UNIVERSITÉ DE NANTES
FACULTÉ DES SCIENCES ET DES TECHNIQUES

ÉCOLE DOCTORALE
SCIENCE ET TECHNOLOGIES DE L'INFORMATION ET DES MATÉRIAUX

N° attribué par la bibliothèque

Année 2006

Résolution de problèmes combinatoires par des approches fondées sur la notion d'explication

THÈSE DE DOCTORAT

Discipline : Informatique

Spécialité : Programmation Par Contraintes

Présentée
et soutenue publiquement par

Hadrien CAMBAZARD

Le 15 novembre 2006, devant le jury ci-dessous

Président	Jean Bézivin (Professeur, Université de Nantes)
Rapporteurs	Francois Fages (Directeur de recherche, INRIA)
	Jean-Charles Régin (Directeur P.P.C, ILOG, HdR)
Examinateurs	Philippe Baptiste (Professeur, École polytechnique)
	Gérard Verfaillie (Maître de recherche, ONERA, CERT)

Directeur de thèse : Narendra Jussien (Maître assistant HdR, EMN)

ED : 366

RÉSOLUTION DE PROBLÈMES COMBINATOIRES PAR DES APPROCHES FONDÉES SUR LA NOTION D'EXPLICATION

Explanation based algorithms in Constraint Programming

Hadrien Cambazard

favet neptunus eunti

Université de Nantes

Remerciements

Cette thèse n'aurait pas vu le jour sans l'aide et le soutien de mon entourage. Je tiens en premier lieu à exprimer ma gratitude à Narendra Jussien, mon encadrant direct, pour la formation qu'il m'a donné, pour la confiance dont il m'a honoré en m'intégrant très tôt à ses activités de recherche et pour la possibilité qu'il m'a offerte de travailler en collaboration avec d'autres équipes. J'ai pu aussi m'instruire au contact de la communauté grâce à son soucis permanent de permettre aux doctorants de suivre les conférences du domaine. Enfin je le remercie pour toute l'aide qu'il m'a apportée avec une amitié généreuse et constante.

Les différents travaux conduits dans cette thèse sont aussi le résultats d'interactions et discussions avec de nombreuses personnes que je remercie ici : Pierre-Emmanuel Hladik, Anne-Marie Deplanche sur le sujet de l'ordonnancement temps-réel ; Thierry Benoist, Nicolas Beldiceanu, Didier Le Botlan et Yoann Padioleau sur le sujet du *Minimum Open Stack Problem* ; Guillaume Richaud et Barry O'Sullivan sur l'enregistrement de *nogoods* ;

Je souhaite vivement remercier toute l'équipe « contraintes »pour avoir su développer mon goût pour l'optimisation combinatoire et fait naître chez moi le désir de poursuivre une carrière académique dans cette discipline. Je remercie aussi sincèrement mes rapporteurs, Jean-Charles Régin et Francois Fages, qui ont accepté de relire cette thèse malgré leur emploi du temps terriblement chargé. Je remercie aussi tous les membres de mon jury Gérard Verfaillie, Philippe Baptiste et jean Bézivin.

Je n'oublie pas non plus tous les doctorants, et particulièrement Guillaume Rochart, pour son aide précieuse et tout de que j'ai appris à son contact.

Je remercie mes parents sans qui rien de tout cela ne serait arrivé. Je pense en particulier à mon père qui m'a insufflé par son exemple, toute sa vie durant, le désir d'être auteur.

Enfin je remercie Ludivine qui est restée à mes côtés et a énormément contribué à ce travail par son soutien.

Table des matières

Chapitre 1

Introduction

1.1 Contexte du travail

L'objet de cette thèse est de contribuer à la résolution de problèmes combinatoires. Mais à quoi reconnaît-on de tels problèmes ? L'analyse combinatoire est rattachée aux mathématiques et étudie les collections (souvent finies) d'objets qui respectent certaines propriétés. Dénombrer toutes ces collections ou exhiber la *meilleure* collection parmi toutes celles possibles, sont des questions combinatoires. Euler, par exemple, pose la question suivante en 1779 : en considérant 6 régiments de 6 officiers de rangs distincts, est-il possible de placer les 36 officiers dans un carré de 6 par 6 de telle sorte qu'il n'y ait pas deux officiers de même rang ou de même régiment dans chaque ligne et colonne ? Le problème est simple à exprimer (sa compréhension ne demande aucune connaissance scientifique particulière) comme le célèbre jeu du Sudoku, et sa difficulté vient du trop grand nombre de *combinaisons* à évaluer pour pouvoir déterminer la bonne. Il s'agit de limiter autant que possible cette *explosion combinatoire*, mais même les problèmes les plus simples sont au-delà des capacités humaines. L'ordinateur est donc un outil indispensable de la résolution, au moins depuis 1953 si on en croit Kuhn[1] [Sch05], l'inventeur de la méthode hongroise pour le problème d'affectation. Les jeux et puzzles logiques sont affaires de mathématiciens comme Gauss et Euler, l'optimisation combinatoire a des vues bien plus larges et s'appuie sur les développements des technologies informatiques. Si son histoire proprement dite semble remonter aux années 50, on peut retrouver son origine très loin dans le passé car elle prend racine dans des problèmes qui ne sont pas la seule préocupation du mathématicien. Ces problèmes *historiques*, tels que le problème d'affectation, de flot maximal, de voyageur de commerce, ou d'arbres couvrants de poids minimal, ont été abordés et résolus de manière indépendante et différente dans l'histoire car ils sont au cœur de questions pratiques que nos sociétés rencontrent. Transporter des biens, établir des réseaux, planifier des activités sont autant de problématiques qui exigent aujourd'hui de *bonnes* réponses car elles se posent à des échelles mondiales. La gestion et l'amélioration de ces situations complexes et réelles sont le propre de la Recherche Opérationnelle. La Programmation Par Contraintes (PPC) s'inscrit dans ce cadre. C'est un paradigme de

[1] *I solved several 12 by 12 assignment problems by hand. Each of them took me under two hours to solve ... This must have been one of the last times when pencil and paper could beat the largest and fastest electronic computer in the world.*

résolution sur lequel des outils génériques, des solveurs, ont été conçus pour permettre à des non-spécialistes d'aborder ces problèmes combinatoires. Aujourd'hui, la volonté de construire des outils informatiques capables de résoudre de manière efficace de larges classes de problèmes est de plus en plus forte. Ces outils s'appuient toujours davantage sur des interfaces déclaratives avec lesquelles un utilisateur décrit son problème, sans indiquer la manière de le résoudre. La programmation par contraintes et la programmation linéaire entrent dans ces catégories et touchent ici au domaine de l'intelligence artificielle selon les mots d'Eugene Freuder :

> *Constraint Programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming : the user states the problem, the computer solves it.*

Si l'efficacité, la souplesse et le champ d'application de ces outils n'ont cessé de croître ces dernières années, ils sont encore loin de répondre parfaitement aux besoins des industriels en matière d'optimisation. Ainsi, la prise en compte d'éléments sur-contraints (absence de solution) ou dynamiques (problème en interaction avec leurs environnements) reste délicate ou impossible dans les outils commerciaux. Ce sont pourtant des caractéristiques fréquentes des problèmes réels. Le concept **d'explications** qui constitue l'objet principal de cette thèse a été introduit pour pallier ces lacunes. Les **explications** constituent une information détaillée sur le processus de résolution et permettent ainsi de le modifier et d'interagir avec lui dynamiquement en ajoutant ou retirant des contraintes par exemple. Néanmoins, le maintien d'une telle information ralentit la résolution, contrebalançant l'intérêt des explications pour les modifications dynamiques ou l'absence de solutions. Nous pensons cependant que les explications sont encore sous-exploitées pour la résolution elle-même et qu'elles offrent de nombreuses perspectives dans ce domaine.

1.2 Problématique

Maintenir des explications est donc particulièrement utile pour le traitement des systèmes sur-contraints et dynamiques de contraintes mais aussi coûteux. Peut-on rentabiliser leurs calculs en les exploitant également pour accélérer la résolution elle-même ? Par le passé, les explications ont constitué le cœur de méthodes de résolution originales connues sous le nom de *backtrack intelligent* dont l'objectif est de limiter le *thrashing*, c'est-à-dire les calculs redondants dont sont parfois victimes les approches de résolution classiques basées sur le *backtrack*. Les explications ont donc déjà été mises en œuvre pour améliorer la résolution. Néanmoins, ces approches n'ont pas, à ce jour suffisamment démontré leur intérêt pour intégrer des solveurs commerciaux et des applications industrielles. Le meilleur algorithme reconnu aujourd'hui pour la résolution des problèmes de satisfaction est l'algorithme MAC (*Maintaining Arc-Consistency*) que nous présenterons au chapitre suivant et qui n'exploite pas le *backtrack intelligent*. Ce n'est pourtant pas le cas dans d'autres domaines très proches comme en Satisfiabilité (SAT) où les explications jouent un rôle crucial dans les solveurs. D'une manière différente, les solveurs linéaires exploitent en permanence des informations issues de la résolution d'un problème, dit *dual*, qui constitue une autre manière d'appréhender le problème d'origine dit *primal*. Les explications sont très proches de ces informa-

tions duales utilisées par les solveurs linéaires. Nous revenons donc dans cette thèse sur la nature de l'information véhiculée par les explications et son rôle au cours de la résolution.

L'objectif de cette thèse est donc d'examiner l'intérêt des explications pour améliorer les algorithmes actuels de résolution des problèmes de satisfaction de contraintes (par ex. MAC) ou en concevoir de nouveaux. Il s'agit aussi de montrer comment les explications peuvent être mise en œuvre efficacement sur des applications réelles.

1.3 Contributions et organisation de la thèse

Le contexte de ce travail est introduit dans les chapitres 2 à 5. On y aborde les lacunes des méthodes complètes de résolution utilisées dans les solveurs et particulièrement le *thrashing*. Les explications sont introduites pour corriger ce défaut et les techniques d'exploration intelligente de l'espace de recherche sont passées en revue. On comprend notamment qu'elles concernent des problèmes structurés. Cette notion est approfondie avant de présenter nos axes de recherche dans ce contexte.

Nous montrons ensuite dans cette thèse que les explications ont un rôle important à jouer au niveau de la résolution pour :

1. identifier des structures du problème et guider la recherche. Nous nous intéressons ici à l'information contenue dans le réseau d'explications pour caractériser les phénomènes de *thrashing* sans passer par le backtrack intelligent (chapitre 6) ;
2. filtrer l'espace de recherche futur en revenant sur les techniques d'enregistrements de *nogoods*. Nous évaluons notamment la possibilité de stocker et propager efficacement de grands ensembles de nogoods par des automates (chapitre 7) ;
3. proposer un autre schéma de résolution fondé sur une décomposition du problème définie par l'utilisateur au moment de la phase de modélisation. Cette décomposition qui peut-être vue comme une forme limitée d'enregistrement de *nogoods* met clairement en évidence que les explications constituent une information duale. Les explications offrent donc de nombreuses perspectives dans des schémas de résolution inspirés des techniques de Recherche Opérationnelle et fondés sur la dualité linéaire (chapitre 8).

Les chapitres 6,7 et 8 de la partie II intitulée *Des explications au cœur des solveurs* abordent les trois points précédents dans un cadre générique. L'objectif de cette thèse est aussi de montrer que les explications peuvent constituer une technique de choix pour aborder des problèmes difficiles. C'est l'objet de la partie III avec les chapitres 9 et 10 :

1. le chapitre 9 aborde la résolution d'un problème de placement de tâches temps réel dans laquelle une stratégie originale de décomposition du problème fondée sur le calcul d'explications et illustrant le chapitre 8 est mise en œuvre. Par ailleurs, des heuristiques de recherche s'appuyant sur le maintien des explications et à l'origine des idées introduites au chapitre 6 sont également présentées ;
2. la deuxième application, au chapitre 10, concerne le problème de MOSP pour *Minimum Open Stacks Problem*. Nous mettons en œuvre ici les idées du chapitre

7 sur l'enregistrement de *nogoods* avec l'utilisation d'un automate pour la compilation de *nogoods* et le calcul de *nogoods* généralisés qui illustrent le mécanisme central des explications *i.e* la généralisation des échecs.

Ces deux applications représentent à nos yeux le cœur de notre travail et sont à l'origine de toutes les idées présentées dans un cadre plus générique de la partie II.

Première partie

Contexte

Chapitre 2

Vertigineuse combinatoire

Un problème combinatoire s'appréhende souvent à travers un modèle mathématique qui s'appuie sur les éléments suivants :

- un ensemble V de variables ;
- un ensemble D de domaines correspondant à chaque variable et représentant les valeurs possibles des variables ;
- un ensemble C de contraintes exprimant des relations logiques que doivent vérifier les valeurs prises par les variables ;
- éventuellement, un (ou plusieurs) critère(s) à optimiser (fonction objectif).

La grande variété des problèmes, de leurs complexités et des approches de résolution est reliée à la nature de ces quatre éléments. Ainsi, le cas des contraintes linéaires et des domaines continus (programmation linéaire) est bien résolu aujourd'hui, même à des échelles surprenantes (centaines de milliers de variables-contraintes). Le passage à des domaines discrets ou des contraintes non linéaires (fonctions polynomiales, trigonométriques, relations logiques) est souvent beaucoup plus épineux. La nature de la fonction objectif (linéaire, convexe, etc.) est également déterminante pour la complexité du problème. Par ailleurs, la réalité dépasse souvent le modèle mathématique et comporte des éléments sur-contraints (contraintes souples), inconnus (variables stochastiques), ou difficiles à quantifier pour le décideur.

Parmi la grande diversité des techniques de résolution, différentes classes se distinguent sur la manière dont l'espace de recherche est exploré et sur les traitements effectués à chaque étape de l'exploration : les méthodes gloutonnes, locales, complètes et incomplètes, exactes et approchées... Les méthodes gloutonnes ne remettent pas en cause les décisions prises et construisent progressivement une solution en utilisant une heuristique. Elles font partie des techniques les plus simples et les plus pragmatiques. Les méthodes complètes font référence à des techniques « d'exploration systématique », notamment capables de prouver l'inconsistance ou l'optimalité d'un problème. Elles s'appuient souvent (mais pas uniquement, comme nous le verrons en partie 3.3.1) sur une recherche arborescente des combinaisons possibles. Ces techniques offrent donc la garantie de l'optimalité par une exploration exhaustive mais souvent en partie implicite de l'espace de recherche. De même, les algorithmes d'approximation garantissent quant à eux de fournir une réponse à une certaine connue de l'optimum. Une autre classe très importante en pratique est celle des recherches locales. À l'inverse des méthodes complètes qui construisent généralement une solution en étendant progressivement des

affectations partielles des variables, ces dernières procèdent en explorant successivement des affectations complètes. Elles partent d'une solution initiale et le passage d'une affectation complète à une autre se fait par l'application de mouvements ou de transformations qui visent à améliorer (au sens de la fonction objectif) la solution courante. Elles souffrent de leur vision locale du problème qui peut les amener à rester bloquer sur un optimum local. Néanmoins, des techniques génériques (par ex. les méta-heuristiques) existent pour diminuer ces risques et pour les guider vers l'optimum global. Dans tous les cas, elles sont souvent le dernier recours pour résoudre les problèmes de très grande taille. De nombreux schémas d'hybridation ont été imaginés depuis, offrant par exemple la possibilité aux recherches locales d'évoluer dans l'espace des affectations partielles ([JL02a]).

Nous nous intéressons dans cette thèse aux méthodes de résolution complètes dans le cadre de domaines discrets sans faire d'hypothèses sur la nature des contraintes et pour des problèmes de satisfaction (obtenir une solution réalisable) comme des problèmes d'optimisation.

2.1 Les méthodes de résolution complètes

Les principes et les mécanismes au cœur de deux techniques complètes de résolution sont présentés ici : la programmation linéaire et la programmation par contraintes. L'objectif n'est pas l'exhaustivité mais plutôt de mettre en lumière ce que sont à nos yeux les concepts fondamentaux et d'introduire les notions plus spécifiques qui seront reprises en détail dans cette thèse. Ces deux techniques de *programmation* sont issues d'horizons très différents et le terme de *programmation* reste une source de confusion [LP01] pour leur compréhension. Pour la première, « programmation »fait référence à la conception de *programmes* (par ex. programme de la journée) ou de plans d'actions pour les forces militaires du *United States Defense Department* et voit le jour en tant que discipline à part entière avec George Dantzig en 1947 et son article intitulé *Programming in a linear structure* [Dan48]. La seconde est plus récente (autour des années 80) et se réfère à une technique de programmation informatique telle que la programmation fonctionnelle ou par objets et nous vient de l'intelligence artificielle et des développements des langages de programmation. Aujourd'hui, les deux se rejoignent chaque jour davantage car elles apportent des outils puissants et souvent complémentaires.

2.1.1 Programmation linéaire

Un programme linéaire (PL) est défini par :
- le vecteur colonne x de ses variables continues $x = [x_1, \ldots, x_n]$;
- la matrice des contraintes A et le vecteur colonne des constantes $b = [b_1, \ldots, b_m]$ de sorte que chaque ligne i corresponde à une contrainte linéaire $\sum_{j=1}^{n} a_{ij}x_j \leq b_i$;
- la fonction objectif $z = \sum_{j=1}^{n} c_j x_j$ définie par le vecteur $c = [c_1, \ldots, c_n]$ des coûts.

La figure 2.1 donne une écriture matricielle d'un tel problème P.

D'un point de vue géométrique, l'ensemble des contraintes linéaires d'un PL définit un polyèdre convexe dans un espace à n dimensions qui correspond à l'espace des solutions. Un polyèdre vide fait dès lors référence à une absence de solution pour le problème tandis qu'un polyèdre non borné peut éventuellement mener à l'absence

P : Max $z = cx$
Tel que : $Ax \leq b$
$\forall i, x_i \geq 0$

FIG. 2.1: PL primal sous forme matricielle

d'optimum fini. Quand P admet des solutions optimales, une solution se trouve sur un point extrême de ce polyèdre (voir figure 2.2) puisque la fonction objectif est linéaire et le domaine des variables continu. La convexité de l'espace des solutions et la linéarité de la fonction objectif sont les deux propriétés essentielles exploitées par l'algorithme du simplexe proposé par Dantzig. Bien que cet algorithme soit de complexité exponentielle, son efficacité pratique lui a permis de s'imposer devant d'autres algorithmes polynômiaux imaginés par la suite [Kar84].

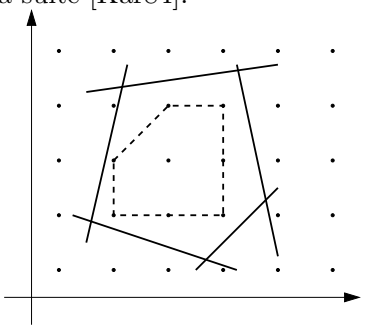


FIG. 2.2: exemple d'un polyèdre et de l'enveloppe convexe de ses points entiers

2.1.1.1 Le simplexe

La formulation algébrique du simplexe passe par l'introduction de variables dites d'écart de telle sorte que chaque contrainte $\sum_{j=1}^{n} a_{ij}x_j \leq b_i$ devienne $\sum_{j=1}^{n} a_{ij}x_j + e_i = b_i$ avec $e_i \geq 0$. Dans ce formalisme, chaque sommet du polyèdre est caractérisé par un sous-ensemble de m variables prises parmi les $n+m$ variables du problème complet (on inclut dans ce décompte les nouvelles variables d'écart e_i). En effet, dans ce système à m équations et $m + n$ variables, il suffit de fixer n variables à zéro pour posséder une solution sur les m restantes (après les avoir rendues linéairement indépendantes). Une telle solution est appelée une solution de base (toutes les variables hors-bases ont donc des valeurs nulles). Le simplexe propose de se déplacer (exactement à l'image d'une recherche locale) de solution de base en solution de base (d'un point extrême à un autre) en améliorant la fonction objectif à chaque mouvement. Le mouvement en question est une opération de pivot où une variable hors-base est choisie pour remplacer une variable actuellement en base. Dès qu'aucun mouvement améliorant n'est possible, l'optimum est atteint (du fait de la convexité de l'espace). Des questions se posent à différents niveaux :

– l'initialisation : comment choisir une solution de base initiale pour démarrer l'algorithme ? Le problème se pose dès que le point origine obtenu en utilisant les variables d'écarts ($\forall i, x_i = 0$) n'est pas une solution de base faisable. Des

techniques reposant sur le simplexe lui-même existent comme la méthode des 2 phases. On résout dans une première phase un problème auxiliaire qui possède le point origine comme solution de base initiale et dont la solution optimale constitue la solution de base recherchée pour le véritable problème. Celui-çi est alors résolu dans une deuxième phase.
- à la terminaison : comment repère-t-on un problème non borné ? Un problème non borné est caractérisé par un polyèdre ouvert. Le simplexe détecte cette situation quand il n'y a aucune variable candidate pour quitter la base. Aucune limite n'est donc imposé à la croissance de la variable entrante qui peut augmenter (et donc l'objectif également) sans remettre en cause la faisabilité de la solution de base.
- en cours de résolution : l'algorithme peut-il cycler sur un *plateau* du polyèdre ? Oui, cela peut parfois se produire en cas de dégénérescence. La dégénérescence désigne une situation ou au moins une variable en base prend la valeur zéro. La variable entrante choisie pour la remplacer est donc également limitée à zéro. La fonction objectif ne croît pas et le simplexe peut donc revenir sur une solution de base de même valeur objectif déjà visitée.

Ces éléments ont été ici introduits de manière informelle et on se référera à [Chv83] pour une présentation approfondie. Nous souhaitons simplement souligner ici comment la nature du domaine des variables (continuité), des contraintes (linéarité) et de la fonction objectif (monotonie dans chaque direction) sont à l'origine du simplexe et de la résolution efficace de cette classe de problème. Par ailleurs, certaines idées issues de la programmation linéaire ont des répercussions sur toute l'optimisation combinatoire. C'est le cas de la dualité.

2.1.1.2 La dualité

En s'interrogeant sur une borne supérieure de z, on peut être amené à formuler un nouveau problème linéaire, D, basé sur les contraintes de P. Imaginons qu'il existe une contrainte $\sum_{j=1}^{n} a_{ij}x_j \leq b_i$ telle que $\forall j, a_{ij} \geq c_j$. b_i constitue dès lors une borne supérieure pour z puisque $z = \sum_{j=1}^{n} c_j x_j \leq \sum_{j=1}^{n} a_{ij}x_j \leq b_i$.
Plus généralement, on peut essayer de chercher une combinaison linéaire des b_i (en associant un coefficient y_i à chaque b_i) assurant pour chaque coefficient c_j que :

$$\forall j, \qquad \sum_{i=1}^{m} a_{ij}y_i \geq c_j \tag{2.1}$$

et garantissant ainsi que $\sum_i b_i y_i$ soit une borne supérieure de z puisque :

$$z = \sum_{j=1}^{n} c_j x_j \leq \sum_{j=1}^{n}(\sum_{i=1}^{m} a_{ij}y_i)x_j = \sum_{i=1}^{m}(\sum_{j=1}^{n} a_{ij}x_j)y_i \leq \sum_{i=1}^{m} b_i y_i$$

La recherche de la meilleure borne supérieure nous conduit à formuler le programme linéaire **dual** D de la figure 2.3 (P porte aussi le nom de problème **primal** par opposition) dans lequel les équations 2.1 constituent les contraintes et où on cherche à minimiser $\sum_{i=1}^{m} b_i y_i$. On note y le vecteur colonne des variables duales $y = [y_1, \ldots, y_m]$.

Ces deux problèmes entretiennent des relations très fortes (voir [Chv97] pour la démonstration) :

D : Min $w = {}^t by$
Tel que : ${}^t Ay \geq c$
$y \geq 0$

FIG. 2.3: PL dual sous forme matricielle

Propriété 2.1.1 *Pour toutes solutions x et y respectivement d'un problème primal P et de son dual D alors $z = cx \leq w = {}^t by$. De plus, si P et D ont des solutions (polyèdre borné et non vide), ils ont des solutions optimales notées $z^* = cx^*$ et $w^* = {}^t by^*$ telles que $z^* = w^*$.*

Par ailleurs, les solutions optimales de P et D sont intimement liées à travers leurs variables d'écart. La seule solution optimale de l'un d'eux permet d'obtenir l'optimum de l'autre ce qui présente la dualité comme un certificat d'optimalité[1]. Le dual justifie donc de manière élégante l'optimalité, il fournit des interprétations économiques intéressantes (en indiquant par exemple le gain potentiel lié à la variation des paramètres b_i : on fait alors référence aux variables duales sous le terme de *shadow prices i.e.* la valeur marginale d'une ressource b_i) mais il est aussi utilisé implicitement à tout moment de la résolution pour guider la recherche dans le polyèdre en s'orientant dans les directions qui entraînent le meilleur gain (les variables duales portent alors le nom de coûts réduits ou *reduced costs*)[2]. Outre son intérêt pour le cadre continu, la dualité est très présente dans toute a littérature des approches complètes de la recherche opérationnelle et au cœur de techniques de décomposition et relaxation aussi variées que la décomposition de Benders [Ben62], la génération de colonnes [BJN+98], ou la relaxation lagrangienne [Eve63].

2.1.1.3 Le passage aux nombres entiers

Le cadre de la programmation linéaire atteint ses limites pour de nombreux problèmes qui requièrent des prises de décisions *discrètes* (on parlera de PLNE pour programmation linéaire en nombres entiers) et dont la solution optimale s'avère potentiellement très éloignée de l'optimum donné par leur approximation continue (on parle de **relaxation continue** puisqu'on relâche les contraintes d'intégralité des variables). La résolution s'appuie alors sur un mécanisme de **séparation** et **évaluation** (*branch and bound*)[3] :

1. La **séparation** consiste à simplifier le problème d'origine en faisant un certain nombre d'hypothèses jusqu'à ce que le problème résultant soit facile à résoudre (procédure de séparation ou de branchement [LMSK63]). En cas d'échec, il faut

[1] si la solution en question est réalisable et les valeurs des fonctions objectifs sont identiques

[2] En réalité les points du polyèdre primal et dual vont par paire et ce constat est à l'origine de la *dual simplex method* au cœur de l'analyse de sensibilité.

[3] Une autre solution (qui peut aussi être utilisée conjointement avec le *branch and bound*) est d'appliquer le simplexe sur l'enveloppe convexe des points entiers, mais comme l'extraction complète de cette enveloppe convexe mène en général à un nombre exponentiel de contraintes, on procède progressivement en appliquant une méthode de *génération de coupes*. Il s'agit de résoudre la relaxation continue et de produire un plan de coupe (une contrainte) *séparant* le point optimal donné par la relaxation du polyèdre courant. On génère ainsi facette par facette, l'enveloppe convexe des points entiers. La procédure peut néanmoins s'arrêter dès que l'optimum de la relaxation continue est entier.

naturellement revenir sur ces hypothèses. L'ensemble de la résolution se présente comme une recherche arborescente (en largeur, profondeur ou un mélange des deux) dans l'ensemble des hypothèses.

2. La phase **d'évaluation** s'enclenche après chaque nouvelle hypothèse et son objectif est d'interrompre si possible l'exploration. La relaxation continue est résolue pour s'assurer qu'elle n'est pas inférieure à la meilleure solution trouvée jusqu'à présent (elle fournit en effet une borne supérieure de z en cas de maximisation).

L'espace de recherche est partitionné sur le choix d'une variable fractionnaire x_i sur laquelle on fera les hypothèses $x_i \leq \lfloor x_i \rfloor$ dans le sous-arbre gauche et $x_i \geq \lceil x_i \rceil$ dans le sous-arbre droit de manière à lui imposer des valeurs entières (voir figure 2.4).

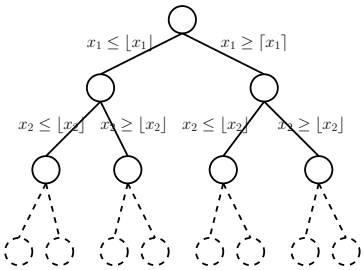


FIG. 2.4: arbre de recherche développé par un solveur linéaire en nombre entiers

Le choix de la variable est critique sur la taille de l'arbre de recherche final [LS99]. Ainsi, [FHT74] explique que brancher au nœud racine sur une variable qui n'affecte pratiquement pas la relaxation continue ne fait que doubler tout le travail de l'algorithme. Les stratégies de branchement ont donc fait l'objet de nombreuses études et s'appuient sur l'estimation de la décroissance de z. En cas de maximisation, la relaxation continue fournit une borne supérieure et chaque étape de séparation fait donc décroître la valeur de la fonction objectif. Deux *pseudocosts* notés P_i^- et P_i^+ (à l'image des coûts réduits du cadre continu) représentent cette dégradation pour chacune des deux décisions $x_i \leq \lfloor x_i \rfloor$ et $x_i \geq \lceil x_i \rceil$ de chaque variable x_i. On considère à un nœud j une variable non entière $x_i = \lfloor x_i \rfloor + f$ avec $0 < f < 1$. On note par ailleurs la décroissance de la fonction objectif D_i^{j-} obtenue en prenant la décision $x_i \leq \lfloor x_i \rfloor$ et D_i^{j+} celle obtenue en prenant $x_i \geq \lceil x_i \rceil$. Les *pseudocosts* représentent la décroissance de z par variation d'une unité de x_i et s'écrivent :

$$P_j^- = \frac{D_i^{j-}}{f} \quad P_j^+ = \frac{D_i^{j+}}{(1-f)}$$

La variable agissant le plus fortement sur l'objectif est retenue[4]. Enfin, comme les premiers choix sont cruciaux, l'initialisation des *pseudocosts* ainsi que la distinction entre *pseudocosts* identiques, sont des questions d'importance. À cette fin, des solutions intitulées *strong branching* [BCCL95] consistent à arbitrer les meilleurs candidats en évaluant une borne inférieure de la dégradation de l'objectif sur la base d'un nombre

[4]les *pseudocosts* peuvent être calculés explicitement en résolvant les programmes linéaires associés aux deux hypothèses pour obtenir la valeur de la relaxation. Cette solution trop coûteuse est remplacée par l'utilisation de leur valeur moyenne observée dans la recherche passée en supposant que cette valeur varie peu d'un nœud à l'autre de l'arbre.

limité d'itérations du simplexe dual pour chaque candidat. Il ne s'agit, ni plus ni moins, que d'une analyse de sensibilité pour chaque hypothèse candidate.

2.1.2 Programmation par contraintes

La programmation par contraintes (PPC) se veut une généralisation du cadre précédent dans laquelle aucune hypothèse n'est faite sur la nature des domaines, contraintes et objectifs. Néanmoins, nous considérerons dans cette thèse des domaines **finis** et **discrets** et nous nous pencheront à la fois sur des problèmes d'optimisation et de satisfaction (identifier une solution faisable). Les contraintes, en revanche, désignent tout type de relation logique, arithmétique, symbolique ..., ou encore décrite en extension par la donnée des combinaisons de valeurs compatibles ou incompatibles. Les premiers systèmes de contraintes tel que CHIP (*Constraint Handling In Prolog*) s'appuient sur la programmation logique et ses atouts déclaratifs [JL87]. Des solveurs plus récents tel que ILOG Solver ont conservé les aspects déclaratifs sans pour autant s'appuyer sur Prolog pour des raisons de performance. La PPC offre ainsi un langage de modélisation très riche avec des contraintes qui expriment des problématiques complexes mais récurrentes : `alldifferent`, `cumulative`, `global cardinality` [BCR05, AB93, Rég94]. La description des procédures de recherche peut également se situer à très haut niveau et faire partie intégrante du langage à l'image d'OPL [Hen02]. Nous nous appuierons par la suite sur les notions suivantes :

Définition 2.1.1 *Un **CSP** [Mon74] (*constraint satisfaction problem*) est défini par le triplet $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ où :*

- $\mathcal{X} = (x_1, \dots x_n)$ *désigne un ensemble de n variables ;*
- $\mathcal{D} = (D_1, \dots D_n)$ *est l'ensemble des domaines de sorte que chaque D_i contienne les valeurs possibles de x_i. On notera d la taille du plus grand domaine ;*
- $C = (C_1, \dots, C_m)$ *désigne l'ensemble des contraintes. Une contrainte C_i d'arité q est définie par un ensemble de q variables $vars(C_i) = \{x_{i1}, \dots x_{iq}\}$ ainsi que le sous-ensemble du produit cartésien $D_{i1} \times \dots \times D_{iq}$ spécifiant les valeurs des variables $(x_{i1}, \dots x_{iq})$ compatibles entre elles (ce sous-ensemble peut être défini en intension).*

Définition 2.1.2 *Une **solution** est un tuple $(s_1, \dots s_n)$ de valeurs pour chaque variable vérifiant toutes les contraintes de $\mathcal{C}$.*

Définition 2.1.3 *Le **réseau de contraintes** d'un* CSP $= \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ *est défini par l'hypergraphe R dont les nœuds correspondent aux variables, $\mathcal{X}$, du* CSP*, et les arêtes sont associées à chaque contrainte C_i et joignent l'ensemble des nœuds $vars(C_i)$.*

Définition 2.1.4 ***L'espace de recherche** désigne l'ensemble des combinaisons possibles de valeurs pour les variables : $D_1 \times \dots \times D_n$.*

La résolution en programmation par contraintes s'appuie sur deux éléments centraux :

- **l'énumération** : comme dans le contexte linéaire, l'énumération consiste à simplifier le problème par des hypothèses. Le processus est fondé sur une exploration arborescente en profondeur d'abord de l'ensemble des hypothèses. L'arbre résultant représente une manière de parcourir l'espace de recherche ;
- **la propagation** : il s'agit d'un mécanisme d'inférence logique destiné à simplifier, autant que possible, le problème dans son état courant (état initial ou déjà simplifié par des hypothèses provenant de l'énumération) de manière à réduire l'espace de recherche et éviter d'explorer inutilement des portions de l'espace dont on peut prouver qu'elles ne contiennent aucune solution (c'est de l'anticipation ou *look-ahead*).

2.1.2.1 La propagation

Cadre théorique. La réduction de l'espace de recherche s'effectue sur la base de mécanismes d'inférence associés à chaque contrainte. Chaque C_i a en effet la charge d'identifier les valeurs de ses variables qui n'appartiennent à aucune solution (valeurs inconsistantes) et de les éliminer. C'est le rôle des algorithmes de filtrage qui passent les domaines des variables au tamis et réduisent la combinatoire en éliminant des branches stériles de l'arbre de recherche. La relaxation continue abordée dans le cadre linéaire entre tout à fait dans cette catégorie puisqu'elle permet d'arrêter l'énumération sur la base d'une déduction concernant la meilleure solution possible du sous-arbre courant. La *finesse* du filtrage effectué par chaque contrainte représente le niveau de consistance atteint par la contrainte. La consistance la plus célèbre est la consistance d'arc (aussi appelée GAC pour *generalized arc consistency*) :

Définition 2.1.5 *Une contrainte C_k est dite **GAC** ou **arc-consistante** dès lors que toute variable x_i de $vars(C_k)$ et toute valeur v_i de D_i appartiennent à au moins un tuple $((x_1, v_1), \dots (x_i, v_i) \dots, (x_q, v_q))$, qui constitue une affectation des variables de $vars(C_k)$ vérifiant C_k et tel que $\forall j \in [1 \dots q], v_j \in D_j$. Cette affectation partielle constitue un **support** de (x_i, v_i).*

Définition 2.1.6 *Un CSP$=\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ est dit **GAC** ou **arc-consistant** si et seulement si chaque contrainte $C_i \in \mathcal{C}$ est GAC.*

L'arc-consistance désigne donc une consistance complète du point de vue d'une contrainte (toutes les valeurs inconsistantes pour chaque contrainte sont éliminées). Elle ne garantit pas néanmoins que toute valeur restante puisse être étendue à une solution, c'est une consistance **locale**.

Exemple 1 *Arc-consistance*

Soit le CSP constitué de trois variables x, y, z de domaines $D_x = \{1, 2, 3, 4\}$, $D_y = \{2, 3, 4\}$, $D_z = \{2, 3\}$ et deux contraintes c_1 : `alldifferent`(x, y, z) et $c_2 : x + 2y - z \leq 4$. Maintenir la GAC pour c_2 signifie éliminer les valeurs 4 de D_x et D_y ramenant les domaines à $\{1, 2, 3\}$, $\{2, 3\}$, $\{2, 3\}$. La propagation de ces réductions à c_1 permet d'éliminer les valeurs 2 et 3 de D_x aboutissant à $D_x = \{1\}$, $D_y = \{2, 3\}$, $D_z = \{2, 3\}$. Cet état constitue le point fixe et on ne peut pas aller plus loin. Le triplet $(x, y, z) = (1, 2, 2)$ constitue par exemple un support de $(x, 1)$ dans c_1 tandis que $(1, 2, 3)$ pourrait être son support dans c_2.

procédure : propage(Q) throws Contradiction

```
1: while Q ≠ ∅ do
2:    evt ← defile(Q) ;
3:    if evt = evt(x) { événement sur une variable} then
4:       for all c tel que x ∈ vars(c) do
5:          propageCt(c,evt,Q) ; {par exemple AC4(c, i ≠ a, Q)}
6:       end for
7:    else if evt = evt(ct) { événement sur une contrainte} then
8:       propageCt(c,evt(c),Q) ; {par exemple AC(scalar≤, evt(c), Q)}
9:    end if
10: end while
```

Algorithme 1: schéma de propagation événementielle

Dans le cas des contraintes définies en extension (par la donnée de toutes les combinaisons faisables ou infaisables de la contrainte), de nombreux algorithmes ont été donnés pour maintenir efficacement et incrémentalement l'arc-consistance : AC3, AC4, AC6, AC2001, etc. [Deb98, Bes06].

Pour maintenir le niveau de consistance du CSP, les déductions effectuées sur une variable sont partagées à toutes les autres contraintes portant sur cette variable par le mécanisme de propagation. La nouvelle information est ainsi propagée au reste du réseau jusqu'à l'obtention d'un point fixe ou d'une contradiction (domaine vide). Une telle procédure assure que le réseau est arc-consistant si chaque contrainte est arc-consistante. Néanmoins, rien n'oblige les contraintes à atteindre le même niveau de consistance.

Cadre algorithmique. Nous présentons à présent les idées précédentes dans un cadre algorithmique plus précis. En pratique, l'incrémentalité du filtrage, comme la granularité du réveil des contraintes, peut être prise en compte pour maintenir la consistance du CSP. Une contrainte est dès lors caractérisée par son comportement vis-à-vis des événements (*evt* sur algorithme 1) survenant au cours de la propagation. Deux types d'événement sont considérés ici :

- les événements $evt(x)$ portant sur une variable x : comme les inférences portent sur les domaines des variables, les événements typiques concernent la mise à jour d'une borne, le retrait d'une valeur ou l'instanciation ;
- les événements $evt(ct)$ portant sur une contrainte ct : il s'agit simplement de la demande du réveil complet de la contrainte (noté $evt(ct)$). C'est un événement moins informatif, souvent considéré dans la plupart des systèmes.

Les événements sur variables ne sont, ni plus ni moins, que l'application de l'algorithme de filtrage à un niveau de réveil plus fin. La procédure $propage(Q)$ de l'algorithme 1 décrit le schéma général de cette propagation événementielle. Il s'appuie sur :

- une file de propagation, Q, contenant les événements à propager ;
- une méthode, $propageCt(C, evt, Q)$, qui fait référence à l'algorithme de filtrage de C. Cet algorithme prend en entrée un événement de propagation, evt, pour pou-

Procédure : $AC4(C_{ij}, i \neq a, Q)$ throws Contradiction

1: **for all** $b \in D_j$ tel que $b \in supports(C_{ij}, a)$ **do**
2: nbSupports(j,b) ← nbSupports(j,b) - 1 ;
3: **if** nbSupports(j,b) = 0 **then**
4: $D_j \leftarrow D_j \backslash b$;
5: **if** $D_j = \emptyset$ **then** raiseContradiction() ;
6: $evt(j) \leftarrow j \neq b$;
7: $Q \leftarrow Q \cup \{evt(j)\}$;
8: **end if**
9: **end for**

Algorithme 2: Propagation de $i \neq a$ par AC4 avec $vars(C_{ij}) = \{i, j\}$.

Procédure : $AC(scalar_{\leq}, evt(c), Q)$ throws Contradiction

1: $lb \leftarrow cste - (\sum_{i=1}^{n_1-1} a_i \times inf(x_i) - \sum_{i=n_1}^{n_2} b_i \times sup(x_i))$
2: **for all** i **do**
3: **if** $i < n_1$, $newSup \leftarrow (lb + a_i \times inf(x_i))/a_i$
4: **else** $newSup \leftarrow (lb - b_i \times sup(x_i))/b_i$
5: **if** $newSup < sup(x_i)$ **then**
6: $sup(x_i) \leftarrow newSup$
7: $Q \leftarrow Q \cup \{x_i \leq newSup\}$
8: **end if**
9: **end for**

Algorithme 3: Propagation de $\sum_{i=1}^{n_1-1} a_i \times x_i - \sum_{i=n_1}^{n_2} b_i \times x_i \leq cste$

voir se comporter de manière incrémentale en maintenant le niveau de consistance de la contrainte sans procéder au réveil complet. Par ailleurs, ses déductions sont ajoutées à Q sous la forme de nouveaux événements sur les domaines de variables de $vars(C)$.

La méthode $propage(Q)$ procède donc simplement au réveil des contraintes susceptibles d'être affectées par l'événement courant jusqu'à l'obtention d'un point fixe ou d'une contradiction.

Le cœur de la propagation réside donc dans la méthode $propageCt(C, evt, Q)$ propre à chaque contrainte. Deux exemples sont donnés par les algorithmes 2 et 3 :

1. L'algorithme 2 ($AC4(C_{ij}, i \neq a, Q)$) présente une implémentation possible de $propageCt(C, evt, Q)$ qui atteint l'arc-consistance dans le cas d'une contrainte binaire (notée C_{ij}) en extension pour un événement de retrait de valeur ($i \neq a$). Cet algorithme, AC4, maintient le nombre de supports courants $nbSupports(j, b)$ pour une valeur b d'une variable j. À l'obtention de $i \neq a$, AC4 décrémente de un le nombre de supports de toutes les valeurs de j pour qui la valeur a était un support (c'est l'ensemble $supports(C_{ij}, a)$). Il élimine les valeurs de j qui n'ont plus de supports conformément à la définition de l'arc-consistance[5].
2. Un autre exemple (algorithme 3) de filtrage est donné pour une combinaison linéaire du type $\sum_{i=1}^{n_1-1} a_i \times x_i - \sum_{i=n_1}^{n_2} b_i \times x_i \leq cste$ (contrainte de type *sac à dos*) où a_i et b_i dénotent des quantités positives. Une telle contrainte est notée $scalar_{\leq}$. Cet exemple montre comment l'exploitation de la sémantique de la contrainte peut améliorer considérablement la propagation (travail uniquement aux bornes) et offrir des représentations beaucoup plus compactes des contraintes.

Le succès de la programmation par contraintes repose en grande partie sur le concept de **contrainte globale** [AB93, BCR05]. La modélisation de problèmes réels fait souvent appel à des problématiques récurrentes, des motifs correspondant à des

[5] nbSupports(j,b) est ici considéré comme un entier restaurable au *backtrack* et qui reste donc toujours consistant avec les domaines courants des variables.

réseaux de contraintes élémentaires très utilisés. Les contraintes globales désignent souvent des contraintes qui portent sur des ensembles de variables et maintiennent un niveau de consistance plus élevé que le niveau atteint par leur décomposition en contraintes élémentaires (voir [BH03] pour une définition exacte). Elles correspondent en général à des problématiques de haut niveau et encapsulent des algorithmes de filtrage très efficaces car ils exploitent la sémantique des contraintes. Ces contraintes sont à la fois des outils de résolution et de modélisation très efficaces.

2.1.2.2 L'énumération

La propagation est rarement suffisante pour réduire chaque domaine de chaque variable à un singleton et obtenir ainsi une solution. Une fois son point fixe atteint, l'étape suivante est d'explorer l'espace de recherche résultant.

Cadre théorique. L'exploration s'effectue de manière arborescente et de nombreux types de contraintes peuvent être utilisés pour *brancher* (construire l'arbre) mais la manière la plus courante est d'explorer les combinaisons possibles en affectant les valeurs une à une (les hypothèses sont donc du type $x_i = j$) :

Définition 2.1.7 *Une* ***contrainte de décision****, dc_i, désigne une hypothèse de la recherche. Par ailleurs, chaque nœud de l'arbre de recherche de profondeur k est identifié par un ensemble unique $DC_k = dc_1, \dots dc_k$ de contraintes de décision.*

Le mécanisme de retour arrière ou *backtrack* permet de revenir dans un état antérieur du problème après avoir rencontré un échec de manière à explorer une autre branche de l'arbre. L'état du problème à un nœud de l'arbre est désigné sous le terme de **monde** :

Définition 2.1.8 *À tout nœud DC_k de l'arbre de recherche est associé un* ***monde*** *de profondeur k désignant l'état du problème (état des domaines des variables et des structures de données du système) après la suite de décisions $DC_k = dc_1, \dots dc_k$.*

À l'image du cadre linéaire, les décisions ont une importance critique sur la taille de l'arbre final exploré. Pour des contraintes de décision du type $x_i = j$, deux heuristiques dites de choix de variables et de choix de valeurs doivent être définies pour décrire l'arbre de recherche. Les concepts évoqués dans le linéaire s'appliquent ici de manière similaire et les décisions permettant de contraindre au maximum le problème sont privilégiées. Ce principe a été reconnu sous le terme de *first-fail* [HE80][6] pour le choix des variables et mis en œuvre avec des heuristiques comme *mindom* qui sélectionne les variables de plus petit domaine d'abord ou *mindom/maxdeg* qui prend en plus en compte les variables de degré maximum dans le réseau de contraintes. Plus récemment, ces notions ont été généralisées sous l'appellation *d'impacts*, qui à l'instar des *pseudo-costs* sont calculés pour chaque variable [Ref04] tout au long de la recherche et mesurent l'influence de chaque choix sur la réduction de l'espace de recherche via la propagation. Nous reviendrons sur la notion d'impact au chapitre 6.

[6] *try first where you are the most likely to fail.*

Par ailleurs, on peut obtenir deux arbres de recherche bien distincts selon la sélection de la variable. On peut insister sur la même variable après un échec jusqu'à ce que son domaine soit vide (*d-way branching*) ou choisir une nouvelle variable (*2-way branching*). Dans ce dernier cas, on construit un arbre binaire avec $x_i = v$ dans la branche droite et $x_i \neq v$ dans la branche gauche.

Cadre algorithmique. Le *backtrack* est couramment implémenté à travers l'utilisation d'une pile (*trailing*) sur laquelle sont stockés les changements de toutes les structures *backtrackables* (les domaines des variables par exemple). La mise à jour de ces structures au moment du retour à un nœud de l'arbre est donc incrémentale puisque seules les modifications sont enregistrées[7]. Par ailleurs, le maintien de l'arc-consistance est effectué à tout instant de la recherche arborescente (algorithme MAC pour *maintaining arc consistency*) c'est-à-dire après l'ajout de chaque nouvelle contrainte de décision. L'algorithme 4 synthétise la procédure de recherche couplée à la propagation. Il s'appuie sur :

- la méthode $choixDecision()$: elle constitue l'heuristique de recherche qui guide l'exploration par le choix de la variable et la valeur de la prochaine affectation. La variable et la valeur indiquant la branche courante sont stockées par deux données restaurables au *backtrack*, i et a ;
- la méthode $worldPush()$: elle permet de sauvegarder la différence entre le monde courant et le monde précédent en marquant le haut de la pile ;
- la méthode $worldPop()$: elle permet de restaurer les valeurs de toutes les structures de données restaurables au *backtrack* : les domaines, l'ensemble des contraintes $\mathcal{C}$ et les valeurs de i et a correspondant à la dernière branche.

procédure : MAC

1: $Q \leftarrow \emptyset$; $(i \leftarrow null, a)$; {variable et valeur backtrackable de la branche courante}
2: **for all** $c \in \mathcal{C}$ **do** $Q \leftarrow Q \cup \{evt(c)\}$;
3: **try** propage(Q) **catch (Contradiction e) return** no solution ;
4: **loop**
5: $(i, a) \leftarrow choixDecision()$; {choix de i (variable non instanciée) et d'une valeur $a \in D_i$}
6: goDown(i,a,Q) ;
7: **if** $\forall x \in \mathcal{V}, |D_x| = 1$ **then return** solution ;
8: **end loop**

Algorithme 4: maintien de l'arc-consistance

Tant que toutes les variables ne sont pas instanciées (ligne 7 de la procédure MAC) ou que la preuve d'absence de solution n'a pas été faite (ligne 2 de $goUp(i, a, Q)$), une décision est prise en faisant appel aux heuristiques de choix de variables/valeurs sur les variables non instanciées et les domaines courants (ligne 5 de MAC). L'état courant du problème est enregistré ($worldPush()$) avant de prendre la décision (ici

[7] Le *trailing* limite en général les solveurs de contraintes à une exploration en profondeur d'abord pour des raisons de mémoire (la profondeur de l'arbre est bornée par le nombre de variables). Seuls les langages de contraintes plus récents supportent d'autres types d'exploration [Hen02]. Les solveurs linéaires en revanche ne rencontrent pas cette limite.

une affectation) et de la propager à l'ensemble du réseau (méthode $goDown(i, a, Q)$). En cas de contradiction (méthode $goUp(i, a, Q)$), l'algorithme revient en arrière (appel à $worldPop()$) de manière à restaurer le problème dans son état antérieur à cette décision. La décision inverse est propagée et à nouveau, si un point fixe est atteint sans croiser de contradiction, la recherche peut se poursuivre. On peut noter que l'absence de solution se caractérise par une valeur nulle pour i à la ligne 2 de $goUp(i, a, Q)$. Une telle valeur signife que l'appel à *worldPop* a permis de remonter au dessus du nœud racine et donc de retrouver la valeur initiale de i.

Procédure : $goDown(i, a, Q)$

```
1: worldPush() ;
2: dc_ia ← i = a ; C ← C ∪ dc_ia ;
3: Q ← {evt(dc_ia)} ;
4: try
5:    propage(Q) ;
6: catch(Contradiction e)
7:    goUp(i,a,Q) ;
8: end try catch
```

Algorithme 5: descente dans l'arbre

Procédure : $goUp(i, a, Q)$

```
1: worldPop() ;
2: if i = null then return no solution ;
3: ¬dc_ia ← i ≠ a ; C ← C ∪ ¬dc_ia ;
4: Q ← {evt(¬dc_ia)} ;
5: try
6:    propage(Q) ;
7: catch(Contradiction e)
8:    goUp(i,a,Q) ;
9: end try catch
```

Algorithme 6: remontée dans l'arbre

2.2 Synthèse des recherches arborescentes

Les approches décrites précédemment correspondent dans les grandes lignes aux mécanismes mis en œuvre dans les solveurs commerciaux même si de nombreux détails ont été passés sous silence. Ce sont donc des éléments qui ont acquis la maturité suffisante pour devenir des composants essentiels de ces outils. Ces approches ont néanmoins des lacunes bien connues auxquelles il n'existe pas encore de solution suffisamment mature pour faire partie intégrante des solveurs mais que la recherche aborde depuis longtemps.

2.2.1 Lacunes

Nous avons évoqué les mauvais points de choix initiaux (en haut de l'arbre) et leurs conséquences désastreuses (un choix qui ne simplifie pas le problème, qui n'apporte pas une nouvelle information significative, ne fait que multiplier le travail de l'algorithme). Un autre phénomène a fait l'objet d'études en PPC, c'est le *thrashing* : des zones identiques de l'espace de recherche peuvent être explorées inutilement de manière redondante. Le solveur re-découvre toujours la même contradiction dont l'origine est située bien en amont dans les hypothèses faites par la recherche.

Exemple 2 *Première illustration du* thrashing

Soit le problème de coloration défini par un CSP de 5 variables x_i de domaines $D_{x_1} = D_{x_2} = D_{x_3} = \{1, 2, 3\}$ et $D_{x_4} = D_{x_5} = \{2, 3\}$ ainsi que les contraintes de

différence $x_1 \neq x_2$, $x_2 \neq x_3$, $x_3 \neq x_4$, $x_3 \neq x_5$, $x_4 \neq x_5$ dont le réseau associé est représenté figure 2.5 (a). Imaginons que l'heuristique de choix de variables re-

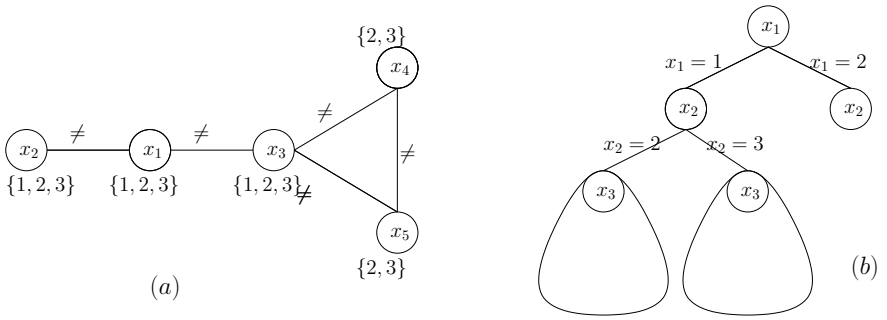


FIG. 2.5: premier exemple de *thrashing*.

tienne les variables dans l'ordre croissant de leurs index et que les domaines soient parcourus de manière croissante (de la borne inférieure vers la borne supérieure), l'arbre de recherche résultant (avec l'algorithme MAC) est représenté figure 2.5 (b). On remarque sur cet arbre que l'instanciation de x_1 à 1, crée un sous-problème infaisable (une clique de différences de 3 variables (x_3,x_4,x_5) avec 2 valeurs dans leurs domaines). Cette infaisabilité[8] est indépendante du choix fait sur x_2 et les deux sous-arbres provenant de $x_2 = 2$ et $x_2 = 3$ sont identiques et explorés en double inutilement, c'est le phénomène de *thrashing*.

L'exemple précédent met en évidence un *thrashing* provenant de certaines structures du problème. On se rend compte qu'un réseau de contraintes peu dense peut vite entraîner la recherche à créer des sous-problèmes indépendants (dynamiquement) et à *bactracker* inutilement dans un sous-problème (x_2 dans l'exemple) alors que l'erreur prend son origine dans une autre partie du problème (x_1,x_3,x_4 et x_5 sur l'exemple). Le *thrashing* peut également survenir dans d'autres circonstances.

Exemple 3 *Deuxième illustration du* thrashing

Soit le CSP défini par 6 variables x_i de domaines $D_{x_i} = \{1, 2, 3, 4\}$ et les 3 contraintes décrites figure 2.6. *occurence*$(\{x_1, x_2, x_3\}, 3) = 2$ spécifie que le nombre d'occurences de la valeur 3 dans la liste $\{x_1, x_2, x_3\}$ doit être égale à 2. On suppose que l'heuristique de recherche est celle donnée par l'arbre de la figure 2.6 (on instancie les variables dans l'ordre x_6, x_3, x_4, x_1 et on parcourt l'arbre de recherche de la figure de la gauche vers la droite).

En instanciant x_3 à 3, la recherche engendre une situation conflictuelle entre $x_1 = x_2$ et le fait qu'il ne doit y avoir qu'un et un seul 3 pour les deux variables. Les deux contraintes réalisent l'arc-consistance mais étant considérées séparément, aucune contradiction n'est décelée par un solveur à ce niveau. La recherche se poursuit donc inutilement sur x_4 et la contradiction sera identifiée en double à nouveau (zone entourée en trait plein sur la figure). On peut noter que la preuve

[8] L'utilisation d'un `alldifferent` pour la clique aurait permis d'éliminer dès la racine les valeurs 2,3 de x_3. Néanmoins, on peut remplacer cette clique par n'importe quel réseau de contraintes dans lequel la valeur 1 de x_3 n'appartient à aucune solution. Nous avons conservé les différences pour des raisons de simplicité.

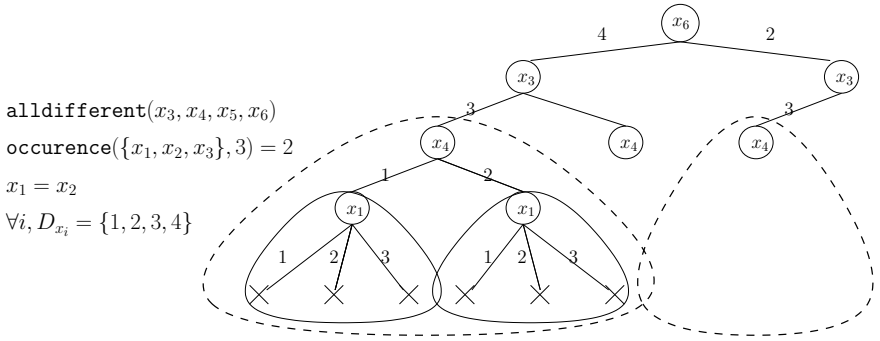


FIG. 2.6: deuxième exemple de thrashing.

réalisée pour $x_3 = 3$ est indépendante à nouveau du premier choix sur x_6 et sera inutilement répétée (zone en pointillé) dans l'autre partie de l'arbre (si on cherche par exemple toutes les solutions de ce problème).

Ce second exemple montre comment un défaut de propagation peut conduire la recherche dans une exploration redondante. Il met également en évidence que le *thrashing* peut se produire bien sûr en aval de l'hypothèse conflictuelle mais aussi finalement, dans n'importe quelle autre partie de l'arbre. C'est un point crucial pour comprendre les algorithmes de *backtrack* intelligent tels que le *backjumping* ou le *dynamic backtracking* (abordés au chapitre 3) imaginés pour compenser ce type de comportement.

La relative indépendance de certaines parties du problème comme des défauts au niveau de la propagation peuvent se combiner avec l'heuristique de recherche pour créer des situations de *thrashing* irrémédiables. Ce phénomène se produit en PPC comme en PLNE. Les **explications** que nous allons aborder dans la suite sont le remède imaginé en PPC.

2.2.2 Synthèse

Les techniques de résolution employées en PLNE comme en PPC sont très proches dans la mesure où elles s'appuient sur une exploration arborescente de l'espace et mettent en place des mécanismes d'inférence pour élaguer cet arbre. Elles souffrent des mêmes lacunes et deux d'entre elles (les choix initiaux et le *thrashing*) ont été abordées ici en détails car elles seront au cœur de cette thèse. D'autres lacunes existent néanmoins. À titre d'exemple, il existe toujours différentes manières de modéliser un problème, c'est-à-dire de décrire une solution au problème à travers un jeu de variables et de contraintes. Ces modèles sont néanmoins rarement équivalents du point de vue du solveur. Du coté PLNE, c'est la qualité de la relaxation continue (le polyèdre implicitement défini par le modèle approchant plus ou moins bien l'enveloppe des points entiers) qui est en cause alors que du côté PPC, c'est la qualité de la propagation (suivant la formulation, certaines déductions peuvent être perdues). Identifier un bon modèle reste un travail d'expert. Les deux domaines partagent donc le concept de contrainte redondante (aussi appelée coupe en PLNE) menant sur des modèles redondants, qui, sans être nécessaires à la définition du problème, participent à une formulation plus forte.

Aujourd'hui, si de nombreux types de coupes peuvent être identifiés automatiquement par les solveurs linéaires, les solveurs de contraintes sont encore incapables d'aider l'utilisateur à construire un bon modèle [Col06]. Tous ces éléments mettent en évidence les similitudes entre deux domaines d'origines différentes qui se sont longtemps ignorés. Dans ces dernières années, il a été clairement reconnu que les deux approches semblent très complémentaires. En effet, si les principes sont les mêmes, la méthodologie est radicalement différente. Un modèle PLNE s'appuie souvent sur des variables binaires pour renforcer facilement la relaxation continue par des coupes alors qu'un modèle PPC préfère les variables entières pour conserver et exprimer les structures combinatoires du problème. L'un concentre ses efforts sur la fonction objectif et le calcul de bornes inférieures, tandis que l'autre capture plus finement l'espace des solutions faisables à travers la propagation. Tous ces aspects ont été souvent soulignés et des approches **hybrides**, avec la volonté d'exploiter les forces respectives de chacun, ont vu le jour [Mil03, JG01, FLM00, HOTK00]. Pour John Hooker [Mil03], les deux technologies sont des instances d'un même et unique *framework* basé sur la recherche, la déduction et la relaxation. Néanmoins, il est rarement observé qu'un élément central au cœur de la programmation linéaire reste complètement absent de la programmation par contraintes : la dualité.

Le dual est un outil qui fournit des capacités "d'introspection" sur le problème comme sur la manière dont il est résolu. Tous les éléments évoqués en 2.1 (*reduced costs*, *shawdow prices*, *pseudocosts*, sensibilité, *strong branching*,...) prennent plus ou moins leurs origines dans le concept de dualité. Par ailleurs, c'est un concept crucial qui apparaît souvent dans la littérature des approches complètes de la recherche opérationnelle (à base de décomposition et de relaxation [Ben04]). La question est donc naturelle : quel est le dual de la programmation par contraintes ?

La définition du dual dans un contexte linéaire discret affaiblit ses propriétés (on parle de dualité faible, primal et dual ne coïncident plus forcément à l'optimum mais constituent toujours des bornes l'un pour l'autre). Elle devient nettement plus délicate dans un contexte non linéaire. Néanmoins, **les explications** dont il va maintenant être question remplissent partiellement ce rôle dans la mesure où elles fournissent certaines informations de nature duale.

Chapitre 3

Explorer l'espace de recherche intelligemment

> *"There is no great mystery in this matter" he said, taking the cup of tea which I had poured out for him ; "the facts appear to admit of only one explanation."*
>
> Sherlock Holmes (Doyle, 1890)

Le cœur de cette thèse porte sur les **explications** et leur rôle dans la résolution. Nous faisons donc ici un tour d'horizon de tous les algorithmes qui s'appuient sur les explications ou des concepts analogues. Les algorithmes de *backtrack* intelligent, dynamique ou les recherches locales basées sur la réparation d'affectations partielles comme les techniques d'enregistrement de *nogoods* sont passés en revue.

3.1 Les explications : concept et calcul

Nous introduisons dans un premier temps les principales définitions relatives aux explications avant de nous intéresser à leur calcul et maintien dans un solveur.

3.1.1 Notions et terminologie

Les explications nous viennent de l'intelligence artificielle à travers les systèmes experts et de manière plus générale les systèmes à base de connaissances. On considère à ce moment là qu'un élément essentiel de ces systèmes réside dans leur capacité à justifier leurs conclusions et leur comportement [Doy79, de 86]. Un utilisateur ou un développeur veut pouvoir comprendre l'enchaînement des règles appliquées par le moteur d'inférences du système pour aboutir à la conclusion. Un ensemble d'hypothèses suffisantes pour atteindre la conclusion constitue une **explication**. Ce type de raisonnement est identifié dans un cadre logique sous le terme **d'abduction** (*Inference to the best explanation*). C'est le raisonnement par excellence du détective ou du médecin qui cherchent tous les deux la meilleure explication. Une abduction logique est définie dans le cadre suivant [Jos03, EG93] :

- une théorie $\mathcal{T}$ définissant le domaine d'application sous la forme d'une conjonction de formules (axiomes) ;

- un ensemble $\mathcal{H}$ de formules qui représentent des hypothèses possibles ;
- un ensemble $\mathcal{M}$ de formules constituant les résultats logiques à justifier (les manifestations).

L'abduction est l'opération qui consiste à identifier un sous-ensemble $S \subseteq H$ tel que $T \cup S$ soit consistant et permette de dériver logiquement $\mathcal{M} : S \models \mathcal{M}$ sous la théorie $\mathcal{T}$. Des questions intéressantes se posent sur les éléments de H concernant leur pertinence, (une hypothèse h appartient-elle à au moins une explication ?) ou encore leur nécessité (h intervient-elle dans toutes les explications ?). Les questions sur S sont relatives à son existence et son calcul. Une explication étant rarement unique, on peut exprimer différentes exigences sur son calcul et chercher par exemple une explication minimale au sens de la cardinalité conformément au principe du rasoir d'Occam (entre deux explications, la meilleure est la plus simple, celle qui fait intervenir le moins d'hypothèses). Le critère de la cardinalité minimale est souvent très difficile à atteindre (pour les complexités détaillées, on pourra se référer à [EG93]) et on peut relâcher avantageusement ce critère pour la minimalité au sens de l'inclusion dont la complexité théorique est inférieure. D'autres critères existent basés sur l'ajout de priorités ou de pénalités aux hypothèses mais en pratique, on s'appuiera dans cette thèse sur des explications minimales au sens de l'inclusion (explications minimales pour la suite) ou des explications arbitraires.

L'abduction a pris différentes formes en programmation par contraintes à travers les termes de *nogoods*, *removal explanations*, *contradiction explanations* ou encore *conflicts sets* [dK86, Gin93]. Il s'agit de justifier les raisonnements logiques effectués au moment de la propagation donc de justifier l'état courant des domaines des variables ou le fait que le problème soit sans solution. Dans tous les cas, le cadre peut s'instancier de la manière (informelle) suivante :

- $\mathcal{T}$ = ensemble des domaines ;
- $\mathcal{H}$ = ensemble des contraintes $\cup$ ensemble des contraintes de décision prises par la recherche jusqu'au nœud courant ;
- $\mathcal{M}$ = le domaine d'une variable x est réduit.

La réduction d'un domaine peut se voir comme le retrait d'une valeur ou la modification de l'une de ses bornes. Toutes les réductions de domaines se ramènent au final à un ensemble de retraits de valeurs ce qui nous amène à la définition suivante de l'explication[Jus03] :

Définition 3.1.1 *Soit $\mathcal{C}$ l'ensemble des contraintes originelles du problème et $\mathcal{DC}$, l'ensemble des contraintes de décisions, l'explication e du retrait de la valeur a du domaine de x est la donnée de deux sous-ensembles $\mathcal{C}_e \subseteq \mathcal{C}$ et $\mathcal{DC}_e \subseteq \mathcal{DC}$ tels que :*

$$\mathcal{C}_e \wedge \mathcal{DC}_e \models x \neq a$$

On notera $expl(x \neq a) = \mathcal{C}_e \cup \mathcal{DC}_e$ une telle explication.

Cette définition fait davantage référence aux *removal explanations* introduites par [Gin93]. Une explication de contradiction désigne l'explication d'un échec rencontré au cours de la recherche (on peut considérer sans perte de généralité qu'un tel échec prend son origine dans un domaine vide[1]).

[1] Une contrainte peut identifier une contradiction sans vider explicitement le domaine d'une variable et calcule dès lors elle-même l'explication correspondante.

Définition 3.1.2 *Une explication de contradiction est l'explication de l'absence de valeurs pour une variable x ($D_x = \emptyset$), de sorte que :*

$$expl(D_x = \emptyset) = \bigcup_{v \in D_x} expl(x \neq v)$$

La définition de l'explication couvre également les concepts de *nogoods* et de *conflicts sets*. Un *nogood* désigne l'ensemble des contraintes de décisions associées à un échec. Il s'agit simplement d'un ensemble d'hypothèses contradictoires.

Définition 3.1.3 *À toute explication de contradiction $expl(D_x = \emptyset) = \mathcal{C}_e \cup \mathcal{DC}_e$ est associée le* nogood *nog_e défini par l'ensemble des décisions :*

$$nog_e = \mathcal{DC}_e$$

A l'origine, les *nogoods* sont sont définis comme des affectations partielles inconsistantes (qui ne peuvent être étendues à aucune solution) et sont donc introduits dans le cadre d'un branchement de type affectation $x_i = val$. Dans ce contexte, $\mathcal{DC}_e$ désigne souvent un ensemble d'affectations.

Exemple 4 *explications,* nogoods

Sur l'exemple 2 (page 23), après avoir posé $x_1 = 1$, la valeur 1 est retirée du domaine de x_2 et x_3 avec les explications $expl(x_2 \neq 1) = \{x_1 = 1, x_1 \neq x_2\}$ et $expl(x_3 \neq 1) = \{x_1 = 1, x_1 \neq x_3\}$. Une fois que le sous-arbre de racine x_3 a été prouvé inconsistant en vidant le domaine de x_3, on dispose de l'explication de contradiction $expl(x_3 = \emptyset) = \{x_1 = 1, x_3 \neq x_4, x_4 \neq x_5, x_5 \neq x_3\}$, $x_1 = 1$ est donc un *nogood*, une affectation partielle qui ne peut être étendue à aucune solution.

Un *conflict set* désigne un ensemble de contraintes justifiant l'inconsistance d'un état de la recherche. Le *conflict set* obtenu au nœud racine est l'explication attendue par l'utilisateur en cas de problème sur-contraint (voir exemple 5).

Définition 3.1.4 *À toute explication de contradiction $expl(D_x = \emptyset) = \mathcal{C}_e \cup \mathcal{DC}_e$ est associée le* conflict set *$cset_e$ tel que :*

$$cset_e = \mathcal{C}_e$$

Exemple 5 *conflicts sets*

En cherchant le nombre minimal de couleurs *nbc* avec lesquelles on peut colorer le graphe de la figure 3.1, on pourrait obtenir trois explications (ou conflicts sets) justifiant $nbc > 2$: $cset_1 = \{x_3, x_4, x_5\}$, $cset_2 = \{x_1, x_2, x_3, x_5, x_6\}$ ou encore $cset_3 = cset_1 \cup cset_2$. $cset_1$ et $cset_2$ sont tous les deux minimaux au sens de l'inclusion tandis que $cset_1$ est en plus minimal pour la cardinalité.

Nous noterons par la suite $|e|$, la cardinalité d'une explication, c'est à dire le nombre de contraintes contenues dans e : $|\mathcal{C}_e \cup \mathcal{DC}_e|$. Les cardinalités du *nogood* et du *conflict set* associées à e sont donc respectivement définies par $|\mathcal{DC}_e|$ et $|\mathcal{C}_e|$. On dira également qu'une explication e_1 est **plus précise** que e_2 dès lors que $|e_1| < |e_2|$.

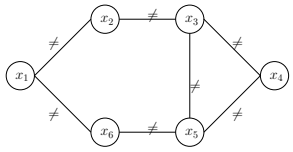


FIG. 3.1: un problème de coloration

3.1.2 Calcul *a posteriori* ou à la volée

Le maintien de l'arc-consistance (d'une manière plus générale, la propagation) étant devenu un composant incontournable dans la résolution des CSP, le calcul des explications comme les algorithmes de *backtrack* intelligent seront abordés dans ce cadre. On peut distinguer deux types de méthodes pour calculer une explication :

Calcul *a posteriori*. Ici le calcul s'effectue après que la preuve de l'inconsistance ou l'état à justifier ait été atteint. Cet état est noté $\perp$. Pour une technique m d'inférence ou résolution (m peut correspondre à l'arc-consistance), l'algorithme XPLAIN proposé par [dSP88] permet d'obtenir une explication minimale au sens de l'inclusion de k hypothèses parmi n en $O(n \times k)$ appels à m. Cet algorithme a été affiné de manière dichotomique par [Jun01] pour devenir QUICKXPLAIN et offrir une complexité $O(n \times log(k+1) + k^2)$, intéressante dès que k est petit par rapport à n. XPLAIN est donné par l'algorithme 7 où C désigne l'ensemble d'hypothèses parmi lesquelles on cherche une explication X relativement à une technique d'inférence m.

procédure : XPLAIN(C, m)

1: $X \leftarrow \emptyset$;
2: $c_1, \ldots, c_n$ une énumération de C
3: $X_{temp} \leftarrow \emptyset$
4: **while** $m(X) \neq \perp$ **do**
5: $k \leftarrow 0$
6: **while** $m(X_{temp}) \neq \perp$ and $k < n$ **do**
7: $k \leftarrow k + 1$
8: $X_{temp} \leftarrow X_{temp} \cup c_k$
9: **end while**
10: **if** $(m(X_{temp}) \neq \perp$ **return** $\emptyset$
11: $X \leftarrow X \cup c_k$
12: $X_{temp} \leftarrow X$
13: **end while**
14: **return** X

Algorithme 7: calcul d'une explication minimale au sens de l'inclusion

Son fonctionnement est assez intuitif :

1. Il procède en ajoutant une à une les hypothèses c (contraintes) du système (lignes 6-8) jusqu'à l'obtention d'une contradiction (notée $\perp$) par m pour la suite $c_0, \ldots, c_p$ ($m(c_0, \ldots, c_p) = \perp$). La dernière hypothèse c_p est donc nécessaire

à l'explication minimale, X (ligne 11), incluse dans $c_0, \ldots, c_p$ puisque $c_0, \ldots, c_{p-1}$ était cohérent.

2. L'étape suivante consiste à recommencer en ajoutant immédiatement c_p (ligne 12) et construire la suite $c_p, c_0, \ldots, c_{p'}$ telle que $m(c_p, c_0, \ldots, c_{p'}) = \perp$ de sorte que c_p et $c_{p'}$ appartiennent à l'explication minimale X.

XPLAIN construit ainsi itérativement l'explication minimale incluse dans $c_0, \ldots, c_p$ et s'arrête dès que $m(X) = \perp$ (ligne 4). L'avantage majeur de cette technique est son caractère non intrusif, elle fonctionne indépendamment de m. Les limites de cette approche sont néanmoins atteintes dès que m fait référence à une technique de résolution non polynomiale. C'est naturellement le cas pour l'algorithme MAC. Ainsi le calcul de $cset_2$ de l'exemple précédent 5 exigerait six résolutions du problème de coloration.

Calcul à la volée. Une autre manière d'obtenir une explication est de tracer le comportement de l'algorithme de résolution afin de conserver en mémoire l'ensemble des hypothèses nécessaires à son exécution. Dans un contexte PPC, il s'agit de justifier toutes les inférences faites au moment de la propagation par les contraintes. Comme ces inférences s'appuient sur l'état des domaines, on doit conserver en mémoire les explications justifiant cet état courant et donc stocker $O(nd)$ explications dans le pire des cas (une pour chaque valeur de chaque domaine). Par ailleurs, l'algorithme de filtrage doit désormais fournir le calcul de l'explication. Prenons l'exemple de l'algorithme 2 pour l'arc-consistance d'une contrainte en extension assurée par $AC4$ [JDB00]. La méthode $AC4(C_{ij}, i \neq a, Q)$ doit être instrumentée de manière à expliquer ses inférences (voir algorithme 8).

procédure : $AC4(C_{ij}, i \neq a, Q)$ throws Contradiction

```
1: for all b ∈ D_j tel que b ∈ supports(C_ij, a) do
2:     nbSupports(j,b) ← nbSupports(j,b) - 1 ;
3:     if nbSupports(j,b) = 0 then
4:         D_j ← D_j \ b ;
5:         expl(j ≠ b) ← ⋃_{s∈supports(C_ij,b)} expl(a ≠ s) ;
6:         if D_j = ∅ then raiseContradiction() ;
7:         evt(j) ← j ≠ b ;
8:         Q ← Q ∪ {evt(j)} ;
9:     end if
10: end for
```

Algorithme 8: propagation de $i \neq a$ par AC4 avec $Vars(C_{ij}) = \{i, j\}$

Les algorithmes de filtrage exploitent la sémantique des contraintes pour offrir des mécanismes efficaces et dédiés au sous-problème considéré par la contrainte. Pour bénéficier dans ce cadre du calcul à la volée des explications, chaque contrainte doit être instrumentée pour fournir une explication associée à chacune de ses inférences. Cette méthode est donc particulièrement intrusive pour le solveur. Par ailleurs, même si chaque contrainte calcule une explication minimale, rien ne garantit la minimalité globale de l'explication, dépendante de l'ordre dans lequel s'enchaînent les inférences

de la propagation. En pratique, on conserve donc une explication arbitraire correspondant au raisonnement courant du moteur de propagation. PaLM (*Propagate, Learn and Move*), [JB00], est à notre connaissance le seul solveur qui propose un tel mécanisme dans un cadre de contraintes en intension. On peut noter qu'avec des consistances locales moins fortes que l'arc-consistance comme le *forward-checking* on pourrait obtenir un mécanisme générique d'explication en utilisant simplement les variables de la contrainte directement responsables de l'échec. Un lecteur intéressé pourra se référer à [Roc05] pour des détails sur l'implémentation du calcul à la volée.

3.2 Backtrack intelligent, backtrack dynamique

Les explications sont au cœur des algorithmes de *backtrack* intelligent qui cherchent à corriger le phénomène de *thrashing* présenté à la section 2.2.1. L'analyse des échecs permet de revenir à des hypothèses pertinentes de la recherche et leur conservation en mémoire vise à éviter de reproduire les mêmes erreurs. Les algorithmes de *backtrack* intelligent s'appuient donc sur un calcul à la volée des explications.

3.2.1 Conflict directed Backjumping

Conflict directed Backjumping (CBJ) est proposé à l'origine par [Gas77] mais son utilisation conjointe avec le maintien de l'arc-consistance (MAC-CBJ) apparaît en 1995 [Pro95]. L'idée est de revenir sur des points de choix qui sont effectivement responsables de la contradiction rencontrée en conservant les relations de dépendance entre les inférences effectuées au moment de la propagation et les contraintes de décision. En mémorisant ces explications, on devrait disposer à tout moment d'une explication de l'échec courant et pouvoir revenir sur la dernière contrainte impliquée dans cette explication et non pas la dernière hypothèse posée par la recherche.

Exemple 6 *backjumping*

Sur le premier exemple de *thrashing* (exemple 2 page 2), à l'obtention de la première contradiction, l'explication de l'échec est la suivante : $expl(x_3 = \emptyset) = \{x_1 = 1, x_3 \neq x_4, x_4 \neq x_5, x_5 \neq x_3\}$. x_2 étant absent de l'explication, on peut effectuer un saut sur x_1 et immédiatement essayer $x_1 = 2$.

L'algorithme se présente donc comme une recherche arborescente non chronologique puisqu'on peut effectuer des sauts dans l'arbre pour revenir sur des points de choix pertinents. En dehors du calcul des explications, MAC-CBJ diffère de MAC sur la méthode $goUp(i, a, Q)$ qui devient CBJ $-goUp(i, a, e, Q)$ où e représente l'explication associée à l'échec courant. Les lignes 1-4, 8 et 13 de l'algorithme 9 de la page 32 mettent en évidence les modifications apportées à l'algorithme 6 pour obtenir le *backjumping* :

- les lignes 1-4 décrivent le processus de saut dans l'arbre par des appels à $worldPop()$ jusqu'à parvenir à la décision conflictuelle dc_{jb} ;
- la ligne 8 justifie le retrait de la valeur b du domaine de la variable i en utilisant l'explication d'échec ;
- enfin, $expl(e)$ ligne 13, dénote l'explication obtenue pour l'échec e dans la mesure où toutes les contraintes sont maintenant expliquées.

procédure : CBJ - $goUp(i, a, e, Q)$

```
1: dc_jb ← dernièreDécision(e) ;
2: while ((i, a) ≠ (j, b) ∧ i ≠ null) do
3:     worldPop() ;
4: end while
5: worldPop() ;
6: if i = null then return no solution ;
7: ¬dc_jb ← j ≠ b ; C ← C ∪ ¬dc_jb ;
8: expl(j ≠ b) ← e\{dc_jb} ;
9: Q ← {evt(¬dc_jb)} ;
10: try
11:     propage(Q) ;
12: catch(Contradiction e)
13:     goUp(i,a,expl(e),Q) ;
14: end try catch
```

Algorithme 9: CBJ - remontée non chronologique dans l'arbre de recherche

Liens avec le principe de résolution. On peut voir un *nogood* nog_e comme une nouvelle contrainte ajoutée au problème sous la forme d'une clause, $\bigvee_{dc_i \in e}(\neg dc_i)$ pour interdire cette situation conflictuelle dans la suite de la recherche. L'explication $expl(j \neq b)$ peut donc se lire $\bigvee_{dc_i \in expl(j \neq b)}(\neg dc_i) \vee (j \neq b)$. Une explication d'échec obtenue par la recherche (*i.e.* quand un domaine est vide après l'essai de chacune des valeurs possibles de la variable) est calculée implicitement en appliquant le **principe de résolution** entre les *nogoods* justifiant l'absence de chaque valeur. Le principe de résolution en logique propositionnelle s'écrit :

$$a \vee A, \neg a \vee B \vdash A \vee B \tag{3.1}$$

Imaginons que le domaine de x_j soit binaire et notons j et $\neg j$ les deux décisions $x_j = 1$ et $x_j = 0$. Une fois le domaine de x_j vide, on procède à la résolution des deux explications :

$$(\bigvee_{dc_i \in expl(j \neq 1)} dc_i) \vee j, (\bigvee_{dc_i \in expl(j \neq 0)} dc_i) \vee \neg j \vdash \bigvee_{dc_i \in expl(j \neq 1) \cup expl(j \neq 0)} dc_i$$

Cette simplification des explications est faite ligne 8 de l'algorithme qui anticipe la résolution des *nogoods* en retirant immédiatement la décision remise en cause. On fait référence au principe de résolution dans un cadre non binaire sous le terme d'**hyper-résolution**. C'est la résolution des *nogoods* entre eux qui permet de construire des *nogoods* intéressants qui contiennent l'information correspondant à une phase de recherche et de déductions.

CBJ évite le *thrashing* produit en aval d'un mauvais point de choix (exemple 2 page 23) mais il ne supprime pas tout le *thrashing* et notamment celui qui se produit latéralement dans l'arbre de recherche (exemple 3 page 24). À cette fin, un algorithme plus subtil a été mis au point : *Dynamic Backtracking*.

3.2.2 Dynamic Backtracking

Dynamic backtracking, DBT, renonce définitivement à la recherche arborescente (mais pas à la complétude) et propose une autre manière d'explorer l'espace de recherche. Le concept de *backtrack* disparaît pour celui de réparation. L'idée est de réparer le chemin de décision courant en retirant les hypothèses coupables sans oublier tout le travail réalisé en aval de ces hypothèses (contrairement à ce que fait le *backtrack* ou le *backjumping*). Le principe est de conserver les inférences effectuées jusqu'à présent tant que leur explication (leur raison d'être) est toujours valide.

Définition 3.2.1 *Soit DC_k le chemin de décision courant $dc_1, \ldots, dc_k$. Une inférence $x \neq i$ est dite* ***valide*** *tant que $nog_{expl(x \neq i)} \subseteq DC_k$.*

En cas de contradiction, on retire la décision la plus récente contenue dans l'explication de contradiction et on répare l'état courant du problème en retirant toutes les inférences qui dépendent de cette décision. On laisse notamment les inférences valides qui sont indépendantes de la contradiction courante. L'exploration n'est plus arborescente, le *trailing* devient impossible et il faut maintenir des explications pour pouvoir réparer l'état courant. Les contraintes doivent avoir désormais le souci de la décrémentalité en plus de l'incrémentalité au niveau de leur structure de données [GJR04, Roc05]. Autant d'éléments qui font de DBT et particulièrement de MAC-DBT [JDB00] un algorithme nettement plus complexe que les algorithmes arborescents présentés jusqu'à présent.

MAC-DBT est décrit à travers les procédures *move* (algorithme 10) et *repair* (algorithme 11) qui remplace *goDown* et *goUp* de MAC ou MAC-CBJ. On notera l'absence de *worldPush*(), *worldPop*(). La réparation s'effectue de la manière suivante :

- ligne 1 : si le *nogood* de l'explication de contradiction, e, est vide, c'est la preuve que le problème est inconsistant indépendamment des hypothèses de la recherche ;
- lignes 3-5 : sinon dc_{jb} dénote la décision la plus récente contenue dans e. Les valeurs supprimées à cause de dc_{jb} sont restaurées dans les domaines ;
- lignes 6-8 : des événements de restauration (notés $\lhd$) sont propagés à l'ensemble du réseau. La raison de cette rétro-propagation est qu'il existe peut-être une autre façon de déduire l'impossibilité de ces valeurs (il y a souvent plusieurs explications pour la même cause) sans faire appel à dc_{jb} ;
- ligne 9 : l'appel récursif à *repair*, en cas de contradiction, provient du fait que le retrait de dc_{jb} peut ne pas être suffisant pour lever la contradiction. Par ailleurs, il est possible que l'explication précédente ne soit plus valide (impliquant une décision elle-même retirée par les appels récursifs à *repair*) ce qui explique le test de la ligne 11 ;
- lignes 12-16 : en cas de rétro-propagation réussie, on poursuit comme en *backjumping* en procédant à la *résolution* des explications (ligne 12) avant de propager à nouveau la négation (lignes 13,14).

D'un point de vue de l'implémentation, on peut noter que la ligne 4 de l'algorithme demande de conserver les relations de dépendance entre une décision et l'ensemble des explications dans lesquelles elle intervient. C'est ce réseau de dépendance qui remplace dans MAC-DBT, la pile utilisée par le trailing de MAC.

Si DBT limite davantage le *thrashing* comparé à CBJ, il ne le supprime pas définitivement. L'algorithme peut en effet oublier certaines preuves d'inconsistance en remettant

Procédure : DBT -$move(i, a, Q)$

```
1: dc_ia ← i = a ;
2: C ← C ∪ dc_ia ;
3: Q ← {evt(dc_ia)} ;
4: try
5:    propage(Q) ;
6: catch(Contradiction e)
7:    repair(expl(e),Q) ;
8: end try catch
```

Algorithme 10: DBT : prise de décision

Procédure : DBT -$repair(e, Q)$

```
1: if e = ∅ then return no solution ;
2: Q ← ∅ ;
3: dc_jb ← dernièreDécision(e) ;
4: for all (k, v)|dc_jb ∈ expl(k ≠ v) do
5:    D_k ← D_k ∪ {v} ; {restauration}
6:    Q ← Q ∪ {k ◁ v} ;
7: end for
8: try propage(Q) ; {rétro-propagation}
9: catch(Contradiction e) repair(expl(e),Q) ;
10: C ← C\dc_jb ;
11: if e\{dc_jb} ⊆ C then
12:    expl(j ≠ b) ← e\{dc_jb} ;
13:    ¬dc_jb ← j ≠ b ; Q ← {evt(¬dc_jb)} ;
14:    try propage(Q) ;
15:    catch(Contradiction e)
16:       repair(expl(e),Q) ;
17: end if
```

Algorithme 11: DBT : réparation du chemin de décision

en question leurs justifications, puis revenir par la suite dans un état où la justification serait à nouveau valide si elle avait été définitivement conservée en mémoire (la correction d'un tel comportement est l'objet de l'enregistrement de *nogoods* abordé section 3.4).

Exemple 7 *Dynamic backtracking*

Sur le deuxième exemple de *thrashing* (exemple 3 page 24), on peut remarquer que l'explication de contradiction après avoir essayé $x_1 = 1$, $x_1 = 2$, $x_1 = 3$ (et éliminé ces hypothèses par résolution) s'écrit simplement $expl(D_{x_1}) = \emptyset) = \{x_3 = 3, occurence(\{x_1, x_2, x_3\}, 3), x_1 = x_2\}$. On revient donc directement sur $x_3 = 3$ en enregistrant $expl(x_3 \neq 3) = \{occurence(\{x_1, x_2, x_3\}, 3), x_1 = x_2\}$ dont le *nogood* est vide. La valeur 3 est à ce moment définitivement éliminée du domaine de x_3 et ne sera jamais plus considérée par l'algorithme. Le *thrashing*, décrit par les ellipses en pointillés, est donc lui aussi éliminé. Par ailleurs, la décision $x_4 = 1$ est conservée et le chemin de décision à l'issue de la réparation est $x_6 = 4, x_4 = 1$.

En revenant sur la décision la plus récente, DBT et MAC-DBT restent des algorithmes complets (voir [Gin93, Jus97] pour les preuves) *i.e.* qu'ils garantissent de trouver une solution ou d'en prouver l'absence. Il existe d'autres schémas de résolution complets qui ne s'apparentent pas à une recherche arborescente comme *partial order dynamic backtracking* [Bli98]. Néanmoins, la complétude impose toujours plus ou moins une recherche systématique et ordonnée et ne permet pas de se déplacer librement dans l'espace de recherche à l'image d'une recherche locale. Les recherches locales fondent précisément leur efficacité sur cette liberté de mouvement mais sont en général inca-

pables de statuer sur un problème inconsistant. L'algorithme présenté dans la section suivante se rapproche des recherches locales tout en conservant ses racines dans le *backtrack* dynamique et la possibilité de prouver l'inconsistance dans certaines circonstances.

3.3 À la frontière de la recherche locale

Pour conserver la complétude de la recherche, DBT revient toujours sur la décision la plus récente de l'explication de contradiction. Afin d'explorer l'espace plus librement, de ne pas souffrir d'un premier choix initial qui semble à présent bien peu pertinent, on peut relâcher cette exigence et remettre en cause n'importe quelle décision apparaissant dans le conflit. On renonce, ce faisant, à la complétude (l'algorithme peut notamment cycler) mais pas à la possibilité de prouver l'inconsistance du problème (qui provient d'une explication de contradiction vide). Cette idée conduit à de nouvelles approches de résolution et synthétise en même temps toutes celles que nous avons croisées jusqu'à présent. Néamoins, aucune technique ne permet à la fois de se déplacer librement dans l'espace de recherche et de conserver de manière pôlynomiale la mémoire de ce qui a été accompli jusque là [GM94].

3.3.1 Decision-repair, vers un cadre unificateur

Decision-repair (DR) [JL02a] se présente comme un cadre algorithmique générique (présenté aussi dans le cadre PLM (*propagate, learn and move* [JL02b]) employé par de nombreux schémas de résolution. Il offre une lecture simple des idées mises en œuvre dans les algorithmes MAC, MAC-CBJ et MAC-DBT.

procédure : $Decision - repair(\mathcal{C})$

```
1: DC ← un ensemble de contraintes de décision ;
2: loop
3:    if conditions d'absence de solution vérifiées then
4:       return  pas de solution
5:    else
6:       Q ← Q ∪ DC
7:       C' ← propage(Q) ;
8:       switch C'
9:          case contradiction e :
10:            k ← expl(e) ;
11:            DC ← voisinage(DC, k, Γ) ;
12:         case ∀x ∈ V, |D_x| = 1 :
13:            return solution ;
14:         case défaut :
15:            DC ← etendre(DC, Γ) ;
16:   end if
17: end loop
```

Algorithme 12: Decision-repair

DR s'appuie toujours sur la méthode $propage(Q)$ et analyse le résultat du réseau de contraintes C' après propagation (ici $propage(Q)$ fait référence à l'arc-consistance mais n'importe quel type de mécanisme d'inférence peut s'y substituer). Trois cas se présentent :

- ligne 9 : une contradiction a été levée, on calcule son explication et on répare le chemin de décision DC (méthode $voisinage(DC, k, \Gamma)$). Cette réparation peut se voir comme un mouvement dans l'espace de recherche vers un voisin de DC. Elle est libre mais demande de remettre en cause au moins une décision de k pour pouvoir quitter l'impasse courante (DBT revient sur la plus récente contenue dans k, MAC ne se soucie pas de k et revient sur la dernière décision, quant à CBJ, il retire toutes les décisions qui ont eu lieu de k à la décision la plus récente). Le rôle du paramètre Γ est de souligner que certaines informations issues des échecs et de la propagation peuvent être conservées et mises à profit dans la suite de la résolution.
- ligne 12 : si toutes les variables sont instanciées, une solution a été obtenue.
- ligne 14 : enfin dans le cas général, une nouvelle hypothèse est ajoutée à DC (méthode $etendre(DC, \Gamma)$) afin de poursuivre la recherche.

DR s'apparente à une recherche locale qui maintient l'arc-consistance et évolue dans l'espace des configurations partielles (DC). Il est inspiré des travaux [GM94] et a été mis en œuvre avec succès sur des problèmes d'ordonnancement de type *open-shop*. Dans le cas de l'*open-shop*, le paramètre Γ représente une liste tabou (de taille fixée) de *nogoods* (les *nogoods* des explications de contradictions rencontrées) de manière à éviter de cycler trop rapidement en vérifiant avant toute extension (méthode $etendre(DC, \Gamma)$) de retomber dans ces situations d'échecs déjà connues. Il s'agit d'une forme limitée d'enregistrement de *nogoods* qui sera abordée en section 3.4. On peut aussi mentionner que de manière surprenante, certaines instances de DR peuvent s'avérer meilleures qu'une recherche systématique pour prouver l'inconsistance [PV04] bien que DR ne soit pas un algorithme complet.

3.4 Enregistrement de *nogoods*

L'enregistrement de *nogoods* [dK86], mentionné aussi sous le terme d'apprentissage apparaît au début des années 90 [Dec90] parfois dans le cadre des CSPs dynamiques [SV94]. L'idée est de conserver en mémoire le *nogood* des explications de contradiction afin de supprimer définitivement le *thrashing* relatif à ces échecs.

3.4.1 Enregistrement de *nogoods* en PPC

DBT est une forme limitée d'enregistrement de *nogoods* qui conserve uniquement en mémoire les *nogoods* pertinents par rapport au chemin de décision courant (les explications valides). Il s'agit de prolonger cet apprentissage pour supprimer définitivement le *thrashing*. [SV94] présente l'enregistrement de *nogoods* comme la construction d'une *approximate description of the frontier of the space explored* à l'image d'une procédure de génération de coupes en PLNE qui approxime l'espace des points entiers[2]. Le problème

[2]Coupes et nogoods entretiennent donc de nombreux points communs et sont une similitude de plus entre PPC et PLNE.

est que le nombre de *nogoods* est exponentiel (comme le nombre de facettes du polytope des points entiers en PLNE) puisqu'il correspond potentiellement au nombre d'échecs de la recherche arborescente (soit d^n). Il n'est donc pas envisageable de conserver tous les *nogoods*, l'espace étant souvent une ressource plus précieuse que le temps. Afin de borner polynomialement l'espace nécessaire à cet apprentissage, des formes restreintes ont été envisagées :

- *i-order bounded learning* [SV94] propose de ne conserver que les *nogoods* de taille plus petite que i (i étant une constante). Le nombre d'affectations supprimées par un *nogood* est en effet d'autant plus important que celui-ci est petit. La complexité en espace est en $O(n \times d^i)$;
- *i-order relevance bounded learning* [BM96, OJB03] préconise de ne conserver que les *nogoods* qui diffèrent du chemin de décision courant sur plus de i paires de variables-valeurs (c'est donc une généralisation de DBT qui peut être vue comme du *1-order relevance bounded learning*). La complexité en espace reste à nouveau en $O(n \times d^i)$.

Les deux schémas peuvent être combinés de manière à ne conserver que les *nogoods* de taille inférieure à i_1 et supprimer les plus grands quand ils diffèrent de i_2 affectations par rapport au chemin courant. L'enregistrement de *nogoods* n'est pas parvenu à s'imposer en PPC à ce jour mais a eu des retombées surprenantes dans le cadre des problèmes de satisfiabilité booléenne (SAT).

3.4.2 Le cadre booléen et l'approche SAT

Étant donnée une formule propositionnelle, le problème SAT consiste à affecter une valeur de vérité à chaque variable propositionnelle de sorte que la formule soit vraie ou de prouver que la formule est toujours fausse (pour toute interprétation).

3.4.2.1 Les solveurs SAT

Cadre théorique. Les solveurs SAT partagent de nombreuses similarités avec les solveurs de contraintes, on se référera à [BHZ05] pour de nombreux détails sur ce sujet. La formule de départ est mise sous forme normale conjonctive dans laquelle chaque clause peut être vue comme une contrainte[3]. Les solveurs SAT procèdent à une recherche arborescente en réalisant une forme de propagation simple mais très efficace : la propagation unitaire. Ils appliquent en fait une forme incomplète de résolution (voir formule 3.1) entre les clauses introduite par [DLL62] sous le nom de **résolution unitaire** :

$$x, \neg x \vee A \vdash A \tag{3.2}$$

En appliquant ce processus de résolution jusqu'à l'obtention d'un point fixe, on réalise la *Boolean Constraint Propagation* (BCP). En pratique, dès que tous les littéraux d'une clause sauf un sont faux, alors le littéral restant doit être vrai et la variable est propagée pour prendre la valeur correspondante.

[3]Une clause est une disjonction de littéraux et chaque littéral correspond à une variable ou sa négation.

Exemple 8 *BCP*

Après avoir pris les décisions $x = 0, y = 1$, la clause $x \vee \neg y \vee z$ permet à la propagation de déduire $z = 1$.

Cadre algorithmique. Si le principe de la BCP semble simple, l'implémentation incrémentale et efficace de ce schéma est subtile. Il s'agit d'identifier très vite la situation courante d'une clause :

- une clause est dite satisfaite dès que l'un des ses littéraux est vrai, et il devient inutile de la propager par la suite ;
- elle est contredite dès que tous ses littéraux sont faux et le solveur doit revenir en arrière ;
- enfin, c'est une *unit clause* dès que tous ses littéraux **sauf un** sont faux et on peut dans ce cas forcer le dernier littéral à prendre la valeur vraie.

Le maintien de compteurs sur le nombre de littéraux à vrai et faux pour caractériser très vite ces différentes situations à vite été abandonné pour l'utilisation de listes *head/tail* [ZS00]. Il s'agit de maintenir un pointeur sur le premier (*head*) et le dernier (*tail*) littéral non faux (soit vrai, soit dont la valeur de vérité n'est pas connue) de chaque clause. En effet, tant que deux littéraux non faux existent, aucune des situations précédentes n'est possible. Dès que ces pointeurs se rencontrent, la propagation s'enclenche ou une contradiction est levée[4]. Les auteurs du solveur CHAFF [MMZ+01] sont allés plus loin encore en proposant un mécanisme, le *2-literal watching* permettant d'optimiser la restauration de ces pointeurs au *backtrack* et d'éviter l'utilisation de structures *backtrackables* gourmandes en mémoire. Nous ne rentrerons pas ici dans les détails de ce procédé. Ces structures nous intéressent néanmoins particulièrement car elles permettent de propager efficacement les clauses du problème comme les *nogoods* appris pendant la recherche. Nous les présenterons section 7.1.2 dans le cadre de la PPC car c'est l'une des clefs de l'efficacité de l'apprentissage réalisé par ces solveurs. Examinons maintenant comment se comporte un solveur SAT à l'obtention d'une contradiction.

3.4.2.2 Analyse des conflits

À l'obtention d'un échec pendant la BCP s'enclenche une phase d'analyse de conflits (*Conflict Driven Learning*) [SS96, ZMMM01]. Durant cette phase, le point de choix pour le *backjumping* est identifié et un certain nombre de clauses sont ajoutées à la base (les *conflicting clause* ou *nogoods*). Le calcul de l'explication d'échec s'appuie sur le graphe d'implication généré par la propagation unitaire. Ce graphe décrit l'enchaînement des déductions qui s'opèrent pendant la propagation en incluant les hypothèses venant de la recherche. C'est un outil très pratique pour comprendre le traitement effectué par les solveurs SAT.

Exemple 9 *Graphe d'implication*

[4] L'évolution est très semblable au passage de AC4 à AC2001 en PPC. Dans le premier cas, on maintient le nombre de supports de chaque valeur et dans le second, on ordonne les supports et on maintient simplement un pointeur sur le premier support disponible. La différence est que les compteurs sur le nombre de supports doivent être mis à jour en permanence alors que le pointeur d'AC2001 ne bouge que si son support disparaît..

On considère l'instance suivante : Les nœuds désignent à la fois des hypothèses de recherche (nœuds grisés) et des déductions de la BCP. Ils sont donc associés à une variable et une valeur de vérité (0 ou 1). Par ailleurs, le numéro du monde où la décision/déduction est faite est indiqué entre parenthèses. Ainsi $x_1 = 1(4)$ signifie que la décision $x_1 = 1$ est prise au monde numéro 4 (c'est la cinquième décision prise). Chaque arc exprime des liens de causalité révélés par la BCP.

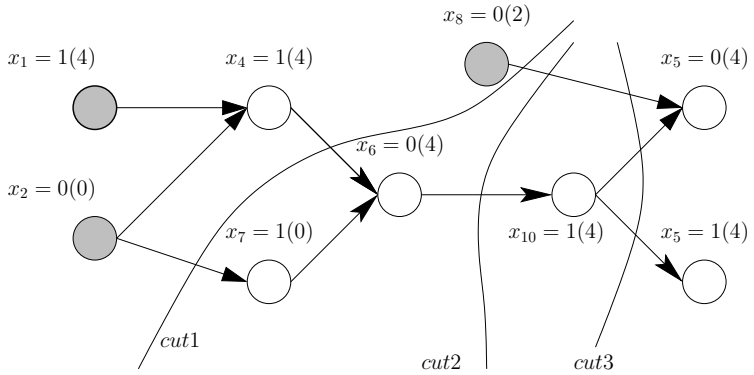


FIG. 3.2: Exemple d'un graphe d'implication

Ainsi $x_8 = 0$ et $x_{10} = 1$ permettent de déduire $x_5 = 0$ (une telle déduction provient de la clause $x_8 \vee \neg x_{10} \vee \neg x_5$). Un conflit apparaît sur le graphe dans la mesure où la BCP a pu déduire à la fois $x_5 = 0$ et $x_5 = 1$.

L'exemple 9 met en évidence un échec sur x_5. On obtient une explication de cet échec en prenant n'importe quelle coupe entre les deux nœuds $x_5 = 1$, $x_5 = 0$ et les décisions $x_1 = 1$, $x_2 = 0$, $x_8 = 0$. Ainsi la coupe comprenant uniquement les décisions $\{x_1 = 1, x_2 = 0, x_8 = 0\}$ est celle qui serait obtenue avec les mécanismes de calcul à la volée des explications. Mais d'autres explications peuvent être obtenues comme $cut_2 = \{x_8 = 0, x_6 = 0\}$, parfois plus précises. Pour l'enregistrement de *nogoods*, l'une d'entre elles, basée sur le *unit implication point* est souvent retenue par les solveurs pour sa précision mais aussi pour guider le *backjump* (section suivante). C'est la coupe qui passe par un point d'articulation (*unit implication point*) entre les décisions et la variable conflictuelle. cut_2 ou cut_3 en sont des exemples. Les clauses apprises par le solveur sont donc des coupes dans le graphe d'implication et représentent des faits parfois établis au bout d'une longue recherche. La différence avec les techniques abordées jusqu'à présent est que l'explication est calculée *a posteriori* car le graphe des implications n'a pas besoin d'être mémorisé. Il est implicitement représenté par l'ensemble des clauses (tous les antécédents d'un sommet s ainsi que s lui-même constituent une clause)[5]. On dispose ainsi de nombreuses explications d'échecs.

L'enregistrement de clauses ou *nogoods* se heurte toujours dans le cas de SAT aux problèmes de mémoire évoqués plus haut et l'ensemble des clauses apprises est périodiquement remis en question pour éliminer les moins précises. Néanmoins, d'autres formes d'enregistrement de *nogoods* existent et ne connaissent pas cette limitation (par

[5]La raison est qu'il y a un mécanisme d'inférence unique, contrairement au contexte non booléen où chaque contrainte encapsule des mécanismes d'inférences évolués.

exemple *Resolution Search* [Chv97]). Les solveurs SAT fonctionnent en *backjumping* et leur schéma d'exploration est fortement relié à l'analyse des conflits.

3.4.2.3 Backjumping en SAT

Le *backjumping* effectué en SAT est différent et peut-être plus malin que celui réalisé en PPC. L'algorithme s'intitule parfois *failure driven assertion* et un lecteur intéressé pourra se référer à [Mit05] pour tous les détails.

La figure 3.3 présente les principes des deux approches à travers les chemins de décision successifs correspondant à l'exploration effectuée à l'issue d'une contradiction. (k) d_i signifie que la décision d_i est prise au monde[6] k. Une contradiction survient après avoir pris k décisions et son explication est $\{d_1, d_2, d_k\}$:

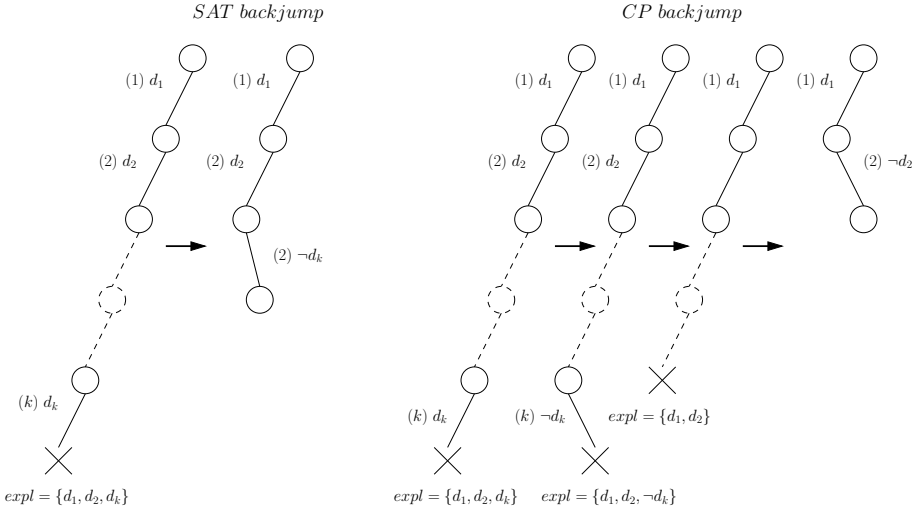


FIG. 3.3: Exemple du processus de *backjumping* en SAT (figure de gauche) et en PPC (figure de droite)

1. Dans le cas de la PPC (figure de droite), on revient sur d_k (la dernière décision du conflit) et on propage $\neg d_k$. On ne peut obtenir un *backjump* immédiatement après avoir pris une décision (sinon la contradiction serait apparue avant sans cette décision). Un *backjump* survient uniquement après la résolution d'explications. Un tel scénario est déroulé sur l'exemple et l'énumération revient au monde (2).
2. Dans le cas de SAT (figure de gauche), une autre stratégie est retenue en effectuant un *backjump* au niveau de l'avant-dernier monde impliqué dans la contradiction et en résolvant le conflit. La décision $\neg d_k$ est ainsi propagée au monde (2) pour faire progresser l'exploration et résoudre le conflit. Les branches de droite et gauche d'un arbre de recherche en SAT portent donc souvent sur des variables différentes. La terminaison d'une telle stratégie est moins évidente. On peut remarquer que toujours plus de décisions (ou déductions) seront ajoutées

[6] Appelé aussi le niveau de décision en SAT.

à des niveaux toujours plus hauts dans l'arbre au fil de l'exploration. Certaines déductions seront définitivement ajoutées au monde 0 jusqu'à ce que la BCP au nœud racine suffise à prouver l'inconsistance.

Ce mécanisme est relié au concept de *unit implication point* et d'*asserting clause*. La phase d'analyse de conflit isole une clause comprenant une seule déduction au monde courant et toutes les autres à des mondes antérieurs. Cette clause est une *asserting clause* et cette déduction unique sera immédiatement propagée après le *backjump* (étant unique, sa négation est impliquée par toutes les déductions antérieures). La clause basée sur la dernière décision est toujours une *asserting clause*.

Exemple 10 *Asserting clause*

Sur le graphe d'implication de la figure 3.2. La première *asserting clause* obtenue est $\{x_8 = 1 \vee x_{10} = 0\}$. Le solveur *bakjump* alors au monde 2 et propage $x_{10} = 0$. On pourrait également utiliser $\{x_8 = 1 \vee x_6 = 1\}$ ou $\{x_8 = 1 \vee x_2 = 1 \vee x_1 = 0\}$ comme *asserting clause*.

Ainsi, la déduction propagée dans la branche droite n'est pas nécessairement la négation de la dernière décision comme c'est le cas sur la figure 3.3 (pour plus de simplicité). L'exploration qui en résulte est clairement différente en PPC si la deuxième explication (obtenue sous le nœud $\neg d_k$) implique des décisions intermédiaires entre les mondes (2) et (k). Aucun *backjump* ne se produit alors en PPC là où un solveur SAT a bien effectué un saut dès la première explication. Notons que ce mécanisme semble parfaitement applicable en PPC. Nous terminerons enfin la présentation des techniques des solveurs SAT en abordant leurs heuristiques de recherche.

3.4.2.4 Heuristiques de recherche

Les heuristiques se focalisent en SAT sur le choix de la variable. Elles sont basées comme en PPC sur l'idée de produire des échecs au plus tôt mais sont également conçues dans un souci de performance de sorte que les critères de sélection soient toujours très rapides à évaluer.

L'originalité des heuristiques en SAT est de s'appuyer sur les *nogoods* appris et de donner la priorité aux variables qui sont apparues dans les conflits récents. Ainsi l'heuristique VSIDS [ZMMM01, GN02, ES04] pour *Variable State Independent Decaying Sum* maintient un compteur (ou impact) par littéral. Ce compteur est initialisé au nombre d'occurrences du littéral correspondant dans les clauses du problème (l'analogue du degré d'une variable dans le réseau de contraintes) et sa valeur est incrémentée à chaque nouvelle clause apprise contenant le-dit littéral. Par ailleurs, les compteurs sont divisés par une constante (de 2 pour CHAFF [ZMMM01] à 4 pour BERKMIN [GN02]) de manière périodique pour faire décroître l'influence des clauses les plus anciennes. Ainsi la préférence est donnée aux clauses récentes. VSIDS choisit pour brancher le littéral qui possède le score ou compteur de valeur maximale.

[GN02] propose d'aller plus loin et de prendre en compte également les clauses utilisées pour la génération de la clause conflictuelle donc implicitement les variables qui participent activement à la propagation unitaire.

Comme ces heuristiques sont indépendantes du chemin de décision courant (*Variable State Independent*), elles sont rapides à évaluer et ne ralentissent pas le solveur pendant la phase de branchement.

Des heuristiques plus récentes existent et explorent davantage les idées de VSIDS tout en conservant l'idée centrale de s'appuyer sur les *nogoods* appris. C'est l'idée que nous souhaitons souligner ici car elle n'a jamais été appliquée en PPC bien que des travaux récents tels que [BHLS04] soient assez proches.

3.5 Synthèse : Look-ahead vs look-back

Le principe central de la programmation par contraintes est le principe d'anticipation mis en œuvre à travers la propagation et les algorithmes de filtrage. C'est le principe de *Look-ahead* (propre des approches prospectives). Par opposition à ce principe, toutes les techniques qui tirent parti des échecs pour revenir à des points de choix pertinents (*intelligent backtracking*) ou pour conserver une information clef provenant d'une longue phase de recherche et de déductions (enregistrement de *nogoods*) sont dites des techniques de *Look-back* (techniques rétrospectives). Le *Look-back*, a cédé le pas devant le développement et le succès des techniques de *Look-ahead* au point d'avoir complètement raté son "transfert technologique" dans les solveurs industriels et commerciaux. Le *thrashing* est la raison d'être du *Look-back* or il est communément admis qu'il existe d'autres moyens plus simples de réduire le *thrashing* sans payer le coût prohibitif des explications :

1. Le *Look-ahead* est précisément le premier moyen si on en croit Haralick [HE80] : *"Look ahead to the future in order not to worry about the past"*. Ainsi plus le niveau de consistance maintenu par le filtrage est élevé, plus le *thrashing* serait réduit. Le passage du *forward checking* à l'arc-consistance comme niveau de consistance standard semble en effet réduire (sans éliminer) en pratique l'intérêt du *backtrack* intelligent (FC-CBJ était ainsi considéré comme le meilleur algorithme avant l'utilisation de l'arc-consistance [SF94]). Ce phénomène pourrait alors expliquer (en partie) le succès des explications dans les solveurs SAT qui maintiennent un niveau de consistance faible dans la mesure où ils perdent toutes les structures combinatoires du problème en le mettant sous forme clausale ;
2. Le deuxième élément plus récent est l'heuristique de recherche. [CvB01] montre qu'il existe toujours une heuristique qui rend CBJ parfaitement inutile[7]. [BR96] introduisent l'heuristique dom/deg et montrent expérimentalement que CBJ n'est pas rentabilisé par le gain d'exploration de l'espace de recherche. Plus récemment encore, les heuristiques qui s'appuient sur les échecs [BHLS04, Ref04] semblent également capables de détecter les problèmes de *thrashing* en isolant les sous-problèmes les plus durs pour les traiter en haut de l'arbre de recherche. Le *2-way branching* semble également une manière de limiter le *thrashing* en offrant plus de souplesse sur le choix de la variable.

À l'évidence, les relations entre *Look-ahead* et *Look-back* ne sont pas simples. Sont-ils orthogonaux ? Sont ils antagonistes ? la réponse n'est pas évidente. Pour répondre à ces questions, il manque avant tout une définition formelle du *thrashing*. Il y a en

[7] Une telle preuve n'a toujours pas été faite pour DBT.

pratique des résultats très positifs pour les explications. Les benchmarks de [CvB01] montrent par exemple que le backjumping, même couplé à l'arc-consistance (MAC-CBJ) peut se révéler critique. DR (qui est fortement lié à MAC-DBT) fait partie des meilleurs algorithmes pour résoudre les problèmes d'*open-shop* en ordonnancement [JL02a, GJP00].

Il y a néanmoins une observation sur laquelle tout le monde semble s'accorder : le *Look-back* joue un rôle important dans le cas de **problèmes structurés**, de problèmes réels :

- si les solveurs SAT peuvent résoudre des problèmes industriels de centaines de milliers de variables, ils sont mis en échec par des problèmes aléatoires parfois de quelques centaines de variables ;
- du côté contraintes, MAC-CBJ ne fait que ralentir MAC pour résoudre les problèmes aléatoires durs à la transition de phase [CKT91]. Il semble en revanche très utile pour résoudre certains problèmes qui apparaissent exceptionnellement avant la transition de phase et semblent bien plus difficiles que les problèmes au pic [HW94, GW94, Smi95].

Il semble donc important de comprendre ce que sont les structures d'un problème et de comprendre leurs relations avec le calcul des explications.

Chapitre 4

Des approches pour des problèmes structurés

Ce chapitre tente de clarifier la notion de structure dans les problèmes combinatoires tout en la mettant en perspective avec la difficulté pratique des problèmes. Nous nous attacherons à montrer en quoi les approches du chapitre précédent basées sur les explications peuvent se montrer efficaces sur certaines structures. Néanmoins ce ne sont pas les seules approches existantes et nous présenterons deux autres types de techniques basées sur la décomposition qui permettent de traiter de tels problèmes.

4.1 Les problèmes structurés

Nous abordons dans un premier temps les études menées sur les problèmes aléatoires pour caractériser les problèmes difficiles et montrer comment la difficulté pratique des problèmes a été étudiée.

4.1.1 Étudier la difficulté des problèmes

Transition de phase : De nombreux problèmes NP-complets peuvent être définis par rapport à un ou plusieurs paramètres (*order parameters*) et de nombreuses techniques pour générer aléatoirement ces problèmes s'appuient sur ces paramètres structurels. Il s'agit par exemple :

- de la connectivité moyenne d'un graphe en coloration ;
- du nombre de clauses portant sur chaque variable en SAT ;
- ou de la dureté et densité du réseau de contraintes pour les CSP.

Les problèmes difficiles se rencontrent pour des valeurs particulières de ces paramètres [CKT91]. De plus, ces valeurs constituent la frontière entre deux régions distinctes. La première région est sous-contrainte, la densité des solutions est donc élevée et le problème relativement facile. La deuxième est sur-contrainte et contient très peu ou pas de solutions. Les problèmes sont également faciles dans cette région, l'exploration étant élaguée très tôt puisque l'espace de recherche est fortement contraint. Entre les deux régions se trouvent des problèmes nettement plus difficiles. La probabilité d'une solution est faible mais pas négligeable. De nombreuses affectations partielles constituent presque des solutions et tous ces *minimums locaux* empêchent d'élaguer la recherche

suffisamment tôt. En étudiant l'existence de solution en fonction d'un paramètre structurel et lorsque la taille du problème tend vers l'infini, on peut exhiber une **transition de phase**. L'observation des temps de résolutions associés à la transition de phase est intéressante [CKT91] comme le montre la figure 4.1 pour des CSP aléatoires binaires de 30 variables, 10 valeurs, une densité (pourcentage de contraintes présentes dans le réseau) de 40% et une dureté p_2 variable (pourcentage de tuples interdits dans les contraintes). Dans la zone de la transition de phase où tous les problèmes passent soudainement de la consistance à l'inconsistance se trouvent les problèmes les plus difficiles.

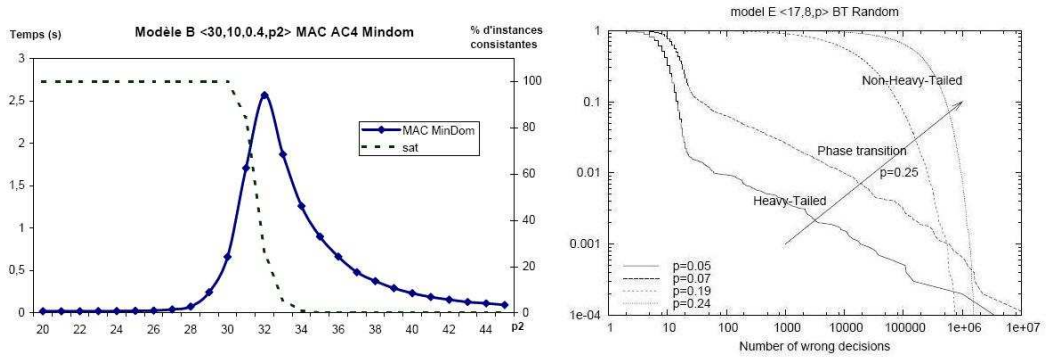


FIG. 4.1: (a)-graphique de gauche- La transition de phase dans les CSP aléatoires modèle B (expérimentations réalisées avec choco en MAC-AC4). Le temps de résolution moyen est donné pour la résolution de 200 instances à chaque valeur de dureté p_2. (b)-graphique de droite- La probabilité $f(x)$ d'une exécution d'une recherche aléatoire de terminer en moins de x backtracks (*wrong decisions* sur le graphique) pour différentes valeurs de dureté p et des CSP aléatoires modèle E. (Graphique provenant de [GFSB04]).

Cette conclusion de [CKT91] largement admise fut remise en question par la suite par de nombreux chercheurs qui observèrent des problèmes exceptionnellement difficiles nettement avant la transition de phase [HW94, GW94, Smi95]. Par ailleurs, en étudiant la complexité des problèmes de cette manière, on ne fait pas abstraction de l'algorithme utilisé pour les résoudre (un problème difficile pour une approche ne l'est pas nécessairement pour une autre). Afin d'éviter ce biais, d'autres chercheurs [GFSB05, HO06, Mil03] se sont penchés sur la distribution des temps de résolution pour une recherche aléatoire répétée sur une même instance.

Distribution des temps de résolution : [GFSB05, HO06, Mil03] ont montré que cette distribution changeait de régime avant la transition de phase en passant d'un comportement *heavy tailed* pour les problèmes sous-contraints à une distribution normale à la transition de phase. Une distribution *heavy tailed* se caractérise par une variance très importante qui passe, à priori, pour un comportement erratique ou pathologique. Ce type de distribution est utilisé pour modéliser de nombreux phénomènes naturels et reflète ici la criticité des premiers choix de la recherche aléatoire. Sa variance ne se stabilise qu'une fois la transition de phase atteinte. La figure 4.1 montre le phénomène (le

graphique est tiré de [GFSB04]) en affichant la probabilité $P(X > x)$ d'une exécution de se terminer en moins de x *backtracks*. Quand le problème est extrêmement facile, la queue décroît très rapidement (absent du graphique), puis en se déplaçant vers la transition de phase, la distribution devient clairement *heavy-tailed* et la queue adopte un comportement linéaire (l'échelle est logarithmique et la distribution adopte donc une décroissance hyperbolique caractéristique de ce type de distribution). Enfin, en atteignant la transition de phase, les problèmes deviennent uniformément difficiles, l'interaction avec l'algorithme de recherche diminue et la queue décroît à nouveau très rapidement (même si elle est maintenant nettement plus épaisse). La queue est la plus *épaisse* à la transition de phase mais elle est plus *large* dans le cas sous-contraint où il existe une probabilité (certes très faible) de réaliser une exploration encore plus coûteuse qu'à la transition de phase dans le pire des cas. Ces résultats illustrent bien l'importance des choix initiaux dans le cas sous-contraints et sont peut-être à mettre en rapport avec l'existence de problèmes exceptionnellement difficiles identifiés en amont de la transition de phase. L'interaction avec l'algorithme de recherche est donc critique dans la région sous-contrainte ce qui pourrait expliquer qu'on y ait parfois observé des problèmes exceptionnellement difficiles.

D'après [HW94], ce n'est pas le cas, il existe des problèmes **intrinsèquement** plus difficiles à cet endroit qu'à la transition de phase. [HW94] met ainsi au jour sur des problèmes de coloration, une double transition de phase. [GW94, Smi95] ont également rencontré ces problèmes dans le cadre des CSP. Ils sont caractérisés par [Smi95] comme :

- des problèmes souvent inconsistants (mais pas uniquement, [SG95]) là où les problèmes sont massivement consistants ;
- une mauvaise distribution des solutions dans l'espace de recherche, notamment pour des solutions regroupées en *clusters*. Une répartition homogène donne ainsi plus de chance à l'algorithme de recherche d'identifier une solution sans avoir à examiner l'ensemble de l'espace de recherche tandis qu'une répartition en *clusters* rend critique l'ordre d'exploration ;
- plus fréquent pour des réseaux de contraintes peu denses [SG95].

On peut se demander si ces problèmes exceptionnellement difficiles ont une topologie, une **structure** vraiment distincte de leurs congénères plus simples pour des mêmes valeurs de densité et de dureté.

4.1.2 De la complexité théorique à la complexité pratique

Un problème est souvent dit **structuré** si ses composantes (variables et/ou contraintes) ne jouent pas toutes le même rôle, ou ne revêtent pas la même importance au sein du problème. Les stratégies de relaxation ou décomposition exploitent ainsi le fait qu'un problème sous-jacent (ou une partie du problème originel) peut être traité comme un problème classique (flot compatible ou maximum, plus courts chemins, sac à dos, etc.) pour lequel on dispose d'algorithmes efficaces et bien connus. Cet aspect du problème est souvent désigné sous le terme de structure.
Il existe deux concepts introduits dans le contexte des problèmes de satisfiabilité SAT qui formalisent davantage la notion de structure, les *backbones* [MZK+99] et les *backdoors* [WGS03b, WGS03a].

1. Un *backbone* désigne un ensemble de littéraux qui sont vrais dans toute affectation vérifiant la formule. Plus généralement, il s'agit d'un ensemble d'affectations qui font partie de toutes les solutions. La taille du *backbone* est associée à la difficulté du problème. Il n'y a en effet qu'une seule façon d'instancier correctement les variables du *backbone* ;

2. Un *backdoor* fait (informellement) référence à un ensemble de variables qui encapsulent toute la combinatoire du problème. Une fois un tel ensemble instancié, le reste du problème peut être résolu efficacement et rapidement. Idéalement, le reste du problème est polynômial (*strong backdoors*). Autrement dit, la difficulté du problème réside dans son *backdoor* puisque pour un *backdoor* de taille k dans un problème de taille n, le problème est résolu en d^k appels à une procédure polynomiale (au lieu de d^n). La résolution devient très efficace pour des petites valeurs de k et particulièrement dès que k est en $log(n)$. On peut noter également que la définition de *backdoor* est implicitement dépendante de l'algorithme utilisé pour la résolution.

Liens avec la difficulté des problèmes : Les concepts de *backdoor* et *backbone* sont effectivement reliés à la difficulté des problèmes :

- De larges *backbones* indiquent que les solutions sont regroupées en *clusters* et reflètent ainsi l'une des caractéristiques soulignées dans [Smi95] ;
- De tous petits *backdoors* rendent polynômial en pratique, des problèmes NP-Complets dans le cas général. Ils proposent une explication aux performances des solveurs SAT sur les problèmes industriels. Ce concept est utilisé pour expliquer le succès de deux stratégies de ces solveurs :
  - des stratégies de *randomization* et *restart* qui donnent ainsi l'opportunité d'identifier un petit *backdoor*. Le *restart* permet notamment d'éliminer dans certain cas les distributions *heavy-tailed* présentées précédemment. [WGS03a] montre d'ailleurs sur des instances SAT réelles que des *backdoors* de très petites tailles existent et sont exploités par les solveurs,
  - Les techniques de *Look-back* qui semblent également désignées pour traiter des problèmes possédant ce type de structure. Bayardo [BS96] propose ainsi de générer des instances SAT très difficiles en cachant un petit sous-problème insatisfiable dans un grand problème sous-contraint (celui-ci constitue ainsi un véritable *backdoor*). Il exploite ainsi une autre caractérisation de ces instances exceptionnellement difficiles soulignées dans [Smi95] : un problème insoluble là où presque tous les problèmes sont solubles. Les solveurs SAT de l'époque sont ainsi incapables de résoudre ces problèmes sans utiliser des techniques de *Look-back*. Il obtient ainsi une conclusion qui sera faite à nouveau par [SG95] dans le cadre des CSP en montrant que CBJ est critique pour résoudre ces instances très difficiles avant la transition de phase.

Les *backdoors* proposent une explication à la différence qui réside entre complexité théorique et pratique. Ils mettent un nom (et proposent un formalisme) sur un très vieux concept en optimisation combinatoire exploité dans les trois communautés SAT, RO ou PPC sous les termes de *variables complicantes*, *impliquants premiers*, *independant variables*, etc.

Liens avec les explications : Les explications offrent la possibilité à la recherche de revenir et se focaliser sur la source des échecs. Les problèmes de Bayardo ou les problèmes peu denses de Smith sont des problèmes qui induisent un *thrashing* très important et un ordre critique d'exploration. Une recherche qui commencerait par instancier toutes les variables en terminant par celles du petit problème inconsistant de Bayardo serait contrainte de prouver l'infaisabilité de ce sous-problème pour toutes les affectations (ou presque puisque le reste du problème est très sous-contraint) du reste des variables du problème. Une technique de *Look-back* identifierait tout de suite qu'aucune ou presque des explications d'échecs n'impliquent les autres variables et s'abstiendrait de backtracker à leur niveau. Le même phénomène se produit dans les graphes peu denses. L'exemple 2 de *thrashing* (page 23) est précisément un exemple construit sur ce principe. Sur un graphe peu dense, on peut rapidement séparer le problème en morceaux indépendants et effectuer une mauvaise exploration en commençant par exemple par des morceaux consistants alors qu'il existe des parties inconsistantes qui n'entretiennent plus à ce stade de la recherche de relations avec le reste du problème. Les explications ne sont néanmoins pas la seule manière de traiter ce type de problèmes structurés.

4.2 Alternative aux explications pour des problèmes structurés

Nous présentons ici deux types d'approches conçues pour la résolution de problèmes structurés. La première est issue de la communauté contraintes et la deuxième provient du monde de la recherche opérationnelle. Toutes les deux sont des techniques de *backdoors* et s'adressent à des problèmes structurés.

4.2.1 Les méthodes de décomposition structurelle

Les problèmes de satisfaction de contraintes sont connus pour être NP-complets dans le cas général. Néanmoins, les communautés en intelligence artificielle et base de données ont beaucoup travaillé pour identifier des classes traitables de CSP. [GLS00] synthétise la nature des CSP traitables selon deux grands axes :

- une structure restreinte du réseau de contraintes ;
- une sémantique restreinte des contraintes.

Les méthodes de décomposition structurelle se sont intéressées aux CSP traitables selon le premier axe, autrement dit, pour des raisons topologiques. Ces techniques reposent directement ou indirectement sur le fait qu'un réseau acyclique de contraintes peut être résolu de manière polynômiale. Les propriétés structurelles exploitées par ces décompositions s'appuient toujours sur une généralisation de l'acyclicité et introduisent toutes un paramètre w caractéristique de l'acyclicité. Pour les techniques de *Tree Clustering* (TC) [DP89] ou décomposition arborescente, w désigne la largeur d'arbre du réseau (voir définition 4.2.2). Pour les stratégies de *cycle cut-set* [DP87], w désigne la cardinalité de l'ensemble coupe-cycle minimal du réseau (le nombre minimal de variables à instancier pour rendre le réseau acyclique). Ces techniques proposent ainsi des complexités théoriques (en temps et en espace) exprimées en fonction de w qui sont meilleures que la complexité des approches envisagées jusqu'à présent (en $O(ed^n)$

où e est le nombre de contraintes, d la taille maximum des domaines et n le nombre de variables). Une décomposition arborescente consiste de manière informelle à effectuer des regroupements de variables qui s'agencent de manière arborescente [JNT05, JT05]. [GLS00, DF01, JNT05, JT05] définissent ainsi la décomposition arborescente (la figure 4.2 en présente un exemple) :

Définition 4.2.1 *Soit un graphe $G = (V, E)$ et une paire (C, T), telle que $T = (I, F)$ soit un arbre et $C = \{C_i : i \in I\}$ une famille de sous-ensembles de V telle que chaque C_i soit un élément de I. Chaque C_i est ainsi associé à un nœud de T et représente un sous-ensemble de V. La paire (C, T) est une* ***décomposition arborescente*** *de G si elle vérifie :*

- $\bigcup_{i \in I} C_i = V$;
- *pour chaque arc $\{x, y\}$ de E, il existe $i \in I$ avec $\{x, y\} \subset C_i$;*
- *pour $i, j, k \in I$ si k est sur un chemin de i à j dans T alors $C_i \bigcap C_j \subseteq C_k$.*

Définition 4.2.2 *Soit (C, T) la décomposition arborescente d'un graphe G. La* ***largeur d'arbre*** *de G, w, est reliée à la taille maximum des regroupements de variables : $w = max_{i \in I}|C_i| - 1$*

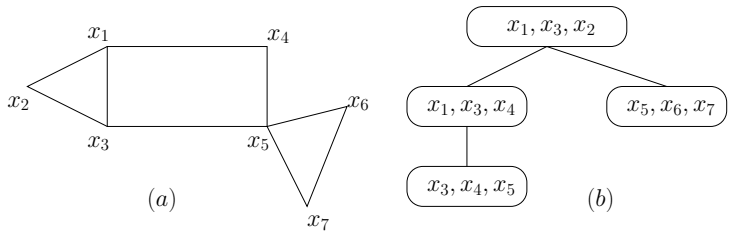


FIG. 4.2: Exemple d'un réseau de contraintes (a) et d'une décomposition arborescente (b) de largeur d'arbre 2.

Parmi les techniques récentes de décomposition, on trouve la méthode BTD [JT03] (*Backtracking with Tree-Decomposition*) qui exploite la décomposition arborescente de manière à explorer une seule et unique fois chaque regroupement de la décomposition une fois que leur consistance (notion de *goods*) ou inconsistance (notion de *nogoods*) est établie. Chaque regroupement est résolu indépendamment des autres, la complexité temporelle théorique est de $O(nd^{w+1})$. En revanche, la méthode implique une complexité en espace nettement supérieure aux techniques de backtracking classiques car il faut conserver des informations sur la résolution de chaque regroupement de variables.

La méthode originelle TC calcule ainsi toutes les solutions de chaque regroupement et mémorise les affectations (tuples) qui portent sur les intersections entre clusters. Ces tuples définissent les contraintes d'un CSP équivalent acyclique dont les variables sont constituées par les nœuds de l'arbre issus de la décomposition arborescente. Ce CSP est ensuite résolu en temps linéaire car il est acyclique. TC construit ainsi un CSP équivalent mais acyclique à partir d'une couverture de l'ensemble des contraintes.

L'ensemble coupe-cycles d'une approche de type *cycle cut-set* constitue typiquement un *backdoor* du problème. Les propriétés topologiques du réseau de contraintes exploitées par TC ne semblent pas directement corrélées avec le concept de *backdoors*. Néanmoins, ces techniques ont précisément pour objectif de traiter les réseaux de

procédure : Algorithme de Benders

1. 1. Résoudre (PM) à l'optimum :
2. 1.a Si (PM) est infaisable alors **return** Pas de solution ;
3. 1.b Si (PM) est non borné alors (P) l'est aussi **return** Problème non borné ;
4. 1.c Sinon (y^*, z^*) est la solution optimale de (PM). **Aller en 2**.
5. 2. Résoudre (PED) à l'optimum :
6. 2.a Si (PED) est infaisable donc (PE) est non borné
7. **return** Problème infaisable ou non borné.
8. 2.b Si (PED) à une solution optimale u^* (un point de son polytope) tel que
9. $z^* < f(y^*) + u^*(a - g(y^*))$ alors on ajoute $z \geq f(y) + u^*(a - g(y))$
10. à PM et **on revient en 1**.
11. 2.c Si (PED) n'est pas borné, (PE) est infaisable. Dans ce cas, le simplexe,
12. au lieu de fournir un point du polytope de (PED), fournit un rayon
13. extrême u^*. on ajoute alors $u^*(a - g(y^*)) \leq 0$ à (PM) et **on revient en 1**.
14. 2.d Sinon (PED) à une solution optimale u^* tel que $z^* \geq f(y^*) + u^*(a - g(y^*))$
15. alors (y^*, x^*) est une solution optimale. **return** (y^*, x^*) ;

Algorithme 13: Algorithme de décomposition de Benders

contraintes peu denses dont il était question précédemment pour éviter les phénomènes de *thrashing* en ne résolvant qu'une et unique fois des parties bien choisies du problème. Contrairement au *Look-back*, ces techniques offrent des garanties théoriques de complexité mais se restreignent aux structures visibles dans le réseau de contraintes. Il s'agit de structures topologiques et non combinatoires qui reposent elles sur la sémantique des contraintes. Les problèmes de Bayardo ne sont, par exemple, pas traitables par ce type d'approche.

4.2.2 Décomposition de Benders

La décomposition de Benders [Ben62] est une technique beaucoup plus ancienne (1962) et introduite dans un contexte linéaire. Elle met en œuvre une stratégie de résolution fondée sur le partitionnement du problème selon ses variables : (y, x). Elle est introduite à l'origine dans le cadre de problèmes linéaires mixtes possédant à la fois des variables entières et continues. On peut la voir comme une forme d'apprentissage par l'échec. Elle s'utilise sur les problèmes ayant la forme suivante :

$$\begin{aligned} &\text{P : Min } f(y) + cx \\ &\text{Tel que : } g(y) + Ax \geq a \\ &\qquad\qquad y \in D, x \geq 0 \end{aligned}$$

L'algorithme 13 récapitule la procédure et procède en deux temps. Nous décrivons d'abord les principes généraux de l'algorithme qui correspondent à ces deux étapes (1.c) et (2.b).

Un problème maître PM est utilisé pour traiter un sous-ensemble de variables y et constitue donc toujours une relaxation de P. Puis, un problème esclave PE complète l'affectation sur x pour obtenir une solution complète à P. L'idée est alors d'interdire dans PM la solution obtenue après la résolution de PE ainsi que toutes celles qui seraient sous-optimales pour les **mêmes raisons** que cette solution particulière. En

résolvant successivement PM et en interdisant à chaque itération des **classes entières** de solutions, l'algorithme converge vers la solution optimale du problème complet P.

À chaque itération PE produit donc une coupe dite de Benders (2.b) ajoutée au problème maître. La coupe est de la forme :

$$z \geq h(y)$$

C'est l'élément clef du processus, elle est fournie par le problème esclave et la résolution de son dual. Considérant une affectation y^* donnée par le maître, le problème esclave PE s'écrit avec son dual PED :

$$\begin{array}{ll} \text{PE : Min } cx & \text{PED : Max } u(a - g(y^*)) \\ \text{Tel que : } Ay \geq a - g(y^*) & \text{Tel que : } uA \leq c \\ x \geq 0 & u \geq 0 \end{array}$$

La dualité nous assure que $cx \geq u(a - g(y^*))$, $u(a - g(y^*))$ est donc une borne inférieure de cy. Comme la faisabilité du dual est indépendante de y^*, l'inégalité suivante est bien valide : $f(y) + cx \geq f(y) + u(a - g(y))$. Il est critique d'observer que y n'apparaît pas dans les contraintes de PED. Autrement dit, cette valeur optimale de u constitue une solution faisable pour le dual quel que soit y. Ainsi chaque point du polytope du problème dual peut potentiellement constituer une coupe de Benders (ce sont les contraintes de P non explicitées dans PM). Cependant, à l'optimum, le nombre de contraintes saturées (actives) ne dépasse pas $m + 1$ où m est le nombre de variables y du maître et le problème peut être résolu en énumérant un nombre réduit de ces contraintes. C'est naturellement l'intérêt d'une procédure de génération dynamique de contraintes. Par ailleurs, le théorème de la dualité assure en plus que la valeur optimale u^* maximisant $u(a - g(y^*))$ est aussi l'optimum de PE. À partir d'une affectation particulière de y, la coupe constitue donc une inégalité valable pour tout y et élimine non seulement la solution particulière qui a permis sa déduction mais aussi toute une classe de solutions impossibles pour les mêmes raisons. Le problème maître s'écrit donc à la I^{eme} itération :

$$\begin{array}{ll} \text{PM : Min } z & \\ \text{Tel que : } z \geq f(y) + u_i^*(a - g(y)) & \forall i < I \end{array}$$

Sur ce schéma général, différentes situations particulières doivent être prises en compte :

- PM est infaisable ou non borné (cas 1.a, 1.b) : comme PM est toujours une relaxation de P, P est également infaisable ou non borné.
- PED est non borné (cas 2.c) : Dans ce cas, il existe un rayon extrême u^* tel que $u^*(a - g(y^*)) > 0$ c'est à dire une direction améliorante le long de laquelle $u(a - g(y^*))$ peut croître indéfiniment. On ajoute dès lors la coupe $u(a - g(y^*)) \leq 0$;
- La terminaison (cas 2.d) : l'algorithme s'arrête dès lors que la solution optimale du maître vérifie la contrainte obtenue après résolution de PED. En effet, une condition nécessaire et suffisante d'optimalité est que (y^*, z^*) satisfasse toutes les contraintes de P non explicitées dans PM. Or si (y^*, z^*) respecte la contrainte ajoutée après la résolution de PED, cette solution les respecte toutes puisque le PED est résolu à l'optimum (le terme $u^*(a - g(y^*))$ est maximum donc $z^* \geq$

$f(y^*) + u^*(a - g(y^*))$ est aussi vrai pour tous les autres points du polytope qui induisent un $u^*(a - g(y^*))$ plus petit). On peut ensuite à partir du théorème des écarts complémentaires ([Chv97]) obtenir à partir de la solution optimale du PED en u^*, la solution optimale de PE y^* : (y^*, x^*) constitue alors la solution optimale du problème complet.

Remarquons finalement que l'algorithme fournit à chaque itération à la fin de (2.b) un encadrement de l'optimum opt : $z^* \leq opt \leq f(y^*) + cx^*$ ce qui est toujours intéressant si on interrompt la résolution avant la fin de la convergence.

Autre forme de dualité : Il faut donc être en mesure de définir les variables duales (multiplicateurs) au sens de la dualité classique pour pouvoir appliquer la décomposition. L'utilisation d'une forme de dualité généralisée est faite dès 1972 par [Geo72]. Nous nous intéressons davantage à [HO03] qui propose également de pallier cette limitation et d'élargir la notion de dual couramment utilisée en introduisant un *inference dual* pour tout type de sous-problème. Il se place dans un cadre plus général : une décomposition de Benders s'appuyant sur la logique. La dualité se réfère alors à la capacité de produire une preuve, la preuve logique de l'optimalité du sous-problème et de la validité de la coupe qui peut prendre une forme différente de l'inégalité sur y. Cette preuve est obtenue dans le cas classique à travers les théorèmes de la dualité qui garantissent que la solution du dual est une borne inférieure du primal (en cas de minimisation du primal) et que si les deux problèmes ont des solutions alors ils ont même optimum.

Applications : La décomposition de Benders possède d'évidentes applications en tant que technique de partitionnement dès que deux groupes de variables jouent des rôles radicalement différents. La programmation linéaire mixte (application originelle) ou la résolution de problèmes partiellement non linéaires constituent souvent ses applications naturelles puisqu'on peut séparer les parties linéaires et non linéaires du problème et faire ainsi appel à des algorithmes dédiés pour un problème maître non linéaire. Pour cette raison, la décomposition a souvent été mise en œuvre comme un cadre de coopération entre Recherche Opérationnelle et Programmation par Contraintes [JG01, HOTK00, BGR02, EW01]. Ces méthodes dites hybrides ont démontré leur efficacité dans une grande variété d'applications parce qu'elles exploitent la présence de *backdoors* dans ces problèmes. Goeffrion en 1972 présente la méthode de Benders comme une technique de *backdoors* :

> *"J.F Benders devised a clever approach for exploiting the structure of mathematical programming problems with* ***complicating variables*** *(variables which, when temporarily fixed, render the remaining optimization problem considerably more tractable).".*

L'efficacité de Benders réside précisément dans la simplicité des sous-problèmes une fois le maître affecté mais aussi sur la qualité de l'information duale qu'on peut extraire de la résolution des sous-problèmes autrement dit, de la précision des coupes.

4.3 Synthèse

Nous avons cherché ici à définir ce qu'on entend couramment sous le terme de "problème structuré" tout en expliquant comment les explications permettent d'exploiter la présence de telles structures (formalisées en terme de *backbones* et *backdoors*) pendant la résolution. Les liens entre l'étude de la complexité des problèmes et celle de leurs structures révèlent la différence qui réside souvent entre complexité théorique et pratique. Deux stratégies qui ne relèvent pas des explications et sont aussi capables de tirer parti de ces structures ont également été présentées ainsi que leurs limitations :

1. les méthodes de décomposition structurelle exploitent uniquement des structures statiques présentes dans le réseau de contraintes et basées sur son acyclicité ;
2. la décomposition de Benders s'adresse à des problèmes en partie linéaires (pour les sous-problèmes) dans la mesure où le calcul de la coupe de Benders repose sur la dualité linéaire.

Les explications offrent d'une certaine manière la possibilité de s'affranchir de ces limites tout en continuant à bénéficier des structures. Elles font d'une part abstraction du réseau de contraintes car elles proviennent de la sémantique des contraintes. D'autre part, elles jouent le rôle d'une information duale qui permet d'identifier une coupe ou un *nogood* à l'issue de la résolution du problème esclave dans la décomposition de Benders et de généraliser la preuve à un ensemble plus large que la solution particulière du maître. C'est ce dernier point que nous approfondirons en détail chapitre 8.

Chapitre 5

Explications prospectives et rétrospectives

Les explications ont été inventées dans le contexte de l'intelligence artificielle comme une technique de raisonnement clef (abduction) chez l'être humain (devant l'induction et la déduction) ; elles ont mûri pour l'optimisation combinatoire au sein de la programmation par contraintes et n'ont réalisé à ce jour leur "transfert technologique" que dans la communauté SAT. C'est en partie ainsi que progresse la recherche scientifique et l'innovation technologique, par un échange permanent d'idées qui mûrissent dans des cadres différents. Il faut de nombreux baptêmes à une seule idée pour lui permettre d'atteindre sa maturité. Le tableau que nous avons dressé dans cet état de l'art mène à deux grands constats qui motivent notre thèse :

1. Les explications sont un élément incontournable en SAT aujourd'hui. Pourquoi n'est-ce pas le cas en PPC ? Les solveurs SAT savent exploiter l'information contenue dans les explications sur trois aspects, les heuristiques de recherche, le filtrage (*Look-ahead*) à travers l'apprentissage de clauses et le backtrack intelligent (*Look-back*) avec le *backjumping*. Seul le troisième élément a été vraiment exploré en PPC. Nous pensons que les explications sont sous-exploitées à l'heure actuelle en PPC même par les techniques les plus pointues comme *decision repair* qui fondent entièrement leurs principes de fonctionnement sur les explications. La double utilisation *Look-back/Look-ahead* qu'en font les solveurs SAT n'est pas directe en PPC. Il semble que la nature des explications ne soit pas exactement la même dans les deux cas et que deux types d'explications soient en jeu :
   - des explications rétrospectives, c'est à dire des explications projetées en permanence sur les décisions courantes pour coordonner les mouvements dans l'espace de recherche.
   - des explications prospectives, c'est à dire des explications qui encapsulent une classe d'affectations (futures) la plus large possible et représentent de manière compacte une inférence provenant d'une longue suite de déductions et de recherches.

   Les deux objectifs s'accordent plus facilement dans le cadre simplifié SAT et il manque encore au cadre CSP une bonne approche pour les conjuger efficacement. De nombreuses questions se posent à ce niveau. Par exemple, est-ce que ces deux types d'explications sont rattachés à des modes de calculs différents (à la volée

ou *a posteriori*) ?, à une gestion différente au sein du solveur ? Dans tous les cas, c'est ce premier constat qui fonde notre effort de recherche sur les deux axes : heuristiques de recherche et nouvelles techniques d'enregistrement de nogoods.

2. Nous pensons par ailleurs que les explications sont apparentées à une vieille notion de l'optimisation combinatoire, la dualité. La dualité s'est épanouie en Recherche Opérationnelle où elle joue un rôle central, à la fois pour calculer des bornes, pour guider la recherche mais aussi pour résoudre des problèmes structurés. Nous avons abordé la décomposition de Benders précédemment car il en sera question dans cette thèse mais la relaxation lagrangienne est aussi un très bon exemple d'utilisation de la dualité pour d'autres types de structures (contraintes couplantes). De ce point de vue, la dualité doit être mise en rapport avec les explications. Les explications s'adressent à des problèmes structurés et constituent une information duale dans la mesure où elles sont une justification de l'état courant atteint par le solveur. Nous pensons que les explications ont un rôle à jouer dans la coopération ou l'hybridation avec d'autres types de résolution. C'est ce deuxième constat qui motive nos investigations sur l'utilisation des explications dans un contexte de décomposition.

Deuxième partie

Des explications au cœur des solveurs

Chapitre 6

Structures du problème et heuristiques génériques

Les stratégies de recherche générique pour résoudre des problèmes d'optimisation combinatoires constituent le Graal pour les communautés Recherche Opérationnelle (RO) et PPC (Programmation par Contraintes) car elles réduisent l'expertise nécessaire à la prise en main d'un solveur. Différentes pistes sont explorées à l'heure actuelle : adapter dynamiquement la manière dont le solveur s'emploie à résoudre un problème, identifier des structures spécifiques à une instance particulière pour accélérer la recherche, etc. Dans tous les cas, la clef est de pouvoir identifier, comprendre et utiliser les structures intimes présentes dans les instances d'un problème combinatoire donné [BCS01, Ref04, WGS03a, WGS03b]. Nous nous intéressons ici aux relations entretenues par les variables du problème. L'objectif est d'identifier et de différencier à la fois les structures dynamiques (créées par l'algorithme de recherche) et statiques (relatives à une instance) du problème. Nous avons abordé la notion de structure en mettant en évidence les liens avec la complexité pratique des problèmes au chapitre 4.1. Une structure est donc entendue ici comme un sous-ensemble de variables qui joue un rôle spécifique au sein du problème. Nous définissons à cette fin, plusieurs mesures d'impacts ainsi que le graphe de relations leur correspondant de manière à pouvoir :

- reconnaître les structures cachées (invisibles dans le réseau de contraintes) ;
- concevoir des stratégies de recherche génériques.

Ce chapitre est organisé de la manière suivante : la section 6.1 introduit le contexte et le but de cette étude ; différentes mesures d'impact et leur graphe associé sont présentés en section 6.2 en distinguant leur capacité respective à refléter des structures dynamiques et statiques sur un exemple concret. Finalement, nous essayons d'évaluer leur intérêt comme heuristique de recherche générique en section 6.3.

6.1 But et contexte de l'étude

6.1.1 Contexte : recherche basée sur l'impact

[BHLS04] propose en 2004 d'exploiter les échecs rencontrés au cours de la résolution pour déterminer les contraintes critiques et implicitement les variables les plus contraintes. Un poids associé à chaque contrainte augmente dès qu'elle est à l'origine

d'une contradiction. Le degré d'une variable est ensuite calculé en fonction des poids des contraintes auxquelles elle appartient. Cette stratégie est analogue aux heuristiques génériques utilisées pour SAT et particulièrement VSIDS abordée section 3.4.2.4 à la différence que ce sont les *nogoods* et non les contraintes qui sont utilisés pour caractériser les variables les plus critiques (toutes les variables de la contrainte ne sont en effet pas forcément impliquées dans l'échec et le *nogood* semble plus précis). La variante utilisée par BerkMin [GN02] pousse le raisonnement plus loin et s'intéresse aux clauses et implicitement aux variables qui ont permis d'atteindre l'échec.

Refalo [Ref04] ne se focalise pas sur les échecs et introduit une mesure d'impact dans l'objectif de détecter les choix qui engendrent le plus de propagation. Il propose de caractériser l'impact d'une décision $x_i = a$, en accord avec le principe du *first fail*, à travers la réduction moyenne de l'espace de recherche engendrée par cette décision. Sa démarche s'inspire des stratégies à base de *pseudocosts* des solveurs linéaires (voir section 2.1.1.3).

Néanmoins, cette réduction n'intervient pas uniquement au moment où la décision est imposée au problème, mais aussi quand d'autres déductions (futures) qui sont en partie basées sur l'hypothèse $x_i = a$ sont faites. L'utilisation d'explications devrait donc fournir une information supplémentaire sur l'implication réelle d'une déduction dans un retrait. Nous pensons ainsi pouvoir pousser plus loin les idées de [BHLS04] et [Ref04] en prenant aussi en compte la chaîne logique de déductions qui permet d'aboutir à l'échec ou à la réduction de l'espace de recherche.

6.1.2 Buts : recherche et compréhension du problème

Notre objectif est d'identifier des variables qui contraignent au maximum le problème, des sous-ensembles de variables particuliers qui entretiennent de fortes relations ou qui ont un impact important sur l'ensemble du problème (analogue à un *backdoor*). Nous avons axé notre étude sur les points suivants :

- l'impact ou l'influence d'une variable sur la réduction directe de l'espace de recherche ;
- l'impact d'une variable au sein d'une chaîne de déductions même longtemps après que cette variable ait été instanciée ;
- la région de l'espace de recherche sur laquelle s'exerce l'influence d'une variable ainsi que les relations précises entre variables.

Les explications pour la programmation par contraintes semblent pertinentes pour fournir de telles informations. Par la suite, on notera :

- E l'ensemble des explications calculées depuis le début de la recherche ;
- E_i^{val} l'ensemble des explications calculées pour tous les retraits de la valeur val du domaine de la variable i ;
- $|e|$ la taille du *nogood* de l'explication e soit le nombre de contraintes de décision contenues dans e. On rappelle qu'une explication e_1 est dite plus précise que e_2 si $|e_1| < |e_2|$. En effet, on peut considérer, que plus une explication est petite, plus elle est précise, car plus le nombre d'hypothèses nécessaires à la déduction que e représente est réduit ;
- a_e^d l'âge d'une décision d au moment du calcul de l'explication e ou le nombre de décisions prises depuis d au moment où e est produite.

6.2 Des explications pour l'identification de structures

Nous abordons dans un premier temps la notion d'impact expliqué pour en faire un outil d'analyse des relations entretenues entre les variables pendant la résolution. Nous présentons ainsi un exemple concret qui explicite l'intérêt des différentes mesures.

6.2.1 Caractériser l'impact

Une décision passée $x_i = a$ a un impact effectif (du point de vue du solveur) sur une valeur val d'une variable x_j si elle apparaît dans l'explication justifiant ce retrait. On note $I_\alpha(x_i = a, x_j, val)$, l'impact de la décision $x_i = a$ sur la valeur val du domaine de x_j. α est l'index utilisé pour distinguer les différentes mesures.

Notre première approche pour mesurer cet impact consiste à considérer le nombre de fois qu'une décision apparaît dans les explications calculées pour la valeur val de x_j en prenant également en compte la taille de l'explication. Les retraits de valeurs s'établissent plus fréquemment pour des explications précises car leurs *pré-conditions* sont réduites. Des explications de petites tailles entraînent donc des relations plus fortes. I_0 propose de quantifier cette relation d'influence :

$$I_0(x_i = a, x_j, val) = \sum_{\{e \in E_j^{val}, x_i = a \in e\}} 1/|e| \tag{6.1}$$

Différentes mesures basées tout d'abord sur l'activité du solveur et le calcul des explications (mesures I_1 et I_2) sont introduites à partir de cette mesure initiale, avec l'objectif de rendre compte des structures dynamiques. Dans un second temps, une mesure s'appuyant plus sur la réduction de l'espace de recherche (I_3) est proposée pour capturer les structures statiques et aider à guider la recherche.

Comme la recherche oriente à l'évidence la propagation (et vice-versa), il semble assez naturel de normaliser cette mesure par rapport à elle.

- L'impact est ici normalisé par rapport au nombre de fois $|x_i = a|$ où une décision $x_i = a$ est prise. Le but étant simplement de ne pas surestimer l'impact des décisions prises fréquemment :

$$I_1(x_i = a, x_j, val) = \frac{I_0(x_i = a, x_j, val)}{|x_i = a|}$$

- Une autre manière de normaliser est de considérer l'âge a_e^d d'une décision d au moment du calcul de l'explication e avec l'objectif de faire décroître l'impact des vieilles décisions. On obtient :

$$I_2(x_i = a, x_j, val) = \sum_{\{e \in E_j^{val}, x_i = a \in e\}} \frac{1}{|e| \times a_e^{x_i = a}}$$

- Le calcul des impacts est dispersé au sein du processus de résolution à chaque calcul d'une explication. C'est une approche sensiblement différente de [Ref04] qui analyse chaque décision séparément pour évaluer son impact instantané. I_3 essaie d'identifier les réductions récurrentes de l'espace de recherche liées à une décision :

$$I_3(x_i = a, x_j, val) = \frac{I_0(x_i = a, x_j, val)}{|\{x_i = a \text{ active } \wedge val \in Dom(x_j)\}|}$$

$I_3(x_i = a, x_j, val)$ peut être considérée comme la probabilité que la valeur val de x_j soit éliminée si la décision $x_i = a$ est prise. Une telle mesure est mise à jour chaque fois qu'un nouveau retrait intervient dès lors que $x_i = a$ est active. Elle prend en compte la fréquence comme la proportion avec lesquelles une décision est impliquée dans une explication de retrait. Enfin, I_1 met en valeur les vieilles décisions et I_2 les plus récentes. La section 6.2.3 montre l'intérêt de ces différents paramètres pour un utilisateur et sa compréhension du comportement du solveur.

Ces mesures d'impacts restent fortement dépendantes de l'exploration effectuée et des techniques semblables à celles présentées dans [Ref04] seront employées afin de les initialiser (par la propagation de chaque valeur de chaque variable) et de les affiner (utilisation d'une procédure de *restart* qui consiste à recommencer la recherche pour profiter dès le noeud racine des impacts obtenus par la résolution précédente).

6.2.2 Relations entretenues par les variables

L'objectif est d'obtenir une représentation synthétique de la structure du problème qui permette à un utilisateur d'analyser et comprendre le problème et le comportement du solveur pendant la résolution. Nous cherchons à exhiber les relations entretenues entre les variables. À cette fin, les mesures d'impacts introduites précédemment sont agrégées sur l'ensemble des valeurs du domaine :

$$\forall \alpha \in [0, 1, 2], \qquad I_\alpha(x_i = a, x_j) = \sum_{val \in D(x_j)} I_\alpha(x_i = a, x_j, val)$$

Un cas particulier se présente pour I_3 où il s'agit de relier l'impact à la réduction de l'espace de recherche engendrée par une variable sur une autre. On considère dès lors la taille du domaine initial :

$$I_3(x_i = a, x_j) = \frac{|D(x_j)| - \sum_{val \in D(x_j)} (1 - I_3(x_i = a, x_j, val))}{|D(x_j)|}$$

Dans ce contexte, $1 - I_3(x_i = a, x_j, val)$ correspond en quelque sorte à la probabilité de présence de la valeur val de la variable x_j après avoir pris la décision $x_i = a$. Cette agrégation permet de quantifier l'influence d'une variable sur une autre de la manière suivante :

$$I_\alpha(x_i, x_j) = \sum_{v \in D(x_i)} I_\alpha(x_i = v, x_j)$$

La structure des relations entre variables est ainsi représentée par un graphe d'impact associé à chaque mesure α. Ce graphe est défini comme un graphe pondéré orienté $GI_\alpha(X, E, W)$ où X dénote l'ensemble des variables du problème et le poids d'un arc $(x_i, x_j) \in E$ $(E = X \times X)$ est donné par $I_\alpha(x_i, x_j)$. Un sommet est donc associé à chaque variable et le poids de chaque arc (x_i, x_j) représente l'influence de x_i sur x_j. Plus le poids est élevé, plus l'influence est grande.

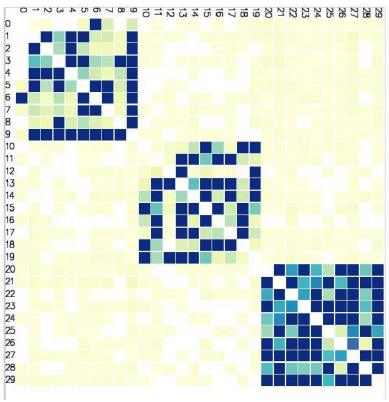


FIG. 6.1: Une représentation du graphe d'impact à l'issue de la phase d'initialisation en utilisant la mesure I_0 comme mesure d'impacts.

6.2.3 Illustration des impacts pour l'analyse de structures

Nous cherchons à illustrer à présent, l'intérêt du graphe d'impact en analysant des structures sur un cas concret. Nous examinerons notamment comment l'information extraite du graphe d'impact permet à l'utilisateur d'effectuer une analyse du problème ainsi que sa résolution.

Nous considérons un problème binaire aléatoire dans lequel une structure est insérée en augmentant la dureté de certaines contraintes de manière à faire apparaître plusieurs sous-ensembles de variables entretenant des relations fortes. Les instances aléatoires sont caractérisées par un tuple $< N, D, p1, p2 >$ (on utilise le modèle B [AKK+97]) où N est le nombre de variables, D la taille des domaines, p_1 la densité du réseau (p_1 est la proportion de contraintes binaires dans le réseau ce qui fixe le nombre de contraintes à $p_1 \times N(N-1)/2$ contraintes) et p_2 la dureté des contraintes (la proportion de couples interdits). Les paramètres de l'instance considérée sont N = 30, D = 10 et $p_1 = 35\%$. Par ailleurs les 30 variables sont divisées en trois sous-ensembles de 10 variables entre lesquelles la dureté est fixée à $p_2 = 53\%$ tandis que le reste du réseau est fixé à 3%.

L'instance spécifique isolée ici à titre d'illustration nous a semblé intéressante à cause de sa difficulté inattendue pour `mindom` [HE80] (la variable de plus petit domaine courant est choisie en priorité). En utilisant les différents graphes d'impact introduits précédemment, nous souhaitons répondre à différentes questions soulevées face à ce problème :

- est-il possible de reconnaître la structure intégrée au problème ?
- pourquoi `mindom` échoue sur cette instance ? Cette difficulté est-elle intrinsèque au problème ou à l'heuristique elle-même ?

Visualiser le graphe d'impact

Les figures 6.1 à 6.4 donnent une représentation du graphe d'impact GI des 30 variables du problème. Nous utilisons une représentation sous forme de matrice d'adjacence [GJF04]. Les cellules à l'intersection de chaque ligne i et colonne j indiquent l'impact de v_j sur v_i. Plus l'impact est fort, plus l'arc possède un poids élevé et plus la cellule est foncée. La matrice est ordonnée dans l'ordre où sont créés les noyaux cachés

de variables[1].

Avant de commencer la recherche, une phase de singleton consistance [PSW00] est appliquée (chaque valeur de chaque variable est propagée) afin d'initialiser l'impact des variables de manière homogène. Bien que le graphe soit entièrement connecté, la visualisation sous forme de matrice donnée en figure 6.1 permet de distinguer clairement les trois noyaux de variables qui entretiennent des relations fortes, juste après cette première étape de propagation (la mesure I_0 est utilisée ici).

La figure 6.2 donne une image du graphe d'impact après 2 minutes de recherche en utilisant `mindom` comme choix de variable.

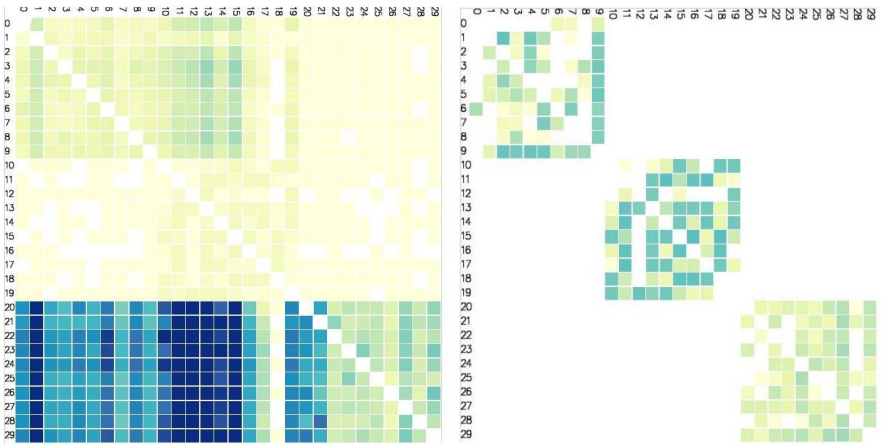


FIG. 6.2: Le graphe d'impact basé sur I_0 (graphe de gauche) et I_3 (droite) après deux minutes de calcul en utilisant `mindom`.

Les impacts ne sont donc pas employés ici pour la recherche mais sont maintenus pendant la recherche et permettent d'analyser la résolution effectuée par `mindom`. On peut noter que I_0 est centrée sur les aspects dynamiques de la recherche (les clusters initiaux ne sont plus du tout visibles comparés à la figure 6.1) alors que I_3 est focalisée sur les structures statiques et relègue dans l'ombre les liens faibles même après 2 minutes de calculs (voir figure 6.2). La zone la plus sombre dans le coin en bas à gauche de la figure 6.2, montre que, selon I_0, les variables des deux premiers noyaux ont apparemment une forte influence sur le troisième noyau. La zone deviendrait de plus en plus sombre si la recherche n'avait pas été interrompue à ce stade à des fins d'analyse. On peut penser que de mauvais choix initiaux sur les deux premiers ensembles ont mené le solveur dans une succession d'échecs sur le troisième ensemble masquant ainsi les structures inhérentes au problème.

La figure 6.3 représente le graphe GI_1 obtenu à partir de la mesure I_1. Il s'agit donc d'un graphe normalisé où l'influence d'une décision prise par le solveur est divisée par le nombre de fois où cette décision est prise pendant la résolution (mesure I_1).

L'objectif est de pousser plus loin l'analyse précédente en distinguant deux types de décisions parmi celles qui possèdent une forte influence : certaines sont répétées fréquemment alors que d'autres ont guidé le solveur dans une branche stérile de l'es-

[1]Nous nous penchons actuellement sur l'utilisation d'algorithmes de *clustering* de manière à identifier cet ordre particulier à partir du graphe d'impact seul.

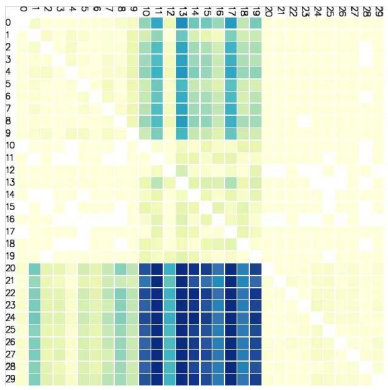


FIG. 6.3: Une représentation du graphe d'impact normalisé par rapport au nombre de fois où une décision est prise (I_1).

pace de recherche et apparaissent dès lors dans de nombreuses explications d'échecs. On cherche à isoler de cette manière les mauvais choix qui semblent se situer sur le deuxième noyau. Enfin la figure 6.4 rapporte l'activité de la résolution en s'appuyant sur un graphe d'impact dans lequel l'effet des anciennes décisions est progressivement éliminé (mesure I_2). Il apparaît alors clairement que le solveur ne cesse d'échouer sur le premier puis plus fréquemment sur le troisième noyau avec très peu d'interaction sur le deuxième où se situent les mauvais choix initiaux. I_2 met en effet l'accent sur les vieilles décisions.

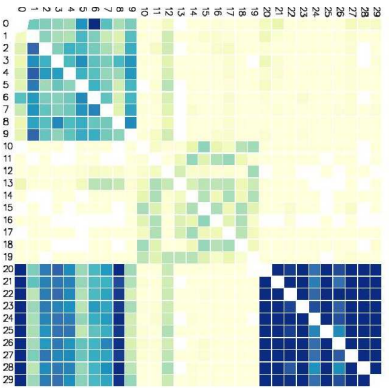


FIG. 6.4: Une représentation du graphe d'impact normalisé sur l'âge des décisions (I_2).

De manière à confirmer cette interprétation, nous avons changé l'heuristique de recherche pour prendre en compte l'impact des variables au cours de la résolution (en s'appuyant sur I_0) et revenir ainsi immédiatement sur les variables dont l'influence s'accroît de manière indue (parce qu'elles interviennent dans toutes les explications de retrait sans fournir une réduction significative de l'espace de recherche). Le problème est alors résolu instantanément.

6.3 Les impacts comme heuristiques de recherche

Nous cherchons ici à évaluer si l'impact introduit précédemment constitue une piste sérieuse pour améliorer les heuristiques de recherche génériques. Les heuristiques de recherche prennent en compte les domaines (`mindom`), comme le degré des variables dans le réseau de contraintes du problème (`dom + deg` ou `dom/deg`) avec pour objectif d'identifier les variables les plus contraintes, celles qui simplifient au maximum le problème. Les impacts présentés jusqu'à présent dans le cadre de l'identification de structures généralisent de façon naturelle cette idée.

6.3.1 Impact orienté variables

Les impacts ont été agrégés jusqu'à présent pour exprimer les relations d'une variable avec une autre. Leur utilisation dans le cadre de la recherche demande une agrégation globale vis à vis de tout le problème. Pour les mesures I_0, I_1 et I_2, l'impact global d'une décision est calculé en accumulant l'impact sur les variables de l'ensemble du problème[2] :

$$\forall \alpha \in [0,1,2], \qquad I_\alpha(x_i = a) = \sum_{x_j \in X} I_\alpha(x_i = a, x_j)$$

La variable x choisie pour le branchement est celle qui maximise $\sum_{a \in D(x)} I(x = a)$ de manière à favoriser la variable la plus influente ($D(x)$ correspond au domaine courant de x). La valeur choisie est celle qui minimise $I(x = a)$ de manière à favoriser les combinaisons futures consistantes.

6.3.2 Impact orienté contraintes

Les impacts ont été définis dans le cadre des contraintes de décision de manière à porter sur les variables et révéler leurs relations. Ils s'appliquent naturellement sur n'importe quelle contrainte ct de la même manière que I_0 :

$$I(ct) = \sum_{\{e \in E, ct \in e\}} 1/\left|e\right| \tag{6.2}$$

L'heuristique reconnue à l'heure actuelle comme la meilleure heuristique standard est sans doute `dom/deg` introduite par [BR96]. `dom/deg` propose de sélectionner la variable qui minimise le rapport entre la taille de son domaine et son degré dans le réseau de contraintes. Il s'agit de favoriser les petits domaines et les variables les plus contraintes. Elle a été améliorée par [BHLS04] `dom/Wdeg` en incrémentant le degré d'une contrainte chaque fois que celle-ci lève une contradiction. Nous proposons de raffiner encore davantage cette heuristique en remplaçant le degré par la mesure de l'impact des contraintes portant sur la variable (aboutissant ainsi à l'heuristique nommée par la suite `dom/Ict`).

[2] Le cas d'I_3 est légèrement différent puisqu'il s'agit de se focaliser sur la réduction de l'espace à l'instar de la formule en section 6.2.2 pour $I_3(x_i = a, x_j)$.

6.4 Expérimentations

Pour des raisons de simplicité et de clarté dans les résultats, deux variantes essentielles sont retenues pour tester l'intérêt des impacts pour la recherche : I_2 et `dom/Ict`. Le cadre des expérimentations est le suivant :

- les explications étant maintenues pour le calcul des impacts, elles peuvent être utilisées sans surcoût pour effectuer la recherche en *backjumping* (algorithme MAC-CBJ présenté section 3.2.1).
- les heuristiques de référence données à titre de comparaison sont `dom/deg`[3], `dom/Wdeg` ainsi que I_R correspondant à notre implémentation de la mesure introduite dans [Ref04]. I_R consiste à mesurer l'espace de recherche (par le produit cartésien des domaines) avant et après chaque choix pour quantifier précisément la réduction. On distingue deux variantes de I_R, $I_R(max)$ consiste à retenir la variable qui maximise l'impact soit la réduction de l'espace de recherche [4]. $I_R(min)$ choisit la variable qui minimise l'espace restant. De manière à séparer l'influence du backjumping, I_R et `dom/deg` sont testés à la fois en mode MAC et MAC-CBJ. La meilleure combinaison est reportée.
- les égalités intervenant dans les heuristiques à base d'impact sont arbitrées au hasard ;
- enfin, nos expérimentations sont conduites sur un Pentium 4, 3 GigaHz, sous Windows XP. Notre solveur de contraintes est la version java la plus récente du solveur libre `choco` (`choco-solver.net`).

[Ref04] mentionne que l'initialisation des impacts est critique et que l'utilisation de *restart* peut s'avérer parfois payante quand l'initialisation échoue à approximer correctement les impacts. Les impacts deviennent en effet de plus en plus pertinents avec le temps. Les deux techniques sont donc également mises en œuvre ici :

- *restart* : les procédures de *restart* imposent une limite croissante sur le nombre de nœuds qui double d'une itération à la suivante (garantissant ainsi à terme la complétude). Les résultats avec *restart* sont indiqués pour une heuristique h (sous la dénomination $h + rest$) dès qu'ils sont meilleurs que h seule ;
- *initialisation* : les impacts sont initialisés à l'aide d'une phase de propagation analogue à de la singleton-consistance [PSW00](chaque valeur de chaque variable est propagée) dont le coût est inclus dans les temps indiqués.

Nous avons considéré trois ensembles de jeux de tests :

1. Le premier jeu de tests provient des expérimentations de Refalo [Ref04] : un ensemble de problèmes de multiknapsack modélisés par des variables binaires, et résolus comme un problème de satisfaction (en fixant la valeur optimale comme une contrainte dure). Une limite de temps est fixée à 1500s. De plus, comme les égalités sont arbitrées aléatoirement et que la variance entre deux exécutions est considérable, une moyenne sur une trentaine d'exécutions est reportée.

[3] `dom/deg` se comporte beaucoup mieux que `mindom` sur nos jeux de tests.

[4] Tout notre *framework* est basé sur la maximisation de l'impact. $I_R(max)$ propose un impact basé sur la réduction de l'espace. Il est intéressant de la comparer à I_2 dont l'impact est basé sur le degré d'implication des variables dans la propagation

2. Le deuxième ensemble de tests est fait de problèmes binaires aléatoires structurés tels que celui décrit à la section 6.2.3. Un problème $< 45, 10, 35, p_2 >$ est structuré avec trois noyaux de 15 variables reliés par une dureté intra-noyaux p_2 et une dureté inter-noyaux de 3%. De nouveau, 100 instances sont considérées pour chaque valeur de p_2. On présente sur ces tests les temps de résolution.

3. Enfin le dernier jeu de tests est issu de problèmes industriels d'allocation de fréquences [CdGL+99] provenant de l'archive FullRLFAP. Le problème consiste à choisir les fréquences f_i de canaux de communication en minimisant les interférences. Ces contraintes d'interférence s'expriment à travers une distance minimale entre les fréquences de certains canaux ($|f_i - f_j| > E_{ij}$ ou $|f_i - f_j| = E_{ij}$). Nous avons suivi l'approche de [BCS01, BHLS04] pour générer des instances difficiles en satisfaction (il s'agit à l'origine d'un problème d'optimisation). Ainsi, scenXX-wY-fZ correspond à l'instance scenXX dans laquelle les contraintes de poids supérieur à Y ainsi que les Z fréquences les plus hautes ont été éliminées. Les résultats sont donnés sur une sélection de 15 instances difficiles identifiées par [BCS01, BHLS04].

6.4.1 Premier benchmark : problèmes de multiknapsack

Sur ce premier benchmark (dont les résultats sont indiqués table 6.1), I_R apparaît comme la meilleure stratégie ($I_R(min)$ et $I_R(max)$ ont des comportements équivalents. On retrouve les résultats de [Ref04] concernant I_R et l'utilisation de *restart* ne profite à aucune des heuristiques testées. L'arbitrage au hasard des égalités prend une trop grande importance pour I_2 et `dom/Ict`[5]. Les deux heuristiques ne sont pas pertinentes et demande un *apprentissage* trop long avant que les impacts ne se stabilisent et permettent de discriminer les variables. Le phénomène s'observe bien pour I_2 en examinant le nombre de nœuds de la dernière itération en cas de *restart* (*cf.* table 6.1). Ainsi, l'utilisation de *restart* rend I_2 de plus en plus pertinente mais trop lentement et le temps global nécessaire à la résolution augmente.

[5]Les résultats sont nettement meilleurs sur ce benchmark (pour l'ensemble des heuristiques) avec un arbitrage fixe des égalités.

	MAC `dom/deg`		MAC I_R		MAC `dom/Wdeg`	
	Tps (s)	Nœuds	Tps (s)	Nœuds	Tps (s)	Nœuds
mknap1-2	0	11.2	0	24.3	0	11.9
mknap1-3	0	85.9	0	165.7	0	89.8
mknap1-4	0.3	2236.7	0.2	1149.5	0.4	2506.1
mknap1-5	3.6	27749.1	3.5	23158.1	4.7	32437.6
mknap1-6	316.8	2031108.5	201.1	1066116.4	452.9	2636561.5

	`dom/Ict`		I_2		I_2 + rest	
	Tps (s)	Nœuds	Tps (s)	Nodes	Tps (s)	Nœuds
mknap1-2	0	32.8	0	26.0	0	26.0
mknap1-3	0.1	334.5	0.1	594.3	0.1	200.8
mknap1-4	4.3	15063.8	2	7141.5	6	6770.8
mknap1-5	723	2881651.4	234	861328.5	317	446652.6
mknap1-6	> 1500		> 1500		> 1500	

TAB. 6.1: Impacts sur les problèmes de *multiknapsack*.

6.4.2 Deuxième benchmark : problèmes binaires aléatoires structurés

Sur ce jeu de test, MAC-CBJ semble critique (ainsi, MAC `dom/deg`, MAC $I_R(max)$ ou MAC $I_R(max)+rest$ ne sont pas du tout compétitives). Les figures 6.5 et 6.6 rapportent le temps de résolution pour les quatre heuristiques. I_2, `dom/Ict` et `dom/Wdeg` sont incomparablement plus performantes que $I_R(max)$, elle-même meilleure que `dom/deg`. On peut noter qu'il est inutile de procéder à un *restart* pour $I_R(min)$, I_2 et `dom/Ict` pour qui la phase d'initialisation est suffisante. Ce n'est pas le cas de $I_R(max)$ qui demande plus de temps pour estimer correctement les impacts et que le *restart* améliore considérablement.

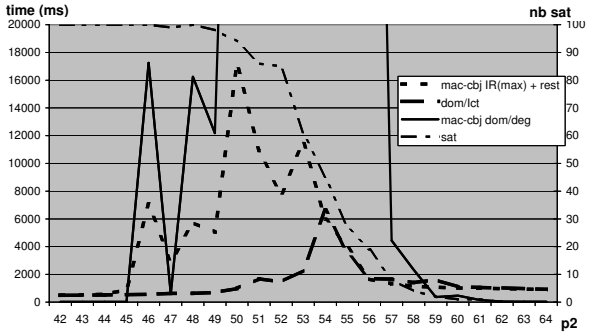


FIG. 6.5: Temps moyen de résolution (axe de gauche) et nombre d'instances faisables (sat) (axe de droite) pour `dom/deg`, $I_R(max)$ et `dom/Ict`.

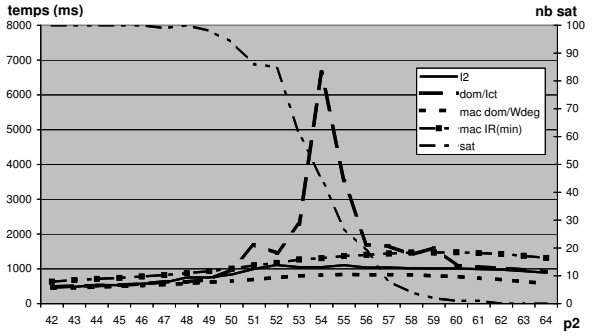


FIG. 6.6: Temps moyen de résolution (axe de gauche) et nombre d'instances faisables (sat) (axe de droite) pour `dom/Ict`, $I_R(min)$, I_2 et `dom/Wdeg`.

La complexité de ce benchmark tient certainement au degré d'interaction avec l'algorithme de recherche et les instances sont résolues instantanément par les heuristiques à base d'impact contrairement à `dom/deg` ou `mindom` qui laisse croire que ces problèmes sont difficiles. La présence de telles structures artificielles favorise de notre point de vue un certain type de comportement *heavy-tailed* [WGS03a] qui se caractérise par une grande variation des temps de résolution d'une recherche purement aléatoire indiquant la présence de mauvais points de choix initiaux très difficile à détecter par la suite.

On peut ainsi remarquer que le mauvais comportement de $I_R(max)$ se manifeste ainsi juste avant la transition de phase.

6.4.3 Dernier benchmark : Allocation de fréquences

Sur ce benchmark, MAC-CBJ semble décisive comme le souligne les deux premières colonnes du tableau 6.2 (sept instances résolues pour MAC-CBJ contre trois pour MAC). On peut essayer d'analyser ce premier constat à l'éclairage du graphe d'impact[6]. En l'affichant après la phase d'initialisation (première image de la figure 6.7), un utilisateur peut immédiatement constater que le réseau de contraintes est particulièrement peu dense. Par ailleurs, même après une recherche aléatoire (de taille fixe) répétée afin d'exhiber les relations indirectes (seconde image de la figure 6.7), on constate que les impacts ne s'étendent pas à l'ensemble du problème mais restent au contraire assez localisés. On se souvient que des phénomènes de *thrashing* peuvent facilement apparaître dans ce genre de situation (voir section 2.2). En effet, comme la propagation reste très locale, une contradiction peut être découverte très tardivement sur une partie du graphe après une longue recherche sur une autre partie (puisque les deux parties entretiennent des relations faibles).

Les performances des stratégies sont donc présentées dans le tableau 6.2 dans leur combinaison avec MAC-CBJ.

		MAC `dom/deg`	MAC-CBJ `dom/deg`	I_2 $+rest$	$I_R(max)$ + rest	$I_R(min)$	`dom/Ict`	`dom/Wdeg`
`scen11`	Tps (s)	47	7.9	38	53	183.4	40	11.5
(sat)	Nœuds	5863	1207	1432	3986	10258	524	907
`scen02-f24`	Tps (s)	0.8	0.1	3	3	3.3	3	0.1
(sat)	Nœuds	620	104	88	90	104	104	95
`scen02-f25`	Tps (s)	$> 1h$	3.6	4.6	3.7	4.1	8.5	1.4
(unsat)	Nœuds	-	610	270	77	91	1252	83
`scen03-f10`	Tps (s)	$> 1h$	1766	11.5	9.7	9.5	10.5	0.5
(sat)	Nœuds	-	572507	1128	415	197	186	188
`scen03-f11`	Tps (s)	$> 1h$	$> 1h$	$> 1h$	$> 1h$	15.6	17.5	19.6
(unsat)	Nœuds	-	-	-	-	1193	788	1369
`scen06-w2`	Tps (s)	$> 1h$	75	14.6	13.5	15.5	15.8	1.1
(unsat)	Nœuds	-	68669	0	0	0	0	78
`scen07-w1-f4`	Tps (s)	0.2	0.2	6	5.9	6.6	6.9	0.3
(sat)	Nœuds	271	202	194	191	197	185	207
`scen07-w1-f5`	Tps (s)	$> 1h$	0	4.4	4.3	4.8	5	0.1
(unsat)	Nœuds	-	26	0	0	0	0	29
`graph08-f10`	Tps (s)	$> 1h$	$> 1h$	$> 1h$	679	41.2	19	14
(sat)	Nœuds	-	-	-	200898	2658	757	1392
`graph08-f11`	Tps (s)	$> 1h$	$> 1h$	$> 1h$	174	14.2	14	3.3
(unsat)	Nœuds	-	-	-	32653	126	25	254
`graph14-f27`	Tps (s)	$> 1h$	$> 1h$	14.9	26.2	10.9	32.9	3.7
(sat)	Nœuds	-	-	4886	9845	723	7080	1817
`graph14-f28`	Tps (s)	$> 1h$	$> 1h$	$> 1h$	$> 1h$	165.3	14.3	4
(unsat)	Nœuds	-	-	-	-	53008	1377	901
nb solved		3/15	7/15	8/15	10/15	12/15	12/15	12/15

TAB. 6.2: Impacts sur les problèmes d'allocation de fréquences.

$I_R(max)$ obtient ici de meilleurs résultats que I_2 qui tous les deux surpassent `dom/deg`. On peut également noter que l'initialisation peut devenir assez coûteuse (autour de 40 sec dans le pire des cas) et qu'elle est parfois suffisante à prouver l'inconsistance (instances avec 0 nœud exploré). `dom/Wdeg` obtient les meilleurs résultats

[6]S'agissant par ailleurs d'un problème industriel réel et structuré, il nous a paru intéressant de donner une image du graphe d'impact.

et permet de résoudre 12 instances sur les 15 faisant partie du jeu de tests originel surpassant ainsi l'ensemble des autres techniques. `dom/Ict` et $I_R(min)$ obtiennent des résultats semblables mais demandent une étape d'initialisation qui les rend moins performantes que `dom/Wdeg` en terme de temps. Nous retrouvons ainsi les résultats de [BHLS04] (les instances `scen11-f1`, `scen11-f2`, `scen11-f3` n'ont pas pu être résolues dans la limite de temps d'une heure par aucun des algorithmes).

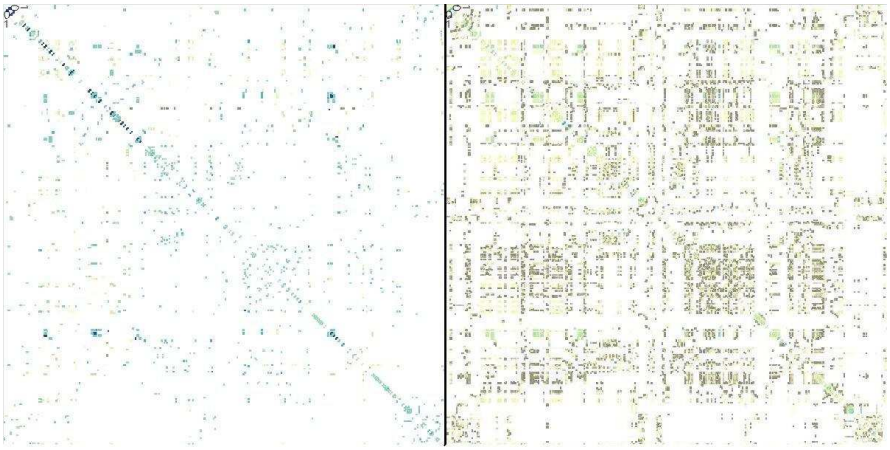


FIG. 6.7: Le graphe d'impact des variables après l'initialisation et après une recherche aléatoire répétée pour essayer de construire une vue homogène des relations.

6.4.4 Premières conclusions sur les stratégies à base d'impacts

1. Les stratégies I_α avec $\alpha \in \{0, 1, 2\}$ sont fortement basées sur l'activité du solveur pendant la recherche (en se focalisant donc sur la partie dynamique des structures). Leur utilisation peut s'avérer extrêmement rentable dans la mesure où elles révèlent les mauvais choix initiaux (dont l'influence grandit démesurément au cours de la recherche sans apporter un élagage utile puisque le solveur ne parvient pas à revenir dessus). Par ailleurs, quand l'impact est appliqué aux contraintes, il révèle celles qui sont responsables à court ou long terme de la propagation. Ainsi `dom/Wdeg` est efficace quand les contraintes dans le réseau ont de faibles arités. Elle perd son efficacité en présence de contraintes globales portant sur de larges portions du problème ou le problème tout entier puisqu'il n'y a aucune discrimination entre les variables en incrémentant simplement le degré de la contrainte. `dom/Ict` corrige en partie ce phénomène en s'appuyant sur les explications pour différencier les variables.
2. En revanche, nous pensons que l'impact basé sur les explications varie trop d'un nœud à l'autre de l'arbre (précisément parce qu'il se focalise sur les aspects dynamiques) pour faire des I_α à elles seules des heuristiques génériques robustes. Or la stabilité des impacts observés dans différentes parties de l'arbre est l'hypothèse sous-jacente à I_R ou aux *pseudocosts* linéaires (voir section 2.1.1.3).

I_3 est trop coûteuse (en terme de temps) dans l'état actuel pour être utilisée comme heuristique par défaut mais d'autres compromis intéressants entre I_R et I_α avec $\alpha \in$

$\{0, 1, 2\}$ peuvent être imaginés de manière à profiter de la robustesse générale de I_R tout en essayant de se prémunir contre des phénomènes de *thrashing* liés au choix initiaux. Ces expérimentations cherchent à affiner notre compréhension des phénomènes mis en jeu pendant la recherche. $I_R(min)$ reste la meilleure heuristique en terme de robustesse sur tous nos tests.

6.5 Conclusion

Nous avons montré ici comment les explications, à travers le graphe d'impact pouvaient permettre de comprendre certaines structures du problème comme le processus de résolution. Les phénomènes de *thrashing* sont ainsi parfaitement visibles et peuvent être compensés par l'heuristique sans passer par des techniques plus coûteuses comme DBT. Le graphe d'impact est ainsi un outil original et synthétique de visualisation des structures au cours de la résolution.

L'utilisation d'un impact expliqué pour guider la recherche manque pour l'instant de robustesse dans un cadre générique. Nos résultats sont clairement dominés par $I_R(min)$ sur ces trois benchmarks. Nous pensons néanmoins que certains mauvais choix ne se détectent pas au niveau de la réduction de l'espace de recherche mais sont liés à la sémantique des contraintes et apparaissent très bien dans les explications comme le montre l'analyse faite sur l'exemple section 6.2.3. Il s'agit d'un premier pas vers des heuristiques basées sur les explications.

L'utilisation de I_2 sur le problème de placement de tâches temps-réel traité au chapitre 9 s'avère ainsi particulièrement efficace. La normalisation de l'impact à la fois par la fréquence des décisions et par leur âge reste à évaluer et devrait présenter davantage de robustesse que I_2. Pour des raisons de temps et de clarté, nous avons préféré séparer les deux critères dans cette première étude.

Ces résultats ont été publiés dans [CJ06, GCFJ05].

Chapitre 7

Retour sur l'enregistrement de *nogoods*

Nous avons introduit au chapitre 3 dans la section 3.4.2 les principes des solveurs SAT et leur utilisation fructueuse des explications. Ces solveurs sont en effet capables de tirer parti des *nogoods* malgré leurs exigences en temps comme en espace. Inspirées par ces succès, plusieurs tentatives ont été faites pour reconsidérer les *nogoods* en PPC [KB03, KB05].

Il y a deux questions fondamentales qui se posent vis à vis des *nogoods*. Il s'agit d'abord de savoir **comment calculer les nogoods ?** Le *nogood* idéal est celui qui élimine la plus grande classe d'affectation possible de l'espace de recherche futur. Étant donné que les *nogoods* ont été avant tout utilisés en PPC pour supporter le *backtrack* intelligent et les CSPs dynamiques [SV94], ils font traditionnellement référence aux contraintes de décision uniquement et ne sont pas particulièrement utiles pour le filtrage de l'espace de recherche. Différents travaux en SAT comme en PPC fournissent des réponses à cette question que nous aborderons en section 7.1.

Il s'agit ensuite de savoir **comment gérer et propager l'ensemble des nogoods appris ?** Leur nombre potentiellement exponentiel rend critique cette deuxième question. En contraintes, les *nogoods* ont été souvent utilisés pour vérifier simplement si le nœud courant peut être ou non étendu à une solution. Les solveurs SAT vont plus loin et réalisent la BCP sur les *nogoods* appris. Nous présenterons section 7.1.2 la méthode pour implémenter en PPC le schéma de propagation reconnu comme le plus efficace dans la communauté SAT et intitulé *two watched literals* [MMZ+01]. Néanmoins, la conception de structures de données efficaces pour l'enregistrement de *nogoods* est toujours une question ouverte en PPC. C'est cette question que nous abordons ici section 7.2 en proposant la compilation des *nogoods* dans un automate et le maintien de l'arc-consistance avec le calcul des explications sur cet automate.

7.1 Calcul et propagation des nogoods

7.1.1 *Nogoods* généralisés

Définition 7.1.1 *Une* ***déduction*** $(x \neq a)$ *est le retrait de la valeur a du domaine de la variable x.*

Définition 7.1.2 *Une* ***explication généralisée****, $g\_expl(x_i \neq a)$ pour la déduction $(x_i \neq a)$ est définie par deux ensembles de contraintes :*
- *$C' \subseteq C$;*
- *Δ, un ensemble de déductions.*

telle que $C' \wedge \Delta \models x_i = a$

$g\_expl_\Delta(x_i \neq a)$ fait référence à l'ensemble Δ associé à une explication généralisée $g\_expl(x_i \neq a)$. Une déduction est due à d'autres déductions sauf si elle est faite au nœud racine. Les inférences faites en cours de propagation peuvent être ainsi remontées jusqu'aux contraintes de décision. On peut ainsi calculer des explications à partir d'explications généralisées. Un ensemble vide Δ pour une déduction $x_i \neq a$ représente la déduction qui est, soit due directement à une contrainte de décision du type $x_i = b$, soit faite au nœud racine.

Exemple 11 *Explications généralisées*

Soit x_1 et x_2, deux variables telles que $D_x = \{0, \ldots, 6\}$ et $D_y = \{0, 2, 3, 4, 6\}$. Les valeurs 1 et 5 ont été retirées du domaine de y et une explication est déjà disponible pour ces déductions. Imaginons que $expl(y \neq 1) = \{x_4 = 2\}$ et $expl(y \neq 5) = \{x_0 = 3, x_8 = 1\}$ et considérons la contrainte $|x - y| = 2$. La valeur 3 de x est retirée en appliquant l'algorithme de filtrage de $|x-y| = 2$. Une explication généralisée s'écrit simplement $g\_expl_\Delta(x \neq 3) = \{y \neq 1, y \neq 5\}$. L'explication est alors $expl(x \neq 3) = expl(y \neq 1) \cup expl(y \neq 5) = \{x_4 = 2, x_0 = 3, x_8 = 1\}$.

Les explications ont été conçues pour le *backtrack* intelligent et font donc systématiquement référence au chemin de décision pour pouvoir le réparer au moment de la contradiction sans avoir à recalculer les contraintes de décision en cause dans l'échec à partir de l'explication de contradiction généralisée. En conservant des raisons intermédiaires au retrait d'une valeur et en repoussant le calcul du *nogood* au moment de la contradiction, on peut obtenir des *nogoods* généralisés [KB05, JB97] qui possèdent un plus grand pouvoir de filtrage et correspondent à un nombre exponentiel de nogoods simples.

Définition 7.1.3 *Un nogood généralisé est un ensemble de contraintes C', un ensemble de déductions Δ et un ensemble de contraintes de décision DC tels que $C' \wedge \Delta \wedge DC$ est inconsistant.*

En conservant les explications généralisées, on dispose en mémoire de l'enchaînement logique des inférences faites durant la recherche. En SAT, cela correspond au graphe d'implication (voir section 3.4.2.2). A partir d'une contradiction sur un domaine vide, on peut calculer plusieurs *nogoods* généralisés (alors qu'un seul nogood est disponible avec la technique classique). Le schéma général pour calculer le nogood[1] est donné par l'algorithme 14. La ligne 1 commence par construire l'explication généralisée de contradiction exprimant le fait que le domaine de x_i a été vidé. Chaque déduction peut être ensuite remplacée par son explication généralisée pour obtenir un nouveau *nogood*.

[1] Ce schéma est équivalent au calcul d'une coupe dans le graphe d'implication introduit en SAT. Le graphe d'implication est connu en CP sous le nom de *proof-tree* [DFJ+03].

Un *nogood* est finalement constitué de déductions comme de décisions. En ajustant le critère d'arrêt, on peut implémenter n'importe quel schéma d'enregistrement SAT comme le point d'implication unique.

procédure : computeGeneralizedNogood(Var x_i)

1: GeneralizedContradictionExplanation $e \leftarrow \bigcup_{j \in D^{orig}} x_i \neq j$;
2: **while** stopping criterion not met **do**
3: $x_k \neq k \leftarrow$ choose a deduction of e ;
4: **if** $g\_expl_\Delta(x_k \neq k)$ is not empty
5: $e \leftarrow e \cup g\_expl(x_k \neq k) - \{x_k \neq k\}$;
6: **else** $e \leftarrow e \cup expl(x_k \neq k) - \{x_k \neq k\}$;
7: **end while**
8: **return** e ;

Algorithme 14: Calcul de nogoods généralisés

Afin que le *nogood* puisse servir à justifier le retrait d'une contrainte de décision (et être enregistré comme explication généralisée), il faut que celui-ci ne fasse néanmoins référence qu'à des déductions antérieures à la dite décision.

7.1.2 Propagation par *watched literals*

La propagation des *nogoods* est généralement limitée à la propagation unitaire utilisée par les solveurs SAT. Nous introduisons ici le mécanisme des *two watched literals* dans le contexte de la PPC qui sera notre référence pour propager des *nogoods*.

Notations : On appelle atome, une paire variable/valeur (x_i, j). Un atome positif fera référence à $x_i = j$ alors qu'un atome négatif, à $x_i \neq j$. Un atome positif (resp. négatif) est dit **satisfait** dès que x_i est instanciée à j (resp. j retirée de x_i), **falsifié** dans le cas contraire où $j \notin x_i$ (resp. x_i instanciée à j) et **libre** autrement. Un *nogood* généralisé (Δ, DC) peut se voir comme une contrainte, c'est à dire une clause sur les atomes correspondants $(\bigvee_{x_k \neq j \in \Delta} x_k = j) \vee (\bigvee_{x_k = j \in DC} x_k \neq j)$, qui doit être vérifiée dans la recherche future. Un *nogood* est libre tant que deux atomes sont libres, satisfait dès qu'un atome est satisfait et falsifié quand tous les atomes sont falsifiés. Par ailleurs, un *nogood* est dit *unitaire* quand il possède un unique atome libre alors que tous les autres sont falsifiés. Dans ce cas, la propagation unitaire force ce dernier atome à être satisfait.

Propagation unitaire : Le statut d'un *nogood* (libre, satisfait, falsifié ou unitaire) peut être déterminé en observant uniquement deux de ses atomes. Chaque *nogood* est **observé** par deux pointeurs sur deux atomes libres. Deux listes de *nogoods* sont maintenues pour chaque atome : la liste positive, $pos\_watch(x_i, j)$ est la liste de *nogoods* où l'atome positif $x_i = j$ est observé, et la liste négative, $neg\_watch(x_i, j)$ dénote la liste des *nogoods* où $x_i \neq j$ est observé. La liste $pos\_watch(x_i, j)$ est parcourue au retrait de la valeur j du domaine de x_i (c'est à dire $x_i \neq j$) et $neg\_watch(x_i, j)$ est considérée en cas d'affectation (c'est à dire $x_i = j$). Pour chaque *nogood* dans la liste,

l'atome observé (qui est maintenant falsifié) doit être mis à jour et plusieurs cas se présentent :

1. l'autre atome observé est déjà satisfait, le pointeur de l'atome falsifié reste inchangé ;
2. un autre atome libre ou satisfait est trouvé et la liste correspondante est mise à jour ;
3. sinon, tous les autres atomes sont falsifiés. Le *nogood* est unitaire et l'autre atome observé est propagé. Le pointeur sur l'atome falsifié reste inchangé.

Il faut remarquer que les pointeurs sont laissés à des positions qui redeviendront valides au *backtrack* de sorte que dès qu'un *nogood* est libre, il est correctement observé par deux atomes libres.

Ajout dynamique : Certaines précautions doivent néanmoins être prises à l'ajout dynamique d'un *nogood* à une feuille de l'arbre :

- les deux pointeurs sont placés sur les deux derniers (chronologiquement) atomes de sorte qu'au *backtrack* après l'ajout du *nogood*, l'un au moins redevient libre (l'autre restant falsifié) ;
- puis, tant que ce deuxième atome reste falsifié, il faut propager le *nogood* à chaque *backtrack*.

Dès que le *nogood* possède deux atomes libres, il ne demande plus aucun traitement particulier et on peut appliquer tel quel le schéma précédent. Ce problème ne se pose pas en SAT dans la mesure où le *backjumping* est effectué différemment (voir section 3.4.2.3).

L'absence de structures de données restaurables au *backtrack* est un gros avantage des *watched literals*, particulièrement utile dans le cas des besoins exponentiels en mémoire des *nogoods*. La propagation efficace comme la gestion optimisée en mémoire des *nogoods* sont toujours les facteurs limitants des approches d'enregistrement de *nogoods* et nous proposons ici une autre gestion, basée sur leur représentation par un automate. L'idée est que les *nogoods* appris dans un même sous-arbre partagent probablement beaucoup de déductions et que beaucoup de données peuvent donc être factorisées. En basant la représentation des *nogoods* sur un automate, nous montrerons que la propagation peut encore être effectuée de manière efficace.

7.2 Vers un encodage des *nogoods* sous forme d'automate

Les *nogoods* généralisés peuvent être vus comme un ensemble de tuples. En particulier, dans le cas des domaines finis, une déduction $(x_i \neq v_j)$ sur le domaine D_i peut être vue comme $\bigvee_{v_k \in D_i \setminus \{v_j\}} (x_i = v_k)$.

7.2.1 Représentation des tuples par un automate

Un ensemble de tuples portant sur n variables peut être encodé par un automate acyclique avec $\ell = (n+1)$ niveaux correspondant à chaque variable plus l'état final F. Un automate fini déterministe est la donnée de $(Q, \Sigma, \delta, q_0, F)$ où Q est un ensemble

fini d'états et $q_0 \in Q$ est l'état initial. Ici, l'alphabet Σ correspond à l'union de tous les domaines des variables et Σ^* à l'ensemble de tous les mots possibles. Chaque variable x_i est associée avec le i^{eme} niveau de l'automate et les transitions sortantes des nœuds appartenant au niveau i portent les valeurs de D_i^{orig}. δ est la fonction de transition de $Q \times \Sigma \mapsto Q$, $\delta(q, val)$ renvoie l'état atteint en appliquant la transition val dans l'état q et la paire (q, val) dénote l'arc correspondant. δ^* étend δ de telle sorte que

$$\delta^*(q,w) = \begin{cases} \delta^*(\delta(q,x),y) & \text{si } w = xy \text{ avec } x \in \Sigma \text{ et } y \in \Sigma^+; \\ \delta(q,w) & \text{si } w \in \Sigma. \end{cases}$$

On note par $\gamma(q_1, q_2)$ les valeurs de transition qui permettent de passer de l'état q_1 à l'état q_2. On notera également $|A|$, le nombre d'états de l'automate A et par $|A_i|$ le nombre d'état du i^{eme} niveau. Un exemple d'un tel automate est donné figure 7.1.

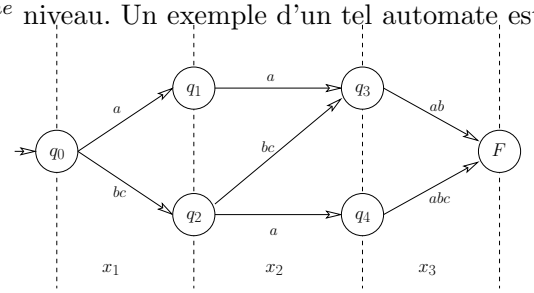


FIG. 7.1: Un automate pour trois variables x_1, x_2, x_3 de domaine $\{a, b, c\}$ représentant les tuples (a,a,a), (a,a,b), (b,b,a), (b,b,b), (b,c,a), (b,c,b), etc.

Cette représentation a déjà été utilisée dans le contexte de problèmes de satisfaction de contraintes [Vem92, Ami99]. L'automate est une manière générique de représenter un ensemble de tuples et donc de définir une contrainte en extension. Pour un ensemble S de tuples autorisés sur une séquence finie de variables X, on notera :

- A l'automate acceptant les tuples autorisés S. Autrement dit, $\mathcal{L}(A) = \{w \in \Sigma^* \mid \delta^*(q_0, w) = F\} = S$.
- $\overline{A}$ acceptant les tuples interdits correspondant à S. Autrement dit, $\mathcal{L}(\overline{A}) = \{w \in \Sigma^\ell \mid w \notin \mathcal{L}(A)\}$ où Σ^ℓ dénote les mots de taille ℓ.
- $A(X)$ l'automate projeté sur l'état courant des domaines des variables X c'est à dire que tous les arcs (q, j) pour un état q situé sur la couche i tels que $j \notin D_i$ sont éliminés.

L'automate minimal est une forme canonique, il est unique pour un ordre donné des variables. Un automate est minimal si il n'y a aucun état n'est équivalent à un autre. Deux états sont équivalents si et seulement si ils définissent le même langage droite : $\overrightarrow{\mathcal{L}}$, i.e. ils ont le même ensemble de chaînes permettant d'atteindre l'état final. Comme nous considérons un automate en couches, on peux efficacement minimiser l'automate par une approche *bottom-up* basée sur une définition récursive du langage droite d'un état :

$$\overrightarrow{\mathcal{L}}(q) = \{a\overrightarrow{\mathcal{L}}(\delta(q,a)) \mid a \in \Sigma \wedge \delta(q,a) \neq \bot\} \cup \begin{cases} \{\varepsilon\} & \text{si } q \in F; \\ \emptyset & \text{sinon.} \end{cases}$$

De manière intéressante, la représentation des tuples autorisés ou interdits est équivalente en terme de taille pour l'automate minimal. On peut facilement montrer que les nombres d'états de A et $\overline{A}$ diffèrent d'au plus de ℓ états.

Propriété 7.2.1 *Si A et $\overline{A}$ sont minimaux alors $abs(|A| - |\overline{A}|) < \ell$.*

Schéma de la preuve : On peut montrer comment construire $\overline{A}$ à partir de A (voir Figure 7.2). En premier lieu, on change l'état final de A en état poubelle de telle sorte que les tuples autorisés de A sont interdits. Deuxièmement, les tuples interdits de A doivent être reconnus et toutes les transitions manquantes (celles qui définissent implicitement des mots non reconnus) doivent être ajoutées (les arcs en gras sur la figure 7.2). Au plus $(\ell-1)$ états sont retirés (arcs en pointillés sur la figure 7.2). En effet, un état est retiré si toutes ses transitions mènent vers l'ancien état final. Or un seul état par couche peut avoir cette propriété (ils seraient autrement équivalents puisque l'automate est minimal). De nouveau, au plus un état est ajouté par niveau (ayant toutes ses transitions menant au nouvel état final) puisque l'automate est minimal. □

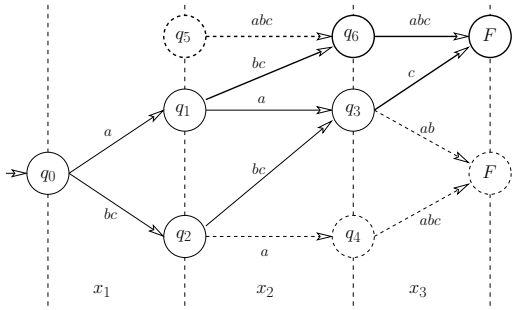


FIG. 7.2: Passage de A (arcs normaux et en pointillé) à $\overline{A}$ (arcs normaux et en gras).

On considère un ensemble de *nogoods* S (tuples interdits), nous avons choisi de représenter S par l'automate $\overline{A}$ correspondant donc, à l'ensemble des tuples autorisés. L'ajout d'un *nogood* dans un tel automate signifie donc le retrait du mot correspondant du langage reconnu par l'automate. Il est en effet plus facile de raisonner sur $\overline{A}$ pour effectuer la propagation et la taille de l'automate est identique (au nombre de niveaux près).

7.2.2 Minimisation Incrémentale de l'automate

Nous introduisons ici brièvement deux stratégies de minimisation de l'automate. L'objectif est de maintenir incrémentalement l'automate des tuples autorisés à l'ajout de nouveaux *nogoods* (retrait du mot correspondant du langage reconnu par l'automate). La minimisation incrémentale est basée sur [DMWW00].

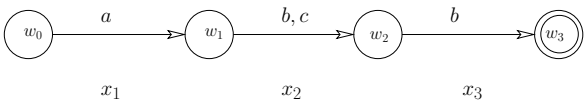
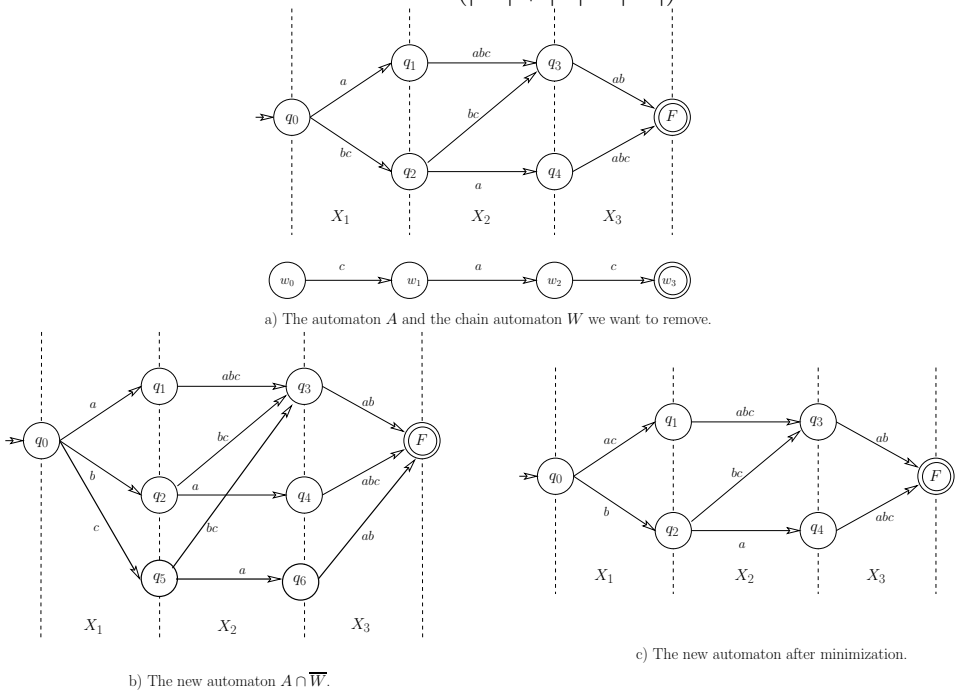


FIG. 7.3: Automate chaîne reconnaissant le *nogood* généralisé $(x_1 = a) \wedge (x_2 \neq a) \wedge (x_3 = b)$ with $\Sigma = \{a, b, c\}$.

L'alphabet Σ est de taille finie donc chaque déduction $(x \neq v)$ du *nogood* peut être remplacée par $\{x \in \Sigma \setminus v\}$. Soit w, le mot à retirer de $\mathcal{L}(A)$. le retrait du mot

w de $\mathcal{L}(A)$ implique de construire un nouvel automate $A \cap \overline{W}$ avec W l'automate chaîne reconnaissant w (voir figure 7.3). W est construit en utilisant le même ordre des variables que A de telle sorte que $\delta^*(w_0, w)$ soit l'état final.

L'algorithme procède en deux étapes (dépeintes figure 7.4). On procède d'abord au calcul de $A \cap \overline{W}$: la différence principale avec les méthodes classiques de minimisation incrémentale pour les automates acycliques réside dans le fait qu'on retire une chaîne au lieu de l'ajouter et que les *nogoods* généralisés représentent plus d'une seule chaîne. Dans un second temps, on procède à la minimisation incrémentale en exploitant le caractère minimal de l'ancien automate et sa nature en couche. La complexité temporelle du retrait et de la minimisation est de $O(|W| + |\Sigma| \times |W|)$.



a) The automaton A and the chain automaton W we want to remove.

b) The new automaton $A \cap \overline{W}$.

c) The new automaton after minimization.

FIG. 7.4: Processus de minimisation.

L'ajout d'un *nogood* w peut ajouter au plus $|w|$ états à l'automate (le pire des cas étant en l'absence de minimisation ou à l'étape (b) de la figure 7.4). Ce n'est en revanche pas vrai pour un *nogood* généralisé. En effet, un automate chaîne est déjà une représentation compacte d'un nombre exponentiel de mots. Rien ne garantit que leur compilation commune sera plus compacte. De plus, il semble très difficile de prédire un tel comportement puisqu'il est très difficile de prédire la taille d'un automate pour un langage donné. Dans le pire des cas, l'ajout d'un *nogood* généralisé peut entraîner l'ajout de $\sum_{k=2}^{l-1} |A_k|$ nouveaux états. L'automate peut au pire doubler de taille à chaque *nogood*.

Deux stratégies de gestion des *nogoods* sont envisageables : la compilation dynamique des *nogoods* au moment où ils sont découverts ou leur compilation à intervalles réguliers.

La compilation dynamique des *nogoods* sur le principe précédent s'est révélée très difficile pour les trois raisons suivantes :

1. dans le cas de *nogoods* généralisés, l'automate peut s'avérer plus large (en nombre d'états) que la somme des états des automates chaînes représentant les *nogoods* individuellement. Il semble donc critique de pouvoir sélectionner un sous-ensemble des *nogoods* qui partagent beaucoup de déductions ;
2. deuxièmement, la taille de l'automate est fortement dépendante de l'ordre des variables. Comme les *nogoods* sont découverts un à un, leur compilation dynamique dans l'automate impliquerait de pouvoir le re-ordonner dynamiquement pour optimiser cet ordre en s'appuyant sur des heuristiques ;
3. enfin, la taille de l'automate peut s'accroître très rapidement et ne décroître que lorsque de nombreux *nogoods* partagent suffisamment de déductions. Ces tailles intermédiaires peuvent être catastrophiques.

En repoussant la compilation, on peut espérer pouvoir exploiter des heuristiques pour sélectionner un sous-ensemble des *nogoods* qui mènerait à un automate plus compact, de chercher un meilleur ordre sur les variables (basé sur des idées similaires que dans le cas des ROBDD [Bry86]) comme un meilleur ordre de minimisation incrémentale au sein des *nogoods* pour éviter des tailles intermédiaires critiques [Mih98]. Les difficultés de la compilation dynamique seront examinés expérimentalement en section 7.4 sur des ensembles réels de *nogoods*.

7.3 Propagation des nogoods

7.3.1 Filtrage sur l'automate

Pesant [Pes04] introduit un algorithme de filtrage pour une contrainte globale définie par un langage régulier. Les principes de cet algorithme sont rappelés ici mais nous laissons le lecteur consulter [Pes04] pour une description formelle. Cet algorithme peut être utilisé pour réaliser l'arc-consistance sur A.

L'idée centrale est de maintenir l'ensemble Q_{ij} des états agissant comme supports de chaque paire variable/valeur (x_i, v_j). Un état q de la couche i est considéré comme un support de (x_i, v_j) tant qu'il existe un chemin de q_0 à q et de $\delta(q, v_j)$ à F dans A. Une fois que Q_{ij} est vide, la valeur j est retirée de la variable x_i.

Exemple 12 *États supports*

Sur la figure 7.1, on a ainsi $Q_{1a} = Q_{1b} = Q_{1c} = \{q_0\}$, $Q_{2a} = \{q_1, q_2\}$, $Q_{2b} = Q_{2c} = \{q_2\}$, $Q_{3a} = Q_{3b} = \{q_3, q_4\}$ et $Q_{3c} = \{q_4\}$.

La propagation incrémentale est assurée en maintenant des structures de données restaurables au *backtrack* concernant les arcs entrants et sortants de chaque nœud ainsi que ses degrés entrants et sortants. Chaque fois qu'une valeur j est retirée de i, le degré des état contenus dans Q_{ij} est décrémenté. Si l'un deux atteint zéro, cette information est propagée à tous les nœuds connectés (les prédécesseurs s'il s'agit du degré sortant et les successeurs pour le degré entrant) en décrémentant leur degré et en maintenant les listes Q_{ij} correspondantes.

Exemple 13 *Filtrage sur l'automate*

Sur la figure 7.1, si les valeurs a et b sont retirées de x_3, le degré sortant de q_3 tombe à zéro et ses arcs entrants sont examinés. Les états q_1 et q_2 sont retirés de Q_{2a}, Q_{2b} et Q_{2c} en itérant sur les arcs entrants. Comme Q_{2b} et Q_{2c} sont mis à jour à $\emptyset$, les valeurs b et c sont retirées de x_2. De plus, le degré sortant de q_1 et q_2 est décrémenté et le processus continue puisque le degré de q_1 tombe à zéro si bien que a est retirée de x_1.

7.3.2 Filtrage expliqué sur l'automate

Dans un cadre expliqué, toute déduction doit être associée à une explication généralisée. Il est donc indispensable d'expliquer le filtrage provenant des *nogoods* compilés dans l'automate. L'explication du filtrage effectué par [Pes04] est faite en expliquant pourquoi un état ne peut pas atteindre F (Algorithme 15) et pourquoi un état ne peut pas être atteint depuis q_0 (Algorithme 16).

Procédure : explainOut(State q, int i)

```
1: Explanation e ← ∅ ;
2: if is_explained(q) is false then
3:     for all j such that δ(q, j) ≠ null do
4:         if j ∈ D_i then e ← e ∪ expl(δ(q, j)) ;
5:         else e ← e ∪ expl(x_i ≠ j) ;
6:     end for
7:     is_explained(q) ← true ;
8:     expl(q) ← e ;
9: end if
```

Algorithme 15: Explique pourquoi q ne peut pas atteindre F. Calcul de $expl(q)$ pour un état q situé sur la couche i dès que le degré sortant de q tombe à zéro.

Procédure : explainIn(State q, int i)

```
1: Explanation e ← ∅ ;
2: if is_explained(q) is false then
3:     for all (p, j) such that δ(p, j) == q do
4:         if j ∈ D_{i−1} then e ← e ∪ expl(p) ;
5:         else e ← e ∪ expl(x_{i−1} ≠ j) ;
6:     end for
7:     is_explained(q) ← true ;
8:     expl(q) ← e ;
9: end if
```

Algorithme 16: Explique pourquoi q ne peut pas être atteint depuis q_0. Calcul de $expl(q)$ pour un état q situé sur la couche i dès que le degré entrant de q tombe à zéro.

Une explication $expl(q)$ et un booléen restaurable au *backtrack*, $is\_explained(q)$, sont associés à chaque q de l'automate. $expl(q)$ enregistre pourquoi q est invalide, c'est à dire pourquoi il ne peut pas être sur un chemin de q_0 à F. $is\_explained(q)$ est vrai si l'invalidité de q a déjà été expliquée et une explication valide $expl(q)$ est disponible. Comme plusieurs explications existent souvent, il est critique de ne pas écraser une explication valide existante parce que l'explication elle-même n'est pas restaurable et qu'on ne retrouverait pas l'explication originelle au *backtrack*. Ce problème se pose souvent quand le calcul de l'explication n'est pas synchrone avec son utilisation pour justifier un retrait de valeur [Roc05].

Une valeur j de x_i est filtrée parce que Q_{ij} est vide. Ce filtrage s'explique parce que chaque état q qui faisait partie de la liste originelle d'états supports de (x_i, v_j) (notée $Qinit_{ij}$) n'est plus sur un chemin de q_0 à F, donc soit q est lui-même invalide, soit $\delta(q, j)$ est invalide (Algorithme 17).

procédure : prune(int i, int j)

```
1: Explanation e ← ∅ ;
2: for all q in Qinit_ij do
3:     if is_explained(q) then e ← e ∪ expl(q) ;
4:     else e ← e ∪ expl(δ(q, j)) ;
5: end for
6: on retire la valeur j de x_i à cause de e ;
```

Algorithme 17: explique le filtrage de la valeur j du domaine de x_i.

Le champ $expl(q)$ est calculé pour chaque état q de la manière suivante :

– Pour expliquer pourquoi un état q_k au niveau i ne peut pas être atteint depuis q_0, les prédécesseurs sont divisés en deux ensembles $rpred$ et $\overline{rpred}$. Les prédécesseurs $rpred$, qui peuvent être atteints depuis q_0, et ceux, $\overline{rpred}$, qui ne le peuvent pas. Pour chaque prédécesseur p de q_k, soit il appartient à $\overline{rpred}$ et l'explication $expl(p)$ attachée à p est utilisée, soit il appartient à $rpred$ et les transitions menant à q_k depuis p ($\gamma(p, q_k)$) ont été retirées du domaine de x_{i-1}. L'algorithme 16 est appelé chaque fois que le degré entrant de q_k tombe à zéro et calcule $expl(q_k)$:

$$expl(q_k) = expl(q_0 \nRightarrow q_k) = \bigcup_{p \in rpred} expl(x_{i-1} \neq \gamma(p, q_k)) \cup \bigcup_{p \in \overline{rpred}} expl(p).$$

– De manière complètement symétrique, l'état q_k ne peut pas atteindre F parce que soit ses successeurs ne peuvent pas atteindre F, soit la valeur menant à un état qui le pourrait est manquante. L'algorithme 15 est appelé chaque fois que le degré sortant de q_k tombe à zéro et calcule $expl(q_k)$:

$$expl(q_k) = expl(q_k \nRightarrow F) = \bigcup_{s \in rsucc} expl(x_i \neq \gamma(q_k, s)) \cup \bigcup_{s \in \overline{rsucc}} expl(s).$$

7.3.3 Filtrage allégé

Le but de l'automate est de compiler de gros ensembles de *nogoods* pour atteindre une meilleure gestion de l'espace que les *watched literals*. La propagation incrémentale est dans un sens très gourmande en mémoire puisqu'elle exige deux listes doublement chaînées (arcs entrants et sortants) et deux entiers (degré entrant et sortant) par état qui soient restaurables au *backtrack*. Elle s'appuie aussi sur une liste Q_{ij} d'états par paire de variable/valeur. En premier lieu, nous avons renoncé à maintenir les listes doublement chaînées. Si le nombre d'arc sortants est borné par la taille de l'alphabet (la taille maximum des domaines), le nombre d'arcs entrants peut atteindre le nombre d'états de la couche précédente ce qui n'est pas raisonnable dans notre cas. Cet algorithme est dénoté $Aut0$ par la suite.

De plus, nous avons examiné différents compromis qui perdent la propagation incrémentale en temps constant à chaque retrait de valeur mais économisent la mémoire :

– Le degré exact de chaque nœud n'est pas indispensable, on a simplement besoin de savoir si ce degré est nul ou non. On peut donc stocker un arc témoin (non

restaurable au *backtrack*) au lieu de stocker le degré. Un arc valide à la profondeur k de l'arbre de recherche est en effet également valide aux profondeurs inférieures à k. L'algorithme $Aut0$ incluant cette modification est noté $Aut1$.
– Comme Pesant [Pes04] le souligne, on n'a pas réellement besoin de tous les supports et seulement l'un d'entre eux peut être conservé en mémoire. Quand ce support devient invalide, on en cherche un autre parmi $Q_{init_{ij}}$. L'algorithme $Aut1$ incluant cette modification est noté $Aut2$.

$Aut0$:	Algorithme complet de Pesant sans le maintien des listes chaînées des arcs entrants et sortants.
$Aut1$:	$Aut0$ + maintien d'arcs témoins non restaurables à la place des degrés.
$Aut2$:	$Aut1$ + maintien d'un unique état-support restaurable.

TAB. 7.1: Variante de l'algorithme de filtrage plus économe en mémoire

7.4 Résultats expérimentaux

Nous étudions dans un premier temps l'intérêt de stocker dans un automate un large ensemble de tuples définissant une contrainte en extension comparé à l'utilisation des algorithmes d'arc consistance classique [BR97]. Ensuite, nous rendons compte des résultats obtenus sur la propagation de *nogoods* par une technique de *watched literals* ainsi qu'un automate pour des *nogoods* simples.

Pour les différentes raisons abordées en section 7.2.2, la compilation dynamique de *nogoods* généralisés est beaucoup trop coûteuse pour être compétitive. Nous montrerons donc le comportement parfois catastrophique de l'automate sur des jeux de *nogoods* réels et les perspectives d'amélioration de cette situation. Les mots croisés et les problèmes d'allocation de fréquence constituent notre jeu de tests. Toutes les expérimentations sont faites sur un Pentium 4 3GHz avec 1 GB de RAM sous linux et le solveur de contraintes choco (*choco-solver.net*).

7.4.1 Évaluation de l'automate : stockage et filtrage

Le problème des mots croisés consiste à remplir une grille donnée en utilisant des mots à partir d'un dictionnaire de référence de telle sorte que chaque mot soit utilisé au plus une fois. Les contraintes doivent donc stocker de gros ensembles de tuples structurés correspondant aux mots autorisés par le dictionnaire.

Une variable x_i de domaine $D(x_i) = \{a, b, \ldots, z\}$ est associée avec chaque case libre de la grille. Une contrainte est posée par mot, c'est à dire, par séquence contiguë de lettres dans le puzzle. Les tuples autorisés sont définis par tous les mots du dictionnaire de référence de la taille correspondante. Un mot ne peut être utilisé qu'une seule fois et une contrainte *not-equal* est ajoutée entre toutes paires de mots de même taille. Nous comparons ici deux approches pour maintenir l'arc-consistance :

1. La GAC introduite par [BR97][2]. L'accès direct aux supports de chaque paire de variable-valeur est assuré par une structure de données partagée par toutes les contraintes. Une liste chaînée de mots de taille k qui possèdent la lettre l à la position p est stockée dans un tableau tri-dimensionnel appelé `supports[l][p][k]`. GAC est ensuite réalisé par l'approche GAC2001 en stockant par entier restaurable au *backtrack* le support courant (cet entier indique l'index du mot dans la liste correspondante de `supports`).
2. La propagation décrite précédemment. Chaque dictionnaire de taille k (tous les mots de taille k) est compilé dans un automate minimal noté $auto_k$. On note Q_{ij}^k, l'ensemble Q_{ij} pour l'automate $auto_k$.

Nous avons considéré les jeux de données de [CvB01] dont les instances varient d'une taille 5×5 jusqu'à 23×23 et proviennent des puzzles du *Herald Tribune*. Nous avons utilisé le dictionnaire *words* qui rassemble 45000 mots (un autre dictionnaire, *UK*, inclut 220000 mots mais les grilles sont souvent plus difficiles avec le dictionnaire *words* parce qu'il y a moins de solutions). La table 7.2 résume les résultats obtenus dans une limite d'une heure pour les différentes versions de l'algorithme de filtrage basé sur l'automate et GAC2001.

	Mac-Aut0		**Mac-Aut1**		**Mac-Aut2**		**Mac-GAC**	
Instances	**Tps(s)**	**Nœuds**	**Tps(s)**	**Nœuds**	**Tps(s)**	**Nœuds**	**Tps(s)**	**Nœuds**
05.01(dico :words)	**0,2**	30	0,2	30	0,3	30	0,3	30
15.01(dico :words)	1,3	75	1,3	75	1,3	75	**0,9**	75
15.02(dico :words)	**12,1**	872	13,7	872	16,6	872	25,5	872
15.07(dico :words)	**321,2**	22859	366,4	22859	554,1	22859	923,4	22859
19.02(dico :words)	**83,8**	17511	95,2	17511	214,6	17511	253,2	17511
19.05(dico :words)	> 1h	**1213314**	> 1h	1066428	> 1h	728579	> 1h	500871
21.03(dico :words)	**63,6**	13017	73,2	13017	203,5	13017	248,7	13017
21.06(dico :words)	> 1h	**494848**	> 1h	416718	> 1h	261107	> 1h	131916
21.07(dico :words)	**20,2**	1825	23,4	1825	34,5	1825	49,5	1825
23.07(dico :words)	> 1h	**256456**	> 1h	227168	> 1h	118092	> 1h	102668

TAB. 7.2: Résultats des trois variantes de l'algorithme de filtrage de l'automate ainsi que GAC2001 sur quelques problèmes de mots croisés.

La propagation initiale de l'automate est coûteuse (l'initialisation de Q_{ij}). GAC2001 est donc plus rapide sur les instances résolues en un faible nombre de nœuds. Cependant, sur les instances dures, l'automate est environ deux à trois fois plus rapide que GAC2001. Les mots du dictionnaire sont des tuples très structurés (qui partagent beaucoup de préfixes et suffixes), c'est à dire $|Q_{pl}^k| < |$`supports[l][p][k]`$|$. La méthode `seekNextSupport` qui est la base de tout algorithme de GAC en extension est plus rapide sur Q_{pl}^k que sur `supports[l][p][k]`.

Si on examine la consommation mémoire, l'automate est plus compact (1,70 Mo) que la structure de donnée `supports[l][p][k]` (3,98 Mo) pour stocker tous les mots de taille 8 du dictionnaire *words*. Cependant, les structures de données nécessaires au filtrage de l'automate consomment plus de mémoire que celle du GAC. Nous avons mesuré la quantité de mémoire consommée tous les 500 backtracks par les quatre approches. La valeur moyenne est rapportée table 7.3. Entre les trois versions de la

[2]La multi-directionnalité n'est pas implémentée dans notre contrainte de GAC .

instance	*Aut*0 (Mo)	*Aut*1 (Mo)	*Aut*2 (Mo)	GAC (Mo)
15.07(dico :words)	31.8	21.1	16.1	19.7
19.02(dico :words)	37.2	25.2	17.6	19.1
21.03(dico :words)	51.6	33.9	22.2	23.7

TAB. 7.3: Consommation moyenne de mémoire en Mo mesurée tous les 500 *backtraks*.

contrainte d'automate, le meilleur compromis espace/temps semble être *Aut*1. Cependant *Aut*2, qui est proche de la GAC d'après son temps d'exécution, demande moins de mémoire.

7.4.2 Enregistrement de *nogoods*

Quatre approches ont été comparées sur les problèmes de mots croisés et de RLFAP (*Radio Link Frequency Allocation Problems*) : MAC-CBJ ; MAC-CBJ+S désigne MAC-CBJ combiné avec l'enregistrement de *nogoods* standards et propagés par des *watched literals* ; MAC-CBJ+G est MAC-CBJ avec des *nogoods* généralisés propagés par *watched literals*. Nous ne présentons pas les résultats de MAC qui sont dominés par les performances de MAC-CBJ sur ces deux jeux de tests (cf. section 6.4.3 pour la comparaison MAC, MAC-CBJ sur le RLFAP et à titre d'exemples les instances 19.05, 23.07, 21.06 présentées précédemment et toutes résolues par MAC-CBJ en quelques secondes).

Les grilles sont particulièrement structurées et quelques affectations peuvent séparer le problème très rapidement. On s'en rend bien compte en examinant par exemple la grille *puzzle21* appartenant aux problèmes de Ginsberg :

```
(_ _ _ _ * _ _ _ * _ _ _ _)
(_ _ _ _ * _ _ _ * _ _ _ _)
(_ _ _ _ * _ _ _ * _ _ _ _)
(_ _ _ _ _ _ * _ _ _ _ _ _)
(* * * _ _ _ * _ _ _ * * *)
(_ _ _ _ _ * * * _ _ _ _ _)
(_ _ _ * * * * * * * _ _ _)
(_ _ _ _ _ * * * _ _ _ _ _)
(* * * _ _ _ * _ _ _ * * *)
(_ _ _ _ _ _ * _ _ _ _ _ _)
(_ _ _ _ * _ _ _ * _ _ _ _)
(_ _ _ _ * _ _ _ * _ _ _ _)
(_ _ _ _ * _ _ _ * _ _ _ _)
```

Mots croisés. Les mots croisés ont été utilisés par Ginsberg [Gin93] pour montrer l'intérêt de DBT et puis par [CvB01, KB05] pour MAC-CBJ et l'enregistrement de *nogoods*. La limite de temps est placée à 2 heures et *Aut*1 est utilisé pour les contraintes correspondant à chaque mot. Les résultats sont reportés table 7.4. L'utilisation de *watched literals* implique un léger surcoût par rapport à MAC-CBJ sur les instances faciles. En revanche, nous avons pu retrouver les résultats de [KB05] qui montrent que les *nogoods* généralisés se rentabilisent très bien sur ce problème.

RLFAP. Les *nogoods* généralisés sont de nouveau effectifs sur ce problème (voir table 7.5), même si les résultats de [BHLS04] s'appuyant sur $dom/Wdeg$ restent meilleurs.

	MAC-CBJ		MAC-CBJ + S		MAC-CBJ + G	
	Tps(s)	**Nœuds**	**Tps(s)**	**Nœuds**	**Tps(s)**	**Nœuds**
15.02(dico :words)	**29,7**	314	46,2	314	43,8	**303**
15.07(dico :words)	**841,6**	18182	1255,6	17918	894,3	**11172**
19.02(dico :words)	**10,8**	264	15,6	264	14,1	**219**
19.05(dico :words)	110,6	2104	64,2	1182	**47,9**	**727**
21.03(dico :words)	**11,8**	292	18,4	292	18,3	**281**
21.06(dico :words)	177,0	2707	265,8	2677	**164,8**	**1879**
21.07(dico :words)	**44,9**	1168	72,6	1157	63,5	**957**
21.04(dico :words)	> 2h	87677	> 2h	73342	> 2h	47527
23.07(dico :words)	**45,3**	954	74,7	892	61,7	**584**
21.05(dico :words)	> 2h	176226	> 2h	102057	**722,8**	9407
21.10(dico :words)	> 2h	74718	> 2h	47346	> 2h	43440
15.04(dico :words)	> 2h	171029	> 2h	105069	**1268,6**	**16315**
15.06(dico :words)	> 2h	150552	> 2h	94936	**5529,1**	**50554**
15.10(dico :words)	> 2h	153260	> 2h	99929	**1634,6**	**14153**
19.03(dico :words)	> 2h	123770	> 2h	88461	> 2h	58632
19.04(dico :words)	> 2h	303559	> 2h	222910	**55,2**	**2294**
19.07(dico :words)	> 2h	462297	> 2h	289966	**131,9**	**4312**
21.01(dico :words)	> 2h	86857	> 2h	53418	> 2h	36336
23.03(dico :words)	> 2h	80810	> 2h	50998	> 2h	42919
23.04(dico :words)	> 2h	74641	> 2h	37265	> 2h	23723
23.05(dico :words)	**11,7**	239	17,1	254	15,7	**226**

TAB. 7.4: Résultats de l'enregistrement de *Nogoods* pour les mots croisés.

	MAC-Cbj		MAC-Cbj + S		MAC-Cbj + G	
	Tps(s)	**Nœuds**	**Tps(s)**	**Nœuds**	**Tps(s)**	**Nœuds**
`scen02-05-24`	**0,3**	104	0,9	104	0,4	104
`scen02-05-25`	**3,0**	610	5,2	610	3,1	**360**
`scen03-05-10`	1659,2	572507	> 2h	343927	**123,8**	**11575**
`scen03-05-11`	> 2h	3506415	> 2h	776095	> 2h	155008
`scen11-05-00`	6,4	1207	8,3	1207	**3,8**	**622**
`scen06-02-00`	73,9	68669	164,2	61866	**5,3**	**1854**
`scen07-01-04`	**0,1**	202	0,2	202	0,2	**201**
`scen07-01-05`	**0**	26	0,1	26	0,1	26
`graph08-05-10`	> 2h	1722485	> 2h	491970	> 2h	175079
`graph08-05-11`	> 2h	1300390	> 2h	494286	**46,6**	**6906**

TAB. 7.5: Résultats de l'enregistrement de *Nogoods* pour le RLFAP.

Les *nogoods* standards peuvent être compilés efficacement car ils partagent tous des variables communes reliés au chemin de décision et sont obtenus en suivant l'ordre de la recherche arborescente. Les benchmarks précédents montrent néanmoins, que leur intérêt est très limité face aux *nogoods* généralisés. Nous avons donc davantage porté notre intérêt sur la compilation des *nogoods* généralisés. Comme leur compilation dynamique s'est révélée très difficile, nous avons procédé à une étude détaillée de cette situation sur des jeux de *nogoods* réels et nous rendons compte ici des principales conclusion.

Les figures 7.5 montrent l'évolution de la taille de l'automate (nombre d'états) à chaque ajout de *nogoods*. Les données sont issues de deux instances du RLFAP (`scen03-05-11` et `scen11-05-03`).

Nous étudions le comportement de la compilation incrémentale basique *dyn* pour laquelle les *nogoods* sont ajoutés un par un dans l'ordre de leur découverte. De plus, un ordre arbitraire des variables est utilisé pour construire l'automate (ordre donné par

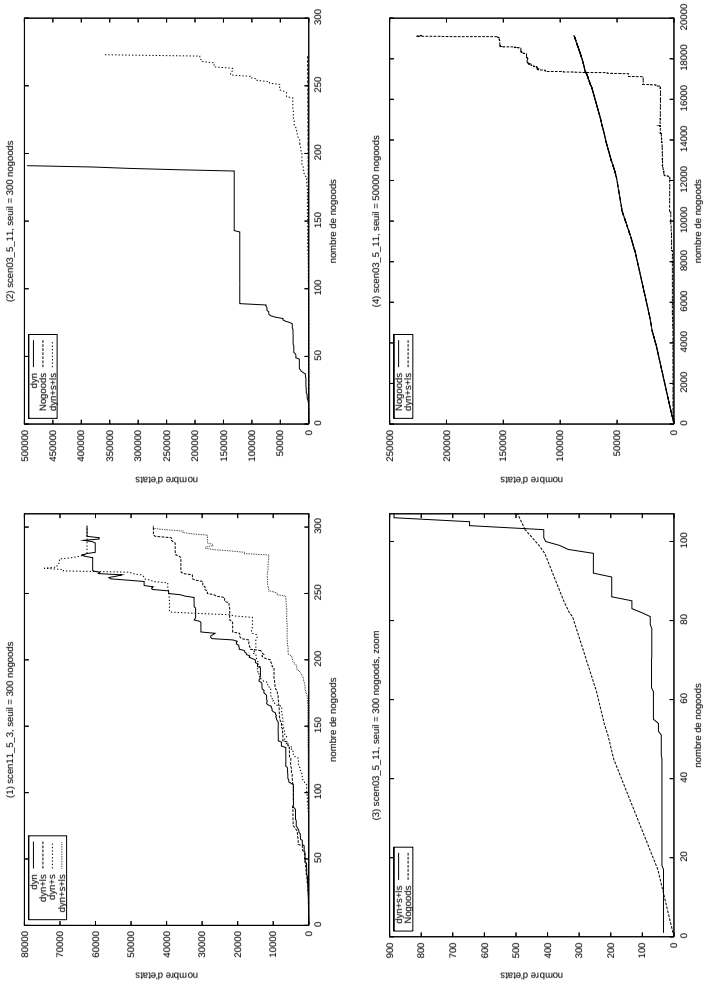


FIG. 7.5: Comportement de l'automate durant la compilation de nogoods généralisés.

celui des *nogoods*). L'influence de deux paramètres est étudiée sur ce schéma de base :

- $+ls$ signifie que l'ordre des variables utilisé pour définir l'automate est optimisé avant la compilation à l'aide d'une heuristique s'inspirant des heuristiques pour ROBDD ;
- $+s$ signifie que les nogoods sont triés lexicographiquement avant la compilation. Ils sont ensuite compilés un à un dans cet ordre ;

Ainsi $+ls$ affecte la taille finale de l'automate (dépendante de l'ordre des couches donc des variables) alors que $+s$ affecte uniquement ses tailles intermédiaires. $+s$ et $+ls$ exigent que la compilation soit repoussée jusqu'à un certain **seuil** de *nogoods* appris pour pouvoir les analyser avant de commencer leur compilation incrémentale. Les quatre figures suivantes présentent des résultats pour des seuils de 300 et 50000 *nogoods*. La courbe *Nogoods* correspond à la taille des *nogoods* dans leur forme non compilée[3].

La figure (1) montre deux paires de courbes correspondant à deux tailles finales différentes de l'automate suivant l'utilisation de $+ls$. Au sein de chaque paire, deux courbes se distinguent selon l'utilisation de $+s$. Nous souhaitons ici faire deux observations :

1. l'ordre des variables est critique pour contrôler la taille finale de l'automate.
2. l'ordre de compilation des *nogoods* est critique pour contrôler sa taille intermédiaire maximum. On peut voir ainsi sur cette figure un cas où il existe une taille intermédiaire plus grande que la taille finale pour un autre ordre de compilation (c'est de manière surprenante, l'ordre lexicographique qui produit ici ce pic).

La figure (2) exhibe un cas dramatique pour l'automate où le schéma de base *dyn* produit un automate de près de 500000 états en moins de 200 *nogoods*. $dyn + s + ls$ est clairement une amélioration en repoussant le moment où l'automate explose à 270 *nogoods*. L'automate est plus compact uniquement sur les 100 premiers *nogoods* (zoom de la figure (3)).

Enfin la figure (4) repousse le seuil à 50000 *nogoods* pour donner l'opportunité à $+s$ de trier des *nogoods* très proches. On est alors en mesure de compiler avantageusement les 16000 premiers *nogoods* avant d'observer la même explosion de l'automate.

L'automate peut donc se révéler avantageux pour des sous-ensembles de *nogoods* compilés à des intervalles de temps assez importants de manière à opérer une sélection judicieuse des *nogoods*. Ces résultats montrent que l'approche initiale de compilation à la volée des *nogoods* est clairement hors de question.

7.5 Conclusion

Nous avons envisagé une autre approche que les *watched literals* pour stocker et propager les *nogoods* de manière à surmonter leurs besoins exponentiels en mémoire. Les limitations et les avantages de l'approche ont été abordés en détail. Nos résultats montrent que l'approche originelle de compilation dynamique est impossible à réaliser. Nos investigations limitées sur l'exploitation de l'ordre des variables de l'automate comme sur l'ordre de minimisation offrent tout de même des perspectives intéressantes.

[3]La somme des états des automates chaînes représentant chaque *nogood*.

Par ailleurs, nous avons fourni un algorithme de calcul des explications associées au filtrage de l'automate et montré que l'automate pouvait être très avantageux par rapport à un algorithme d'arc-consistance traditionnel pour les contraintes en extension.

Enfin, nous mettrons en œuvre les idées présentées ici, c'est-à-dire *nogoods* généralisés et compilation par un automate, dans le cadre du problème de MOSP (*Minimum Open Stack Problem*) au chapitre 10.

De nombreux éléments restent à approfondir sur les *nogoods*. Nous n'avons pas abordé ici la question du *nogood* à conserver et retenu arbitrairement le premier *nogood* valide obtenu. Nous n'avons pas non plus examiné comment mettre en place un *backjumping* à l'image de celui effectué en SAT et décrit section 3.4.2.3.

Ces résultats ont publiés dans un atelier de travail [RCOJ06].

Chapitre 8

Des explications pour la décomposition

Dans le but de surmonter les problèmes de mémoire exponentielle, nous avons cherché à compiler les *nogoods* dans la section précédente plutôt que de les oublier comme le suggèrent les approches actuelles du type *i-order relevance bounded learning* (voir section 3.4). Nous abordons dans ce chapitre une autre forme limitée d'enregistrement de *nogoods* guidée par une décomposition du problème.

Les liens qui existent entre les explications et la dualité en optimisation sont mis ici en évidence et les explications apparaissent comme une information duale. Ces liens suggèrent que des méthodes issues de la Recherche Opérationnelle s'appuyant sur la dualité pourraient être mises en œuvre en PPC. Nous examinons ainsi le cas de la décomposition de Benders abordée section 4.2.2 dans le cas linéaire. La décomposition de Benders a jusqu'ici été proposée comme un bon schéma de combinaison pour la programmation linéaire et la PPC et implémentée de manière générique dans les solveurs hybrides PLNE/PPC telle que Eclipse [EW01, XEW04]. Nous pensons qu'elle présente aussi de l'intérêt dans le seul cadre PPC.

8.1 Cadre théorique

Nous abordons dans un premier temps la question de la dualité en programmation par contraintes et la formulation de la décomposition dans le cadre générique. Nous donnerons les formulations des problèmes maîtres et esclaves (sous-problèmes) ainsi que les hypothèses nécessaires à la décomposition.

8.1.1 Dualité en programmation par contraintes

Nous avons abordé la dualité dans le cadre de la programmation linéaire section 2.1.1.2. C'est une notion délicate à définir dans un cadre plus général. Dans le contexte de l'optimisation combinatoire, on peut voir (de manière informelle) la dualité comme une relation mathématique entre deux problèmes d'optimisation, dits **primal** et **dual** qui constituent toujours des bornes l'un pour l'autre et se présentent comme des certificats de faisabilité et d'optimalité du même problème. On parle ainsi de *dualité faible*

quand les valeurs optimales du primal et du dual ne coïncident pas et de *dualité forte* dans le cas contraire.

John Hooker dans [Hoo06] a abordé très récemment la question générale de la dualité en optimisation en analysant les différentes formes de dualité et leurs relations. [Hoo06] explique que parmi les très nombreux duals existants, deux grands types se dégagent : les duals logiques (*inference dual*) et ceux basés sur la notion de relaxation (*relaxation duals*).

8.1.1.1 Dualité logique

La notion de dualité logique est particulièrement adaptée pour penser la dualité en PPC. Introduisons le concept de dualité logique en suivant [Hoo06]. Considérons le problème d'optimisation suivant :

$$min_{x \in D}\{f(x)|\mathcal{C}\} \tag{8.1}$$

où x dénote un ensemble de variables $(x_1, \ldots, x_n)$ et D, un ensemble de domaines associés, $f(x)$ une fonction à valeur réelle et C un ensemble de contraintes. Le dual logique de 8.1 est le problème de déduire **la meilleure borne** inférieure possible **à partir de** C dans un système de preuves donné. Ce dual s'écrit :

$$max_{P \in \mathcal{P}}\{v|\mathcal{C} \vdash^P (f(x) \geq v)\} \tag{8.2}$$

$\mathcal{C} \vdash^P (f(x) \geq v)$ signifie que la preuve P permet de déduire $f(x) \geq v$ à partir de $\mathcal{C}$. Le domaine de la variable P est donc une famille de preuves $\mathcal{P}$. Une solution du dual est donc **une preuve**, c'est une paire $(\overline{v}, \overline{P})$ telle que $P \in \mathcal{P}$ et $\mathcal{C} \vdash^{\overline{P}} (f(x) \geq \overline{v})$. Cette solution est optimale si $\overline{v} \geq v$ pour toute solution (v, P).

La famille de preuves est *complète* quand la solution optimale v^* de 8.2 est aussi celle de 8.1. La preuve P^* solution est donc une justification de l'optimalité de 8.1, le fameux certificat d'optimalité dont il est toujours question en programmation linéaire. À l'éclairage du dual logique, on peut se pencher sur le cas de la programmation linéaire dont la famille de preuves est justement une famille complète.

8.1.1.2 Liens avec la dualité linéaire

Nous avons introduit la dualité section 2.1.1.2 en partant **des contraintes** du primal pour construire **un raisonnement** permettant d'obtenir **une borne** supérieure du problème linéaire primal (qui était alors un problème de maximisation). C'est cette démarche qui conduit au problème dual et on pourra se référer à la section 2.1.1.2, pour retrouver ces éléments (indiqués en gras dans le texte) qui sont les éléments clefs du dual logique abordé précédemment. Il nous reste à mettre en évidence le système de preuves utilisé en programmation linéaire. Reformulons le problème primal de la figure 2.1 (page 13) comme un problème de minimisation :

$$min_{x \geq 0}\{cx|Ax \geq b\} \tag{8.3}$$

Son dual logique se présente sous la forme suivante :

$$max_{P \in \mathcal{P}}\{v|Ax \geq b \vdash^P (cx \geq v)\} \tag{8.4}$$

La famille de preuves utilisée pour aboutir à la formulation classique du dual linéaire est basée sur des combinaisons linéaires du système $Ax \geq b$ et la notion de domination. Ainsi $ax \geq \alpha$ domine $bx \geq \beta$ quand $a \leq b$ et $\alpha \geq \beta$. Une preuve P de $\mathcal{P}$ de $cx \geq v$ est une combinaison linéaire du système $Ax \geq b$. En notant u le vecteur colonne des coefficients définissant la combinaison linéaire, celle-çi s'écrit ${}^tAux \geq {}^tbu$. ${}^tAux \geq {}^tbu$ domine $cx \geq v$ si ${}^tAu \leq c$ et ${}^tbu \geq v$. De plus, on peut voir l'obtention de la meilleure preuve comme la maximisation de v et donc de ${}^tbu$ sous les contraintes ${}^tAu \leq c$, ce qui nous renvoie sur le problème linéaire dual traditionnel :

$$max_{u \geq 0}\{{}^tbu | {}^tAu \leq c\} \tag{8.5}$$

La preuve est ainsi constituée par les valeurs des variables duales u. On se référera à [Chv83] pour vérifier que les preuves fondées sur des combinaisons linéaires et la notion de domination forment une famille complète.

8.1.1.3 Liens avec la programmation par contraintes

Les algorithmes de filtrage sont des mécanismes d'inférence qui constituent un dual logique. Ainsi, dans le cadre d'un problème d'optimisation, la propagation à chaque nœud tente de dériver les meilleures bornes possibles pour la variable objectif. En considérant l'ensemble des contraintes de décisions DC qui caractérisent un nœud de l'arbre de recherche, le problème dual restreint à ce nœud s'écrit :

$$max_{P \in \mathcal{P}}\{v | DC \cup C \vdash^P (f(x) \geq v)\} \tag{8.6}$$

De nouveau, on utilise les contraintes afin de dériver la meilleure borne pour $f(x)$ et si l'arc-consistance est maintenue, cette meilleure borne est atteinte (dualité forte). L'énumération des valeurs de 8.1 peut se voir comme le problème primal et fournit des solutions faisables tandis que la résolution simultanée de 8.6 fournit le certificat d'optimalité.

8.1.2 Les explications, une information duale

Considérons une solution de 8.6 notée $(\overline{v}, \overline{P})$. L'examen de $\overline{P}$ permet d'identifier le sous-ensemble de $DC \cup C$ nécessaire à P. Un tel sous-ensemble correspond précisément à la définition d'une explication (section 3.1.1). Ainsi $e = expl(f(x) \geq v)$ est toujours telle que :

$$DC_e \cup C_e \vdash^{\overline{P}} (f(x) \geq \overline{v}) \tag{8.7}$$

Les explications constituent donc *une information duale* dans le sens où elles encodent la solution duale en conservant les hypothèses nécessaires au déroulement de la preuve. De la même façon, les multiplicateurs ou variables duales de la programmation linéaire encodent la preuve solution du dual.

8.1.3 Un cadre d'implémentation de la *logic based benders decomposition*

La décomposition de Benders abordée section 4.2.2 s'appuie sur l'accumulation de coupes de Benders à chaque itération i de la résolution entre problèmes maître

et esclave(s). Ces coupes justifient la sous-optimalité de l'affectation courante y^* du problème maître en éliminant de son espace de recherche un ensemble de solutions sous-optimales pour les mêmes raisons que l'affectation courante. Elles permettent au maître de converger vers la solution optimale du problème complet. Ces coupes ont la forme $z \geq h(y)$ où $h(y)$ est dans le contexte linéaire une combinaison du type $f(y) + u_i^*(a - g(y))$ qui provient de la résolution du dual de l'esclave.

Si ce dual est un dual logique (*inference dual*), la décomposition (*logic based benders decomposition* de [HO03]) exige d'analyser la preuve pour identifier $h(y)$. Hooker explique donc que ce dual logique doit être conçu pour chaque classe de sous-problème ainsi que l'algorithme permettant d'identifier $h(y)$. Il fait implicitement référence ici aux algorithmes de filtrage et aux algorithmes de calculs des explications qui sont conçus au sein des contraintes globales pour les problématiques les plus récurrentes. La propagation de contraintes est ensuite un moyen générique d'obtenir le dual logique comme son explication. L'équation 8.7 constitue dès lors une coupe de Benders.

Nous pensons à ce titre que les explications permettent de fournir une implémentation générique de la *logic based benders decomposition*. Les coupes obtenues sont donc des *nogoods* et le schéma de recherche résultant est une forme d'enregistrement de *nogoods* limité à certaines structures du problème. Seuls les *nogoods* portant sur un sous-ensemble des variables (les variables maîtres) et résultants de la résolution d'un sous-problème sont conservés. Compte tenu du large spectre d'applications qui possèdent ce type de structure maître/esclaves, la possibilité de définir ces structures au niveau de la phase de modélisation et d'appliquer un algorithme générique de décomposition semble intéressante. C'est ce que nous proposons ici.

8.1.4 Schéma de décomposition

Nous nous intéressons ici à des problèmes qui s'écrivent de la manière suivante :

$$\begin{array}{ll} \text{P :} & \text{Min } obj \\ & \text{s.t : } Ct(x, y) \\ & \text{with : } x \in D_x, y \in D_y \end{array}$$

$Ct(x, y)$ dénote un ensemble de contraintes sur les variables x, y et obj peut être égal à $\{f(x, y), f(y), 0\}$ (respectivement, une fonction objectif portant sur toutes les variables, une fonction objectif restreinte aux variables y du problème maître et un problème de satisfaction). Le problème P sera écrit $\{P_{xy}, P_y, P_0\}$ selon la fonction objectif correspondante. La décomposition est faite entre x et y. Nous supposerons donc que le problème restreint à x peut être formulé à partir de n sous-problèmes indépendants (cas d'une structure idéale) ou entretenant de faibles relations entre eux au sens de la propagation c'est à dire des impacts abordés précédemment (cas d'une structure approximative). Dans une situation idéale, les sous-problèmes doivent être aussi nombreux et indépendants que possibles pour garantir que le problème sur x soit facile. Nous faisons donc l'hypothèse d'une structure idéale ou approximative pour le sous-problème. P est ainsi supposé avoir une structure idéale tandis que $P^{'}$ désigne un sous-problème avec une structure approximative sur les variables x (on a ainsi de la même façon $\{P^{'}_{xy}, P^{'}_y, P^{'}_0\}$).
On note par la suite :

– obj^* la meilleure solution connue pour obj ;

- K, le nombre de sous-problèmes ;
- cut_i la coupe de Benders correspondant à 8.7 ajoutée au problème maître à la $i^{\text{ème}}$ itération. On peut la voir comme une contrainte associée au *nogood* nog_e sous sa forme clausale :

$$\bigvee_{x_k=j \in DC} x_k \neq j$$

Dans le cas d'une explication généralisée, elle correspond au nogood généralisé (Δ, DC) :

$$(\bigvee_{x_k \neq j \in \Delta} x_k = j) \vee (\bigvee_{x_k=j \in DC} x_k \neq j)$$

Le problème maître à l'itération I et les K sous-problèmes ont la forme générique suivante :

$$\begin{array}{lll|ll} MP: & \text{Min } z & & SP^k: & \text{Min } sz_k \\ & \text{s.t} : Ct(x,y) & & & \text{s.t} : Ct(x_k, \overline{y}) \\ & z < obj^* & & & x_k \in D_x \\ & \forall i < I, cut_i & & & \\ & y \in D_y & & & \end{array}$$

Les algorithmes de décomposition 18 et 19 (pages 97 et 98) que nous abordons ensuite s'appuient sur certaines hypothèses. La première concerne le problème maître.

Définition 8.1.1 *Le problème maître MP est dit valide si il constitue une borne inférieure de P :* $\forall y,\ z \leq obj$.

Pour appliquer ces algorithmes dans un cadre générique, on peut simplement restreindre la fonction objectif aux variables y du maître et obtenir une relaxation dans le cas de MP_y et MP_0 :

$$\begin{array}{lll|ll} MP_y: & \text{Min } f(y) & & MP_0: & \text{Min } 0 \\ & \text{s.t} : Ct(x,y) & & & \text{s.t} : Ct(x,y) \\ & f(y) < obj^* & & & \forall i < I, cut_i \\ & \forall i < I, cut_i & & & y \in D_y \\ & y \in D_y & & & \end{array}$$

Cependant, ce n'est pas vrai dans le cas général de P_{xy} où l'utilisateur doit formuler une décomposition qui respecte cette propriété. Sans cette formulation, le problème maître peut toujours ignorer l'objectif portant sur les variables y et s'écrire avec une variable objectif z initialement non contrainte exactement comme la formulation générique MP (cette formulation correspond à la définition du problème maître section 4.2.2).

Il y a de nombreux cas néanmoins, où la fonction objectif originelle réduite aux variables y est une relaxation. Ainsi, dans le cas d'un problème de coloration de graphes où le nombre de couleurs doit être minimisé, on obtient directement une relaxation en restreignant la fonction objectif aux variables y.

$$\begin{array}{ll} P_1: & \text{Min } NbDifferentValues(x,y) \\ & \text{s.t} : x_i \neq x_j\ \forall (i,j) \in G \end{array}$$

On a de manière évidente $NbDifferentValues(y) \leq NbDifferentValues(x, y)$ et $NbDifferentValues(y)$ est une fonction objectif valide pour le problème maître. Nous considérerons également l'exemple suivant qui sera développé plus tard et dont l'objectif s'exprime par une combinaison linéaire :

Exemple 14 *Problème exemple pour la décomposition*

$$
\begin{array}{lll}
P_2: & \text{Min} & y_1 + y_2 + 4x_1 + 2x_2 + 5x_3 + x_4 \\
& \text{s.t}: & |y_1 - y_2| < 2 \\
& & 5x_1 + x_2 + 2y_1 + y_2 \geq 5 \\
& & 3x_3 + 2x_4 + y_2 \geq 4 \\
& & x_1 \neq y_2 \\
& & occurrence(0, \{x_3, x_4, y_2\}) = 1 \\
& & occurrence(0, \{x_3, x_4, y_1\}) = 1 \\
& & y_1, y_2, x_1, x_2, x_3, x_4 \in [0, 5]
\end{array}
$$

Nous rappelons que $occurrence(i, L) = k$ impose que la valeur i apparaisse k fois dans la liste de variables L.

Dans ce cas de nouveau, l'utilisation de Min y_1+y_2 fournit un problème maître correct.

L'hypothèse critique de l'approche par décomposition concerne les coupes.

Définition 8.1.2 *Une coupe cut_i obtenue après l'itération i est dite valide si :*

1. *Elle n'exclut aucune solution faisable du problème originel (par rapport à la borne supérieure courante obj^*).*
2. *Elle exclut au moins la solution $\overline{y}$ obtenue à l'itération i du maître et prouvée sous-optimale ou inconsistante.*

Le point (1) assure l'optimalité de la solution finale en garantissant que le problème maître reste toujours une borne inférieure du problème complet tandis que (2) garantit la terminaison de l'algorithme, les domaines de y étant finis.

8.2 Cadre algorithmique

Nous présentons ici en détail les algorithmes pour montrer concrètement comment un cadre de décomposition générique peut être mis en place. Nous distinguons le cas des problèmes dont l'objectif ne porte que sur les variables du maître et celui où l'objectif porte sur toutes les variables du problème.

8.2.1 Cas de P_0 et P_y

P_0 et P_y sont fortement corrélés car ils s'appuient tous les deux sur des problèmes de satisfaction en sous-problèmes. L'algorithme de décomposition prend en entrée une solution initiale du problème maître $\overline{y}$:

Terminaison : Les coupes ajoutées ligne 11 excluent au moins la solution courante $\overline{y}$ de MP (propriété 8.1.2). Les domaines étant finis, on ne peut énumérer qu'un nombre

procédure : Benders solve($\overline{y}$)

1: **repeat**
2: $Cut = \emptyset$;
3: **while** continueCutGeneration(Cut) et $k \leq K$ **do**
4: résoudre SP^k
5: **if** SP^k est infaisable **then**
6: $Cut \leftarrow Cut \cup expl_k$
7: **else**
8: $P_0^{'}, P_y^{'} : SP^{k+1} = \bigcup_{i \leq k+1, i > k'} SP^i$ avec $k' = argmax_k(SP^k$ infaisable).
9: **end while**
10: **if** $Cut \neq \emptyset$ **then**
11: $\forall c \in Cut$ on ajoute c à MP
12: résoudre MP à l'optimum
13: **end if**
14: **until** MP est infaisable $\bigvee Cut = \emptyset$
15: la solution $(\overline{y}, \overline{x})$ est optimale si $Cut = \emptyset$. Sinon P est infaisable .

Algorithme 18: Un cadre de décomposition générique pour P_0, P_y, $P_0^{'}$ et $P_y^{'}$

fini de solutions pour le maître. Un problème maître infaisable dénote un problème originel infaisable. La terminaison se produit aussi à l'obtention d'une solution pour les SP^k ce qui se traduit par un ensemble Cut vide à la ligne 14.

Optimalité : Dans le cas de P_y, MP_y constitue toujours une borne inférieure (propriété 8.1.1), la première solution de MP_y qui s'étend à des solutions faisables sur les SP^k est donc la solution optimale de P.

La résolution pour P_0 et P_y s'apparente à une stratégie d'enregistrement de *nogoods* guidée par des structures définies au niveau de la phase de modélisation par l'utilisateur. Un utilisateur peut également contrôler le processus de génération de *nogoods* par le biais de la méthode *continueCutGeneration*(Cut). Cette méthode peut ainsi renvoyer faux dès qu'une coupe au moins a été obtenue. Il est critique de trouver un bon compromis entre le temps passé à la résolution des sous-problèmes et la précision des coupes obtenues à l'issue de la résolution des sous-problèmes.

CBJ peut être utilisé au niveau des sous-problèmes pour calculer des explications et fournir les coupes. Par ailleurs, l'utilisation de CBJ est possible également pour le problème maître et permet de limiter le *thrashing* à l'ajout dynamique de coupes. C'est une réponse possible à Thorsteinsson [Tho01] sur sa préoccupation de limiter les calculs redondants sur le maître.

Structures approximatives : Certaines précautions doivent être prises dans le cas de $P_0^{'}$ et $P_y^{'}$ (ligne 8). À chaque itération, une fois que le problème maître et les k sous-problèmes ont été résolus, le sous-problème $k+1$ est défini selon la règle suivante : soit $k' < k$ l'index du dernier sous-problème infaisable $SP^{k'}$, SP^{k+1} est défini par l'union de tous les sous-problèmes intermédiaires trouvés consistants depuis $k^{'}$: $SP^{k+1} = \bigcup_{i \leq k, i > k'} SP^i$. Les sous-problèmes n'étant pas parfaitement indépendants, la solution d'un sous-problème consistant peut être incompatible avec le sous-problème suivant. Le sous-problème suivant continue donc la résolution en incluant dans son es-

pace de recherche les variables des problèmes consistants précédents. Une telle stratégie n'implique pas de surcoût par rapport à la résolution du sous-problème complet en une seule fois, elle oriente simplement l'ordre des variables en espérant obtenir des coupes différentes à cause de la relative indépendance des sous-problèmes. Il est de nouveau évident qu'un compromis entre le temps passé à la résolution des sous-problèmes et la qualité des explications obtenues est critique aux performances de la résolution. Par exemple, dès qu'une coupe est obtenue, il semble raisonnable de limiter l'obtention d'autres coupes aux sous-problèmes en eux-mêmes.

8.2.2 Cas de P_{xy}

L'algorithme de décomposition 19 dans le cas de P_{xy} diffère davantage de l'enregistrement de *nogoods* classique. Il prend aussi en entrée une solution $\overline{y}$ du problème maître :

procédure : Benders solve($\overline{y}$)

1: **repeat**
2: $Cut = \emptyset$;
3: **while** continueCutGeneration(Cut) et $k \leq K$ **do**
4: ajoute $sz_k < ub_k = computeUb(k, \{z^*, sz_1^*, \ldots, sz_{k-1}^*\})$ à SP^k,
5: résoudre SP^k à l'optimum
6: $Cut \leftarrow Cut \cup expl_k$
7: **end while**
8: stocker $(\overline{y}, \overline{x})$ si c'est une solution améliorante
9: $Cut = computeCut(Cut, \{z^*, sz_1^*, \ldots, sz_n^*\}, \{ub_1, \ldots, ub_n\})$
10: $\overline{obj} \leftarrow computeUb(0, \{z^*, sz_1^*, \ldots, sz_n^*\})$
11: on ajoute $z < \overline{obj}$ à MP
12: $\forall c \in Cut$ on ajoute c à MP
13: résoudre MP à l'optimum
14: **until** MP est infaisable
15: la solution $(\overline{y}, \overline{x})$ est l'optimum de P, autrement P est infaisable.

Algorithme 19: Un cadre de décomposition générique pour pour P_{xy}

Terminaison : À toute itération de l'algorithme, on dispose ligne 10 d'une borne supérieure ($\overline{obj}$) et d'un ensemble d'explications (Cut) qui vont éliminer de l'espace de recherche de MP, une classe de solutions dont la valeur objectif est supérieure à cette borne. Après la ligne 13, on dispose d'une nouvelle borne inférieure du problème puisque MP est une relaxation (propriété 8.1.1). L'algorithme termine lorsque ces deux bornes se rencontrent, c'est-à-dire au moment où le problème maître devient infaisable pour P_{xy}. Comme précédemment, les domaines étant finis et les coupes respectant la propriété 8.1.2, cette rencontre se produit en un nombre fini d'étapes.

Optimalité : L'optimalité est assurée par le fait que les coupes n'éliminent pas de solutions de MP de coûts meilleurs que la borne supérieure courante (propriété 8.1.2) et que MP est résolu à l'optimum. La résolution à l'optimum de MP est critique pour que l'explication justifie bien l'absence de meilleure solution que $\overline{obj}$ pour **le problème complet** P. Toute variation de MP laissant inchangée l'explication conserve au moins sz^* et engendre aussi un coût supérieur pour MP puisqu'il était résolu à l'optimum.

Autrement dit, on ne peut pas faire décroître MP en jouant sur les variables n'apparaissant pas dans l'explication.

Pour conserver des sous-problèmes isolés, la fonction objectif n'est pas ajoutée comme une contrainte qui pourrait propager entre les sous-problèmes. À la place, la méthode *computeUb* fournit un moyen de passer les bornes obtenues d'un sous-problème à l'autre et de prendre ainsi (éventuellement) en compte les autres bornes connues (bornes du maître et des sous-problèmes). La propagation est donc effectuée *à la main* de manière à seulement incriminer la solution du maître pour chaque sous-problème. L'intérêt de cette implémentation est d'autoriser la résolution séparée de chaque sous-problème pour pouvoir combiner leurs valeurs optimales à loisir de manière à obtenir plusieurs coupes pour le maître. Les gestions des coupes et des valeurs objectifs sont intimement liées et les deux méthodes suivantes doivent être définies pour chaque type de fonction objectif et permettre de calculer des coupes **valides** :

- *computeUb(int k, double[] sopt)* (lignes 4-10) : calcule une borne supérieure de sz_k du sous-problème k en s'appuyant sur la liste, *sopt*, des différentes valeurs optimales connues pour chaque sz_i (avec $i < k$). Le fait d'imposer une borne à sz_k peut accélérer sa résolution mais si il s'avère infaisable sous cette borne, on perd sa valeur optimale exacte qui permettrait peut-être de générer plusieurs explications. Dans le cas de $k = 0$ (le problème maître) la méthode renvoie la valeur de la solution de P obtenue en combinant au sens de la fonction objectif toutes les bornes connues ;

- *computeCut(Explanation[] expls, double[] sopt, double[] ubs)* (ligne 9) : calcule la/les explications à ajouter à MP en s'appuyant sur les différentes valeurs optimales connues pour les SP^k et MP à l'itération précédente (tableau *sopt*) ainsi que sur les bornes supérieures utilisées pour leur résolution (tableau *ubs*). ubs_k permet de remplacer un $sopt_k$ en cas d'infaisabilité d'un SP^k pour disposer d'une valeur correcte permettant de calculer l'explication.

Exemple 15 *Implémentation de computeCut*

Considérons un problème de type de P_2 (exemple page 96) (la fonction objectif est une combinaison linéaire) et supposons qu'aucun SP^k ne s'est révélé infaisable (aucune des bornes supérieures *ubs* n'a été contrainte au moment de la résolution des sous-problèmes par *computeUb*). On peut implémenter, dans ce cas, *computeCut* comme une union de tout sous-ensemble de *expls* dont la somme des bornes dans *sopt* combinée avec celle du maître excède la borne supérieure courante.

La méthode *continueCutGeneration* joue le même rôle que précédemment et permet, conjointement à *computeUb* de spécifier des stratégies de résolution des sous-problèmes. Ainsi sur l'exemple de P_1, on peut s'interrompre au premier sous-problème dont le nombre de couleurs est incompatible avec la borne supérieure ou continuer à résoudre le sous-problème suivant en espérant obtenir des explications disjointes et peut-être plus précises. La clef est, comme dans le cas de P_0 et P_y, de pouvoir tirer parti

de la structure du sous-problème et générer des coupes qui vont éviter un maximum de *thrashing* au niveau du problème maître.

Exemple 16 *Exécution de l'algorithme 19*

L'algorithme 19 est ici déroulé à titre d'exemple sur le problème P_2 de l'exemple 14 (page 96). On peut noter qu'une fois y_1 et y_2 instanciées, les deux sous-problèmes suivants apparaissent :

SP1 : Min	$sz_1 = 5x_3 + x_4$		SP2 : Min	$sz_2 = 4x_1 + 2x_2$
	$3x_3 + 2x_4 \geq 4 - \overline{y_2}$			$5x_1 + x_2 \geq 5 - 2\overline{y_1} - \overline{y_2}$
	$occurrence(0, \{x_3, x_4, \overline{y_2}\}) = 1$			$x_1 \neq \overline{y_2}$
	$occurrence(0, \{x_3, x_4, \overline{y_1}\}) = 1$			

Itération 1	$(\overline{y_1}, \overline{y_2}) = (0, 0)$ $sz_1^* = 6$ $sz_2^* = 4$ $z = 10$	 $expl(sz_1 \geq 6) = \{y_2 = 0\}$ $expl(sz_2 \geq 4) = \{y_2 = 0\}$ $y_1 + y_2 < 10,\ y_2 \neq 0$ ajoutée
Itération 2	$(\overline{y_1}, \overline{y_2}) = (1, 1)$ $sz_1^* = 2$ $sz_2^* = 4$ $z = 8$	 $expl(sz_1 \geq 2) = \{y_2 = 1\}$ $expl(sz_2 \geq 4) = \{y_1 = 1, y_2 = 1\}$ $y_1 + y_2 < 8, y_1 \neq 1 \vee y_2 \neq 1$ ajoutées
Itération 3	$(\overline{y_1}, \overline{y_2}) = (2, 1)$ $sz_1^* = 2$ $sz_2^* = 0$ $z = 5$	 $expl(sz_1 \geq 2) = \{y_2 = 1\}$ $expl(sz_2 \geq 0) = \{y_1 = 2, y_2 = 1\}$ $y_1 + y_2 < 5, y_1 \neq 2 \vee y_2 \neq 1$ ajoutées
Itération 4	$(\overline{y_1}, \overline{y_2}) = (1, 2)$ $sz_1^* = 1$ SP2 infeas $z = 5$	 $expl(sz_1 \geq 2) = \{y_2 = 2\}$ $expl(sz_2 \geq 0) = \{y_1 = 1, y_2 = 2\}$ $y_1 \neq 1 \vee y_2 \neq 2$ ajoutées
Itération 5	$(\overline{y_1}, \overline{y_2}) = (2, 2)$ $sz_1^* = 1$	 $expl(sz_2 \geq 2) = \{y_2 = 2\}$ $y_2 \neq 2$ ajoutée

À l'itération 1, on peut noter que deux explications existent pour $sz_1 \geq 6$: $\{y_2 = 0\}$ ou $\{y_1 = 0\}$. Chacune est suffisante pour expliquer que $sz_1 \geq 6$. L'explication obtenue dépend de l'ordre de propagation des deux contraintes d'occurrence. Imaginons que $occurrence(0, \{x_3, x_4, y_2\})$ a été propagée en premier, retirant la borne inférieure 0 de x_3 et x_4. Après la résolution de chaque sous-problème, nous savons que z ne peut pas valoir moins que 10 tant que y_2 est nulle. À l'itération 5, la borne supérieure calculée avec $computeUb(2)$ est négative et l'explication pour $SP2$ est vide. Après cette itération, le problème maître devient donc inconsistant et l'algorithme termine avec la solution optimale $(y_1, y_2, x_3, x_4, x_1, x_2) = (2, 1, 0, 2, 0, 0)$.

On peut noter, que dans le cas général, la borne supérieure ne décroît pas nécessairement à chaque itération comme c'est le cas habituellement en CP. Au contraire, c'est ici la borne inférieure qui croît à chaque itération jusqu'à sa rencontre avec la meilleure borne supérieure connue.

8.2.3 API

Nous avons cherché à concevoir et à implémenter un cadre générique de décomposition dans un solveur expliqué tel que PaLM ([JB00]). L'utilisateur peut donc définir une structure maître/esclaves sur son problème au niveau de la phase de modélisation. De cette manière, une stratégie de décomposition ou d'enregistrement de *nogoods* dédiée à ce type de structure se met automatiquement en place. Voici ainsi, l'API supplémentaire disponible au niveau du solveur :

```
Problem pb = new Problem();
IntVar x = pb.makeEnumIntVar(1, 6);
pb.addMasterVariables(x);
...
IntVar w = pb.makeEnumIntVar(1, 6);
// variable w appartient au troisième sous-problème
pb.addSubVariables(3, w);
...
// la structure est-elle idéale ou approximative
pb.setApproximatedStructure();
...
//problème de satisfaction
pb.solve();
// fonction objectif uniquement sur le maître
pb.minimize(z);
// objectif: z + z1 + z2
pb.minimize(z, new IntVar[]{z1,z2}, new SumRelation());
```

8.3 Conclusion

La stratégie générique de décomposition présentée ici est de notre point de vue, un argument très fort pour le maintien d'explications par les solveurs de contraintes. Les applications de décomposition en ordonnancement sont nombreuses mais plus généralement, les problèmes industriels impliquent souvent différents niveaux de décision (affecter des tâches sur des machines puis ordonnancer, affecter des produits dans des camions puis router, etc.). Les travaux qui s'appuient sur un solveur de contraintes ou SAT en sous-problème sont donc nombreux [MG04, BG04, CX05, BBGM06] et des schémas génériques existent déjà [JG01, HOTK00]. Or les algorithmes de filtrage sont des boites noires qui rendent impossible l'extraction par un utilisateur de cette information duale critique pour ces techniques de décomposition. Hooker n'est pas le seul à souligner ce manque et Grossmann revient sur cet état de fait dans [MG04] :

> *Good integer cuts include only the binary variables that are* ***responsible*** *for the infeasibility of a solution, and it is usually very difficult to generate, mainly because the* ***source*** *of infeasibility is usually not revealed when the CP subproblem is found infeasible.*

L'implémentation de ce cadre est encore à l'état de prototype et n'a été testée que sur des problèmes aléatoires en satisfaction et des problèmes jouets en optimisation. Des expérimentations de ce cadre générique sont donc encore à réaliser. Nous avons préféré mettre en œuvre cette décomposition logique sur une application de placement de tâches temps-réel au chapitre 9.

Ces idées ont été publiées en papier court dans [CJ05]

Troisième partie

Applications

Chapitre 9

Placement de tâches temps réel

Nous mettons ici en œuvre les idées présentées au chapitre 8 sur la décomposition logique basée sur le calcul des explications ainsi que les idées du chapitre 6 sur les heuristiques génériques de recherche. Le problème traité provient d'une collaboration avec l'équipe Systèmes Temps Réel de l'IRCCyN (Institut de Recherche en Communications et Cybernétique de Nantes) et particulièrement avec Pierre-Emmanuel Hladik et Anne-Marie Déplanche. Il s'agit d'un problème courant qui se pose au cours de la conception de systèmes embarqués et dont les applications se rencontrent dans de nombreux domaines et particulièrement dans le milieu automobile. Nous présentons ici le problème en détail avant d'aborder sa résolution avec une technique de décomposition de Benders et une approche classique passant par la conception d'une contrainte globale.

9.1 Description du problème

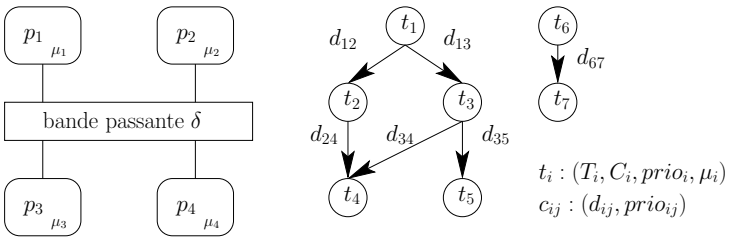
Les problématiques temps réel interviennent au cœur des systèmes embarqués dans de nombreux domaines : automobile, robotique, aéronautique, télécommunications, ... Les applications actuelles mettent en œuvre un ensemble de calculateurs spécialisés (contrôle moteur, ABS, contrôle de chassis, climatisation ...) qui reçoivent des données de capteurs, les traitent et calculent des commandes adéquates, qu'ils envoient vers des actionneurs. Leur caractéristique centrale réside dans le respect d'exigences fonctionnelles mais aussi extra-fonctionnelles telles que les contraintes temporelles. Un système temps réel *dur* est un système pour lequel le non-respect d'une échéance peut causer des défaillances graves et mettre en danger tout le système. Nous nous intéressons aux systèmes temps réel durs distribués (les processeurs sont interconnectés par un réseau de terrain) pour lesquels les unités de traitement ou processeurs respectent une politique d'ordonnancement préemptif à priorités fixes. Les applications considérées sont implémentées par des tâches périodiques, communiquantes et dont les temps de réponse sont contraints. [Hla04] donne une présentation générale de tels systèmes.

9.1.1 L'architecture du système temps réel

Le système temps réel est ici modélisé sous la forme d'une architecture logicielle (l'ensemble des tâches applicatives) et d'une architecture matérielle (le support physique d'exécution des tâches). Ce modèle, synthétisé figure 9.1, est celui utilisé par Tindell [TBW92].

9.1.1.1 L'architecture matérielle

L'architecture matérielle est une paire $(\mathcal{P}, \mathcal{N})$ constituée d'un ensemble $\mathcal{P} = \{p_1, \ldots, p_k, \ldots, p_m\}$ de m processeurs de capacité mémoire μ_k s'échangeant des informations sur un réseau $\mathcal{N}$. Les



(a) Architecture matérielle (b) Architecture logicielle

FIG. 9.1: Architecture matérielle et logicielle d'un système temps-réel

processeurs sont considérés identiques du point de vue du traitement (le temps d'exécution d'une tâche est le même quel que soit le processeur).

Les processeurs communiquent via un réseau caractérisé par son débit δ (la quantité d'information transmissible par unité de temps) et son protocole. Deux protocoles utilisés par les réseaux de terrain (du fait de leur caractère déterministe) ont été pris en compte dans notre approche :

- *Controller Area Network* (CAN) [Bos91] : il s'agit à la fois d'un protocole et d'un réseau physique. CAN fonctionne comme un réseau *broadcast* dans lequel tous les messages sont envoyés à tous les processeurs. Les messages sont mis en attente au niveau de chaque processeur et le protocole assure que le message de plus haute priorité est transmis en premier. En revanche, une fois que la transmission a commencé, le message ne peut plus être préempté. Par ailleurs, nous considérerons ici un mode de communication désynchronisé (l'activation de la tâche réceptrice est indépendante de l'arrivée du message). CAN est un protocole initialement conçu par l'industrie automobile et possède aujourd'hui un très large spectre d'applications dans d'autres domaines temps-réel ;
- L'anneau à jeton (*token ring*) : un processeur peut transmettre des données sur le réseau s'il est en possession du jeton, et cela seulement pendant une durée donnée. Après cette durée, ou si le processeur n'a plus de données à envoyer, le jeton est passé au processeur suivant de l'anneau. La durée maximale de présence du jeton sur un processeur doit garantir un temps suffisant pour envoyer tous les messages en attente sur les processeurs.

9.1.1.2 L'architecture logicielle

Le modèle que nous employons pour décrire **l'architecture logicielle** se compose d'un graphe $(\mathcal{T}, \mathcal{C})$ orienté, acyclique et valué, pour lequel l'ensemble des nœuds $\mathcal{T} = \{t_1, ..., t_n\}$ désigne les **tâches** (ou **processus**) et l'ensemble des arcs, $\mathcal{C} \subseteq \mathcal{T} \times \mathcal{T}$, représente les communications entre tâches. Un arc $c_{ij} = (t_i, t_j) \in \mathcal{C}$ modélise l'envoi d'un message de t_i vers t_j. Une tâche t_i est définie par : sa période, T_i, son temps maximum d'exécution sans préemption, C_i et son besoin en mémoire m_i[1]. Deux tâches qui communiquent ont la même période. Les arcs $c_{ij} = (t_i, t_j) \in \mathcal{C}$ sont caractérisés par la quantité d_{ij} d'information échangée. Les tâches peuvent communiquer en utilisant le réseau et son protocole ou la mémoire locale du processeur. Dans ce dernier cas, la communication s'effectue sans délai, mais les tâches doivent être situées sur le même processeur. Les tâches sont activées périodiquement de façon indépendante. Elles lisent et écrivent des données au début et à la fin de leur exécution. Un message c_{ij} est également périodique et hérite de la période de sa tâche émettrice et réceptrice de telle sorte qu'on a toujours $T_{ij} = T_i = T_j$. Enfin, plusieurs grandes catégories de politiques d'ordonnancement existent. Nous considérerons que tous les processeurs sont munis d'un ordonnanceur

[1] Comme pour les temps d'exécution, nous ferons l'hypothèse que le besoin mémoire est indépendant du processeur.

préemptif à priorités fixes. Un niveau de priorité unique $prio_i$ est affecté à chaque tâche et une tâche t_i est dite plus prioritaire que t_j si et seulement si $prio_i > prio_j$. L'exécution d'une instance de tâche doit être préemptée (suspendue) jusqu'à ce que toutes les instances de tâches plus prioritaires prêtes, terminent leur exécution. Le tableau 9.1 synthétise ces notations.

Architecture matérielle	
Processeurs	
$\mathcal{P}$	Ensemble des processeurs, $\|\mathcal{P}\| = m$
p_i	Processeur i
μ_i	Capacité mémoire du processeur p_i
Réseau	
δ	Débit du réseau
Architecture logicielle	
Tâches	
$\mathcal{T}$	Ensemble des tâches, $\|\mathcal{T}\| = n$
t_i	Tâche i
T_i	Période de t_i
C_i	Pire temps d'exécution sans préemption de t_i
R_i	Pire temps de réponse de t_i
m_i	Besoin mémoire de t_i
$prio_i$	Priorité de t_i
Messages	
$\mathcal{C}$	Ensemble des messages, $\|\mathcal{C}\| = c$
c_{ij}	Message de t_i vers t_j
d_{ij}	Taille de c_{ij}
C_{ij}	Pire temps de transmission sans préemption de c_{ij}
R_{ij}	Pire temps de réponse de c_{ij}
$prio_{ij}$	Priorité de c_{ij}

TAB. 9.1: Synthèse des notations

9.1.2 Le problème de placement/ordonnançabilité

Un placement ou une allocation est une application A qui, à une tâche t_i de $\mathcal{T}$, associe un processeur p_k de $\mathcal{P}$:

$$\begin{array}{rccc} A: & \mathcal{T} & \rightarrow & \mathcal{P} \\ & t_i & \mapsto & A(t_i) = p_k \end{array} \tag{9.1}$$

Le problème de placement consiste donc à trouver l'application A respectant l'ensemble des contraintes ci-dessous.

9.1.2.1 Contraintes de ressources

Trois types de ressources sont considérées : mémoire des processeurs, utilisation des processeurs et utilisation du réseau.

– **Capacité mémoire**[2] : la somme des besoins mémoire de l'ensemble des tâches résidant sur un processeur p_k ne doit pas excéder la capacité (μ_k) du dit processeur :

$$\forall k = 1..m, \sum_{A(t_i)=p_k} m_i \leqslant \mu_k \tag{9.2}$$

[2]Il serait possible, sans nuire à la généralité du problème traité, de différencier la mémoire en différents types (RAM, ROM, EEPROM ...), chaque type ayant une taille propre pour un processeur donné.

– **Facteur d'utilisation** : pour que toutes les tâches puissent s'exécuter, il est nécessaire que le taux d'utilisation des processeurs ne soit pas supérieur à leur capacité de traitement. Le ratio $r_i = C_i/T_i$ signifie qu'un processeur est utilisé r_i % du temps par la tâche t_i. Une première condition nécessaire d'ordonnançabilité s'écrit :

$$\forall k = 1..m, \sum_{A(t_i)=p_k} \frac{C_i}{T_i} \leqslant 1 \tag{9.3}$$

– **Utilisation du réseau** : pour que tous les messages puissent être acheminés sur le réseau, il est nécessaire que la somme des quantités de données transportées par le réseau par unité de temps soit inférieure au débit du réseau :

$$\sum_{\substack{c_{ij} = (t_i, t_j) \\ A(t_i) \neq A(t_j)}} \frac{d_{ij}}{T_i} \leqslant \delta \tag{9.4}$$

9.1.2.2 Contraintes de placement

Les contraintes dites de placement concernent l'affectation des tâches sur l'ensemble des processeurs. Trois types de contraintes interviennent sur le placement : résidence, co-résidence et exclusion.

– **Résidence** : cette contrainte est dûe à la nécessité pour une tâche d'utiliser des ressources logicielles ou matérielles particulières mises à disposition par certains processeurs (un capteur de vitesse couplé avec un dispositif matériel présent sur un calculateur donné). Il s'agit d'un couple (t_i, α) où $t_i \in \mathcal{T}$ est une tâche et $\alpha \subseteq \mathcal{P}$ est l'ensemble de processeurs sur lesquels la tâche peut s'exécuter. Un placement donné A doit alors vérifier :

$$A(t_i) \in \alpha \tag{9.5}$$

– **Co-résidence** : cette contrainte stipule que plusieurs tâches doivent être placées sur le même processeur (ces tâches nécessitent le partage d'une ressource commune). Une contrainte de co-résidence est définie par un ensemble de tâches $\beta \subseteq \mathcal{T}$ et tout placement A doit satisfaire :

$$\forall (t_i, t_j) \in \beta^2, A(t_i) = A(t_j) \tag{9.6}$$

– **Exclusion** : il s'agit d'interdire qu'un ensemble de tâches résident sur les mêmes processeurs. C'est en particulier le cas lorsque des tâches sont des répliques redondantes d'autres tâches à des fins de tolérance aux fautes. Il s'agit d'un ensemble $\gamma \subseteq \mathcal{T}$ de tâches et tout placement A doit satisfaire :

$$\forall (t_i, t_j) \in \gamma^2, A(t_i) \neq A(t_j) \tag{9.7}$$

9.1.2.3 Contraintes temporelles

Les contraintes temporelles s'expriment par des **échéances** portant sur la terminaison des tâches. Ces échéances sont relatives aux dates d'activation[3] des tâches et doivent assurer que la durée séparant la date d'activation de toute instance d'une tâche t_i et sa date de terminaison soit inférieure à sa période T_i. En *token ring* et en cas d'envoi de messages, la date de terminaison est remplacée par la date de fin de transmission des données émises.

Les approches courantes pour vérifier les contraintes temporelles s'appuient sur des conditions nécessaires et suffisantes d'ordonnançabilité. Ces conditions sont liées à la nature de l'architecture matérielle et logicielle et sont établies à travers le pire scénario d'exécution. Les principes de ces techniques analytiques sont donnés ici.

[3]La date d'activation est l'instant où la tâche se réveille. Une tâche peut ainsi être active mais en attente parcequ'elle est préemptée par une tâche plus prioritaire

Calcul du pire temps de réponse d'une tâche sans réseau : les premières analyses d'ordonnançabilité ont été proposées par Liu et Layland [LL73] pour des systèmes temps réel mono-processeur à priorités fixes avec des tâches indépendantes dont les échéances sont inférieures ou égales aux périodes. L'analyse consiste à calculer pour chaque tâche t_i son pire temps de réponse, R_i, en construisant un scénario d'exécution qui pénalise au maximum l'exécution de t_i. Dans le cas des systèmes de tâches indépendantes, il a été prouvé que le pire scénario pour une tâche t_i survient lorsque toutes les tâches plus prioritaires sur le même processeur se réveillent au même instant que t_i (la figure 9.2 dépeint un tel scénario et la non-ordonnançabilité d'une tâche).

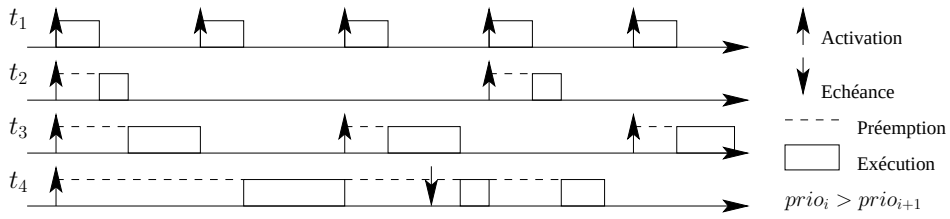


FIG. 9.2: Pire scénario d'exécution pour 4 tâches dans lequel t_4 ne respecte pas son échéance, étant préemptée par 3 tâches plus prioritaires t_1, t_2, t_3 puis à nouveau t_1.

Le pire temps de réponse de t_i est alors :

$$R_i = C_i + \sum_{t_j \in hp_i(A)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{9.8}$$

$hp_i(A)$ désigne ici l'ensemble des tâches plus prioritaires que t_i sur le processeur $A(t_i)$ pour un placement A donné. Le calcul de R_i s'obtient par la recherche du point fixe de l'équation[4]. La comparaison des pires temps de réponse avec les échéances suffit alors pour connaître l'ordonnançabilité du système et une condition nécessaire et suffisante s'écrit :

$$R_i \leq T_i \tag{9.9}$$

Calcul du pire temps de réponse d'une tâche avec réseau : Dans le cadre d'un réseau CAN en mode désynchronisé, les messages sont ordonnancés de manière indépendante. En revanche pour l'anneau à jeton, le plus grand délai possible d'émission des données sur le réseau, TRT (*Token Rotation Time*) doit être pris en compte. Cette durée est égale au temps nécessaire pour transmettre l'ensemble des données sur le réseau (au pire une donnée est produite lorsque le processeur sur lequel la tâche émettrice perd le jeton et doit attendre un tour complet) :

$$TRT = \sum_{\substack{\{c_{ij} = (t_i, t_j) | \\ A(t_i) \neq A(t_j)\}}} \frac{d_{ij}}{\delta} \tag{9.10}$$

La condition nécessaire s'écrit alors

$$R_i \leq T_i - TRT \tag{9.11}$$

Calcul du pire temps de réponse d'un message : Pour un réseau CAN en mode désynchronisé, l'ordonnancement des messages peut se voir comme un problème d'ordonnancement non préemptif à priorité fixe (les messages sont assimilés à des tâches). L'équation (9.8)

[4]Ce type d'équation sera abordé plus en détail section 9.2.2.1.

peut dès lors être appliquée aux messages avec certains raffinements. Ainsi, le calcul prend en compte le fait qu'un message ne puisse être préempté qu'au moment de la transmission de son premier bit et non pendant toute son exécution (d'où le $+\tau_{bit}$ du terme $L_{ij} + \tau_{bit}$, Eq. 9.13). Deuxièmement, le temps maximal pendant lequel un message est bloqué par un message de priorité inférieure doit être ajouté (terme $\max_{M_{kl} \in lp_{ij}(A)}(C_{kl} - \tau_{bit})$ de Eq. 9.13). Le pire temps de réponse d'un message CAN s'écrit [TC94] comme la somme de son pire temps de transmission C_{ij} et un temps de latence L_{ij} lié au protocole :

$$R_{ij} = C_{ij} + L_{ij} \tag{9.12}$$

avec

$$L_{ij} = \sum_{M_{kl} \in hp_{ij}(A)} \left\lceil \frac{L_{ij} + \tau_{bit}}{T_{kl}} \right\rceil C_{kl} + \max_{M_{kl} \in lp_{ij}(A)} (C_{kl} - \tau_{bit}) \tag{9.13}$$

où $hp_{ij}(A)$ (respectivement $lp_{ij}(A)$) est l'ensemble des messages de priorités plus élevées (respectivement plus faibles) que $prio_{ij}$; τ_{bit} désigne le temps de transmission d'un bit ($\tau_{bit} = \frac{1}{\delta}$). De nouveau, la résolution de Eq. (9.13) est faite itérativement par la recherche de son point fixe.

Le calcul du temps de transmission d'un message C_{ij} est fait d'après l'équation Eq. (9.14) [TC94] :

$$C_{ij} = s_{ij}\tau_{bit} \tag{9.14}$$

où

$$s_{ij} = \left\lfloor \frac{34 + 8d_{ij}}{5} \right\rfloor + 47 + 8d_{ij} \tag{9.15}$$

On peut remarquer que la taille du message prend aussi en compte des données propres au protocole CAN (identifiants, CRC, etc. requièrent par exemple 47 bits). La condition d'ordonnançabilité s'écrit à nouveau

$$R_{ij} \leq T_i \tag{9.16}$$

Les messages ont en effet des périodes identiques à celles des tâches émettrices et réceptrices et on a toujours $T_{ij} = T_i = T_j$. Il est crucial de noter ici que l'échange de messages en CAN n'affecte pas l'échéance des tâches comme c'est le cas pour l'anneau à jeton. La prise en compte de contraintes de précédence (mode de communication synchronisé entre les tâches) n'est pas abordée ici mais introduirait de manière analogue à l'anneau à jeton des dépendances entre tâches et messages.

Une allocation est dite **valide** si elle vérifie les contraintes de placement et de ressources. Elle est dite **ordonnançable** si elle vérifie les contraintes temporelles. Une allocation valide et ordonnançable est une solution du problème.

9.2 Résolution

La résolution du problème de placement a déjà fait l'objet de nombreuses études. Des approches heuristiques et incomplètes très variées à base d'algorithmes génétiques [MT93, San96, MBD98], de recherche taboue [PR93, VO05], de recuit simulé [TBW92, DS95] ou d'heuristiques dédiées [SFOC97, EW99] ont été appliquées au problème. Les approches complètes sont moins présentes mais existent néanmoins et font référence à l'algorithmique de graphe [Sto77] et à des recherches arborescentes de type *branch and bound* [ML95]. Enfin, il existe quelques travaux en programmation par contraintes pour des problématiques proches [SW00, SK03, SGK00, Eke04]. Néanmoins, ces travaux proposent des ordonnancements **hors ligne** des tâches (à l'exception de [RRC03] qui considère en plus l'affectation des priorités aux tâches mais ignore les contraintes de placement) et ne sont pas confrontés au problème des contraintes temporelles **temps-réel** en ordonnancement dynamique.

Ces contraintes temporelles sont des équations en nombres entiers qui font intervenir des parties entières et demandent une technique de résolution dédiée. Elles semblent pour cette raison difficiles à exprimer par des contraintes primitives disponibles dans les solveurs. On peut, en revanche tout à fait imaginer d'encapsuler ces techniques analytiques dans une contrainte globale d'ordonnançabilité. Cette approche est abordée section 9.2.2. Par ailleurs, les contraintes de ressources et de placement définissent un problème d'affectation qui relève typiquement de la programmation par contraintes ou de la programmation linéaire. Le calcul d'une affectation valide comme celui d'une affectation ordonnançable sont tous les deux des problèmes NP-Complets [TBW92] de nature assez différente. Les contraintes relèvent en effet de techniques de résolution très différentes et la décomposition du problème originel selon les deux axes semble une piste assez naturelle. Néanmoins, [TBW92] pense que la séparation des problèmes est impossible étant donné leur degré d'imbrication et propose de traiter globalement et simultanément les deux par un recuit simulé :

> *In general, the three activities of task allocation, processor scheduling and network scheduling are all NP-Hard problems. This lead to the view that they should be considered separately. Unfortunately, it is often not possible to obtain optimal solution (or even feasible solutions) if the three activities are treated in isolation. ... The tradeoffs can become very complex as the hard real-time architecture becomes more expressive ; a simple and scalable approach is required.*

Nous pensons, au contraire, que la coopération étroite entre les problèmes est possible à travers le calcul d'explications d'échecs exploitant les techniques de la communauté temps-réel.

9.2.1 Résolution : une approche fondée sur le calcul d'explications

Notre première approche est fondée sur la séparation entre l'allocation et l'ordonnançabilité et une coopération selon le principe de la décomposition de Benders abordée section 8. La figure 9.3 synthétise l'approche.

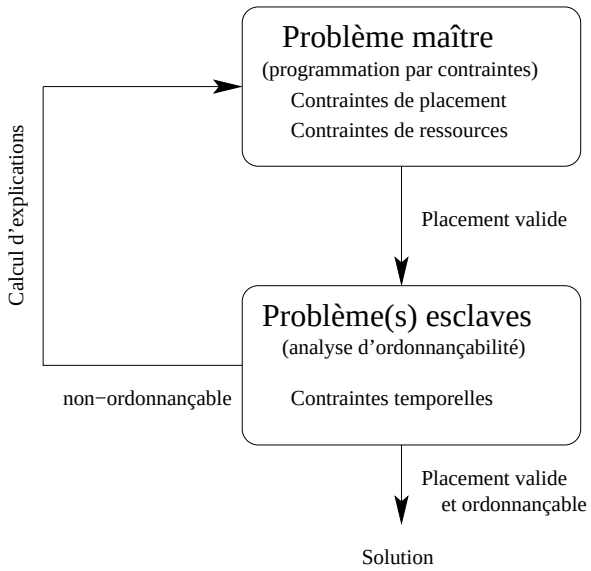


FIG. 9.3: Schéma de décomposition

Un problème maître résout le problème d'allocation en traitant uniquement les contraintes de ressources et de placement. Il fournit des placements **valides**. L'ordonnançabilité d'un tel

placement est ensuite établie par un ou des problèmes esclaves (sous-problèmes) à l'aide de techniques analytiques. En cas d'échec, le rôle du sous problème est de calculer une coupe de Benders, une explication. La coupe se traduit comme une contrainte ajoutée au problème maître qui interdit, non seulement le placement courant prouvé non-ordonnançable, *mais aussi tous les placements qui seraient non-ordonnançables pour les même raisons.* Les explications sont un mécanisme de généralisation des échecs et la convergence de l'algorithme est naturellement d'autant plus rapide que cette généralisation est efficace. Enfin, le problème maître étant une relaxation du problème originel, s'il ne possède pas de solutions, le problème complet non plus.

9.2.1.1 Problème maître

Le modèle utilisé pour le problème maître s'appuie sur trois types de variables, x_i, y_{ip} et w_{ij}. La variable x_i désigne le processeur affecté à la tâche i. y_{ip} et w_{ij} sont des variables booléennes qui indiquent respectivement si la tâche i est placée sur le processeur p et si les tâches i et j sont sur des processeurs différents ou non. Les variables x_i suffisent à définir une solution, ce sont les variables de décision du problème. On note $ppcm(X)$ le plus petit commun multiple de l'ensemble d'entiers X. Le modèle complet du problème maître est donné tableau 9.2.

Variables	
$x_i = [1..m]$,	$\forall 0 < i \leq n$
$y_{ip} = \{0,1\}$,	$\forall 0 < i \leq n, 0 < p \leq m$
$w_{ij} = \{0,1\}$,	$\forall 0 < i \leq n, 0 < j \leq n$
Contraintes	
Résidence	$\forall (i,\alpha),\ x_i \neq \alpha$
Co-Résidence :	$\forall (t_i,t_j) \in \beta^2, x_i = x_j$
Exclusion :	$alldifferent(x_i \vert t_i \in \gamma)$
Capacité mémoire :	$\forall p \in \{1..m\},\ \sum_{i\in\{1..n\}} y_{ip} \times m_i \leq \mu_p$
Facteur d'utilisation :	$\forall p \in \{1..m\},\quad \sum_{i\in\{1..n\}} \frac{ppcm(T)\times C_i \times y_{ip}}{T_i} \leq ppcm(T)$
Utilisation du réseau :	$\sum_{i\in\{1..n\}} \frac{ppcm(T)\times d_{ij}\times w_{ij}}{T_i} \leq ppcm(T) \times \delta$
Intégrité des modèles 1 :	$\forall i, x_i = j \Leftrightarrow y_{ij} = 1$
Intégrité des modèles 2 :	$\forall c_{ij}, x_i = x_j \Leftrightarrow w_{ij} = 1$

TAB. 9.2: Modèle du problème maître

Les contraintes d'intégrité du modèle garantissent la cohérence des modèles redondants définis sur x, y et w et sont implémentées par des contraintes dites de *channeling* [CLW96]. Les contraintes de capacité mémoire et facteur d'utilisation peuvent se modéliser plus efficacement par des contraintes *cumulative* [BC02] ou encore de *bin packing* [Sha04]. La contrainte *cumulative* est définie de la manière suivante :

$$cumulative(\{O_1,\ldots,O_n\},\{D_1,\ldots,D_n\},\{R_1,\ldots,R_n\},C)$$

O_i, D_i, R_i désignent respectivement l'origine, la durée et la consommation d'une tâche i et C est la capacité de la ressource. La *cumulative* assure que pour tout instant i :

$$\sum_{k|O_k\leq i\leq O_k+D_k} R_k \leq C$$

Nous considérons chaque tâche x_i comme la variable de date de début d'une tâche de durée unitaire et de hauteur m_i. Chaque processeur est ainsi considéré comme un instant d'exécution

pour cette tâche et le profil cumulatif est défini par les capacités mémoires μ_i de chaque processeur. C est donc une fonction discontinue $\mathcal{P} \rightarrow N$, telle que $\forall i \in \mathcal{P}, C(i) = \mu_i$.

$$\forall p \in \{1..m\}, \quad cumulative(\{x_1, \ldots, x_n\}, \{1, \ldots, 1\}, \{m_1, \ldots, m_n\}, C(i))$$
$$\forall p \in \{1..m\}, \quad cumulative(\{x_1, \ldots, x_n\}, \{1, \ldots, 1\}, \{\frac{ppcm(T) \times C_1}{T_1}, \ldots, \frac{ppcm(T) \times C_n}{T_n}\}, ppcm(T))$$

Un tel modèle permettrait d'éviter l'introduction des variables y_{ip}. Enfin, le problème maître est résolu en *backjumping* pour intégrer efficacement les contraintes fournies par le problème esclave et éviter des calculs redondants liés à l'ajout dynamique de contraintes.

9.2.1.2 Problèmes esclaves : expliquer la non-ordonnançabilité

L'ordonnançabilité d'un placement est établie sur celle de chaque tâche t_i à travers le calcul des R_i et celle de chaque message c_{ij} à travers R_{ij}. L'examen des équations 9.8 et 9.10 montre que R_i uniquement repose sur :

- les tâches plus prioritaires situées sur le même processeur ;
- le temps de latence dû à des messages transitant sur le réseau soit des paires de tâches situés sur des processeurs différents.

Les équations 9.12 et 9.13 montrent que R_{ij} ne dépend quant à lui que des messages plus prioritaires et du message moins prioritaire le plus long. Pour un placement donné, l'explication d'échec est donc définie de la manière suivante :

Définition 9.2.1 *Une explication d'échec pour un placement A est une paire $(T \subseteq \mathcal{T}, M \subseteq \mathcal{C})$ telle que :*

- $\forall t_i, t_j \in T, A(t_i) = A(t_j)$;
- $\forall c_{ij} \in M, A(t_i) \neq A(t_j)$;
- *il existe au moins une tâche de T ou un message de M non ordonnançable.*

Une explication e est notée $e = (T, M)$ et on fera référence aux ensembles T et M de e par e_T et e_M. Cette définition souligne simplement que l'explication repose sur le fait que des tâches ont été placées ensemble à tort sur un même processeur et d'autres au contraire ont été séparées à tort (impliquant l'envoi de messages qui auraient autrement transité par la mémoire interne du processeur). En particulier, les processeurs mis en jeu ne sont pas incriminés dans la mesure où ils sont tous identiques. Cette symétrie est donc exploitée à ce niveau.

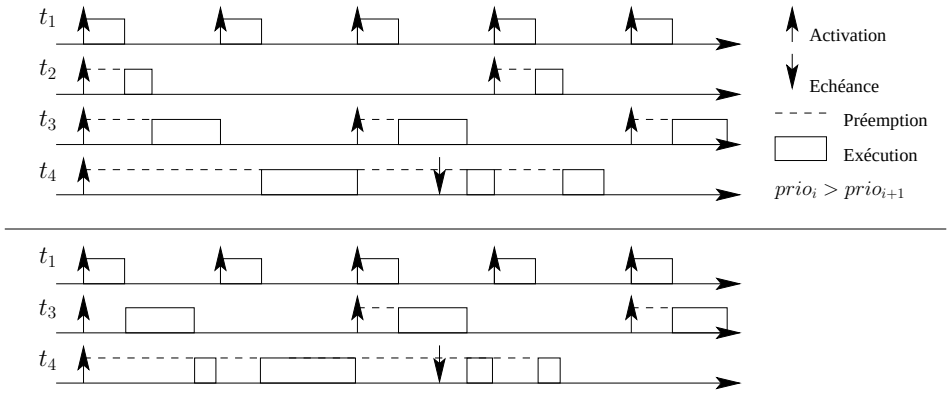


FIG. 9.4: L'ensemble $\{t_1, t_3\}$ est suffisant pour faire rater son échéance à t_4

Tâches indépendantes : En l'absence de message, le problème esclave se décompose en m problèmes d'ordonnançabilité indépendants sur chaque processeur. En cas d'échec sur un processeur p, il existe une tâche i telle que $x_i = p$ avec $R_i > T_i$. L'équation 9.8 nous indique déjà que seules les tâches plus prioritaires que i sont à l'origine de $R_i > T_i$. On peut néanmoins, obtenir une explication souvent plus précise. Reprenons l'exemple 9.2 dans lequel t_4 est non ordonnançable à cause des tâches plus prioritaires $\{t_1, t_2, t_3\}$. On se rend compte que t_2 agit peu sur l'ordonnançabilité étant donnés son temps d'exécution très court et sa période très longue. La présence de $\{t_1, t_3\}$ est en fin de compte suffisante pour entraîner la non-ordonnançabilité de t_4 (la figure 9.4 illustre cet exemple).

On peut donc dériver les ensembles minimaux (au sens de l'inclusion) de tâches conflictuelles en appliquant l'algorithme XPLAIN (voir section 3.1.2), dans lequel la méthode d'inférence employée est l'équation 9.9 (donc implicitement 9.8) et les tâches jouent alors le rôle des contraintes ajoutées une à une sur le processeur. L'explication prend donc la forme $(T, \emptyset)$ où

$$T = \text{XPLAIN}(\{t_i, x_i = p\}, Eq.9.9)$$

Elle se traduit par une contrainte de type *NotAllEqual*[5] sur les variables x :

$$NotAllEqual(x_i | t_i \in T)$$

Présence du réseau : Le même type de réduction peut être effectuée en CAN pour les messages par l'appel à $M = \text{XPLAIN}(\{c_{ij}, x_i \neq x_j\}, Eq.9.16)$ pour obtenir une explication d'échec de la forme $(\emptyset, M)$ et se traduisant par une contrainte du type :

$$\sum_{c_{ij} \in M} w_{ij} < |M|$$

Dans le cadre du réseau à jeton, le réseau est toujours ordonnançable du fait du protocole mais il influe sur l'ordonnançabilité des tâches et l'explication est cette fois de la forme (T, M) avec $(T, M) = \text{XPLAIN}(\{t_i, x_i = p\} \cup \{c_{ij}, x_i \neq x_j\}, Eq.9.11)$. La contrainte correspondante s'écrit :

$$NotAllEqual(x_i | t_i \in T) \bigvee \sum_{c_{ij} \in M} w_{ij} < |M|$$

Les trois contraintes précédentes peuvent être ajoutées au problème maître à chaque itération. Dans le cas de dépendance entre tâches et réseau, l'ordonnançabilité des tâches seules et du réseau seul est d'abord testée avant d'incriminer l'interaction des deux.

9.2.1.3 Reformulation

La reconnaissance de *pattern* au sein d'un ensemble de contraintes exprimant des problématiques spécifiques est un élément critique de la phase de modélisation. Réaliser cette reconnaissance de *pattern* de façon automatique entre dans le cadre de l'apprentissage de contraintes [Col06]. On aimerait ici exploiter une idée analogue en cherchant à extraire au sein d'un ensemble de contraintes élémentaires, des *patterns* correspondant à des contraintes globales. Par exemple, l'extraction de *alldifferent* parmi un ensemble de différences est une problématique bien connue et résolue par la recherche d'une clique de taille maximum sur le graphe des contraintes de différences. Nous avons implémenté à cette fin une version de l'algorithme de Bron et Kerbosh [BK73] (des différences interviennent en cas de $NotAllEqual$ de taille 2). De manière similaire, un ensemble de $NotAllEqual$ peut être équivalent à une contrainte globale de cardinalité : *global cardinality constraint* (gcc) [Rég96]. L'obtention de

[5] *NotAllEqual* porte sur un ensemble V de variables et assure que deux variables au moins parmi V prennent des valeurs distinctes.

gcc pose de nombreuses difficultés car elle demande l'extraction de cliques maximales dans un hyper-graphe.

L'utilisation de l'automate pour compiler les $NotAllEqual$ s'est heurtée aux mêmes problèmes que ceux décrits au chapitre 7 pour les *nogoods* généralisés. Nous n'avons pas encore obtenu de réponse satisfaisante à ce problème mais il constitue une piste clef pour améliorer les résultats, notamment dans des cas d'inconsistance du point de vue de l'ordonnançabilité.

9.2.1.4 Stratégies de recherche

Le problème de placement mêle différentes problématiques qui peuvent prendre le pas l'une sur l'autre selon la typologie des instances. Les contraintes de mémoire sont des contraintes de sac-à-dos qui peuvent poser des difficultés dès que la $\sum_{t_i \in \mathcal{T}} m_i$ est proche de $\sum_{k \in \mathcal{P}} \mu_k$. La même chose se produit avec le facteur d'utilisation dès que $\sum_{t_i \in \mathcal{T}} C_i/T_i$ est proche de m. En cas de gros trafic de messages, il peut être judicieux de chercher en priorité à rassembler les tâches qui s'échangent des données. Pour tous ces éléments, des stratégies dédiées peuvent être mises en œuvre, mais c'est souvent l'interaction de toutes ces problématiques entre elles qui rend le problème difficile.

Cependant, la stratégie de décomposition du problème exige que l'interaction entre l'allocation et l'ordonnançabilité (problème maître et esclaves) soit aussi efficace que possible. La contrainte du facteur d'utilisation peut se voir comme une relaxation de l'ordonnançabilité et contraint déjà les solutions du maître vers des solutions qui ont une chance d'être ordonnançables. Une deuxième façon de renforcer la prise en compte par le maître de la problématique de l'esclave passe par l'heuristique de recherche. Une stratégie (à l'origine de la notion d'impact expliquée présentée chapitre 6) basée sur les explications extraites au fil de la résolution a été mise en place dans cet objectif. L'ensemble des explications de la forme (T, M) extraites à un instant de la résolution est noté E. Une tâche est d'autant plus difficile à ordonnancer qu'elle intervient **fréquemment** dans de **petites** explications d'échecs. Pour une tâche t_i son *impact* TI_i est donc évalué par rapport à E de la manière suivante :

$$TI_i = \sum_{e \in E | x_i \in e_T} \frac{1}{|e_T|} + \sum_{e \in E | \exists k, (c_{ik} \vee c_{ki}) \in e_M} \frac{1}{|e_M|}$$

La tâche non instanciée de plus grand TI_i est choisie en priorité. Le choix du processeur est ensuite effectué de manière à minimiser l'impact de la tâche avec ses voisines. On évalue à cette fin un impact entre paire de variables :

$$TI_{ij} = \sum_{e \in E | (x_i, x_j) \in e_T^2} \frac{1}{|e_T|} - \sum_{e \in E | (c_{ij} \vee c_{ji}) \in e_M} \frac{1}{|e_M|}$$

Pour chaque processeur p_k du domaine de x_i on évalue $\sum_{\forall j, x_j = p_k} TI_{ij}$ et le processeur qui minimise cette somme est retenu pour l'affectation. En minimisant cette quantité, on cherche bien à séparer les tâches impliquées dans e_T et à rassembler celles qui sont impliquées dans e_M.

9.2.1.5 Positionnement de l'approche

Cette approche est fortement reliée à la décomposition de Benders (abordée dans son contexte générique chapitre 8) et plus particulièrement au cadre de Hooker intitulé *logic based Benders decomposition* [HO03]. Le sous-problème n'est pas linéaire ; la preuve logique de son inconsistance ainsi que le calcul de sa justification relèvent d'un *inference dual* dédié au problème de l'ordonnançabilité temps-réel et encapsulé dans un algorithme de calcul d'explication. Néanmoins, la décomposition ne s'effectue pas à proprement parler au sein des variables mais plutôt au sein des contraintes. Il s'agit plus précisément de *branch and check* [Tho01]

qui peut être vue comme une forme de décomposition de Benders dans laquelle la définition du problème esclave est faite sur un ensemble vide de variables et un ensemble non vide de contraintes.

L'intérêt de l'approche est de limiter la résolution d'un sous-problème coûteux (ici des équations en nombre entiers avec parties entières résolues en temps pseudo-polynômial) en extrayant une coupe qui synthétise le résultat de cette résolution et la généralise à de nombreux sous-problèmes semblables. Le sous-problème est ainsi résolu une seule fois et l'information issue de sa résolution est ensuite propagée beaucoup plus efficacement pour le maître. La complexité exacte du sous-problème est inconnue dans notre cas (Pas de preuve de NP-Complétude mais aucun algorithme polynômial connu) mais l'algorithme utilisé est pseudo-polynômial.

Dans notre cas, les techniques analytiques de la communauté temps-réel constituent des preuves d'ordonnançabilité et se présentent naturellement comme un *inference dual* (au sens de la dualité logique présentée section 8.1.1).

[Tho01, HOTK00] soulignent les problèmes de calculs redondants pour le maître et de son incrémentalité face à l'ajout dynamique de coupes. La prise en compte des coupes en *dynamic backtracking* par le maître est une réponse à ce problème. C'est MAC-CBJ qui s'est révélé dans notre cas le meilleur compromis entre MAC et MAC-DBT pour intégrer efficacement les coupes sans payer un sur-coût trop important dû à la gestion des explications. Enfin, l'accumulation de coupes ayant un faible pouvoir de filtrage peut finir par ralentir significativement la propagation. Deux pistes ont été explorées sans succès à cette fin : la reformulation de ces coupes par la reconnaissance de contraintes globales ou leur compilation dans un automate. Cette question reste donc ouverte dans notre approche et pourrait constituer une piste sérieuse d'amélioration.

9.2.2 Résolution : vers une contrainte globale d'ordonnançabilité

Avec le recul de la décomposition, il semble également possible de s'attaquer à l'ordonnançabilité avec une contrainte globale dédiée à cette problématique. Nous revenons donc ici sur les équations d'ordonnançabilité, leur résolution et complexité afin de concevoir un algorithme de filtrage.

9.2.2.1 Equations en nombres entiers avec partie entière

Les équations mises en jeu dans l'analyse temps-réel ont la forme suivante :

$$x = \sum_{i=1}^{k} \left\lceil \frac{x}{a_i} \right\rceil b_i + c \tag{9.17}$$

où $x \in \mathbb{N}$. En notant $f(x) = \sum_{i=1}^{k} \left\lceil \frac{x}{a_i} \right\rceil b_i + c$, le problème consiste simplement à trouver le point fixe x^* de f tel que $f(x^*) = x^*$. La résolution d'une telle équation ne semble pas triviale étant donné que la partie entière n'est pas continue. Néanmoins, on peut montrer facilement qu'une condition nécessaire et suffisante de l'existence d'une solution s'écrit $\sum_i \frac{b_i}{a_i} < 1$ et qu'un simple algorithme de point fixe (voir algorithme 20) converge vers x^*[6]. La preuve de cette convergence est donnée annexe A.

La preuve annexe A permet aussi de montrer que la complexité de l'algorithme du point fixe est pseudo-polynômiale en fonction des b_i, a_i. Cela signifie notamment que le nombre d'itération peut grandement varier selon la valeurs de ces b_i, a_i pour un même nombre de tâches k. A notre connaissance, la complexité exacte de ce problème n'est pas connue et seuls des algorithmes d'approximation polynômiaux existent [Ric06]. Ce résultat justifie la légitimité de l'approche par décomposition qui vise à limiter la résolution de ces équations par une approche non

[6] [LM01] va même jusqu'à reformuler l'équation comme un programme linéaire en nombres entiers pour la résoudre !

procédure : PointFixe(f)

```
1: x ← c ;
2: while x ≠ f(x) do
3:     x ← f(x) ;
4: end while
5: return x ;
```

Algorithme 20: Résolution de $f(x) = x$ pour $f(x) = \sum_{i=1}^{k} \left\lceil \frac{x}{a_i} \right\rceil b_i + c$.

polynômiale. La contrainte globale va nous permettre de vérifier en pratique si ces équations sont trop difficiles pour être résolues de manière répétée à des fins de filtrage à tout nœud de l'arbre.

9.2.2.2 Filtrage simple pour l'ordonnançabilité

Nous proposons ici d'encapsuler la résolution des équations précédentes au sein d'une contrainte globale pour effectuer le filtrage lié à l'ordonnançabilité du problème :

Définition 9.2.2 *La contrainte Ordonnancabilite*$(\{x_1, \ldots, x_n\}, AM = (\mathcal{P}, \mathcal{N}), AL = (\mathcal{T}, \mathcal{C}))$ *assure l'ordonnançabilité du placement défini par les* $n = |\mathcal{T}|$ *variables* $\{x_1, \ldots, x_n\}$ *de domaine* $[1, \ldots, |\mathcal{P}|]$ *dans le contexte d'une architecture matérielle AM et logicielle AL donnée.*

La contrainte d'ordonnançabilité est abordée ici pour un réseau CAN en mode désynchronisé.

Propriété 9.2.1 *Réaliser l'arc consistance généralisée (GAC) pour la contrainte d'ordonnançabilité est NP-Difficile.*

Preuve : On peut réduire le problème 3-PARTITION à notre problème d'ordonnançabilité temps-réel (on ignore la présence du réseau - la preuve s'appuie sur [LW82]). 3-PARTITION est un problème NP-Complet au sens fort [GJ79] et se définit de la manière suivante : Soit un ensemble $A = \{a_1, \ldots, a_{3m}\}$ de $3m$ entiers positifs et un entier positif B tel que a_i satisfait $B/4 < a_i < B/2$ pour tout $1 \leq i \leq 3m$ et $\sum_{i=1}^{3m} a_i = mB$. Existe-t-il une partition de A en m ensembles disjoints $A_1, \ldots, A_m$ tels que $\sum_{a_i \in A_j} a_i = B$ pour tout $1 \leq j \leq m$? (On peut noter que les contraintes sur a_i impliquent que chaque A_j contient exactement trois éléments de A.)

On considère une instance $A = \{a_1, \ldots, a_{3m}\}$ et B du problème de 3-PARTITION. On construit un système temps-réel de $3m$ tâches $t_1, \ldots, t_{3m}$ avec $T_i = B$, $C_i = a_i$ et une priorité arbitraire $prio_i = i$ pour tout $1 \leq i \leq 3m$. On peut montrer que le système est ordonnançable si et seulement si 3-PARTITION possède une solution.

- $\Leftarrow$: Si 3-PARTITION possède une solution, donc telle que $\sum_{a_i \in A_j} a_i = B$, alors $\forall a_i \in A_j, R_i \leq B$. En effet la tâche la moins prioritaire parmi les trois tâches affectées a toujours B pour pire temps de réponse. B est bien solution de Eq.9.8 :

$$\begin{aligned} R_i &= C_i + \sum_{t_j \in hp_i(A)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \\ &= a_i + \sum_{t_j \in hp_i(A)} \left\lceil \frac{R_i}{B} \right\rceil a_j \\ &= a_i + \sum_{t_j \in hp_i(A)} a_j \\ &= \sum_{a_i \in A_j} a_i \\ &= B \end{aligned}$$

  Comme B est également la période, on a $R_i \leq B$ et le système est bien ordonnançable.

Procédure : filtreOrdoInst($x_i = p$) throws Contradiction

1: **if** $(a(i) = -1)$
2: **for all**$(x_k \in link_i)$ **do**
3: instancie x_k à p ;
4: MAJMessages(x_i) ;
5: MAJPlacement($x_i = p$) ;

Algorithme 21: Instanciation de x_i à p.

Procédure : filtreOrdoRem($x_i \neq p$) throws Contradiction

1: **for all** $(x_k \in link_i)$ **do**
2: retire p de D_k
3: MAJMessages(x_i) ;

Algorithme 22: Retrait de la valeur p de x_i

– $\Rightarrow$: Si le système est ordonnançable, il possède exactement 3 tâches sur chaque processeur. Dans le cas contraire, le système est non ordonnançable en considérant uniquement les temps d'exécutions $C_i = a_i$. En effet, un processeur avec deux tâches signifie quatre tâches sur un autre et comme $a_i > B/4$, $\sum_{a_i \in A_j} a_i > B$ et donc $R_i > B$ ce qui contredit l'ordonnançabilité. De plus $\sum_{a_i \in A_j} a_i = B$ pour chaque processeur puisque $\sum_{i=1}^{3m} a_i = mB$ (si on avait $\sum_{a_i \in A_j} a_i < B$ sur un processeur on aurait $\sum_{a_i \in A_j} a_i > B$ sur un autre).

La construction précédente se fait en temps polynômial ce qui prouve que 3-PARTITION est réductible à l'ordonnançabilité et que la GAC sur l'ordonnançabilité est NP-Dur. La complexité de l'équation 9.8 étant inconnue, on ne sait pas pour le moment si il s'agit d'un problème NP-Complet. □

Notre algorithme de filtrage est dès lors basé sur une relaxation de l'arc-consistance. Pour décrire précisément l'algorithme, la notion de placement (section 9.1.2) est étendue à celle de placement partiel.

Définition 9.2.3 *Un placement partiel est une application de* $a : \mathcal{U} \subset \mathcal{T} \rightarrow \mathcal{P} \cup \{-1\}$ *telle que :*

$$\begin{array}{ll} t_i \in \mathcal{U} & \mapsto a(t_i) = p_k \\ t_i \in \mathcal{T} - \mathcal{U} & \mapsto a(t_i) = -1 \end{array} \tag{9.18}$$

On note $a_{\mathcal{U}}$ *le sous-ensemble de tâches* $\mathcal{U}$ *sur lequel porte* a.

Définition 9.2.4 *L'extension d'un placement partiel* a *par l'affectation d'une tâche* t_i *sur un processeur* p *(*$x_i = p$*) telle que* $t_i \notin a_{\mathcal{U}}$ *est noté* $a' \leftarrow a + (x_i = p)$ *avec* $\forall t_k \in a_{\mathcal{U}}$, $a'(t_k) = a(t_k)$ *et* $a'(t_i) = p$.

Le filtrage réalise de simples raisonnements hypothétiques analogues à du *forward-checking* qui suppriment les valeurs menant directement à un échec. Les algorithmes 21 et 22 synthétisent le filtrage et s'appuient sur trois structures de données restorables au *backtrack* :

– le placement partiel courant a ;
– l'ensemble mes des messages qui transitent sur le réseau (mes contient des messages dont les tâches émettrice/receptrice ne font pas encore partie de a mais dont les domaines sont disjoints) ;
– une liste $link_i$ de tâches qui échangent des données avec t_i et dont la co-résidence avec t_i est établie sous peine de violer l'ordonnançabilité du réseau. Autrement dit, l'invariant suivant est maintenu : $\forall x_k \in link_i, x_k = x_i$.

Le cœur de l'algorithme réside dans les méthodes $ordo(a, b)$ et $ordoMessages(mes, q)$ qui font appel à l'algorithme 20 sur les équations 9.9 et 9.16. $ordo(a, b)$ (resp. $ordoMessages(mes, q)$) établit l'ordonnançabilité de b (resp. de q) sachant que b (resp. q) est une extension de a (resp.

Procédure : MAJPlacement($x_i = p$) throws Contradiction

```
1: a_1 ← a + (x_i = p) ;
2: for all(x_k ∈ link_i) do
3:    a_1 ← a_1 + (x_k = v) ;
4: if ( !ordo(a, a_1))
5:    throw new Contradiction() ;
6: a ← a_1 ;
7: for all x_j telle que |D_j| > 1 et p ∈ D_j do
8:    if (link_j ≠ ∅)
9:       for all (x_k ∈ link_j avec k > j) do
10:         if ( !ordo(a, a + (x_j, x_k) = p))
11:            retire p de D_k, D_j ;
12:   else if ( !ordo(a, a + (x_j = p)))
13:      retire p de D_j ;
```

Algorithme 23: Filtrage lié à la mise à jour du placement partiel a sur une nouvelle instanciation $(x_i = p)$.

Procédure : MAJMessages(x_i) throws Contradiction

```
1: mesChanged ← false ;
2: for all c_ij ∉ mes avec D_i ∩ D_j = ∅
3:    if ( !ordoMessage(mes,mes ∪ {c_ij}))
4:       throw new Contradiction() ;
5:    mes ← mes ∪ {c_ij} ;
6:    mesChanged ← true ;
7: if (mesChanged)
8:    for all (c_kl ∈ C avec c_kl ∉ mes) do
9:       if !ordoMessage(mes,mes ∪ {c_kl})
10:         link_k ← link_k ∪ {x_l} ;
11:         link_l ← link_l ∪ {x_k} ;
12:         D_k, D_l ← D_k ∩ D_l ;
13:         for all p ∈ D_k ;
14:            if ( !ordo(a, a + (x_l, x_k) = p))
15:               retire p de D_k, D_l ;
```

Algorithme 24: Filtrage lié à la mise à jour de l'ensemble sûr des messages m et des nouvelles relations de co-résidence entre tâches $(link_i)$.

mes) dont l'ordonnançabilité est connue. Leur incrémentalité s'appuie sur les deux remarques suivantes (valables en CAN) :

- l'affectation d'une tâche t_i sur un processeur p remet en question l'ordonnançabilité des tâches déjà placées sur p et de priorités inférieures à t_i uniquement ;
- l'envoi d'un nouveau message M_{ab}, remet en question l'ordonnançabilité des messages de priorités plus faibles et ceux de priorités plus élevées M_{ij} telle que $C_{ab} > max_{M_{kl} \in lp_{ij}(a)} C_{kl}$ est remise en question.

Le filtrage est lié à la modification du placement courant a (algorithme 23) ou la modification de l'ensemble sûr des messages (algorithme 24). Dans les deux cas, l'ordonnançabilité du nouvel état est vérifiée (lignes 4-5 de 23 et 2-3 de 24) et les structures de données a, mes et $link_i$ sont mises à jour. En l'absence d'échec, une étape de filtrage s'enclenche :

1. à l'ajout d'un nouveau message dans l'ensemble mes, l'ordonnançabilité de tous les autres messages non sûrs ($\notin mes$) est examinée. En cas d'échec, un lien de co-résidence est ajouté entre les tâches (lignes 10 et 11) et leur ordonnançabilité conjointe sur tous les processeurs à l'intersection de leurs domaines est évaluée (lignes 12-15) ;
2. à l'ajout d'une tâche dans le placement a sur un processeur p, c'est l'ordonnançabilité de toutes les tâches non instanciées et susceptibles d'aller sur p qui est évaluée en tenant compte de leurs liens de co-résidence (lignes 7 à 13 de 21).

Ces traitements sont appliqués en cas d'un événement de retrait de valeur (algorithme 22) et d'instanciation (algorithme 21). Les lignes 1, 2 et 2, 3 des algorithmes 22 et 21 assurent notamment la propagation des liens de co-résidence connus.

Propriété 9.2.2 *Les algorithmes 21 et 22 garantissent que :*

- *Le placement partiel a, défini par les tâches instanciées, est ordonnançable.*
- *Toute extension de a par l'affectation d'une tâche t_i non instanciée sur un processeur p tel que $p \in D_i$ $(x_i = p)$ est ordonnançable.*

Preuve : Une fois le point fixe de 21 et 22 atteint, l'ordonnançabilité du placement courant a est assurée par les lignes 1-6 de l'algorithme 23 et 1-6 de l'algorithme 24 qui garantissent l'ordonnançabilité des processeurs et du réseau. Une extension de a par $x_i = p$ peut être non ordonnançable pour deux raisons seulement : x_i est non ordonnançable avec les tâches placé sur p (les lignes 7-13 garantissent que ce n'est pas le cas puisque $p \in D_i$) ; l'affectation de x_i engendre un message non ordonnançable parce qu'elle est en communication avec une tâche affectée ailleurs (les lignes 8-15 préviennent cette situation) □

La complexité du traitement d'un événement de propagation est dominée par le filtrage sur les messages (lignes 8 et 13 de l'algorithme 24) et les tâches (ligne 7 de l'algorithme 21). La condition $k > j$ de la ligne 9 garantit que chaque tâche n'est testée qu'une seule fois. La complexité est donc en $O(n+c\times m)$. Pour limiter au maximum les appels à *ordo* et $ordoMessage$ qui résolvent des équations en nombres entiers avec des parties entières, on peut pré-calculer des relations de dominance entre les tâches et les exploiter au moment du filtrage :

Propriété 9.2.3 *Si $x_i = p_k$ rend t_i non ordonnançable, alors p_k peut être éliminé du domaine D_j des tâches t_j telles que $prio_j < prio_i \wedge C_j \geq C_i \wedge T_j \leq T_i$.*

Propriété 9.2.4 *Si $x_i = p_k$ rend la tâche t_b non ordonnançable, alors p_k peut être éliminé du domaine D_j des tâches t_j telles que $prio_j > prio_b \wedge C_j \geq C_i \wedge T_j \leq T_i$.*

Preuve : Les preuves de ces deux règles de dominance sont immédiates par récurrence sur la valeur de R_j^k ou R_b^k à chaque itération k de l'algorithme du point fixe. □

Pour utiliser ce filtrage dans un cadre de *backtrack* intelligent tous les retraits de valeurs et instanciations effectués doivent être expliqués. Ce sont les mêmes explications que précédemment à la seule différence qu'elles ne sont pas minimisées par XPLAIN, étant calculées à la volée. On notera que cette approche réalise des inférences, absents de l'approche par décomposition, en établissant et testant l'ordonnançabilité conjointe des tâches qui génèrent des messages non ordonnançables.

Le filtrage introduit reste très simple (mais déjà relativement coûteux par la résolution des équations du type 9.17) et passe à côté de déductions qui pourraient énormément améliorer la résolution. Nous en donnons ici un exemple :

Exemple 17 *Filtrage Ordonnançabilité*

Soit un système de 5 tâches, $\mathcal{T} = \{t_1, \ldots, t_5\}$ et deux processeurs $\mathcal{P} = \{p_1, p_2\}$. Imaginons que tous les triplets de tâches (10 triplets existent) soit non-ordonnançables. L'affectation de 5 tâches sur 2 processeurs impose que trois d'entre elles au moins soient finalement regroupées ce qui est impossible et l'approche décrite doit brancher sur deux tâches pour détecter l'inconsistance.

L'exemple précédent nous suggère des pistes d'amélioration du filtrage abordé dans la section suivante.

9.2.2.3 Filtrage avancé

Le problème peut se voir comme un problème d'affectation dans un graphe biparti des tâches et processeurs (voir figure 9.5). L'ordonnançabilité impose que certains ensembles d'arcs entrants au niveau des processeurs ne soient pas pris simultanément. Par ailleurs, ces ensembles ne sont pas connus *a priori*, leur nombre étant potentiellement exponentiel. On peut ainsi voir la contrainte comme un problème de coloration dans un hyper-graphe $H = (E_1, \ldots, E_c)$ (une famille de parties de $\mathcal{T}$ telle que $E_i \subsetneq E_j$) pour lequel chaque noeud est associé à une tâche et chaque hyper-arc E_i à un ensemble incompatible de tâches au sens des équations d'ordonnançabilité. Une condition nécessaire et suffisante d'ordonnançabilité est donc de savoir si il

existe ou non une m-coloration de H (la couleur représente le processeur affecté à chaque tâche). Or il y a en plus un nombre exponentiel de sous-ensembles de tâches incompatibles [7]. Néanmoins, l'exemple 17 montre que l'estimation du nombre maximal de tâches (ub_i sur la figure 9.5) ordonnançables sur chaque processeur p_i pourrait apporter un filtrage supplémentaire précieux.

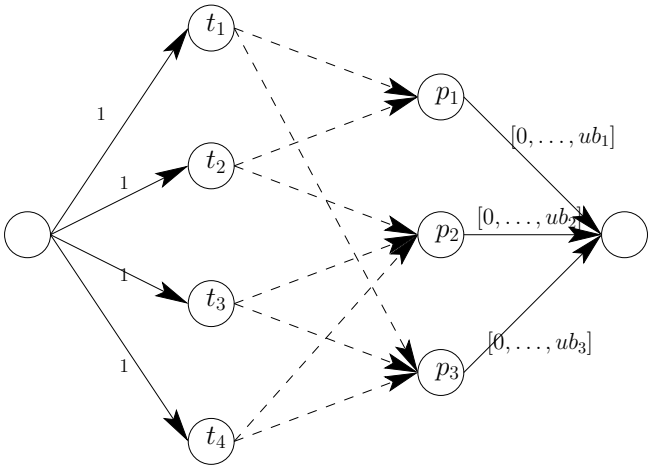


FIG. 9.5: Représentation du problème d'ordonnançabilité comme un problème d'affectation des tâches sur les processeurs.

La connaissance des ub_i permettrait d'obtenir le filtrage d'une contrainte de *global cardinality* (gcc) représentée par le flot de la figure 9.5. On aurait ainsi un modèle redondant du type :

$$\begin{array}{ll} & gcc(\{x_1, \ldots, x_n\}, \{[0, ub_1], \ldots, [0, ub_m]\}) \\ \forall 0 < p \leq m & ub_p = max_{S \subset \{t_i \in \mathcal{T}, |D_i|>1\}}(|S|, ordo(S)) + |\{x_i, x_i = p\}| \end{array}$$

La contrainte $max_{S \subset \{t_i \in \mathcal{T}, |D_i|>1\}}(|S|, ordo(S))$ fait référence à un problème de stable maximum dans l'hyper-graphe H des ensembles incompatibles de tâches non instanciées. Néanmoins, une borne supérieure de la cardinalité de ce stable pourrait déjà améliorer la prise en compte de l'ordonnançabilité.

9.3 Évaluation expérimentale

9.3.1 Jeux de données

Pour les problèmes de placement, il n'existe pas de benchmark au sein de la communauté temps réel. En général, les expérimentations sont menées sur des exemples *didactiques* [TBW92, AH98] ou à partir d'un générateur aléatoire de configurations [Ric02, Ram90, MBD98]. Nous avons choisi cette dernière solution. Les paramètres d'entrée du générateur sont :

- n, m, mes : le nombre de tâches, de processeurs (par la suite : $n = 40$ et $m = 7$) et de messages ;
- $\%_{global}$: le pourcentage sur le facteur global d'utilisation ;

[7] C'est aussi l'intérêt de la procédure de génération dynamique de contraintes (de la section précédente) qui peut aboutir à la solution sans avoir à énumérer toutes ces coupes.

Alloc.	$\%_{mem}$	$\%_{res}$	$\%_{co-res}$	$\%_{exc}$		Sched.	$\%_{global}$		Mes.	mes/n	$\%_{msize}$
1	80	0	0	0		1	40		1	0.5	40
2	40	15	15	15		2	60		2	0.5	70
3	30	25	25	25		3	75		3	0.75	70
4	15	35	35	35		4	90		4	0.875	150

TAB. 9.3: Classes de dureté

- $\%_{mem}$: la surcapacité mémoire ;
- $\%_{res}$: le pourcentage de tâches ayant une contrainte de résidence ;
- $\%_{co-res}$: le pourcentage de tâches ayant une contrainte de co-résidence ;
- $\%_{exc}$: le pourcentage de tâches ayant une contrainte d'exclusion ;
- $\%_{msize}$: la taille des messages est évaluée comme un pourcentage des périodes des tâches émettrices/réceptrices.

Les périodes des tâches ainsi que les priorités sont obtenues de façon aléatoire. Les temps d'exécution sont tirés aléatoirement et réévalués pour que $\sum_{i=1}^{n} \frac{WCET_i}{T_i} = m\%_{global}$. Le besoin mémoire d'une tâche est proportionnel à son pire temps d'exécution. Les capacités mémoires des processeurs sont obtenues aléatoirement et respectent $\sum_{k=1}^{m} \mu_k = (1 + \%_{mem}) \sum_{i=1}^{n} m_i$. Pour des raisons de simplicité, on se limite à des communications linéaires (le graphe de communication est réduit à un arbre).

Pour pouvoir caractériser nos expérimentations et cerner la transition de phase du problème, nous avons défini plusieurs catégories d'instances dépendant de la dureté respective de l'allocation et de l'ordonnancement. Des notes de 1 à 3 qualifient cette difficulté. La dureté de l'ordonnancement est évaluée sur le facteur global d'utilisation, $\%_{global}$, qui varie de 40 à 90 %. Celle du placement est basée sur le nombre des tâches incluses dans des contraintes de résidence, co-résidence et exclusion ($\%_{res}$, $\%_{co-res}$ et $\%_{exc}$). La surcapacité mémoire, $\%_{mem}$, allouée à l'ensemble des processeurs entre aussi en jeu (une très faible surcapacité conduit à résoudre un problème de *packing* qui peut devenir difficile). Enfin, les ratios mes/n et $\%_{msize}$ caractérisent la dureté du réseau. Le tableau 9.3 résume ces catégories. Les problèmes sont ensuite décrits par un quadruplet W-X-Y-Z qui désigne un problème avec les paramètres mémoire de la classe W, allocation de la classe X, ordonnançabilité de la classe Y et messages de la classe Z.

Huit catégories seront considérées par la suite avec cent instances par catégorie soit un jeu de tests constitué de huit cents instances au total.

9.3.2 Résultats

Deux approches sont comparées sur ces données :

- Benders : l'approche fondée sur le calcul d'explications et la décomposition de l'allocation et l'ordonnançabilité (décrite section 9.2.1) ;
- CP : l'approche par programmation par contraintes pures (décrite section 9.2.2) ;

EXPL et CP sont mises en œuvre en *backjumping*. Le but n'est pas d'évaluer, l'intérêt du *backjumping* pour ce problème, mais l'intérêt de l'approche par décomposition et calcul d'explications. Nous présentons dans un premier temps, les meilleurs résultats (obtenus avec I_R introduite au chapitre 6) pour les approches CP et Benders. Nous étudions également le comportement de Benders en détail. Dans un deuxième temps, nous examinons sur le même jeu d'instances les résultats obtenus par les différentes heuristiques. Enfin, le comportement des algorithmes pour des systèmes de tailles supérieures est analysé.

9.3.2.1 Meilleurs résultats

Le tableau 9.4 synthétise les résultats par pourcentage d'instances résolues (%Suc), nombre d'instances consistantes (NbC), temps de résolution en secondes (Tps(s)), nombre de nœuds explorés (Nœud), nombre d'itérations (Iter), nombre de coupes extraites en distinguant les *notAllEqual* (Noe) et le nombre de combinaisons linéaires (Comb). Les valeurs moyennes de ces données sont présentées sur les instances résolues (solution trouvée ou preuve d'inconsistance effectuée) parmi 100 instances par catégorie dans une limite de temps de 10 minutes.

	Benders (I_R)							**CP (I_R)**			
	%Suc	NbC	Tps(s)	Nœud	Iter	Noe	Cut	%Suc	NbC	Tps(s)	Nœud
1-1-3-1	99	99	5,6	3105,9	107,5	482,5	NA	99	99	**3,8**	**1685,3**
1-2-2-3	99	76	15,5	7227,4	629,6	192,8	1853,2	**100**	76	**5,1**	**1085,8**
2-2-2-1	100	56	**0,4**	**370,6**	19,9	79,7	NA	100	56	5,3	1985,9
2-2-2-2	100	70	0,3	282,7	17,2	48,5	12,5	100	70	0,3	**36,6**
2-2-2-3	92	70	28,2	10276,8	662	262,2	1784,9	**95**	73	**13,3**	**4323,3**
2-2-3-1	90	30	**8,3**	3267,8	30,1	154,2	NA	90	30	10,4	**4059,8**
2-3-2-1	100	19	0,4	172,6	6,8	37,3	NA	100	19	**0,3**	**9,2**
3-2-2-1	99	57	1,4	960,2	39,6	143,2	NA	99	57	1,4	**458,1**

TAB. 9.4: Valeurs moyennes des données obtenues sur la résolution. Seules les instances résolues parmi 100 instances par catégorie dans une limite de temps de 10 minutes (par instance) sont considérées.

Le tableau 9.5 détaille les valeurs moyennes, médianes, minimales et maximales des temps de résolution.

	Benders (I_R)					**CP (I_R)**				
	%Suc	Tps(s)				%Suc	Tps(s)			
		Moy	Med	Min	Max		Moy	Med	Min	Max
1-1-3-1	99	5,6	2,8	**0,27**	**60,6**	99	**3,8**	**0,45**	0,3	78,6
1-2-2-3	99	15,5	2,6	**0,19**	519,8	**100**	**5,1**	**0,58**	0,23	**336,2**
2-2-2-1	100	**0,4**	**0,27**	**0,17**	**4,2**	100	5,3	0,34	0,23	314,9
2-2-2-2	100	0,3	**0,29**	0	**2,6**	100	0,3	0,31	0	4,1
2-2-2-3	92	28,2	4,7	0	386,9	**95**	**13,3**	**0,73**	0	403,4
2-2-3-1	90	**8,3**	**0,25**	0,17	**344**	90	10,4	0,36	**0**	375
2-3-2-1	100	0,4	**0,26**	0,17	6,5	100	**0,3**	0,37	**0**	**1,9**
3-2-2-1	99	1,4	**0,32**	**0,17**	**15**	99	1,4	0,33	0,23	91,2

TAB. 9.5: Valeurs moyennes, médianes, minimales et maximales des temps de résolution sur les instances résolues parmi 100 instances dans une limite de temps de 10 minutes (par instance) pour chaque catégorie.

Ces résultats montrent que l'approche par décomposition est très compétitive (97,3 % d'instances résolues) avec l'approche CP classique représentée ici par la contrainte globale d'ordonnançabilité (97,8 % d'instances résolues). Les valeurs minimales et médianes des temps de résolution montrent que Benders peut souvent s'avérer plus rapide dans le meilleur des cas (sauf quand la contrainte peut prouver l'inconsistance au nœud racine ce qui se produit sur une instance au moins quand les valeurs minimales sont à zéro pour CP). Ainsi, quand le problème d'ordonnançabilité est facile (cas de la catégorie 2-2-2-1 où le temps moyen de Benders est significativement plus faible que CP), CP s'embarrasse de la résolution coûteuse des équations d'ordonnançabilité sans obtenir de réduction significative de l'espace de recherche et Benders est dès lors plus rapide. On peut ainsi générer des classes de problèmes où la décomposition sera un choix judicieux par rapport au filtrage. Enfin le nombre de nœuds de CP est nettement

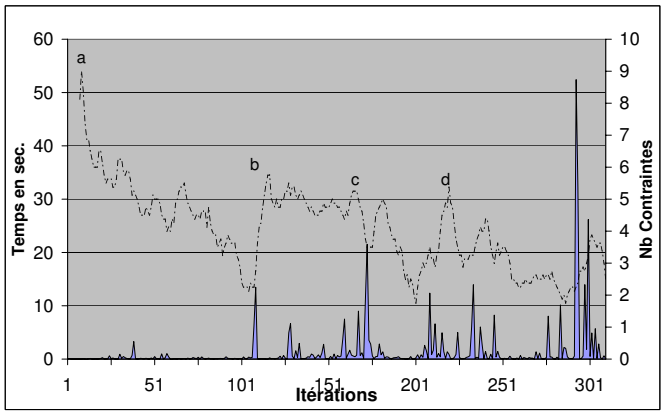


FIG. 9.6: Détail d'exécution comprenant le temps de résolution ainsi qu'une moyenne glissante (pas de 10) du nombre de coupes extraites à chaque itération (en pointillés). (310 itérations, 1192 *NotAllEqual*, 75 *différences* en partie reformulée en 12 *alldifferent*, près de 7 min. de résolution pour obtenir une solution)

plus faible puisque Benders demande une phase d'exploration plus longue pour extraire des coupes permettant de converger.

Une instance particulière est détaillée figure 9.6[8]. On peut observer l'évolution du temps de résolution et du nombre de contraintes apprises à chaque itération. L'apprentissage est très rapide grâce à l'incrémentalité de la résolution qui adapte la solution courante. La solution *pivote* (zone *a*-*b*) dans un voisinage restreint et le nombre de coupes extraites décroît jusqu'à la formulation d'un problème dur de satisfaction (point *b*) qui oblige le problème maître à changer radicalement de région de l'espace pour fournir une solution faisable. Le processus recommence alors avec un apprentissage à nouveau très rapide (*b*-*c*,*c*-*d*, ...).

9.3.2.2 Comparaison des heuristiques

La table 9.6 donne les pourcentages de réussite de chaque heuristique. I_R et I_2 sont les heuristiques introduites section 6 et *Learn* est l'heuristique présentée à la section 9.2.1.4 qui représente une forme d'impact limité à l'ordonnançabilité. I_R et I_2 utilisent une procédure de *restart* (espace de recherche initial égal à $10 \times n$ et multiplié par 1,5 à chaque *restart*) et d'initialisation (propagation de chaque valeur de chaque variable). La procédure de *restart* reprend à chaque itération pour Benders puisque le problème évolue avec l'ajout de coupes.

I_2 et I_R obtiennent les meilleures performances avec un léger avantage pour I_R. Les valeurs moyennes, médianes, minimales et maximales pour toutes les heuristiques et sur toutes les données utiles pour comprendre la résolution en distinguant les problèmes consistants et inconsistants sont données en annexe B.

9.3.2.3 Passage à l'échelle

La taille des systèmes évalués (n = 40, m = 7) correspond à l'ordre de grandeur des systèmes réels traités dans les systèmes embarqués. [RRC03] présente ainsi un cas d'étude sur un système réel de 24 tâches et 5 processeurs identiques. La figure 9.7 permet d'analyser la robustesse de notre approche sur des systèmes de plus grandes tailles. La méthode CP est utilisée pour cette étude. De nouveau, seules les données sur les instances résolues en moins de 10 minutes sont

[8]Il s'agit ici d'une résolution avec DBTpour le maître

	Benders				CP		
	I_2	I_R	Learn	Mindom	I_2	I_R	Mindom
1-1-3-1	92	**99**	75	67	96	**99**	86
1-2-2-3	93	99	81	43	97	**100**	73
2-2-2-1	**100**	**100**	95	90	**100**	**100**	90
2-2-2-2	**100**	**100**	**100**	95	**100**	**100**	94
2-2-2-3	85	92	69	43	94	**95**	75
2-2-3-1	73	**90**	38	43	85	**90**	72
2-3-2-1	100	**100**	91	81	**100**	**100**	90
3-2-2-1	99	99	82	85	**100**	99	87
Moy	92,75	97,375	78,875	68,375	96,5	97,875	83,375

TAB. 9.6: Pourcentage d'instances résolues dans une limite de temps de 10 minutes (par instance) pour différentes heuristiques

prises en compte mais seulement 5% des instances ne sont pas résolues au pic (6 processeurs et 80 tâches). Cent instances sont considérées pour chaque point.

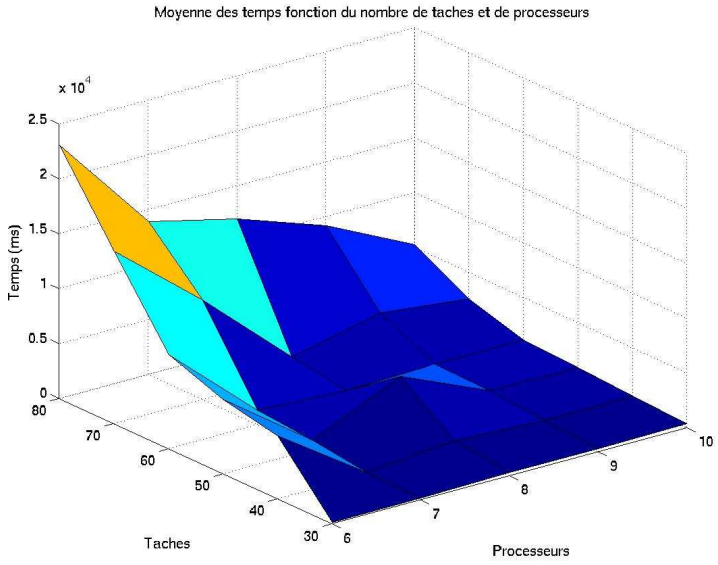


FIG. 9.7: Moyenne des temps de résolutions obtenues en faisant varier le nombre de processeurs et le nombre de tâches pour la catégorie 2-2-2-2.

Les résultats avec Benders présentent le même profil avec des temps moyens deux fois plus élevés. Ces résultats montrent que la méthode est parfaitement applicable dans l'état actuel aux échelles des systèmes industriels.

9.4 Conclusion

Nous avons proposé différentes approches complètes pour un problème dont la résolution optimale a été peu étudié jusqu'à présent. Un cadre original de décomposition fondé sur le calcul des explications a été mis en place de manière à intégrer les problématiques temps-réel d'ordonnançabilité dans une approche PPC. L'approche constitue ainsi un cas particulier de

la décomposition de Benders présentée au chapitre 8 en illustrant aussi les idées du chapitre 6 sur les heuristiques de recherche.

L'originalité du *branch and check* mis en œuvre est de repousser la prise en compte de contraintes délicates et coûteuses pour le solveur et d'éviter leurs résolutions répétées en conservant une coupe de Benders que le problème maître peut par ailleurs traiter beaucoup plus efficacement. La complexité non polynômiale de l'algorithme de résolution pour ces équations justifie ainsi l'approche.

Les efforts pour mettre en œuvre un filtrage incrémental et efficace basé sur les équations d'ordonnançabilité ont finalement permis de surpasser la décomposition sur le jeu de tests envisagé. Il semble donc que ces équations à parties entières soient suffisamment faciles à résoudre pour intégrer leur traitement à tout nœud de l'arbre de recherche. Ce constat n'était pas évident à l'origine et les résultats montrent que la décomposition est très compétitive et pourrait devenir plus efficace avec des analyses d'ordonnançabilité de plus en plus complexes.

Ces résultats ont été publiés dans [CHD+04].

Chapitre 10

Le MOSP (*Minimum Open Stack Problem*)

Cette application illustre les idées du chapitre 7 en proposant une approche reposant entièrement sur le calcul de *nogoods généralisés* et leur stockage dans une forme simple d'automate. Ce travail est également le résultat de discussions avec Nicolas Beldiceanu et d'une collaboration avec Thierry Benoist. Le problème est introduit section 10.1, les sections 10.2, 10.3 décrivent respectivement des approches basées d'une part sur l'enregistrement de *nogoods* et le maintien d'explications, d'autre part sur des raisonnements de filtrage adaptés au problème. Les résultats expérimentaux sont finalement présentés section 10.4.

10.1 Description du problème

Le problème du *Minimum Open Stack* (MOSP) est un problème d'ordonnancement qui met en jeu un ensemble de produits P et un ensemble de commandes C. Une commande est constituée d'un sous-ensemble de P et son traitement demande la mobilisation d'une ressource (ou d'un *stack*). Quand le premier produit d'une commande est mis en production, une ressource est mobilisée et la commande est dite **ouverte**. Quand tous les produits requis par une commande ont été fabriqués, la commande est dite **fermée** et la ressource est libérée. Un seul produit est fabriqué à chaque instant et l'objectif est de déterminer un ordre de production des produits qui minimise le nombre maximal de ressources ou *stack* ouvertes simultanément au cours de la production. Une solution du MOSP est donc la donnée d'un ordre total sur les produits décrivant la séquence de production. La table 10.1 présente une instance du problème et sa solution optimale.

	Instance						Solution											
	P_1	P_2	P_3	P_4	P_5	P_6	P_1	P_2	P_6	P_4	P_3	P_5	P_1	P_2	P_6	P_4	P_3	P_5
c_1	0	0	1	0	1	0	0	0	0	0	1	1	-	-	-	-	1	1
c_2	0	1	0	0	0	0	0	1	0	0	0	0	-	1	-	-	-	-
c_3	1	0	1	1	0	0	1	0	0	1	1	0	1	1	1	1	1	-
c_4	1	1	0	0	0	1	1	1	1	0	0	0	1	1	1	-	-	-
c_5	0	0	0	1	1	1	0	0	1	1	0	1	-	-	1	1	1	1

TAB. 10.1: Une instance du MOSP (6x5), sa solution optimale de valeur 3 représentée par la séquence de production et par les plages d'exécution des commandes

Ce problème est équivalent au problème de la largeur de chemin d'un graphe (*path-width problem*) ou celui de la séparation des sommets (*vertex separation problem*).

10.1.1 Modèle mathématique

Les notations suivantes seront utilisées par la suite :
- P est l'ensemble des m produits disponibles
- C est l'ensemble des n commandes.
- $P(c)$ est l'ensemble des produits constituant la commande c ;
- $C(p)$ est l'ensemble des commandes qui utilisent le produit p. Une extension naturelle de cette dernière notation est utilisée pour des ensembles de produits de sorte que $C(s_P)$ désigne l'ensemble des commandes qui utilisent au moins un produit de s_P ;
- $O^K(S)$ correspond à l'ensemble des commandes ouvertes par la mise en production de l'ensemble $S \subseteq K$ de produits : $O^K(S) = |C(S) \cap C(K-S)|$. $O(S)$ est un raccourci pour $O^P(S)$;
- $f(S)$ est le nombre minimal de ressources nécessaires pour compléter un ensemble S de produits et $f^A(S)$ est le nombre minimal de ressources nécessaires au traitement de S sous l'hypothèse d'un ensemble A de commandes ouvertes initialement ;
- p_j indique le produit affecté à la position j dans la séquence ;
- $open_j$ donne le nombre de commandes ouvertes à l'instant j (*i.e.* à la position j de la séquence).
- enfin, D_x^{orig} indique le domaine originel de x et D_x, son domaine courant.

Sur la base de ces notations, on peut donner un modèle mathématique du problème :

$$\begin{array}{ll} \min(& \max_{j<m} open_j) \quad \text{s.t.} \\ & \forall\, 0 < j \leq m, \quad p_j \in [1..m] \\ & \forall\, 0 < j \leq m, \quad open_j \in [1..n] \\ & \text{alldifferent}(\{p_1, \ldots, p_m\}) \\ & open_j = |C(\{p_1, \ldots, p_j\}) \cap C(\{p_j, \ldots, p_m\})| \end{array} \tag{10.1}$$

10.1.2 Résultats connus

Nous présentons à présent succinctement quelques éléments importants pour la résolution efficace du MOSP.

10.1.2.1 Pré-traitement

Propriété 10.1.1 *Tous les produits p pour lesquels $\exists p^{'}, C(p) \subseteq C(p')$ peuvent être retirés sans changer la valeur optimale du problème. On dira que p est dominé par $p^{'}$.*

Preuve : On peut en effet résoudre le problème sans les produits dominés p et insérer ces derniers dans la solution optimale à coté des produits $p^{'}$ sans changer le nombre maximal de commandes ouvertes. □

Ce pré-traitement est très utile et peut également être appliqué dynamiquement pendant la recherche dans le cadre d'une énumération chronologique : si S est l'ensemble courant des produits chronologiquement affectés jusqu'à l'instant t, on peut alors insérer immédiatement après S, les produits p tels que $C(p) \subseteq O(S)$ sans modifier la valeur optimale.

10.1.2.2 Bornes inférieures

Les bornes inférieures sont souvent basées sur le graphe de **co-demande** G qui est défini dans la littérature par [BYS04]. Les nœuds de G sont associés à des commandes et un arc (i, j) est présent si et seulement si les commandes i et j partagent au moins un produit. De nombreuses bornes inférieures peuvent être définies sur ce graphe. Nous avons retenu les trois bornes suivantes :
- le degré minimum + 1 de G ;

– la taille de la clique maximum dans G (obtenue avec l'algorithme de Bron et Kerbosh [BK73]) ;
– la taille de la clique obtenue dans un mineur de G [BYS04] par contraction d'arcs.

La dernière borne apparaît comme la plus puissante dans nos expérimentations, particulièrement avec une bonne heuristique de contraction d'arcs sélectionnant les nœuds parmi ceux de degré minimal pour produire une clique au plus tôt dans la procédure de contraction.

Ces bornes peuvent aussi être utilisées dynamiquement pendant la recherche sur le problème restreint à $P-S$ en prenant en compte les commandes ouvertes à l'instant courant. Inspirée par [GS05], la borne inférieure suivante s'est révélée très intéressante : soit G' la version *orientée* du graphe de co-demande (les arcs (c_1, c_2) et (c_2, c_1) existent tous les deux). Soit $d^+_{G'(c)}$ le degré sortant du nœud c dans G'. G' est défini de la manière suivante : $G' = (V, E - \{(c_1, c_2)|c_2 \in C(S)\})$, c'est à dire tous les arcs pointant vers des commandes ouvertes ont été retirés.

Propriété 10.1.2 *Le nombre minimum de ressources requises pour l'ensemble $P - S$ de produits est au moins de :*

$$lb(P - S) = min_{c \in C(P-S)}(|O(S) \cup c| + d^+_{G'(c)}) \tag{10.2}$$

Preuve : Si c est la première commande fermée, alors, avant de fermer c, toutes les commandes adjacentes à c (les destinations des arcs sortants de c) seront ouvertes. c étant la première commande fermée, aucune de ces commandes adjacentes ne peut être fermée avant c, pas plus que les commandes de $c \cup O(S)$. Ces deux ensembles étant disjoints puisque aucune commande de $c \cup O(S)$ n'est adjacente à c (par définition de G'), la somme de leur taille est une borne inférieure du nombre de ressources ouvertes simultanément. □

10.1.2.3 Techniques de résolution

Un grand nombre d'approches ont été proposées pour résoudre ce problème au cours du challenge de modélisation d'IJCAI 2005 [SG05]. L'une d'entre elles, parmi les plus efficaces, identifiée par [GS05] et [Ben05] est basée sur la programmation dynamique : on considère un ensemble S de produits qui ont été placés chronologiquement jusqu'à l'instant t depuis le début de la séquence ($|S| = t$ et les produits de S sont positionnés depuis l'instant 0 jusqu'à l'instant $t-1$). On peut remarquer que $f^{O(S)}(P-S)$ reste identique quelle que soit la permutation de S. En effet, le problème $P - S$ est uniquement relié au problème P par l'ensemble de commandes actives à l'instant t : $O(S)$ qui ne dépend pas d'un ordre particulier de S (une commande c est en effet ouverte si $P(c) \cap S \neq \varnothing$ et $P(c) \cap (P-S) \neq \varnothing$). Cette observation permet d'obtenir une formulation assez naturelle du problème en programmation dynamique et la fonction objectif peut s'écrire récursivement[1] de la manière suivante :

$$f(P) = min_{j \in P}(max(f(P - \{j\}), |O(P - \{j\})|) \tag{10.3}$$

L'énorme avantage de cette approche est de passer d'un espace de recherche de taille $m!$ à un espace de taille 2^m puisqu'on limite l'exploration aux sous-ensembles de P. Du point de vue de la programmation par contraintes, si S est un *nogood*, c'est à dire un ensemble de produits dont l'infaisabilité a été démontrée (par rapport à la borne supérieure courante), toute permutation de S conduit au même sous-problème infaisable $P - S$. L'enregistrement de ces *nogoods* pendant une énumération chronologique de la séquence de production conduit à un même espace de recherche en 2^m. Les deux approches sont donc équivalentes dans la mesure où elles travaillent sur le même espace de recherche (mais diffèrent en terme d'implémentation).

On reviendra section 10.3.1 sur les différents modèles contraintes du problème. On peut mentionner également les différents travaux suivants :

[1] On considère que si $|P| = 1$ tel que $P = \{p\}$ alors $f(\{p\}) = |C(p)|$.

– [Pes05] s'appuie sur le graphe de *co-demande* et résout un problème de coloration sous contraintes ;
– [WP05] propose d'énumérer les ordres possibles d'élimination des commandes au lieu d'énumérer les permutations des produits. Cette approche originale se révèle très efficace.

Aucune des approches proposées durant le challenge IJCAI 05 n'a mis en œuvre des techniques rétrospectives (*backtrack* intelligent ou s'appuyant sur un calcul d'explications d'échecs). Nous souhaitons montrer que le MOSP est un bon candidat pour de telles approches parce qu'il s'agit d'un problème structuré. Le nombre minimal de ressources est en effet souvent relié à de petits ensembles de produits.

10.2 Résolution : une approche fondée sur le calcul des explications

Cette section met en œuvre les idées du chapitre 7 dans le cadre du MOSP. Nous présenterons dans un premier temps l'approche d'enregistrements de *nogoods* simples dans un cadre algorithmique précis de manière à pouvoir l'étendre pour intégrer le calcul de *nogoods* généralisés et d'explications (section 10.2.1) . Les idées générales de l'approche seront présentées à l'aide d'exemples concrets (section 10.2.2) avant de passer à une définition formelle et algorithmique de deux approches exploitant des raisonnements de généralisation des échecs (section 10.2.3) .

10.2.1 Enregistrement de *nogoods* simples

L'approche par enregistrement de *nogoods* est simplement basée sur une énumération chronologique des p_i de p_1 à p_m. L'algorithme 25 prend en entrée une séquence partielle de produits et essaye de l'étendre à une séquence complète :

– si la séquence n'est pas encore complète (ligne 1), l'algorithme branche sur chaque produit restant de p_t (l. 2) ;
– une étape de filtrage (l. 7) est appliquée. Ici, seul D_{p+1} est filtré par rapport à la borne supérieure *ub* (la valeur de la meilleure solution trouvée jusqu'à présent) ;
– si aucune contradiction n'est levée, l'énumération continue (appel récursif l. 8) ;
– une fois que la séquence $p_1, \ldots, p_t$ est prouvée infaisable, elle est stockée en mémoire (l. 10) de manière à interdire toutes ses permutations dans la recherche future.
– la ligne 12 enclenche le *backtrack* une fois que le domaine de p_t a été vidé par la recherche ou après la découverte d'une nouvelle solution (l. 11) pour prouver son optimalité.

Notons que [SL05] souligne cette idée tout en optant finalement pour un autre schéma de branchement. Le calcul de bornes inférieures et l'utilisation des règles de dominance sur les produits pourraient être introduits en étendant la fonction $filtre$ appelée ligne 7. Une heuristique sur l'ordre d'énumération s'inscrirait ligne 2.

10.2.2 Idées générales : généraliser les échecs

En introduisant l'enregistrement de *nogoods* au chapitre 7, nous avons souligné la nécessité de calculer des *nogoods* qui contiennent la classe d'affectations la plus large possible de l'espace de recherche futur. Examinons donc comment généraliser les échecs rencontrés par l'approche précédente. À partir d'un *nogood* S, on cherche à concevoir toute une classe de *nogoods* équivalents en exploitant certaines propriétés du problème. Nous avons exploré deux types de généralisation :

– calculer des sous-ensembles de S qui demeurent des *nogoods* en identifiant des conditions d'échecs plus précises ;
– calculer des ensembles de *nogoods* équivalents à S en *expliquant* la preuve réalisée par le solveur pendant la recherche.

procédure : NogoodRecMOSP($\{p_1, \ldots, p_{t-1}\}$) throws Contradiction

```
1: if t − 1 ≠ m do
2:   for all i ∈ D_{p_t}
3:     p_t ← i ;
4:     ∀k > t, retire i de D_{p_k} ;
5:     S ← {p_1, ..., p_t} ;
6:     try
7:       filtre(S,p_{t+1}) ;
8:       NogoodRecMOSP(S) ;
9:     catch (Contradiction c)
10:      ajouter le nogood {p_1, ..., p_t} ;
11: else enregistre la nouvelle solution et met à jour ub ;
12: throw new Contradiction() ;
```

Algorithme 25: Schéma d'enregistrement de *nogood* simple

procédure : filtre($S = \{p_1, \ldots, p_t\}$, p_{t+1}) throws Contradiction

```
for all i ∈ D_{p_{t+1}} do
  if |O^P(S) ∪ C(i)| ≥ ub ou S ∪ {i} est un nogood then
    retire i de D_{p_{t+1}} ;
```

Algorithme 26: Filtrage simple de l'instant t+1

10.2.2.1 Réduire la taille des nogoods

L'idée est de répondre à la question suivante : *dès lors que* $f^{O(S)}(P - S) \geq ub$ *est établie, quelles sont les conditions sur* S *sous lesquelles cette preuve reste correcte ?*

Comme la valeur optimale $f^{O(S)}(P - S)$ dépend de $P - S$ et $O(S)$, retirer un produit de S qui ne fait pas décroître $O(S)$ fournit un nouveau *nogood* valable. L'ajout de ce produit à $P - S$ ne peut en effet que faire croître $f^{O(S)}(P - S)$. On peut donc chercher à calculer des sous-ensembles minimaux de S qui conservent $O(S)$.

Exemple 18 *Calculs de nogoods pour le MOSP*

Examinons le tableau 10.2. $S = \{P_1, P_2, P_3, P_4, P_5\}$ est un *nogood*. P_1, P_2 et P_3 peuvent être retirés sans changer $O(S)$ de telle sorte que $\{P_4, P_5\}$ est aussi un *nogood*. Dans ce cas, les commandes déjà fermées impliquent des produits qui ne sont pas nécessaires à l'échec.

	P_1	P_2	P_3	P_4	P_5	O(S)	...
c_1	1	0	0	1	0	1	
c_2	0	1	1	0	0	0	
c_3	0	0	1	0	1	1	...
c_4	1	1	0	0	0	0	
c_5	0	0	0	1	0	1	
c_6	0	0	0	0	1	1	

TAB. 10.2: Exemple : calculs de *nogoods* pour le MOSP

Le calcul, pendant la recherche, de l'ensemble coupable de commandes (un sous-ensemble de $O(S)$) qui soit suffisant pour valider la preuve $f^{O(S)}(P-S) \geq ub$ relève d'un calcul d'explications (voir la section suivante). Dans l'exemple précédent, en imaginant que les commandes 1, 3 et 6 soient suffisantes pour obtenir le *nogood* $\{P_1, P_2, P_3, P_4, P_5\}$, on pourrait en déduire que $\{P_1, P_5\}$ est également un *nogood*.

10.2.2.2 Calculer des nogoods équivalents

La question centrale devient à présent : *dès lors que $f^{O(S)}(P-S) \geq ub$ est établie, quelles sont les conditions sur $P-S$ sous lesquelles cette preuve reste correcte ?* Peut-on à partir de ces conditions, construire des ensembles plus larges de *nogoods* ?

Ce problème est rattaché aux explications. Au lieu de calculer des conditions S_1 sur S qui peuvent être vues comme les décisions prises jusqu'à présent, on peut calculer des conditions S_2 sur $P-S$ qui peuvent se voir comme des contraintes originelles du problème. Une contradiction sera donc justifiée logiquement par $S_1 \cup S_2$ mais on peut se contenter de conserver S_2 comme explication car S_1 peut toujours être recalculé au moment de l'échec (contrairement à S_2) et disparaît de toute façon par résolution. Une explication sera donc définie de la manière suivante :

Définition 10.2.1 *Soit $S = p_1, \ldots, p_{j-1}$ une séquence de produits et $S' = p_1, \ldots, p_j$ une séquence qui étend S avec $p_j = i$. Une explication du retrait de la valeur i de p_j, $expl(p_j \neq i)$ est définie par un ensemble e, $e \subseteq P-S$ tel que $|O^{S \cup e}(S) \cup C(i)| \geq ub$ ou $f^{O(S')}(e - \{i\}) \geq ub$ (en d'autre termes, le problème réduit à e est infaisable).*

Tous les mécanismes de filtrage doivent être expliqués. Dans le cas très simple de l'algorithme 26, la valeur i peut être éliminée de D_{p_j} si $open_j$ est incompatible avec la borne supérieure courante ub. Une explication est donc seulement un sous-ensemble des produits restants qui conserve ouvertes les commandes ouvertes à l'instant j. Si $S = \{p_0, .., p_{j-1}\}$, $e = expl(p_j \neq i)$ est dans ce cas définie comme :

$$|O^{(S \cup e)}(S) \cup C(i)| \geq ub$$

Comme $open_{j-1}$ est compatible avec ub, une fois que S est prouvé infaisable (à la fois par recherche et par filtrage), $expl(p_{j-1} \neq k) = \bigcup_{v \in D^{orig}(p_j)} expl(p_j \neq v)$.

Exemple 19 *Calculs d'explications pour le MOSP*

Considérons le tableau 10.3. $S = \{P_1, P_2\}$, $P - S = \{P_3, P_4, P_5, P_6, P_7\}$, $O(S) = \{c_2, c_3, c_4\}$. À l'étape 1, la borne supérieure vaut 4 and $p_2 \neq P_2$ à cause de $f^{O(S)}(\{P_3, P_4, P_5, P_6, P_7\}) \geq 4$. C'est cependant toujours vrai tant que $O(S)$ reste inchangé, donc pour $\{P_3, P_5, P_6\}$ ou $\{P_4, P_5, P_6\}$. Toutes les valeurs v de p_3 sont retirées par filtrage et $\{P_4, P_5, P_6\}$ est enregistré pour chaque $expl(p_3 \neq v)$ de telle sorte que $expl(p_2 \neq P_2) = \{P_4, P_5, P_6\}$. À l'étape suivante, l'énumération essaye $p_2 = P_3$ et une explication telle que $\{P_4, P_5, P_6\}$ ou $\{P_2, P_5, P_7\}$ est calculée. Le premier ensemble mène à $expl(p_2 \neq \{P_2, P_3\}) = \{P_4, P_5, P_6\}$ et le processus continue.

En cas de filtrage dû à un *nogood* N, une explication $expl(N)$, a déjà été enregistrée. Une contradiction levée par la borne inférieure doit aussi être expliquée en identifiant un sous-graphe de G' qui déclenche l'échec de $lb(P-S)$. De nombreux ensembles de produits peuvent

	Exemple 1							Exemple 2			
	Étape 1			Étape 2							
	P_1P_2	$O(S)$	$P_3P_4P_5P_6P_7$	P_1P_3	$O(S)$	$P_2P_4P_5P_6P_7$		$P_1P_2P_3$	$O(S)$	P_4...
c_1	0 0	0	1 0 0 1 1	0 1	1	0 0 0 1 1	c_1	1 0 0	1	1
c_2	0 1	1	0 0 0 1 0	0 0	0	1 0 0 1 0	c_2	0 1 1	0	0
c_3	1 0	1	1 1 0 0 0	1 1	1	0 1 0 0 0	c_3	1 0 1	0	0 ...
c_4	1 1	1	0 0 1 0 0	1 0	1	1 0 1 0 0	c_4	0 1 0	1	0
c_5	0 0	0	0 1 1 0 1	0 0	0	0 1 1 0 1	c_5	0 0 1	1	1
							c_6	0 0 0	0	1

TAB. 10.3: Exemple : calculs d'explications pour le MOSP

être à l'origine du même graphe puisque plusieurs produits sont généralement associés à un arc de G'.

À chaque séquence infaisable S, en expliquant la preuve faite sur $P - S$, on peut dans un premier temps incriminer un sous-ensemble de $O^P(S)$ qui permettrait de calculer un sous-ensemble de S plus précis menant à la même contradiction (et donc, un point de *backtrack* plus pertinent). Deuxièmement, l'explication fournit aussi un moyen de généraliser encore ce *nogood* en exploitant les produits qui ne sont pas impliqués dans l'explication.

Exemple 20 *Calculs de nogoods équivalents*

Sur le deuxième exemple de la table 10.3, P_4 peut être échangé avec $\{P_1, P_3\}$ si P_4 n'est pas nécessaire à prouver que $\{P_1, P_2, P_3\}$ est un *nogood*. $\{P_4, P_2\}$ est donc aussi un *nogood*.

Les explications sont reliées à l'idée que l'indépendance et la redondance des produits dans P peuvent être à l'origine de petits sous-ensembles de P qui possèdent la même valeur optimale. Les explications cherchent à tirer parti de ces structures dynamiquement pendant la recherche.

10.2.3 Nogoods généralisés

Un *nogood* traditionnel est défini comme une affectation partielle qui ne peut pas être étendue à une solution (chapitres 3 et 7). Un tel *nogood* devient inutile dès que l'un de ses sous-ensembles devient un *nogood*. Cependant, ce n'est pas vrai dans le cas des *nogoods* présentés ci-dessus pour le MOSP. Le *nogood* $\{P_1, P_3, P_4\}$ est un sous-ensemble de $\{P_1, P_2, P_3, P_4\}$ mais n'interdit pas la séquence $\{P_1, P_3, P_2, P_4\}$ de telle sorte que le *nogood* $\{P_1, P_2, P_3, P_4\}$ reste utile. Un *nogood* pour le MOSP est en effet défini comme une séquence de produits qui interdit de **commencer** la séquence de production par n'importe laquelle de ses **permutations**.

Cette remarque suggère que la taille n'est pas ici un bon critère de pertinence des *nogoods*. On note $\mathcal{P}(S)$, l'ensemble de tous les sous-ensembles de S. Les *nogoods* considérés par notre approche sont définis de la manière suivante :

Définition 10.2.2 *Un nogood généralisé N est défini par une paire d'ensembles disjoints de produits, (R, T) (root et tail), qui interdisent de commencer la séquence de production par une permutation d'un ensemble appartenant à $\{R \cup T_i, T_i \in \mathcal{P}(T)\}$.*

Cette définition fournit un moyen de factoriser un nombre exponentiel de *nogoods* (à travers les ensembles *root* et *tail*). En ce sens, ils sont analogues aux *nogoods* généralisés dans un cadre générique introduits section 7.1.1.

La propriété suivante est utilisée pour caractériser les *nogoods* généralisés valides lorsque l'algorithme est confronté à une séquence de production infaisable.

Propriété 10.2.1 *Si S est une séquence infaisable de produits et $expl(S) \subseteq P - S$ une explication de cette situation. Le nogood généralisé (R, T) tel que,*
- $(R \cup T) \cap expl(S) = \emptyset$,
- $O^{S \cup expl(S)}(S) \subseteq O(R)$,

est un nogood généralisé valide.

Preuve : Comme S est un *nogood*, $f^{O(S)}(P - S) \geq ub$. Cependant, $expl(S)$ est un sous-ensemble de $P - S$ tel que, après avoir affecté de manière chronologique S, le problème restreint à $expl(S)$ est infaisable. On a donc $f^{O^{S \cup expl(S)}(S)}(expl(S)) \geq ub$. Grâce à $O^{S \cup expl(S)}(S) \subseteq O(R)$ et $R \cap expl(S) = \emptyset$, on peut déduire $f^{O(R)}(expl(S)) \geq ub$ de l'inégalité précédente. On remarque ici qu'on peut incriminer un sous-ensemble des commandes ouvertes $O^P(S)$. En effet $O(R) \subseteq O^P(S)$ puisque $S \cup expl(S) \subseteq P$. $f^{O(R)}(expl(S)) \geq ub$ montre donc que $(R, S - R)$ est un *nogood* généralisé valide. Par ailleurs, l'ajout de produits non inclus dans $expl(S)$ dans la queue de $(R, S - R)$ ne peut pas faire décroître $O(R)$. Chaque commande de $O(R)$ est en effet ouverte à cause d'au moins un des produits de $expl(S)$. Ainsi, (R, T) reste un *nogood* tant que $T \cap expl(S) = \emptyset$. □

En pratique, de tels *nogoods* sont obtenus en appliquant sur S, les raisonnements de la section précédente. Les différents algorithmes qui s'appuient sur de tels *nogoods* sont abordés à présent.

10.2.3.1 Enregistrement de *nogoods* généralisés

Pour mettre en œuvre les idées données section 10.2.2.1, les lignes 9-10 de l'algorithme 25 sont modifiées pour introduire le calcul du *nogood* généralisé et le *backjumping*. Le pseudo-code suivant (l'algorithme 27) suppose qu'une contradiction c connaît le *niveau* auquel elle se produit (c.level ligne 10a). En d'autres termes, l'infaisabilité de $p_1, \ldots p_k$ provoquée par le domaine vide de p_{k+1} engendre une contradiction associée au niveau $k+1$ (*throw new Contradiction(k+1)*).

procédure : catch (Contradiction c)

```
10a:  if t < c.level
10b:     R^1 ← minimize(S, O^P(S));
10c:     R^2 ← minimize({p_t, p_{t-1}, ..., p_1}, O^P(S));
10d:     ∀j ∈ {1,2} ajouter les nogoods (R^j, S − R^j);
10e:     newLevel ← argmax_k(p_k ∈ R_1)
10f:     if newLevel < t
10g:        throw new Contradiction(newLevel);
10h:  else if (t > c.level) throw new Contradiction(c.level)
10i:  else throw new Contradiction(c.level)
```

Algorithme 27: Extension de l'algorithme 25 aux lignes 9-11.

La fonction $minimize(S, O)$ calcule S', un sous-ensemble de S, tel que $O \subseteq O(S')$ en s'appuyant sur XPLAIN. De plus, l'ordre de S est utilisé pour guider la génération du sous-ensemble de S. Si $S = p_1, \ldots p_i$, XPLAIN assure que $argmax_k(p_k \in S')$ est minimal[2]. Deux *nogoods*, R^1 et R^2, sont enregistrés à chaque contradiction. L'objet de R^1 est de fournir le meilleur point de *backjump* (puisque le dernier produit de R^1 correspondra au produit placé le

[2]Il n'y a aucune sous-séquence de S avec un produit p_j tel que $j < k$.

plus tôt possible) alors que R^2 est celui qui a le plus de chance d'être minimal (la contradiction peut ne dépendre que des produits ajoutés le plus récemment, S est inversé pour focaliser le sous-ensemble sur les derniers produits ajoutés). Le *backjumping* est assuré lignes 10g,10h en levant immédiatement une contradiction si le niveau de l'échec n'est pas atteint.

10.2.3.2 Enregistrement de nogoods et calcul d'explications

Pour mettre en œuvre les idées données section 10.2.2.2, l'algorithme 28 remplace l'algorithme 26 pour expliquer le filtrage dû à *ub* et aux *nogoods* existants.

procédure : filtre($S = \{p_1, \ldots, p_t\}$, p_{t+1}) throws Contradiction

1: **for all** $i \in D_{p_{t+1}}$ **do**
2: **if** $|O^P(S) \cup C(i)| \geq ub$
3: retire i de $D_{p_{t+1}}$;
4: $expl(p_{t+1} \neq i) \leftarrow e$ tel que $|O^{S \cup e}(S) \cup C(i)| \geq ub$;
5: **else if** $S \cup \{i\}$ est un nogood N ;
6: retire i de $D_{p_{t+1}}$;
7: $expl(p_{t+1} \neq i) \leftarrow expl(N)$;

Algorithme 28: Filtrage expliqué

On suppose également qu'une explication (c.exp) peut également être attachée à une contradiction c en plus du niveau (c.level). L'algorithme 25 est étendu à nouveau au niveau du traitement de la contradiction pour assurer la gestion des explications et mène à l'algorithme 29 :

- une explication de contradiction est calculée (l. 10b) à partir du domaine original de p_{t+1}. Elle est sera utilisée pour expliquer le retrait de la valeur i qui vient d'être essayée pour p_t (l. 10n, 10o) une fois le monde correspondant atteint (les notations i, p_t et S viennent de l'algorithme 25). Quatre *nogoods* au plus sont ensuite enregistrés ;
- R^1 et R^2 sont les mêmes que précédemment à l'exception que $S \cup e$ peut être plus précis que P pour $O^{S \cup e}(S)$ (l. 10c, 10d). Une telle situation est cependant relativement rare sans un filtrage plus fin[3] ;
- R^3 et R^4 sont des généralisations de R^1 et R^2. C'est ici l'apport principal des explications. Cette généralisation intervient lignes 10f, 10g avec l'utilisation de $\overline{e}$ pour construire les racines R^3,R^4. Elle intervient également ligne 10i avec l'expression de la queue : $\{\overline{e} \cup S\} - R^j$. On peut vérifier que les *nogoods* vérifient bien la propriété 10.2.1 énoncée précédemment ;
- enfin, l'explication courante e est attachée au quatre *nogoods* ligne 10j pour pouvoir justifier le filtrage futur engendré par ces *nogoods*.

Une manière générique d'obtenir une explication pour tous les raisonnements précédents (filtrage simple et borne inférieure) est d'appliquer XPLAIN sur les produits jusqu'à l'obtention de la propriété désirée. L'alternative est de concevoir un algorithme glouton qui construit l'explication sans la critiquer. Il s'agit de trouver le bon compromis entre la précision de l'explication et son temps de calcul. Nous avons ici choisi d'utiliser XPLAIN afin d'évaluer avant tout dans quelle mesure le problème peut être expliqué précisément[4].

[3] C'est effectivement possible grâce à la borne inférieure.

[4] La représentation des produits/commandes par des BitSet rend les tests d'inclusion vis-à-vis des commandes ouvertes très rapides et l'appel à la méthode d'inférence de XPLAIN est efficace, il ne s'agit pas d'une étape de propagation coûteuse qui rend en général XPLAIN inapplicable.

procédure : catch (Contradiction c)

```
10a: if t < c.level
10b:     e ← ⋃_{j∈D^orig(p_{t+1})} expl(p_{t+1} ≠ j);
10c:     R^1 ← minimize(S, O^{S∪e}(S));
10d:     R^2 ← minimize({p_t, p_{t-1}, ..., p_1}, O^{S∪e}(S));
10e:     ē ← P − S − e;
10f:     R^3 ← minimize(R^1 ∪ ē, O^{S∪e}(S));
10g:     R^4 ← minimize(R^2 ∪ ē, O^{S∪e}(S));
10h:     for each j ∈ {1, 2, 3, 4}
10i:           on ajoute le nogood N_j = (R^j, {ē ∪ S} − R^j);
10j:           expl(N_j) ← e;
10k:     newLevel ← argmax_k(p_k ∈ R_1)
10l:     if newLevel < t
10m:           throw new Contradiction(newLevel,e);
10n:     else expl(p_t ≠ i) ← e
10o: else if (t = c.level) expl(p_t ≠ i) ← c.exp;
10p: else throw new Contradiction(c.level);
```

Algorithme 29: étend lignes 9-11 de l'algorithme 25.

10.2.3.3 Stockage et filtrage des nogoods généralisés

La gestion efficace des *nogoods* est toujours un problème délicat. Les clauses apprises sont ainsi critiques pour l'efficacité des solveurs SAT mais leur propagation est subtile. Nous renvoyons le lecteur aux sections 3.4.2 et 7.1.2 pour comprendre ces schémas de propagation. Les *nogoods* généralisés définis à la section précédente correspondent à un nombre exponentiel de *nogoods* simples utilisés par la programmation dynamique et il est impossible de les stocker tous individuellement. On dispose donc d'une collection C de *nogoods* généralisés, deux points principaux doivent être analysés :

- vérifier efficacement qu'un ensemble S appartient à C. Un ensemble S est interdit si et seulement si $\exists(R^k, T^k) \in C$ tel que : $R^k \subseteq S \subseteq T^k \cup R^k$;
- ajouter efficacement un nouvel (R^k, T^k).

L'extraction de toutes les racines incluses dans S, telles que leurs queues contiennent S, est basée sur une forme simple d'automate fini déterministe connu sous le nom de TRIE [Knu97]. La forme particulière des *nogoods* nous empêche en effet d'utiliser la représentation canonique sous forme d'automate minimal du chapitre 7. Un TRIE est une structure de données arborescente ordonnée où un ensemble est associé à chaque nœud et déterminé par le chemin menant de la racine à ce nœud. Ainsi, tous les descendants d'un nœud ont un sous-ensemble commun associé à ce nœud. La figure 10.1(a) montre un exemple de TRIE. Identifier et ajouter un ensemble spécifique dans le TRIE s'effectue en $O(m)$ (m est le nombre de produits). En revanche, obtenir tous les sous-ensembles d'un ensemble donné n'est plus polynômial. Le TRIE est utilisé de la manière suivante :

- un nœud du TRIE est associé à une racine d'un *nogood* R^k et contient la liste de toutes les queues T^i telles que (R^k, T^i) est un *nogood* ;
- un nœud I est également marqué par un ensemble Un_I correspondant à l'union des racines et des queues de tous ses descendants.

La vérification d'un ensemble S est faite par une recherche en profondeur d'abord récursive dans le TRIE qui consomme les éléments de S pour atteindre toutes les racines R^k incluses dans S. Pour chaque racine R^k atteinte, on vérifie simplement pour toutes les queues du nœud

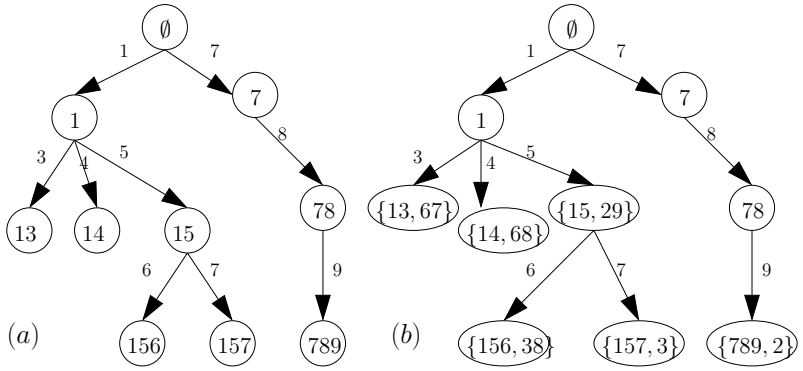


FIG. 10.1: (a) Exemple d'un TRIE. (b) Exemples de *nogoods stockés* à l'aide d'un TRIE (une unique queue est représentée pour chaque racine).

si $T^k \cup R^k \supseteq S$. La recherche est par ailleurs interrompue à tout nœud I dès que $Un_I \subsetneq S$. Par exemple, le TRIE étant ordonné, les éléments 3 et 4 n'apparaîtront jamais dans un descendant de $\{1, 5\}$. Si de tels éléments appartiennent à S, ils doivent donc apparaître dans $Un_{\{1,5\}}$. En pratique, le TRIE permet d'arrêter la recherche assez tôt et n'est pas coûteux à maintenir pour ajouter un *nogood*. Vérifier l'ensemble $\{1, 2, 3, 5\}$ ne demande que de parcourir les nœuds $\{1\}, \{1, 3\}, \{1, 5\}$ sur l'exemple (b) de la figure 10.1.

La propagation très simple effectuée sur les *nogoods* est cependant très coûteuse par rapport à la programmation dynamique qui peut vérifier en temps constant si S est ou non un *nogood*. Une limitation très importante de notre approche actuelle est la vérification incrémentale d'un ensemble et l'élimination des *nogoods* dominés.

10.2.4 Positionnement

L'approche présentée ici relève de l'enregistrement de *nogoods* et du *backjumping*. Elle suit les idées du chapitre 7 en :

- exploitant des propriétés du problème pour extraire à chaque échec des *nogoods généralisés* représentant de manière compacte un nombre exponentiel de *nogoods* classiques ;
- stockant les *nogoods* dans une forme simple d'automate.

La résolution *expliquée* est en revanche utilisée de manière originale pour poursuivre la généralisation des échecs en utilisant tous les éléments du problème qui n'apparaissent pas dans l'explication. L'obtention d'un sous-ensemble des contraintes originelles du problème pour justifier un échec permet de dériver des sous-ensembles équivalents de décisions à celui incriminé comme *nogood*. Naturellement, ce mécanisme repose sur des propriétés particulières du problème. Néanmoins, il semble intéressant de conserver cette possibilité à l'esprit quand le problème présente des formes de symétries partielles qui ne sont valables que pour certaines parties du problème parfois incriminé par l'explication d'échec.

L'approche repose sur la nature de la fonction objectif et l'énumération chronologique. Elle est parfaitement applicable pour des variantes du problème avec d'autre types de contraintes comme des précédences entre tâches, des consommations variables de la ressource, etc. L'approche à base d'explications exige en plus que le filtrage des contraintes introduites soit *expliqué* sur $P - S$.

10.3 Résolution : approches contraintes

L'approche précédente n'exploite pas de raisonnements forts de filtrage et se concentre entièrement sur les échecs. Pour combler ce manque, différents modèle contraintes du problème utilisés pendant le challenge sont présentés dans cette section et un défaut majeur de ces modèles est mis en lumière. Nous présentons ensuite une extension de la contrainte *cumulative*, intitulé *cumulative convexe* pour remédier à cette lacune.

10.3.1 Modèles

10.3.1.1 Modèle simple

Le modèle le plus intuitif, introduit dans [SG05] comme le modèle de base, s'appuie sur les variables p_i indiquant le produit affecté à chaque instant i. La contrainte (1) reflète le fait qu'un seul et unique produit est traité à chaque instant. Les instants d'ouverture et de fermeture d'une commande sont notés st_i et end_i et s'expriment simplement comme le $\min_{j \in P(i)} p_j$ et le $\max_{j \in P(i)} p_j$ (contraintes (2) et (3)). On peut représenter par une matrice Y de variables booléennes y_{ij} le fait qu'une commande i s'exécute ou non à l'instant j et contraindre ainsi y_{ij} avec st_i et end_i pour obtenir la plage d'exécution d'une commande (contrainte (4)). Le nombre de commandes ouvertes simultanément à chaque instant k correspond alors à la somme sur i des y_{ik} (5) et l'objectif est de minimiser le nombre maximum de commandes ouvertes simultanément (6).

Variables		Contraintes	
$p_i = [1, \ldots, m]$,	$\forall i \leq m$	(1) $alldifferent(\{p_1, \ldots, p_m\})$	
$st_i = [1..m]$,	$\forall i \leq n$	(2) $st_i = \min_{j \in P(i)} p_j$	$\forall j \leq n$
$end_i = [1..m]$,	$\forall i \leq n$	(3) $end_i = \max_{j \in P(i)} p_j$	$\forall j \leq n$
$y_{ij} = \{0, 1\}$,	$\forall i \leq n, j \leq m$	(4) $y_{ij} = (st_i \leq j) \wedge (j \leq end_i)$	$\forall i \leq n, j \leq m$
$open_j = [1, \ldots, n]$,		(5) $open_j = \sum_{i \leq m} y_{ij}$,	$\forall j \leq m$
		(6) $objectif = max_{j \leq m} open_j$	

TAB. 10.4: Modèle simple pour le MOSP

Enfin, un modèle dual ou symétrique est fréquemment introduit. Il s'appuie sur les variables $psym_i = [1, \ldots m]$ (symétriques de p) indiquant pour chaque produit sa position dans la séquence. Ce modèle simple possède de nombreux défauts de propagation qui vont être mis en lumière sur les différents modèles redondants suivants.

10.3.1.2 Modèle ordonnancement

On peut également aborder ce problème sous l'angle de l'ordonnancement [Bel05]. On considère ainsi les commandes comme des tâches de hauteur 1 et on cherche à minimiser la capacité ou la hauteur h de la ressource nécessaire à l'exécution des tâches. On note t_i la tâche constituée d'une date de départ st_i, d'une durée d_i et d'une date de fin end_i correspondant à la commande i. Le modèle s'écrit :

[SL05] évoque aussi la possibilité de tirer parti des relations de précédences implicite qui s'établissent entre les tâches pour respecter l'objectif à travers une contrainte de *balance* [Lab03].

Les avantages de ce modèle résident dans sa compacité et les raisonnements énergétiques effectués par la *cumulative*. [Bel05] souligne que $min_{j \in P(i)} p_j$ peut être amélioré en prenant

Variables		Contraintes	
$p_i = [1, \ldots, m]$,	$\forall i < m$	(1) $alldifferent(\{p_1, \ldots, p_n\})$	
$st_i = [1, \ldots, m]$,	$\forall i \leq n$	(2) $cumulative(\{t_i, \ldots, t_n\}, H)$	
$end_i = [1, \ldots, m]$,	$\forall i \leq n$	(3) $st_i = min_{j \in P(i)} p_j$	$\forall i \leq n$
$d_i = [\|P(i)\|, \ldots, m]$	$\forall i \leq n$	(4) $end_i = max_{j \in P(i)} p_j + 1$	$\forall i \leq n$
		(5) $objectif = minimiser(H)$	

TAB. 10.5: Modèle ordonnancement pour le MOSP

en compte le fait que les variables p_j doivent prendre des valeurs distinctes. Nous avons travaillé sur la formulation alternative intitulée modèle convexe, pour mieux capturer ces plages d'exécutions.

10.3.1.3 Modèle convexe

On introduit dans le modèle simple une matrice, X, des variables booléennes supplémentaires. Ainsi, x_{ij} exprime le fait qu'un produit de la commande i soit utilisé à l'instant j. Les contraintes (2),(3) et (4) du modèle simple sont remplacées par les contraintes (2) et (3) du modèle convexe. La contrainte (2) relie les variables x_{ij} au modèle entier sur les produits. Par ailleurs, chaque ligne de Y est désormais maintenue comme l'enveloppe convexe de la ligne de X correspondante (3).

$x_{ij} = 0, 1,$	$\forall i \leq n, j \leq m$	(2) $element(C[i], p_j, x_{ij})$,	$\forall i \leq n, j \leq m$
$y_{ij} = 0, 1,$	$\forall i \leq n, j \leq m$	(3) $enveloppeConvexe(X[i], Y[i], \|P(i)\|)$,	$\forall i \leq n$

TAB. 10.6: Modèle convexe pour le MOSP

La contrainte enveloppe convexe prend en entrée deux vecteurs de variables booléennes U et V ainsi qu'un entier k indiquant le nombre d'occurrences de la valeur 1 dans U. Elle assure que V est l'enveloppe convexe de U. En notant f la position de la première variable de U prenant la valeur 1 et l la dernière, la contrainte assure que $\forall i \in [f, l], V[i] = 1$ et $V[i] = 0$ sinon. Elle exprime exactement la plage d'exécution de la commande.

Exemple 21 *Parties obligatoires établies par l'enveloppe convexe*

L'apport de l'enveloppe convexe se situe au niveau de la propagation de parties obligatoires et des plages interdites pour chaque commande. Prenons le cas des parties obligatoires. Soit $m = 13$ et une commande i comprenant cinq produits ($|P(i)| = 5$) qui ne peuvent pas être placés aux instants 2,4,9,11 et 12. Les vecteurs X[i] et Y[i] correspondant sont donnés ici :

	1	2	3	4	5	6	7	8	9	10	11	12	13
X =	-	0	-	0	-	-	-	-	0	-	0	0	-
Y =	-	-	-	-	-	1	1	-	-	-	-	-	-

L'intervalle [6,7] est la partie obligatoire de i maintenue par la contrainte.

Ce modèle convexe calcule des parties obligatoires plus précises que la cumulative précédente qui ne prend pas en compte les instants où les produits de la commande ne peuvent pas s'insérer et se contente de bornes sur les dates de départs au plus tard et dates de fin au plus tôt. Un dernier modèle intègre le souci d'une borne inférieure plus globale à partir de la coloration du graphe de *co-commande*.

10.3.1.4 Modèle de coloration

Le graphe de *co-demande* G introduit section 10.1.2.2 fournit des bornes inférieures très pertinentes. On peut donc imposer un modèle redondant de coloration pour maintenir dynamiquement une borne du nombre de *stacks*. C'est l'approche suivie par [SL05, Pes05]. On associe simplement une variable $c_i = [1, \ldots, m]$ pour représenter la couleur de la commande i.

$$c_i = [1, \ldots, m], \quad \forall i \leq n \;\Big|\; \begin{array}{l} (1)\ st_i \leq end_j \wedge st_j \leq end_i \Rightarrow c_i \neq c_j \quad \forall i, j \leq n \\ (2)\ objectif \geq max_{1 \leq i \leq m} c_i \end{array}$$

TAB. 10.7: Modèle de coloration pour le MOSP

10.3.2 Parties obligatoires fines, vers une cumulative convexe

La propagation efficace du nombre de commandes actives passe par celle des plages d'exécution obligatoires et interdites des commandes. Reprenons l'exemple 21.

Exemple 22 *Parties obligatoires fines*

Dans l'exemple 21, la commande i est constituée de 5 produits et 5 variables parmi $\{x_{i1}, \ldots x_{im}\}$ doivent prendre la valeur 1. Ces cinq points d'exécutions obligatoires sont définis par les variables domaines $psym_k$ pour tout k de $P(i)$. La prise en compte de ces domaines permet d'affiner davantage les parties obligatoires. Supposons qu $P(i) = \{1, 3, 5, 6, 9\}$ avec $D_{psym_1} = \{1, 3, 5, \ldots, 8\}$, $D_{psym_3} = D_{psym_5} = D_{psym_9} = \{5, \ldots, 8\}$ et $D_{psym_6} = \{5, \ldots, 8, 10, 13\}$. On se rend compte que l'intervalle [5,8] plus précis que le [6,7] précédent constitue une plage d'exécution obligatoire. En effet, les positions 1 et 3 (resp. 10 et 13) ne peuvent être occupées que par $psym_1$ (resp. $psym_6$).

Si la *cumulative* estime mal ces plages obligatoires, c'est qu'une tâche n'est pas définie ici par la donnée classique des variables de départ/fin/durée.

Définition 10.3.1 *Une tâche convexe T est définie par un ensemble de variables distinctes $P_1, \ldots, P_k$ exprimant des points d'exécution de la tâche. La plus petite et la plus grande valeur de $P_1, \ldots, P_k$ définissent l'intervalle d'exécution de T.*

Définition 10.3.2 *Étant donné une tâche convexe T dont le domaine d'exécution est caractérisé par l'enveloppe convexe d'un ensemble de points $P_1, \ldots, P_k$. On appelle **partie obligatoire** de T, l'intervalle $[s_T, e_T]$ avec :*

- *s_T, le plus grand entier tel que toutes les variables $P_1, \ldots P_k$ puissent prendre des valeurs distinctes supérieures ou égale à s_T.*
- *e_T, le plus petit entier tel que toutes les variables $P_1, \ldots P_k$ puissent prendre des valeurs distinctes inférieures ou égales à e_T.*

s_T est appelée date de début au plus tard de T et e_T, la date de fin au plus tôt de T.

On considère un graphe biparti $G = (A, B, E)$ où A représente l'ensemble des points $P_1, \ldots, P_k$, B, l'union des domaines de $P_1, \ldots, P_k$ et E l'ensemble des arêtes (i, j) tel que $j \in P_i$. C'est le graphe variables/valeurs classique des contraintes s'appuyant sur des flots ou couplages. Soit $K \subset E$ un couplage maximum de G. On note max_K (resp. min_K) le nœud d'index le plus élevé (resp. le plus petit) dans B saturé par K. Le calcul de la date de départ

au plus tard (resp. fin au plus tôt) est fait en cherchant un couplage ou max_K est minimal (resp. min_K maximal).

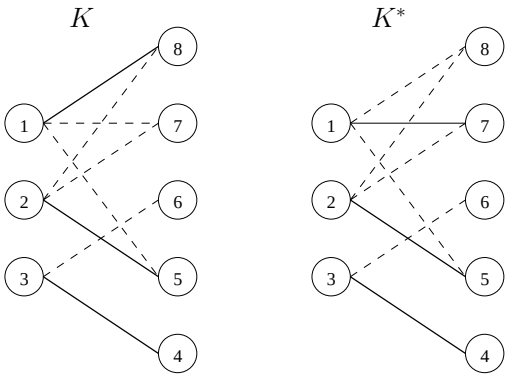


FIG. 10.2: Exemple d'un couplage maximal et d'un couplage maximal *optimal* avec max_{K^*}=7.

Considérons en détail le cas de max_K (l'autre cas étant symétrique). On note K^* un tel couplage maximum **optimal** avec un max_{K^*} minimal (voir figure 10.2). Par ailleurs, une chaîne alternante améliorante de K est une chaîne impaire qui permet de réduire max_K (en échangeant les arcs le long de cette chaîne). Le nouveau couplage conserve sa taille puisque la chaîne est impaire.

Propriété 10.3.1 *Il existe toujours un ensemble disjoint de chaînes alternantes L dans G qui permet de passer de K à K^*.*

Preuve : On considère G' le graphe induit par l'ensemble d'arcs suivant : $U = \{(i,j), (i,j) \in K/K^* \wedge (i,j) \in K^*/K\}$. G' constitue la différence symétrique $K\Delta K^*$. On peut construire L à partir de G' en commençant itérativement par le nœud d'index le plus élevé qui n'a pas encore été ajouté à une chaîne alternante de L (initialement L est donc vide et max_K est le nœud de plus grand index), et en suivant alternativement un arc de K et un arc de K^*. Toutes les chaînes sont disjointes car le degré de G' est au plus de 2 (en effet pour chaque nœud il y a au plus un arc de K et de K^* dans la différence symétrique). □

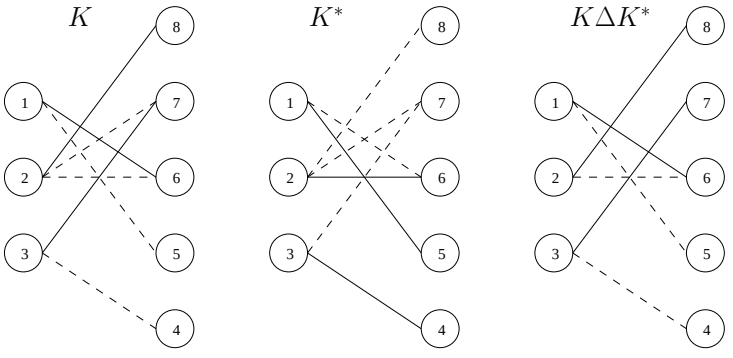


FIG. 10.3: Couplage maximal, couplage maximal *optimal* et leur différence symétrique

Exemple 23 *chaînes alternantes pour passer de K à K^**

On observe sur la figure 10.2, l'ensemble $L = \{(8, 1, 5, 2, 7)\}$ et sur la figure 10.3, $L = \{(8, 2, 6, 1, 5), (7, 3, 4)\}$.

On sait donc, d'après la propriété 10.3.1 qu'un couplage maximal K possède un max_K minimal s'il n'y a pas de chaîne alternante améliorante commençant depuis max_K. Une procédure simple peut être appliquée pour calculer K^* à partir de n'importe quel couplage maximum K de G. On recherche une chaîne alternante par une exploration en largeur depuis max_K itérativement tant qu'une telle chaîne existe. Si aucune ne peut être identifiée, K est optimal. Une telle procédure coûte $O(nm)$. Cette approche n'atteint pas en revanche la complexité $O(\sqrt{n}m)$ de l'algorithme de Tarjan pour le couplage maximum.

Vers une cumulative convexe.

La contrainte *cumulative convexe* semble dès lors une extension naturelle d'une contrainte *cumulative* portant sur des tâches convexes. Un algorithme de filtrage simple consiste à :

1. calculer les parties obligatoires $[s_T, e_T]$ de chaque tâche T par la résolution de couplages optimaux et construire le profil cumulatif associé ;
2. considérer les intervalles $[i, j]$ où l'exécution d'une tâche aux points i et j entraînerait le dépassement de la capacité de la *cumulative* (i et j sont des pics de consommation). On peut éliminer du domaine de $P_1, \ldots P_k$ les valeurs de $[i, j]$ s'il n'existe pas de couplage maximum compris entre i et j. Autrement dit, si les P_k ne peuvent pas prendre toutes des valeurs distinctes entre i et j.

Les raisonnements énergétiques qui viennent souvent compléter le filtrage de la *cumulative* s'appuient sur les parties obligatoires et sont donc tout à fait valides dans ce contexte. Le calcul des parties obligatoires et l'interdiction pour une tâche de couper un intervalle de temps, qui sont les deux éléments fondamentaux d'une contrainte *cumulative*, ont été ici étendus au cas convexe.

10.4 Évaluation expérimentale

Nos expérimentations sont réalisées sur les instances du challenge[5] sur un portable Pentium M cadencé à 1.7Ghz avec 1Go de RAM. Les algorithmes ont été implémentés en Java sur la base du solveur de contraintes choco (`choco-solver.net`)[6].

10.4.1 Bornes inférieures

Les bornes inférieures du tableau 10.8 correspondent à $max_{p \in P}|C(p)|$ pour « naiveLb », au degré minimal du graphe de co-demande plus un pour « mindeg + 1 », la taille de la clique maximum dans ce même graphe pour « clique »et la borne par contraction d'arcs décrite section 10.1.2.2 pour « HAC ». Ces bornes sont très pertinentes dans la mesure où 4132 bornes inférieures sont égales à la valeur optimale sur 5802 instances pour la borne par contraction d'arcs HAC. Ce résultat provient de notre heuristique spécifique de contraction des arcs qui sélectionne les arcs dont la somme des degrés de leurs extrémités est minimale de manière à produire un mineur de G égal à une clique le plus tôt possible.

[5] Disponibles sur *http ://www.dcs.st-and.ac.uk/~ipg/ challenge/*

[6] Une implémentation directe permettrait certainement d'avoir de bien meilleures performances mais ce choix nous permet de coupler facilement cette approche avec de la propagation.

naiveLb	mindeg + 1	clique	HAC
199	2359	2716	4132

TAB. 10.8: Nombres de bornes inférieures égales à la valeur optimale sur 5802 instances pour chacune des bornes

10.4.2 Pertinence des explications

Nous analysons dans un premier temps la pertinence des explications en cherchant une explication de la valeur optimale du problème. L'explication fournit ainsi un sous-ensemble de produits suffisants pour expliquer cette valeur (le problème réduit à cet ensemble possède donc la même valeur optimale). À des fins de comparaison, le problème est également résolu de manière répétée dans le cadre d'une approche de type XPLAIN. Une telle approche fournit au moins un ensemble minimal au sens de l'inclusion. Le pourcentage de réduction est donné au tableau 10.9 pour une sélection d'instances (2985 instances sur les 5802) de l'ensemble du jeu de tests originel[7]. Alors que XPLAINest capable de calculer des explications plus petites (36.4% de produits éliminés au lieu de 22.6 % en moyenne), il est 40 fois plus lent[8] (8.8 secondes contre 0.3 secondes en moyenne) et reste incapable de traiter des instances plus larges telles que les 30_30. Par ailleurs, on peut souvent rapidement améliorer l'explication finale en résolvant une fois de plus le problème réduit à son explication. On peut noter que les instances de tailles n_m avec $m > n$ peuvent être de plus en plus réduites quand m augmente (l'inverse est aussi vrai).

		%reduc	
	NbInst	Explication	XPLAIN
wbo10_10	40	20,8	41,8
wbo10_20	40	29,8	48,4
wbo10_30	40	38,6	53,1
simonis15_30	120	53,7	62,5
wbo20_10	40	2,5	13
wbo30_10	40	0,5	6,3
wbo20_20	90	15,4	39
wbop20_20	90	13,1	28,6
wbp20_20	90	27,8	41,7
simonis20_20	220	39	51,8
testset	2985	22,6	36,4

TAB. 10.9: Pourcentage de réduction obtenu pour l'approche à base d'explications et l'approche XPLAIN.

La réduction est assez impressionnante dans certains cas montrant effectivement que les explications sont capables de capturer certaines structures du problème basées sur la redondance et l'indépendance de produits.

10.4.3 Résolution

Les résultats sont donnés pour trois approches :

[7]Les instances plus grandes, de tailles supérieures à 30_30 ont été exclues à cause des temps de calculs de XPLAIN.

[8]De plus, les bornes intiales n'ont pas été expliquées (à l'exception de $max_{p \in P}|C(p)|$ qui est expliqué avec le produit correspondant p) donc l'explication est entièrement produite par recherche. XPLAINà l'avantage de tirer parti des bornes inférieures.

- NR : le schéma d'enregistrement de *nogoods* simple équivalent à la programmation dynamique ;
- GNR : notre schéma d'enregistrement de nogoods généralisés couplé à du *backjumping* ;
- EXP : GNR augmenté avec le calcul complet des explications.

Les petites instances de taille inférieure à 15_30 telles que les séries de 20_20 sont résolues en moins d'une seconde dans le pire des cas par les trois algorithmes. Les instances sont résolues optimalement à l'exception des trois dernières instances $SP2, SP3, SP4$ qui restent ouvertes pour toutes les approches du challenge.

		NR			GNR			EXP		
Inst	OptM	TAvg (s)	NAvg	BkAvg	TAvg	NAvg	BkAvg	TAvg	NAvg	BkAvg
wbo15_30	11,58	0,1	304	4596	0,1	259	1750	0,2	228	1121
wbo30_30	22,56	4,1	10944	125732	2,6	8351	47467	16,3	7261	35858
wbop15_30	12,15	0,1	249	4114	0,1	241	1884	0,3	231	1439
wbop30_30	23,84	4,3	11319	132295	3	9589	62332	28,4	9043	52751
wbp30_30	24,46	4,6	12375	145366	3	9661	53151	24,8	8730	42694
simonis30_30	28,32	2,7	6400	82885	1,7	4904	25089	9,3	4047	17682
simonis40_20	36,38	0,2	1133	6728	0,1	821	3824	2	764	3534
nwrsLarger4	12,5	0,2	320	5447	0,4	315	2406	1	298	1683
gp50by50	38,75	0	49	20	0	49	19	0	49	15
gp100by100	76,25	2,4	1002	17065	1,6	732	3943	17,8	634	1765

TAB. 10.10: Moyennes des temps, nœuds et *backtracks* sur les instances difficiles du challenge

	NR			GNR			EXP		
Inst	Tmax (s)	NMax	BkMax	Tmax	NMax	BkMax	Tmax	NMax	BkMax
wbo15_30	1,2	2533	42493	1,2	2079	13286	1,2	1705	7737
wbo30_30	31,8	89601	1024219	19,1	64791	314141	103	54028	233466
wbop15_30	1,3	2415	44178	1,4	2317	16202	2,2	2150	10687
wbop30_30	35,9	105617	1053597	23,2	82491	435781	162	76188	374592
wbp30_30	64,1	168259	1994950	37,5	122575	538753	261	103699	420546
simonis30_30	27,5	77593	886794	14,2	50133	208991	70	39138	157517
simonis40_20	1	6196	37385	0,7	3847	16458	8,4	3587	15893
nwrsLarger4	0,9	1230	21788	1,7	1208	9624	4,2	1139	6734
gp50by50	0	101	47	0	101	46	0,1	101	35
gp100by100	8,1	3357	65240	5	2283	14186	56	1897	6143

TAB. 10.11: Valeurs maximales des temps, nœuds, *backtracks* sur les instances difficiles du challenge

Les mesures moyennes (table 10.10) et maximales (table 10.11) du temps (en secondes) et l'espace de recherche (nombre de nœuds/points de choix ainsi que les *backtracks*) sont indiqués pour les plus dures instances du challenge (à l'exception de la série SP4). Ces résultats montrent clairement que le *backjumping* couplé à l'enregistrement de *nogoods généralisés* est une amélioration des approches précédentes d'enregistrement de *nogoods* ou programmation dynamique. L'espace de recherche est significativement réduit (en moyenne par $61,5\%$ et jusqu'à 70%) et l'algorithme est 32% plus rapide en moyenne et 39% sur les instances les plus difficiles. Nous avons néanmoins eu du mal à comparer nos résultats avec [GS05]. Le nombre de nœuds de l'approche d'enregistrement de *nogoods* simples est significativement plus petit et le nombre de *backtracks* plus grand que la mesure de l'effort de recherche indiqué dans [GS05] pour la programmation dynamique et donné comme nombre de nœuds. La programmation dynamique effectue néanmoins une recherche en arrière (*backward search*) contrairement à l'enregistrement de *nogoods* qui procède en avant (*forward search*).

Cependant, il semble que l'utilisation des explications ne soit pas rentable. L'espace de recherche est réduit à nouveau (une réduction supplémentaire de 22%), confirmant la pertinence des explications analysée sur les petits problèmes. Cette technique reste néanmoins compétitive avec les meilleures approches par PPC. Ce résultat est assez intéressant dans la mesure où l'algorithme est capable de fournir en plus une explication de la solution optimale du problème qui peut se révéler très intéressante pour un utilisateur final en mettant en lumière les sous-ensembles critiques de produits responsables du nombre minimum de ressources à mobiliser.

10.5 Conclusion

Nous avons examiné sur le problème du MOSP, comment les raisonnements rétrospectifs basés sur les explications pouvaient être utilisés pour filtrer l'espace de recherche. Nous avons ainsi illustré le chapitre 7 en concentrant notre effort sur les conditions de généralisation des échecs et proposé la définition d'un *nogood* généralisé pour ce problème. Les résultats expérimentaux démontrent l'intérêt d'une telle approche pour le MOSP. Le *backjumping* couplé à l'enregistrement de *nogoods généralisés* est clairement une amélioration de l'enregistrement de *nogoods* simples. Nous pensons donc que l'approche par programmation dynamique pourrait bénéficier des idées présentées dans cette section. Il existe de nombreuses voies d'amélioration de EXP qui pourraient éventuellement mener à un gain en temps. La structure de données courante de stockage des *nogoods* est un élément critique qui pourrait être considérablement amélioré en autorisant une propagation incrémentale. L'utilisation de techniques de *watched literals* pourrait aussi se révéler nettement plus avantageuse. Nous avons dans un premier temps analysé la pertinence des explications pour ce problème et ouvert de nombreuses perspectives sur ce sujet. Nous envisageons en premier lieu l'ajout sur GNR de filtrage plus fort et notamment un calcul des parties obligatoires à partir de la cumulative convexe qui permettrait peut-être d'anticiper des contradictions.

Chapitre 11

Conclusion

Dans ce mémoire, nous nous sommes penchés sur l'intérêt des explications pour améliorer la résolution dans les solveurs de contraintes. Nos résultats montrent que les explications sont sous-exploitées à l'heure actuelle et qu'elles offrent de nombreuses perspectives qui ne passent pas forcément par le *backtrack intelligent* pour la résolution elle-même. Ces travaux ont été menés avec le souci de mettre en œuvre les explications dans des techniques efficaces de résolution sur des applications industrielles et académiques.

Contributions

L'apport des explications a été étudié selon trois grands axes et nous avons montré que :

1. Le réseau d'explications constitue une information particulièrement pertinente pour révéler à un utilisateur les structures ou relations entretenues par différents éléments du problème. Cette information, également exploitable dynamiquement par le solveur est un pas supplémentaire vers des approches de résolution génériques. Nous avons ainsi introduit une notion d'impact raffinant celle présentée par Refalo et focalisée sur les phénomènes de *thrashing*. Cette notion, appliquée aux heuristiques de recherche, s'est révélée très compétitive sur notre application d'ordonnancement temps-réel tout en étant sur d'autres benchmarks nettement moins robuste que les impacts de Réfalo.
2. Les explications ont été utilisées dans un cadre rétrospectif et pourraient l'être davantage dans un cadre prospectif, pour l'énumération (point précédent) comme pour la propagation à l'image de leur exploitation dans la communauté SAT. Nous avons proposé ici de faire face aux besoins mémoire exponentiels des *nogoods* par une compilation dynamique dans un automate. Les résultats obtenus dans un contexte générique sont décevants mais la mise en œuvre des *nogoods* sur le problème du MOSP a conduit à une technique originale de résolution. L'application met notamment en lumière le rôle des explications comme mécanisme de généralisation des échecs.
3. Les explications ont un rôle à jouer au cœur de techniques génériques de décomposition (analogue à la décomposition de Benders) et permettent la coopération de techniques de résolution variées. C'est certainement l'argument le plus fort en faveur du maintien des explications dans les solveurs. De nombreuses applications demandent l'accès à une information permettant de suivre les raisonnements mis en œuvre par les algorithmes de filtrage. Les explications constituent, à ce titre, une information duale très pertinente. La mise au point d'un tel algorithme dans le cadre d'un problème d'ordonnancement temps réel a montré la souplesse de la technique pour la coopération de techniques analytiques avec une résolution par *backjumping*.

Ces axes ont été principalement étudiés dans le cadre de deux applications. Ces études constituent des contributions originales sur ces deux problèmes. De plus, des approches comparatives ont toujours été envisagées en détail sur ces applications. Les travaux présentés sur ces approches sont parfois novateurs (comme la *cumulative convexe*) au même titre que les approches fondées sur les explications. De nombreux benchmarks ont été menés pour établir des comparaisons aussi pertinentes que possibles. Ces résultats montrent que les explications peuvent et doivent constituer une technique de choix pour la résolution et sont tout à fait capables de rivaliser en performance avec des techniques bien reconnues.

Perspectives

Sur le sujet des impacts : La notion d'impact basé sur les explications mérite d'être approfondie. Avec le recul, nous pensons qu'une étude plus théorique des impacts pourrait peut-être aboutir à une caractérisation précise du *thrashing* ou, d'une manière plus générale, les mauvais points de choix initiaux. Différents modèles d'évolution de l'impact d'une décision dans le temps pourraient peut-être être établis et permettre une modélisation précise des comportements erratiques du solveur. De tels modèles fourniraient ainsi des indicateurs plus pertinents pour détecter les mauvais points de choix. Cette étude permettrait par ailleurs de quantifier de manière plus théorique l'information contenue dans les explications pour la résolution. C'est en revanche une étude ambitieuse qui exigerait des expérimentations approfondies sur de nombreuses classes de problèmes. Notre première perspective passe donc par une définition exacte du *thrashing* et sa caractérisation théorique basée sur l'impact.

D'autres utilisations des impacts sont possibles. Dans un contexte d'interaction avec un utilisateur, les impacts peuvent quantifier le degré d'implication des contraintes qui apparaissent dans l'explication finale. L'explication devient alors un ensemble de contraintes pondérées qui permet de détecter les contraintes qui ont été incluses un peu arbitrairement (du fait de l'ordre arbitraire de la propagation) de celles qui sont absolument critiques à la preuve. La propagation elle-même pourrait être ordonnée selon ces poids pour essayer de construire des preuves de manière heuristique qui impliquent le moins de contraintes possibles. Il s'agit donc ici d'obtenir des explications précises de manière heuristique en s'appuyant sur les impacts.

L'exploitation du graphe d'impact est également une piste intéressante d'un point de vue utilisateur. Nous aurions souhaité mettre en place un algorithme de *clustering* pour identifier les groupes de variables fortement corrélés sur des problèmes industriels. De telles structures peuvent par exemple donner l'idée de stratégies de décomposition à un utilisateur.

Sur le sujet de la décomposition : Une deuxième perspective prometteuse réside dans une implémentation plus poussée du cadre de décomposition et son application (dans sa forme générique) à des problèmes d'ordonnancement. Nous avons présenté les résultats de la décomposition dans le cadre d'un problème d'allocation de tâches temps-réels. Peu d'expérimentations ont été menées sur le cadre générique car son implémentation est encore à l'état de prototype. Il semble très intéressant de finaliser cette implémentation et comparer la décomposition aux techniques d'enregistrements de *nogoods* classiques dans un cadre d'optimisation. Son application sur des problèmes d'ordonnancement devrait permettre également d'évaluer si la démarche est vraiment pertinente dans une forme générique. L'intérêt de l'ordonnancement est de fournir de nombreux raisonnements de filtrage très efficaces et donc des explications précises.

Sur le sujet des *nogoods* : Les perspectives qui existent sur l'enregistrement de *nogoods* sont innombrables. Les résultats en SAT pourraient s'avérer très fructueux dans le contexte PPC. Si certains travaux ont commencé à exploiter ces résultats, de nombreuses recherches, particulièrement sur leur manière de calculer des explications restent à faire. Le *backjumping*

original mis en œuvre dans ces solveurs serait de notre point de vue un bon point de départ. L'identification de *nogoods* basés sur la notion de point d'implication unique, la manière efficace de maintenir et de calculer ce graphe d'implication en PPC, comme l'utilisation de *restart* combiné à l'enregistrement de *nogoods* sont autant de pistes qui méritent d'être explorées à notre avis.

Enfin sur le problème du MOSP, l'étude expérimentale de la contrainte cumulative convexe et son couplage avec l'approche d'enregistrement de *nogoods* semble intéressante. Nous pensons également que l'emploi de techniques de *watched literals* pour propager les *nogoods* permettrait d'améliorer considérablement l'efficacité et de se passer du TRIE trop coûteux à l'heure actuelle.

Annexe A

Résolution de $x = \sum_{i=1}^{k} \left\lceil \frac{x}{a_i} \right\rceil b_i + c$

$$x = \sum_{i=1}^{k} \left\lceil \frac{x}{a_i} \right\rceil b_i + c \tag{A.1}$$

avec $x \in \mathbb{N}$ et les constantes $a_i, b_i, c \in \mathbb{N}^*$.

L'objectif est d'identifier le premier point fixe de la fonction $f(x) = \sum_{i=1}^{k} \left\lceil \frac{x}{a_i} \right\rceil b_i + c$, noté x^* et tel que $f(x^*) = x^*$. On peut déjà remarquer que f est une fonction croissante puisque pour $x_1 \leq x_2$ on a bien $f(x_1) \leq f(x_2)$. En effet $\lceil x \rceil \geq x$ et $a_i, b_i, c \in N^*$.

Lemme 1 *$f(x) = x$ admet une solution si et seulement si $\sum_i \frac{b_i}{a_i} < 1$*

Preuve : f étant une fonction croissante en escalier avec à l'origine $f(0) = c > 0$, $f(x) = x$ admet une solution si et seulement si elle est bornée par une droite de coefficient inférieur à 1. Existe-t-il $g(x) = k \times x + C$ avec $k < 1$ telle que $f(x) \leq g(x)$? On a

$$f(x) \leq \sum_i (\frac{x}{a_i} + 1) \times b_i + c$$

$$f(x) \leq \sum_i \frac{b_i}{a_i} x + \sum_i b_i + c$$

En posant $k = \sum_i \frac{b_i}{a_i}$ et $C = \sum_i b_i + c$ on obtient une condition nécessaire et suffisante de l'existence d'une solution avec $k < 1$ soit $\sum_i \frac{b_i}{a_i} < 1$. □

Lemme 2 *L'algorithme du point fixe :*

1: $x \leftarrow c$;
2: ***while*** $x \neq f(x)$ ***do***
3: $\quad x \leftarrow f(x)$;
4: ***end while***
5: ***return*** x ;

converge vers x^* et calcule une solution de l'équation A.1.

Preuve : On se rend compte simplement *qu'on ne peut pas dépasser* x^* et que le nombre d'itérations est fini. On note x_i la valeur de x à l'itération i de l'algorithme du point fixe, de sorte que $x_{i+1} = f(x_i)$.

1. On peut montrer par récurrence qu'à toute itération i, x_i est toujours inférieure à x^*. Au pas zéro, on a toujours $c \leq x^*$ puisque $\forall x, \sum_{i=1}^{k} \left\lceil \frac{x}{a_i} \right\rceil b_i \geq 0$. Or puisque la fonction est croissante, on a $f(c) \leq f(x^*)$ donc $f(c) \leq x^*$ puisque $f(x^*) = x^*$. Ainsi, si on suppose à l'itération i, $x_i \leq x^*$ on obtient $x_{i+1} = f(x_i) \leq f(x^*) = x^*$.

2. Par ailleurs, le nombre d'itération est fini car on a $\forall x < x^*$, $f(x) > x$. On peut de nouveau le montrer par récurrence. Au pas zéro, $f(x_0) = f(0) = c > 0$. On suppose à l'étape i que $x_i < x^*$ et $f(x_i) > x_i$. À l'étape $i+1$, si $x_{i+1} < x^*$, on a

$$\begin{aligned} f(x_{i+1}) &= \sum_{i=1}^{k} \left\lceil \frac{x_{i+1}}{a_i} \right\rceil b_i + c \\ &= \sum_{i=1}^{k} \left\lceil \frac{f(x_i)}{a_i} \right\rceil b_i + c \\ &> \sum_{i=1}^{k} \left\lceil \frac{x_i}{a_i} \right\rceil b_i + c = f(x_i) = x_{i+1} \end{aligned}$$

On a bien $f(x_{i+1}) > x_{i+1}$.

Ainsi à chaque itération de l'algorithme du point fixe, la valeur x_i augmente et finit par atteindre x^*. □

Il existe peut-être plusieurs solutions à cette équation et l'algorithme permet d'obtenir la première.

Lemme 3 *L'algorithme du point fixe est pseudo-polynômial.*

Preuve : À partir de l'inégalité $f(x) \leq \sum_i \frac{b_i}{a_i} x + \sum_i b_i + c$, on a :

$$f(x^*) = x^* \leq \sum_i \frac{b_i}{a_i} x^* + \sum_i b_i + c$$

$$x^*(1 - \sum_i \frac{b_i}{a_i}) \leq \sum_i b_i + c$$

$$x^* \leq \frac{\sum_i^k b_i + c}{1 - \sum_i \frac{b_i}{a_i}}$$

Sous l'hypothèse que $\sum_i \frac{b_i}{a_i} < 1$. La valeur optimale est donc bornée par une fonction polynômiale en k (à cause du terme $\sum_i^k b_i$ où k correspond au nombre tâches dans le problème d'ordonnancement temps-réel). Néanmoins la valeur de cette fonction est dépendante des valeurs des b_i, a_i ce qui conduit à une complexité pseudo-polynômiale pour l'algorithme du point fixe qui énumère dans le pire des cas toutes les valeurs entières entre 0 et x^*. □

Cela signifie que le nombre d'itération peut grandement varier selon la valeurs de ces b_i, a_i pour un même nombre de tâches k. La complexité de l'algorithme du point fixe est donc pseudo-polynômiale mais nous ignorons toujours la complexité du problème lui-même.

Notons que la difficulté réside dans l'obtention du **premier** point fixe. Un algorithme polynômial existe pour trouver une solution quelconque de cette équation en procédant par dichotomie. Dès que l'équation possède plusieurs solutions comme c'est le cas sur la figure A.1, une procédure dichotomique n'est plus envisageable.

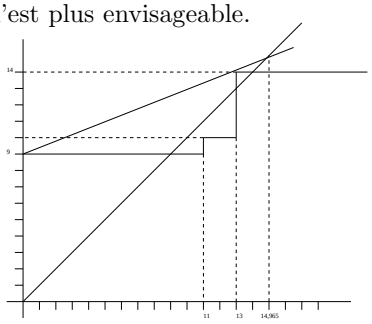


FIG. A.1: représentation graphique d'une fonction $x = \sum_{i=1}^{k} \left\lceil \frac{x}{a_i} \right\rceil b_i + c$

Annexe B

Résultats détaillés du placement de tâches

Nous rapportons ici des données détaillées sur la résolution du problème de placement. 100 instances dans 8 catégories (soit 800 instances au total) ont été générées aléatoirement selon le mode décrit section 9.3.1. Les valeurs moyennes, médianes, minimales et maximales sont présentées sur les données suivantes :

- CPU : temps de résolution en ms ;
- ITER : nombre d'itération entre problèmes maître et esclaves ;
- NODE : taille de l'espace de recherche exploré en nombre de nœuds ;
- NOE : nombre de contraintes de $NotAllEqual$ apprises (incluant les différences) ;
- COMB : nombre de combinaisons linéaires apprises.

Ces données sont mesurées sur les instances résolues (solution trouvée ou preuve d'inconsistance réalisée) en distinguant les instances consistantes et inconsistantes pour chaque heuristique de recherche. Chaque tableau correspond ainsi à une heuristique. I_2 et I_R utilisent une politique de *restart* qui augmente progressivement une limite sur le nombre de nœuds. Cette limite est initialisée au nombre de tâches fois 10 et croît à chaque itération en étant multiplié par 1,5.

1-1-3-1	67 instances résolues					67 instances consistantes					0 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	30842.9	83.7	20704.7	337.5	0.0	30842.9	83.7	20704.7	337.5	0.0	0.0	0.0	0.0	0.0	0.0
Médiane	718.0	43.0	816.0	231.0	0.0	718.0	43.0	816.0	231.0	0.0	0.0	0.0	0.0	0.0	0.0
Min	141.0	18.0	255.0	66.0	0.0	141.0	18.0	255.0	66.0	0.0	0.0	0.0	0.0	0.0	0.0
Max	468906.0	493.0	393308.0	1680.0	0.0	468906.0	493.0	393308.0	1680.0	0.0	0.0	0.0	0.0	0.0	0.0
1-2-2-3	43 instances résolues					27 instances consistantes					16 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	35274.7	97.4	19759.5	104.4	244.9	51156.7	154.3	28084.9	158.6	386.7	8473.8	1.6	5710.5	12.9	5.5
Médiane	3687.0	22.0	2145.0	55.0	39.0	8437.0	96.0	3873.0	117.0	232.0	296.5	1.0	211.0	8.0	4.0
Min	47.0	1.0	40.0	1.0	1.0	109.0	11.0	116.0	29.0	20.0	47.0	1.0	40.0	2.0	1.0
Max	338296.0	783.0	235348.0	546.0	1657.0	338296.0	783.0	235348.0	546.0	1657.0	71360.0	7.0	38343.0	54.0	20.0
2-2-2-1	90 instances résolues					55 instances consistantes					35 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	15071.9	27.4	14683.6	116.2	0.0	11029.2	43.9	12385.8	181.4	0.0	21424.7	1.4	18294.5	13.9	0.0
Médiane	188.0	15.0	342.5	67.0	0.0	234.0	26.0	434.0	124.0	0.0	78.0	1.0	89.0	12.0	0.0
Min	15.0	1.0	34.0	2.0	0.0	47.0	7.0	97.0	23.0	0.0	15.0	1.0	34.0	2.0	0.0
Max	260578.0	247.0	316801.0	789.0	0.0	196922.0	247.0	207477.0	789.0	0.0	260578.0	3.0	316801.0	47.0	0.0
2-2-2-2	95 instances résolues					70 instances consistantes					25 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	7845.0	26.3	5350.4	82.4	16.1	9001.1	35.3	6326.5	108.2	21.3	4608.0	1.3	2617.5	10.0	1.4
Médiane	203.0	22.0	364.0	77.0	5.0	250.0	26.5	398.5	98.5	9.0	32.0	1.0	50.0	9.0	1.0
Min	0.0	1.0	0.0	0.0	0.0	78.0	8.0	123.0	14.0	0.0	0.0	1.0	0.0	0.0	0.0
Max	318719.0	342.0	249085.0	264.0	359.0	318719.0	342.0	249085.0	264.0	359.0	91109.0	4.0	47892.0	34.0	3.0
2-2-2-3	43 instances résolues					28 instances consistantes					15 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	41274.3	152.4	23349.1	124.3	372.7	52318.7	233.5	31586.2	186.8	569.8	20658.3	1.2	7973.3	7.8	4.7
Médiane	3329.0	56.0	1839.0	48.0	95.0	6820.5	103.5	3182.0	84.5	268.0	219.0	1.0	113.0	6.0	5.0
Min	0.0	1.0	0.0	0.0	0.0	94.0	4.0	85.0	1.0	12.0	0.0	1.0	0.0	0.0	0.0
Max	385828.0	1414.0	288879.0	2149.0	3389.0	385828.0	1414.0	288879.0	2149.0	3389.0	219453.0	3.0	96312.0	19.0	12.0
2-2-3-1	43 instances résolues					13 instances consistantes					30 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	44087.2	23.4	39879.0	105.3	0.0	68469.9	73.6	55407.0	299.7	0.0	33521.4	1.7	33150.2	21.1	0.0
Médiane	6875.0	1.0	5143.0	18.0	0.0	7922.0	64.0	7803.0	279.0	0.0	3976.5	1.0	2499.0	14.5	0.0
Min	16.0	1.0	27.0	5.0	0.0	594.0	22.0	752.0	127.0	0.0	16.0	1.0	27.0	5.0	0.0
Max	499578.0	152.0	421409.0	599.0	0.0	499578.0	152.0	421409.0	599.0	0.0	306828.0	11.0	384986.0	118.0	0.0
2-3-2-1	81 instances résolues					16 instances consistantes					65 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	30984.9	9.3	18982.7	47.2	0.0	26020.3	40.1	35150.4	173.0	0.0	32207.0	1.7	15003.0	16.2	0.0
Médiane	109.0	1.0	115.0	13.0	0.0	1195.0	25.0	1525.0	122.0	0.0	63.0	1.0	68.0	12.0	0.0
Min	15.0	1.0	28.0	4.0	0.0	94.0	12.0	139.0	39.0	0.0	15.0	1.0	28.0	4.0	0.0
Max	595078.0	140.0	481142.0	569.0	0.0	307703.0	140.0	481142.0	569.0	0.0	595078.0	14.0	335672.0	93.0	0.0
3-2-2-1	85 instances résolues					48 instances consistantes					37 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	13101.7	36.8	13960.4	134.1	0.0	19466.2	64.1	21856.8	226.9	0.0	4844.9	1.4	3716.5	13.7	0.0
Médiane	235.0	16.0	330.0	50.0	0.0	352.0	36.0	546.0	132.0	0.0	93.0	1.0	82.0	11.0	0.0
Min	15.0	1.0	28.0	2.0	0.0	47.0	9.0	75.0	20.0	0.0	15.0	1.0	28.0	2.0	0.0
Max	379156.0	419.0	486926.0	1344.0	0.0	379156.0	419.0	486926.0	1344.0	0.0	110656.0	6.0	82015.0	64.0	0.0

TAB. B.1: Méthode Benders Mindom

1-1-3-1	75 instances résolues					75 instances consistantes					0 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	40489.6	101.1	23581.6	466.0	0.0	40489.6	101.1	23581.6	466.0	0.0	0.0	0.0	0.0	0.0	0.0
Médiane	5000.0	79.0	3688.0	435.0	0.0	5000.0	79.0	3688.0	435.0	0.0	0.0	0.0	0.0	0.0	0.0
Min	141.0	16.0	283.0	64.0	0.0	141.0	16.0	283.0	64.0	0.0	0.0	0.0	0.0	0.0	0.0
Max	471531.0	433.0	264536.0	1495.0	0.0	471531.0	433.0	264536.0	1495.0	0.0	0.0	0.0	0.0	0.0	0.0
1-2-2-3	81 instances résolues					60 instances consistantes					21 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	42197.2	236.7	17983.2	87.6	571.1	56757.3	318.9	24172.3	114.6	768.3	596.8	1.6	300.1	10.5	7.6
Médiane	3891.0	126.0	2033.0	34.0	269.0	11140.5	231.0	5836.0	81.0	443.0	219.0	1.0	100.0	6.0	4.0
Min	31.0	1.0	29.0	0.0	1.0	125.0	10.0	99.0	2.0	15.0	31.0	1.0	29.0	1.0	1.0
Max	499829.0	1592.0	171703.0	699.0	4287.0	499829.0	1592.0	171703.0	699.0	4287.0	3266.0	5.0	1824.0	41.0	35.0
2-2-2-1	95 instances résolues					51 instances consistantes					44 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	5227.6	17.5	6459.7	78.0	0.0	9465.7	31.4	11869.5	133.2	0.0	315.3	1.4	189.3	14.1	0.0
Médiane	109.0	12.0	207.0	39.0	0.0	203.0	23.0	387.0	106.0	0.0	47.0	1.0	50.0	11.5	0.0
Min	15.0	1.0	34.0	2.0	0.0	62.0	9.0	129.0	22.0	0.0	15.0	1.0	34.0	2.0	0.0
Max	447328.0	109.0	555065.0	495.0	0.0	447328.0	109.0	555065.0	495.0	0.0	8469.0	9.0	4345.0	75.0	0.0
2-2-2-2	100 instances résolues					70 instances consistantes					30 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	3661.3	17.8	3647.6	47.8	12.2	5189.5	24.9	5176.1	65.5	16.9	95.7	1.1	81.1	6.6	1.3
Médiane	125.0	12.0	196.5	31.0	4.5	156.0	17.0	229.0	47.0	9.0	31.0	1.0	41.5	5.5	1.0
Min	0.0	1.0	0.0	0.0	0.0	32.0	4.0	54.0	2.0	0.0	0.0	1.0	0.0	0.0	0.0
Max	216500.0	121.0	265270.0	280.0	113.0	216500.0	121.0	265270.0	280.0	113.0	1156.0	3.0	679.0	16.0	3.0
2-2-2-3	69 instances résolues					51 instances consistantes					18 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	64592.8	239.0	32787.7	103.1	557.2	79983.7	322.0	42162.1	134.9	750.0	20985.4	3.9	6227.0	13.1	10.8
Médiane	10234.0	104.0	5332.0	49.0	298.0	19579.0	165.0	9989.0	87.0	486.0	55.0	1.0	40.0	6.5	4.5
Min	0.0	1.0	0.0	0.0	0.0	203.0	14.0	118.0	1.0	13.0	0.0	1.0	0.0	0.0	0.0
Max	519281.0	1431.0	348262.0	637.0	2891.0	519281.0	1431.0	348262.0	637.0	2891.0	365547.0	52.0	104271.0	125.0	124.0
2-2-3-1	38 instances résolues					7 instances consistantes					31 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	33447.9	17.4	28207.6	92.9	0.0	166538.0	89.0	141541.6	428.3	0.0	3395.3	1.3	2616.1	17.1	0.0
Médiane	79.0	1.0	84.5	17.0	0.0	59125.0	80.0	37379.0	483.0	0.0	78.0	1.0	61.0	15.0	0.0
Min	15.0	1.0	34.0	6.0	0.0	6234.0	53.0	8177.0	264.0	0.0	15.0	1.0	34.0	6.0	0.0
Max	478641.0	167.0	399235.0	650.0	0.0	478641.0	167.0	399235.0	650.0	0.0	48969.0	3.0	38956.0	47.0	0.0
2-3-2-1	91 instances résolues					16 instances consistantes					75 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	6414.3	9.2	4079.4	42.5	0.0	20601.6	45.9	15075.9	183.1	0.0	3387.7	1.3	1733.5	12.5	0.0
Médiane	47.0	1.0	55.0	11.0	0.0	382.5	23.5	540.5	108.0	0.0	32.0	1.0	42.0	9.0	0.0
Min	15.0	1.0	28.0	3.0	0.0	47.0	7.0	69.0	17.0	0.0	15.0	1.0	28.0	3.0	0.0
Max	136015.0	179.0	73091.0	648.0	0.0	113156.0	179.0	73091.0	648.0	0.0	136015.0	9.0	62740.0	59.0	0.0
3-2-2-1	82 instances résolues					43 instances consistantes					39 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	13750.8	30.0	12132.1	123.2	0.0	26098.2	56.1	23033.9	223.6	0.0	137.0	1.1	112.3	12.4	0.0
Médiane	148.5	12.0	242.5	51.0	0.0	1360.0	42.0	1227.0	149.0	0.0	32.0	1.0	49.0	11.0	0.0
Min	15.0	1.0	33.0	2.0	0.0	78.0	10.0	172.0	39.0	0.0	15.0	1.0	33.0	2.0	0.0
Max	545281.0	303.0	385230.0	964.0	0.0	545281.0	303.0	385230.0	964.0	0.0	1657.0	3.0	863.0	29.0	0.0

TAB. B.2: Méthode Benders Learn

1-1-3-1	99 instances résolues					99 instances consistantes					0 instances inconsistantes				
	CPU	ITER	NODE	NOE	CUT	CPU	ITER	NODE	NOE	CUT	CPU	ITER	NODE	NOE	CUT
Moyenne	5652.7	107.5	3105.9	482.5	0.0	5652.7	107.5	3105.9	482.5	0.0	0.0	0.0	0.0	0.0	0.0
Médiane	2828.0	78.0	2125.0	367.0	0.0	2828.0	78.0	2125.0	367.0	0.0	0.0	0.0	0.0	0.0	0.0
Min	266.0	15.0	288.0	57.0	0.0	266.0	15.0	288.0	57.0	0.0	0.0	0.0	0.0	0.0	0.0
Max	60657.0	835.0	31146.0	3627.0	0.0	60657.0	835.0	31146.0	3627.0	0.0	0.0	0.0	0.0	0.0	0.0
1-2-2-3	99 instances résolues					76 instances consistantes					23 instances inconsistantes				
	CPU	ITER	NODE	NOE	CUT	CPU	ITER	NODE	NOE	CUT	CPU	ITER	NODE	NOE	CUT
Moyenne	15495.6	629.6	7227.4	192.8	1853.2	20088.6	819.6	9374.5	247.7	2411.8	318.6	1.8	132.3	11.7	7.5
Médiane	2641.0	152.0	1425.0	69.0	306.0	6891.0	302.5	3510.5	116.0	822.0	281.0	1.0	85.0	10.0	5.0
Min	187.0	1.0	1.0	0.0	0.0	297.0	6.0	175.0	13.0	10.0	187.0	1.0	1.0	0.0	0.0
Max	519765.0	10690.0	161059.0	2415.0	38642.0	519765.0	10690.0	161059.0	2415.0	38642.0	984.0	5.0	843.0	37.0	28.0
2-2-2-1	100 instances résolues					56 instances consistantes					44 instances inconsistantes				
	CPU	ITER	NODE	NOE	CUT	CPU	ITER	NODE	NOE	CUT	CPU	ITER	NODE	NOE	CUT
Moyenne	427.4	19.9	370.6	79.7	0.0	595.4	34.1	592.4	128.4	0.0	213.5	1.8	88.2	17.8	0.0
Médiane	266.0	12.0	239.5	44.0	0.0	343.5	18.5	355.0	83.5	0.0	203.0	1.0	74.0	13.0	0.0
Min	172.0	1.0	34.0	2.0	0.0	203.0	5.0	123.0	13.0	0.0	172.0	1.0	34.0	2.0	0.0
Max	4250.0	136.0	3428.0	504.0	0.0	4250.0	136.0	3428.0	504.0	0.0	344.0	10.0	293.0	100.0	0.0
2-2-2-2	100 instances résolues					70 instances consistantes					30 instances inconsistantes				
	CPU	ITER	NODE	NOE	CUT	CPU	ITER	NODE	NOE	CUT	CPU	ITER	NODE	NOE	CUT
Moyenne	354.8	17.2	282.7	48.5	12.5	410.3	23.9	369.4	64.2	17.0	225.5	1.8	80.4	11.9	2.0
Médiane	289.5	12.0	230.5	35.5	4.5	313.0	18.0	293.5	52.5	7.5	226.5	1.0	71.0	8.0	1.5
Min	0.0	1.0	0.0	0.0	0.0	187.0	4.0	103.0	2.0	0.0	0.0	1.0	0.0	0.0	0.0
Max	2656.0	149.0	1488.0	322.0	129.0	2656.0	149.0	1488.0	322.0	129.0	297.0	8.0	232.0	53.0	7.0
2-2-2-3	92 instances résolues					70 instances consistantes					22 instances inconsistantes				
	CPU	ITER	NODE	NOE	CUT	CPU	ITER	NODE	NOE	CUT	CPU	ITER	NODE	NOE	CUT
Moyenne	28228.5	662.0	10276.8	262.2	1784.9	28399.8	833.5	11664.4	318.5	2296.9	27683.3	116.4	5862.0	82.9	155.9
Médiane	4726.5	179.5	2196.0	72.5	380.0	7476.5	289.0	4038.0	155.5	704.0	265.0	1.0	69.0	7.0	4.5
Min	0.0	1.0	0.0	0.0	0.0	407.0	14.0	235.0	5.0	18.0	0.0	1.0	0.0	0.0	0.0
Max	386891.0	5953.0	96149.0	2772.0	18604.0	319469.0	5953.0	96149.0	2772.0	18604.0	386891.0	2086.0	95987.0	1169.0	2474.0
2-2-3-1	90 instances résolues					30 instances consistantes					60 instances inconsistantes				
	CPU	ITER	NODE	NOE	CUT	CPU	ITER	NODE	NOE	CUT	CPU	ITER	NODE	NOE	CUT
Moyenne	8364.9	30.1	3267.8	154.2	0.0	24346.3	87.2	9445.8	421.6	0.0	374.2	1.6	178.8	20.5	0.0
Médiane	250.0	2.0	114.0	22.5	0.0	7586.0	80.0	4388.0	383.0	0.0	219.0	1.0	73.0	15.5	0.0
Min	171.0	1.0	28.0	4.0	0.0	328.0	15.0	261.0	48.0	0.0	171.0	1.0	28.0	4.0	0.0
Max	344062.0	308.0	103358.0	1396.0	0.0	344062.0	308.0	103358.0	1396.0	0.0	3063.0	6.0	1932.0	86.0	0.0
2-3-2-1	100 instances résolues					19 instances consistantes					81 instances inconsistantes				
	CPU	ITER	NODE	NOE	CUT	CPU	ITER	NODE	NOE	CUT	CPU	ITER	NODE	NOE	CUT
Moyenne	437.7	6.8	172.6	37.3	0.0	1213.9	29.3	614.9	127.9	0.0	255.6	1.6	68.8	16.1	0.0
Médiane	265.0	1.0	63.0	14.0	0.0	422.0	29.0	398.0	111.0	0.0	250.0	1.0	60.0	13.0	0.0
Min	172.0	1.0	27.0	1.0	0.0	235.0	9.0	151.0	26.0	0.0	172.0	1.0	27.0	1.0	0.0
Max	6516.0	56.0	2912.0	322.0	0.0	6516.0	56.0	2912.0	322.0	0.0	375.0	10.0	248.0	112.0	0.0
3-2-2-1	99 instances résolues					57 instances consistantes					42 instances inconsistantes				
	CPU	ITER	NODE	NOE	CUT	CPU	ITER	NODE	NOE	CUT	CPU	ITER	NODE	NOE	CUT
Moyenne	1418.2	39.6	960.2	143.2	0.0	2181.7	66.3	1529.4	231.0	0.0	382.0	3.3	187.8	24.0	0.0
Médiane	328.0	20.0	314.0	61.0	0.0	594.0	42.0	660.0	147.0	0.0	219.0	1.0	69.5	11.0	0.0
Min	172.0	1.0	31.0	2.0	0.0	234.0	7.0	179.0	31.0	0.0	172.0	1.0	31.0	2.0	0.0
Max	15046.0	276.0	8107.0	827.0	0.0	15046.0	276.0	8107.0	827.0	0.0	6859.0	66.0	4574.0	332.0	0.0

TAB. B.3: Méthode Benders J_R

1-1-3-1	92 instances résolues					92 instances consistantes					0 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	48229.0	90.1	13058.4	497.4	0.0	48229.0	90.1	13058.4	497.4	0.0	0.0	0.0	0.0	0.0	0.0
Médiane	6547.5	71.5	3230.0	366.0	0.0	6547.5	71.5	3230.0	366.0	0.0	0.0	0.0	0.0	0.0	0.0
Min	219.0	11.0	174.0	43.0	0.0	219.0	11.0	174.0	43.0	0.0	0.0	0.0	0.0	0.0	0.0
Max	564984.0	363.0	237884.0	2370.0	0.0	564984.0	363.0	237884.0	2370.0	0.0	0.0	0.0	0.0	0.0	0.0
1-2-2-3	93 instances résolues					70 instances consistantes					23 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	37493.7	895.6	13560.8	396.7	2586.6	49707.3	1189.1	17977.5	522.8	3433.4	322.0	2.2	118.5	13.2	9.5
Médiane	5687.0	241.0	2210.0	117.0	587.0	10937.0	450.5	5527.5	263.5	1168.5	266.0	1.0	104.0	9.0	5.0
Min	187.0	1.0	1.0	0.0	0.0	344.0	11.0	174.0	23.0	15.0	187.0	1.0	1.0	0.0	0.0
Max	531531.0	11100.0	159066.0	3413.0	20502.0	531531.0	11100.0	159066.0	3413.0	20502.0	812.0	7.0	352.0	46.0	42.0
2-2-2-1	100 instances résolues					56 instances consistantes					44 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	1419.1	22.5	602.7	101.7	0.0	2360.8	38.7	1006.8	167.3	0.0	220.6	1.8	88.3	18.2	0.0
Médiane	281.0	13.5	251.5	55.5	0.0	360.0	23.5	383.5	117.0	0.0	219.0	1.0	72.0	12.0	0.0
Min	172.0	1.0	34.0	2.0	0.0	219.0	11.0	134.0	24.0	0.0	172.0	1.0	34.0	2.0	0.0
Max	80797.0	230.0	15966.0	787.0	0.0	80797.0	230.0	15966.0	787.0	0.0	281.0	8.0	252.0	83.0	0.0
2-2-2-2	100 instances résolues					70 instances consistantes					30 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	558.1	19.5	346.3	60.0	12.6	697.2	27.1	463.1	80.3	17.4	233.3	2.0	73.9	12.5	1.4
Médiane	312.0	13.0	234.5	47.5	4.5	328.5	17.5	280.0	65.5	8.0	234.5	1.0	70.0	7.0	1.0
Min	0.0	1.0	0.0	0.0	0.0	203.0	6.0	143.0	7.0	0.0	0.0	1.0	0.0	0.0	0.0
Max	13063.0	183.0	4554.0	417.0	192.0	13063.0	183.0	4554.0	417.0	192.0	344.0	20.0	317.0	110.0	5.0
2-2-2-3	85 instances résolues					65 instances consistantes					20 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	38449.8	519.9	14268.4	309.2	1489.2	50076.0	678.7	18597.0	400.6	1943.8	664.9	3.8	200.3	11.9	11.7
Médiane	5266.0	151.0	2525.0	112.0	401.0	17031.0	288.0	6949.0	198.0	757.0	266.0	1.0	61.0	5.5	4.5
Min	0.0	1.0	0.0	0.0	0.0	265.0	5.0	97.0	4.0	7.0	0.0	1.0	0.0	0.0	0.0
Max	374313.0	5356.0	111871.0	1871.0	12739.0	374313.0	5356.0	111871.0	1871.0	12739.0	8157.0	54.0	2782.0	121.0	149.0
2-2-3-1	73 instances résolues					14 instances consistantes					59 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	28947.4	21.1	11491.5	115.4	0.0	149116.3	102.9	58817.2	508.6	0.0	432.7	1.7	261.6	22.1	0.0
Médiane	234.0	1.0	72.0	18.0	0.0	114664.5	93.0	39654.5	481.0	0.0	219.0	1.0	68.0	16.0	0.0
Min	172.0	1.0	26.0	5.0	0.0	610.0	28.0	492.0	131.0	0.0	172.0	1.0	26.0	5.0	0.0
Max	441360.0	284.0	177411.0	1147.0	0.0	441360.0	284.0	177411.0	1147.0	0.0	6922.0	9.0	4635.0	118.0	0.0
2-3-2-1	100 instances résolues					19 instances consistantes					81 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	657.7	7.4	200.2	43.2	0.0	2339.6	31.7	748.3	154.5	0.0	263.1	1.7	71.6	17.1	0.0
Médiane	266.0	1.0	62.0	14.0	0.0	469.0	29.0	394.0	129.0	0.0	250.0	1.0	60.0	12.0	0.0
Min	188.0	1.0	25.0	2.0	0.0	234.0	9.0	155.0	27.0	0.0	188.0	1.0	25.0	2.0	0.0
Max	19266.0	87.0	2903.0	495.0	0.0	19266.0	87.0	2903.0	495.0	0.0	484.0	14.0	342.0	125.0	0.0
3-2-2-1	99 instances résolues					57 instances consistantes					42 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	4418.7	29.9	2157.9	130.6	0.0	7473.1	50.3	3663.4	212.0	0.0	273.5	2.2	114.7	20.0	0.0
Médiane	328.0	16.0	281.0	64.0	0.0	609.0	32.0	528.0	125.0	0.0	219.0	1.0	70.0	14.0	0.0
Min	172.0	1.0	30.0	2.0	0.0	234.0	8.0	161.0	32.0	0.0	172.0	1.0	30.0	2.0	0.0
Max	141360.0	238.0	70750.0	922.0	0.0	141360.0	238.0	70750.0	922.0	0.0	1672.0	26.0	1239.0	173.0	0.0

TAB. B.4: Méthode Benders I_2

1-1-3-1	86 instances résolues					86 instances consistantes					0 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	28710.2	1.0	19715.6	0.0	0.0	28710.2	1.0	19715.6	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Médiane	141.0	1.0	96.5	0.0	0.0	141.0	1.0	96.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Min	16.0	1.0	28.0	0.0	0.0	16.0	1.0	28.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Max	495313.0	1.0	322387.0	0.0	0.0	495313.0	1.0	322387.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1-2-2-3	73 instances résolues					55 instances consistantes					18 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	62872.9	1.0	19672.5	0.0	0.0	77878.5	1.0	24533.2	0.0	0.0	17022.6	1.0	4820.7	0.0	0.0
Médiane	1656.0	1.0	568.0	0.0	0.0	3157.0	1.0	1021.0	0.0	0.0	336.0	1.0	130.5	0.0	0.0
Min	16.0	1.0	10.0	0.0	0.0	47.0	1.0	25.0	0.0	0.0	16.0	1.0	10.0	0.0	0.0
Max	582344.0	1.0	189909.0	0.0	0.0	582344.0	1.0	189909.0	0.0	0.0	185546.0	1.0	52658.0	0.0	0.0
2-2-2-1	90 instances résolues					56 instances consistantes					34 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	20846.2	1.0	10645.3	0.0	0.0	7159.8	1.0	4551.1	0.0	0.0	43388.4	1.0	20682.7	0.0	0.0
Médiane	39.0	1.0	37.0	0.0	0.0	31.0	1.0	35.5	0.0	0.0	1125.0	1.0	484.5	0.0	0.0
Min	15.0	1.0	1.0	0.0	0.0	15.0	1.0	25.0	0.0	0.0	15.0	1.0	1.0	0.0	0.0
Max	557156.0	1.0	262938.0	0.0	0.0	298578.0	1.0	186952.0	0.0	0.0	557156.0	1.0	262938.0	0.0	0.0
2-2-2-2	94 instances résolues					70 instances consistantes					24 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	7656.1	1.0	5130.4	0.0	0.0	5431.1	1.0	4849.6	0.0	0.0	14145.9	1.0	5949.2	0.0	0.0
Médiane	47.0	1.0	35.0	0.0	0.0	32.0	1.0	34.0	0.0	0.0	1070.0	1.0	428.5	0.0	0.0
Min	16.0	1.0	0.0	0.0	0.0	31.0	1.0	24.0	0.0	0.0	16.0	1.0	0.0	0.0	0.0
Max	364781.0	1.0	328625.0	0.0	0.0	364781.0	1.0	328625.0	0.0	0.0	99859.0	1.0	42785.0	0.0	0.0
2-2-2-3	75 instances résolues					59 instances consistantes					16 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	76060.6	1.0	21935.9	0.0	0.0	83195.2	1.0	23910.8	0.0	0.0	49751.9	1.0	14653.4	0.0	0.0
Médiane	8594.0	1.0	2410.0	0.0	0.0	9546.0	1.0	2410.0	0.0	0.0	8570.5	1.0	2929.0	0.0	0.0
Min	0.0	1.0	0.0	0.0	0.0	46.0	1.0	25.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
Max	589625.0	1.0	288623.0	0.0	0.0	589625.0	1.0	288623.0	0.0	0.0	408219.0	1.0	114630.0	0.0	0.0
2-2-3-1	72 instances résolues					24 instances consistantes					48 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	42469.8	1.0	24493.3	0.0	0.0	37126.3	1.0	22379.5	0.0	0.0	45141.5	1.0	25550.1	0.0	0.0
Médiane	2867.0	1.0	1570.0	0.0	0.0	12031.0	1.0	8369.5	0.0	0.0	1117.5	1.0	451.0	0.0	0.0
Min	0.0	1.0	0.0	0.0	0.0	47.0	1.0	36.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
Max	431516.0	1.0	273902.0	0.0	0.0	180375.0	1.0	94811.0	0.0	0.0	431516.0	1.0	273902.0	0.0	0.0
2-3-2-1	90 instances résolues					19 instances consistantes					71 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	26830.2	1.0	14130.7	0.0	0.0	3269.7	1.0	2061.8	0.0	0.0	33135.1	1.0	17360.4	0.0	0.0
Médiane	695.5	1.0	404.5	0.0	0.0	63.0	1.0	35.0	0.0	0.0	1328.0	1.0	660.0	0.0	0.0
Min	0.0	1.0	0.0	0.0	0.0	31.0	1.0	23.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
Max	402703.0	1.0	253778.0	0.0	0.0	37641.0	1.0	18220.0	0.0	0.0	402703.0	1.0	253778.0	0.0	0.0
3-2-2-1	87 instances résolues					51 instances consistantes					36 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	47694.9	1.0	31188.3	0.0	0.0	42914.2	1.0	28864.0	0.0	0.0	54467.6	1.0	34481.1	0.0	0.0
Médiane	94.0	1.0	58.0	0.0	0.0	47.0	1.0	39.0	0.0	0.0	844.0	1.0	321.5	0.0	0.0
Min	16.0	1.0	8.0	0.0	0.0	16.0	1.0	25.0	0.0	0.0	16.0	1.0	8.0	0.0	0.0
Max	577797.0	1.0	369188.0	0.0	0.0	577797.0	1.0	369188.0	0.0	0.0	498750.0	1.0	365704.0	0.0	0.0

TAB. B.5: Méthode CP Mindom

1-1-3-1	99 instances résolues					99 instances consistantes					0 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	3876.1	1.0	1685.3	0.0	0.0	3876.1	1.0	1685.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Médiane	453.0	1.0	78.0	0.0	0.0	453.0	1.0	78.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Min	297.0	1.0	22.0	0.0	0.0	297.0	1.0	22.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Max	78610.0	1.0	30142.0	0.0	0.0	78610.0	1.0	30142.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1-2-2-3	100 instances résolues					76 instances consistantes					24 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	5144.1	1.0	1085.8	0.0	0.0	2251.9	1.0	531.8	0.0	0.0	14302.8	1.0	2840.2	0.0	0.0
Médiane	578.0	1.0	72.0	0.0	0.0	851.5	1.0	160.0	0.0	0.0	297.0	1.0	1.0	0.0	0.0
Min	234.0	1.0	1.0	0.0	0.0	296.0	1.0	16.0	0.0	0.0	234.0	1.0	1.0	0.0	0.0
Max	336250.0	1.0	68141.0	0.0	0.0	15125.0	1.0	5525.0	0.0	0.0	336250.0	1.0	68141.0	0.0	0.0
2-2-2-1	100 instances résolues					56 instances consistantes					44 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	5331.8	1.0	1985.9	0.0	0.0	9276.6	1.0	3545.5	0.0	0.0	311.1	1.0	1.0	0.0	0.0
Médiane	343.0	1.0	32.0	0.0	0.0	359.5	1.0	35.0	0.0	0.0	297.0	1.0	1.0	0.0	0.0
Min	234.0	1.0	1.0	0.0	0.0	281.0	1.0	26.0	0.0	0.0	234.0	1.0	1.0	0.0	0.0
Max	314984.0	1.0	121977.0	0.0	0.0	314984.0	1.0	121977.0	0.0	0.0	531.0	1.0	1.0	0.0	0.0
2-2-2-2	100 instances résolues					70 instances consistantes					30 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	364.1	1.0	36.6	0.0	0.0	407.8	1.0	51.9	0.0	0.0	262.1	1.0	1.0	0.0	0.0
Médiane	312.0	1.0	30.0	0.0	0.0	328.0	1.0	33.0	0.0	0.0	266.0	1.0	1.0	0.0	0.0
Min	0.0	1.0	0.0	0.0	0.0	204.0	1.0	19.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
Max	4141.0	1.0	1033.0	0.0	0.0	4141.0	1.0	1033.0	0.0	0.0	391.0	1.0	1.0	0.0	0.0
2-2-2-3	95 instances résolues					73 instances consistantes					22 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	13320.1	1.0	4323.3	0.0	0.0	12526.8	1.0	4280.2	0.0	0.0	15952.5	1.0	4466.0	0.0	0.0
Médiane	734.0	1.0	120.0	0.0	0.0	1016.0	1.0	166.0	0.0	0.0	289.5	1.0	1.0	0.0	0.0
Min	0.0	1.0	0.0	0.0	0.0	297.0	1.0	19.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
Max	403454.0	1.0	154098.0	0.0	0.0	403454.0	1.0	154098.0	0.0	0.0	319750.0	1.0	95039.0	0.0	0.0
2-2-3-1	90 instances résolues					30 instances consistantes					60 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	10444.2	1.0	4059.8	0.0	0.0	30686.8	1.0	12177.5	0.0	0.0	322.9	1.0	1.0	0.0	0.0
Médiane	359.0	1.0	1.0	0.0	0.0	1906.5	1.0	604.5	0.0	0.0	328.0	1.0	1.0	0.0	0.0
Min	16.0	1.0	0.0	0.0	0.0	328.0	1.0	25.0	0.0	0.0	16.0	1.0	0.0	0.0	0.0
Max	375062.0	1.0	142721.0	0.0	0.0	375062.0	1.0	142721.0	0.0	0.0	438.0	1.0	1.0	0.0	0.0
2-3-2-1	100 instances résolues					19 instances consistantes					81 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	365.2	1.0	9.2	0.0	0.0	528.8	1.0	44.4	0.0	0.0	326.8	1.0	0.9	0.0	0.0
Médiane	375.0	1.0	1.0	0.0	0.0	437.0	1.0	25.0	0.0	0.0	344.0	1.0	1.0	0.0	0.0
Min	0.0	1.0	0.0	0.0	0.0	250.0	1.0	20.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
Max	1938.0	1.0	254.0	0.0	0.0	1938.0	1.0	254.0	0.0	0.0	469.0	1.0	1.0	0.0	0.0
3-2-2-1	99 instances résolues					57 instances consistantes					42 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	1443.4	1.0	458.1	0.0	0.0	2283.5	1.0	795.0	0.0	0.0	303.2	1.0	1.0	0.0	0.0
Médiane	328.0	1.0	30.0	0.0	0.0	359.0	1.0	34.0	0.0	0.0	296.5	1.0	1.0	0.0	0.0
Min	234.0	1.0	1.0	0.0	0.0	266.0	1.0	27.0	0.0	0.0	234.0	1.0	1.0	0.0	0.0
Max	91172.0	1.0	35120.0	0.0	0.0	91172.0	1.0	35120.0	0.0	0.0	453.0	1.0	1.0	0.0	0.0

TAB. B.6: Méthode CP I_R

1-1-3-1	96 instances résolues					96 instances consistantes					0 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	7926.9	1.0	3810.8	0.0	0.0	7926.9	1.0	3810.8	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Médiane	1140.0	1.0	443.5	0.0	0.0	1140.0	1.0	443.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Min	297.0	1.0	24.0	0.0	0.0	297.0	1.0	24.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Max	105328.0	1.0	50229.0	0.0	0.0	105328.0	1.0	50229.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1-2-2-3	97 instances résolues					74 instances consistantes					23 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	10076.9	1.0	2287.0	0.0	0.0	13104.0	1.0	2995.1	0.0	0.0	337.7	1.0	8.9	0.0	0.0
Médiane	703.0	1.0	83.0	0.0	0.0	1633.0	1.0	397.0	0.0	0.0	328.0	1.0	8.0	0.0	0.0
Min	234.0	1.0	1.0	0.0	0.0	281.0	1.0	20.0	0.0	0.0	234.0	1.0	1.0	0.0	0.0
Max	265375.0	1.0	68747.0	0.0	0.0	265375.0	1.0	68747.0	0.0	0.0	516.0	1.0	26.0	0.0	0.0
2-2-2-1	100 instances résolues					56 instances consistantes					44 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	745.6	1.0	170.0	0.0	0.0	1065.0	1.0	293.3	0.0	0.0	339.2	1.0	13.0	0.0	0.0
Médiane	344.0	1.0	30.0	0.0	0.0	351.5	1.0	34.0	0.0	0.0	328.0	1.0	2.5	0.0	0.0
Min	250.0	1.0	1.0	0.0	0.0	281.0	1.0	26.0	0.0	0.0	250.0	1.0	1.0	0.0	0.0
Max	35906.0	1.0	12164.0	0.0	0.0	35906.0	1.0	12164.0	0.0	0.0	516.0	1.0	87.0	0.0	0.0
2-2-2-2	100 instances résolues					70 instances consistantes					30 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	346.8	1.0	30.4	0.0	0.0	379.3	1.0	42.1	0.0	0.0	270.8	1.0	3.1	0.0	0.0
Médiane	313.0	1.0	28.0	0.0	0.0	344.0	1.0	32.0	0.0	0.0	273.5	1.0	1.0	0.0	0.0
Min	0.0	1.0	0.0	0.0	0.0	203.0	1.0	15.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
Max	1735.0	1.0	435.0	0.0	0.0	1735.0	1.0	435.0	0.0	0.0	375.0	1.0	15.0	0.0	0.0
2-2-2-3	94 instances résolues					74 instances consistantes					20 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	25193.8	1.0	5749.4	0.0	0.0	31919.1	1.0	7302.0	0.0	0.0	310.1	1.0	4.5	0.0	0.0
Médiane	726.5	1.0	74.0	0.0	0.0	2469.0	1.0	437.5	0.0	0.0	312.5	1.0	1.0	0.0	0.0
Min	0.0	1.0	0.0	0.0	0.0	266.0	1.0	22.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
Max	497031.0	1.0	158445.0	0.0	0.0	497031.0	1.0	158445.0	0.0	0.0	516.0	1.0	18.0	0.0	0.0
2-2-3-1	85 instances résolues					25 instances consistantes					60 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	24402.7	1.0	9273.3	0.0	0.0	82034.3	1.0	31461.2	0.0	0.0	389.6	1.0	28.3	0.0	0.0
Médiane	407.0	1.0	18.0	0.0	0.0	16297.0	1.0	6598.0	0.0	0.0	359.0	1.0	8.0	0.0	0.0
Min	0.0	1.0	0.0	0.0	0.0	422.0	1.0	26.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
Max	567906.0	1.0	233212.0	0.0	0.0	567906.0	1.0	233212.0	0.0	0.0	766.0	1.0	299.0	0.0	0.0
2-3-2-1	100 instances résolues					19 instances consistantes					81 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	2967.0	1.0	1002.0	0.0	0.0	764.9	1.0	96.1	0.0	0.0	3483.6	1.0	1214.5	0.0	0.0
Médiane	422.0	1.0	18.0	0.0	0.0	438.0	1.0	27.0	0.0	0.0	406.0	1.0	14.0	0.0	0.0
Min	0.0	1.0	0.0	0.0	0.0	344.0	1.0	18.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
Max	66031.0	1.0	28500.0	0.0	0.0	4610.0	1.0	627.0	0.0	0.0	66031.0	1.0	28500.0	0.0	0.0
3-2-2-1	100 instances résolues					58 instances consistantes					42 instances inconsistantes				
	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB	CPU	ITER	NODE	NOE	COMB
Moyenne	1219.9	1.0	347.9	0.0	0.0	1815.2	1.0	574.2	0.0	0.0	398.0	1.0	35.2	0.0	0.0
Médiane	375.0	1.0	31.0	0.0	0.0	391.0	1.0	36.0	0.0	0.0	328.0	1.0	9.0	0.0	0.0
Min	250.0	1.0	1.0	0.0	0.0	281.0	1.0	25.0	0.0	0.0	250.0	1.0	1.0	0.0	0.0
Max	30265.0	1.0	8401.0	0.0	0.0	30265.0	1.0	8401.0	0.0	0.0	1687.0	1.0	616.0	0.0	0.0

TAB. B.7: Méthode CP I_2

Bibliographie

[AB93] A. Aggoun and N. Beldiceanu. Extending chip in order to solve complex scheduling and placement problems. *Mathematical Computing and Modelling*, 17(7) :57–73, 1993.

[AH98] P. Altenbernd and H. Hansson. The Slack Method : A New Method for Static Allocation of Hard Real-Time Tasks. *Real-Time Systems*, 15 :103–130, 1998.

[AKK+97] D. Achlioptas, L. Kirousis, E. Kranakis, D. Krizanc, M. Molloy, and Y. Stamatiou. Random constraint satisfaction : a more accurate picture. In *Principles and Practice of Constraint Programming (CP'97)*, pages 121–135, Linz, Autriche, 1997.

[Ami99] J. Amilhastre. *Représentation par automate d'ensemble de solutions de problèmes de satisfaction de contraintes*. PhD thesis, Université de Toulouse III, 1999.

[BBGM06] L. Benini, D. Bertozzi, A. Guerri, and M. Milano. Allocation, scheduling and voltage scaling on energy aware mpsocs. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'04)*, pages 44–58, 2006.

[BC02] N. Beldiceanu and M. Carlsson. A new multi-resource cumulatives constraint with negative heights. In *Principles and Practice of Constraint Programming (CP'02)*, pages 63–79, 2002.

[BCCL95] R. E. Bixby, W. Cook, A. Cox, and E. K. Lee. Parallel mixed integer programming. Technical Report CRPC–TR95554, Houston, Texas, 1995. Technical Report Research Monograph.

[BCR05] N. Beldiceanu, M. Carlsson, and J.-X. Rampon. Global constraint catalog. Technical Report T2005-06, Swedish Institute of Computer Science, 2005.

[BCS01] C. Bessière, A. Chmeiss, and L. Sais. Neighborhood-based variable ordering heuristics for the constraint satisfaction problem. In *Principles and Practice of Constraint Programming (CP'01)*, pages 565–569, 2001.

[Bel05] N. Beldiceanu. A cumulative model for a pattern sequencing problem. In *International Joint Conference on Artificial Intelligence (IJCAI), Fifth Workshop on Modelling and Solving Problems with Constraints*, Edinburgh, Ècosse, Juillet 2005.

[Ben62] J. F. Benders. Partitionning procedures for solving mixed-variables programming problems. *Numerische Mathematik*, 4 :238–252, 1962.

[Ben04] T. Benoist. *Relaxations et décompositions combinatoires*. PhD thesis, 2004.

[Ben05] T. Benoist. A dynamic programming approach. In *International Joint Conference on Artificial Intelligence (IJCAI'05), Fifth Workshop on Modelling and Solving Problems with Constraints*, Edinburgh, Scotland, Juillet 2005.

[Bes06] C. Bessiere. Constraint propagation. Technical report, LIRMM 06020, 2006. also as Chapter 3 of the Handbook of Constraint Programming, F. Rossi, P. van Beek and T. Walsh eds. Elsevier 2006.

[BG04] A. Bemporad and N. Giorgetti. Sat-based branch & bound and optimal control of hybrid dynamical systems. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'04)*, pages 96–111, 2004.

[BGR02] T. Benoist, E. Gaudin, and B. Rottembourg. Constraint programming contribution to benders decomposition : A case study. In *Principles and Practice of Constraint Programming (CP'02)*, number 2470 in Lecture Notes in Computer Science, pages 603–617. Springer-Verlag, 2002.

[BH03] C. Bessière and P. Van Hentenryck. To be or not to be ... a global constraint. In *Principles and Practice of Constraint Programming (CP'03)*, number 2833 in Lecture Notes in Computer Science, pages 789–794. Springer-Verlag, 2003.

[BHLS04] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *European Conference on Artificial Intelligence (ECAI'04)*, pages 146–150, 2004.

[BHZ05] L. Bordeaux, Y. Hamadi, and L. Zhang. Propositional satisfiability and constraint programming : A comparative survey. Technical report, Microsoft research, 2005.

[BJN+98] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance. Branch-and-price : column generation for solving huge integer programs. *Operations Research*, 46 :316–329, 1998.

[BK73] C. Bron and J. Kerbosch. Algorithm 457 : finding all cliques of an undirected graph. *Commun. ACM*, 16(9) :575–577, 1973.

[Bli98] C. Bliek. Generalizing partial order and dynamic backtracking. In *Fifth National Conference on Artificial Intelligence – AAAI'98*, 1998.

[BM96] R. J. Bayardo and D. P. Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *National Conference on Artificial Intelligence (AAAI-1996)*, pages 298–304, 1996.

[Bos91] Bosch. *CAN Specification version 2.0*, 1991.

[BR96] C. Bessiere and J.C. Régin. MAC and combined heuristics : Two reasons to forsake FC (and CBJ ?) on hard problems. In *Principles and Practice of Constraint Programming (CP'96)*, pages 61–75, 1996.

[BR97] C. Bessière and J.C. Régin. Arc consistency for general constraint networks : Preliminary results. In *International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 398–404, 1997.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8) :677–691, 1986.

[BS96] R. J. Bayardo and R. Schrag. Using csp look-back techniques to solve exceptionally hard sat instances. In *Principles and Practice of Constraint Programming (CP'96)*, pages 46–60, 1996.

[BYS04] J.C. Becceneri, H.H. Yannasse, and N.Y. Soma. A method for solving the minimization of the maximum number of open stacks problem within a cutting process. *Computer and Operations Research*, 31 :2315–2332, Juillet 2004.

[CdGL+99] B. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J.P. Warners. Radio link frequency assignment. *Constraints*, 4 :79–89, 1999.

[CHD+04] H. Cambazard, P.E. Hladik, A.M. Déplanche, N. Jussien, and Y. Trinquet. Decomposition and learning for a real time task allocation problem. In *Principles and Practice of Constraint Programming (CP'04)*, volume 3258 of *Lecture Notes in Computer Science*, Toronto, Canada, 2004. Springer-Verlag.

[Chv83] V. Chvátal. *Linear Programming*. A Series of Books in the Mathematical Sciences. Freeman, 1983.

[Chv97] V. Chvátal. Resolution search. *Discrete Applied Mathematics*, 73 :81–99, 1997.

[CJ05] Hadrien Cambazard and Narendra Jussien. Integrating benders decomposition within constraint programming. In *Principles and Practice of Constraint Programming (CP 2005)*, number 3709 in Lecture Notes in Computer Science, pages 752–756, Sitges, Spain, 2005. Springer-Verlag. Short paper.

[CJ06] H. Cambazard and N. Jussien. Identifying and exploiting problem structures using explanation-based constraint programming. *Constraints*, 2006.

[CKT91] P. Cheeseman, B. Kanefsky, and W. M. Taylor. Where the really hard problems are. In *International Joint Conference on Artificial Intelligence (IJCAI'91)*, pages 331–337, 1991.

[CLW96] B. M. W. Cheng, J. H. M. Lee, and J. C. K. Wu. Speeding up constraint propagation by redundant modeling. In *Principles and Practice of Constraint Programming*, pages 91–103, 1996.

[Col06] R. Colleta. *Apprentissage de Contraintes*. PhD thesis, Université Montpellier II, 2006.

[CvB01] X. Chen and P. van Beek. Conflict-directed backjumping revisited. *Journal of Artificial Intelligence Research*, 14 :53–81, 2001.

[CX05] Y. Chu and Q. Xia. A hybrid algorithm for a class of resource constrained scheduling problems. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'04)*, pages 110–124, 2005.

[Dan48] G. B. Dantzig. Programming in a linear structure. 1948.

[de 86] Johan de Kleer. An assumption-based tms. *Artificial Intelligence*, 28 :127–162, 1986.

[Deb98] R. Debruyne. *Ètude des consistances locales pour les problèmes de satisfaction de contraintes de grande taille*. PhD thesis, LIRMM-Université de Montpellier II, 1998.

[Dec90] R. Dechter. Enhancement schemes for constraint processing : Backjumping, learning and cutset decomposition. *Artificial Intelligence*, 41 :273–312, 1990.

[DF01] R. Dechter and Y. E. Fattah. Topological parameters for time-space tradeoff. *Artificial Intelligence*, 125(1-2) :93–118, 2001.

[DFJ+03] R. Debruyne, G. Ferrand, N. Jussien, W. Lesaint, S. Ouis, and A. Tessier. Correctness of constraint retraction algorithms. In *FLAIRS'03 : Sixteenth international Florida Artificial Intelligence Research Society conference*, pages 172–176, St. Augustine, Florida, USA, mai 2003. AAAI press.

[dK86] Johan de Kleer. Problem solving with the ATMS. *Artificial Intelligence*, 28 :197–224, 1986.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7) :394–397, 1962.

[DMWW00] J. Daciuk, S. Mihov, B. Watson, and R. Watson. Incremental construction of minimal acyclic finite state automata. *Computational Linguistics*, 26(1) :3–16, 2000.

[Doy79] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12 :231–272, 1979.

[DP87] R. Dechter and J. Pearl. The cycle cutset method for improving search performance in ai. In *Third IEEE on Artificial Intelligence Applications*, pages 224–230, Orlando, 1987.

[DP89] R. Dechter and J. Pearl. Tree-clustering for constraint networks. *Artificial Intelligence*, 38 :353–356, 1989.

[DS95] M. DiNatale and J. A. Stankovic. Applying of simulated annealing methods to real-time scheduling and jitter control. In *proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS 1995)*, 1995.

[dSP88] J.L. de Siqueira and J.F. Puget. Explanation-based generalisation of failures. In *European Conference on Artificial Intelligence (ECAI'88)*, pages 339–344, 1988.

[EG93] T. Eiter and G. Gottlob. The complexity of logic-based abduction. In *Symposium on Theoretical Aspects of Computer Science*, pages 70–79, 1993.

[Eke04] C. Ekelin. *An Optimization Framework for Scheduling of Embedded Real-Time Systems*. PhD thesis, Chalmers University of Technology, 2004.

[ES04] N. Eén and N. Sörensson. An extensible SAT-solver. *Theory and Applications of Satisfiability Testing*, 2919 :502–518, 2004.

[Eve63] H. Everett. Generalized lagrange multiplier method for solving problems of optimum allocation of resources. *Operations Research*, 11(3) :399–417, 1963.

[EW99] A. A. Elsadek and B. E. Wells. A heuristic model for task allocation in heterogeneous distributed computing systems. *The International Journal of Computers and Their Applications*, 6(1), 1999.

[EW01] A. Eremin and M. Wallace. Hybrid benders decomposition algorithms in constraint logic programming. In Toby Walsh, editor, *Proceedings of the International Conference on Principles and Practice of Constraint Programming, CP'01*, volume 2239 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2001.

[FHT74] J. J. H. Forest, J. P. H. Hirst, and J. A. Tomlin. Practical solution of large mixed integer programming problems with Umpire. *Management Science*, 20(5) :736–773, 1974.

[FLM00] F. Focacci, A. Lodi, and M. Milano. Cutting planes in constraint programming : An hybrid approach. In *Principles and Practice of Constraint Programming (CP'00)*, number 1894 in Lecture Notes in Computer Science, pages 187–201, Singapore, September 2000. Springer-Verlag.

[Gas77] J. Gaschning. A general backtrack algorithm that eliminates most redundant tests. In *Fifth International Joint Conference on Artificial Intelligence*, page 457, 1977.

[GCFJ05] M. Ghoniem, H. Cambazard, J.D. Fekete, and N. Jussien. Peeking in solver strategies using explanations – visualization of dynamic graphs for constraint programming. In *ACM symposium on Software Visualization (SOFTVIS'05)*, Saint-Louis, MO, USA, 2005.

[Geo72] A. M. Geoffrion. Generalized benders decomposition. *Journal of Optimization Theory And Practice*, 10(4), 1972.

[GFSB04] C. P. Gomes, C. Fernandez, B. Selman, and C. Bessière. Statistical regimes across constrainedness regions. In *Principles and Practice of Constraint Programming (CP'04)*, pages 32–46, 2004.

[GFSB05] C. P. Gomes, C. Fernandez, B. Selman, and C. Bessière. Statistical regimes across constrainedness regions. *Constraints*, 10(4) :317–337, 2005.

[Gin93] M. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1 :25–46, 1993.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Tractability.* A Series of Books in the Mathematical Sciences. Victor Klee, 1979.

[GJF04] M. Ghoniem, N. Jussien, and J-D. Fekete. VISEXP : visualizing constraint solver dynamics using explanations. In *FLAIRS'04 : Seventeenth international Florida Artificial Intelligence Research Society conference*, pages 263–268, Miami, Florida, USA, mai 2004. AAAI press.

[GJP00] C. Guéret, N. Jussien, and C. Prins. Using intelligent backtracking to improve branch-and-bound methods : an application to open-shop problems. *European Journal of Operational Research*, 127 :344–354, 2000.

[GJR04] E. Gaudin, N. Jussien, and G. Rochart. Implementing explained global constraints. In *CP04 Workshop on Constraint Propagation and Implementation (CPAI'04)*, pages 61–76, Toronto, Canada, September 2004.

[GLS00] G. Gottlob, N. Leone, and F. Scarcello. A comparison of structural csp decomposition methods. *Artificial Intelligence*, 124(2) :243–282, 2000.

[GM94] M. Ginsberg and D.A. McAllester. GSAT and dynamic backtracking. In *International Conference on the Principles of Knowledge Representation*, pages 226–237, 1994.

[GN02] E. Goldberg and Y. Novikov. Berkmin : A fast and robust SAT-solver. In *Design, Automation, and Test in Europe (DATE)*, pages 142–149, 2002.

[GS05] M. Garcia de la Banda and P. J. Stuckey. Dynamic programming to minimize the maximum number of open stacks. In *International Joint Conference on Artificial Intelligence (IJCAI'05), Fifth Workshop on Modelling and Solving Problems with Constraints*, Edinburgh, Scotland, Juillet 2005.

[GW94] I. P. Gent and T. Walsh. Easy problems are sometimes hard. *Artificial Intelligence*, 70(1-2) :335–345, 1994.

[HE80] R. Haralick and G. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial intelligence*, 14(9) :263–313, 1980.

[Hen02] P. Van Hentenryck. Constraint and integer programming in opl. *INFORMS Journal on computing*, 14(4) :345–372, 2002.

[Hla04] P.E. Hladik. *Ordonnançabilité et placement des systèmes temps réel distribués, préemptifs et à priorités fixes.* PhD thesis, Université Nantes, 2004.

[HO03] J.N. Hooker and G. Ottosson. Logic-based benders decomposition. *Mathematical Programming*, 96 :33–60, 2003.

[HO06] T. Hulubei and B. O'Sullivan. The impact of search heuristics on heavy-tailed behaviour. *Constraints*, 11(2-3) :159–178, 2006.

[Hoo06] J.N. Hooker. Duality in optimization and constraint satisfaction. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'06)*, pages 3–15, 2006.

[HOTK00] J.N. Hooker, G. Ottosson, E. S. Thorsteinsson, and H. Kim. A scheme for unifying optimization and constraint satisfaction methods. *Knowledge Engineering Review, special issue on AI/OR*, 15(1) :11–30, 2000.

[HW94] T. Hogg and C. P. Williams. The hardest constraint problems : A double phase transition. *Artificial Intelligence*, 69 :359–377, 1994.

[JB97] N. Jussien and P. Boizumault. Dynamic backtracking with constraint propagation – application to static and dynamic csps. In *CP'97 Workshop on The Theory and Practice of Dynamic Constraint Satisfaction*, 1997.

[JB00] N. Jussien and V. Barichard. The PaLM system : explanation-based constraint programming. In *Proceedings of TRICS : Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, pages 118–133, Singapore, September 2000.

[JDB00] N. Jussien, R. Debruyne, and P. Boizumault. Maintaining arc-consistency within dynamic backtracking. In *Principles and Practice of Constraint Programming (CP'00)*, number 1894 in Lecture Notes in Computer Science, pages 249–261, Singapore, September 2000. Springer-Verlag.

[JG01] V. Jain and I. E. Grossmann. Algorithms for hybrid milp/cp models for a class of optimization problems. *INFORMS Journal on Computing*, 13 :258–276, 2001.

[JL87] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL '87)*, pages 111–119, New York, NY, USA, 1987. ACM Press.

[JL02a] N. Jussien and O. Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, 139(1) :21–45, July 2002.

[JL02b] N. Jussien and O. Lhomme. Vers une unification des algorithmes de résolution de CSP. In *8ièmes Journées nationales sur la résolution pratique de problèmes NP-complets (JNPC'02)*, pages 155–168, Nice, France, mai 2002.

[JNT05] P. Jégou, S. N. Ndiaye, and C. Terrioux. Sur la génération et l'exploitation de décompositions pour la résolution de réseaux de contraintes. In *Actes des Premières Journées Francophones de Programmation par Contraintes (JFPC'2005)*, pages 149–158, Lens, juin 2005.

[Jos03] J. R. Josephson. *Abductive Inference : Computation, Philosophy, Technology.* Cambridge University Press, 2003.

[JT03] P. Jégou and C. Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146(1) :43–75, 2003.

[JT05] P. Jégou and C. Terrioux. Un compromis temps-espace pour la résolution de réseaux de contraintes par décomposition. In *Actes des Premières Journées Francophones de Programmation par Contraintes (JFPC'2005)*, pages 159–168, Lens, juin 2005.

[Jun01] U. Junker. Quickxplain : Conflict detection for arbitrary constraint propagation algorithms. In *International Joint Conference on Artificial Intelligence (IJCAI), Workshop on Modelling and Solving problems with constraints (CONS-1)*, Seattle, WA, USA, August 2001.

[Jus97] N. Jussien. *Relaxation de Contraintes pour les problèmes dynamiques*. PhD thesis, Université de Rennes I, Rennes, France, 24 October 1997.

[Jus03] N. Jussien. The versatility of using explanations within constraint programming. Thèse d'habilitation de l'université de Nantes, 18 September 2003. aussi disponible comme rapport de recherche RR-03-04 à École des Mines de Nantes.

[Kar84] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4) :373–395, 1984.

[KB03] G. Katsirelos and F. Bacchus. Unrestricted nogood recording in CSP search. In *Principles and Practice of Constraint Programming (CP'03)*, pages 873–877, 2003.

[KB05] G. Katsirelos and F. Bacchus. Generalized nogoods in csps. In *National Conference on Artificial Intelligence (AAAI-2005)*, pages 390–396, 2005.

[Knu97] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, third edition, 1997.

[Lab03] P. Laborie. Algorithms for propagating resource constraints in ai planning and scheduling : existing approaches and new results. *Artificial Intelligence*, 143(2) :151–188, 2003.

[LL73] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real time environment. *Journal ACM*, 20(1) :46–61, 1973.

[LM01] B. Lisper and P. Mellgren. Response-time calculation and priority assignment with integer programming methods. In Eduardo Tovar and Christer Norström, editors, *Work-in-progress and Industrial Sessions, 13th Euromicro Conference on Real-Time Systems*, pages 13–16, Delft, June 2001.

[LMSK63] J.D.C. Little, K.G. Murty, D.W. Sweeney, and C. Karel. An algorithm for the traveling salesman problem. *Operations Research*, 11 :972–989, 1963.

[LP01] I. J. Lustig and J-F. Puget. Program does not equal program : Constraint programming and its relationship to mathematical programming. *Interfaces*, 31(6) :29–53, 2001.

[LS99] J. T. Linderoth and M. W. P. Savelsbergh. A computational study of branch and bound search strategies for mixed integer programming. *INFORMS Journal on Computing*, 11 :173–187, 1999.

[LW82] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2 :237–250, 1982.

[MBD98] Y. Monnier, J.-P. Beauvais, and A.-M. Déplanche. A Genetic Algorithm for Scheduling Tasks in a Real-Time Distributed System. *24th Euromicro Conference*, 2, 1998.

[MG04] C. T. Maravelias and I. E. Grossmann. Using milp and cp for the scheduling of batch chemical processes. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'04)*, pages 1–20, 2004.

[Mih98] S. Mihov. Direct building of minimal automaton for given list. In *Annuaire de l'Université de Sofia St. Kl. Ohridski*, volume 91. Sofia, Bulgaria, 1998.

[Mil03] M. Milano, editor. *Constraint and Integer Programming : Toward a unified methodology*. Kluwer Academic Publishers, 2003.

[Mit05] D. G. Mitchell. A sat solver primer. *Bulletin of the EATCS*, 85 :112–132, 2005.

[ML95] M. Mutka and J-P. Li. A tool for allocating periodic real-time tasks to a set of processors. *The Journal of Systems and Software*, 29(2) :135–164, 1995.

[MMZ+01] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff : Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.

[Mon74] U. Montanari. Networks of constraints : fundamental properties and applications to picture processing. In *Information Sciences*, volume 7, pages 95–132, 1974.

[MT93] T. Muntean and E-G. Talbi. Hill-climbing, simulated annealing and genetic algorithms, a comparative study. In *proceedings of the 26th Hawaii International Conference on Task Scheduling in Parallel and Distributed Systems (HICSS-26)*, 1993.

[MZK+99] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. Determining computational complexity from characteristic 'phase transitions'. *Nature*, 400 :133–137, 1999.

[OJB03] S. Ouis, N. Jussien, and P. Boizumault. k-relevant explanations for constraint programming. In *FLAIRS'03 : Sixteenth international Florida Artificial Intelligence Research Society conference*, pages 192–196, St. Augustine, Florida, USA, mai 2003. AAAI press.

[Pes04] G. Pesant. A regular language membership constraint for finite sequences of variables. In LNCS Springer, editor, *Tenth International Conference on Principles and Practice of Constraint Programming (CP'04)*, volume 3258, 2004.

[Pes05] G. Pesant. Trying hard to solve the simultaneously open stacks problem with cp. In *International Joint Conference on Artificial Intelligence (IJCAI'05), Fifth Workshop on Modelling and Solving Problems with Constraints*, Edinburgh, Scotland, Juillet 2005.

[PR93] S. C. S. Porto and C.C. Ribeiro. A tabu search approach to task scheduling on heterogeneous processors under precedence constraints. *International Journal of High-Speed Computing*, 7(2), 1993.

[Pro95] P. Prosser. MAC-CBJ : maintaining arc consistency with conflict-directed backjumping. Technical Report Research Report/95/177, Dept. of Computer Science, University of Strathclyde, 1995.

[PSW00] P. Prosser, K. Stergiou, and T. Walsh. Singleton consistencies. In *Principles and Practice of Constraint Programming (CP'00)*, pages 353–368, Singapore, 2000.

[PV04] C. Pralet and G. Verfaillie. Travelling in the world of local searches in the space of partial assignments. In *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'04)*, pages 240–255, 2004.

[Ram90] K. Ramamritham. Allocation and Scheduling of Complex Periodic Tasks. *10th International Conference on Distributed Computing Systems*, pages 108–115, 1990.

[RCOJ06] Guillaume Richaud, Hadrien Cambazard, Barry O'Sullivan, and Narendra Jussien. Automata for nogood recording in constraint satisfaction problems. In *CP06 Workshop on the Integration of SAT and CP techniques*, Nantes, France, 2006.

[Ref04] P. Refalo. Impact-based search strategies for constraint programming. In *Principles and Practice of Constraint Programming (CP'04*, pages 556–571, 2004.

[Rég96] J.C. Régin. Generalized arc consistency for global cardinality constraint. In *National Conference on Artificial Intelligence (AAAI'96)*, pages 209–215, 1996.

[Rég94] J.C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, pages 362–367, 1994.

[Ric02] M. Richard. *Contribution à la Validation des Systèmes Temps Réel Distribués : Ordonnancement à Priorités Fixes & Placement.* PhD thesis, Université de Poitiers, November 2002.

[Ric06] P. Richard. Polynomial-time approximate schedulability tests for fixed-priority real-time tasks : Some numerical experimentations. In *14th Int. Conf. on Real-Time and Network Systems*, pages 191–199, 2006.

[Roc05] G. Rochart. *Explications et programmation par contraintes avancée.* PhD thesis, Université Nantes, 2005.

[RRC03] M. Richard, P. Richard, and F. Cottet. Allocating and scheduling tasks in multiple fieldbus real-time systems. In *IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 16, pages 137–144, 2003.

[San96] F. E. Sandnes. A hybrid genetic algorithm applied to automatic parallel controller code generation. *8th IEEE Euromicro Workshop on Real-Time Systems*, 1996.

[Sch05] A. Schrijver. *Handbook of Discrete Optimization*, chapter On the history of combinatorial optimization (till 1960), pages 1–68. Elsevier, Amsterdam, 2005.

[SF94] D. Sabin and E. C. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In Alan Borning, editor, *Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming, PPCP'94, Rosario, Orcas Island, Washington, USA*, volume 874, pages 10–20, 1994.

[SFOC97] J. Santos, E. Ferro, J. Orozco, and R. Cassials. A heuristic approach to multitask-multiprocessing assignment problem using the empty-slots method and rate monotonic scheduling. *Real-Time Systems*, 13(2) :167–199, 1997.

[SG95] B. Smith and S. Grant. Sparse constraint graphs and exceptionally hard problems. In Chris Mellish, editor, *International Joint Conference on Artificial Intelligence (IJCAI'95)*, Montreal, 1995.

[SG05] B. Smith and I. Gent. Constraint modelling challenge 2005, Juillet 2005.

[SGK00] R. Szymanek, F. Gruian, and K. Kuchcinski. Digital systems design using constraint logic programming. In *proceedings of The Practical Application of Constraint technologies and Logic Programming (PACLP 2000)*, 2000.

[Sha04] P. Shaw. A constraint for bin packing. In *Principles and Practice of Constraint Programming (CP'04)*, pages 648–662, 2004.

[SK03] R. Szymanek and K. Kuchcinski. Partial task assignment of task graphs under heterogeneous resource constraints. In *Conference on Design automation (DAC '03)*, pages 244–249, 2003.

[SL05] P. Shaw and P. Laborie. A constraint programming approach to the min-stack problem. In *International Joint Conference on Artificial Intelligence (IJCAI'05), Fifth Workshop on Modelling and Solving Problems with Constraints*, Edinburgh, Scotland, Juillet 2005.

[Smi95] B. M. Smith. In search of exceptionally difficult constraint satisfaction problems. In *Constraint Processing, Selected Papers*, pages 139–155, 1995.

[SS96] J. Silva and K. Sakallah. Conflict analysis in search algorithms for propositional satisfiability. Technical report, Cadence European Laboraties, 1996.

[Sto77] H. Stone. Multiprocessor scheduling the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, 3(1) :85–93, 1977.

[SV94] T. Schiex and G. Verfaillie. Nogood recording for static and dynamic constraint satisfaction problem. *International Journal of Artificial Intelligence Tools*, 3(2) :187–207, 1994.

[SW00] K. Schild and J. Würtz. Scheduling of time-triggered real-time systems. *Constraints*, 5(4) :335–357, 2000.

[TBW92] K. Tindell, A. Burns, and A. Wellings. Allocation Hard Real-Time tasks : An NP-Hard Problem Made Easy. *The Journal of Real-Time Systems*, 4(2) :145–165, 1992.

[TC94] K. Tindell and J. Clark. Holistic scheduling analysis for distributed hard real-time systems. *Euromicro Journal*, pages 40–117, 1994.

[Tho01] E. S. Thorsteinsson. Branch-and-check : A hybrid framework integrating mixed integer programming and constraint logic programming. In *Principles and Practice of Constraint Programming (CP'01)*, 2001.

[Vem92] N. R. Vempaty. Solving constraint satisfaction problems using finite state automata. In *National Conference on Artificial Intelligence (AAAI'92)*, pages 453–458, 1992.

[VO05] L. Vargas and R. Olivaira. Empirical study of tabu search, simulated annealing and multi-start in fieldbus scheduling. In *IEEE Conference on Emerging Technologies and Factory Automation ETFA*, volume 1, pages 101–108, 2005.

[WGS03a] R. Williams, C. Gomes, and B. Selman. On the connections between backdoors and heavy-tails on combinatorial search. In *the International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2003.

[WGS03b] R. Williams, C P. Gomes, and B. Selman. Backdoors to typical case complexity. In *International Joint Conference on Artificial Intelligence (IJCAI'03)*, 2003.

[WP05] N. Wilson and K. Petrie. Using customer elimination orderings to minimise the maximum number of open stacks. In *International Joint Conference on Artificial Intelligence (IJCAI'05), Fifth Workshop on Modelling and Solving Problems with Constraints*, Edinburgh, Scotland, Juillet 2005.

[XEW04] Q. Xia, A. Eremin, and M. Wallace. Problem decomposition for traffic diversions. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'04)*, pages 348–363, 2004.

[ZMMM01] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in boolean satisfiability solver. In *Internation Conference on Computed Aided Design (ICCAD)*, pages 279–285, 2001.

[ZS00] H. Zhang and M. E. Stickel. Implementing the Davis-Putnam method. *Journal of Automated Reasoning*, 24(1/2) :277–296, 2000.

Résolution de problèmes combinatoires par des approches fondées sur la notion d'explication

Hadrien Cambazard

La programmation par contraintes est un paradigme de résolution des problèmes combinatoires sur lequel ont été bâtis des outils génériques de résolution, des solveurs. De nombreuses recherches sont menées pour élargir le champ d'application de ces outils à des problèmes dynamiques et sur-contraints. Un axe prometteur s'appuie sur la notion d'explications. Les explications constituent une trace explicite du comportement du solveur et ont été initialement introduites pour améliorer les algorithmes de recherche arborescente. Depuis ce jour, elles ont ouvert la voie à des méthodes d'exploration plus intelligentes (mais aussi plus coûteuses) de l'espace de recherche. Cette thèse porte sur l'élaboration d'algorithmes de résolution s'appuyant sur la notion d'explications et les étudie sur des problèmes autant académiques qu'industriels.

D'une part, nous examinons l'intérêt des explications dans le cadre de techniques génériques de décomposition. La mise au point d'un tel algorithme dans le contexte d'ordonnancement temps réel a montré la souplesse de la technique pour permettre la coopération de méthodes analytiques pointues avec un solveur de contraintes.

D'autre part, nous montrons que le réseau d'explication constitue une information particulièrement pertinente pour révéler à un utilisateur les structures ou relations entretenues par différents éléments (variables/contraintes) du problème. Cette information, également exploitable dynamiquement par le solveur est un pas supplémentaire vers des approches de résolution génériques.

Enfin, les explications ont été jusqu'ici très utilisées dans un cadre rétrospectif et pourraient l'être davantage dans un cadre prospectif (à l'image de leur exploitation par la communauté SAT). Nous revenons ainsi dans cette thèse sur des techniques de *nogoods recording* dans le cadre du problème de MOSP (Minimum Open Stack Problem).

Mots-clés : Programmation par contraintes, Explications, Décomposition

Constraint programming is a search paradigm for solving combinatorial optimization problems, that has been used to design generic solvers. Numerous researches are conducted to deal with over-constrained and dynamic problems. One of those, is based on the concept of explanations. Explanations provide a trace of the behavior of the solver and have been initially introduce to improve backtracking based algorithms. They have been used to design clever but costly ways of exploring the search space since that day. This phd thesis study explanation based algorithms on industrial as well as academical problems.

We study the interest of explanation within generic decomposition techniques and implement such an algorithm for a hard real time task allocation problem. This approach outlines the role of explanations within the cooperation of different solving techniques.

We also show that the explanation network is a relevant information to analyse the structures of a problem and understand the relationships between its different parts (variables and constraints). This information, used to improve the search heuristic, is another step toward generic search techniques.

Finally, explanations have been often used for look-back but are still under-exploited for look-ahead in CP. Nogood recording techniques have never been successful contrary to what happended in the SAT community. We implement in this thesis such a nogood recording in the case of the minimum open stack problem.

Keywords : Constraint Programming, Explanations, Decomposition