GLOBALCARDINALITY et BINPACKING. L'auto-décomposition peut également s'appliquer à une multitude de contraintes *ad hoc*, qui maintiennent une certaine propriété sur des variables qui sont *proches* selon une ou plusieurs dimensions (le temps, l'espace, *etc.*).

Nous avons formalisé deux décompositions de contraintes globales existantes, la $f^{\cap}_{cc}$-décomposition et la $f^{\mathcal{D}}$-décomposition, et en avons suggéré une approche dynamique. De plus, nous avons étendu ces décompositions pour en créer de nouvelles, la $f^{\cap}_{vn}$-décomposition et la famille des $f^{\mathcal{D}}_{\mathcal{P}_*}$-décompositions, qui offrent de nombreux compromis possibles entre puissance du filtrage et temps de calcul.

Nous nous sommes restreints au cas fort où l'algorithme de la contrainte globale décomposée s'applique tel quel aux sous-ensembles de variables induits par la décomposition. On pourrait étendre ce travail en autorisant un traitement supplémentaire, tel une agrégation de bornes locales par exemple.

De plus, nous avons proposé un schéma d'implémentation générique pour réaliser la $f^{\cap}_{vn}$-décomposition d'une contrainte globale. Ce schéma tire profit de la structure des domaines pour rendre un algorithme de filtrage incrémental, et ainsi espérer réduire le temps de propagation. Cet algorithme est enrichi de quelques conditions dans le but d'éviter tout comportement pathologique. De manière générale, puisque la décomposition est réalisée de manière dynamique, on préserve un point de vue global sur la contrainte. Il est donc possible d'activer ou de désactiver l'auto-décomposition à souhait durant la recherche, selon qu'elle semble pertinente ou non.

Nous avons illustré ce concept sur la contrainte CUMULATIVE, qui apparait dans de nombreuses applications de programmation par contraintes. Cette contrainte peut impliquer un grand nombre de tâches qui soient déjà grossièrement réparties dans le temps. Nos résultats expérimentaux démontrent l'intérêt pratique de notre approche sur certains problèmes à large échelle. De plus, elle met également en lumière l'intérêt d'appliquer localement des raisonnements plus forts. La décomposition par voisinage a été mise en place avec succès dans le solveur Choco pour les contraintes CUMULATIVE et DIFFN. Il serait intéressant d'élargir notre étude expérimentale en incluant d'autres contraintes, comme par exemple la contrainte BIN-

PACKING. Néanmoins, l'auto-décomposition ne conviendra pas à toutes les contraintes, ni à toutes les instances possibles. Pour ce qui est de la contrainte ALLDIFFERENT par exemple, les algorithmes existant sont déjà très efficaces. De plus, il est assez rare qu'une contrainte ALLDIFFERENT porte sur des milliers de variables et dont les domaines sont structurés. Il est plus fréquent d'avoir de nombreuses contraintes ALLDIFFERENT, chacune portant sur un nombre raisonnable de variables.

Un autre axe d'amélioration de notre évaluation empirique porte sur l'étude de l'auto-décomposition au sein d'une recherche à voisinage large (LNS, de l'anglais *Large Neighborhood Search*) [Sha98]. Considérons par exemple une contrainte CUMULATIVE au sein d'une LNS relâchant les variables associées à un intervalle de temps glissant. Ce choix de LNS est une approche très répandue, notamment lorsque l'on cherche à minimiser la fin de l'ordonnancement. Alors, la $f^{\cap}_{vn}$-décomposition (où une $f^{\mathcal{D}}_{\mathcal{P}_*}$-décomposition spécifique) n'impliquera naturellement que les variables relâchées et les quelques variables en recoupement partiel avec cet intervalle de temps. Autrement dit, l'auto-décomposition de contraintes globales est une manière d'éliminer la plupart des variables fixées et d'ainsi améliorer la complexité pratique des algorithmes de filtrage. Ceci vient répondre à un besoin exprimé par [Per13].

Etonnamment, nos travaux sont un peu à contre courant, puisque notre approche est à l'inverse des derniers travaux portant sur l'amélioration du passage à l'échelle de la contrainte CUMULATIVE [LCB13]. En effet, alors qu'ils suggèrent d'accélérer l'atteinte du point fixe en n'utilisant qu'une seule *méta* contrainte globale représentant plusieurs contraintes CUMULATIVE, nous proposons de décomposer chaque contrainte CUMULATIVE et de ne pas chercher à atteindre le point fixe usuel, mais de se contenter d'un niveau de cohérence plus faible. Chacune des deux approches présente son intérêt et est plus ou moins intéressante selon l'instance traitée. Néanmoins, notre approche est plus simple à mettre en oeuvre car le schéma d'implémentation proposé est à la fois très simple et générique. Il est valable quelle que soit la contrainte auto-décomposable traitée, alors que pour filtrer des conjonctions de contraintes globales, il faut introduire de nouveaux algorithmes qui complexifient petit à petit l'univers des contraintes globales.

Nous concluons ce chapitre en soulignant que, les graphes étant avant tout des catalyseurs d'idées, les f-décompositions que nous avons suggéré pourront sûrement servir à inspirer d'autres travaux complètement différents, portant sur les heuristiques de recherche ou l'analyse de conflits par exemple.

9

Conclusion

Nous avons vu dans cette thèse diverses façons d'utiliser les graphes pour améliorer la résolution de problèmes en programmation par contraintes : filtrage basé sur des propriétés de graphe (chapitres II.5 et III.7), filtrage reformulé en terme de graphe (chapitre III.7) et filtrage décomposé selon un graphe (chapitre III.8). En plus de ces considérations techniques, nous avons également abordé des points, plus généraux, liés à la modélisation même d'un problème de graphe orienté et à la façon d'explorer un arbre de recherche (chapitres II.6 et III.7). Enfin, nous avons implémenté ces idées avec soin au sein du solveur `Choco`, de telle sorte qu'elles puissent être enrichies à l'avenir. Nous avons mené de nombreuses expérimentations, d'une part, pour mieux comprendre les mécanismes de résolution en jeu, et d'autre part, pour démontrer la compétitivité de nos approches et contribuer ainsi à la diffusion de la programmation par contraintes.

Nos résultats montrent clairement la capacité de la programmation par contraintes à passer à l'échelle sur de nombreux problèmes. Par exemple, nous avons pu résoudre des instances du problème de la recherche d'un cycle Hamiltonien dans un graphe éparse à 40000 noeuds en l'espace de 10 secondes (chapitre II.6). Nous avons également vu comment l'heuristique de recherche permettait d'augmenter de 50% la taille des instances du problème du voyageur de commerce que l'on peut traiter en programmation par contraintes (chapitre II.6). De plus, nous avons résolu à l'optimum des problèmes d'affectation de ressources impliquant plus d'un millier d'activités, passant ainsi devant les approches par programmation linéaire en nombres entiers (chapitre III.7), ainsi que des instances cumulatives impliquant quelques dizaines de milliers d'activités (chapitre III.8).

Au delà des considérations d'échelle, nous avons montré qu'un bon niveau de filtrage permettait à la programmation par contraintes de rivaliser avec l'usage d'explications sur des problèmes difficiles de circuits Hamiltoniens (chapitre II.5). De plus, nous avons proposé un cadre générique permettant d'appliquer des algorithmes de filtrage, potentiellement coûteux d'un point de vue algorithmique, sur des sous-ensembles de variables. Cette plateforme est un outil qui permettra de concevoir et d'exploiter au mieux de nombreux algorithmes de filtrage.

Concernant la diffusion de ces travaux, l'algorithme de filtrage linéaire pour la contrainte TREE du chapitre II.5 a donné lieu à une publication à la conférence internationale *CP'11* [FL11]. L'étude expérimentale sur les heuristiques de branchement du chapitre II.6 a été acceptée dans le journal international *Constraints* [FLR14]. Le chapitre III.7 à été publié à la conférence internationale *CP'13*, où il a reçu le prix *best student paper*, et dans le journal in-

ternational *Artificial Intelligence* [FL14]. Enfin, le chapitre III.8 a été publié à la conférence internationale *ECAI'14* [FLP14].

Enfin, si ce travail de thèse a principalement porté sur la notion de propagation de contraintes, il offre également de bonnes perspectives pour le branchement, avec la notion d'heuristique audacieuse. Les expériences réalisées à propos des heuristiques de branchement avaient initialement pour unique but d'améliorer la compétitivité de nos approches face à l'état de l'art. Un objectif bien pragmatique. Néanmoins, à deux reprises, l'énergie ainsi déployée a conduit à des résultats très intéressants : la redécouverte de *Last Conflict* et la découverte des heuristiques audacieuses. Si le premier sujet semble déjà relativement clos [LSTV09], je suis convaincu que le deuxième a un potentiel considérable. Comme détaillé en section 7.5 du chapitre III.7, ces heuristiques devraient apporter des gains significatifs sur de nombreux problèmes, notamment si elles se basent sur des algorithmes gloutons. Bien sûr, le plus élégant serait d'en concevoir une générique, qui parvienne ainsi à exploiter automatiquement la structure du modèle. Nous avons déjà identifié un bon nombre de pistes permettant d'aller en ce sens. Néanmoins, parvenir à ce résultat représente à nos yeux un vrai défi pour l'intelligence artificielle.

FIGURE 9.1 – Les graphes sont partout.

Bibliographie

[ABC+09] Magnus Ågren, Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey, Emmanuel Poder, Rida Sadek, Sbihi Mohamed, Charlotte Truchet, and Stéphane Zampelli. A generic geometrical constraint kernel in space and time for handling polymorphic k-dimensional objects, 2009. Confidential report of the Net-WMS European project. 11, 98

[ABCC07] David L. Applegate, Robert E. Bixby, Vasek Chvatal, and William J. Cook. *The Traveling Salesman Problem : A Computational Study (Princeton Series in Applied Mathematics)*. Princeton University Press, Princeton, NJ, USA, 2007. 55

[AER+10] Sigal Asaf, Haggai Eran, Yossi Richter, Daniel P. Connors, Donna L. Gresh, Julio Ortega, and Michael J. Mcinnis. Applying constraint programming to identification and assignment of service professionals. In David Cohen, editor, *Principles and Practice of Constraint Programming – CP 2010*, volume 6308 of *Lecture Notes in Computer Science*, pages 24–37. Springer Berlin Heidelberg, 2010. Available from : http://dx.doi.org/10.1007/978-3-642-15396-9_5, doi:10.1007/978-3-642-15396-9_5. 102, 104

[AHLT99] Stephen Alstrup, Dov Harel, Peter W. Lauridsen, and Mikkel Thorup. Dominators in linear time. *SIAM J. Comput.*, 28(6) :2117–2132, August 1999. Available from : http://dx.doi.org/10.1137/S0097539797317263, doi:10.1137/S0097539797317263. 22, 44

[ARU13] Arum : adaptive production management, 2013. European project, grant no 314056. Available from : http://arum-project.eu. 11

[Bar03] Roman Barták. Dynamic global constraints in backtracking based environments. *Annals of Operations Research*, 118(1-4) :101–119, 2003. Available from : http://dx.doi.org/10.1023/A:1021805623454, doi:10.1023/A:1021805623454. 102, 103

[BC01] Nicolas Beldiceanu and Mats Carlsson. Pruning for the minimum constraint family and for the number of distinct values constraint family. In Toby Walsh, editor, *Principles and Practice of Constraint Programming – CP 2001*, volume 2239 of *Lecture Notes in Computer Science*, pages 211–224. Springer Berlin Heidelberg, 2001. Available from : http://dx.doi.org/10.1007/3-540-45578-7_15, doi:10.1007/3-540-45578-7_15. 78, 82, 112

[BCDP07] Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey, and Thierry Petit. Global constraint catalogue : Past, present and future. *Constraints*, 12(1) :21–62, 2007. Available from : http://dx.doi.org/10.1007/

s10601-006-9010-8, doi:10.1007/s10601-006-9010-8. 12, 26, 29, 31, 32

[BCRT05] Nicolas Beldiceanu, Mats Carlsson, Jean-Xavier Rampon, and Charlotte Truchet. Graph invariants as necessary conditions for global constraints. In Peter van Beek, editor, *Principles and Practice of Constraint Programming – CP 2005*, volume 3709 of *Lecture Notes in Computer Science*, pages 92–106. Springer Berlin Heidelberg, 2005. Available from : http://dx.doi.org/10.1007/11564751_10, doi:10.1007/11564751_10. 12

[BCS+14] David Bergman, Andre A. Cire, Ashish Sabharwal, Horst Samulowitz, Vijay Saraswat, and Willem-Jan van Hoeve. Parallel combinatorial optimization with decision diagrams. In Helmut Simonis, editor, *Integration of AI and OR Techniques in Constraint Programming CPAIOR – 2014*, volume 8451 of *Lecture Notes in Computer Science*, pages 351–367. Springer International Publishing, 2014. Available from : http://dx.doi.org/10.1007/978-3-319-07046-9_25, doi:10.1007/978-3-319-07046-9_25. 62

[Bel00] Nicolas Beldiceanu. Global constraints as graph properties on a structured network of elementary constraints of the same type. In Rina Dechter, editor, *Principles and Practice of Constraint Programming – CP 2000*, volume 1894 of *Lecture Notes in Computer Science*, pages 52–66. Springer Berlin Heidelberg, 2000. Available from : http://dx.doi.org/10.1007/3-540-45349-0_6, doi:10.1007/3-540-45349-0_6. 29

[Ben04] Thierry Benoist. *Relaxations et décompositions combinatoires*. Thèse de doctorat, Université d'Avignon et des Pays de Vaucluse, Juin 2004. 11, 12

[BFL05] Nicolas Beldiceanu, Pierre Flener, and Xavier Lorca. The tree constraint. In Roman Barták and Michela Milano, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems – CPAIOR 2005*, volume 3524 of *Lecture Notes in Computer Science*, pages 64–78. Springer Berlin Heidelberg, 2005. Available from : http://dx.doi.org/10.1007/11493853_7, doi:10.1007/11493853_7. 5, 13, 29, 38, 41, 42, 48, 49

[BFL08] Nicolas Beldiceanu, Pierre Flener, and Xavier Lorca. Combining tree partitioning, precedence, and incomparability constraints. *Constraints*, 13(4) :459–489, 2008. Available from : http://dx.doi.org/10.1007/s10601-007-9040-x, doi:10.1007/s10601-007-9040-x. 12, 29

[BHH+06] Christian Bessiere, Emmanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, and Toby Walsh. Filtering algorithms for the nvalue constraint. *Constraints*, 11(4) :271–293, December 2006. Available from : http://dx.doi.org/10.1007/s10601-006-9001-9, doi:10.1007/s10601-006-9001-9. 78, 79, 82, 83

[BHLS04] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In Ramon López de Mántaras and Lorenza Saitta, editors, *European Conference on Artificial Intelligence – ECAI 2004*, pages 146–150. IOS Press, 2004. Available from : http://dblp.uni-trier.de/db/conf/ecai/ecai2004.html. 28, 98

[BK73] Coen Bron and Joep Kerbosch. Algorithm 457 : Finding all cliques of an undirected graph. *Commun. ACM*, 16(9) :575–577, September 1973. Available from : http://doi.acm.org/10.1145/362342.362367, doi:10.1145/362342.362367. 82, 85

[BKNW09] Christian Bessiere, George Katsirelos, Nina Narodytska, and Toby Walsh. Circuit complexity and decompositions of global constraints. In Craig Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 412–418, 2009. 103

[BKRW98] Adam L. Buchsbaum, Haim Kaplan, Anne Rogers, and Jeffery R. Westbrook. A new, simpler linear-time dominators algorithm. *ACM Trans. Program. Lang. Syst.*, 20(6) :1265–1296, November 1998. Available from : http://doi.acm.org/10.1145/295656.295663, doi:10.1145/295656.295663. 22, 44

[BL07] Nicolas Beldiceanu and Xavier Lorca. Necessary condition for path partitioning constraints. In Pascal Van Hentenryck and Laurence Wolsey, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems – CPAIOR 2007*, volume 4510 of *Lecture Notes in Computer Science*, pages 141–154. Springer Berlin Heidelberg, 2007. Available from : http://dx.doi.org/10.1007/978-3-540-72397-4_11, doi:10.1007/978-3-540-72397-4_11. 29

[BO79] Jon L. Bentley and Thomas A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, 28(9) :643–647, 1979. Available from : http://dx.doi.org/10.1109/TC.1979.1675432. 110

[Bou99] Eric Bourreau. *TRAITEMENT DE CONTRAINTES SUR LES GRAPHES EN PROGRAMMATION PAR CONTRAINTES*. PhD thesis, Université de Paris 13, 1999. 29, 31

[BPR06] Nicolas Beldiceanu, Thierry Petit, and Guillaume Rochart. Bounds of graph parameters for global constraints. *RAIRO - Operations Research*, 40 :327–353, 10 2006. Available from : http://www.rairo-ro.org/article_S0399055907000017, doi:10.1051/ro:2007001. 12, 29, 80

[BPW04] J. Christopher Beck, Patrick Prosser, and Richard J. Wallace. Failing first : An update. In Ramon López de Mántaras and Lorenza Saitta, editors, *ECAI*, pages 959–960. IOS Press, 2004. Available from : http://dblp.uni-trier.de/db/conf/ecai/ecai2004.html. 64

[BPW05] J.Christopher Beck, Patrick Prosser, and RichardJ. Wallace. Trying again to fail-first. In BoiV. Faltings, Adrian Petcu, François Fages, and Francesca Rossi, editors, *Recent Advances in Constraints*, volume 3419 of *Lecture Notes in Computer Science*, pages 41–55. Springer Berlin Heidelberg, 2005. Available from : http://dx.doi.org/10.1007/11402763_4, doi:10.1007/11402763_4. 64

[BS11] Nicolas Beldiceanu and Helmut Simonis. A constraint seeker : Finding and ranking global constraints from examples. In Jimmy Lee, editor, *Principles and*

Practice of Constraint Programming – CP 2011, volume 6876 of *Lecture Notes in Computer Science*, pages 12–26. Springer Berlin Heidelberg, 2011. Available from : http://dx.doi.org/10.1007/978-3-642-23786-7_4, doi:10.1007/978-3-642-23786-7_4. 98

[BVH03] Christian Bessière and Pascal Van Hentenryck. To be or not to be ... a global constraint. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming – CP 2003*, volume 2833 of *Lecture Notes in Computer Science*, pages 789–794. Springer Berlin Heidelberg, 2003. Available from : http://dx.doi.org/10.1007/978-3-540-45193-8_54, doi:10.1007/978-3-540-45193-8_54. 26

[BVHR+12] Pascal Benchimol, Willem-Jan Van Hoeve, Jean-Charles Régin, Louis-Martin Rousseau, and Michel Rueher. Improved filtering for weighted circuit constraints. *Constraints*, 17(3) :205–233, 2012. Available from : http://dx.doi.org/10.1007/s10601-012-9119-x, doi:10.1007/s10601-012-9119-x. 8, 13, 26, 56, 57, 58, 62, 63, 66, 71, 72, 73

[Cam14] Hadrien Cambazard. Contraintes np-difficiles avec des coûts : exemples d'applications et de filtrage, 2014. Exposé invité. 62

[CB04] Hadrien Cambazard and Eric Bourreau. Conception d'une contrainte globale de chemin. In *Actes des 10e Journées nationales sur la résolution pratique de problèmes NP-complets – JNPC'04*, pages 107–121, Angers, France, France, 2004. Available from : http://hal.archives-ouvertes.fr/hal-00448531. 40, 45

[CJL09] Gilles Chabert, Luc Jaulin, and Xavier Lorca. A constraint on the number of distinct vectors with application to localization. In IanP. Gent, editor, *Principles and Practice of Constraint Programming – CP 2009*, volume 5732 of *Lecture Notes in Computer Science*, pages 196–210. Springer Berlin Heidelberg, 2009. Available from : http://dx.doi.org/10.1007/978-3-642-04244-7_18, doi:10.1007/978-3-642-04244-7_18. 97

[CL97] Yves Caseau and François Laburthe. Solving Small TSPs with Constraints. In Lee Naish, editor, *Proceedings of the 14th International Conference on Logic Programming – ICLP 1997*, pages 316–330. MIT Press, 1997. 12, 44, 46, 50, 56, 58

[DB01] Romuald Debruyne and Christian Bessiére. Domain filtering consistencies. *J. Artif. Int. Res.*, 14(1) :205–230, May 2001. Available from : http://dl.acm.org/citation.cfm?id=1622394.1622402. 26

[DBGLT13] Jérémie Du Boisberranger, Danièle Gardy, Xavier Lorca, and Charlotte Truchet. When is it worthwhile to propagate a constraint ? a probabilistic analysis of alldifferent. In Markus E. Nebel and Wojciech Szpankowski, editors, *Proceedings of the Tenth Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, pages 80–90. SIAM, 2013. Available from : http://epubs.siam.org/doi/abs/10.1137/1.9781611973037.10, doi:10.1137/1.9781611973037.10. 111

[DDCG99] Marco Dorigo, Gianni Di Caro, and Luca M. Gambardella. Ant algorithms for discrete optimization. *Artif. Life*, 5(2) :137–172, April 1999. Available from : http://dx.doi.org/10.1162/106454699568728, doi:10.1162/106454699568728. 12

[DDD05] Gregoire Dooms, Yves Deville, and Pierre Dupont. Cp(graph) : Introducing a graph computation domain in constraint programming. In Peter van Beek, editor, *Principles and Practice of Constraint Programming – CP 2005*, volume 3709 of *Lecture Notes in Computer Science*, pages 211–225. Springer Berlin Heidelberg, 2005. Available from : http://dx.doi.org/10.1007/11564751_18, doi:10.1007/11564751_18. 12, 29, 34

[DDV13a] Thi-Bich-Hanh Dao, Khanh-Chuong Duong, and Christel Vrain. A declarative framework for constrained clustering. In Hendrik Blockeel, Kristian Kersting, Siegfried Nijssen, and Filip Železný, editors, *Machine Learning and Knowledge Discovery in Databases*, volume 8190 of *Lecture Notes in Computer Science*, pages 419–434. Springer Berlin Heidelberg, 2013. Available from : http://dx.doi.org/10.1007/978-3-642-40994-3_27, doi:10.1007/978-3-642-40994-3_27. 97

[DDV13b] Thi-Bich-Hanh Dao, Khanh-Chuong Duong, and Christel Vrain. Une approche en programmation par contraintes pour la classification non supervisée. In Christel Vrain, André Péninou, and Florence Sèdes, editors, *EGC*, volume RNTI-E-24 of *Revue des Nouvelles Technologies de l'Information*, pages 55–66. Hermann-Éditions, 2013. 97

[DKMS97] Denis Dowling, Mohan Krishnamoorthy, Harley Mackenzie, and David Sier. Staff rostering at a large international airport. *Annals of Operations Research*, 72(0) :125–147, 1997. doi:10.1023/A:1018992120116. 11, 77

[DKS+94] Matthijs C. Dijkstra, Leo G. Kroon, Marc Salomon, Jo. A. E. E. Van Nunen, and Luk N. Van Wassenhove. Planning the Size and Organization of KLM's Aircraft Maintenance Personnel. *Interfaces*, 24(6) :47–58, 1994. 11, 77

[Doo06] Grégoire Dooms. *The CP(Graph) Computation Domain in Constraint Programming*. PhD thesis, Faculté des Sciences Appliquées, Université Catholique de Louvain, 2006. 12, 29, 34

[DR09] Ian Davidson and S.S. Ravi. Using instance-level constraints in agglomerative hierarchical clustering : theoretical and empirical results. *Data Mining and Knowledge Discovery*, 18(2) :257–282, 2009. Available from : http://dx.doi.org/10.1007/s10618-008-0103-4, doi:10.1007/s10618-008-0103-4. 97

[EMJT14] Achref El Mouelhi, Philippe Jégou, and Cyril Terrioux. Different classes of graphs to represent microstructures for csps. In Madalina Croitoru, Sebastian Rudolph, Stefan Woltran, and Christophe Gonzales, editors, *Graph Structures for Knowledge Representation and Reasoning*, volume 8323 of *Lecture Notes in Computer Science*, pages 21–38. Springer International Publishing, 2014. Available from : http://dx.doi.org/10.1007/978-3-319-04534-4_3, doi:10.1007/978-3-319-04534-4_3. 12, 28

[FL11] Jean-Guillaume Fages and Xavier Lorca. Revisiting the tree constraint. In Jimmy Lee, editor, *Principles and Practice of Constraint Programming – CP 2011*, volume 6876 of *Lecture Notes in Computer Science*, pages 271–285. Springer Berlin Heidelberg, 2011. Available from : http://dx.doi.org/10.1007/978-3-642-23786-7_22, doi:10.1007/978-3-642-23786-7_22. 5, 13, 38, 42, 44, 47, 48, 49, 51, 52, 54, 119

[FL12] Jean-Guillaume Fages and Xavier Lorca. Improving the asymmetric tsp by considering graph structure. *CoRR*, abs/1206.3437, 2012. Available from : http://arxiv.org/abs/1206.3437. 45

[FL13] Jean-Guillaume Fages and Tanguy Lapègue. Filtering atmostnvalue with difference constraints : Application to the shift minimisation personnel task scheduling problem. In Christian Schulte, editor, *Principles and Practice of Constraint Programming – CP 2013*, volume 8124 of *Lecture Notes in Computer Science*, pages 63–79. Springer Berlin Heidelberg, 2013. Available from : http://dx.doi.org/10.1007/978-3-642-40627-0_8, doi:10.1007/978-3-642-40627-0_8. 13, 81

[FL14] Jean-Guillaume Fages and Tanguy Lapègue. Filtering atmostnvalue with difference constraints : Application to the shift minimisation personnel task scheduling problem. *Artificial Intelligence*, 212(0) :116 – 133, 2014. Available from : http://www.sciencedirect.com/science/article/pii/S0004370214000423, doi:http://dx.doi.org/10.1016/j.artint.2014.04.001. 13, 81, 120

[FLM99] Filippo Focacci, Andrea Lodi, and Michela Milano. Cost-based domain filtering. In Joxan Jaffar, editor, *Principles and Practice of Constraint Programming – CP'99*, volume 1713 of *Lecture Notes in Computer Science*, pages 189–203. Springer Berlin Heidelberg, 1999. Available from : http://dx.doi.org/10.1007/978-3-540-48085-3_14, doi:10.1007/978-3-540-48085-3_14. 28, 58

[FLM02] Filippo Focacci, Andrea Lodi, and Michela Milano. Embedding relaxations in global constraints for solving tsp and tsptw. *Annals of Mathematics and Artificial Intelligence*, 34(4) :291–311, April 2002. Available from : http://dx.doi.org/10.1023/A:1014492408220, doi:10.1023/A:1014492408220. 12, 26, 28, 56, 58

[FLP14] Jean-Guillaume Fages, Xavier Lorca, and Thierry Petit. Self-decomposable global constraints. In *Proceedings of the 2014 Conference on ECAI 2014 : 21st European Conference on Artificial Intelligence*. IOS Press, 2014. to appear. 13, 102, 120

[FLR14] Jean-Guillaume Fages, Xavier Lorca, and Louis-Martin Rousseau. The salesman and the tree : the importance of search in cp. In *Constraints*. Springer, 2014. accepted paper. 13, 119

[FMT87] Matteo Fischetti, Silvano Martello, and Paolo Toth. The fixed job schedule problem with spread-time constraints. *Oper. Res.*, 35(6) :849–858, November 1987. Available from : http://dx.doi.org/10.1287/opre.35.6.849, doi:10.1287/opre.35.6.849. 11, 77

[FPS+09] Pierre Flener, Justin Pearson, Meinolf Sellmann, Pascal Hentenryck, and Magnus Ågren. Dynamic structural symmetry breaking for constraint satisfaction problems. *Constraints*, 14(4) :506–538, 2009. Available from : http://dx.doi.org/10.1007/s10601-008-9059-7, doi:10.1007/s10601-008-9059-7. 98

[FS14] Kathryn Glenn Francis and Peter J. Stuckey. Explaining circuit propagation. *Constraints*, 19(1) :1–29, 2014. Available from : http://dx.doi.org/10.1007/s10601-013-9148-0, doi:10.1007/s10601-013-9148-0. 13, 31, 47, 48, 50, 51, 52, 53, 54

[Fut11] Panel of the Future of CP, 2011. Perugia, CP'11. 113

[GFM+14] Steven Gay, François Fages, Thierry Martinez, Sylvain Soliman, and Christine Solnon. On the subgraph epimorphism problem. *Discrete Appl. Math.*, 162 :214–228, January 2014. Available from : http://dx.doi.org/10.1016/j.dam.2013.08.008, doi:10.1016/j.dam.2013.08.008. 12

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979. 11, 17, 20, 81

[GLS00] Georg Gottlob, Nicola Leone, and Francesco Scarcello. A comparison of structural CSP decomposition methods. *Artificial Intelligence*, 124(2) :243 – 282, 2000. Available from : http://www.sciencedirect.com/science/article/pii/S0004370200000783, doi:http://dx.doi.org/10.1016/S0004-3702(00)00078-3. 12, 28

[GM95] Michel Gondran and Michel Minoux. *Graphes et algorithmes*. Collection de la Direction des études et recherches d'Électricité de France. Eyrolles, Paris, 1995. Available from : http://opac.inria.fr/record=b1077464. 11, 17

[GM12] Stefano Gualandi and Federico Malucelli. Exact solution of graph coloring problems via constraint programming and column generation. *INFORMS Journal on Computing*, 24(1) :81–100, 2012. Available from : http://pubsonline.informs.org/doi/abs/10.1287/ijoc.1100.0436, arXiv:http://pubsonline.informs.org/doi/pdf/10.1287/ijoc.1100.0436, doi:10.1287/ijoc.1100.0436. 12, 85

[Hab13] Michel Habib. On the power of graph searching, 2013. Invited lecture at CPAIOR 2012. 12

[HE79] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. In *Proceedings of the 6th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'79, pages 356–364, San Francisco, CA, USA, 1979. Morgan Kaufmann Publishers Inc. Available from : http://dl.acm.org/citation.cfm?id=1624861.1624942. 5, 12, 28, 53, 64

[Hel00] Keld Helsgaun. An effective implementation of the Lin-Kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1) :106–130, October 2000. Available from : http://ideas.repec.org/a/eee/ejores/v126y2000i1p106-130.html. 12, 64, 69

[HK71] Michael Held and Richard M. Karp. The traveling-salesman problem and minimum spanning trees : Part II. *Mathematical Programming*, 1(1) :6–25, 1971. Available from : http://dx.doi.org/10.1007/BF01584070, doi:10.1007/BF01584070. 58, 60, 61

[HLM13] Fabien Hermenier, Julia L. Lawall, and Gilles Muller. Btrplace : A flexible consolidation manager for highly available applications. *IEEE Trans. Dependable Sec. Comput.*, 10(5) :273–286, 2013. 11

[HR94] Magnús Halldórsson and Jaikumar Radhakrishnan. Greed is good : Approximating independent sets in sparse and bounded-degree graphs. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Theory of Computing*, STOC '94, pages 439–448, New York, NY, USA, 1994. ACM. Available from : http://doi.acm.org/10.1145/195058.195221, doi:10.1145/195058.195221. 82

[ILS10] Giuseppe F. Italiano, Luigi Laura, and Federico Santaroni. Finding strong bridges and strong articulation points in linear time. In Weili Wu and Ovidiu Daescu, editors, *Combinatorial Optimization and Applications*, volume 6508 of *Lecture Notes in Computer Science*, pages 157–169. Springer Berlin Heidelberg, 2010. Available from : http://dx.doi.org/10.1007/978-3-642-17458-2_14, doi:10.1007/978-3-642-17458-2_14. 22, 42

[Jau14] Luc Jaulin. Pure range-only slam with indistinguishable marks, 2014. submitted to Artificial Intelligence. 97

[JT03] Philippe Jégou and Cyril Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146(1) :43 – 75, 2003. Available from : http://www.sciencedirect.com/science/article/pii/S0004370202004009, doi:http://dx.doi.org/10.1016/S0004-3702(02)00400-9. 12, 28

[JT14] Philippe Jégou and Cyril Terrioux. Bag-connected tree-width : A new parameter for graph decomposition. In *International Symposium on Artificial Intelligence and Mathematics – ISAIM 2014*, 2014. 12, 28

[Jus03] Narendra Jussien. *The versatility of using explanations within constraint programming*. Hdr, Université de Nantes, Sep 2003. Available from : http://tel.archives-ouvertes.fr/tel-00293905. 12, 26

[KAS10] Madjid Khichane, Patrick Albert, and Christine Solnon. Strong combination of ant colony optimization with constraint programming optimization. In Andrea Lodi, Michela Milano, and Paolo Toth, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems – CPAIOR 2010*, volume 6140 of *Lecture Notes in Computer Science*, pages 232–245. Springer Berlin Heidelberg, 2010. Available from : http://dx.doi.org/10.1007/978-3-642-13520-0_26, doi:10.1007/978-3-642-13520-0_26. 12

[KEB12] M. Krishnamoorthy, A.T. Ernst, and D. Baatar. Algorithms for large scale shift minimisation personnel task scheduling problems. *European Journal of Operational Research*, 219(1) :34 – 48, 2012. Available from : http://www.sciencedirect.com/science/article/pii/S0377221711010435, doi:http://dx.doi.org/10.1016/j.ejor.2011.11.034. 8, 11, 78, 79, 88, 94, 95, 96

[KSVW94] Leo G. Kroon, Marc Salomon, and Luk N. Van Wassenhove. Exact and approximation algorithms for the tactical fixed interval scheduling problem. In *Operations Research Proceedings 1993*, volume 1993 of *Operations Research Proceedings*, pages 506–506. Springer Berlin Heidelberg, 1994. Available from : http://dx.doi.org/10.1007/978-3-642-78910-6_165, doi:10.1007/978-3-642-78910-6_165. 77

[KV12] Bernhard Korte and Jens Vygen. The traveling salesman problem. In *Combinatorial Optimization*, volume 21 of *Algorithms and Combinatorics*, pages 557–592. Springer Berlin Heidelberg, 2012. Available from : http://dx.doi.org/10.1007/978-3-642-24488-9_21, doi:10.1007/978-3-642-24488-9_21. 61

[Lau78] Jean-Louis Lauriere. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10(1) :29 – 127, 1978. Available from : http://www.sciencedirect.com/science/article/pii/0004370278900292, doi:http://dx.doi.org/10.1016/0004-3702(78)90029-2. 12, 25, 31

[LCB13] Arnaud Letort, Mats Carlsson, and Nicolas Beldiceanu. A synchronized sweep algorithm for the k-dimensional cumulative constraint. In Carla Gomes and Meinolf Sellmann, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems – CPAIOR 2013*, volume 7874 of *Lecture Notes in Computer Science*, pages 144–159. Springer Berlin Heidelberg, 2013. Available from : http://dx.doi.org/10.1007/978-3-642-38171-3_10, doi:10.1007/978-3-642-38171-3_10. 12, 98, 114, 117

[LFM11] Renato Leone, Paola Festa, and Emilia Marchitto. A bus driver scheduling problem : a new mathematical model and a grasp approximate solution. *Journal of Heuristics*, 17(4) :441–466, 2011. Available from : http://dx.doi.org/10.1007/s10732-010-9141-3, doi:10.1007/s10732-010-9141-3. 77

[LFPBM13] Tanguy Lapègue, Jean-Guillaume Fages, Damien Prot, and Odile Bellenguez-Morineau. Personnel Task Scheduling Problem Library, 2013. Available from : https://sites.google.com/site/ptsplib/smptsp/home. 88

[Lor11] Xavier Lorca. *Tree-based Graph Partitioning Constraint*. ISTE/Wiley, 2011. Available from : http://iste.co.uk/index.php?f=x&ACTION=View&id=418. 34, 38, 42

[LPPRS02] Claude Le Pape, Laurent Perron, Jean-Charles Régin, and Paul Shaw. Robust and parallel solving of a network design problem. In Pascal Van Hentenryck, editor, *Principles and Practice of Constraint Programming – CP 2002*, vo-

lume 2470 of *Lecture Notes in Computer Science*, pages 633–648. Springer Berlin Heidelberg, 2002. Available from : http://dx.doi.org/10.1007/3-540-46135-3_42, doi:10.1007/3-540-46135-3_42. 12, 34

[LSSL13] Vianney Le clément de saint-Marcq, Pierre Schaus, Christine Solnon, and Christophe Lecoutre. Sparse-Sets for Domain Implementation. In *CP workshop on Techniques foR Implementing Constraint programming Systems (TRICS)*, pages 1–10, September 2013. Available from : http://liris.cnrs.fr/publis/?id=6281. 49

[LSTV06] Christophe Lecoutre, Lakhdar Sais, Sébastien Tabary, and Vincent Vidal. Last conflict based reasoning. In *Proceedings of the 2006 Conference on ECAI 2006 : 17th European Conference on Artificial Intelligence August 29 – September 1, 2006, Riva Del Garda, Italy*, pages 133–137, Amsterdam, The Netherlands, The Netherlands, 2006. IOS Press. Available from : http://dl.acm.org/citation.cfm?id=1567016.1567050. 5, 28, 53, 63, 64

[LSTV09] Christophe Lecoutre, Lakhdar Saïs, Sébastien Tabary, and Vincent Vidal. Reasoning from last conflict(s) in constraint programming. *Artificial Intelligence*, 173(18) :1592 – 1614, 2009. Available from : http://www.sciencedirect.com/science/article/pii/S0004370209001040, doi:http://dx.doi.org/10.1016/j.artint.2009.09.002. 63, 120

[LY14] Shih-Wei Lin and Kuo-Ching Ying. Minimizing shifts for personnel task scheduling problems : A three-phase algorithm. *European Journal of Operational Research*, 237(1) :323 – 334, 2014. Available from : http://www.sciencedirect.com/science/article/pii/S0377221714000563, doi:http://dx.doi.org/10.1016/j.ejor.2014.01.035. 94, 95

[Mah09] Michael J. Maher. Open contractible global constraints. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, IJCAI'09, pages 578–583, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc. Available from : http://dl.acm.org/citation.cfm?id=1661445.1661537. 102, 103

[MDL14] Jean-Baptiste Mairy, Yves Deville, and Christophe Lecoutre. Domain k-wise consistency made as simple as generalized arc consistency. In Helmut Simonis, editor, *Integration of AI and OR Techniques in Constraint Programming – CPAIOR 2014*, volume 8451 of *Lecture Notes in Computer Science*, pages 235–250. Springer International Publishing, 2014. Available from : http://dx.doi.org/10.1007/978-3-319-07046-9_17, doi:10.1007/978-3-319-07046-9_17. 26, 98

[MFMS14] Thierry Martinez, François Fages, Philippe Morignot, and Sylvain Soliman. Search as constraint satisfaction, 2014. submitted to CP. 28, 98

[MH12] Laurent Michel and Pascal Hentenryck. Activity-based search for black-box constraint programming solvers. In Nicolas Beldiceanu, Narendra Jussien, and Éric Pinson, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems – CPAIOR 2012*, volume

7298 of *Lecture Notes in Computer Science*, pages 228–243. Springer Berlin Heidelberg, 2012. Available from : http://dx.doi.org/10.1007/978-3-642-29828-8_15, doi:10.1007/978-3-642-29828-8_15. 5, 28, 53, 98

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff : Engineering an efficient sat solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM. Available from : http://doi.acm.org/10.1145/378239.379017, doi:10.1145/378239.379017. 5, 53

[Mon74] Ugo Montanari. Networks of constraints : Fundamental properties and applications to picture processing. *Information Sciences*, 7(0) :95 – 132, 1974. Available from : http://www.sciencedirect.com/science/article/pii/0020025574900085, doi:http://dx.doi.org/10.1016/0020-0255(74)90008-5. 12, 25

[NIC] NICTA. Minizinc 1.6 : Global constraints. Available from : http://www.minizinc.org/downloads/doc-1.6/mzn-globals.html. 31

[NS11] SambaNdojh Ndiaye and Christine Solnon. Cp models for maximum common subgraph problems. In Jimmy Lee, editor, *Principles and Practice of Constraint Programming – CP 2011*, volume 6876 of *Lecture Notes in Computer Science*, pages 637–644. Springer Berlin Heidelberg, 2011. Available from : http://dx.doi.org/10.1007/978-3-642-23786-7_48, doi:10.1007/978-3-642-23786-7_48. 12

[Per11] Laurent Perron. Operations research and constraint programming at google. In Jimmy Lee, editor, *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming – CP 2011*, volume 6876 of *Lecture Notes in Computer Science*, pages 2–2. Springer Berlin Heidelberg, 2011. Available from : http://dx.doi.org/10.1007/978-3-642-23786-7_2, doi:10.1007/978-3-642-23786-7_2. 111

[Per13] Laurent Perron, 2013. Remark during Arnaud Letort's PhD defense. 117

[PLDJ14] Charles Prud'homme, Xavier Lorca, Rémi Douence, and Narendra Jussien. Propagation engine prototyping with a domain specific language. *Constraints*, 19(1) :57–76, 2014. Available from : http://dx.doi.org/10.1007/s10601-013-9151-5, doi:10.1007/s10601-013-9151-5. 26

[PR99] François Pachet and Pierre Roy. Automatic generation of music programs. In Joxan Jaffar, editor, *Principles and Practice of Constraint Programming – CP 1999*, volume 1713 of *Lecture Notes in Computer Science*, pages 331–345. Springer Berlin Heidelberg, 1999. Available from : http://dx.doi.org/10.1007/978-3-540-48085-3_24, doi:10.1007/978-3-540-48085-3_24. 82

[PRB01] Thierry Petit, Jean-Charles Régin, and Christian Bessière. Specific filtering algorithms for over-constrained problems. In Toby Walsh, editor, *Principles and Practice of Constraint Programming – CP 2001*, volume 2239 of *Lecture*

Notes in Computer Science, pages 451–463. Springer Berlin Heidelberg, 2001. Available from : http://dx.doi.org/10.1007/3-540-45578-7_31, doi:10.1007/3-540-45578-7_31. 28, 82

[Que06] Luis Quesada. *Solving Constrained Graph Problems using Reachability Constraints based on Transitive Closure and Dominators*. PhD thesis, Faculté des Sciences Appliquées, Université Catholique de Louvain, 2006. 29, 34

[QVRDC06] Luis Quesada, Peter Van Roy, Yves Deville, and Raphaël Collet. Using dominators for solving constrained path problems. In *Proceedings of the 8th International Conference on Practical Aspects of Declarative Languages*, PADL'06, pages 73–87, Berlin, Heidelberg, 2006. Springer-Verlag. Available from : http://dx.doi.org/10.1007/11603023_6, doi:10.1007/11603023_6. 12, 29, 34, 45

[Ref04] Philippe Refalo. Impact-based search strategies for constraint programming. In Mark Wallace, editor, *Principles and Practice of Constraint Programming – CP 2004*, volume 3258 of *Lecture Notes in Computer Science*, pages 557–571. Springer Berlin Heidelberg, 2004. Available from : http://dx.doi.org/10.1007/978-3-540-30201-8_41, doi:10.1007/978-3-540-30201-8_41. 28, 98

[Rég94] Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, volume 1 of *AAAI '94*, pages 362–367, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence. Available from : http://dl.acm.org/citation.cfm?id=199288.178024. 12, 26, 27, 28, 44, 79

[Rég96] Jean-Charles Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, volume 1 of *AAAI'96*, pages 209–215. AAAI Press, 1996. Available from : http://dl.acm.org/citation.cfm?id=1892875.1892906. 12, 28, 107

[Rég02] Jean-Charles Régin. Cost-based arc consistency for global cardinality constraints. *Constraints*, 7(3-4) :387–405, 2002. Available from : http://dx.doi.org/10.1023/A:1020506526052, doi:10.1023/A:1020506526052. 28

[Rég03] Jean-Charles Régin. Using constraint programming to solve the maximum clique problem. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming – CP 2003*, volume 2833 of *Lecture Notes in Computer Science*, pages 634–648. Springer Berlin Heidelberg, 2003. Available from : http://dx.doi.org/10.1007/978-3-540-45193-8_43, doi:10.1007/978-3-540-45193-8_43. 12

[Rég04] Jean-Charles Régin. Modeling problems in constraint programming, 2004. Tutorial. 34

[Rég08] Jean-Charles Régin. Simpler and incremental consistency checking and arc consistency filtering algorithms for the weighted spanning tree constraint. In Laurent Perron and MichaelA. Trick, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*,

volume 5015 of *Lecture Notes in Computer Science*, pages 233–247. Springer Berlin Heidelberg, 2008. Available from : http://dx.doi.org/10.1007/978-3-540-68155-7_19, doi:10.1007/978-3-540-68155-7_19. 26, 59, 61

[Rég11] Jean-Charles Régin. Global constraints : A survey. In Pascal van Hentenryck and Michela Milano, editors, *Hybrid Optimization*, volume 45 of *Springer Optimization and Its Applications*, pages 63–134. Springer New York, 2011. Available from : http://dx.doi.org/10.1007/978-1-4419-1644-0_3, doi:10.1007/978-1-4419-1644-0_3. 12, 26

[Rég13] Jean-Charles Régin. Simple solutions for complex problems, 2013. ACP Research Excellence Award, CP 2013. 62

[RGJM07] Dirk Richter, Boris Goldengorin, Gerold Jäger, and Paul Molitor. Improving the efficiency of helsgaun's lin-kernighan heuristic for the symmetric tsp. In *Proceedings of the 4th Conference on Combinatorial and Algorithmic Aspects of Networking*, CAAN'07, pages 99–111, Berlin, Heidelberg, 2007. Springer-Verlag. Available from : http://dl.acm.org/citation.cfm?id=1778487.1778499. 12, 57, 64, 73

[RRM13] Jean-Charles Régin, Mohamed Rezgui, and Arnaud Malapert. Embarrassingly parallel search. In Christian Schulte, editor, *Principles and Practice of Constraint Programming*, volume 8124 of *Lecture Notes in Computer Science*, pages 596–610. Springer Berlin Heidelberg, 2013. Available from : http://dx.doi.org/10.1007/978-3-642-40627-0_45, doi:10.1007/978-3-642-40627-0_45. 28

[RRRVH10] Jean-Charles Régin, Louis-Martin Rousseau, Michel Rueher, and Willem-Jan Van Hoeve. The weighted spanning tree constraint revisited. In Andrea Lodi, Michela Milano, and Paolo Toth, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 6140 of *Lecture Notes in Computer Science*, pages 287–291. Springer Berlin Heidelberg, 2010. Available from : http://dx.doi.org/10.1007/978-3-642-13520-0_31, doi:10.1007/978-3-642-13520-0_31. 60, 62, 73

[RvBW06] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006. 12, 25

[Sel04] Meinolf Sellmann. Theoretical foundations of cp-based lagrangian relaxation. In Mark Wallace, editor, *Principles and Practice of Constraint Programming – CP 2004*, volume 3258 of *Lecture Notes in Computer Science*, pages 634–647. Springer Berlin Heidelberg, 2004. Available from : http://dx.doi.org/10.1007/978-3-540-30201-8_46, doi:10.1007/978-3-540-30201-8_46. 12, 62

[SG98] Barbara M. Smith and Stuart A. Grant. Trying harder to fail first. In *ECAI*, pages 249–253, 1998. Available from : http://dblp.uni-trier.de/db/conf/ecai/ecai98.html. 64

[Sha98] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In Michael Maher and Jean-Francois Puget, editors, *Principles and Practice of Constraint Programming – CP 1998*, volume 1520 of *Lecture Notes in Computer Science*, pages 417–431. Springer Berlin Heidelberg, 1998. Available from : http://dx.doi.org/10.1007/3-540-49481-2_30, doi:10.1007/3-540-49481-2_30. 11, 12, 28, 73, 113, 117

[Sol10] Christine Solnon. Alldifferent-based filtering for subgraph isomorphism. *Artificial Intelligence*, 174(12-13) :850–864, August 2010. Available from : http://dx.doi.org/10.1016/j.artint.2010.05.002, doi:10.1016/j.artint.2010.05.002. 12

[ST06] Christian Schulte and Guido Tack. Views and iterators for generic constraint implementations. In *Proceedings of the 2005 Joint ERCIM/CoLogNET International Conference on Constraint Solving and Constraint Logic Programming – CSCLP 2005*, pages 118–132, Berlin, Heidelberg, 2006. Springer-Verlag. Available from : http://dx.doi.org/10.1007/11754602_9, doi:10.1007/11754602_9. 34

[ST09] Christian Schulte and Guido Tack. Weakly monotonic propagators. In IanP. Gent, editor, *Principles and Practice of Constraint Programming – CP 2009*, volume 5732 of *Lecture Notes in Computer Science*, pages 723–730. Springer Berlin Heidelberg, 2009. Available from : http://dx.doi.org/10.1007/978-3-642-04244-7_56, doi:10.1007/978-3-642-04244-7_56. 47, 51, 52, 83, 103

[STW+13] Tom Schrijvers, Guido Tack, Pieter Wuille, Horst Samulowitz, and PeterJ. Stuckey. Search combinators. *Constraints*, 18(2) :269–305, 2013. Available from : http://dx.doi.org/10.1007/s10601-012-9137-8, doi:10.1007/s10601-012-9137-8. 28

[SWMB14] Pieter Smet, Tony Wauters, Mihail Mihaylov, and Greet Vanden Berghe. The shift minimisation personnel task scheduling problem : A new hybrid approach and computational insights. *Omega*, 46(0) :64 – 73, 2014. Available from : http://www.sciencedirect.com/science/article/pii/S0305048314000176, doi:http://dx.doi.org/10.1016/j.omega.2014.02.003. 88, 94, 95, 97

[Tar72] Robert E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2) :146–160, 1972. Available from : http://epubs.siam.org/doi/abs/10.1137/0201010, arXiv:http://epubs.siam.org/doi/pdf/10.1137/0201010, doi:10.1137/0201010. 11, 17, 21, 22

[VH89] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge, MA, USA, 1989. 12, 25

[VHDT92] Pascal Van Hentenryck, Yves Deville, and Choh-Man Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57(2–3) :291 – 321, 1992. Available from : http://www.sciencedirect.com/science/article/pii/000437029290020X, doi:http://dx.doi.org/10.1016/0004-3702(92)90020-X. 103

[VHSD95] Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(fd). In Andreas Podelski, editor, *Constraint Programming : Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*, pages 293–316. Springer Berlin Heidelberg, 1995. Available from : http://dx.doi.org/10.1007/3-540-59155-9_15, doi:10.1007/3-540-59155-9_15. 26

[WQD14] Xiang Wang, Buyue Qian, and Ian Davidson. On constrained spectral clustering and its applications. *Data Mining and Knowledge Discovery*, 28(1) :1–30, 2014. Available from : http://dx.doi.org/10.1007/s10618-012-0291-9, doi:10.1007/s10618-012-0291-9. 97

l'université nantes angers le mans
PÔLE DE RECHERCHE ET D'ENSEIGNEMENT SUPÉRIEUR

MINES Nantes
Institut Mines-Télécom

Thèse de Doctorat

Jean-Guillaume FAGES

Exploitation de structures de graphe en programmation par contraintes

On the use of graphs within constraint-programming

Résumé
De nombreuses applications informatiques nécessitent de résoudre des problèmes de décision qui sont difficiles d'un point de vue mathématique. La programmation par contraintes permet de modéliser et résoudre certains de ces problèmes, parfois définis sur des graphes. Au delà des difficultés intrinsèques aux problèmes étudiés, la taille des instances à traiter contribue à la difficulté de la résolution. Cette thèse traite de l'utilisation des graphes en programmation par contraintes, dans le but d'en améliorer la capacité de passage à l'échelle.
Une première partie porte sur l'utilisation de contraintes pour résoudre des problèmes de graphes impliquant la recherche d'arbres, de chemins et de cycles Hamiltoniens. Ce sont des problèmes importants que l'on retrouve dans de nombreuses applications industrielles. Nous étudions à la fois le filtrage et les stratégies d'exploration de l'espace de recherche. Nous chercherons ensuite à nous extraire progressivement des problèmes classiquement définis sur les graphes pour exploiter ce concept sur des problèmes définis sur les entiers, voire les réels. Une seconde partie porte ainsi sur l'utilisation des graphes pour le filtrage de contraintes globales très répandues. Nous proposerons entre autres d'utiliser des graphes comme support pour décomposer dynamiquement des algorithmes de filtrage, de manière générique.
Le fil conducteur de ces travaux sera d'une part l'utilisation du concept de graphe à la base de chaque raisonnement et d'autre part, la volonté pratique d'augmenter la taille des problèmes pouvant être traités en programmation par contraintes.

Mots clés
Programmation par contraintes, théorie des graphes, recherche opérationnelle, algorithmes, intelligence artificielle.

Abstract
Many IT applications require to solve decision problems which are hard from a mathematical point of view. Constraint-programming enables to model and solve some of these problems. Among them, some are defined over graphs. Beyond the difficulty stemming from each of these problems, the size of the instance to solve increases the difficulty of the task. This PhD thesis is about the use of graphs within constraint-programming, in order to improve its scalability.
First, we study the use of constraint-programming to solve some graph problems involving the computation of trees and Hamiltonian paths and cycles. These problems are important and can be found in many industrial applications. Both filtering and search are investigated. Next, we move on problems which are no longer defined in terms of graph properties. We then study the use of graphs to propagate global constraints. In particular, we suggest a generic schema, relying on a graph structure, to dynamically decompose filtering algorithms.
The central theme in this work is the use of graph concepts at the origin of every reasoning and the practical will to increase the size of problems that can be addressed in constraint-programming.

Key Words
Constraint-Programming, Graph Theory, Operational Research, Algorithms, Artificial Intelligence.

L'UNIVERSITÉ NANTES ANGERS LE MANS