UNIVERSITÉ D'ARTOIS

Exploiter les conflits pour réduire l'effort de recherche en satisfaction de contraintes

THÈSE

présentée et soutenue publiquement le 22 novembre 2007

pour l'obtention du

Doctorat de l'université d'Artois – Faculté des sciences J. Perrin

(spécialité informatique)

par

Sébastien Tabary

Composition du jury

Rapporteurs :	Gérard Verfaillie (Maître de Recherches)	ONERA Toulouse
	Narendra Jussien (Maître-assistant, HDR)	École des Mines de Nantes
Examinateurs :	Christian Bessière (Directeur de Recherches CNRS)	LIRMM Montpellier
	Eric Grégoire (Professeur des Universités)	CRIL Lens
	Ulrich Junker (Docteur en informatique)	ILOG Valbonne
	Christophe Lecoutre (Maître de Conférences)	CRIL Lens
	Lakhdar Sais (Professeur des Universités)	CRIL Lens
	Vincent Vidal (Maître de Conférences)	CRIL Lens

Centre de Recherche en Informatique de Lens — FRE 2499

cril

Remerciements

En premier lieu, je tiens à remercier Narendra Jussien et Gérard Verfaillie qui ont eu la gentillesse d'accepter de rapporter cette thèse. Merci pour vos nombreux conseils.

Mes remerciements vont également à Christophe Lecoutre, Lakhdar Sais et Vincent Vidal, mes encadrants de thèse. Ils m'ont fait découvrir le monde de la recherche. Merci pour votre confiance et les multiples discussions qui ont permis l'aboutissement de cette thèse. Je tiens également à remercier Eric Grégoire, directeur de thèse et directeur du CRIL pour m'avoir fait confiance et m'avoir accepté au sein de son laboratoire. J'ai pu grâce à son soutien participer activement à de nombreuses conférences et découvrir ainsi la communauté scientifique.

Je souhaite également remercier tous les membres du CRIL, doctorants et permanents. Ce fut un réel plaisir de travailler à vos cotés durant ces dernières années.

Merci également aux membres du département Informatique de l'IUT de Lens et notamment Valérie, Claire et les autres. J'ai rencontré au sein de l'IUT de nouveaux collégues mais aussi de nouveaux amis. Sans eux cette thèse n'aurait pas été ce qu'elle est aujourd'hui. Je n'oublie pas non plus les anciens du département qui ont continué leur chemin, merci à David, Nadège et bien d'autres encore.

Finalement je tiens à remercier mes parents sans qui rien de tout cela n'aurait été possible. Merci pour leur patience et pour m'avoir permis de développer ce goût pour la science. Merci également au reste de ma famille.

Table des matières

Introduction générale

1 Cadre général

De nombreux problèmes de notre vie quotidienne peuvent être définis à l'aide de contraintes. Par exemple, lorsqu'on doit établir un emploi du temps, il est nécessaire de prendre en considération un ensemble de contraintes, définissant par exemple les disponibilités des intervenants ou encore celles des salles de formation. On peut alors représenter les créneaux horaires par des variables dans lesquelles on peut placer différentes formations. Ces formations sont alors considérées comme des "valeurs" que peuvent prendre les variables. La notion de "Problème de Satisfaction de Contraintes" (ou CSP pour Constraint Satisfaction Problem) regroupe sous une étiquette unique les problèmes modélisables sous la forme de contraintes. La résolution de ce type de problème devient rapidement très compliquée dès que l'on complexifie un peu les données initiales du problème. Par exemple si l'on reprend le problème d'emploi du temps précédent, l'augmentation du nombre de matières rend en général la résolution du problème plus difficile. Ceci est dû au fait que ces problèmes sont hautement combinatoires dans la mesure ou il est nécessaire d'évaluer un nombre très important de combinaisons pour trouver éventuellement une solution acceptable pour le problème.

Très vite la puissance de calcul d'un ordinateur devient nécessaire si l'on veut résoudre des problèmes de taille conséquente représentant notamment des problèmes réels. Pour cela, des programmes informatiques dédiés à leur résolution, appelés solveurs de contraintes, ont été développés. On distingue en général les solveurs génériques capables de résoudre plus ou moins efficacement n'importe quel problème de satisfaction de contraintes (à partir du moment ou celui-ci est correctement défini) des solveurs spécifiques, intégrant de nombreux modules dédiés à des types de problèmes ou de contraintes particuliers. Ces derniers permettent une résolution plus efficace puisqu'ils prennent en compte les spécificités du problème à résoudre ainsi que le domaine d'application. Bien sûr en contrepartie, ils nécessitent un effort de développement très important et un paramétrage fin (identifier les modules adéquats, ajuster les paramètres de réglage, ...).

Pour résoudre un problème de satisfaction de contraintes, une approche consiste à alterner des étapes de recherche et des étapes de filtrage. Plus précisément, les phases de filtrage permettent d'effectuer des inférences ou déductions alors que les phases de recherche tentent de construire une solution en étendant itérativement une solution partielle également appelée instanciation partielle. Différentes techniques de filtrage plus ou moins sophistiquées peuvent être utilisées, comme par exemple la suppression de valeurs ne pouvant apparaître dans aucune solution. En général on recherche le meilleur compromis entre la puissance de filtrage (c'est-à-dire le nombre de déductions effectuées) et le temps nécessaire pour l'établir. Quand le filtrage a atteint ses limites, c'est-à-dire que l'on a atteint un point fixe à partir duquel plus aucune déduction n'est possible, on reprend alors la phase d'exploration. Celle-ci consiste à étendre la solution partielle à l'aide d'une nouvelle décision, i.e. une variable et une valeur pour cette variable, choisie par une

"heuristique". Cette décision, sélectionnée en priorité par l'heuristique, est supposée être la plus pertinente dans la mesure où elle permet d'accélérer la découverte d'une solution ou la preuve d'inconsistance. Malheureusement ces choix ne sont pas toujours fiables et l'on rencontre souvent des échecs pendant la recherche. Dans ce cas la solution partielle ne peut être étendue et il faut rebrousser chemin, c'est-à-dire défaire des choix effectués au préalable.

2 Problématique

Lorsqu'un échec apparaît après avoir pris une décision, au lieu de remettre en cause systématiquement le dernier choix effectué, il peut être intéressant de connaître les raisons précises de cet échec. Ceci permet de corriger directement les mauvais choix qui ont été faits, en évitant ainsi de répéter plusieurs fois les mêmes erreurs et donc de rencontrer de nombreuses fois les mêmes échecs. L'analyse des échecs permet de remonter aux sources des conflits en cherchant alors une raison suffisante pour les expliquer. Cette analyse permet notamment d'identifier des espaces de recherche dans lesquels aucune solution ne peut apparaître. Bien sûr, cette analyse a un coût et on cherche toujours à obtenir le meilleur compromis entre le temps consacré à l'analyse des conflits et le degré de précision des explications souhaitées.

Alors que dans le cadre de problèmes générés aléatoirement, analyser les raisons des échecs n'est pas vraiment pertinent, cette approche s'avère efficace sur les problèmes issus du monde réel. En effet ces problèmes, qui sont souvent des problèmes de grande taille, représentent des applications pratiques qui comportent bien souvent une structure interne. Certaines parties du problème sont plus contraintes que d'autres et on peut identifier au sein des problèmes des parties difficiles et d'autres très faciles. Par exemple, considérons un problème de logistique où un transporteur doit déposer des marchandises dans différentes communes tout en minimisant la distance parcourue. Si une ville n'est desservie que par une ou deux routes, le choix de la bonne route et les possibilités de se tromper sont limités. Cela correspond alors à une partie facile du problème, par opposition à une zone où les villes sont fortement connectées entres elles et où le nombre de chemins différents est important. Il est alors possible d'exploiter cette structure pour accélérer la résolution du problème, en concentrant par exemple la recherche sur les parties difficiles des problèmes.

Dans le contexte du problème de satisfaisabilité booléenne (SAT), l'arrivée dans les années 1990 de nouvelles instances encodant des applications pratiques a permis un regain d'intérêt pour ces méthodes basées sur les explications des échecs. Aujourd'hui les récents progrès effectués en SAT sont dûs principalement à une analyse précise et efficace des conflits. Dans les compétitions de prouveurs SAT, beaucoup de solveurs utilisent un mécanisme d'analyse des échecs. Ils sont d'ailleurs connus sous le nom de solveurs de type "Conflict Driven Clause Learning". En SAT, pour obtenir un solveur efficace, l'utilisation d'une telle approche semble même pour le moment incontournable. Dans le domaine de la satisfaction de contraintes, ces techniques basées sur les échecs n'ont pas encore "fait leurs preuves". Ceci est notamment dû au fait que peu d'instances réelles sont disponibles dans la communauté CSP et comparer l'efficacité de ces méthodes de résolution basées sur les échecs sur des problèmes aléatoires n'est pas vraiment pertinent.

3 Contributions et organisation de la thèse

Le contexte de ce travail est introduit dans les chapitres 1 et 2. Le premier chapitre de ce document présente un modèle de représentation des problèmes de satisfaction de contraintes : les réseaux de contraintes. Différentes approches permettant la simplification de ces réseaux de

contraintes sont alors abordées. Nous décrivons ensuite en détail les principales méthodes de résolution. La plupart d'entres-elles sont sensibles au phénomène de thrashing, qui introduit du "bruit" dans la résolution pénalisant fortement la recherche. Nous discutons alors des effets du thrashing et le chapitre 2 se focalise sur des techniques permettant d'atténuer son impact, principalement basées sur l'analyse des conflits. On y présente notamment différentes techniques permettant d'identifier les explications (nogoods) ou encore de remonter directement aux origines (retours-arrière intelligents) des échecs.

Nous proposons ensuite dans cette thèse plusieurs techniques d'analyse et d'exploitation des conflits qui permettent de réduire de façon significative l'effort de recherche nécessaire à la résolution de problèmes de satisfaction de contraintes.

Dans le chapitre 3, nous montrons tout d'abord qu'il est possible de guider efficacement la recherche en se basant sur les derniers conflits rencontrés. Tout en étant une technique prospective, i.e. une technique dont le but est de se diriger rapidement vers une solution, cette approche permet de lutter contre le thrashing en orientant la recherche sur l'origine du dernier conflit rencontré. Plus précisément, la variable impliquée dans le dernier conflit est sélectionnée en priorité par l'heuristique tant que celle-ci mène directement à un échec. Dans un second temps cette technique est généralisée pour identifier non plus une raison de l'échec mais un ensemble de raisons. Brièvement, cela correspond à identifier un ensemble de variables incompatibles (nogoods) tel qu'il n'est pas possible de le "traverser", c'est-à-dire qu'il n'est pas possible de trouver au moins un ensemble de valeurs pour ces variables ne provoquant pas d'échec. Nous discutons finalement de l'adaptation de la méthode présentée ici pour la résolution de problèmes de planification.

Pour lutter contre les effets du thrashing, une solution possible consiste à effectuer des redémarrages. L'algorithme de recherche est relancé plusieurs fois pour diversifier la recherche, ce qui permet d'explorer différentes parties de l'espace de recherche en introduisant par exemple un caractère aléatoire dans l'heuristique. Cependant, dans ce contexte, les mêmes portions de l'espace de recherche peuvent être explorées inutilement plusieurs fois, ce qui représente un défaut majeur. Nous apportons dans le chapitre 4 une solution à cet inconvénient en calculant a posteriori, i.e. à chaque redémarrage, les explications des échecs rencontrés au cours de la recherche sous la forme d'un ensemble de nogoods. Ces explications sont enregistrées dans une base dont l'exploitation évite que des situations identiques ne réapparaissent grâce à un mécanisme de déduction basé sur l'utilisation d'une structure de données paresseuse : les watched literals. Nous nous intéressons également à la minimisation de certaines de ces explications, dans la mesure où elles permettent même d'aller plus loin que l'identification stricte de l'espace de recherche déjà exploré.

Finalement nous mettons en évidence dans le chapitre 5 le fait que des situations (i.e. des états de la recherche à un instant donné) similaires peuvent apparaître au cours de la recherche telles qu'il est inutile de considérer plusieurs occurrences de ces situations. Nous introduisons des opérateurs originaux permettant l'identification de ces situations jugées équivalentes. Ces opérateurs sont basés sur la détection d'états partiels inconsistants. Brièvement, un état partiel inconsistant représente un sous-ensemble de l'état de la recherche à un instant donné, suffisant pour démontrer l'absence de solution à partir de cet état. Si plus tard l'état courant de la recherche est équivalent à un état partiel inconsistant précédemment enregistré, il n'est alors plus nécessaire de poursuivre la recherche à partir de cet état courant. Une autre forme de détection basée sur la dominance (inclusion) d'un état dans un état partiel inconsistant est également envisagée. Nous montrons également que ces opérateurs permettent de détecter et casser dynamiquement certaines formes de symétries.

Table des figures

Première partie

Etat de l'art

1

CSP et résolution

La programmation par contraintes offre un cadre théorique et pratique pour la représentation et la résolution des problèmes CSP (problème de satisfaction de contraintes ou "Constraint Satisfaction Problem" en anglais). Le terme CSP désigne l'ensemble des problèmes définis par des contraintes, et pour lesquels l'objectif consiste à trouver une (ou plusieurs) solutions tout en respectant les contraintes. Ce cadre permet de regrouper sous une même étiquette de nombreux problèmes issus du monde réel tels que les problèmes d'emploi du temps, d'allocation de ressources, de conception de circuits, d'ordonnancement... Les réseaux de contraintes sont les fondations sur lesquelles repose la représentation des problèmes CSP. Ils permettent la formalisation des problèmes de satisfaction de contraintes par une représentation à base de variables et de contraintes reliant ces variables. Les algorithmes dédiés à la résolution des problèmes CSP sont appelés des "solveurs" de contraintes. On peut les appliquer à la résolution de problèmes réels qui à l'origine semblent pourtant bien éloignés les uns des autres. Ainsi la même technique de résolution peut tout à fait être utilisée pour résoudre un problème d'emploi du temps et un problème de conception de circuits.

Dans ce chapitre, l'objectif est de fixer le contexte dans lequel se situe ce travail. Après avoir introduit dans un premier temps les définitions fondamentales associées aux réseaux de contraintes (et aux CSP), nous présenterons ensuite différentes formes de consistances. Si dans le contexte de la programmation par contraintes, les réseaux de contraintes sont les fondations alors les consistances sont le ciment. Elles permettent des simplifications dans les réseaux de contraintes, et accélèrent ainsi la recherche de solutions. Nous aborderons également la résolution pratique des CSP en présentant différentes techniques de résolution.

1.1 Réseaux de contraintes

1.1.1 Définition

Les réseaux de contraintes permettent une représentation des problèmes de satisfaction de contraintes. Ils reposent sur les notions de variables et de contraintes reliant ces variables.

Définition 1 (variable)**.** *Une variable discrète X représente une valeur choisie dans un domaine fini noté $dom(X)$.*

On associe à chaque variable un domaine fini qui représente l'ensemble des valeurs qui peuvent être affectées à X. Une variable peut par exemple prendre ses valeurs dans un sous-ensemble des entiers relatifs (correspondant à son domaine). Par abus de langage, lorsque le domaine d'une

variable est réduit à un singleton (soit après une assignation, soit après des réductions successives du domaine), on utilise alors le terme de variable singleton.

Une instanciation d'un ensemble S de variables associe à chaque variable de S une valeur de son domaine.

Définition 2 (instanciation). *Une instanciation t d'un ensemble $\{X_1, ..., X_q\}$ de variables est un ensemble $\{(X_i, v_i) \mid i \in [1, q] \wedge v_i \in dom(X_i)\}$.*

La valeur v_i affectée à X_i dans t sera notée par $t[X_i]$ et on parle alors de la valeur v_i assignée à la variable X_i.

Une contrainte est un objet impliquant un ensemble de variables, appelée *portée* et auquel on associe une relation représentant l'ensemble des instanciations autorisées pour les variables de sa portée.

Définition 3 (contrainte). *Une contrainte C implique un ensemble de variables, appelé portée et noté $scp(C)$, et représente une relation, notée $rel(C)$, définie par l'ensemble des instanciations autorisées pour les variables de $scp(C)$.*

Le nombre de variables impliquées dans une contrainte C, i.e $|scp(C)|$, détermine son arité.
- Une contrainte est unaire si son arité est égale à un ;
- Une contrainte est binaire si son arité est égale à deux ;
- ...
- De manière générale une contrainte est n-aire si son arité est égale à n.

Par exemple la contrainte $C_1 : X_1 \leq 3$ est une contrainte unaire puisqu'elle n'implique qu'une seule variable dans sa portée ($scp(C_1) = \{X_1\}$) tandis que la contrainte $C_2 : X_1 \neq X_2$ est une contrainte binaire ($scp(C_2) = \{X_1, X_2\}$).

La relation $rel(C)$ (associée à une contrainte C) peut également être définie comme un ensemble de n-uplets de valeurs si l'on considère un ordre sur les variables de sa portée. Cet ordre arbitraire permet de définir la relation $rel(C)$ comme un sous-ensemble du produit cartésien des domaines des variables appartenant à $scp(C)$, représentant alors les n-uplets autorisés par la contrainte C. Par la suite et pour faciliter la lecture, on utilisera en fonction du contexte l'une de ces deux définitions. La contrainte C_2 de l'exemple précédent est définie par une relation de différence entre les variables X_1 et X_2. En supposant que $dom(X_1) = dom(X_2) = \{0, 1\}$ et en considérant l'ordre arbitraire sur les variables $X_1 < X_2$, les deux représentations de la contrainte C_2 suivantes sont équivalentes :
- $\{\{(X_1, 0)(X_2, 1)\}, \{(X_1, 1)(X_2, 0)\}\}$;
- $< X_1, X_2 > \in \{(0, 1), (1, 0)\}$.

Un réseau de contraintes est défini par un ensemble fini de variables et un ensemble fini de contraintes qui établissent certaines dépendances entre les variables.

Définition 4 (réseau de contraintes). *Un réseau de contraintes discret (CN pour Constraint Network) P est un couple $(\mathscr{X}, \mathscr{C})$ où :*
- *$\mathscr{X} = \{X_1, \ldots, X_n\}$ est un ensemble fini de n variables discrètes ;*
- *$\mathscr{C} = \{C_1, \ldots, C_e\}$ est un ensemble fini de e contraintes.*

L'ensemble des variables de P sera noté $vars(P)$ et chaque contrainte $C \in \mathscr{C}$ implique dans sa portée un sous-ensemble de variables de $\mathscr{X}$. Une instanciation t est dite *partielle* par rapport à P lorsqu'elle implique un sous-ensemble strict de variables de $vars(P)$. t est une instanciation *totale* par rapport à P lorsque toutes les variables de $vars(P)$ apparaissent dans t.

Etant donné un réseau $P = (\mathscr{X}, \mathscr{C})$, le domaine de X dans P est noté $dom^P(X)$ et la relation associée à une contrainte C de P est notée $rel^P(C)$. Lorsque le contexte le permet, pour alléger les notations, on notera $dom(X)$ le domaine de X dans P et $rel(C)$ la relation associée à C dans P.

Etant donné un réseau de contraintes P, on note :
- n le nombre de variables du réseau P ;
- d la taille du plus grand domaine ;
- e le nombre de contraintes du réseau P ;
- r l'arité maximale des contraintes de P.

La représentation des contraintes

Les contraintes peuvent être exprimées soit en intension soit en extension. Lorsqu'on définit une contrainte en intension, on utilise des opérateurs mathématiques connus (relation d'ordre, d'égalité ou de différence, somme algébrique...) pour exprimer la relation entre les variables de sa portée. Dans ce cas, la relation est définie par une formule impliquant des *prédicats* (arithmétiques, booléens...). En extension, la contrainte est représentée sous la forme d'une liste de n-uplets acceptables pour cette relation, i.e une liste d'instanciations autorisées pour les variables de sa portée. La contrainte peut également être représentée sous la forme d'une liste de n-uplets interdits pour la relation. Elle exprime alors la liste des conflits impliquant les variables de sa portée. La sémantique (encodage des conflits ou des supports) doit être spécifiée au moment de la déclaration de la contrainte. Récemment dans [Boussemart *et al.*, 2005], un format de représentation, basé sur la norme XML, a été proposé pour standardiser la description des problèmes CSP. Ce format admet la représentation des contraintes aussi bien en intension (pour un certain nombre de prédicats) qu'en extension (selon la sémantique supports ou conflits).

Une contrainte est satisfaite par une instanciation des variables de sa portée, lorsque cette instanciation est une instanciation autorisée par la relation associée à cette contrainte.

Définition 5 (contrainte satisfaite). *Soit $P = (\mathscr{X}, \mathscr{C})$ un réseau de contraintes. Une contrainte $C \in \mathscr{C}$ est satisfaite par une instanciation t d'un ensemble S de variables de P si et seulement si $scp(C) \subseteq S$ et $\{(X, v) \mid X \in scp(C) \wedge t[X] = v\} \in rel(C)$.*

Lorsque toutes les instanciations construites à partir du domaine des variables de la portée d'une contrainte satisfont celle-ci, la contrainte est dite universelle.

Définition 6 (contrainte universelle). *Soit $P = (\mathscr{X}, \mathscr{C})$ un réseau de contraintes. Une contrainte $C \in \mathscr{C}$ est universelle ssi C est satisfaite par toutes les instanciations de $scp(C)$.*

Si toutes les contraintes d'un réseau de contraintes P sont binaires, P est lui-même dit binaire. Réciproquement, lorsqu'au moins une contrainte est d'arité supérieure à deux, le réseau de contraintes est dit non-binaire. Un réseau de contraintes non-binaire peut toujours être transformé en un réseau de contraintes binaire en appliquant une technique de binarisation. Les deux méthodes les plus utilisées sont l'encodage du graphe dual (dual graph encoding) [Dechter et Pearl, 1988] et l'encodage des variables cachées (hidden variable encoding) [Rossi *et al.*, 1990].

Exemple d'un réseau de contraintes

Soit le réseau de contraintes $P = (\mathscr{X}, \mathscr{C})$ suivant :
- $\mathscr{X} = \{X_0, X_1, X_2, X_3\}$ avec $dom(X_0) = dom(X_1) = dom(X_2) = dom(X_3) = \{0, 1\}$;

- $\mathscr{C} = \left\{ \begin{array}{ll} C_0 : & X_0 \neq X_1 \\ C_1 : & X_1 + X_2 = X_3 \\ C_2 : & < X_0, X_4 > \notin \{(0,0),(1,1)\} \end{array} \right\}$.

Ce réseau de contraintes comporte quatre variables X_0, X_1, X_2, X_3 dont chacune peut prendre deux valeurs 0 ou 1. De plus, ces variables doivent respecter deux contraintes (C_0 et C_1) exprimées en intension : $X_0 \neq X_1, X_1 + X_2 = X_3$ ainsi qu'une contrainte (C_2) exprimée en extension, représentant les conflits entre les variables X_0 et X_4 : l'ensemble des instanciations ne satisfaisant pas C_2 est $\{\{(X_0,0),(X_4,0)\}, \{(X_0,1),(X_4,1)\}\}$. Remarquons que la contrainte C_2 exprime également une contrainte de différence entre les variables X_4 et X_0. On aurait donc pu exprimer cette contrainte de manière équivalente en intension : $X_0 \neq X_4$.

$vars(P)$ représente l'ensemble des variables $\{X_0, X_1, X_2, X_3\}$ du réseau P et la portée de la contrainte C_0 représente les variables impliquées par cette contrainte, $scp(C_0) = \{X_0, X_1\}$. L'arité de la contrainte C_0 est donc égale à 2.

Ensemble solution

Etant donné un réseau de contraintes P, une instanciation t d'un sous-ensemble S de variables de P est dite consistante si toutes les contraintes C dont la portée est incluse dans S sont satisfaites.

Définition 7 (instanciation consistante)**.** *Soit $P = (\mathscr{X}, \mathscr{C})$ un réseau de contraintes et $S \subseteq \mathscr{X}$ un sous-ensemble de variables. Une instanciation t de S est consistante si et seulement si $\forall C \in \mathscr{C}$ telle que $scp(C) \subseteq S, C$ est satisfaite par t.*

Une solution de P est une instanciation de $vars(P)$ qui satisfait toutes les contraintes du réseau. Autrement dit, une solution de P est une instanciation totale consistante.

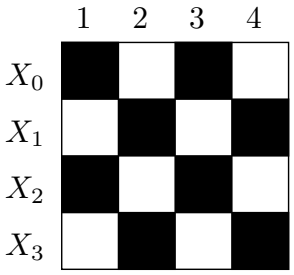
Définition 8 (solution d'un réseau de contraintes)**.** *Soit $P = (\mathscr{X}, \mathscr{C})$ un réseau de contraintes. Une instanciation t de $vars(P)$ est une solution de P si et seulement si $\forall C \in \mathscr{C}, C$ est satisfaite par t.*

L'ensemble de toutes les solutions de P est noté $sol(P)$. Un réseau de contraintes est satisfaisable si et seulement si il admet au moins une solution ($sol(P) \neq \emptyset$). Si pour un réseau de contraintes P, le domaine de l'une de ses variables est vide ou si ce réseau comporte une relation vide (c'est-à-dire n'autorisant aucun n-uplet) alors aucune solution ne peut satisfaire P, et celui-ci est clairement insatisfaisable, noté $P = \bot$.

1.1.2 Un exemple de représentation : Le problème des n-reines

Envisageons maintenant pour illustrer notre propos, le problème académique des *n-reines*. Ce problème consiste à disposer n reines sur un échiquier de taille nxn de manière à ce qu'aucune d'entre elles ne soit en prise avec les autres. Deux reines sont en prise si elle se trouvent sur une même ligne, une même colonne ou une même diagonale de l'échiquier.

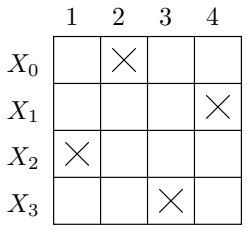
Le problème des 4-reines est une instance particulière du problème général des n-reines, où l'on doit placer 4 reines sur un échiquier de taille 4x4. Puisque par définition deux reines ne peuvent pas être placées sur la même ligne, il y aura exactement une reine par ligne de l'échiquier (il faut placer quatre reines sur un échiquier comportant quatre lignes). Le problème se ramène donc à déterminer sur quelle colonne on doit placer la reine de la ligne i. Ce problème peut être modélisé par un réseau de contraintes P avec quatre variables L_0, L_1, L_2, L_3 représentant le



$P = (\mathscr{X}, \mathscr{C})$:

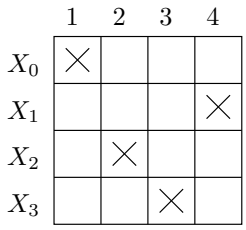
- $\mathscr{X} = \{X_0, X_1, X_2, X_3\}$ avec $dom(X_0) = dom(X_1) = dom(X_2) = dom(X_3) = \{1, 2, 3, 4\}$;
- $$\mathscr{C} = \left\{ \begin{array}{lllll} C_0 : & X_0 \neq X_1 & \wedge & |X_1 - X_0| \neq 1 \\ C_1 : & X_0 \neq X_2 & \wedge & |X_2 - X_0| \neq 2 \\ C_2 : & X_0 \neq X_3 & \wedge & |X_3 - X_0| \neq 3 \\ C_3 : & X_1 \neq X_2 & \wedge & |X_2 - X_1| \neq 1 \\ C_4 : & X_1 \neq X_3 & \wedge & |X_3 - X_1| \neq 2 \\ C_5 : & X_2 \neq X_3 & \wedge & |X_3 - X_2| \neq 1 \end{array} \right\}.$$

FIG. 1.1 – Le problème des 4-reines et le réseau de contraintes associé



$t_1 = \{(X_0, 2), (X_1, 4), (X_2, 1), (X_3, 3)\}$

(a)



$t_2 = \{(X_0, 1), (X_1, 4), (X_2, 2), (X_3, 3)\}$

(b)

FIG. 1.2 – Deux instanciations totales du problème des 4-reines

numéro de la ligne, et dont le domaine initialement égal à $\{1, \ldots, 4\}$ représente les numéros des colonnes sur lesquelles on peut disposer les reines.

Les contraintes interdisent de placer deux reines sur la même ligne, sur la même colonne et sur la même diagonale. Tout d'abord, avec cette modélisation, deux reines ne peuvent pas être disposées sur la même ligne puisqu'une variable X_i représente la reine disposée sur la ligne i. Par construction, une seule valeur peut être affectée à une variable donc il ne peut pas y avoir plus d'une reine par ligne. Chaque reine doit être placée sur une colonne différente de celles où sont placées les autres reines. Ceci est exprimé par les contraintes de différence suivantes : $X_0 \neq X_1, X_0 \neq X_2, X_0 \neq X_3, X_1 \neq X_2, X_1 \neq X_3, X_2 \neq X_3$. Finalement on interdit de disposer deux reines sur la même diagonale : $|X_1 - X_0| \neq 1, |X_2 - X_1| \neq 1, |X_3 - X_2| \neq 1, |X_2 - X_0| \neq 2, |X_3 - X_1| \neq 2, |X_3 - X_0| \neq 3$. Ici, le réseau de contraintes a été normalisé, c'est-à-dire que les contraintes de même portée ont été fusionnées.

La figure 1.1 illustre le problème des 4-reines et le réseau de contraintes associé. La figure 1.2 illustre deux instanciations t_1 et t_2 des variables du problème des 4-reines. L'instanciation t_1 (représentée en (a)) est une solution pour cette modélisation puisqu'elle satisfait toutes les contraintes du réseau. L'instanciation t_2 (représentée en (b)) n'est pas une solution. En effet, la contrainte C_5 est violée : la valeur 2 assignée à la reine X_2 et la valeur 3 assignée à la reine X_3 placent les deux reines sur une même diagonale.

Un réseau de contraintes peut être représenté sous la forme d'un hypergraphe de contraintes (ou simplement graphe de contraintes dans le cas d'un réseau binaire). Chaque variable d'un réseau de contraintes donné représente alors un sommet du graphe et chaque contrainte une hyper-arête reliant les variables qu'elles impliquent. Cette représentation graphique est également appelée macro-structure du réseau de contraintes puisqu'on ne représente que les variables et les

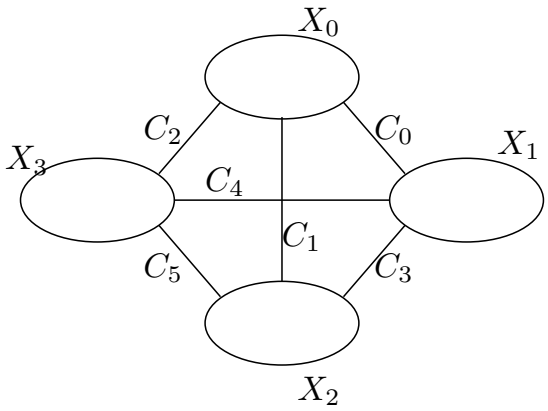


FIG. 1.3 – Graphe de contraintes associé au problème des 4-reines

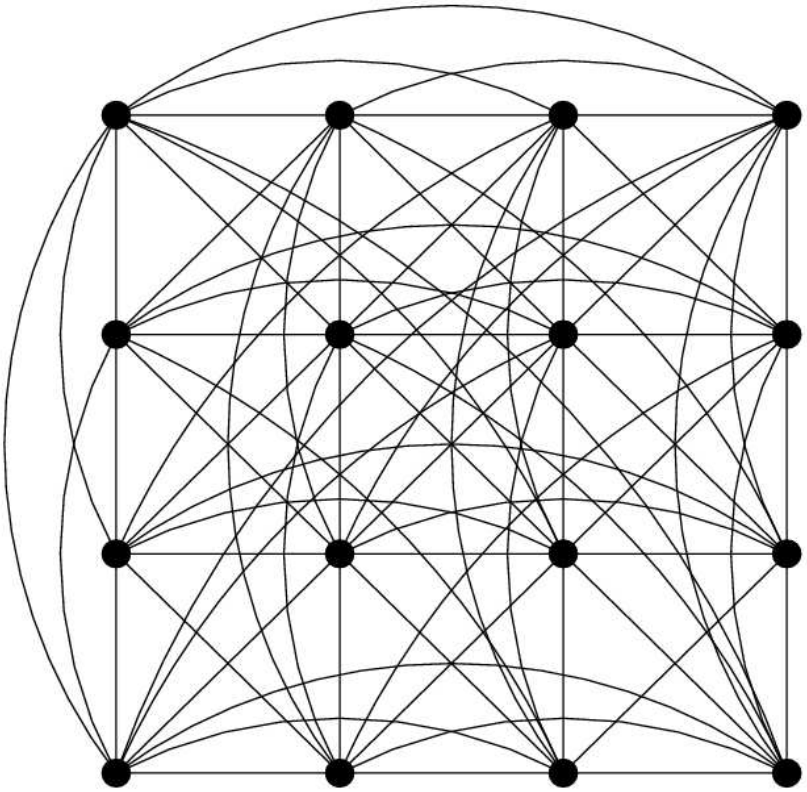


FIG. 1.4 – Micro-structure complémentaire du problème des 4-reines

contraintes reliant ces variables. Les domaines des variables et les n-uplets autorisés pour les contraintes du réseau ne sont pas représentés. La figure 1.3 représente le graphe construit à partir du réseau de contraintes modélisant le problème des 4-reines. On peut noter que ce réseau de contraintes est binaire puisque chaque contrainte n'implique que deux variables. Ce graphe est aussi complet puisque tous les sommets sont reliés deux à deux par une arête.

Un réseau de contraintes peut également être représenté par sa micro-structure. Dans ce cas, on construit un hypergraphe dit de compatibilité où un sommet représente un couple $(variable, valeur)$, et une hyper-arête un n-uplet autorisé par l'une des contraintes du problème. Un tel hypergraphe permet une représentation plus précise du problème, même si l'hypergraphe obtenu est beaucoup plus grand (en terme de nombre de sommets et d'hyper-arêtes). On utilise la micro-structure des réseaux de contraintes pour notamment détecter (et éliminer) des symétries au niveau du problème [Fahle *et al.*, 2001, Gent et Smith, 2000] ou encore dans le développement de méthodes de décomposition de problèmes [Chmeiss *et al.*, 2003]. On peut également exploiter le graphe d'incompatibilité plutôt que le graphe de compatibilité. Il s'agit de la micro-structure complémentaire, c'est-à-dire le graphe dual où une arête n'encode pas un support mais un conflit entre des valeurs. La figure 1.4 représente le graphe d'incompatibilité associé au problème des 4-reines.

1.2 Le problème de satisfaction de contraintes et sa résolution

Les réseaux de contraintes permettent de représenter les problèmes de satisfaction de contraintes (CSP pour Constraint Satisfaction Problem). Déterminer si un réseau de contraintes donné est satisfaisable ou non, est un problème NP-complet (Non-déterministe Polynomial). Cette classe regroupe les problèmes de décision pour lesquels la réponse peut être décidée par un algorithme non-déterministe en un temps polynomial par rapport à la taille du problème. Autrement dit, cette classe regroupe les problèmes qui n'ont pas d'algorithme de résolution polynomial (sauf si $P = NP$), mais pour lesquels on est capable de vérifier en un temps polynomial si une proposition de solution est ou non une solution du problème.

Une instance CSP peut être représentée par un réseau de contraintes, et la résoudre consiste à trouver une (ou plusieurs) solution(s) à ce réseau de contraintes ou alors à prouver son insatisfaisabilité. Pour résoudre une instance CSP, on peut modifier le réseau de contraintes en utilisant des méthodes d'inférence ou de recherche. Très souvent, les domaines des variables sont réduits en supprimant des valeurs[1] qui ne peuvent apparaître dans aucune solution (dites valeurs inconsistantes). En effet il est possible de filtrer les domaines en tenant compte de certaines propriétés du réseau de contraintes.

1.2.1 La propagation de contraintes (filtrage)

Certaines valeurs du domaine des variables d'un réseau de contraintes peuvent n'apparaître dans aucune solution. Si ces valeurs sont assignées aux variables, celles-ci mèneront donc forcément plus ou moins rapidement à un échec. L'efficacité d'un algorithme de recherche peut être améliorée en filtrant le domaine des variables et en éliminant les valeurs inconsistantes, c'est-à-dire ne menant à aucune solution. Le domaine des variables est ainsi réduit et les valeurs supprimées ne pourront jamais être affectées. Pour effectuer cette propagation de contraintes (également appelée filtrage) on applique sur le réseau de contraintes un opérateur d'inférence établissant une consistance ϕ, $\phi(P)$ représentant le réseau de contraintes obtenu après application de cet opérateur sur le réseau P.

[1] Dans un réseau $P = (\mathscr{X}, \mathscr{C})$, lors de la suppression d'une valeur b de $dom(X)$, on suppose que l'on supprime également $\forall C \in \mathscr{C} \mid X \in scp(C)$, les instanciations partielles $t \in rel(C) \mid (X, b) \in t$. Par la suite, pour simplifier les algorithmes, on omettra de mentionner la modification des relations lors des suppressions de valeurs. Notons qu'en pratique il n'est pas forcément nécessaire d'effectuer réellement ces modifications de relations car la contrainte C ne pourra jamais être satisfaite par l'un de ces tuples. En résumé, le réseau de contraintes P est dit "embarqué" [Bessiere, 2006].

La notion de consistance d'arc

La consistance d'arc (AC pour Arc Consistency), ou 2-consistance est l'une des propriétés fondamentales des CSPs qui s'applique aux contraintes binaires. La notion de base est celle de *support*.

Définition 9 (valeur arc-consistante). *Une valeur a d'une variable X est arc-consistante pour une contrainte binaire C telle que $scp(C) = \{X, Y\}$ ssi $\exists b \in dom(Y) \mid \{(X, a)(Y, b)\} \in rel(C)$. La valeur b est alors appelée support de a pour la contrainte C.*

Lorsque toutes les valeurs du domaine de X et du domaine de Y sont arc-consistantes, la contrainte binaire C (impliquant X et Y dans sa portée) est elle même arc-consistante.

Définition 10 (contrainte arc-consistante). *Soit $P = (\mathscr{X}, \mathscr{C})$ un CN, et $C \in \mathscr{C}$ telle que $scp(C) = \{X, Y\}$. La contrainte binaire C est arc-consistante si et seulement si $\forall X' \in scp(C), dom(X') \neq \emptyset$ et $\forall a \in dom(X), \exists b \in dom(Y)$ tel que $\{(X, a), (Y, b)\} \in rel(C)$ et $\forall b \in dom(Y), \exists a \in dom(X)$ tel que $\{(X, a), (Y, b)\} \in rel(C)$.*

La notion de consistance d'arc peut se généraliser à un réseau de contraintes.

Définition 11 (réseau arc-consistant). *Un réseau de contraintes $P = (\mathscr{X}, \mathscr{C})$ est arc-consistant si toutes les contraintes binaires de $\mathscr{C}$ sont arc-consistantes.*

Suppression de valeurs par consistance d'arc

Pour rendre une contrainte arc-consistante, il suffit de supprimer pour chaque variable définie dans sa portée, les valeurs n'ayant pas de support dans le domaine de l'autre variable de sa portée. Les suppressions s'effectuent itérativement jusqu'à obtenir un point fixe, c'est-à-dire jusqu'à ce que toutes les contraintes du réseau soient arc-consistantes et qu'aucune valeur ne puisse être encore supprimée. Si au cours de ce processus, le domaine d'une variable devient vide (par la suppression successive de toutes les valeurs de ce domaine), la contrainte ne pourra jamais être satisfaite et le réseau de contraintes est clairement insatisfaisable.

La figure 1.5 illustre la suppression de valeurs par consistance d'arc sur deux réseaux de contraintes P_1 et P_2. Les réseaux de contraintes P_1 et P_2 sont composés de trois variables X_0, X_1 et X_2 dont le domaine est égal à $\{0, 1\}$.

Le réseau P_1 est composé de trois contraintes :
- $C_0 : X_0 \neq X_1$;
- $C_1 : X_0 = X_2$;
- $C_2 : X_1 \geq X_2$.

Le graphe de contraintes associé au réseau P_1 est représenté en (a). P_1 est un réseau de contraintes arc-consistant puisque chaque contrainte de ce réseau est elle même arc-consistante. Aucune valeur du domaine des variables de P_1 ne peut donc être supprimée, puisque chaque valeur de chaque variable possède au moins un support dans le domaine des autres variables, reliées à celle-ci par une contrainte.

Le graphe de contraintes associé au réseau P_2 est représenté en (b). Ce réseau est également composé de trois contraintes :
- $C_0 : X_0 \neq X_1$;
- $C_1 : X_0 = X_2$;
- $C_2 : (X_1 + 1) \times X_2 = 0$.

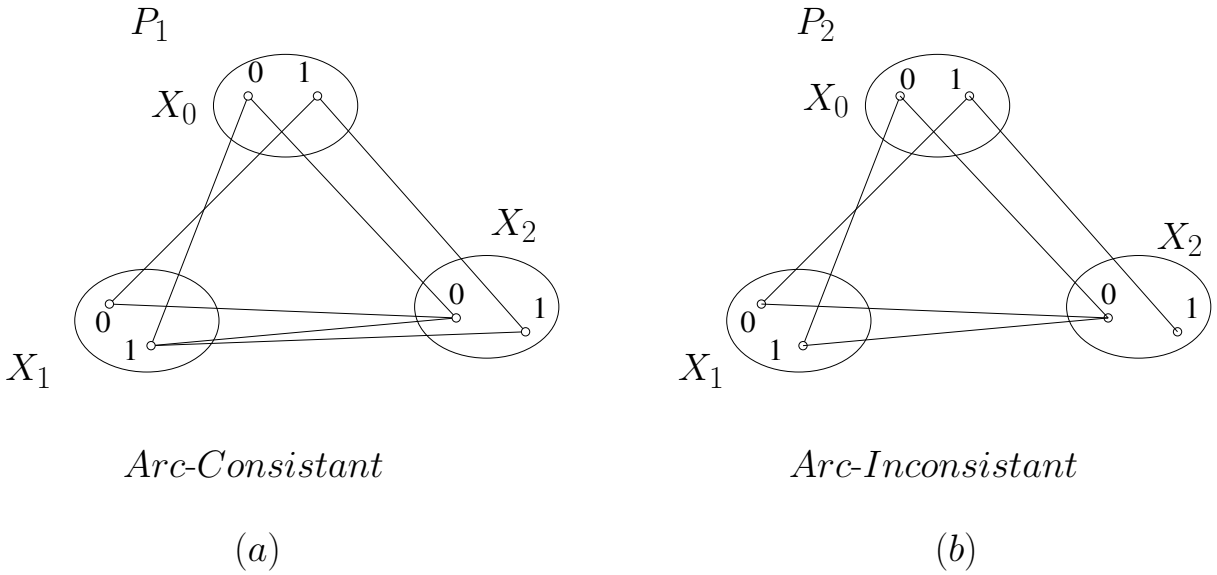


FIG. 1.5 – Suppression de valeurs par consistance d'arc

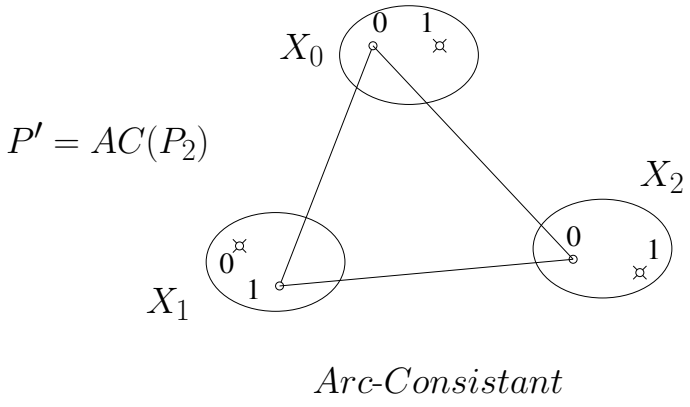


FIG. 1.6 – Consistance d'arc et processus de propagation

La contrainte C_2 n'est pas arc-consistante puisque la valeur 1 du domaine de la variable X_2 n'a pas de support dans le domaine de la variable X_1, i.e il n'existe aucune valeur compatible dans le domaine de X_1 avec la valeur 1 de X_2. Cette contrainte n'étant pas arc-consistante, le réseau P_2 n'est par conséquent pas arc-consistant.

Pour établir la consistance d'arc du réseau P_2, il faut rendre la contrainte C_2 arc-consistante. Pour cela la valeur 1 du domaine de la variable X_2 est éliminée. La suppression de cette valeur rend la contrainte C_1 à son tour arc-inconsistante (la valeur 1 de X_0 n'a plus de support dans X_2). Un raisonnement similaire est à l'origine de la suppression de la valeur 1 du domaine de la variable X_0. On réitère le processus jusqu'à obtenir un point fixe, i.e. lorsque toutes les contraintes du réseau sont arc-consistantes. Cette cascade de suppression de valeurs est appelée *processus de propagation de contraintes* ou filtrage. Le réseau de contraintes obtenu après filtrage est représenté en figure 1.6. Ce réseau est maintenant arc-consistant. La consistance d'arc est une consistance locale dans la mesure ou elle ne garantit pas l'existence de solutions.

Comme indiqué plus haut, on note ϕ une consistance donnée et $P' = \phi(P)$ le réseau de

contrainte P' obtenu après avoir établi ϕ sur le réseau P. Par exemple, $P' = AC(P)$ est le réseau de contraintes P' obtenu après avoir établi la consistance d'arc sur le réseau P. On peut introduire une relation d'ordre sur les réseaux de contraintes P et P' selon la définition suivante.

Définition 12 (ordre sur les réseaux de contraintes)**.** *Soit P et P' deux réseaux de contraintes définis sur le même ensemble de variables $\mathscr{X}$ et de contraintes $\mathscr{C}$. $P' \preceq P$ si et seulement si $\forall X \in \mathscr{X}, dom^{P'}(X) \subseteq dom^{P}(X)$ et $\forall C \in \mathscr{C}, rel^{P'}(C) \subseteq rel^{P}(C)$.*

On appelle un sous-réseau de P, un réseau P' tel que $P' \preceq P$.

Lorsqu'on établit la consistance d'arc, si au cours du processus de propagation, le domaine d'une variable devient vide, le réseau de contraintes est clairement insatisfaisable, noté $AC(P) = \bot$. Comme les contraintes et les domaines sont en nombre fini, le processus de propagation converge donc toujours soit vers une contradiction (c'est-à-dire $AC(P) = \bot$), soit vers un point fixe dès lors que toutes les déductions possibles (i.e. suppression de valeurs) ont été faites. Dans ce dernier cas, on obtient alors le plus grand sous-réseau arc-consistant équivalent à P en terme de solutions (également appelé fermeture par arc-consistance ou *AC-closure*(P)) tel que $AC(P) \preceq P$ et $sol(AC(P)) = sol(P)$.

Si un réseau de contraintes est arc-inconsistant, le processus de propagation va converger vers une contradiction, c'est-à-dire que le domaine de l'une des variables va devenir vide. En fonction de l'ordre dans lequel les valeurs sont éliminées au cours du filtrage (i.e. en fonction de l'ordre dans lequel les révisions sont effectuées – c.f. algorithme 1 –), l'événement "domaine vide" peut survenir sur différentes variables. L'inconsistance du réseau peut donc être détectée plus ou moins rapidement en fonction de l'ordre dans lequel les valeurs sont supprimées. On peut utiliser une heuristique [Boussemart *et al.*, 2004b](comme par exemple l'heuristique qui minimise la taille du domaine dans le cadre d'un algorithme de consistance d'arc à gros grain – c.f. section 1.2.1 –) pour guider le processus de propagation, i.e. déterminer l'ordre dans lequel on effectue les révisions. Si le réseau est arc-consistant, quelque soit l'ordre dans lequel les valeurs sont supprimées des domaines, on obtiendra le même point fixe et donc le même sous-réseau : la consistance d'arc est en cela confluente. Ceci résulte de l'une des propriétés de la consistance d'arc : la stabilité par union [Bessiere, 2006].

Les algorithmes de consistance d'arc

La propagation de contraintes permet de réduire les domaines des variables, dans le but d'accélérer le parcours ultérieur d'un arbre de recherche [Montanari, 1974]. De nombreux algorithmes ($AC3$ [Mackworth, 1977], $AC4$ [Mohr et Henderson, 1986], $AC6$ [Bessiere, 1994], $AC7$ [Bessiere *et al.*, 1999], $AC2001$ [Bessiere *et al.*, 2005], $AC3_d$ [van Dongen, 2002], $AC3^{rm}$ [Lecoutre et Hemery, 2007 ...) ont été proposés afin d'établir la consistance d'arc d'un réseau de contraintes. Chaque nouvelle version étant (en général) plus optimisée (et donc plus efficace) que la précédente.

Certains de ces algorithmes ont connu un grand succès (e.g. $AC2001/3.1$) et semblent plus attractifs que d'autres puisqu'ils possèdent une complexité temporelle optimale en $O(ed^2)$ tout en restant relativement simples à implémenter.

On distingue généralement les algorithmes à gros grain des algorithmes à grain fin. Dans un algorithme à gros grain la suppression d'une valeur pour une variable est propagée directement aux variables associées. Plus précisément, le principe est d'effectuer des révisions successives d'arcs, i.e. de couples (C, X) composés d'une contrainte C et d'une variable X appartenant à sa portée. Dans un algorithme à grain fin, la suppression d'une valeur pour une variable est propagée uniquement aux valeurs correspondantes des autres variables. Plus précisément, le principe est d'effectuer des révisions successives de valeurs, i.e. de triplets (C, X, a) composés d'un arc (C, X)

et d'une valeur a du domaine de X. Même si les algorithmes à grain fin effectuent moins de révisions inutiles (puisque le niveau de détail est plus fin), la complexité des structures de données nécessaires peut être pénalisante.

Algorithme 1 : *doAC*

Entrées : $P = (\mathscr{X}, \mathscr{C})$: CN
Sorties : Booléen

```
1 Q ← {(C, X) | C ∈ 𝒞 ∧ X ∈ scp(C)} ;
2 tant que Q ≠ ∅ faire
3 |   choisir puis éliminer (C, X) de Q ;
4 |   si revise(C,X) alors
5 |   |   si dom(X) = ∅ alors Retourner faux ;
6 |   |   Q ← Q ∪ {(C', Y) | C' ∈ 𝒞, C' ≠ C, Y ≠ X, {X, Y} ⊆ scp(C')} ;
7 |   fin
8 fin
9 retourner vrai ;
```

Algorithme 2 : *revise*-3

Entrées : C : Contrainte, X : Variable
Sorties : Booléen

```
1 Supprime=faux ;
2 Soit {X, Y} = scp(C) ;
3 pour chaque a ∈ dom(X) faire
4 |   si ∄w ∈ dom(Y) tel que {(X, a)(Y, w)} ∈ rel(C) alors
5 |   |   dom(X)=dom(X)\{a} ;
6 |   |   Supprime=vrai ;
7 |   fin
8 fin
9 retourner Supprime ;
```

Les algorithmes AC3 [Mackworth, 1977] et AC2001/3.1 [Bessiere *et al.*, 2005] L'algorithme 1 présente le fonctionnement d'un algorithme à gros grain, basé sur la révision des arcs (C, X). Initialement tous les arcs sont placés dans une queue de propagation Q et révisés à tour de rôle. Lorsque la révision d'un arc (C, X) est effective, c'est-à-dire lorsqu'au moins une valeur du domaine de X a été éliminée, la queue Q est mise à jour. La révision d'un arc est effectuée par un appel à la fonction *revise-X* associée à l'algorithme 1. L'algorithme se termine lorsqu'un domaine devient vide (le réseau de contraintes P passé en paramètre est alors prouvé inconsistant) ou que l'ensemble Q devient vide (et dans ce cas le réseau obtenu est arc-consistant).

La fonction *revise-X* (algorithme 2 associé à $AC3$ et algorithme 3 associé à $AC2001$) élimine les valeurs de X qui sont inconsistantes par rapport à la contrainte C passée en paramètre. La fonction renvoie vrai lorsqu'au moins une révision est effective, sinon faux.

L'algorithme 2 n'est pas optimal. Etant donné une variable X et une contrainte binaire C dont la portée implique les variables X et Y, pour chaque valeur de X, on parcourt le domaine de

Algorithme 3 : *revise*-2001/3.1

Entrées : C : Contrainte, X : Variable
Sorties : Booléen

```
1  Supprime=faux ;
2  Soit {X, Y} = scp(C) ;
3  pour chaque a ∈ dom(X) faire
4  |  b ← Last((X, a), Y) ;
5  |  si b ∉ dom(Y) alors
6  |  |  b ← succ(b, dom(Y)) ;
7  |  |  tant que (b ≠ NIL) et ({(X, a), (Y, b)} ∉ rel(C)) faire b ← succ(b, dom(Y)) ;
8  |  |  si b ≠ NIL alors
9  |  |  |  Last((X, a), Y) ← b ;
10 |  |  sinon
11 |  |  |  dom(X)=dom(X)\{a} ;
12 |  |  |  Supprime=vrai ;
13 |  |  fin
14 |  fin
15 fin
16 retourner Supprime ;
```

Y à la recherche d'un support compatible avec la relation associée à la contrainte C. Ce parcours peut être effectué plusieurs fois pour identifier la même valeur.

L'algorithme 3 est optimal. Il nécessite une structure de données supplémentaire, mais évite les recherches de supports inutiles. On mémorise dans la structure $Last((X, a), Y)$ le dernier support b (trouvé) de la variable Y compatible avec une valeur a de X donnée. Lorsqu'on cherchera ultérieurement un support compatible pour la valeur a de X dans le domaine de Y, il suffira de vérifier si le support enregistré au préalable dans la structure $Last$ est toujours valide, i.e si la valeur b appartient toujours au domaine de Y. Si tel est le cas, aucune recherche de support n'est nécessaire, sinon on recherche à partir de b une valeur compatible et on met à jour la structure $Last$.

Récapitulatif des algorithmes Les algorithmes à grain fin ont généralement une complexité temporelle optimale, cependant leur mise en oeuvre est souvent complexe. Le tableau 1.1 résume la complexité spatiale et temporelle de quelques algorithmes établissant la consistance d'arc.

Les deux algorithmes les plus utilisés sont les algorithmes $AC3$ (à cause de la simplicité d'implémentation) et $AC2001$ (pour l'optimalité de l'algorithme). Plus récemment, l'algorithme $AC3^{rm}$ proposé dans [Lecoutre et Hemery, 2007] a relancé l'intérêt pour $AC3$ en exploitant le concept de *support résiduel* afin d'éviter des recherches de supports inutiles. Un support résiduel (ou plus simplement résidu) est un support qui a été trouvé et enregistré au cours d'un appel précédent à la procédure qui détermine si une valeur est supportée par une contrainte (ligne 4 de l'algorithme 2). L'algorithme $AC3$ peut être raffiné en testant la validité du résidu avant de chercher un nouveau support pour une valeur.

Cet algorithme est optimal pour les contraintes de dureté faible ou élevée (i.e. pour les contraintes acceptant beaucoup ou très peu de n-uplets valides) ce qui le rend compétitif face à $AC2001$. De plus, le maintien d'$AC3^{rm}$ à chaque étape d'un algorithme de recherche (voir section 1.2.2) est très souvent plus efficace que le maintien d'$AC2001$ et évite également les cas

pathologiques (connus). Le tableau 1.2 présente les résultats obtenus, en terme de nombre de tests de consistance (ccks) et de temps cpu (cpu), par différents algorithmes de filtrage établissant la consistance d'arc sur les instances du problème *Domino* introduit par [Zhang et Yap, 2001]. Chaque instance *domino-n-d* est caractérisée par un couple (n, d) où n représente le nombre de variables dont les domaines sont $\{1 \dots d\}$. Ce problème comporte $n-1$ contraintes d'égalité (du type $X_i = X_{i+1}$, $\forall i \in [1, n-1]$) et une contrainte de déclenchement $(X_1 = X_n + 1 \wedge X_1 < d) \vee (X_1 = X_n \wedge X_1 = d)$.

La généralisation aux contraintes n-aires

L'arc-consistance s'étend très facilement aux contraintes non binaires. Dans ce cas on l'appelle l'*hyperarc-consistance* ou consistance d'arc généralisée (notée GAC pour Generalized Arc Consistency).

Définition 13 (contrainte hyperarc-consistante)**.** *Soit $P = (\mathscr{X}, \mathscr{C})$ un CN. Une contrainte $C \in \mathscr{C}$ est hyperarc-consistante si et seulement si $\forall X \in scp(C)$ on a : $\forall a \in dom(X), \exists t \in rel(C)$ tel que $t[X] = a \wedge (\forall Y \neq X \in scp(C), t[Y] \in dom^P(Y))$.*

Définition 14 (réseau hyperarc-consistant)**.** *Un réseau de contraintes $P = (\mathscr{X}, \mathscr{C})$ est hypearc-consistant si toutes les contraintes de $\mathscr{C}$ sont hyperarc-consistantes.*

Les algorithmes établissant l'arc-consistance se généralisent également de la même manière en utilisant ce nouveau concept d'hyperarc-consistance. On parle alors des algorithmes $GAC3$, $GAC2001 \dots$

Survol non exhaustif d'autres formes de consistance

La consistance de nœuds aussi appelée 1-consistance, garantit que toutes les contraintes unaires d'une variable donnée sont satisfaites par toutes les valeurs du domaine de cette variable. Autrement dit, toute affectation vide peut se prolonger sur chacune des variables en une affectation consistante de cette variable, c'est-à-dire qu'il existe dans chaque domaine une valeur qui respecte l'éventuelle contrainte unaire pesant sur cette variable, ou encore qu'aucun domaine n'est vide après élimination des valeurs ne respectant pas l'éventuelle contrainte unaire. Il s'agit de la forme la plus simple de consistance. Le nœud représentant une variable X dans le graphe de contraintes est nœud-consistant si chaque valeur a du domaine de X satisfait l'ensemble des contraintes unaires associées à X. Cette condition peut être établie en réduisant le domaine de

Algorithme	Temps	Espace	Grain	Auteur
$AC3$	$O(ed^3)$	–	gros	[Mackworth, 1977]
$AC4$	$O(ed^2)$	$O(ed^2)$	fin	[Mohr et Henderson, 1986]
$AC6$	$O(ed^2)$	$O(ed)$	fin	[Bessiere, 1994]
$AC7$	$O(ed^2)$	$O(ed)$	fin	[Bessiere *et al.*, 1999]
$AC3_d$	$O(ed^3)$	$O(e+nd)$	gros	[van Dongen, 2002]
$AC2001/3.1$	$O(ed^2)$	$O(ed)$	gros	[Bessiere *et al.*, 2005]
$AC3.2/3.3$	$O(ed^2)$	$O(ed)$	gros	[Lecoutre *et al.*, 2003]
$AC3^{rm}$	$O(ed^2/ed^3)$	$O(ed)$	gros	[Lecoutre et Hemery, 2007]

TAB. 1.1 – Algorithmes établissant l'arc-consistance et complexité

Instances		$AC3$	$AC3^{rm}$	$AC2001$
Domino-100-100	*cpu*	$1,81$	$0,16$	$0,23$
	ccks	$18M$	$990K$	$1\,485K$
Domino-300-300	*cpu*	134	$3,40$	$6,01$
	ccks	$1\,377M$	$27M$	$40M$
Domino-500-500	*cpu*	951	$15,0$	$21,4$
	ccks	$10\,542M$	$125M$	$187M$
Domino-800-800	*cpu*	$6\,144$	60	87
	ccks	$67\,778M$	$511M$	$767M$

TAB. 1.2 – Efficacité des algorithmes établissant l'arc-consistance sur les problèmes *Domino*

chaque variable aux valeurs qui satisfont les contraintes unaires de cette variable. Par conséquent, les contraintes unaires peuvent être supprimées grâce à la réduction des domaines des variables.

Par exemple, considérons une variable X telle que $dom(X) = \{1, 2, 3, 4\}$ et une contrainte unaire $C_1 = X \leq 3$. La consistance de nœud restreindra le domaine de X aux valeurs $\{1, 2, 3\}$ et C_1 pourra être ensuite ignorée.

La singleton consistance d'arc [Debruyne et Bessiere, 1997b] (SAC pour Singleton Arc Consistency) consiste en quelque sorte à réaliser un parcours en largeur d'abord jusqu'à une profondeur égale à un. C'est une forme de consistance plus forte que la consistance d'arc, c'est-à-dire que la singleton consistance d'arc permet d'identifier plus de valeurs inconsistantes que la consistance d'arc. Un réseau de contraintes est singleton arc-consistant si et seulement si après avoir réduit le domaine d'une variable à un singleton (c'est-à-dire effectué une assignation) et établi la consistance d'arc, le réseau de contraintes n'est pas détecté inconsistant. On parle alors de test singleton. Cette propriété doit être vérifiée quelque soit la variable du réseau de contraintes et la valeur assignée à cette variable. En d'autres termes, SAC garantit qu'appliquer la consistance d'arc après avoir effectué n'importe quelle assignation de variable ne provoque pas un échec. De nombreux algorithmes ont été proposés pour établir la singleton arc-consistance, e.g. $SAC1$ [Debruyne et Bessiere, 1997b], $SAC2$ [Bartak et Erben, 2004], *SAC-Opt* [Bessiere et Debruyne, 2004], *SAC-SDS* [Bessiere et Debruyne, 2005] et *SAC*-3 [Lecoutre et Cardon, 2005].

La consistance de chemin [Mackworth, 1977] (PC pour Path Consistency) permet de filtrer plus que la consistance d'arc, en considérant non plus les relations entre deux variables (arc-consistance) mais entre trois variables. Il s'agit d'une consistance qui implique de modifier les relations associées aux contraintes et même éventuellement la structure du graphe de contraintes associé au réseau. Un réseau de contraintes est 3-consistant (ou chemin-consistant) si toute affectation de deux variables du problème peut être étendue à l'affectation d'une troisième variable. Un réseau de contraintes est fortement chemin-consistant (également appelé strong PC) s'il garantit à la fois la consistance d'arc et la consistance de chemin. Même si la consistance de chemin forte filtre plus que la consistance d'arc (au sens du nombre de n-uplets sur l'ensemble des contraintes d'un réseau donné), il peut être coûteux de maintenir ce type de consistance au cours de la recherche. De nombreux algorithmes ont été proposés pour établir la consistance de chemin, e.g. $PC3$ [Mohr et Henderson, 1986], $PC4$ [Han et Lee, 1988], $PC8$ [Chmeiss et Jégou, 1998], $PC2001$ [Bessiere *et al.*, 2005] et sDC_2 [Lecoutre *et al.*, 2007a].

La consistance de chemin restreinte [Berlandier, 1995] (RPC pour Restricted Path Consistency) ne s'applique qu'aux réseaux de contraintes binaires. C'est une combinaison des avantages de la consistance d'arc et de la consistance de chemin. RPC augmente la capacité de filtrage de l'algorithme de consistance d'arc en garantissant la chemin consistance des couples $((X_i, v_i), (X_j, v_j))$ qui sont tels que (X_j, v_j) est l'unique support de v_i dans $dom(X_i)$ pour la contrainte impliquant X_i et X_j. La consistance de chemin restreinte est plus forte que la consistance d'arc puisqu'elle supprime au moins le même nombre de valeurs inconsistantes qu'AC. Cependant, elle est plus faible que la consistance de chemin puisqu'elle n'établit PC que pour certains couples (variable,valeur).

Max-RPC est une extension de RPC proposée dans [Debruyne et Bessiere, 1997a]. Le nombre de tests de consistance effectués sur des triplets de variables est plus important. Un réseau de contraintes est dit max-RPC consistant si toutes les valeurs ont au moins un support chemin consistant sur chaque contrainte.

1.2.2 Algorithmes de recherche

Plusieurs algorithmes ont été proposés pour rechercher (de manière plus ou moins efficace) une (ou l'ensemble des) solution(s) d'un réseau de contraintes donné.

Algorithme *générer et tester*

La première approche (assez naïve) que l'on peut envisager pour résoudre une instance CSP est appelée *générer et tester*. Cet algorithme construit une à une toutes les instanciations de l'ensemble des variables d'un réseau de contraintes P donné et vérifie si celles-ci sont des solutions ou non. Toutes les instanciations possibles de toutes les variables de $vars(P)$ sont donc énumérées et testées successivement. Toute instanciation qui viole au moins une des contraintes de P est éliminée de l'ensemble des solutions. La complexité temporelle dans le pire des cas d'un tel algorithme est très élevée. En effet, il peut être nécessaire de construire d^n instanciations différentes (pour trouver toutes les solutions ou alors pour prouver l'insatisfaisabilité) et pour chaque instanciation il faut effectuer er tests pour vérifier si aucune contrainte n'est violée (en admettant qu'un test s'effectue en temps linéaire par rapport à l'arité des contraintes). On obtient alors une complexité temporelle dans le pire des cas de $O(d^n er)$ avec d la taille du plus grand domaine, n le nombre de variables, e le nombre de contraintes, et r l'arité maximale des contraintes. Pour rechercher l'ensemble des solutions du (simple) exemple des 4-reines, cet algorithme génère et vérifie la validité de $4^4 = 256$ instanciations différentes. Le problème des 8-reines admet $8^8 = 16\,777\,216$ instanciations différentes. Il paraît alors évident qu'évaluer successivement toutes les instanciations possibles n'est vraiment pas une méthode efficace.

La figure 1.7 illustre l'énumération des différentes instanciations effectuées par l'algorithme *générer et tester*. Celui-ci va générer et vérifier la validité de l'instanciation $t_1 = \{(X_0, 1), (X_1, 1), (X_2, 1), (X_3, 1)\}$, puis $t_2 \ldots$ jusqu'à t_k.

Algorithme de recherche avec retours-arrière

L'algorithme de recherche avec retours-arrière ((S)BT pour (Standard) BackTracking) est l'algorithme principalement utilisé pour la résolution d'instances CSP. Il consiste à étendre incrémentalement une instanciation partielle en une solution. Pour cela il effectue une recherche en profondeur d'abord dans le but de trouver une solution en assignant une à une les variables d'un réseau de contraintes donné et utilise un mécanisme de retours-arrière lorsqu'un conflit est détecté.

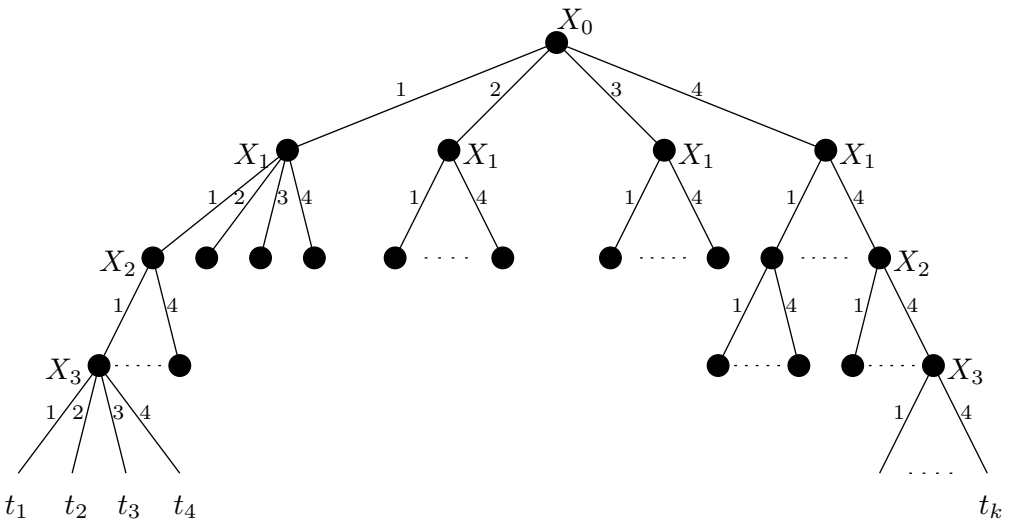


FIG. 1.7 – Enumération des instanciations par l'algorithme "générer et tester" sur le problème des 4-reines

Le réseau de contraintes initial est modifié en réduisant successivement le domaine de chaque variable, tout en vérifiant à chaque étape de la recherche la satisfaisabilité de toutes les contraintes du réseau. Lorsqu'une contrainte est violée, la dernière modification effectuée sur le réseau (comme par exemple l'assignation d'une valeur à une variable ou alors sa réfutation) est alors remise en cause, ce qui se traduit par un retour-arrière sur la variable sélectionnée précédemment.

Cet algorithme est plus efficace que l'algorithme "générer et tester" puisqu'il permet de couper certaines branches ne menant à aucune solution et ainsi éviter leur exploration.

L'algorithme construit un arbre de recherche où chaque nœud de l'arbre peut être associé à un sous-réseau de contraintes (du réseau initial) qu'il faut résoudre. La racine de cet arbre correspond au réseau de contraintes initial. Dans la littérature, la construction d'un tel arbre se fait généralement en considérant deux schémas de branchement différents : le schéma de branchement binaire et le schéma de branchement non-binaire. Chaque branche de l'arbre de recherche correspond à une séquence de décisions (c.f. Définition 15), i.e. des assignations dans le cas d'un schéma de branchement non-binaire, et des assignations et/ou des réfutations dans le cas d'un schéma de branchement binaire. Après avoir réduit le domaine de toutes les variables à des singletons, si toutes les contraintes sont satisfaites alors le sous-réseau obtenu est trivial et une solution a été trouvée. Cette solution correspond à l'instanciation courante, c'est-à-dire aux assignations effectuées le long de la branche menant de la racine à une feuille de l'arbre de recherche. Lorsque l'arbre de recherche a été complètement exploré et qu'aucune solution n'a pu être identifiée alors le réseau de contraintes a été démontré insatisfaisable.

Schéma de branchement non-binaire En considérant un schéma de branchement non-binaire (c.f algorithme 4), on vérifie d'abord la consistance du réseau initial grâce à la fonction $isConsistent(P)$[2]. Celle-ci vérifie pour chaque contrainte C de P impliquant uniquement des variables dont le domaine est réduit à un singleton que C n'est pas violée. Cette fonction retourne la première contrainte violée rencontrée ou alors NO dans le cas ou aucune contrainte n'est violée. Remarquons que si toutes les variables d'un réseau P sont singletons et que cette fonction retourne

[2]Cette fonction sera réutilisée ultérieurement dans les algorithmes.

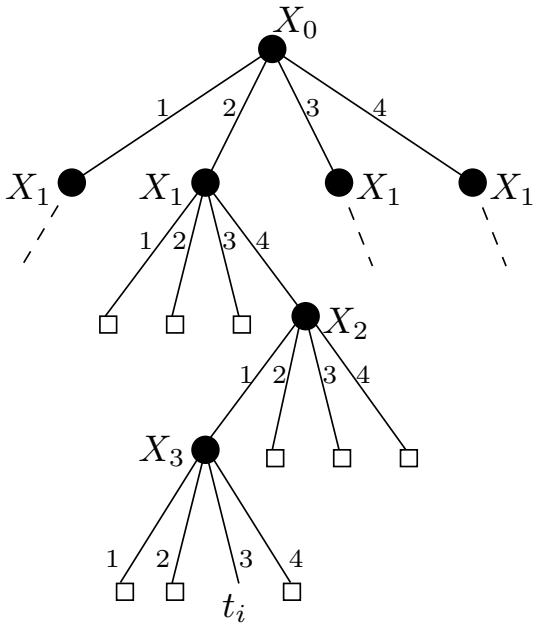


FIG. 1.8 – Arbre de recherche partiel (correspondant au problème des 4-reines) construit par un algorithme de recherche avec retours-arrière utilisant un schéma de branchement non-binaire

NO, alors P est satisfaisable et l'instanciation totale $\{(X, v)\} \mid X \in vars(P) \wedge v \in dom(X)$ est une solution de P.

A chaque étape de la recherche on sélectionne une variable X qui n'est pas encore assignée. On essaye d'assigner successivement toutes les valeurs du domaine de cette variable, ce qui correspond à un nœud de l'arbre de recherche. Si l'assignation d'une valeur mène à un échec, elle est remise en cause et on assigne alors une autre valeur appartenant au domaine de X. Lorsque toutes les valeurs de la variable X mènent à un échec, l'algorithme effectue un retour-arrière et remet en cause le dernier choix de valeur pour la variable précédemment sélectionnée.

La figure 1.8 illustre un arbre de recherche partiel correspondant au problème des 4-reines (décrit en section 1.1.2) construit par un algorithme de recherche utilisant un schéma de branchement non-binaire. Les variables et les valeurs sont ordonnées selon l'ordre lexicographique. Dès qu'une contrainte n'est plus satisfaite, la branche correspondante de l'arbre de recherche est aussitôt abandonnée et le sous-arbre n'est pas exploré. Toutes les instanciations de l'ensemble des variables issues de ce sous-arbre ne sont donc pas évaluées. Sur la figure 1.8, l'instanciation $t_i = \{(X_0, 2)(X_1, 4)(X_2, 1)(X_3, 3)\}$ est une solution découverte par l'algorithme après avoir (entre autre) coupé la branche correspondant à l'instanciation $\{(X_0, 2)(X_1, 3)\}$ violant la contrainte C_0.

Schéma de branchement binaire A chaque étape de la recherche, une paire (X, v) est sélectionnée où X est une variable qui n'est pas encore assignée et v une valeur appartenant au domaine de X. Deux cas sont alors considérés : l'assignation $X = v$ et la réfutation $X \neq v$.

Définition 15 (décisions positives et négatives). *Soit $P = (\mathscr{X}, \mathscr{C})$ un CN. Une décision δ construite à partir de P correspond soit à une assignation $X = v$ (appelée décision positive) soit à une réfutation $X \neq v$ (appelée décision négative) où $X \in \mathscr{X}$ et $v \in dom(X)$. $\neg(X = v)$ représente $X \neq v$ et $\neg(X \neq v)$ représente $X = v$.*

Algorithme 4 : *solveNonBinary*

Entrées : $P = (\mathscr{X}, \mathscr{C})$: CN
Sorties : Booléen

```
1 si isConsistent(P) ≠ NO alors retourner faux ;
2 si ∀X ∈ 𝒳, |dom(X)| = 1 alors retourner vrai ;
3 choisir une variable X avec |dom(X)| > 1 ;
4 pour chaque a ∈ dom(X) faire
5 |   si (solveNonBinary(P|X=a)) alors retourner vrai ;
6 fin
7 retourner faux ;
```

On considérera qu'aucun ensemble de décision Δ ne peut contenir deux décisions positives impliquant la même variable.

Un réseau de contraintes initial P est modifié à chaque étape de la recherche par les assignations et/ou les réfutations de valeurs effectuées. Étant donné un ensemble de décisions Δ (correspondant à l'ensemble des assignations et/ou réfutations effectuées sur une branche d'un arbre de recherche), $P' = P|_\Delta$ est le réseau de contraintes dérivé de P tel que, pour chaque décision positive $(X = v) \in \Delta$, $dom(X)$ est restreint à $\{v\}$, et pour chaque décision négative $(X \neq v) \in \Delta$, v est supprimé de $dom(X)$. A chaque nœud de l'arbre de recherche, on peut extraire un sous-réseau de contraintes P' tel que $P' \preceq P$. Par commodité, $P|_{\{X=v\}}$ et $P|_{\{X\neq v\}}$ seront simplement notés $P|_{X=v}$ et $P|_{X\neq v}$.

L'algorithme 5 construit un arbre binaire de recherche. Généralement on commence par assigner une valeur à une variable avant de réfuter la valeur précédemment assignée, i.e. les décisions positives sont prises en premier. La figure 1.9 illustre un arbre de recherche partiel correspondant au problème des 4-reines (décrit en section 1.1.2) construit par un algorithme de recherche utilisant un schéma de branchement binaire. Les branches de gauche de l'arbre de recherche correspondent aux décisions positives, i.e aux assignations, et les branches de droite aux décisions négatives, i.e aux réfutations de valeurs précédemment assignées. Après avoir assigné la valeur 1 à la variable X_0 et exploré le sous arbre correspondant, l'algorithme réfute cette valeur $(X_0 \neq 1)$ et choisit un nouveau couple (X, v) à assigner.

Pendant la recherche on peut distinguer les *nœuds ouverts* pour lesquels un seul cas a été considéré et les *nœuds fermés* pour lesquels les deux cas ont été considérés.

Algorithme 5 : *solveBinary*

Entrées : $P = (\mathscr{X}, \mathscr{C})$: CN
Sorties : Booléen

```
1 si isConsistent(P) ≠ NO alors retourner faux ;
2 si ∀X ∈ 𝒳, |dom(X)| = 1 alors retourner vrai ;
3 choisir un couple (X, a) avec |dom(X)| > 1 ∧ a ∈ dom(X) ;
4 retourner (solveBinary(P|X=a) ou solveBinary(P|X≠a)) ;
```

Ces deux schémas de branchement ne sont pas équivalents puisqu'il a été montré dans [Hwang et Mitchell, 2005], que le schéma binaire était plus puissant (notamment pour réfuter des instances insatisfaisables) que le schéma non-binaire.

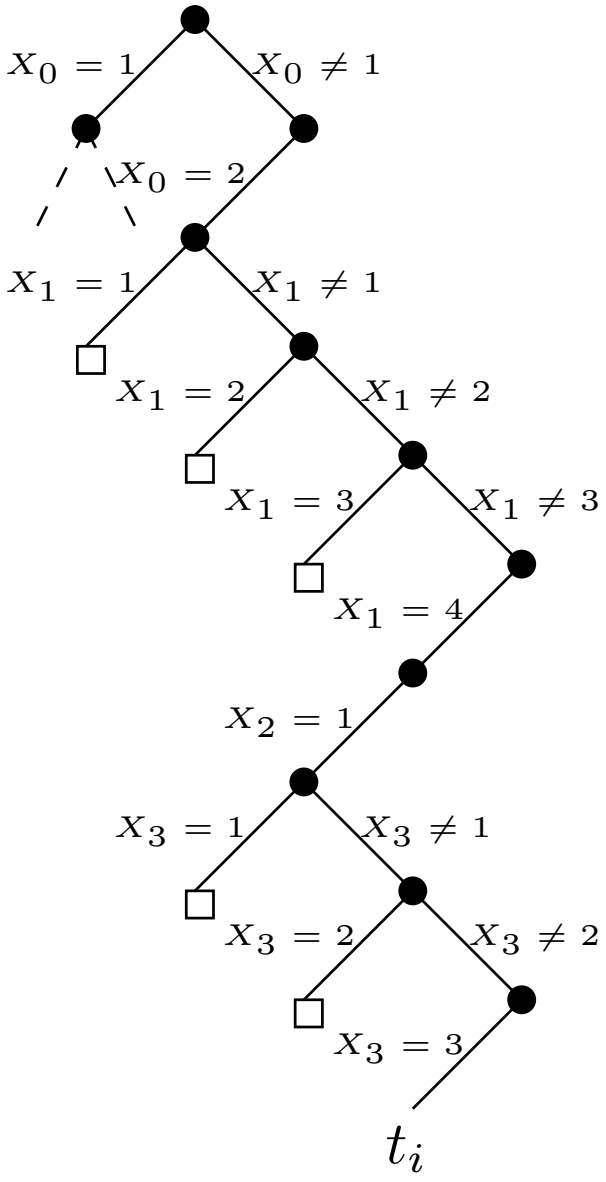


FIG. 1.9 – Arbre de recherche partiel (correspondant au problème des 4-reines) construit par un algorithme de recherche avec retours-arrière utilisant un schéma de branchement binaire

Le maintien de l'arc-consistance pendant la recherche Afin d'améliorer l'efficacité de la recherche, le réseau de contraintes associé à chaque nœud de l'arbre de recherche peut être maintenu arc-consistant en appliquant l'un des algorithmes de filtrage présentés en section 1.2.1. GAC est maintenu pendant la recherche par l'algorithme MGAC (pour Maintaining Generalized Arc-Consistency) ou MAC [Sabin et Freuder, 1994] dans le cas d'un réseau de contraintes binaires. Celui-ci intègre l'un des algorithmes présentés en section 1.2.1.

Algorithme 6 : *ϕ-solve*

Entrées : $P_{init} = (\mathscr{X}, \mathscr{C})$: CN
Sorties : Booléen

1 $P = \phi(P_{init})$;
2 **si** $P = \bot$ **alors retourner** $faux$;
3 **si** $\forall X \in \mathscr{X}, |dom(X)| = 1$ **alors retourner** $vrai$;
4 choisir un couple (X, a) avec $|dom(X)| > 1 \wedge a \in dom(X)$;
5 **retourner** (ϕ-$solve(P|_{X=a})$ ou ϕ-$solve(P|_{X \neq a})$) ;

L'algorithme 6 présente l'intégration d'un opérateur établissant une consistance ϕ au sein d'un algorithme de recherche utilisant un schéma de branchement binaire. Le réseau de contraintes passé en paramètre est filtré grâce à l'opérateur d'inférence. Si celui-ci n'est pas inconsistant, la recherche se poursuit en considérant l'assignation puis la réfutation d'une valeur pour une variable sélectionnée. Ainsi si $\phi = AC$, la consistance d'arc est établie à chaque nœud de la recherche, et on obtient MAC (Maintaining Arc-Consistency).

Le forward checking [Haralick et Elliott, 1980] (FC) est une forme affaiblie de MAC. Cette technique est intégrée au sein d'un algorithme de recherche avec retours-arrière (c.f. section 1.2.2) et établit la consistance d'arc au voisinage des variables qui sont successivement assignées par l'algorithme de recherche (et non pas sur toutes les contraintes du réseau). Cette technique s'applique aux réseaux de contraintes binaires et a été étendue aux réseaux non-binaires dans [Bessiere *et al.*, 2002]. Plus précisément, la propriété de consistance d'arc est maintenue entre la variable assignée et les variables connectées à celle-ci par une contrainte.

1.2.3 Guider la recherche

On peut rendre la résolution d'une instance CSP plus efficace en effectuant les "bons" choix, c'est-à-dire en sélectionnant les "bons" couples $(variable, valeur)$ à assigner à chaque étape de la recherche. Jusqu'à présent, les algorithmes de recherche présentés en section 1.2.2 ne fournissent aucun renseignement sur l'ordre dans lequel on doit instancier les variables, ni sur l'ordre dans lequel on doit affecter les valeurs aux variables. A une étape donnée de la recherche, le choix de ce couple est déterminé par une heuristique qui évalue la (potentielle) meilleure variable et la (potentielle) meilleure valeur à assigner à cette variable parmi tous les candidats possibles. Une heuristique est une règle approximative (dans le sens où elle n'est pas fiable à 100%) qui nous donne des indications sur la direction à prendre dans l'arbre de recherche.

Classiquement on distingue les heuristiques de choix de variable des heuristiques de choix de valeur. L'efficacité des heuristiques de choix de valeur est très dépendante de l'application à résoudre et il est difficile de les généraliser. On peut par exemple utiliser l'heuristique *lexico* ou alors l'heuristique *min-Conflict* [Frost et Dechter, 1995]. La première sélectionne une valeur suivant l'ordre lexicographique dans lequel elle apparaît dans l'énoncé du problème initial tandis

que la seconde sélectionne la valeur minimisant le nombre de conflits avec les valeurs des domaines adjacents (i.e. avec les valeurs des domaines des variables connectées à la variable précédemment sélectionnée par l'heuristique de choix de variable).

De nombreux travaux ont été effectués sur les heuristiques de choix de variable. Celles-ci sont classées en trois catégories : statiques, dynamiques et adaptatives. Les heuristiques statiques (*SVOs* pour *Static Variable Ordering*) conservent le même ordre de priorité tout au long de la recherche. Cet ordre est établi au départ à partir de la structure du problème. Les heuristiques dynamiques (*DVOs* pour *Dynamic Variable Ordering*) exploitent différentes informations sur l'état courant du problème afin de déterminer la variable la plus prometteuse à un instant donné de la recherche. Les heuristiques adaptatives combinent élégamment les informations dynamiques et statiques du problème. Elles exploitent notamment des informations sur les états passés de la recherche, qui se situent non seulement sur la branche courante mais aussi sur les branches déjà explorées de la recherche.

Classiquement les heuristiques suivent l'un des deux principes suivants [Beck *et al.*, 2004, Wallace, 2005] : "fail first" et "promise". Ces deux politiques orientent la recherche de façon complètement opposée. Le principe "promise" tente de minimiser les échecs alors que le principe "fail first" guide la recherche plutôt vers les conflits : "Pour réussir, essayons d'abord là où l'on est susceptible d'échouer". Traditionnellement, l'heuristique de choix de variable et l'heuristique de choix de valeur ne suivent pas la même politique. On préfère sélectionner la variable la plus contrainte (c'est-à-dire suivre le principe du "fail first") tandis qu'on sélectionne la valeur la moins contrainte (c'est-à-dire suivre le principe du "promise").

Les heuristiques dynamiques, basées sur le domaine et le degré

La taille du domaine courant d'une variable X ($|dom^P(X)|$) correspond au nombre de valeurs qui peuvent être affectées à X. En d'autres termes, la taille du domaine courant d'une variable X correspond au nombre de valeurs toujours présentes à un instant donné dans le domaine de X. L'heuristique *dom* [Haralick et Elliott, 1980] ordonne de façon croissante les variables suivant la taille courante de leur domaine.

Le degré d'une variable X correspond au nombre de contraintes liant celle-ci aux autres variables du problème à un instant donné de la recherche. L'heuristique *ddeg* ordonne les variables de manière décroissante en fonction de leur degré courant (i.e. dynamique). En d'autres termes, l'heuristique *ddeg* sélectionne en priorité la variable connectée avec le plus grand nombre de variables non encore assignées.

L'heuristique dom/ddeg [Bessiere et Régin, 1996] Cette heuristique est une heuristique dynamique qui sélectionne en priorité la variable ayant le plus petit ratio taille du domaine courant sur degré dynamique courant. Cette heuristique améliore nettement les performances de la recherche sur certaines séries d'instances.

L'heuristique brelaz [Brelaz, 1979] Cette heuristique a été initialement développé spécifiquement pour résoudre les problèmes de coloriage de graphes. Elle choisit en priorité la variable ayant le plus petit domaine mais en cas d'égalité, c'est-à-dire lorsqu'un ensemble de variables ont un domaine de même taille, elle choisit dans cet ensemble la variable ayant le plus grand degré dynamique. Très souvent l'heuristique *brelaz* est notée $dom + ddeg$. Des variantes ($bz2$, $bz3$) existent [Smith, 1999] même si les améliorations qu'elles apportent semblent d'intérêt limité.

Les heuristiques adaptatives

L'heuristique wdeg basée sur le degré des contraintes [Boussemart *et al.*, 2004a] permet de diriger la recherche vers les parties difficiles ou inconsistantes d'un problème, suivant donc le principe "fail first" décrit précédemment. Tout au long du processus de recherche, cette heuristique exploite l'expérience des états précédents. Elle réduit le phénomène de thrashing (c.f. section 1.3) en choisissant en priorité les variables liées aux contraintes qui conduisent le plus souvent à des situations de conflit. Cette information est enregistrée en associant un compteur à chaque contrainte du problème. Ces compteurs sont utilisés pour pondérer les contraintes. Le poids d'une contrainte est incrémenté lorsque celle-ci est violée au cours de la recherche. Les contraintes les plus violées sont donc celles qui possèdent les poids les plus élevés.

En utilisant ces compteurs, on définit l'heuristique de choix de variable *wdeg*, qui donne une évaluation appelée *degré pondéré* de chaque variable. Le degré pondéré d'une variable X correspond à la somme des poids des contraintes impliquant X et au moins une autre variable non-instantiée. Pour bénéficier au début de la recherche d'informations pertinentes sur les degrés pondérés courants des variables, tous les compteurs sont initialisés à 1.

En combinant les degrés pondérés avec la taille des domaines des variables, on obtient une heuristique *dom/wdeg* qui sélectionne en priorité la variable avec le plus petit ratio taille du domaine courant sur degré pondéré courant.

Grâce à la pondération, l'heuristique peut éventuellement détecter le noyau difficile d'une instance CSP donnée et diriger la recherche sur cette partie. Cette approche permet de traiter plus efficacement les problèmes structurés, où certaines contraintes apparaissent comme plus importantes que d'autres et où par conséquence certaines parties du problème sont soit inconsistantes, soit difficiles à résoudre. De plus cette heuristique est une alternative efficace aux techniques de retours-arrière intelligentes (c.f. section 2.3). De nombreux travaux ont démontré l'efficacité de l'heuristique *dom/wdeg* par rapport aux autres précédemment mentionnées [Boussemart *et al.*, 2004a, Lecoutre *et al.*, 2004, Hulubei et O'Sullivan, 2005, van Dongen, 2005, Grimes et Wallace, 2007].

L'heuristique impact [Refalo, 2004] L'impact d'une décision est quantifié par la réduction moyenne de l'espace de recherche engendrée par cette décision. Au cours de la recherche, à chaque assignation $X = a$, l'impact de cette décision est mis à jour dynamiquement à partir de la réduction des domaines impliquée par cette assignation. Cette heuristique suit le principe "fail first" discuté précédemment : on sélectionne la décision ayant l'impact maximal sur l'espace de recherche. La taille de l'arbre de recherche (à une étape donnée de la recherche) est estimée en considérant toutes les combinaisons de valeurs possibles pour les variables (c'est-à-dire le produit cartésien). L'impact d'une décision $\delta : (X = a)$ correspond alors au rapport entre $1 - \frac{P_{after}}{P_{before}}$ où P_{before} représente la taille de l'arbre de recherche avant l'assignation et P_{after} après l'assignation.

Même si l'heuristique *impact* semble à la fois un peu plus complexe à mettre en place et plus coûteuse à utiliser (sauf si l'heuristique est statique et que le calcul des impacts n'est effectué qu'initialement) que *wdeg*, il serait intéressant de comparer le comportement de ces deux heuristiques. Les premiers résultats issus de cette comparaison [Cambazard et Jussien, 2006] semblent confirmer l'hypothèse selon laquelle *wdeg* est au moins aussi efficace qu'*impact*. Les auteurs ont notamment étudié différentes stratégies d'impact [Cambazard et Jussien, 2005] basées sur des explications comme par exemple le nombre d'occurrences d'une décision dans les raisons de la suppression d'une valeur.

1.3 Thrashing

Le thrashing est le fait d'explorer de façon répétitive les mêmes sous-arbres. Autrement dit, les mêmes impasses (ou succès partiels) sont redécouvertes plusieurs fois au cours de la recherche. Ce phénomène doit être étudié avec soin, car un algorithme de recherche sujet au thrashing peut vraiment être très inefficace. Parfois le thrashing peut être expliqué par les mauvais choix effectués au préalable par l'heuristique de choix de variable.

Prenons par exemple le problème académique des *reines et cavaliers*. Ce problème insatisfaisable fusionne deux sous-problèmes, le problème des reines et celui des cavaliers. Le problème des reines est un problème satisfaisable. Il consiste à disposer n reines sur un échiquier de taille nxn de manière à ce qu'aucune d'entre elles ne soit en prise avec les autres. Une modélisation de ce problème est décrite en section 1.1.2. Le problème des cavaliers est un problème insatisfaisable lorsque le nombre de cavaliers est impair (ce qui rend le problème des *reines et cavaliers* lui même insatisfaisable). Il consiste à disposer k cavaliers sur un échiquier (de taille nxn) de manière à ce que les cavaliers forment une chaîne, c'est-à-dire que l'on puisse sauter d'un cavalier à un autre suivant la règle de prise des cavaliers du jeu d'échecs. Ces deux problèmes peuvent être fusionnés de deux manières. Si on autorise une reine et un cavalier à se trouver sur une même case, les deux sous-problèmes sont alors fusionnés sans interaction l'un avec l'autre. Ce problème sera noté *queensKnights-add*. Si on interdit qu'une reine et un cavalier se trouvent sur une même case, on ajoute alors des contraintes entres les deux sous-problèmes. Celui-ci sera alors noté *queensKnights-mul*.

Comme le problème des reines possède de nombreuses solutions (le nombre de solutions augmente d'ailleurs avec la taille de l'échiquier) et que le problème des cavaliers est insatisfaisable un phénomène de thrashing peut apparaître. Une approche possible pour résoudre ce problème consiste à résoudre d'abord le sous-problème des reines puis celui des cavaliers. Pour chaque solution du problème des reines, on est donc amené à résoudre le sous-problème des cavaliers pour prouver l'insatisfaisabilité globale du problème. L'insatisfaisabilité du sous-problème des cavaliers va donc être démontrée plusieurs fois au cours de la recherche, ce qui entraîne l'apparition d'un phénomène de thrashing. L'instance des 8-reines comportant 92 solutions, si l'on fait l'hypothèse que les reines sont sélectionnées prioritairement par rapport aux cavaliers (ce qui est le cas avec une heuristique de choix de variable comme *dom* par exemple), il faudra démontrer au minimum 92 fois l'insatisfaisabilité du problème des cavaliers pour prouver l'insatisfaisabilité globale du problème des *QueensKnights-add*-8. De plus, le nombre de solutions croît exponentiellement avec le nombre de reines. Donc plus le nombre de reines augmente et plus il est difficile pour les algorithmes sujets au trashing de prouver l'insatisfaisabilité du problème. Ainsi l'instance des 12-reines comporte 14 200 solutions et l'instance des 16-reines 14 772 512 solutions.

Pour éviter le thrashing (et par conséquent accélérer la recherche), une meilleure stratégie aurait consisté à résoudre d'abord le sous-problème des cavaliers puis celui des reines. Le problème des cavaliers étant lui-même inconsistant, le problème global des reines et cavaliers est donc également globalement inconsistant.

L'heuristique *wdeg* décrite précédemment (c.f. section 1.2.3) permet de lutter (en partie) contre les effets du thrashing. Sur l'exemple des reines et cavaliers, les contraintes les plus violées sont celles qui correspondent au sous-problème insatisfaisable des cavaliers (puisque le sous-problème des reines admet de nombreuses solutions). Même si initialement les poids sont initialisés à la même valeur (et par conséquent on ne peut faire aucune supposition sur le choix de la variable à assigner), très rapidement l'heuristique *wdeg* va sélectionner en priorité les variables impliquées dans les contraintes les plus violées, c'est-à-dire les variables des cavaliers, qui seront donc par la suite assignées en priorité. L'insatisfaisabilité du sous-problème des cavaliers

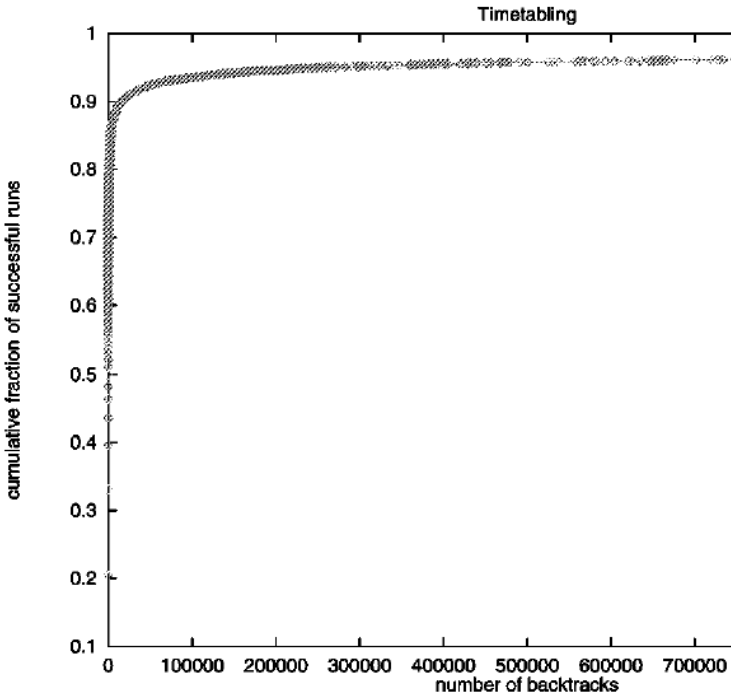


FIG. 1.10 – Mise en évidence du phénomène heavy-tailed

va donc être prouvée très rapidement (i.e. après quelques conflits), ce qui évite de démontrer l'insatisfaisabilité du sous-problème des cavaliers pour chaque solution du sous-problème des reines.

1.3.1 Phénomène heavy-tailed

L'identification de structures propres à un problème est souvent une étape clef pour l'efficacité des algorithmes de résolution associés comme pour la compréhension de la complexité du problème. Dans [Gomes *et al.*, 2000] les auteurs étudient le comportement de différentes instances de problèmes de satisfaisabilité booléenne (communément appelé problème SAT) et de problèmes de satisfaction de contraintes sur plusieurs exécutions différentes générées aléatoirement. Les auteurs ont mis en évidence des performances très variables pour résoudre certaines instances qui peuvent être caractérisées par des moments quasi-infinis (i.e. temps d'exécution moyen très élevé). Ce type de distribution possède des propriétés singulières souvent caractérisées par une longue traînée (ou queue : heavy-tailed). Cette étude permet d'établir pour chaque instance, une courbe représentative du nombre cumulé d'exécutions en fonction du nombre de retours-arrière nécessaires à la résolution du problème. Un même problème peut alors soit être résolu très rapidement, soit au contraire ne pas être résolu en un temps raisonnable selon l'approche utilisée pour le résoudre.

Pour une instance i donnée, en traçant une courbe représentant sur l'axe des abscisses le nombre de backtracks nécessaires pour résoudre cette instance et en ordonnée le nombre cumulé d'exécutions ayant résolu le problème, on peut facilement mettre en évidence le phénomène heavy-tailed. La figure 1.10 montre l'allure d'une telle courbe sur un exemple de problème d'emploi

du temps. On observe le phénomène heavy-tailed sur la partie droite de la courbe. Il s'agit d'une sorte de traînée caractéristique. Les courbes ont souvent une allure logarithmique. On ne représente pas le nombre cumulé d'exécutions réussies mais son complément à un. La figure 1.11 montre l'allure logarithmique des courbes obtenues sur une instance présentant un comportement heavy-tailed (a) et ne présentant pas un comportement heavy-tailed (b). La droite représentée sur la figure (a) est typique d'un phénomène heavy-tailed.

1.3.2 Redémarrage et politique de redémarrage

Les techniques de redémarrage permettent d'éviter de tomber et de persister sur des exécutions du type heavy-tailed ou défavorables. Il est alors possible d'améliorer considérablement l'efficacité de la recherche en introduisant le concept de redémarrage et l'introduction d'un caractère aléatoire [Kautz *et al.*, 2002]. Si un algorithme de recherche ne se termine pas avant un nombre déterminé de retours-arrière autorisés (ou n'importe quel autre critère équivalent), appelé valeur de coupure (cutoff), l'exécution courante est alors interrompue et une nouvelle exécution est démarrée. L'introduction d'un phénomène aléatoire permet aux différentes exécutions de se comporter différemment. Cet aléa peut être utilisé notamment pour départager les variables à sélectionner (par une heuristique), et peut être initialisé avec une nouvelle graine (random seed) à chaque nouvelle exécution.

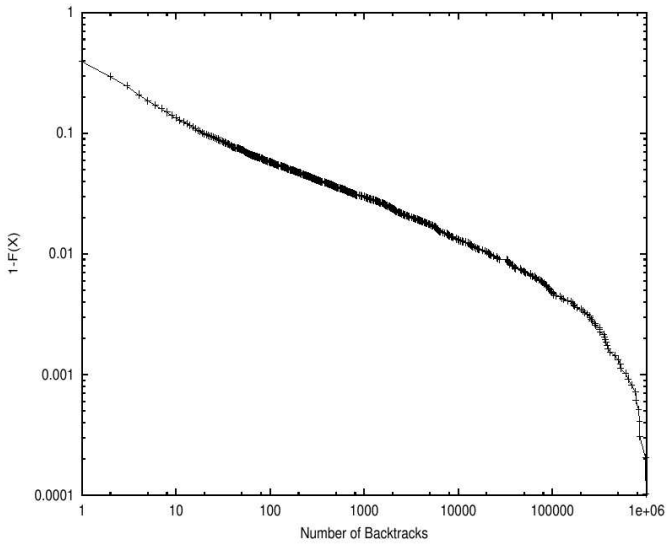
La valeur de coupure est mise à jour à chaque exécution. Plus précisément, si celle-ci est systématiquement incrémentée, la complétude de l'algorithme de recherche est préservée. Une politique de redémarrage détermine la fréquence des redémarrages à effectuer. On utilise en général deux paramètres pour définir une politique de redémarrage. Il faut en effet initialiser la première valeur de coupure (i.e. la valeur de coupure utilisée pour la première exécution) ainsi que déterminer le facteur d'accroissement. Ce facteur d'accroissement permet de fixer la prochaine valeur de coupure à utiliser. Ce facteur peut par exemple correspondre à une suite géométrique du type $C_i = \beta \times C_{i-1}$ où C_i est la valeur de coupure de l'exécution i [Walsh, 1999]. On définit alors la politique de redémarrage comme suit :

- $C_1 = \alpha$;
- $C_i = \beta \times C_{i-1}$.

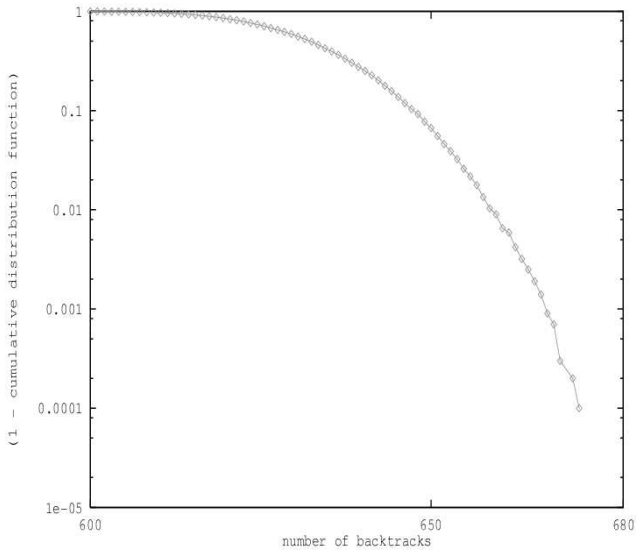
α correspond à la valeur de coupure initiale et β au facteur d'accroissement.

1.4 *Abscon* : un solveur CSP générique

Les approches présentées dans la suite de cette thèse ont été expérimentées en utilisant le solveur *Abscon109* [Lecoutre et Tabary, 2007] disponible à l'adresse `http://www.cril.univ-artois.fr/~lecoutre/research/tools/abscon.html`. Le solveur est dérivé d'*Abscon*, une plateforme générique pour la satisfaction de contraintes développée en Java (J2SE 5.0). Celui-ci, dans sa configuration initiale, maintient une consistance d'arc généralisé ($GAC3^{rm}$) pendant la recherche et explore l'espace de recherche en utilisant une heuristique de choix de variables "orientée vers les conflits" : *dom/wdeg*. Un algorithme de recherche avec retours-arrière chronologiques effectue une recherche en profondeur d'abord pour assigner les variables et utilise un mécanisme de retours-arrière lorsqu'une impasse apparaît. Cet algorithme est implémenté en utilisant un schéma de branchement binaire, les décisions positives étant prises en premier. Différentes techniques anti-thrashing (c.f. chapitre suivant) ont été implémentées et testées ainsi que différentes heuristiques de choix de variables (*brelaz*, *dom/ddeg*, ...). Le format de représentation utilisé pour la description des problèmes CSP dans *Abscon* est le XCSP [Boussemart *et al.*, 2005]. Ce format nous permet notamment (1) de pouvoir participer aux compétitions internationales



(a)



(b)

FIG. 1.11 – Mise en évidence du comportement heavy-tailed (échelle logarithmique)

de solveurs CSP et (2) d'avoir accés à une base de benchmarks de plus de 4 000 instances, aussi bien aléatoires, structurées que réelles. Ces instances sont disponibles à l'adresse : `http://www.cril.univ-artois.fr/~lecoutre/research/benchmarks/benchmarks.html`. Les expérimentations faites pour valider l'efficacité des approches présentées dans cette thèse ont été effectuées en utilisant ces benchmarks.

Abscon a été présenté aux compétitions internationales de solveur CSP 2005 et 2006. Notons par ailleurs que ce solveur a participé à la compétition 2006 en embarquant la pluspart des approches (c.f. notamment les chapitres 2 et 3) présentées dans cette thèse. Les résultats obtenus à ces compétitions sont disponibles aux adresses `http://cpai.ucc.ie/05/CPAI.html` et `http://www.cril.univ-artois.fr/CPAI06`.

2

Analyse des conflits

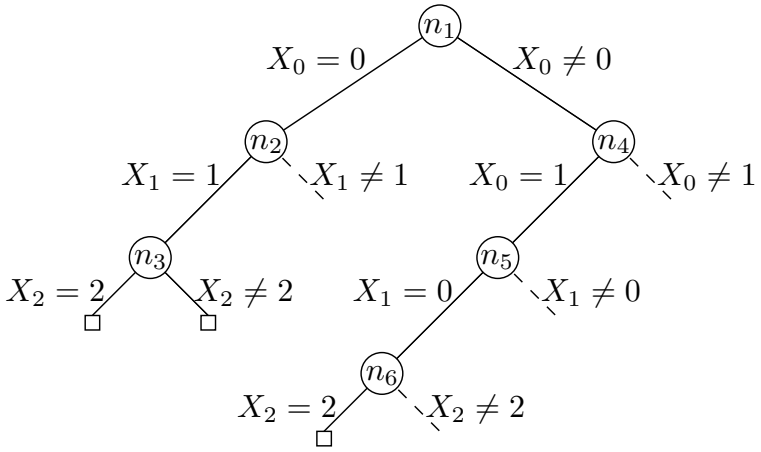
Très souvent les algorithmes de recherche ont tendance à être soumis au phénomène de "thrashing". Ce phénomène décrit en section 1.3 consiste à explorer de façon répétitive les mêmes sous-arbres et à éventuellement répéter les mêmes erreurs. Différentes techniques d'apprentissage, basées sur l'analyse des échecs, permettent aux algorithmes de recherche de minimiser les effets du "thrashing" sur la recherche. Ainsi en étudiant les raisons des échecs, on peut éviter que ceux-ci ne se reproduisent ultérieurement au cours de la recherche. Cette analyse est effectuée entre autre lorsqu'une impasse vient d'être rencontrée et que l'algorithme de recherche se prépare à effectuer un retour-arrière. Ces techniques sont généralement appelées techniques rétrospectives (lookback), puisqu'elles sont appliquées après l'apparition des conflits (par opposition aux techniques prospectives –lookahead– dont le but est de se diriger rapidement vers une solution). Elles permettent d'enrichir la connaissance qu'un algorithme de recherche peut avoir sur un problème, bien évidemment dans le but d'accélérer sa résolution.

L'analyse des raisons d'un échec permet d'identifier un nogood, c'est-à-dire un ensemble de décisions suffisant pour expliquer l'échec. Ce nogood peut être utilisé pour éviter que le même échec ne se répète plus tard au cours de la recherche (en enregistrant par exemple un nogood sous la forme d'une nouvelle contrainte), et/ou alors pour effectuer des retours-arrière dits intelligents (*backjumping*). Dans ce dernier cas, on effectue un retour-arrière sur la dernière décision responsable de l'échec et non pas systématiquement sur la dernière décision effectuée. On évite ainsi des retours-arrière sur des points de choix inutiles par rapport à la raison du conflit en effectuant des sauts dans l'arbre de recherche. Les premiers résultats expérimentaux obtenus par des algorithmes utilisant des techniques d'apprentissage sont apparus au début des années 1990 [Dechter, 1990, Prosser, 1993, Schiex et Verfaillie, 1994a, Frost et Dechter, 1994, Schiex et Verfaillie, 1994b].

2.1 Définitions préliminaires

Nous rappelons que l'application d'un ensemble de décisions (positives ou négatives) Δ sur un réseau de contraintes $P = (\mathscr{X}, \mathscr{C})$ permet de réduire les domaines des variables de P. Le réseau de contraintes ainsi obtenu, noté $P|_\Delta$ est un sous-réseau de P tel que $P|_\Delta \preceq P$. Ce sous-réseau est défini sur le même ensemble de variables $\mathscr{X}$ et de contraintes $\mathscr{C}$ et tel que $\forall X \in \mathscr{X}, dom^{P|_\Delta}(X) \subseteq dom^P(X)$.

Un tel ensemble de décisions Δ permet notamment de caractériser un nœud particulier de l'arbre de recherche (construit par exemple par un algorithme de recherche avec retours-arrière) grâce aux assignations et réfutations (dans le contexte d'un algorithme de recherche utilisant



$P = (\mathscr{X}, \mathscr{C})$:

- $\mathscr{X} = \{X_0, X_1, X_2, X_3, \}$ avec $dom(X_0) = dom(X_1) = dom(X_2) = dom(X_3) = \{0, 1, 2\}$;
- $\mathscr{C} = \left\{ \begin{array}{ll} C_0 : & X_0 \neq X_1 \\ C_1 : & X_0 \neq X_2 \\ C_2 : & X_0 \neq X_3 \\ C_3 : & X_1 \neq X_2 \\ C_4 : & X_1 \neq X_3 \\ C_5 : & X_2 \neq X_3 \end{array} \right\}$.

FIG. 2.1 – Caractérisation d'un nœud d'un arbre de recherche grâce à un ensemble de décision sur le problème des pigeons

un schéma de branchement binaire) effectuées sur la branche menant à ce nœud, autrement dit grâce à la séquence de décisions menant à ce nœud. Notons que si l'on considère un algorithme de recherche utilisant un schéma de branchement non-binaire, cet ensemble de décisions est restreint aux décisions positives. Ainsi à chaque nœud de l'arbre de recherche, on peut associer un sous-réseau de contraintes (du réseau de contraintes initial) restreint par la séquence de décisions menant à ce nœud.

La figure 2.1 représente un arbre partiel de recherche associé à la résolution du problème académique des *4-pigeons*. Ce problème est insatisfaisable puisqu'il consiste à placer 4 pigeons dans 3 boites, sachant qu'on ne peut placer qu'un seul pigeon par boite. La séquence de décisions $\Sigma = \langle \delta_1 : (X_0 \neq 0), \delta_2 : (X_0 = 1), \delta_3 : (X_1 = 0) \rangle$ prise le long d'une branche de l'arbre de recherche mène au nœud n_6. Ce nœud peut être caractérisé par le réseau de contraintes $P|_\Delta$ construit à partir de l'ensemble des décisions $\Delta = \{X_0 \neq 0, X_0 = 1, X_1 = 0\}$ de Σ. Le sous-réseau de contraintes obtenu après application de Δ est représenté par la figure 2.2.

Une impasse (ou échec) apparaît lorsqu'une variable X_i ne possède dans son domaine courant plus aucune valeur compatible avec l'instanciation partielle courante [Dechter, 1990]. Par exemple, après avoir assigné les variables X_0, X_1 et X_2, la variable X_3 ne possède plus dans son domaine de valeurs compatibles avec l'instanciation courante, ce qui mène donc à un échec.

Définition 16 (ensemble-conflit). *Soit P un réseau de contraintes et t une instanciation partielle consistante d'un sous-ensemble S de variables de P. $X \notin S$ est une variable en échec et t est un ensemble-conflit pour X si $\forall a \in dom(X)$, $P|_{t \cup \{X=a\}} = \bot$.*

Reprenons l'exemple précédent. Sur la figure 2.1, l'instanciation partielle $t = \{(X_0, 0), (X_1, 1), (X_2, 2)\}$ est un ensemble-conflit. Après avoir assigné les variables X_0, X_1 et X_2, plus aucune valeur du domaine de la variable X_3 n'est compatible avec t. En effet, on peut remarquer que :

- La contrainte C_2 rend la valeur 0 de X_3 incompatible avec l'assignation $X_0 = 0$;
- La contrainte C_4 rend la valeur 1 de X_3 incompatible avec l'assignation $X_1 = 1$;
- La contrainte C_5 rend la valeur 2 de X_3 incompatible avec l'assignation $X_2 = 2$.

X_3 est donc une variable en échec et t l'ensemble-conflit correspondant.

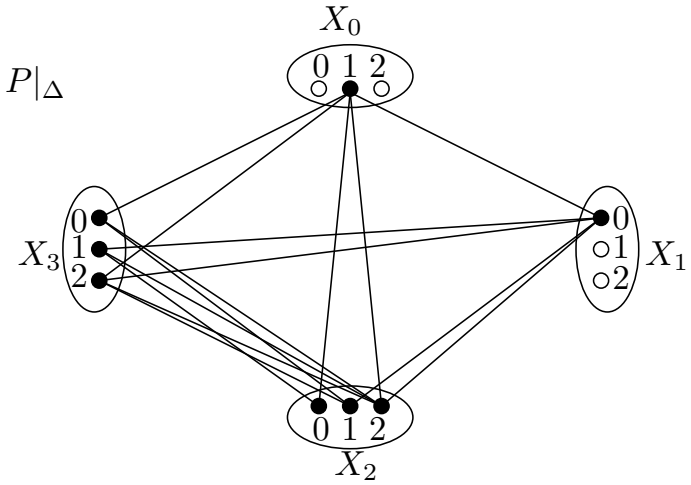


FIG. 2.2 – Sous-réseau obtenu après application de $\Delta = \{X_0 \neq 0, X_0 = 1, X_1 = 0\}$ sur le réseau de contraintes P

2.2 Identifier les raisons d'un conflit

2.2.1 Les nogoods

Un nogood est un ensemble de décisions qui n'apparaît dans aucune solution d'un problème P donné [Stallman et Sussman, 1977]. En d'autres termes, un nogood correspond à une instanciation partielle qui mènera toujours à un échec (soit directement, soit après avoir effectué une recherche) et ne pourra donc jamais être étendu à une solution de P. Par exemple étant donné un réseau de contraintes P, une instanciation partielle qui viole une contrainte de P est un nogood. Plus généralement, les ensembles conflits (définis en section 2.1) sont des nogoods. Cependant la réciproque n'est pas vraie. Il existe des nogoods qui expriment un conflit entre plusieurs variables simultanément et qui dans ce cas ne sont pas des ensembles conflits. Considérons à nouveau le réseau de contraintes décrit par la figure 2.1. Le problème des pigeons est un problème globalement insatisfaisable. Quelque soit la valeur (par exemple 0) affectée à la variable X_0, celle-ci mènera donc toujours à un échec : $\Delta = \{X_0 = 0\}$ est donc un nogood du problème. Pourtant Δ n'est pas un ensemble-conflit puisqu'on ne peut pas trouver une autre variable dont le domaine n'est pas compatible avec cette unique assignation.

On peut étendre la notion de nogoods lorsqu'on utilise un opérateur d'inférence établissant une consistance ϕ préservant la satisfaisabilité.

Définition 17 (nogood). *Soit P un réseau de contraintes, Δ un ensemble de décisions et ϕ une consistance.*

- *Δ est un nogood de P si et seulement si $P|_\Delta$ est insatisfaisable ;*
- *Δ est un ϕ-nogood de P si et seulement si $\phi(P|_\Delta) = \bot$;*
- *Δ est un ϕ-nogood minimal de P si et seulement si $\nexists \Delta' \subset \Delta$ tel que $\phi(P|_{\Delta'}) = \bot$.*

Les nogoods de taille 1 sont des nogoods particuliers. Les propriétés identifiant de tels nogoods sont appelées consistances de domaine (*domain filtering consistencies*). Ils sont très intéressants dans la mesure ou il est facile de les exploiter. En effet ces nogoods de taille 1 correspondent à des valeurs inconsistantes, qu'il suffit de supprimer des domaines des variables. Plus généralement,

lorsqu'un échec est rencontré, on peut extraire un nogood correspondant à l'instanciation partielle menant à cet échec.

Un nogood est violé par un instanciation t si celui-ci est inclus dans t. L' instanciation t mènera donc toujours à un échec (puisque celle-ci est un sur-ensemble d'un nogood).

Définition 18 (nogood violé)**.** *Soit P un réseau de contraintes, Δ un nogood de P et t un ensemble de décisions. Δ est violé par t si et seulement si $P|_t \preceq P|_\Delta$.*

En pratique, un nogood n'est pas exploité sous la forme d'une conjonction de décisions mais de manière équivalente, sous la forme de la négation d'une disjonction de décisions. En effet comme au cours d'une recherche l'instanciation partielle ne doit pas devenir un sur-ensemble d'un nogood déjà rencontré, il est préférable de stocker les nogoods sous une forme négative. Cette représentation facilite le processus de déduction car les décisions inférées à partir des nogoods ne nécessitent alors aucune transformation (i.e. négation). Plus précisément, si on considère un nogood Δ représenté sous la forme d'une conjonction de décisions, pour éviter que celui-ci ne soit violé il faut prendre la négation d'au moins une des décisions de Δ.

Les *nogoods standard* ont été introduit dans [Dechter, 1990]. Un nogood standard est un ensemble de décisions uniquement positives, c'est-à-dire du type $X = a$, ne menant à aucune solution.

Définition 19 (nogood standard)**.** *Soit P un CN et Δ un nogood de P. Δ est un nogood standard si et seulement si $\forall \delta \in \Delta, \delta$ est une décision positive.*

Etant donné un nogood standard $\Delta = \{X_1 = a_1, \ldots, X_k = a_k\}$, une variable X_i apparaissant dans une décision de Δ peut être sélectionnée et le nogood Δ réécrit sous la forme :

$$\bigwedge_{j \in [1..k] \setminus i} (X_j = a_j) \rightarrow X_i \neq a_i$$

Classiquement, la partie gauche de l'implication constitue une explication du retrait (i.e. de la suppression) [Jussien *et al.*, 2000] de la valeur a_i du domaine de la variable X_i (notée $expl(X_i \neq a_i)$). Une autre représentation [Ginsberg, 1993] consiste à considérer l'explication d'un retrait d'une valeur a_i du domaine de X_i comme la donnée de deux sous-ensembles $C_e \subseteq \mathscr{C}$ et $DC_e \subseteq DC$ où $\mathscr{C}$ représente l'ensemble des contraintes du problème et DC l'ensemble des décisions effectuées. On a alors $C_e \wedge DC_e \rightarrow X_i \neq a_i$. Remarquons cependant que dans les travaux présentés dans cette thèse, il n'est pas nécessaire de distinguer l'explication d'un retrait et le retrait lui même.

Dans [Katsirelos et Bacchus, 2005], les auteurs étendent la notion de nogood standard en introduisant la définition de *nogood généralisé*. Dans un nogood généralisé, les décisions peuvent apparaître aussi bien positivement (i.e sous la forme $X = a$) que négativement (i.e. sous la forme $X \neq a$). La définition d'un nogood généralisé rejoint notre définition générale (c.f. Définition 17).

Définition 20 (nogood généralisé)**.** *Soit P un CN et Δ un nogood de P. Δ est un nogood généralisé si et seulement si $\forall \delta \in \Delta, \delta$ est une décision positive ou négative.*

Un nogood généralisé capture de manière compacte un ensemble (éventuellement de taille exponentielle) de nogoods standard. Ces nogoods sont donc plus expressifs que les nogoods standard.

Dans l'exemple suivant, on considère un réseau de contraintes $P = (\mathscr{X}, \mathscr{C})$ tel que :

- $\mathscr{X} = \{X, Y, Z, T\}$ avec $dom(X)$=$\{0, \ldots, 99\}$, $dom(Y) = \{0, 1, 2\}$ et $dom(Z) = dom(T) = \{1, 2, 3\}$;

– $\mathscr{C} = \left\{ \begin{array}{ll} C_0 : & X + Y < Z \\ C_1 : & Z = T \end{array} \right\}$.

Si la valeur 1 est assignée à la variable T, après réduction des domaines, la variable Z est également assignée à 1 ce qui satisfait la contrainte C_1. On peut alors remarquer que la contrainte C_0 est falsifiée pour n'importe quelle valeur de X strictement supérieure à 0. En effet 0 est la plus petite valeur que l'on puisse assigner à la variable Y et sous cette hypothèse la contrainte C_0 ne peut être vérifiée que si l'on affecte la valeur 0 à la variable X. On peut donc en déduire la liste de nogoods standard suivante :

– $\Delta_1 = \{X = 1, Z = 1, T = 1\}$;
– $\Delta_2 = \{X = 2, Z = 1, T = 1\}$;
– ...
– $\Delta_{98} = \{X = 98, Z = 1, T = 1\}$;
– $\Delta_{99} = \{X = 99, Z = 1, T = 1\}$.

Cependant les 99 nogoods standard énumérés précédemment peuvent être exprimés à l'aide d'un seul nogood généralisé :$\Delta_i = \{X \neq 0, Z = 1, T = 1\}$. A l'aide d'un seul nogood généralisé, on capture alors les 99 nogoods standard précédent. Non seulement ceci permet un gain d'espace (puisqu'on enregistre un seul nogood) mais aussi un gain de temps. Remarquons tout de même que dans le pire des cas, la taille d'un nogood généralisé est bornée par nd tandis que celle-ci est bornée par n pour un nogood standard. Au moment d'exploiter ces nogoods, par exemple au cours de la recherche et étant donné un ensemble de décisions t (prises le long d'une branche d'un arbre de recherche), au lieu de vérifier que les 99 nogoods standard ne sont pas violés par t, il est suffisant de vérifier que le nogood généralisé Δ_i n'est pas violé par t.

Les nogoods généralisés bénéficient de propriétés intéressantes. Ils ont notamment une capacité d'élagage supérieure (dans le contexte d'un arbre de recherche) à celle des nogoods standard. Ils peuvent être "déclenchés" (i.e. utilisés pour faire de l'inférence) même si les variables de la branche courante ne sont pas encore assignées, i.e. même si les variables de la branche courante correspondent à des réfutations, ce qui est impossible avec un nogood standard ou n'apparaissent que des décisions positives. Les coupures dans l'arbre de recherche peuvent donc avoir lieu plus tôt. Un nogood généralisé peut élaguer éventuellement des branches d'un arbre de recherche qu'un ensemble de nogoods standard ne peut détecter. Pour illustrer ce propos, envisageons le nogood $\Delta = \{X = 1, X_0 \neq 0, X_1 \neq 0\}$ appris précédemment au cours de la recherche et l'arbre partiel de recherche représenté par la figure 2.3. Après avoir réfuté $X_0 = 0$ et $X_1 = 0$, l'assignation $X = 1$ est automatiquement évitée puisque dans le cas contraire le nogood Δ aurait été violé. L'utilisation de nogoods standard (comme par exemple $\Delta' = \{X = 1, X_0 = 1, X_1 = 1\}$) aurait nécessité l'assignation (d'une valeur autre que 0) des variables X_0 et X_1 pour pouvoir couper la branche correspondant à l'assignation $X = 1$.

2.2.2 Exploiter les nogoods

Contrairement au domaine de la satisfaction de contraintes, les progrès récents effectués au sein de la communauté SAT (vérification de la satisfaisabilité de formules booléennes) ont été permis grâce à l'enregistrement de nogoods (apprentissage de clauses) dans le cadre d'une politique de redémarrage et en utilisant une structure de données paresseuse très efficace : les "watched literals" [Zhang *et al.*, 2001]. En effet, l'intérêt pour l'apprentissage de clauses a augmenté avec la mise à disposition de larges instances (encodant des applications pratiques) qui contiennent certaines structures et dans lesquelles on peut mettre en évidence un phénomène heavy-tailed. L'apprentissage en SAT est une technique efficace, obtenue à partir de

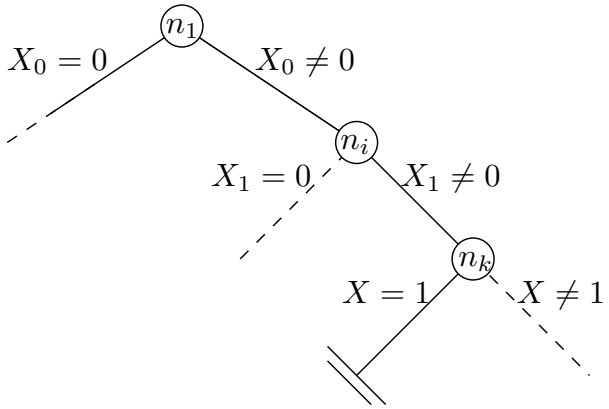


FIG. 2.3 – Arbre de recherche partiel, nogood généralisé et inférence

la fertilisation croisée des deux domaines de recherche SAT et CSP : l'enregistrement de nogoods a été tout d'abord introduit en CSP puis plus tard importé dans les solveurs SAT [Bayardo et Shrag, 1997, Marques-Silva et Sakallah, 1996].

Les nogoods sont exploités pour élaguer des branches d'un arbre de recherche dans lesquelles aucune solution ne peut apparaître. L'idée principale est d'identifier un ensemble de décisions menant à une impasse et d'enregistrer un nogood correspondant à l'explication de cet échec. Par la suite, on peut détecter si l'instanciation partielle courante inclut l'ensemble des décisions d'un nogood précédemment enregistré et si tel est le cas, on peut directement élaguer cette branche de l'arbre de recherche (puisque l'instanciation partielle courante est un sur-ensemble d'un nogood). Plusieurs techniques sont utilisées, l'objectif étant de concilier le coût supplémentaire en temps et en espace induit par l'enregistrement des nogoods avec les capacités d'élagage de l'arbre que ces nogoods permettent. Ils évitent ainsi en partie l'apparition du phénomène de thrashing. A chaque fois qu'un algorithme de recherche avec retours-arrière rencontre une impasse, c'est-à-dire à chaque fois que l'instanciation partielle courante est identifiée comme étant un ensemble-conflit, on peut enregistrer un nogood. Celui-ci sera exploité plus tard (sous réserve de certaines conditions : minimisation du nogood, redémarrage de l'algorithme de recherche) au cours de la recherche pour éviter que des conflits similaires ne réapparaissent. Ce nogood représente alors une explication du conflit qui vient d'être rencontré.

Les nogoods peuvent être considérés comme des contraintes supplémentaires du problème, qu'il faut satisfaire au même titre que les contraintes définies dans le réseau de contraintes initial. La taille d'un nogood standard est bornée par le nombre de variables du problème. Plus précisément, cette taille est limitée par le nombre maximal d'assignations possibles effectuées sur une branche de l'arbre de recherche, soit n dans le pire des cas. Dans le cas de nogoods généralisés, la taille des nogoods est bornée par nd, c'est-à-dire par le nombre maximal de réfutations possibles effectuées sur une branche de l'arbre de recherche. Comme n variables différentes peuvent apparaître dans un même nogood, l'arité de la contrainte (associée à ce nogood) peut donc être dans le pire des cas de n. Chaque nogood peut être assimilé à une nouvelle contrainte dont la portée correspond au nombre de variables qu'il implique. Appliquer un algorithme générique de filtrage, comme celui établissant GAC, sur ce type de contrainte n'est pas vraiment envisageable du fait de la (possible) grande arité des contraintes. Une solution est de limiter la taille des nogoods (et donc l'arité des contraintes associées), ou alors d'utiliser un algorithme de filtrage spécifique (comme dans le cas des contraintes globales). De plus si on assimile un nogood à une nouvelle contrainte, cela suppose une modification du graphe de contraintes associé au problème

et cela peut éventuellement influer sur le comportement des heuristiques de choix de variables (notamment celles basées sur le degré des variables). Néanmoins son intégration au sein d'un algorithme comme MGAC est assez naturelle et l'exploitation de ces nogoods est donc très facile à mettre en oeuvre.

Dans le contexte d'un algorithme de recherche avec retours-arrière, enregistrer un nogood correspondant à une instanciation partielle menant à une impasse n'est pas intéressant puisque les nogoods ne pourront pas être rencontrés une deuxième fois au cours de la recherche. En effet la construction de l'arbre de recherche (quelque soit le schéma de branchement binaire ou non-binaire utilisé) nous assure qu'une même instanciation partielle ne peut être rencontrée plusieurs fois au cours de la recherche. Considérons le nogood $\Delta = \{X = a, Y = b\}$ extrait à partir d'une instanciation partielle $\Sigma = \{(X, a), (Y, b)\}$ (correspondant à une branche de l'arbre de recherche) et démontrée inconsistante. Un algorithme de recherche utilisant un schéma de branchement binaire réfutera la dernière décision $Y = b$ prise dans Σ, puis la précédente $X = a$, etc. De manière similaire, un algorithme de recherche utilisant un schéma de branchement non-binaire assignera une autre valeur pour Y, puis une autre valeur pour X, etc. L'instanciation partielle Σ ne sera donc jamais rencontrée une nouvelle fois au cours de la recherche. Cette propriété assure d'ailleurs la terminaison de la recherche. Ces nogoods peuvent néanmoins être rencontrés à nouveau si l'on considère un algorithme de recherche effectuant des redémarrages. Dans ce cas, les nogoods extraits dans une exécution donnée peuvent être exploités lors des exécutions suivantes.

Comme le nombre de nogoods enregistrés peut être de taille exponentielle, il est parfois nécessaire d'effectuer certaines restrictions pour pouvoir traiter des problèmes de grande taille. Le nombre de nogoods croît exponentiellement avec la taille du problème et il faut éventuellement limiter leur enregistrement à un sous-ensemble. Par exemple en SAT, les nogoods appris ne sont pas minimaux (c.f. section 2.2.3) et on limite l'enregistrement d'un nogood par échec en utilisant le concept du "First Unique Implication Point (First UIP)" [Zhang *et al.*, 2001]. D'autres approches ont été proposées (e.g. [Bayardo et Shrag, 1997]), chacune d'entre elles essayant d'obtenir le meilleur compromis entre le surcoût dû à l'apprentissage et les améliorations des performances. Classiquement on fixe une borne k et on limite l'enregistrement aux nogoods de taille inférieure à k. On parle alors d'apprentissage d'ordre k [Dechter, 1990]. Cette méthode se base sur le principe que plus un nogood est de petite taille, plus son pouvoir d'élagage est important. De plus le nombre de nogoods est alors borné par (d^k) où d correspond à la taille du plus grand domaine. D'autres techniques basées sur la topologie du réseau ou la pertinence (relevance) des nogoods [Ginsberg, 1993] peuvent également être envisagées. Tous les nogoods n'étant pas enregistrés, ils ne peuvent mener à une élimination totale de la redondance dans les arbres de recherche.

2.2.3 Minimisation des nogoods

Présentation

Pour qu'un nogood extrait à partir d'une instanciation partielle menant à une impasse puisse être réutilisé au cours d'une même exécution, une autre approche consiste à isoler dans celui-ci les raisons de l'échec [Petit *et al.*, 2003]. Autrement dit, toutes les décisions d'un nogood Δ ne participent pas forcément aux causes de l'échec et il est parfois possible d'identifier un sous-ensemble de Δ suffisant (mais pas forcément minimal) pour expliquer le conflit. Identifier un nogood minimal Δ revient alors à déterminer s'il n'existe pas un ensemble de décisions $\Delta' \subset \Delta$ tel que Δ' soit également un nogood. Remarquons que la notion de nogood minimal est différente

de la notion de nogood minimum. Un nogood minimal est un nogood dans lequel toutes les décisions participent à l'explication de l'échec. Autrement dit, si l'on supprime une décision de ce nogood, celui-ci ne mène plus forcément à une impasse et par conséquent ce n'est plus un nogood. Un nogood minimum est le plus petit nogood (en terme de taille) qu'il est possible d'extraire et expliquant l'échec. Un nogood minimum est minimal mais la réciproque n'est pas forcément vraie.

Traditionnellement lorsqu'on s'intéresse à l'identification de nogoods minimaux, on parle de techniques d'apprentissage "profondes" (*deep learning*) alors que l'identification de nogoods non-minimaux correspond à un apprentissage de "surface" (*shallow learning*).

Identifier des nogoods minimaux nécessite une analyse plus coûteuse et plus fine des ensembles conflits. Néanmoins ceux-ci sont plus intéressants puisqu'ils possèdent une capacité d'élagage supérieure et que ceux-ci peuvent être "déclenchés" (i.e. utilisés pour faire de l'inférence) plus tôt au cours de la recherche. De plus les nogoods non-minimaux nécessitent plus d'espace mémoire pour être enregistrés et il est plus coûteux de les utiliser (puisqu'ils impliquent plus de variables).

La figure 2.4 illustre ce propos sur le réseau de contraintes P défini par :
- $\mathscr{X} = \{X, Y, Z, T\}$ avec $dom(X) = dom(Y) = dom(Z) = dom(T) = \{0, 1\}$;
- $\mathscr{C} = \left\{ \begin{array}{ll} C_0 : & Y \geq X \\ C_1 : & X \neq Z \\ C_2 : & X \neq T \\ C_3 : & Z \neq T \end{array} \right\}$.

L'instanciation $i = \{(X, 0), (Y, 0), (Z, 1)\}$ mène à une impasse, puisque la variable T ne possède pas de valeurs compatibles avec l'instanciation i. On peut donc associer le nogood standard $\Delta = \{X = 0, Y = 0, Z = 1\}$ à cet échec. Si on analyse plus précisément les raisons de cet échec, on remarque que la décision $Y = 0$ ne fait pas partie de l'explication du conflit. En effet si le domaine de la variable T est vide après l'instanciation i c'est parce que (1) la valeur 0 de T est

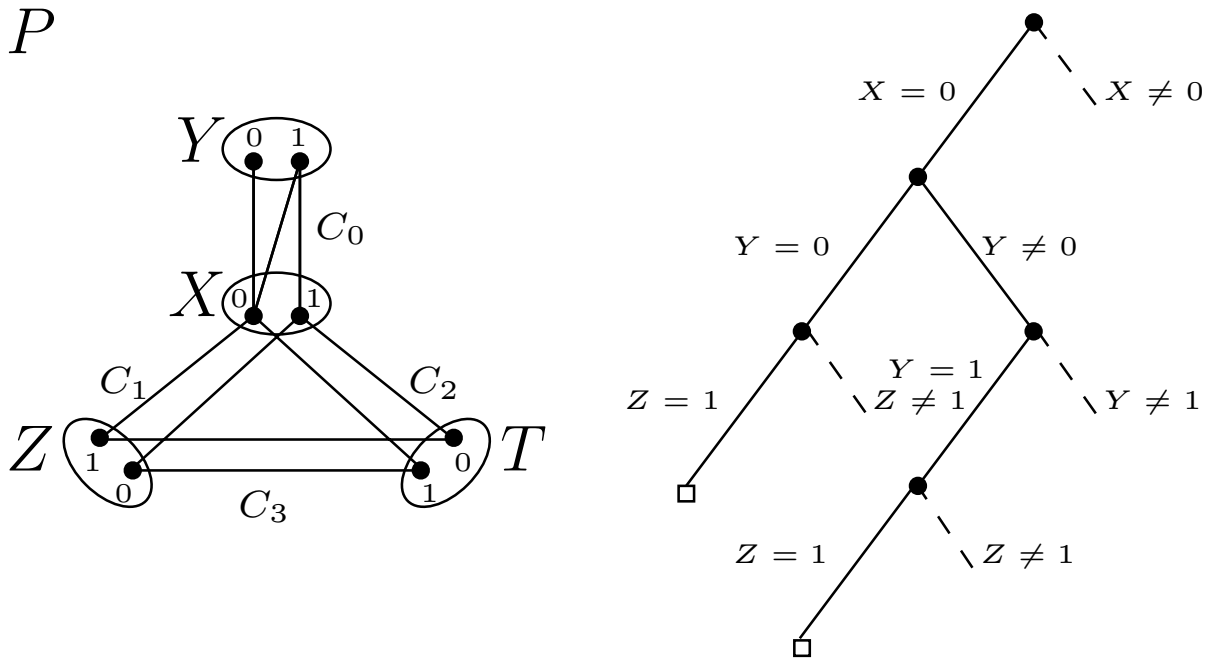


FIG. 2.4 – Capacité d'élagage des nogoods minimaux et non-minimaux

incompatible avec la décision $X = 0$ et (2) la valeur 1 de T est incompatible avec la décision $Z = 1$. Le nogood $\Delta' = \{X = 0, Z = 1\}$ est donc suffisant pour expliquer ce conflit.

Remarque :

- Comme $\Delta' \subset \Delta$, Δ n'est pas un nogood minimal ;
- Δ' est un nogood minimal, puisque $\Delta_1 = \{X = 0\}$ et $\Delta_2 = \{Z = 1\}$ ne sont pas des nogoods de P ($P|_{\Delta_1} \neq \bot$ et $P|_{\Delta_2} \neq \bot$).

Si l'on considère une base de connaissance (c'est-à-dire une base de nogoods) $\mathscr{B} = \{\Delta\}$, l'instanciation $i' = \{(X, 0), (Y, 1), (Z, 1)\}$ menant pourtant également à un échec ne peut pas être évitée puisque le nogood Δ n'est pas déclenchable. Le nogood Δ n'est pas violé du fait de la décision $Y = 0$ de Δ qui n'est pas falsifiée sous l'hypothèse de l'instanciation i' (puisque la valeur 1 et non 0 est assignée à la variable Y). Considérons maintenant que la base de nogoods $\mathscr{B}$ ne contienne plus Δ mais le nogood (minimal) Δ'. L'impasse survenue après l'instanciation i' peut être évitée puisque cette fois-ci Δ' est violé par i'. Le nogood Δ' peut cette fois être déclenché et la branche courante de l'arbre de recherche être élaguée.

Techniques de minimisation

Etant donné un nogood Δ, on peut identifier tous les nogoods minimaux inclus dans Δ par énumération. Pour cela, il faut déterminer dans un premier temps tous les nogoods de taille 1 construits à partir des décisions apparaissant dans Δ. Puis, tous les nogoods de taille 2, etc.

Même si les nogoods minimaux sont plus informatifs, cette approche est extrêmement coûteuse et il n'est pas envisageable en pratique d'identifier tous les nogoods minimaux construits à partir d'un ensemble-conflit donné. Calculer un tel ensemble peut mener à une explosion aussi bien en temps qu'en espace.

Comme extraire un nogood minimal est une activité limitée à une branche de l'arbre de recherche, l'identification de tels nogoods peut également être envisagée en utilisant une approche constructive, destructive ou alors dichotomique. Le principe est d'identifier itérativement les *décisions de transition*, c'est-à-dire les décisions qui font obligatoirement partie des raisons de l'échec. Pour cela on peut construire une instanciation partielle composée uniquement des décisions de transition expliquant l'échec. Plusieurs approches, inspirées de l'algorithme *Xplain* [de Siqueira et Puget, 1988], ont été proposées pour identifier les décisions de transition. Notons notamment les approches constructive, destructive et dichotomique intitulées RobustXplain, ReplayXplain et QuickXplain [Junker, 2001, Junker, 2004].

L'algorithme 7 illustre l'identification d'un nogood minimal par une approche constructive (*Xplain*) à partir d'un réseau de contraintes et d'une séquence de décisions Δ associée. Le nogood minimal Δ_{exp} est construit en identifiant itérativement les décisions de transition qui le composent. A chaque tour de boucle (ligne 5 à 8) on identifie une nouvelle décision de transition δ_k. Pour cela, on construit une instanciation partielle Δ_{tmp} composée des décisions de transition précédemment identifiées et des décisions de la séquence Δ (ligne 7 et 11). Si cette instanciation partielle est inconsistante (par rapport à un opérateur d'inférence établissant une consistance ϕ), la dernière décision δ_k ajoutée représente une décision de transition que l'on ajoute à l'ensemble Δ_{exp} (ligne 10). L'algorithme se termine lorsque $\phi(P|_{\Delta_{exp}})$ est inconsistant. Un ensemble minimal de décisions de transition est alors identifié. Si la séquence de décisions Δ est consistante, aucune décision de transition ne peut être identifiée et l'algorithme retourne dans ce cas l'ensemble vide (ligne 9).

Algorithme 7 : *Xplain*

Entrées : $\Delta = \langle \delta_1, \ldots, \delta_n \rangle$: Séquence de décisions, $P = (\mathscr{X}, \mathscr{C})$: CN
Sorties : ensemble de décisions

```
1  Δ_exp = ∅ ;
2  Δ_tmp = ∅ ;
3  tant que φ(P|_Δexp) ≠ ⊥ faire
4  |  k = 0 ;
5  |  tant que (φ(P|_Δtmp) ≠ ⊥) et (k < n) faire
6  |  |  k = k + 1 ;
7  |  |  Δ_tmp = Δ_tmp ∪ {δ_k} ;
8  |  fin
9  |  si P|_Δtmp ≠ ⊥ alors retourner ∅ ;
10 |  Δ_exp = Δ_exp ∪ {δ_k} ;
11 |  Δ_tmp = Δ_exp ;
12 fin
13 retourner Δ_exp ;
```

2.3 Les techniques de retours-arrière intelligents

2.3.1 Introduction

Présentation

Lorsqu'un algorithme de recherche avec retours-arrière rencontre une impasse, il remet en cause systématiquement le dernier point de choix effectué (c'est-à-dire la dernière décision prise). Ce comportement est appelé retour-arrière chronologique.

Imaginons maintenant le scénario suivant. Au cours de la recherche, un algorithme de recherche avec retours-arrière rencontre une impasse et remet en cause la dernière décision prise. Les raisons de cet échec sont expliquées par des décisions prises sur la branche menant à cette impasse. On peut raisonnablement supposer que toutes les décisions de la branche ne participent pas à l'explication de l'échec (d'où le principe de vouloir extraire des nogoods minimaux) et on peut faire l'hypothèse que la dernière décision prise (sur laquelle l'algorithme vient d'effectuer un retour-arrière) ne fasse pas partie de ces explications. Après avoir effectué le retour-arrière, l'algorithme poursuit sa recherche. Cependant les mêmes causes produisant les mêmes effets, un conflit similaire va être détecté tôt ou tard sur cette nouvelle branche. En résumé, il n'est pas toujours approprié de remettre en cause systématiquement la dernière décision de l'arbre de recherche lorsqu'un conflit est détecté, cette décision ne faisant pas forcément partie des raisons de l'échec précédent (c.f. section 2.2.3). Dans ces conditions, la remettre en cause n'empêchera pas l'apparition d'un conflit similaire.

Certains retours-arrière sont donc inutiles et il faut fournir un effort de recherche supplémentaire pour explorer les sous-arbres associés qui n'est pas nécessaire. Une technique de retours-arrière dits intelligents permet d'éviter ces retours-arrière systématiques en effectuant des "sauts" dans l'arbre de recherche sur les décisions identifiées comme responsable des échecs. On évite ainsi l'exploration de sous-arbres dans lesquels aucune solution ne peut apparaître mais il est nécessaire d'analyser les causes de l'échec pour identifier les variables qui en sont à l'origine. Ceci nécessite un surcoût de travail que l'on peut compenser par l'élagage de branches inutiles. Cette technique est appelée *backjumping* ou *backtracking intelligent*. Plus précisément, étant donné une

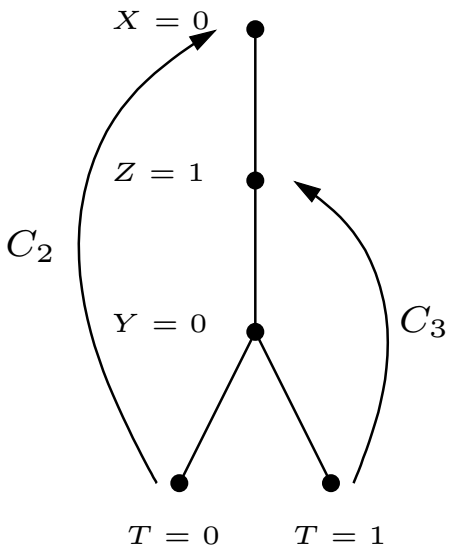


FIG. 2.5 – Retour-arrière intelligent

assignation partielle $t = \{X_1 = a_1, \ldots, X_k = a_k\}$ menant à une impasse (car la variable X_{k+1} ne possède pas de valeurs compatibles avec t), ce raisonnement se base sur le fait qu'il est possible d'identifier un préfixe de t de taille $j < k$: l'assignation partielle $t' = \{X_1 = a_1, \ldots, X_j = a_j\}$ ne peut alors être étendue à une solution quelque soit la valeur assignée à X_{k+1} et l'on peut donc effectuer un saut en toute sécurité sur la décision X_j et éviter d'explorer les branches intermédiaires. Les nogoods peuvent servir à identifier une telle décision $X_j = a_j$ dite *coupable*.

Le but de ce type d'algorithme est d'effectuer un saut à partir d'une variable X_{k+1} vers une variable X_j telle que l'instanciation partielle courante des variables $\{X_1, \ldots, X_j\}$ ne peut mener à aucune solution quelque soit la valeur assignée à la variable X_{k+1}. Dans ce cas, le retour-arrière effectué jusqu'à X_j est appelé une saut sûr (*safe jump*). L'efficacité d'un algorithme de backjumping dépend de sa capacité à effectuer les sauts les plus longs, c'est-à-dire à déterminer si un saut est sûr ou non. En règle générale, pour des raisons d'efficacité, ces algorithmes fournissent une approximation, c'est-à-dire qu'ils identifient à moindre coût, une variable sur laquelle il est possible d'effectuer un saut sûr.

Exemple

On considère le réseau de contraintes P défini en section 2.2.3 et dont le graphe de contraintes est représenté par la figure 2.4. La figure 2.5 illustre deux instanciations $i = \{(X, 0), (Z, 1), (Y, 0), (T, 0)\}$ et $i' = \{(X, 0), (Z, 1), (Y, 0), (T, 1)\}$ de P construites par un algorithme de recherche et menant toutes les deux à un échec. Le nogood $\Delta = \{X = 0, Z = 1, Y = 0\}$ est extrait de la branche courante lorsque l'algorithme de recherche effectue un retour-arrière et remet en cause la décision $Y = 0$.

Si Δ est un nogood c'est parce que toutes les valeurs de T sont incompatibles avec Δ. Plus précisément, (1) la valeur 0 de T est incompatible avec la décision $X = 0$ et (2) la valeur 1 de T est incompatible avec la décision $Z = 1$. En d'autres termes, la décision $Y = 0$ ne fait pas partie des raisons du conflit.

La décision $Z = 1$ est une décision coupable (au sens de la définition 25). Remettre en cause la

décision $Y = 0$ (en assignant la valeur 1 à Y par exemple) n'est pas utile puisque la variable Y ne fait pas partie des raisons du conflit. On peut donc en toute sûreté effectuer un *saut* dans l'arbre de recherche et remettre en cause directement la décision $Z = 1$ identifiée comme la décision coupable. Il n'est pas nécessaire de prouver l'insatisfaisabilité du sous-arbre correspondant à l'assignation $Y = 1$ et on peut en toute sûreté élaguer cette branche.

2.3.2 Gaschnig backjumping

Présentation

Etant donné une séquence de décisions $\Sigma = \langle \delta_1, \ldots, \delta_k \rangle$ menant à un échec car la variable X_{k+1} ne possède pas dans son domaine de valeur compatible avec l'instanciation partielle courante $t = \{\delta_1, \ldots, \delta_k\}$, t est par définition un ensemble-conflit et donc un nogood. La technique présentée ici permet d'identifier un préfixe de Σ, dont l'instanciation partielle associée est également un nogood.

Lorsqu'une impasse est rencontrée, cette technique introduite dans [Gaschnig, 1979] identifie une variable dite "coupable" (au sens de Gaschnig) et effectue un saut sur celle-ci. Pour cela une analyse du domaine de la variable en échec (ici X_{k+1}) est effectuée afin de déterminer la longueur du saut maximal, i.e. déterminer la décision "coupable" la plus haute dans l'arbre de recherche à remettre en cause. Plus précisément cet algorithme sélectionne tout d'abord pour chaque valeur a du domaine de X_{k+1} la décision (de l'instanciation partielle courante) la plus ancienne incompatible avec a. L'ensemble des décisions ainsi identifié est utilisé dans un second temps pour effectuer un saut sur la décision (de cet ensemble) la plus récente (i.e. la décision la plus basse dans la branche de l'arbre de recherche menant à cette impasse).

Pour la définition et la proposition suivante, on considérera donné un réseau de contraintes P et $\Sigma = \langle \delta_1, \ldots, \delta_k \rangle$ une séquence de décisions telle que l'instanciation partielle $\Delta = \{\delta_1, \ldots, \delta_k\}$ est un ensemble-conflit pour une variable X_{k+1}.

Définition 21 (décision coupable au sens de Gaschnig)**.** *La décision coupable (au sens de Gaschnig) relative à Σ est définie par δ_i telle que $i = min(j \leq k \mid$ l'instanciation $\{\delta_1, \ldots, \delta_j\}$ est un ensemble-conflit pour $X_{k+1})$.*

Proposition 1. *Si δ_i est la décision coupable (au sens de Gaschnig) relative à Σ, alors le saut effectué sur la décision δ_i est sûr.*

En d'autre terme, cela revient à déterminer le préfixe de Σ le plus petit tel que l'instanciation partielle associée à ce préfixe soit un ensemble-conflit pour X_{k+1}.

La figure 2.6 illustre sur un exemple l'identification d'une variable coupable au sens de Gaschnig. L'instanciation partielle $t = \{\delta_1, \ldots, \delta_k\}$ associée à la séquence de décisions $\Sigma = \langle \delta_1, \ldots, \delta_k \rangle$ mène à un échec sur la variable X_{k+1}. $\Delta = \{\delta_1, \ldots, \delta_k\}$ est par conséquent un ensemble-conflit. Pour identifier la décision coupable, on détermine dans un premier temps pour chaque valeur a du domaine de X_{k+1} le plus petit préfixe $\langle \delta_1, \ldots, \delta_i \rangle$ (avec $i \leq k$) incompatible avec l'assignation $X_{k+1} = a$.

Sur l'illustration précédente, pour chaque valeur a du domaine de X_{k+1}, une flèche délimite l'instanciation partielle (de la branche courante) à partir de laquelle celle-ci n'est plus compatible avec la valeur a de X_{k+1}. Pour chaque valeur de X_{k+1} on peut alors identifier les préfixes suivants :

- $\langle \delta_1, \delta_2 \rangle$ est le plus petit préfixe incompatible avec la valeur a_{k+1} (i.e $P|_{\{\delta_1, \delta_2, X_{k+1} = a_{k+1}\}} = \bot$ et $P|_{\{\delta_1, X_{k+1} = a_{k+1}\}} \neq \bot$;
- $\langle \delta_1, \delta_2, \delta_3 \rangle$ est le plus petit préfixe incompatible avec la valeur a_2 ;
- $\langle \delta_1, \ldots, \delta_i \rangle$ est le plus petit préfixe incompatible avec la valeur a_1.

Le préfixe $\langle \delta_1, \ldots, \delta_i \rangle$ est le plus grand préfixe incompatible avec la variable X_{k+1}. On en déduit donc que l'instanciation partielle $\{\delta_1, \ldots, \delta_i\}$ est un ensemble-conflit pour X_{k+1} et que le retour-arrière effectué sur la décision δ_i est un saut sûr.

Remarquons que cette technique ne permet d'identifier des sauts sûrs que lorsqu'une impasse apparaît, c'est-à-dire à partir des feuilles de l'arbre de recherche. Il ne peut donc y avoir au maximum qu'un seul saut par branche de l'arbre.

Algorithme

En pratique on associe à chaque variable (éventuellement en échec) un pointeur noté *latest* qui identifie la position (dans la séquence de décisions courante) de la décision coupable (au sens de Gaschnig) d'un éventuel ensemble-conflit pour cette variable. Autrement dit, ce pointeur délimite le préfixe (de la séquence de décisions associée à la branche courante) de longueur minimale conflictuel avec la variable que l'on tente d'instantier. *latest* marque la plus récente décision de la branche courante (i.e. la décision prise la plus récemment) inconsistante avec au moins une valeur de la variable que l'on essaye d'instantier. Si une impasse est finalement détectée (c'est-à-dire que la variable courante ne possède aucune valeur compatible avec l'instanciation partielle), ce marqueur fournit la position d'une décision sûre exploitable pour effectuer un retour-arrière intelligent.

A chaque étape de la recherche, on tente d'instantier une nouvelle variable notée X. Deux cas sont alors envisageables. Si cette variable possède dans son domaine une valeur compatible avec l'instanciation partielle courante, *latest* est réinitialisé à 0 et la recherche se poursuit en sélectionnant une nouvelle variable à assigner. Si toutes les valeurs de X sont incompatibles avec l'instanciation partielle courante, on peut identifier un ensemble-conflit (correspondant à l'instanciation partielle courante). Le pointeur *latest* identifie alors la décision (coupable) la plus récente de l'arbre de recherche incompatible avec au moins une valeur de X. En pratique le calcul de la décision coupable ne s'effectue pas lorsque l'impasse est détectée mais la mise à jour du pointeur *latest* est faite au fur et à mesure, suite aux essais successifs des différentes valeurs de

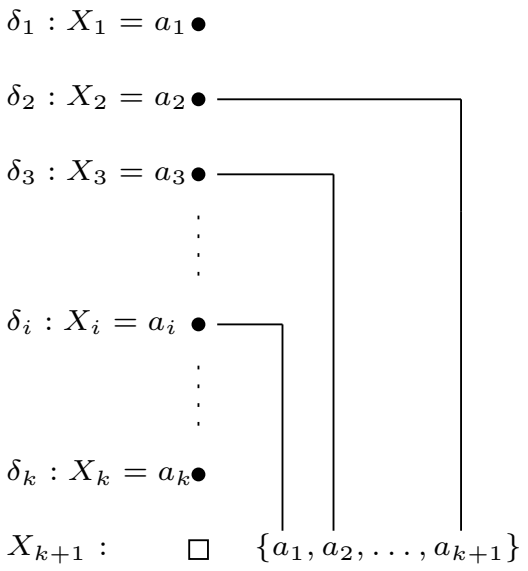


FIG. 2.6 – Identification d'un retour-arrière sûr

X. Plus précisément, on identifie pour chaque valeur que l'on tente d'assigner à la variable X_j, le plus petit préfixe de la séquence de décisions courante incompatible avec cette valeur. La dernière décision du préfixe que l'on vient d'identifier est une décision coupable (au sens de Gaschnig) potentielle. Si celle-ci est plus récente (c'est-à-dire qu'elle a été prise plus récemment au cours de la recherche) que la décision pointée précédemment par *latest*, le pointeur est mis à jour.

Algorithme 8 : *GBackjumping*

Entrées : $P = (\mathscr{X}, \mathscr{C})$: CN, *currentLevel* : Entier
Sorties : *consistent* : Booléen, *backjumping* : Booléen, *level* : Entier

```
1  si isConsistent(P) ≠ NO alors retourner (faux, faux, computeLevel(P)) ;
2  si ∀X ∈ 𝒳, |dom(X)| = 1 alors retourner (vrai, faux, 0) ;
3  choisir une variable X avec |dom(X)| > 1 ;
4  latest = 0 ;
5  pour chaque a ∈ dom(X) faire
6  |  (consistent, backjumping, level) = GBackjumping(P|X=a, currentLevel + 1) ;
7  |  si consistent alors retourner (vrai, faux, 0) ;
8  |  si backjumping alors
9  |  |  si level ≠ currentLevel alors
10 |  |  |  retourner (faux, vrai, level) ;
11 |  |  sinon
12 |  |  |  latest=currentLevel − 1 ;
13 |  |  fin
14 |  sinon
15 |  |  si latest < level alors latest=level ;
16 |  fin
17 fin
18 retourner (faux, vrai, latest) ;
```

L'algorithme 8 présente l'intégration de la technique de retours-arrière intelligents "à la Gaschnig" au sein d'un algorithme de recherche avec retours-arrière utilisant un schéma de branchement n-aire. A chaque appel, l'algorithme récursif sélectionne une variable non encore instantiée (ligne 3) et tente d'assigner une à une toutes les valeurs du domaine de cette variable (ligne 6). En cas de succès (c'est-à-dire quand *consistent* vaut *vrai*) l'algorithme se poursuit. En cas d'échec, soit on effectue un retour-arrière intelligent (ligne 8 à 14) jusqu'au niveau *level* (*backjumping* vaut *vrai*), soit l'assignation précédente a provoqué un échec direct et le pointeur *latest* indiquant la position de la décision incompatible est éventuellement mise à jour (ligne 15). Si toutes les valeurs d'une variable sont incompatibles (ligne 18) le retour-arrière intelligent est déclenché en retournant la valeur *vrai* au paramètre *backjumping*, ainsi que la position de la décision coupable sur laquelle on effectue le saut au travers de la variable *latest*. Un retour-arrière chronologique est assuré dès lors que l'on se trouve à un nœud interne de l'arbre de recherche (ligne 12).

Lorsqu'un échec direct est rencontré après une assignation, l'algorithme 9 détermine la décision coupable (de la branche courante). L'algorithme renvoie la position de cette décision dans la séquence de décisions menant à l'échec (i.e le niveau ou cette décision a été prise dans la branche courante). Pour cela, on teste successivement la compatibilité de la dernière assignation (ayant provoqué l'échec) avec les différents préfixes correspondant à la branche courante.

Algorithme 9 : *computeLevel*

Entrées : $P = (\mathscr{X}, \mathscr{C})$: CN
Sorties : *level* : Entier

```
1 Soit Σ = ⟨δ_1, ..., δ_i⟩ la séquence de décisions menant à un échec de la racine à la feuille de
  l'arbre de recherche ;
2 level = 0 ;
3 pour k = 1 à i − 1 faire
4 |   si P|_{δ_1,...,δ_k}∪{δ_i} = ⊥ alors retourner level ;
5 |   level = level + 1 ;
6 fin
7 retourner level ;
```

2.3.3 Graph-based backjumping

Lorsque l'algorithme de retours-arrière intelligents à la Gaschnig rencontre une impasse, il effectue un saut dans l'arbre de recherche sur une décision δ_i (identifiée comme coupable) en identifiant l'ensemble-conflit correspondant à cette impasse. Cependant si la variable X_i (associée à la décision δ_i) ne possède plus dans son domaine de valeurs compatibles à assigner, on rencontre alors une *impasse interne* (*internal dead-end*). Cette variable est appelées une *variable impasse interne*. L'algorithme de retours-arrière intelligents à la Gaschnig ne peut effectuer des sauts qu'à partir des feuilles de l'arbre de recherche. Plus précisément, lorsqu'une impasse interne est rencontrée (par exemple après un saut), l'algorithme se comporte comme un algorithme de recherche avec retour-arrière chronologique et remet en cause uniquement la décision précédente, même si des sauts plus important dans l'arbre de recherche sont éventuellement possibles.

Présentation

La technique de retours-arrière intelligents basé sur l'analyse du graphe de contraintes (graph-based backjumping) utilise la structure de celui-ci (représentant un problème donné) afin d'approximer les ensembles-conflits et la longueur du saut maximale autorisé [Freuder, 1982, Dechter et Pearl, 198
Cette approche combine les ensembles-conflits de toutes les impasses pertinentes (aussi bien celles survenues aux feuilles de l'arbre de recherche qu'à des nœuds internes) pour identifier la plus récente variable coupable au sens de la technique basée sur le graphe. L'origine d'un saut peut aussi bien être une feuille de l'arbre de recherche qu'un nœud interne, ce qui permet d'effectuer éventuellement des sauts "en cascade".

Si l'instanciation partielle courante ne peut être étendue à une prochaine variable X (ce qui provoque un échec), l'algorithme basé sur le graphe effectue un retour-arrière intelligent sur la plus récente variable Y reliée à X dans le graphe de contraintes. Si à son tour, la variable Y ne possède plus dans son domaine de valeurs compatibles avec l'instanciation partielle courante (i.e. si la décision impliquant Y est une impasse interne), l'algorithme effectue une nouvelle fois un saut sur la plus récente variable Z connectée soit à X, soit à Y dans le graphe de contraintes. La remontée dans l'arbre de recherche se poursuit tant que des impasses internes sont découvertes.

Nous introduisons maintenant les définitions qui seront utilisées par la suite pour identifier la décision coupable sur laquelle on peut effectuer un saut sûr.

Définition 22 (ancêtre d'une variable)**.** *Etant donné un graphe de contraintes associé à un problème P et une séquence de décisions $\Sigma = \langle \delta_1, \ldots, \delta_i, \ldots, \delta_k \rangle$, l'ensemble des ancêtres de*

la variable X_i (associée à la décision δ_i), noté $anc(X_i)$ correspond à l'ensemble des variables apparaissant dans le préfixe $\Sigma' = \langle \delta_1, \ldots, \delta_{i-1} \rangle$ et connectées à X_i dans le graphe de contraintes.

Le parent de X_i, notée $p(X_i)$, représente la variable de $anc(X_i)$ la plus proche de X_i dans Σ.

Etant donné une séquence de décisions $\Sigma = \langle \delta_1, \ldots, \delta_i, \ldots, \delta_k \rangle$, l'ensemble des variables en échec et des variables impasses internes rencontrées sur une branche de l'arbre de recherche jusqu'à une décision δ_i (impliquant une variable X_i) est noté $impasses(X_i)$. Cet ensemble est construit récursivement à partir de la feuille δ_k de l'arbre de recherche à chaque fois qu'une nouvelle impasse est rencontrée.

On peut étendre alors la notion d'ancêtres à celle d'ancêtres induits.

Définition 23 (ancêtre induit)**.** *Soit $\Sigma = \langle X_1 = a_1, \ldots, X_i = a_i, \ldots, X_k = a_k \rangle$ une séquence de décision et $Y = impasses(X_i)$. L'ensemble des ancêtres induits de X_i par rapport à Y, noté $I_{X_i}(Y)$, est l'union des ancêtres de toutes les variables de Y, restreint à celles précédent X_i dans Σ. Formellement, $I_i(Y) = \cup_{y \in Y} anc(y) \cap \{X_1, \ldots, X_{i-1}\}$.*

Le parent induit de X_i par rapport à Y, noté $P_{X_i}(Y)$ est la variable de $I_{X_i}(Y)$ la plus proche de X_i dans Σ. Cette variable est appelée : variable coupable au sens de la technique basée sur le graphe de contraintes.

Si X_j est le parent induit de X_i par rapport à Y, alors la décision impliquant X_j dans Σ représente alors un saut sûr.

Proposition 2. *Soit la variable X_i telle que $Y = impasses(X_i)$ et $P_{X_i}(Y) = X_j$ la décision coupable identifiée au sens de la technique basée sur le graphe de contraintes. Si δ_j est la décision impliquant X_j alors le saut effectué sur δ_j est sûr.*

La figure 2.7 illustre sur un exemple l'identification d'un saut sûr sur lequel il est possible d'effectuer un retour-arrière intelligent. La séquence de décision $\Sigma = \{\delta_1, \ldots, \delta_6\}$ mène à un échec, les valeurs du domaine de la variable X_7 étant incompatible avec l'instanciation partielle courante. La variable X_7 est reliée aux (précédentes) variables X_5 et X_3 de la séquence Σ par des contraintes, i.e. il existe une contrainte impliquant X_7 et X_5 et une contrainte impliquant X_7 et X_3. Ceci se traduit sur la figure par un lien entre la variable X_7 et les variables X_5 et X_3. Lorsque l'impasse est détectée on en déduit donc : $anc(X_7) = \{X_5, X_3\}$ et $Y = impasses(X_7) = \{X_7\}$.

A partir de ces informations, l'ensemble des ancêtres induits de X_7 est égal à $I_{X_7}(Y) = anc(X_7) \cap \{X_1, \ldots, X_6\} = \{X_3, X_5\}$. On peut donc identifier la variable coupable $P_{X_7}(Y) = X_5$ et effectuer un retour-arrière intelligent sur la décision correspondante δ_5.

Imaginons maintenant que la variable X_5 n'ait plus de valeurs compatibles dans son domaine et qu'une impasse interne soit alors rencontrée. On calcule donc à nouveau :

- $anc(X_5) = \{X_1, X_2\}$;
- $Y' = impasses(X_5) = Y \cup \{X_5\} = \{X_5, X_7\}$;
- $I_{X_5}(Y') = (anc(X_7) \cup anc(X_5)) \cap \{X_1, \ldots, X_4\} = \{X_3, X_2, X_1\}$.

On peut donc identifier la variable coupable $P_{X_5}(Y') = X_3$ et effectuer un retour-arrière intelligent sur la décision correspondante δ_3. Ce processus se répète à chaque fois qu'une impasse interne est rencontrée.

Bien que l'identification de la décision coupable permette de réduire la taille de l'ensemble-conflit initialement détecté, cette définition ne permet d'obtenir qu'une approximation de l'ensemble-conflit minimal. En effet, contrairement à la technique de Gaschnig basée sur l'analyse des domaines des variables, la définition de variable coupable introduite pour cette méthode ne prend en compte que le graphe de contraintes. On peut tout à fait imaginer alors le scénario suivant. Etant donné un ensemble-conflit pour une variable X_{i+1}, la plus récente décision coupable δ_j

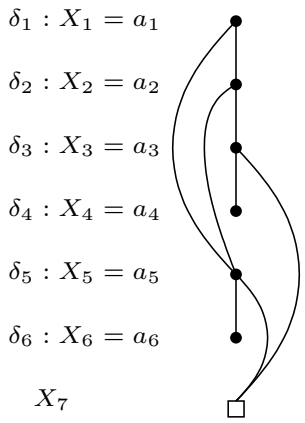


FIG. 2.7 – Identification d'un retour-arrière sûr

(impliquant la variable X_j) est identifiée et l'on peut effectuer un retour-arrière intelligent sur cette décision. Or si toutes les valeurs de X_{i+1} sont compatibles avec la variable X_j (la valeur assignée à X_j est un support universel pour X_{i+1}), il existe un conflit avec une décision prise auparavant dans l'arbre de recherche. L'ensemble-conflit identifié n'est donc pas minimal et on peut trouver un sous-ensemble suffisant pour expliquer l'échec. Il est alors possible d'effectuer un saut plus long dans l'arbre de recherche.

Algorithme

Algorithme 10 : *GbBackjumping*

Entrées : $P = (\mathscr{X}, \mathscr{C})$: CN, *varAssignees* : Ensemble de variables
Sorties : *consistent* : Booléen, *backjumping* : Booléen, VtB : Variable, *ancInduits* : Ensemble de variables

```
1  si isConsistent(P) ≠ NO alors retourner (faux, faux, ∅, ∅) ;
2  si ∀X ∈ 𝒳, |dom(X)| = 1 alors retourner (vrai, faux, ∅, ∅) ;
3  choisir une variable X avec |dom(X)| > 1 ;
4  localAncInduits = voisins(X) ∩ varAssignees ;
5  pour chaque a ∈ dom(X) faire
6  |  (consistent, backjumping, VtB, ancInduits) = GBackjumping(P|X=a, Y ∪ {X}) ;
7  |  si consistent alors retourner (vrai, faux, ∅, ∅) ;
8  |  si backjumping alors
9  |  |  si VtB ≠ X alors
10 |  |  |  retourner (faux, vrai, VtB, ancInduits) ;
11 |  |  sinon
12 |  |  |  localAncInduits = localAncInduits ∪ (ancInduits ∩ varAssignees) ;
13 |  |  fin
14 |  fin
15 fin
16 retourner (faux, vrai, parentInduit(localAncInduits) , localAncInduits) ;
```

L'algorithme 10 présente l'intégration de la technique de retours-arrière intelligents "basée sur

le graphe de contraines" au sein d'un algorithme de recherche avec retours-arrière utilisant un schéma de branchement n-aire. A chaque appel, P représente le réseau de contraintes qu'il faut résoudre (et correspondant à un nœud de l'arbre de recherche) et $varAssignees$ l'ensemble des variables déjà assignées (initialement vide). Pour toute variable X du problème, les voisins de X sont calculés initialement et accessibles via la structure de données $voisins(X)$. Cette structure fournit pour chaque variable X, l'ensemble des variables connectées à X dans le graphe de contraintes. Ces informations sont statiques et n'évolueront pas au cours de la recherche.

Pour chaque variable X sélectionnée (i.e. à chaque nœud de l'arbre de recherche), l'algorithme calcule les ancêtres de cette variable par rapport à l'ensemble des décisions prises sur la branche courante (ligne 4). Si aucune valeur ne peut être assignée à cette variable (ligne 16), une impasse (éventuellement interne, si il n'existe plus de valeurs compatibles) vient d'être rencontrée et l'algorithme récursif effectue alors un retour-arrière intelligent. Pour cela, on retourne l'ensemble des ancêtres induits (par le paramètre $localAncInduits$), composé des ancêtres locaux précédemment calculés et éventuellement d'ancêtres induits obtenus lors des retours-arrière. Les ancêtres induits permettent de calculer le parent induit (en appelant $parentInduit$) sur lequel il faut effectuer un saut (VtB). La fonction $parentInduit$ retourne la variable de l'ensemble $localAncInduits$ correspondant à la décision prise la plus récemment. A chaque saut (ligne 12), l'ensemble des ancêtres induits est mis à jour (en fonction de ceux retournés lors du retour-arrière intelligent).

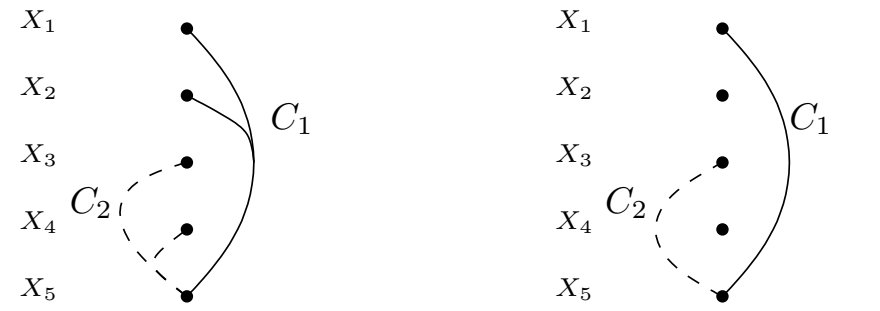
2.3.4 Conflict-directed backjumping

Cette technique de retours-arrière dirigée par les conflits [Prosser, 1993] (également connue sous le nom de CBJ) combinent élégamment les deux approches présentées précédemment (Gaschnig et Graph-based backjumping). Elle peut être intégrée au sein d'un algorithme maintenant la consistance d'arc (M(G)AC) [Prosser, 1995] de façon assez efficace. Avec cette approche, on détecte les décisions conflictuelles avec la variable courante ainsi que les impasses internes (obtenues après avoir effectué des retours-arrière), permettant ainsi des sauts plus importants dans l'arbre de recherche.

Présentation

La méthode proposée est assez similaire (d'un point de vue algorithmique) à celle employée dans l'algorithme basé sur le graphe de contraintes. Cependant les informations (nécessaires au saut) ne sont plus collectées uniquement à partir du graphe de contraintes mais sont recueillies également au cours de la recherche. Pour chaque variable, l'algorithme maintient une liste de variables (déjà instantiées), appelé ensemble des "sauts induits" (de façon analogue à l'ensemble des ancêtres induits maintenu par l'algorithme basé sur le graphe de contraintes). A chaque retour-arrière, On identifie l'ensemble des variables (dont les valeurs prises dans l'instanciation courante suffisent à expliquer le retour-arrière) dites "en conflit", calculé à partir du graphe de contraintes et des domaines courants des variables. Plus précisément, lorsqu'une impasse (éventuellement interne) est rencontrée, l'ensemble des sauts éventuels est calculé à partir des variables participant aux ensembles conflits minimaux (notés *var-emc* et définis ci-dessous) de toutes les impasses rencontrées depuis la dernière feuille de l'arbre de recherche.

Rappelons tout d'abord qu'étant donné un réseau de contraintes $P = (\mathscr{X}, \mathscr{C})$, en considérant un ordre total sur $\mathscr{X}$, toute contrainte porte sur un sous-ensemble ordonné de variables de $\mathscr{X}$. Il est alors assez naturel de définir un ordre total sur les contraintes en considérant un ordre particulier des variables. On définit alors un ordre lexicographique sur les contraintes. Une contrainte C_i est inférieure à C_k, noté $C_i < C_k$, si la dernière variable apparaissant dans



$scp(C_1) = \{X_1, X_2, X_5\}$
$scp(C_2) = \{X_3, X_4, X_5\}$

(a)

$scp(C_1) = \{X_1, X_5\}$
$scp(C_2) = \{X_3, X_5\}$

(b)

FIG. 2.8 – Relation d'ordre entre les contraintes

$scp(C_i) - scp(C_k)$ précède la dernière variable apparaissant dans $scp(C_k) - scp(C_i)$. La figure 2.8 illustre sur un exemple la relation d'ordre définie précédemment dans le cas de contraintes non-binaires (a) et binaires (b). Etant donné l'ordre des variables $(X_1, X_2, X_3, X_4, X_5)$, La contrainte C_1 est inférieure à C_2 sur l'illustration (a) puisque $X_2 < X_4$. De façon analogue, sur l'illustration (b) la contrainte binaire C_1 est inférieure à C_2.

Si un conflit apparaît après avoir assigné une variable (par exemple sur la figure 2.8 (a), l'assignation de la variable X_5 viole les contraintes C_1 et C_2), la relation d'ordre définie précédemment permet d'identifier la contrainte violée permettant un saut de longueur maximale. Sur la figure 2.8 (a), on remarque qu'il est inutile de remettre en cause les variables $\{X_3, X_4\}$ de C_2. En effet, les contraintes C_1 et C_2 sont violées et $C_1 < C_2$. Il faut donc remettre en cause directement les variables $\{X_1, X_2\}$ apparaissant dans la portée de C_1.

Soit une instanciation partielle t menant à une impasse sur une variable X_{i+1}. L'ensemble-conflit minimal le plus récent de t, noté $var\text{-}emc(X_{i+1})$ (pour Earliest Minimal Conflict-set) est composé des variables apparaissant dans la portée de contraintes C_i violant l'instanciation partielle $t \cup \{X_{i+1} = b\}$ ou $b \in dom(X_{i+1})$ et tel que $\nexists C_k < C_i \mid C_k \neq C_i$ et C_k est également violée par $t \cup \{X_{i+1} = b\}$.

Définition 24 ($var\text{-}emc$). *Soit $P = (\mathscr{X}, \mathscr{C})$ un réseau de contraintes et t une instanciation partielle menant à une impasse sur X_{i+1}. L'ensemble-conflit minimal le plus récent de t, noté $var\text{-}emc(X_{i+1})$ représente $\{X \in scp(C_i)$ où $X \neq X_{i+1}$ et C_i est une contrainte violée par $P_{t \cup \{X_{i+1}=b\}}$ telle que $b \in dom(X_{i+1})$ et $\nexists C_k < C_i \mid C_k \neq C_i$ et C_k violée par $P_{t \cup \{X_{i+1}=b\}}\}$.*

La figure 2.9 illustre la définition 24 sur un réseau de contraintes binaires, en considérant un ordre lexicographique des variables. L'instanciation $t = \{X_1 = a_1, X_2 = a_2, X_3 = a_3, X_4 = a_4, X_5 = a_5, X_6 = a_6\}$ mène à une impasse sur la variable X_7. Pour chaque valeur du domaine de X_7 on peut déterminer les contraintes violées.

- L'instanciation $t \cup \{X_7 = 0\}$ viole la contrainte C_1 ;
- L'instanciation $t \cup \{X_7 = 1\}$ viole les contraintes C_2 et C_4 et $C_2 < C_4$;
- L'instanciation $t \cup \{X_7 = 2\}$ viole les contraintes C_2 et C_5 et $C_2 < C_5$.

On en déduit donc l'ensemble-conflit minimal suivant : $var\text{-}emc(X_7) = \{X \in scp(C_i) \mid C_i \in \{C_1, C2\} \wedge X \neq X_7\} = \{X1, X2\}$.

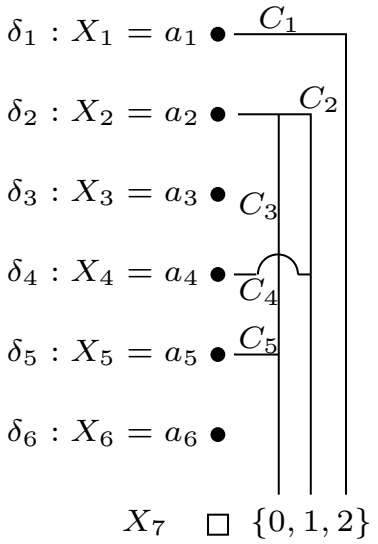


FIG. 2.9 – Identification de l'ensemble-conflit minimal le plus récent

Soit X_{i+1}, une variable dont les valeurs du domaine sont incompatibles avec l'instanciation partielle courante $t = \{X_1 = a_1, \ldots, X_i = a_i\}$ (i.e. t est un ensemble-conflit). L'ensemble des sauts, noté J_{i+1}, d'une variable X_{i+1} est son *var-emc*(X_{i+1}). L'ensemble des sauts d'une impasse interne X_i (détectée après un retour-arrière provoqué par l'impasse de la variable X_{i+1}) inclus l'ensemble des *var-emc* correspondant à toutes les impasses rencontrées depuis la feuille (i.e X_{i+1}) de l'arbre de recherche. Plus formellement $J_i = \bigcup\{\textit{var-emc}(X_k) \mid \forall$ impasse X_k détectée depuis la feuille $X_{i+1}\}$.

La notion d'explications extraites à partir d'une impasse rencontrée au cours de la recherche est similaire à celle décrite dans l'approche basée sur le graphe de contraintes, en remplaçant $anc(X_i)$ par *var-emc*(X_i). De façon analogue, les ensembles J_i calculés aux impasses internes sont à rapprocher des ancêtres induits $(I_{X_i}(Y))$ dans l'approche basée sur le graphe de contraintes. Plus précisément, *var-emc*(X_i) est un sous-ensemble des variables de $anc(X_i)$. Les variables ne faisant pas partie de *var-emc*(X_i) (mais de $anc(X_i)$) sont des variables apparaissant soit dans la portée de contraintes qui ne provoquent pas d'échec, soit dans la portée de contraintes violées mais "subsumées" par une autre contrainte d'ordre inférieure.

Proposition 3. *Soit Σ une séquence de décision menant à une impasse, t l'instanciation partielle correspondante et J_i son ensemble de sauts associé. La décision apparaissant la plus à droite dans Σ et dans J_i représente un saut sûr et la variable associée à cette décision est la variable coupable au sens de CBJ.*

Poursuivons l'exemple 2.9. A une feuille de l'arbre de recherche, l'ensemble des sauts associé à la variable X_7 correspond à son ensemble-conflit minimal précédemment identifié. On a donc $J_7 = \textit{var-emc}(X_7) = \{X_1, X_2\}$. Etant donné la séquence de décision $\Sigma = \langle\delta_1, \delta_2, \delta_3, \delta_4, \delta_5, \delta_6\rangle$ menant à cet échec, la décision δ_2 (associée à la variable X_2) représente alors un saut sûr.

Algorithme

L'algorithme 11 illustre l'intégration d'un mécanisme de retours-arrière intelligents dirigés par les conflits au sein d'un algorithme de recherche utilisant un schéma de branchement non-binaire.

Algorithme 11 : *CBJBackjumping*

Entrées : $P = (\mathscr{X}, \mathscr{C})$: CN, Y : Ensemble de variables
Sorties : *consistent* : Booléen, *backjumping* : Booléen, VtB : Variable, J : Ensemble de variables

```
1  si isConsistent(P) ≠ NO alors retourner (faux, faux, ∅,
   computeConflictsVariables(P, Y)) ;
2  si ∀X ∈ 𝒳, |dom(X)| = 1 alors retourner (vrai, faux, ∅, ∅) ;
3  choisir une variable X avec |dom(X)| > 1 ;
4  J_X = ∅ ;
5  pour chaque a ∈ dom(X) faire
6  |  (consistent, backjumping, VtB, J) = CBJBackjumping(P|_{X=a}, Y ∪ {X}) ;
7  |  si consistent alors retourner (vrai, faux, ∅, ∅) ;
8  |  si backjumping alors
9  |  |  si VtB ≠ X alors
10 |  |  |  retourner (faux, vrai, VtB, J) ;
11 |  |  sinon
12 |  |  |  J_x = J_x ∪ J ;
13 |  |  fin
14 |  sinon
15 |  |  J_x = J_x ∪ J ;
16 |  fin
17 fin
18 Soit Σ la séquence de décisions menant au nœud courant ;
19 retourner (faux, vrai, selectJump(J_X ∩ Y, Σ) , J_X ∩ Y) ;
```

Le calcul des sauts induits n'est pas fait lorsque une impasse est détectée mais itérativement lorsque (1) une assignation vient de provoquer un échec et (2) lorsqu'un retour-arrière vient d'être effectué.

Pour chaque variable X sélectionnée (i.e. à chaque nœud de l'arbre de recherche), J_X mémorise l'ensemble des sauts induits pour cette variable. Initialement $J_X = \emptyset$ (ligne 4). J_X est mis à jour suite à l'échec d'une assignation, i.e lorsque l'algorithme récursif renvoie $consistent = faux$ et $backjumping = faux$ (ligne 15) ou alors suite à un retour-arrière intelligent (ligne 12). A chaque échec d'une assignation, l'algorithme identifie la contrainte d'ordre inférieure violée et retourne les variables impliquées dans la portée de cette contrainte (fonction $computeConflicts Variables$, ligne 1).

Lorsque l'algorithme rencontre une impasse, c'est-à-dire lorsque toutes les valeurs du domaine courant de la variable X ont été prouvées incompatibles, le saut est déclenché (ligne 19). L'ensemble des sauts associés à cette impasse (collectés au travers de la variable J_x) est retourné ainsi que la variable identifiée comme sûre (noté VtB pour Variable to Backjump), sur laquelle on effectue le retour-arrière. Notons que la variable VtB est identifiée grâce à la fonction $selectJump$ en utilisant les sauts induits ($J_x \cap Y$) et la séquence de décisions menant au nœud courant.

2.3.5 Les retours-arrière dynamiques (DBT)

Présentation

Jusqu'à présent, les techniques de retours-arrière intelligents présentées ici identifient une variable coupable et effectuent un saut sur celle-ci en "oubliant" l'exploration de l'espace de recherche effectuèe depuis cette affectation. Ces approches ignorent donc une très grande quantitè d'information qui pourtant aurait pu être utile, ce qui conduit éventuellement à l'apparition d'un phénomène de thrashing.

Les retours-arrière dynamiques (DBT) [Ginsberg, 1993] tout comme CBJ identifient la décision la plus récente impliquée dans le dernier conflit. Cependant comme dans une méthode de réparation, DBT ne supprime que l'information dépendante de cette décision (précédemment identifiée) grâce aux explications enregistrées. L'information utile est conservée et il n'y a pas de retours-arrière à proprement parler (tout comme dans une méthode de réparation). Seules les affectations responsables de la contradiction sont défaites et il n'y a pas de retours-arrière (intelligents) à proprement parlé. De plus une approche comme CBJ associe les explications aux variables alors que DBT associe une explication pour chaque valeur (i.e. au niveau valeur). Remarquons cependant qu'une variante de CBJ est envisageable en associant les explications aux valeurs.

Cette approche associe un nogood standard à chaque valeur supprimée du domaine des variables. Plus précisément, pour une valeur supprimée donnée (par exemple $X \neq a$), le nogood associé correspond à une explication (par exemple $expl(X \neq a)$) du retrait de cette valeur. Le nogood est composé alors des décisions positives de la première contrainte non satisfaite rencontrée. Lorsqu'un domaine vide apparaît, on calcule un nouveau nogood résultant de l'union des explications relatives à la suppression de chaque valeur du domaine, c.f. l'algorithme 16 décrit ci-après. Autrement dit, un nouveau nogood est calculé en faisant l'union des nogoods associés à chaque valeur supprimée ($\bigcup_{a \in dom(X)} expl(X \neq a)$). Lorsqu'un conflit apparaît suite à une assignation, on effectue une réparation en réfutant la dernière décision impliquée dans le nogood associé à l'assignation conflictuelle (on fait de même lorsqu'une impasse apparait). Les explications impliquant cette décision sont alors mises à jour (i.e. supprimées).

Algorithme

L'algorithme 12 assigne itérativement les valeurs aux variables tant qu'une solution n'a pas été trouvée. $fut(P)$ représente l'ensemble des variables dont le domaine n'est pas réduit à un singleton, c'est-à-dire l'ensemble des variables qui n'ont pas encore été assignées. A chaque tour de la boucle principale, on assigne une valeur à une variable. Si cette assignation n'est pas consistante, la fonction *isConsistent* extrait un nogood (composé des décisions impliquant la première contrainte du réseau qui n'est pas satisfaite).

La fonction *handleContradiction* (algorithme 14) effectue une réparation de la contradiction précédemment rencontrée. Pour cela, la dernière décision $X = a$ présente dans le nogood est réfutée (ligne 3) et les décisions impliquées dans celui-ci constituent alors une explication pour cette réfutation. Toutes les explications impliquant $X = a$ sont alors éliminées (algorithme 15), ce qui revient à restaurer les valeurs des domaines qui n'ont plus d'explication valide. Lorsqu'un domaine vide (i.e. une impasse) est rencontré (ligne 5), un nouveau nogood est calculé en faisant l'union des explications des retraits de chaque valeur du domaine (algorithme 16). La fonction *handleContradiction* retourne $faux$ lorsque la réparation a été effectuée (la variable nogood vaut dans ce cas NO) ou $vrai$ lorsqu'un nogood vide a été rencontré. Si un nogood vide est rencontré, aucune solution ne peut être trouvée et la recherche est terminée.

Algorithme 12 : search

Entrées : $P = (\mathscr{X}, \mathscr{C})$: CN
Sorties : Booléen

```
1  expl(X ≠ a) ← NO, ∀X ∈ 𝒳 ∧ ∀a ∈ dom(X) ;
2  finished ← faux ;
3  tant que ¬finished faire
4  |  Choisir une paire (X, a) telle que X ∈ fut(P) ∧ a ∈ dom(X) ;
5  |  doAssignment(X, a) ;
6  |  nogood ← isConsistent(P) ;
7  |  si nogood ≠ NO alors
8  |  |  finished ← ¬handleContradiction(nogood) ;
9  |  sinon si fut(P) = ∅ alors
10 |  |  finished ← vrai     // Solution ;
11 |  fin
12 fin
```

Algorithme 13 : doAssignment

Entrées : X : Variable, a : valeur

```
1  pour chaque valeur b ∈ dom(X) telle que b ≠ a faire
2  |  expl(X ≠ b) ← {(X, a)} ;
3  fin
```

Dans [Jussien *et al.*, 2000], les auteurs proposent une approche pour intégrer la propagation par contraintes et plus précisément le maintien de la consistance d'arc (MAC) dans DBT. L'intégration d'un mécanisme de propagation au sein de cet algorithme n'est pas trivial. En effet lors du processus de propagation, des valeurs sont supprimées et il est alors nécessaire de conserver des explications pour les suppressions effectuées au cours du filtrage. De même lorsqu'on

Algorithme 14 : handleContradiction

Entrées : *nogood* : Nogood
Sorties : Booléen

tant que $nogood \neq NO \wedge nogood \neq \emptyset$ **faire**
  $(X, a) \leftarrow$ dernière assignation effectuée et présente dans *nogood* ;
  $undoAssignment(X, a)$;
  $expl(X \neq a) \leftarrow nogood \backslash (X, a)$ // *a est supprimée de dom(X)* ;
  si $dom(X) = \emptyset$ **alors**
    $nogood \leftarrow handleEmptyDomain(X)$;
  sinon
    $nogood \leftarrow NO$;
  fin
fin
retourner $nogood \neq \emptyset$;

Algorithme 15 : undoAssignment

Entrées : X : Variable, a : valeur

pour chaque *paire* (Y, b) *avec* $Y \in \mathscr{X} \wedge b \in dom^{P_0}(Y)$ *telle que* $(X, a) \in expl(Y \neq b)$ **faire**
  si $Y \in fut(\mathscr{X})$ **alors**
    $expl(Y \neq b) \leftarrow NO$;
  sinon
    $expl(Y \neq b) \leftarrow \{(Y, c)\}$ où c est la valeur assignée à Y
  fin
fin

Algorithme 16 : handleEmptyDomain

Entrées : X : Variable
Sorties : Nogood

$nogood \leftarrow \emptyset$;
pour chaque *valeur* $a \in dom^{P_0}(X)$ **faire** $nogood \leftarrow nogood \cup expl(X \neq a)$;
retourner *nogood* ;

désaffecte une variable, il ne suffit pas de restaurer les valeurs qui n'ont plus d'explication valide puisque plusieurs raisons peuvent justifier le retrait d'une valeur et qu'ici une seule raison n'est enregistrée.

2.3.6 Comparaison des algorithmes

Tous les algorithmes présentés précédemment ne possèdent pas la même capacité d'élagage. Autrement dit, l'espace de recherche parcouru par ces différents algorithmes varie en fonction de la capacité de ceux-ci à détecter (et éviter) des branches de l'arbre de recherche dans lesquelles n'apparaissent aucune solution. De plus, ils sont généralement intégrés à un algorithme de recherche, effectuant une certaine forme de filtrage à chaque nœud de l'arbre de recherche (consistance d'arc, forward-checking...). Leur capacité d'élagage est alors accrue.

La figure 2.10 représentent différents algorithmes de recherche dans lesquels on greffe éventuellement un mécanisme de retours-arrière intelligents ou une technique de filtrage spécifique. Les algorithmes sont comparés en fonction de l'espace de recherche exploré par ceux-ci. Pour que la comparaison soit valable, l'heuristique de choix utilisée doit être statique, c'est-à-dire qu'elle conserve le même ordre de priorité tout au long de la recherche. Plus précisément, un algorithme A est supérieur à un algorithme B (c'est-à-dire qu'il existe une flèche allant de B vers A) si l'espace de recherche exploré par l'algorithme A est inclus dans celui exploré par l'algorithme B. S'il n'existe pas de flèche entre deux algorithmes, cela signifie qu'ils sont incomparables. Les espaces de recherche parcourus par ces deux algorithmes sont simplement différents.

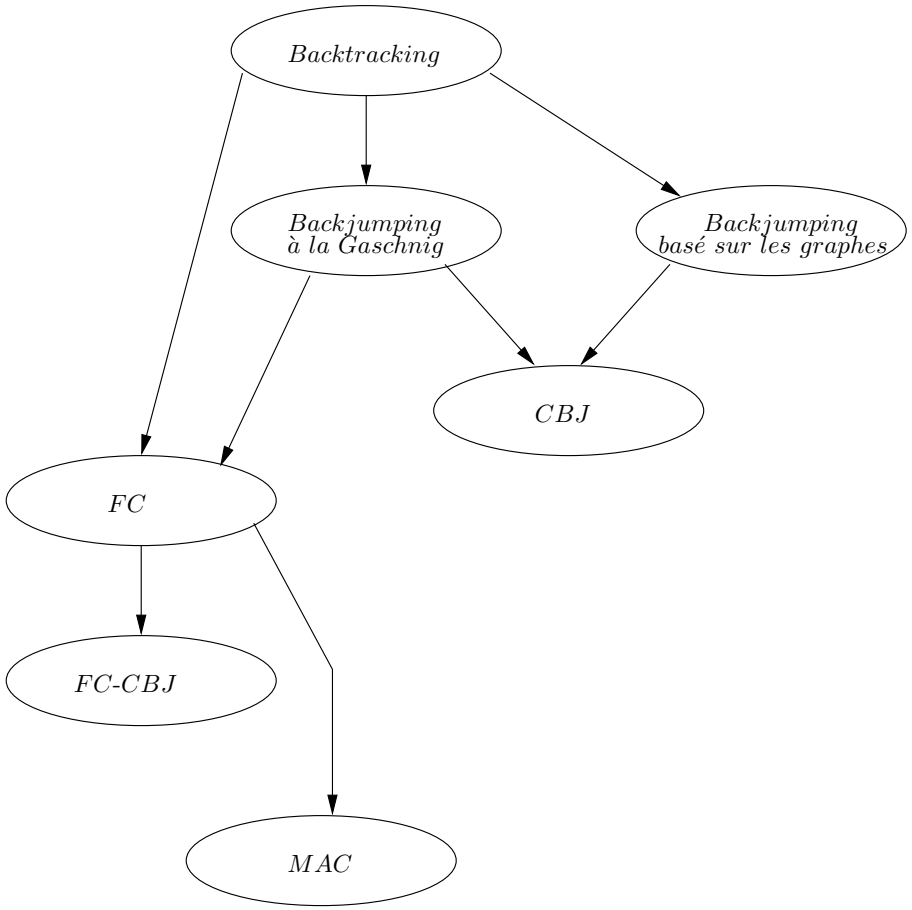


FIG. 2.10 – Comparaison des algorithmes de retours-arrière (intelligents)

Deuxième partie

Contributions

1

Raisonner à partir des derniers conflits

Le raisonnement à partir des derniers conflits [Lecoutre *et al.*, 2006] offre la possibilité de remonter (indirectement) sur une variable identifiée comme coupable de la dernière impasse rencontrée. Pour atteindre celle-ci, la variable ayant provoquée le conflit (i.e. la variable impliquée dans la dernière décision prise) demeure prioritairement la variable à sélectionner tant que les assignations successives l'impliquant rendent le réseau arc inconsistant. Cela correspond alors à vérifier la singleton consistance de cette variable de la feuille vers la racine de l'arbre de recherche jusqu'à ce qu'une valeur puisse lui être assignée sans provoquer directement un échec. En d'autres termes, l'heuristique de choix de variable est violée (i.e. le raisonnement à partir des derniers conflits se substitue à l'heuristique de choix de variable), jusqu'à ce que l'on effectue un retour-arrière sur la variable coupable et qu'une valeur puisse être assignée à la variable impliquée dans le dernier conflit. Pour résumer, l'approche décrite ici, a pour objectif de guider la recherche de manière à détecter dynamiquement une raison du dernier conflit rencontré. Il est important de remarquer que, contrairement aux techniques sophistiquées de retours-arrière intelligents, cette approche peut être greffée très facilement sur un algorithme de recherche sans structures de données supplémentaires à gérer. Bien que le raisonnement à partir des derniers conflits permet de guider la recherche et donc peut être considéré comme une méthode prospective (i.e. une méthode qui dirige la recherche), celui-ci permet de bénéficier d'un "effet backjumping" sans pour autant analyser les raisons précises des conflits rencontrés. La généralisation de ce raisonnement aux n derniers conflits permet d'envisager la détection de nogoods. Brièvement, ces nogoods correspondent à un ensemble de n variables ne pouvant être "traversées", c'est-à-dire successivement assignées sans mener à un échec (après avoir établi une consistance ϕ). Cela revient à identifier une variable coupable telle qu'au moins une instanciation de ces n variables soit consistante avec l'instanciation partielle courante.

1.1 Raisonner à partir du dernier conflit

Il est possible d'identifier un nogood à partir d'une séquence de décisions menant à un conflit, et d'exploiter ce nogood pendant la recherche. Ce raisonnement sera exploité pour remonter aux origines de l'échec et éliminer ainsi (en partie) le phénomène de thrashing. Lorsqu'une branche d'un arbre de recherche mène à une impasse, les décisions prises sur cette branche forment un nogood Δ. Notons que celui-ci peut être généralisé si des décisions négatives ont été prises sur la branche. Le raisonnement à partir du dernier conflit permet d'identifier un nogood plus petit que Δ mais pas forcément minimal en effectuant des retours-arrière successifs le long de la branche courante. Ces retours-arrière permettent d'identifier la variable de la branche courante la plus

récente responsable de l'échec.

1.1.1 L'identification de nogoods

On considère ici un opérateur d'inférence établissant une consistance ϕ présumée satisfaire les propriétés usuelles (notamment la confluence). Cet opérateur peut être employé à n'importe quelle étape de l'arbre de recherche construit par un ϕ-algorithme de recherche et utilisant un schéma de branchement binaire. Par exemple, MAC correspond à un AC-algorithme de recherche où AC est l'opérateur d'inférence établissant la consistance d'arc.

On rappelle qu'un nogood est un ensemble de décisions qui n'apparaît dans aucune solution d'un problème P donné. Soit P un CN et Δ un ensemble de décisions.

- Δ est un nogood de P ssi $P|_\Delta$ est insatisfaisable ;
- Δ est un ϕ-nogood de P ssi $\phi(P|_\Delta) = \bot$;
- Δ est un ϕ-nogood minimal de P ssi $\nexists \Delta' \subset \Delta$ tel que $\phi(P|_{\Delta'}) = \bot$.

Nous introduisons tout d'abord les concepts de *décision* et *sous-séquence coupable*. La décision coupable d'une séquence de décisions $\Sigma = \langle \delta_1, \ldots, \delta_i \rangle$ par rapport à un ensemble de variables S et une consistance ϕ est la décision δ_j la plus à droite de Σ telle qu'il est impossible d'étendre la séquence $\langle \delta_1, \ldots, \delta_j \rangle$ avec une instanciation de S, sans détecter une inconsistance via ϕ. Plus formellement, ces notions peuvent être définies comme suit :

Définition 25 (pivot et décision coupable)**.** *Soient P un réseau de contraintes, $\Sigma = \langle \delta_1, \ldots, \delta_i \rangle$ une séquence de décisions construite à partir de P telle que $\{\delta_1, \ldots, \delta_i\}$ est un nogood de P, ϕ une consistance et $S = \{X_1, \ldots, X_k\} \subseteq vars(P)$.*

- *un pivot de Σ par rapport à ϕ et S est une décision $\delta_j \in \Sigma$ telle que $\exists a_1 \in dom(X_1), \ldots, \exists a_k \in dom(X_k) \mid \phi(P|_{\{\delta_1,\ldots,\delta_{j-1},\neg\delta_j,X_1=a_1,\ldots,X_k=a_k\}}) \neq \bot$;*
- *la décision coupable de Σ par rapport à ϕ et S est le pivot le plus à droite de Σ par rapport à ϕ et S, s'il existe.*

La sous-séquence coupable de Σ par rapport à ϕ et S est soit la séquence vide $\langle\rangle$ si Σ ne contient pas de décision coupable, soit la séquence $\langle \delta_1, \ldots, \delta_j \rangle$ où δ_j est la décision coupable de Σ par rapport à ϕ et S. S est appelé l'ensemble-test de culpabilité.

Naturellement, on peut montrer que l'ensemble des variables d'une sous-séquence coupable est un nogood.

Proposition 4. *Soit P un réseau de contraintes, $\Sigma = \langle \delta_1, \ldots, \delta_i \rangle$ une séquence de décisions construites à partir de P telle que $\{\delta_1, \ldots, \delta_i\}$ est un nogood de P, ϕ une consistance et $S \subseteq vars(P)$. L'ensemble des décisions contenues dans la sous-séquence coupable de Σ par rapport à ϕ et S est un nogood de P.*

Démonstration. Soit $\langle \delta_1, \ldots, \delta_j \rangle$ la sous-séquence (non vide) coupable de Σ. Démontrons par récurrence que pour tout entier k tel que $j \leq k \leq i$, l'hypothèse suivante, notée H(k), est vraie :

$$\text{H}(k) : \{\delta_1, \ldots, \delta_k\} \text{ est un nogood}$$

Tout d'abord, montrons que H(i) est vraie. Nous savons que $\{\delta_1, \ldots, \delta_i\}$ est un nogood par hypothèse. Ensuite, montrons que, pour $j < k \leq i$, si H(k) est vraie alors H($k-1$) est vraie également. Comme $k > j$ et H(k) est vraie, nous savons que, par hypothèse de récurrence, $\{\delta_1, \ldots, \delta_{k-1}, \delta_k\}$ est un nogood. En outre, δ_k n'est pas un pivot de Σ (puisque $k > j$ et δ_j est la décision coupable de Σ). Grâce à la définition 25, nous savons que $\forall a_1 \in dom(X_1), \ldots, \forall a_k \in dom(X_k)$, $\phi(P|_{\{\delta_1,\ldots,\delta_{k-1},\neg\delta_k,X_1=a_1,\ldots,X_k=a_k\}}) = \bot$. En conséquence, l'ensemble $\{\delta_1, \ldots, \delta_{k-1}, \neg\delta_k\}$ est un nogood. En utilisant le principe de résolution, $\{\delta_1, \ldots, \delta_{k-1}, \delta_k\}$ et $\{\delta_1, \ldots, \delta_{k-1}, \neg\delta_k\}$ étant

des nogoods, nous en déduisons que $\{\delta_1, \ldots, \delta_{k-1}\}$ est également un nogood. Pour une sous-séquence coupable vide, on peut aisément adapter le raisonnement précédent pour en déduire que $\emptyset$ est un nogood. □

Il est important de remarquer que ce nouveau nogood que l'on vient d'identifier peut correspondre au nogood initial. Le cas apparaît lorsque la décision coupable de la séquence $\Sigma = \langle \delta_1, \ldots, \delta_i \rangle$ est δ_i. D'un autre coté, lorsque la sous-séquence coupable de Σ est vide, cela veut dire que P est insatisfaisable.

En pratique, lorsqu'un conflit est rencontré au cours d'une résolution effectuée par un algorithme de recherche avec retours-arrière, cela veut dire qu'un nogood vient d'être identifié : il correspond à l'ensemble des décisions prises le long de la branche menant à cet échec. On peut donc imaginer détecter des nogoods plus petits en utilisant la proposition 4 pour effectuer un saut dans l'arbre de recherche. Il n'y a pas un unique moyen d'effectuer un retour-arrière intelligent puisqu'on peut envisager l'utilisation d'ensembles-tests de culpabilité différents. Une politique assez naturelle est de simplement considérer la variable impliquée dans la dernière décision prise. C'est ce que nous appelons le *raisonnement à partir du dernier conflit (LC)*. Une sous-séquence coupable identifiée en utilisant cette politique est alors appelée une lc-sous-séquence.

Définition 26 (lc-sous-séquence)**.** *Soit P un réseau de contraintes, $\Sigma = \langle \delta_1, \ldots, \delta_i \rangle$ une séquence de décisions construite à partir de P telle que $\{\delta_1, \ldots, \delta_i\}$ est un nogood de P et ϕ une consistance. La lc-sous-séquence de Σ par rapport à ϕ est la sous-séquence coupable de Σ par rapport à ϕ et $\{X_i\}$ où X_i est la variable impliquée dans δ_i. L'ensemble-test de culpabilité $\{X_i\}$ est appelé le lc-ensemble-test de Σ.*

En d'autres termes, la lc-sous-séquence d'une séquence de décisions Σ menant à une inconsistance se termine par la plus récente décision telle que, lorsque celle-ci est négativée, il existe une valeur qui peut être assignée, sans mener à une inconsistance avec ϕ, à la variable impliquée dans la dernière décision de Σ. Remarquons qu'il est possible que la décision coupable δ_j de Σ soit une décision négative et, comme cela a été indiqué auparavant, soit la dernière décision de Σ. Si $j = i$, il est possible de trouver une autre valeur dans le domaine de la variable impliquée dans la dernière décision de Σ compatible avec toutes les autres décisions de Σ. Plus précisément, si δ_i est la décision coupable de Σ et correspond à une décision négative ($X_i \neq a_i$), alors nous avons nécessairement $\phi(P|_{\{\delta_1,\ldots,\delta_{i-1},X_i=a_i\}}) \neq \bot$. D'un autre coté, si δ_i est la décision coupable de Σ mais correspond à une décision positive ($X_i = a_i$) alors il existe une valeur $a'_i \neq a_i$ dans $dom(X_i)$ telle que $\phi(P|_{\{\delta_1,\ldots,\delta_{i-1},X_i\neq a_i,X_i=a'_i\}}) \neq \bot$.

1.1.2 Le raisonnement à partir du dernier conflit

L'identification et l'exploitation des nogoods décrits plus haut peut facilement être intégrée dans un ϕ-algorithme de recherche grâce à une simple modification de l'heuristique de choix de variable. Cette approche implémente le *raisonnement à partir du dernier conflit* (LC en abrégé) [Lecoutre *et al.*, 2006]. Lorsqu'un ϕ-nogood est identifié à partir d'une séquence de décisions Σ menant à un échec, il est possible d'effectuer un saut sûr, jusqu'à la dernière décision contenue dans la lc-sous-séquence de Σ. On peut également noter que l'ensemble des décisions contenues dans une lc-sous-séquence peut ne pas être un nogood minimal. Les techniques de backjumping effectuent un retour-arrière intelligent en deux étapes. Lorsqu'un échec est rencontré, ces techniques analysent tout d'abord la décision coupable responsable de l'échec, puis dans un deuxième temps effectuent un saut sur cette décision. Le raisonnement à partir du dernier conflit identifie la décision coupable sans analyser directement les raisons de l'échec. Cette décision est identifiée en

effectuant des retours-arrière successifs dans l'arbre de recherche (de ce fait aucun saut n'est nécessaire). On bénéficie ainsi de "l'effet backjumping" sans pour autant effectuer de retours-arrière intelligents.

En pratique, LC n'est exploité que lorsqu'une impasse a été atteinte à partir d'un nœud ouvert de l'arbre de recherche, c'est-à-dire, à partir d'une décision positive puisque dans une recherche utilisant un schéma de branchement binaire, les décisions positives sont généralement prises en premier lieu. Cela veut dire que LC sera déclenchée si et seulement si δ_i (la dernière décision de la séquence mentionnée en définition 26) est une décision positive. Pour implémenter LC il est alors suffisant de (1) enregistrer la variable dont l'assignation a mené directement à une inconsistance, et de (2) toujours préférer cette variable dans les décisions suivantes plutôt que les choix fournis par l'heuristique principale – quelque soit l'heuristique utilisée. Remarquons également que LC ne nécessite aucun coût en espace supplémentaire.

La figure 1.1 illustre le raisonnement à partir du dernier conflit. La branche la plus à gauche sur la figure correspond aux décisions positives $X_1 = v_1, \ldots, X_i = v_i$, qui combinées mènent à un conflit. A cette étape, X_i est enregistrée par LC pour une utilisation ultérieure, et v_i est supprimée de $dom(X_i)$, c'est-à-dire $X_i \neq v_i$. Au lieu de poursuivre la recherche avec une nouvelle variable, X_i est assignée avec une nouvelle valeur v'. Dans notre illustration, ceci mène une fois encore à un conflit, v' est supprimée de $dom(X_i)$, et le procédé est réitéré jusqu'à ce que toutes les valeurs de $dom(X_i)$ soient supprimées, ce qui nous amène à un domaine vide. L'algorithme revient alors sur l'assignation $X_{i-1} = v_{i-1}$, et poursuit la recherche vers la branche de droite $X_{i-1} \neq v_{i-1}$. Comme la variable X_i est toujours enregistrée par LC, elle est sélectionnée en priorité et toutes les valeurs de $dom(X_i)$ sont exclues par le même procédé que précédemment. L'algorithme retourne finalement sur la décision $X_j = v_j$, et poursuit dans la branche de droite $X_j \neq v_j$. Puisque X_i est toujours la variable enregistrée, celle-ci est encore préférée et les valeurs de $dom(X_i)$ testées. Mais cette fois-ci, l'une d'entre elles (v) ne mène pas à un échec et la recherche peut continuer avec l'assignation $X_i = v$. La variable X_i est alors libérée, et le choix des décisions suivantes est laissé à l'heuristique principale jusqu'à ce qu'un autre conflit survienne.

En utilisant un opérateur d'inférence établissant la consistance d'arc ($\phi = AC$) pour identifier la sous-séquence coupable, on obtient les résultats de complexité suivants.

Proposition 5. *Soit P un réseau de contraintes, ϕ une consistance et $\Sigma = \langle \delta_1, \ldots, \delta_i \rangle$ une séquence de décisions construite à partir de P telle que $\{\delta_1, \ldots, \delta_i\}$ est un ϕ-nogood de P. Le calcul de la lc-sous-séquence de Σ par rapport à ϕ est dans le pire des cas $O(id\gamma)$ où γ correspond à la complexité temporelle dans le pire des cas pour établir ϕ.*

Démonstration. Le cas le plus défavorable apparaît lorsque la lc-sous-séquence de Σ que l'on calcule est vide. Dans ce cas, cela veut dire que pour chaque décision, il faut vérifier la singleton ϕ-consistance de X_i. Comme contrôler la singleton ϕ-consistance d'une variable correspond à au plus d appels à l'algorithme établissant ϕ, la complexité temporelle dans le pire des cas est id fois la complexité de l'algorithme ϕ, noté γ ici. La complexité temporelle globale est donc de $O(ed\gamma)$. □

Lorsque LC est greffé à un algorithme MAC on obtient les résultats de complexité suivants :

Corollaire 1. *Soit $P = (\mathscr{X}, \mathscr{C})$ un réseau de contraintes binaires et $\Sigma = \langle \delta_1, \ldots, \delta_i \rangle$ une séquence de décisions correspondant à une branche construite par MAC et menant à un échec. La complexité temporelle dans le pire des cas, pour MAC, d'effectuer un retour-arrière sur la décision coupable de Σ (par rapport à AC) est $O(end^3)$.*

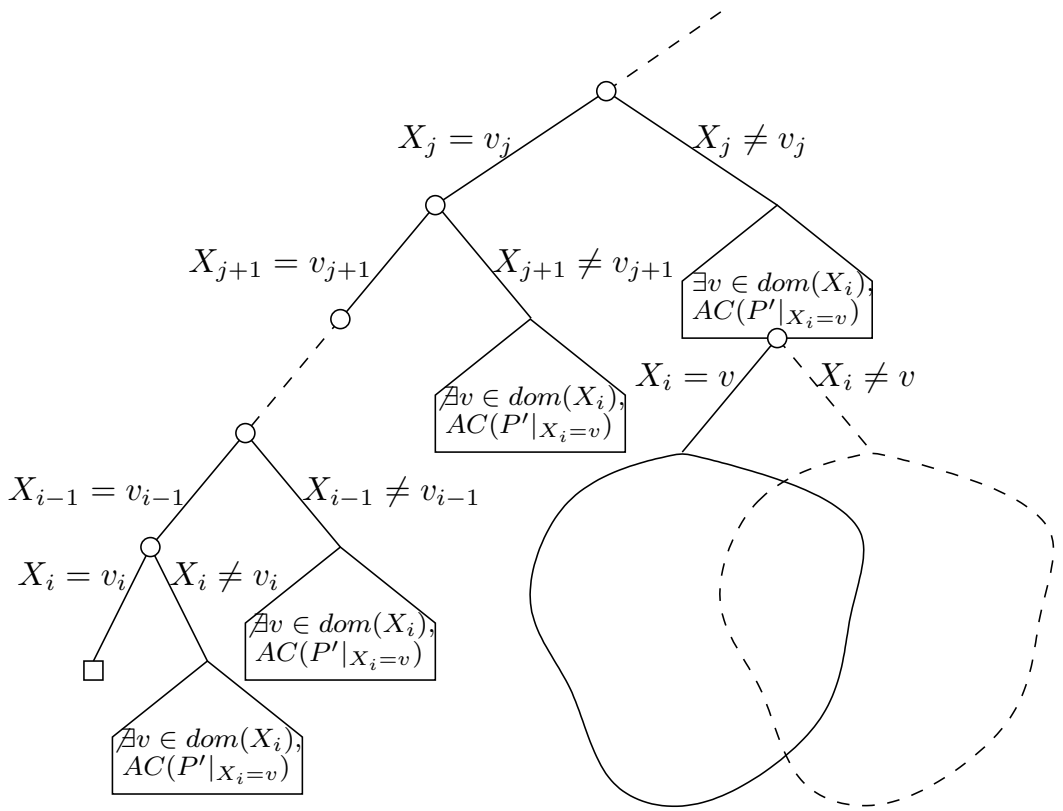


FIG. 1.1 – Raisonnement à partir du dernier conflit illustré sur une partie d'un arbre de recherche. P' représente le réseau de contraintes obtenu à chaque nœud après avoir effectué les décisions de la branche courante et appliqué l'opérateur d'inférence ϕ.

Démonstration. Tout d'abord, nous savons que les décisions positives sont choisies en premier lieu par MAC, le nombre de nœuds ouverts dans une branche de l'arbre de recherche est donc au plus n. De plus, pour chaque nœud que l'on ferme, il n'est pas nécessaire de contrôler la singleton arc consistance de la variable X_i. La complexité temporelle d'un algorithme optimal établissant AC étant $O(nd^2)$, nous obtenons une complexité globale de $O(end^3)$. □

Remarquons que s'il existe une variable singleton arc-inconsistante dans un réseau de contraintes P, alors dès que celle-ci est sélectionnée par l'heuristique de choix de variable, la réfutation du réseau de contraintes P peut se faire en temps polynomial. Plus formellement :

Corollaire 2. *Soit $P = (\mathscr{X}, \mathscr{C})$ un réseau de contraintes tel que $\exists X \in \mathscr{X} \mid \forall v \in dom(X), \phi(P|_{X=v}) = \bot$. Si on utilise MAC-LC alors dès que X est sélectionnée par l'heuristique de choix de variable, la réfutation de P se termine en temps polynomial par rapport à e, n et d.*

Evidemment P peut être vu comme le CN obtenu à chaque nœud de l'arbre de recherche. Pour illustrer ceci, considérons encore la figure 1.1, et supposons que la variable X_i est une variable singleton arc-inconsistante du réseau associé au nœud j. A partir du moment où l'algorithme essaiera d'assigner la variable X_i, la durée nécessaire pour l'exploration du sous-arbre dont la racine est le nœud j, est polynomiale par rapport à e, n et d.

TAB. 1.1 – Coût de MAC-brelaz (temps limite fixé à 2 heures)

Instance		*SBT*	*CBJ*	*DBT*	*LC*
qk-25-25-5-*add*	*cpu*	$> 2h$	11,7	12,5	58,9
	nuds	–	703	691	10 053
qk-25-25-5-*mul*	*cpu*	$> 2h$	$> 2h$	$> 2h$	66,6
	nuds	–	–	–	9 922

1.1.3 Prévenir le phénomène de thrashing avec LC

Le thrashing est le fait d'explorer de façon répétitive les mêmes sous-arbres. Ce phénomène doit être étudié avec soin car un algorithme sujet au thrashing peut être vraiment très inefficace. Parfois, le thrashing peut être expliqué par les mauvais choix effectués plus tôt durant la recherche. Pour chaque valeur supprimée du domaine d'une variable, on peut trouver une explication (avec plus ou moins de précision) des raisons de cette suppression en déterminant les décisions qui provoquent la suppression de cette valeur. En enregistrant de telles explications et en exploitant ces informations, on peut espérer effectuer un saut dans l'arbre de recherche à un niveau où la variable coupable sera ré-assignée, évitant ainsi le thrashing.

Dans certains cas, les techniques de retours-arrière intelligents ne peuvent identifier des variables coupables pertinentes même s'il existe bien un phénomène de thrashing. Par exemple, considérons une instance insatisfaisable du problème jouet des reines et cavaliers comme proposé dans [Boussemart *et al.*, 2004a]. Ce problème combine deux sous-problèmes : le problème satisfaisable des reines et celui insatisfaisable des cavaliers. Quand les deux sous-problèmes sont fusionnés sans aucune interaction (il n'y a pas de contraintes impliquant à la fois une variable des reines et une variable des cavaliers comme dans l'instance *qk*-25-25-5-*add*), une technique de retours-arrière intelligents telle que CBJ ou DBT peut prouver l'insatisfaisabilité du problème à partir de l'insatisfaisabilité du sous-problème des cavaliers (en effectuant un retour-arrière jusqu'à la racine de l'arbre de recherche). Quand les deux sous-problèmes sont fusionnés avec des interactions (les reines et les cavaliers ne peuvent pas être disposés sur la même case de l'échiquier comme sur l'instance *qk*-25-25-5-*mul*), CBJ et DBT deviennent sujettes au thrashing (quand elles sont utilisées avec une heuristique de choix de variable standard telle que *dom*, *brelaz* et *dom/ddeg*). En effet ces techniques considèrent que la dernière reine assignée fait partie des raisons de l'échec. Le problème est que, même s'il existe plusieurs explications pour la suppression d'une valeur, seule la première rencontrée est enregistrée. Or sur ce problème, les variables reine font parties de la première explication rencontrée. On peut remarquer qu'il est également possible de calculer après chaque échec un nogood minimal (sans avoir à maintenir d'explications) avec, par exemple, une technique telle que QuickXplain [Junker, 2004].

Le raisonnement à partir des derniers conflits est un nouveau moyen d'éviter le thrashing, bien qu'il s'agisse d'une méthode prospective, i.e. une méthode qui dirige la recherche et non qui s'intéresse aux échecs rencontrés (méthode rétrospective). En effet, guider la recherche vers la dernière décision coupable revient à effectuer une sorte de retour-arrière sur cette décision.

Le tableau 1.1 nous montre les possibilités de LC pour éviter le phénomène de thrashing sur deux instances citées ci-dessus. D'un coté, SBT, CBJ et DBT ne peuvent pas éviter le thrashing pour l'instance *qk*-25-25-5-*mul* puisque l'instance reste non résolue en plus de 2 heures (même avec d'autres heuristiques standard). De l'autre, en 1 minute, LC (avec SBT) peut prouver l'insatisfaisabilité de cette instance. La raison est que toutes les variables cavalier sont singleton

		brelaz				*dom/wdeg*			
		SBT	*CBJ*	*DBT*	LC_1	*SBT*	*CBJ*	*DBT*	LC_1
aim-100 (#24)	*cpu*	647 (12)	8,94	7,18	50,4	0,60	0,48	0,38	0,54
	nuds	9 488*K*	82 751	33 625	718*K*	3 106	507	710	2 485
aim-200 (#24)	*cpu*	985 (18)	305 (4)	394 (5)	740 (14)	5,82	3,16	66,1 (1)	4,75
	nuds	12*M*	1 441*K*	752*K*	9 071*K*	64 798	11 808	94 127	52 857
dsjc/myciel/... (#22)	*cpu*	105 (1)	57,5	77,8 (1)	9,54	13,6	20,2	57,5	10,2
	nuds	1 500*K*	358*K*	231*K*	93 070	150*K*	110*K*	182*K*	110*K*
e0ddr1-10 (#10)	*cpu*	720 (6)	604 (5)	720 (6)	600 (5)	511 (4)	561 (4)	840 (7)	445 (3)
	nuds	6 487*K*	619*K*	689*K*	5 608*K*	4 588*K*	674*K*	1 550*K*	4 164*K*
enddr1-10 (#10)	*cpu*	360 (3)	360 (3)	360 (3)	259 (2)	123 (1)	133 (1)	241 (2)	124 (1)
	nuds	3 162*K*	379*K*	350*K*	2 274*K*	1 101*K*	155*K*	390*K*	1 127*K*
ehi-85-297 (#100)	*cpu*	301 (8)	137 (1)	175 (2)	0,69	0,87	2,04	14,3	0,43
	nuds	362*K*	67 897	36 653	311	1 292	926	1646	146
ehi-90-315 (#100)	*cpu*	402 (14)	152 (4)	207	0,70	0,85	2,56	14,8	0,44
	nuds	431*K*	63 377	38 258	282	1 210	1 192	1 514	140
driverlogw (#7)	*cpu*	79,4	9,88	178 (1)	14,0	3,09	6,88	23,6	2,58
	nuds	30 651	2 179	5 752	4 651	3 429	2 136	1 382	2 362
fapp02 (#11)	*cpu*	318 (2)	233 (2)	241 (2)	7,2	9,14	22,7	30,0	7,51
	nuds	244*K*	7 690	6 514	291	966	489	527	369
fapp03 (#11)	*cpu*	115 (1)	130 (1)	22,3	8,41	7,48	21,2	21,6	7,79
	nuds	11 023	7 425	110	237	168	109	109	181
graphs (#14)	*cpu*	86,8 (1)	3,4	89,0 (1)	1,59	1,19	2,95	3,04	1,28
	nuds	521*K*	318	3 579	497	313	314	314	313

TAB. 1.2 – Résultats obtenus par MAC, MAC-LC_1, DBT et CBJ en utilisant les heuristiques *brelaz* et *dom/wdeg*

arc inconsistantes. Quand une telle variable est atteinte, LC guide la recherche jusqu'à la racine de l'arbre de recherche (c.f. Corollaire 2). Le tableau 1.2 présente les résultats obtenus sur différentes séries de problèmes de la seconde compétition internationale de solveurs CSP par différentes techniques dites "anti-thrashing" (i.e. DBT, CBJ, LC...). Le temps limite a été fixé à 1 200 secondes par instance et pour chaque série, le nombre d'instances non résolues est indiqué entre parenthése. Les résultats sont comparés en terme de temps cpu (cpu) en seconde et de nombre de nœuds (nœuds). Les séries présentées dans ce tableau sont composées d'instances structurées (ou réelles) sur lesquelles des techniques de retours-arrière intelligents peuvent être efficaces. Globalement, LC améliore l'efficacité de l'algorithme MAC et celui-ci (équippé du raisonnement à partir du dernier conflit) s'avère être plus compétitif que des algorithmes comme DBT et CBJ. Seule, la résolution de la série *aim* reste plus efficace avec une technique de retours-arrière intelligents lorsqu'on utilise une heuristique standard (brelaz). Les résultats sont moins évidents lorsqu'on utilise l'heuristique *wdeg* puisque celle-ci permet déjà d'éviter en partie le phénomène de thrashing.

1.2 Généralisation aux k derniers conflits

1.2.1 Principe

Une fois qu'une lc-sous-séquence a été identifiée, on peut imaginer étendre le raisonnement à partir du dernier conflit en prenant en compte également dans l'ensemble-test de culpabilité la variable X_j impliquée dans la décision coupable. Pour cela, on met à jour l'ensemble-test de culpabilité utilisé pour identifier la lc-sous-séquence. Celui-ci n'est alors plus seulement composé de la variable X_i impliquée dans la dernière décision mais des variables $\{X_i, X_j\}$. L'idée sous-jacente est d'utiliser l'ensemble $\{X_i, X_j\}$ comme nouvel ensemble-test de culpabilité dans le but d'aider à effectuer des retours-arrière plus haut dans l'arbre de recherche. Il est même possible

de généraliser ce raisonnement de manière récursive. Avec un tel mécanisme, on peut espérer identifier de petits ensembles de variables incompatibles, impliquées dans des décisions de la branche courante et entremêlées avec d'autres décisions non pertinentes. On peut alors espérer éviter l'exploration de nombreux sous-arbres inutiles.

Avant d'illustrer notre approche, nous définissons formellement le concept de lc(k)-sous-séquence.

Définition 27 (lc(k)-sous-séquence)**.** *Soit P un réseau de contraintes, $\Sigma = \langle \delta_1, \ldots, \delta_i \rangle$ une séquence de décisions construite à partir de P telle que $\{\delta_1, \ldots, \delta_i\}$ est un nogood de P et ϕ une consistance. On définit récursivement la k^{ieme} lc-sous-séquence et le k^{ieme} lc-ensemble-test de Σ, notés lc(k)-sous-séquence et lc(k)-ensemble-test, par rapport à ϕ comme suit :*

- *La lc(1)-sous-séquence et le lc(1)-ensemble-test de Σ par rapport à ϕ correspondent respectivement à la lc-sous-séquence et au lc-ensemble-test de Σ par rapport à ϕ ;*
- *Si la lc(k)-sous-séquence de Σ est vide, alors la lc(k+1)-sous-séquence et le lc(k+1)-ensemble-test de Σ par rapport à ϕ correspondent respectivement à la lc(k)-sous-séquence et au lc(k)-ensemble-test de Σ par rapport à ϕ. Dans le cas contraire, avec Σ_k et S_k représentant respectivement la lc(k)-sous-séquence et le lc(k)-ensemble-test de Σ, la lc(k+1)-sous-séquence de Σ est la sous-séquence coupable de Σ_k par rapport à ϕ et $S_k \cup \{X_k\}$ où X_k est la variable impliquée dans la dernière décision de Σ_k. Le lc(k+1)-ensemble-test de Σ est $S_k \cup \{X_k\}$.*

Si on considère une fonction qui construit itérativement les lc(k)-sous-séquences, on peut montrer que celle-ci atteint un point fixe en temps fini.

Proposition 6. *Soit f(k) une fonction qui construit les lc(k)-sous-séquences. $\exists j \geq 1$ telle que $\forall k > j$, la lc(k)-sous-séquence et le lc(k)-ensemble-test sont respectivement égaux à la lc(j)-sous-séquence et au lc(j)-ensemble-test.*

Démonstration. Pour atteindre le point fixe, deux cas doivent être considérés. Le premier intervient lorsqu'on obtient une séquence vide à une étape j donnée. A une étape $k > j$ la lc(k)-sous-séquence est également une séquence vide puisqu'à partir de l'étape j le lc(j)-ensemble-test n'est plus mis à jour. Le second cas intervient lorsque la décision coupable d'une lc(j)-sous-séquence est la dernière décision de la sous-séquence. Il existe alors une valeur ne menant pas à un échec pour la variable impliquée dans cette dernière décision. □

La proposition suivante est une généralisation de la proposition 4. Celle-ci peut être démontrée par récurrence sur k.

Proposition 7. *Soit P un réseau de contraintes et $\Sigma = \langle \delta_1, \ldots, \delta_i \rangle$ une séquence de décisions construite à partir de P telle que $\{\delta_1, \ldots, \delta_i\}$ est un nogood de P. Pour tout $k \geq 1$, l'ensemble des décisions contenues dans la lc(k)-sous-séquence de Σ est un nogood de P.*

Démonstration. Soit $\langle \delta_1, \ldots, \delta_j \rangle$ la lc(k)-sous-séquence (non vide) coupable de Σ. Démontrons par récurrence sur k que $\forall k \geq 1$ l'hypothèse suivante, notée H(k), est vraie :

H(k) : la lc(k)-sous-séquence de Σ est un nogood

Tout d'abord, montrons que H(1) est vraie. Grâce à la proposition 4 nous savons que la lc(1)-sous-séquence est un nogood.

Ensuite, montrons que, pour $k > 1$, si H($k-1$) est vraie alors H(k) est vraie également. Comme $k > 1$ et H(k) est vraie, nous savons que, par hypothèse de récurrence, la lc(k-1)-sous-séquence est un nogood. Notons S_{k-1} le lc(k-1)-ensemble-test correspondant à la lc(k-1)-sous-séquence. D'après la définition 27, le lc(k)-ensemble-test est égal à $S_{k-1} \cup \{X_k\}$ où X_k est la variable impliquée dans la dernière décision de la lc(k-1)-sous-séquence. D'après la proposition 4 la lc(k)-sous-séquence correspond à la sous-séquence coupable et est un nogood. □

En pratique, on utilise la version généralisée de LC dans le contexte d'une recherche effectuée par un algorithme de recherche avec retours-arrière. LC_i dénote l'approche qui consiste à calculer la lc(i)-sous-séquence d'un nogood donné. Naturellement, si pendant ce calcul, un point fixe est identifié, celui-ci peut être directement retourné.

Finalement, on restreindra les pivots aux décisions positives uniquement. Si on identifie un pivot associé à une décision négative, d'après la définition 25 cela veut dire que la décision positive correspondante ne provoque pas d'échec après avoir établi une consistance ϕ (et en considérant la séquence de décisions prise sur la branche menant à cette décision et un lc(k)-ensemble-test donné). Or si cette décision a été réfutée, les décision positives étant prises en premier, cette décision positive mène donc forcément à un échec et il n'est pas pertinent de considérer cette décision comme un pivot. Néanmoins, la décision δ_i qui est systématiquement insérée dans les séquences coupables n'est pas nécessairement une décision positive puisqu'il n'est pas possible de trouver sur n'importe quelle branche d'un arbre de recherche une décision positive suivie d'une décision négative impliquant la même variable.

1.2.2 Illustration

Principe général

La figure 1.2 représente une vue partielle d'un arbre de recherche construit par un ϕ-algorithme de recherche basé sur un schéma de branchement binaire pour résoudre un réseau de contraintes P donné. La branche la plus à gauche correspond à une séquence de décisions Σ et se termine par un échec après avoir pris la décision $Z = a$ (par conséquent on peut déduire un ϕ-nogood de P à partir de Σ).

Par définition, le lc(1)-ensemble-test de Σ est seulement composé de la variable Z (qui est impliquée dans la dernière décision de Σ). L'algorithme assigne donc Z en priorité pour identifier la décision coupable de Σ (et en même temps la lc(1)-sous-séquence) : après avoir réfuté la décision $Z = a$, toutes les valeurs restantes dans le domaine de Z sont successivement assignées. Sur notre illustration, chacune d'entre elles mène à un échec, ce qui est représenté sur la figure par une ligne pleine pour dessiner la base du triangle dont les côtés sont étiquetés par $Z = b$ et $Z = z$. Par conséquent, comme Z n'est pas la décision coupable de Σ, l'algorithme effectue un retour-arrière sur la décision $Y = a$. On applique alors un raisonnement similaire après avoir réfuté la décision $Y = a$ et l'algorithme effectue à nouveau un retour-arrière. Lorsqu'on revient sur X, la variable coupable est détectée puisqu'il y a au moins une valeur pour Z qui ne mène pas à un échec. Ceci est représenté sur la figure en utilisant une ligne pointillée pour dessiner la base du triangle dont les côtés sont étiquetés par $Z = a$ et $Z = z$ après avoir réfuté la décision $X = a$. Remarquons que si notre objectif est de calculer la lc(1)-sous-séquence, la recherche peut alors continuer, guidée par l'heuristique de choix de variable.

Sinon, pour calculer la lc(2)-sous-séquence, le lc(2)-ensemble-test est construit en ajoutant X au lc(1)-ensemble-test. Maintenant les variables Z et X seront assignées en priorité. La décision $W = a$ n'est pas coupable puisqu'il n'est pas possible de "traverser" (i.e. trouver des valeurs compatibles pour) Z et X. La variable coupable est détectée lorsqu'on effectue un retour-arrière sur la variable V.

Si on doit calculer la lc(3)-sous-séquence, le lc(3)-ensemble-test est construit en ajoutant la variable V au lc(2)-ensemble-test. Cependant comme les variables du lc(3)-ensemble-test peuvent maintenant être assignées, le raisonnement à partir du dernier conflit est alors arrêté et la recherche continue normalement.

Le problème des reines et pions

Introduisons maintenant un problème "jouet" illustrant les possibilités du raisonnement à partir du dernier conflit généralisé pour identifier des noyaux de variables difficiles. Dans notre modélisation, chaque case d'un échiquier est numérotée de 1 à $n \times n$ et la distance entre deux cases correspond à la valeur absolue de la différence des numéros des cases. Le problème des pions consiste à aligner p pions sur $p-1$ cases successives d'un échiquier de taille $n \times n$ selon une numérotation pré-établie des cases. Plus précisément, deux pions ne peuvent pas être disposés sur une même case de l'échiquier et la distance entre deux pions doit être strictement inférieure à $p-1$ (ce qui traduit l'alignement des pions). p variables représentent les pions du problème et leur domaine représente les $n \times n$ cases de l'échiquier. A partir de $p \geq 2$, ce problème est insatisfaisable puisque l'on veut placer p pions sur $p-1$ cases. De manière intéressante, on peut montrer qu'il existe des ensembles de décision contenant $p-3$ décisions positives qui sont AC-consistantes (i.e. qui ne sont pas des AC-nogoods), mais que tous les ensembles de décision contenant $p-2$ décisions positives sont des AC-nogoods. En d'autres termes, pendant la recherche effectuée par MAC, il est possible d'instantier $p-3$ variables mais jamais $p-2$.

On peut fusionner ce problème avec le problème classique des reines. Les pions et les reines ne peuvent pas être disposés sur une même case de l'échiquier. Ce problème (comme celui des

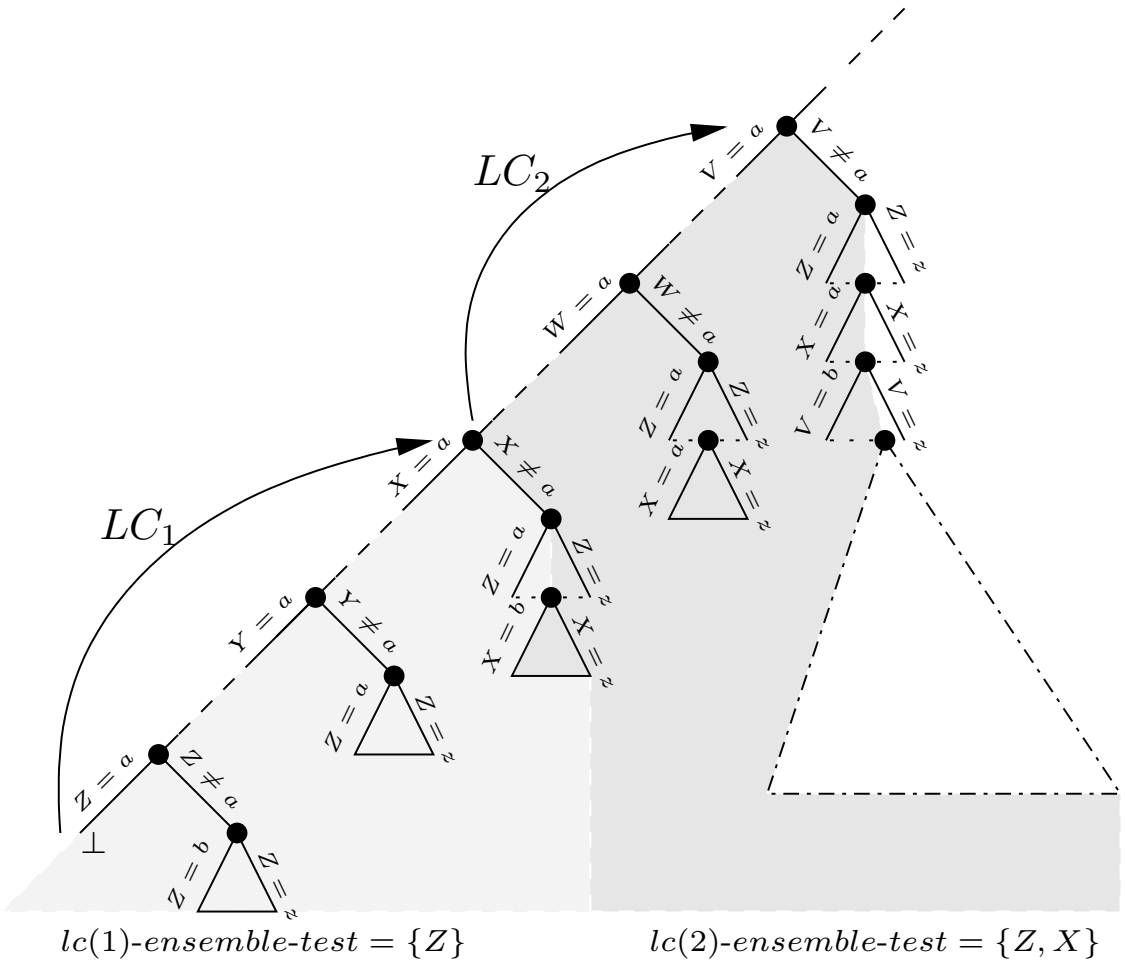


FIG. 1.2 – Le raisonnement à partir du dernier conflit généralisé

reines et cavaliers) est un problème qui génère beaucoup de thrashing. En effet, dans le pire des cas il faut démontrer l'insatisfaisabilité du problème des pions pour chaque solution du problème des reines. Le raisonnement à partir du dernier conflit généralisé à l'ordre k (LC_k) permet d'identifier un sous-ensemble de k variables incompatibles. L'idée ici, est d'identifier le sous-ensemble de variables correspondant au problème des pions, et de l'utiliser comme lc(k)-sous-ensemble pour démontrer l'insatisfaisabilité globale du problème. Plus précisément, si l'on considère un problème avec p pions, on peut alors identifier un sous-ensemble de $p-2$ variables incompatibles (puisque tous les ensembles de décisions contenant $p-2$ décisions positives sont des AC-nogoods). En utilisant le raisonnement à partir du dernier conflit à l'ordre $k-2$ (LC_{k-2}) on peut espérer identifier les variables pion incompatibles et les utiliser comme lc(k-2)-ensemble-test.

Les instances du problème des reines et pions sont notés *qp-n-p* avec p représentant le nombre de pions et n le nombre de reines. Le tableau 1.3 présente les résultats obtenus sur les instances du problème des reines et pions par l'algorithme MAC, en utilisant les heuristiques *brelaz* et *dom/wdeg* ainsi que le raisonnement LC_k ($\neg LC$ représente l'algorithme MAC seul). Plus précisément, les instances modélisent un échiquier de taille 12×12 sur lequel on dispose 12 reines et p pions allant de 3 à 9. Le temps limite a été fixé à deux heures.

Comme prévu, pour résoudre une instance *qp*-12-*p*, il est préférable d'utiliser LC_{p-2} puisque les variables correspondant aux pions peuvent être collectées par cette approche. On remarque également que si l'on utilise le raisonnement à partir du dernier conflit à l'ordre k avec $k \geq p-2$, quelque soit k, le nombre de nœuds nécessaires pour résoudre l'instance reste identique. Les variables pions sont donc bien collectées et utilisées comme lc(k)-sous-ensemble. Si $k < p-2$, la résolution du problème est nettement plus difficile (notamment avec l'heuristique *brelaz*). En effet, on ne peut identifier alors qu'un sous ensemble des $p-2$ variables incompatibles.

1.3 Expérimentations

1.3.1 Résultats obtenus avec le solveur CSP Abscon

Pour montrer l'intérêt pratique du raisonnement à partir des derniers conflits, nous avons mené des expérimentations (sur un PC Pentium IV, cadencé à $2,4GHz$ et $512Mio$ de mémoire sous Linux). Les performances ont été mesurées en terme de nombre de nœuds visités (noté nœuds dans les tableaux) ainsi qu'en terme de temps cpu (cpu) en secondes. Le temps limite a été fixé à 1200 secondes. Remarquons que pour nos expérimentations, nous avons utilisé l'algorithme MAC (avec SBT, i.e. retour-arrière chronologique), et étudié l'impact de LC_k avec différentes heuristiques de choix de variable ($dom/ddeg, dom/wdeg, brelaz$).

Nous avons tout d'abord expérimenté LC_1 et LC_2 sur des séries d'instances aléatoires. Les résultats obtenus sont indiqués dans le tableau 1.4. Le nombre d'instances non résolues en moins de 1 200 secondes est noté entre parenthèse, dans ce cas, le temps cpu indiqué doit être considéré comme une borne inférieure. Globalement, sur les instances aléatoires, l'utilisation du raisonnement à partir du dernier conflit semble pénalisante. En effet, ces instances ne contiennent aucune structure qu'il est possible d'exploiter. De manière générale, MAC seul ($\neg$ LC) est meilleur que LC_1, qui lui-même est meilleur que LC_2. Cependant les instances *geom*, $\langle 40, 80, 103, 0.8\rangle$ et $\langle 40, 180, 84, 0.9\rangle$ sont des exceptions. En effet celles-ci impliquent des contraintes de dureté élevée (c'est-à-dire qu'il y a très peu de supports par rapport au nombre de conflits) et on peut considérer que ces instances contiennent une petite structure.

Nous avons également expérimenté LC_1 et LC_2 sur des séries d'instances structurées. Dans les tableaux 1.5 et 1.6, on peut observer l'impact du raisonnement à partir du dernier conflit. Lorsque les heuristiques standard, c'est-à-dire *dom/ddeg* et *brelaz* sont utilisées, il apparaît clairement que

		brelaz								
		$\neg LC$	LC_1	LC_2	LC_3	LC_4	LC_5	LC_6	LC_7	LC_8	LC_9
qp-12-3	*cpu*	323	2,10	1,71	2,16	2,14	2,19	2,13	2,25	2,12	2,24
	nuds	1 839*K*	709	708	708	708	708	708	708	708	708
qp-12-4	*cpu*	815	1,19	0,98	1,09	1,25	1,22	1,12	1,12	0,94	2,07
	nuds	6 558*K*	2 713	2 719	2 719	2 719	2 719	2 719	2 719	2 719	2 719
qp-12-5	*cpu*	2 620	3,16	2,66	2,24	2,95	2,87	2,33	2,39	2,84	2,56
	nuds	28*M*	13 181	13 140	12 523	12 523	12 523	12 523	12 523	12 523	12 523
qp-12-6	*cpu*	*time-out*	471	11,0	10,7	9,39	9,61	9,69	10,2	9,21	9,13
	nuds		5 271*K*	66 701	75 812	67 335	67 335	67 335	67 335	67 335	67 335
qp-12-7	*cpu*	*time-out*	*time-out*	74,5	469	62,7	55,9	54,8	55,2	54,2	54,3
	nuds			584*K*	5 144*K*	432*K*	418*K*	418*K*	418*K*	418*K*	418*K*
qp-12-8	*cpu*	*time-out*	*time-out*	*time-out*	5 587	710	669	385	389	383	383
	nuds				63*M*	6 003*K*	5 820*K*	2 978*K*	2 978*K*	2 978*K*	2 978*K*
qp-12-9	*cpu*	*time-out*	*time-out*	*time-out*	*time-out*	*time-out*	6 944	*time-out*	3 126	3 092	3 116
	nuds						67*M*		24*M*	24*M*	24*M*

		dom/wdeg									
		$\neg LC$	LC_1	LC_2	LC_3	LC_4	LC_5	LC_6	LC_7	LC_8	LC_9
qp-12-3	*cpu*	3,09	1,73	2,19	2,17	2,82	2,27	2,27	2,06	1,65	2,19
	nuds	3 840	840	842	708	708	708	708	708	708	708
qp-12-4	*cpu*	1,12	1,34	1,15	2,07	1,19	1,58	1,19	1,13	1,14	1,74
	nuds	4 273	3 530	3 255	2 719	2 719	2 719	2 719	2 719	2 719	2 719
qp-12-5	*cpu*	2,36	2,77	2,95	2,13	2,33	2,57	2,39	2,33	2,28	2,38
	nuds	12 847	14 497	16 064	12 523	12 523	12 523	12 523	12 523	12 523	12 523
qp-12-6	*cpu*	9,88	12,4	13,4	10,2	9,39	9,21	9,55	9,28	9,23	9,6
	nuds	79 191	80 794	94 225	70 832	67 335	67 335	67 335	67 335	67 335	67 335
qp-12-7	*cpu*	67,0	80,2	89,0	71,8	66,2	54,9	55,4	51,8	54,9	53,7
	nuds	568*K*	544*K*	638*K*	515*K*	478*K*	418*K*	418*K*	418*K*	418*K*	418*K*
qp-12-8	*cpu*	744	589	687	554	538	459	390	364	385	363
	nuds	6 240*K*	4 083*K*	4 897*K*	3 961*K*	3 841*K*	3 390*K*	2 978*K*	2 978*K*	2 978*K*	2 978*K*
qp-12-9	*cpu*	5 884	4 887	5 651	4 743	4 722	4 328	3 689	2 947	3 127	2 932
	nuds	49*M*	34*M*	39*M*	33*M*	32*M*	31*M*	27*M*	24*M*	24*M*	24*M*

TAB. 1.3 – Résultats obtenus par MAC-LC$_k$ avec $k \in 1..9$, en utilisant les heuristiques *brelaz* et *dom/wdeg*, sur le problème des "reines et pions"

		dom/ddeg			*dom/wdeg*			*brelaz*		
		$\neg LC$	LC_1	LC_2	$\neg LC$	LC_1	LC_2	$\neg LC$	LC_1	LC_2
Instances aléatoires du Model D (100 instances par séries)										
$\langle 40, 8, 753, 0.1\rangle$	*cpu*	12,1	60,8	85,3	10,5	55,6	78,8	12,1	59,8	83,4
	nuds	45 388	232*K*	326*K*	45 393	241*K*	322*K*	45 388	232*K*	326*K*
$\langle 40, 11, 414, 0.2\rangle$	*cpu*	12,8	44,6	59,7	14,6	54,2	68,9	16,0	47,4	60,8
	nuds	58 443	203*K*	266*K*	70 560	253*K*	312*K*	73 004	213*K*	280*K*
$\langle 40, 16, 250, 0.35\rangle$	*cpu*	12,2	33,3	44,1	14,5	35,9	48,2	21,0	41,2	47,6
	nuds	59 448	158*K*	215*K*	72 556	182*K*	237*K*	104*K*	200*K*	233*K*
$\langle 40, 25, 180, 0.5\rangle$	*cpu*	16,7	27,3	41,2	17,0	34,7	41,6	46,7	44,9	44,2
	nuds	82 836	134*K*	205*K*	81 921	173*K*	200*K*	238*K*	227*K*	225*K*
$\langle 40, 40, 135, 0.65\rangle$	*cpu*	11,8	15,3	23,2	11,0	16,1	21,5	52,21	22,65	26,02
	nuds	52 814	70 113	110*K*	47 665	72 547	101*K*	242*K*	102*K*	123*K*
$\langle 40, 80, 103, 0.8\rangle$	*cpu*	13,9	10,9	16,2	6,45	9,62	14,0	129 (5)	15,3	19,5
	nuds	49 923	39 926	67 513	20 994	34 375	57 227	487*K*	57 115	74 583
$\langle 40, 180, 84, 0.9\rangle$	*cpu*	21,7	15,8	26,3	8,48	11,9	19,8	111 (3)	16,4	21,1
	nuds	55 403	39 281	79 280	17 348	29 047	62 003	317*K*	40 407	61 516
Instances aléatoires forcées du Model RB (5 instances par séries)										
frb35-17	*cpu*	4,30	5,01	5,52	3,26	4,94	7,39	6,39	5,35	5,89
	nuds	15 844	18 983	21 439	10 160	18 816	29 564	24 872	20 518	22 952
frb40-19	*cpu*	32,7	111	64,2	25,3	98,7	126	47,3	106	128
	nuds	135*K*	463*K*	271*K*	103*K*	452*K*	564*K*	196*K*	447*K*	549*K*
geom (100 instances par séries)										
geom	*cpu*	10,2	24,5	32,8	6,92	27,4	34,3	41,6 (1)	26,6	33,8
	nuds	30 847	76 706	103*K*	21 712	85 865	115*K*	179*K*	85 396	106*K*

TAB. 1.4 – Resultats obtenus avec MAC, MAC-LC_1 et MAC-LC_2 sur des instances aléatoires (temps limite fixé à 1 200 secondes)

		dom/ddeg			*dom/wdeg*			*brelaz*		
		$\neg LC$	LC_1	LC_2	$\neg LC$	LC_1	LC_2	$\neg LC$	LC_1	LC_2
aim (24 instances par séries)										
aim-100	*cpu*	636 (10)	30,2	35,6	0,60	0,54	0,53	647 (12)	50,4	50,2
	nuds	9 150*K*	428*K*	489*K*	3 106	2 485	2 330	9 488*K*	718*K*	718*K*
aim-200	*cpu*	977 (18)	737 (13)	740 (13)	5,82	4,75	4,85	985 (18)	740 (14)	738 (14)
	nuds	12*M*	8 455*K*	8 598*K*	64 798	52 857	55 387	12*M*	9 071*K*	9 165*K*
Instances Composed (10 instances par séries)										
25-1-40	*cpu*	1 200 (10)	0,51	0,54	0,51	0,53	0,49	0,47	0,42	0,48
	nuds	13*M*	74	74	161	74	74	4	4	4
25-10-20	*cpu*	27,1	0,64	0,63	0,58	0,60	0,58	229 (1)	0,77	0,74
	nuds	272*K*	161	160	200	159	159	2 599*K*	220	198
Instances Coloring (22 instances par séries)										
dsjc/myciel/...	*cpu*	6,88	4,50	6,77	13,6	10,2	10,7	105 (1)	9,54	7,52
	nuds	41 020	32 046	38 065	150*K*	110*K*	108*K*	1 500*K*	93 070	75 256
Instances job-shop Sadeh (10 instances par séries)										
e0ddr1-10	*cpu*	960 (8)	548 (4)	501 (4)	511 (4)	445 (3)	498 (4)	720 (6)	600 (5)	492 (4)
	nuds	9 811*K*	5 412*K*	4 591*K*	4 588*K*	4 164*K*	4 615*K*	6 487*K*	5 608*K*	4 647*K*
enddr1-10	*cpu*	600 (5)	142 (1)	124 (1)	123 (1)	124 (1)	124 (1)	360 (3)	259 (2)	243 (2)
	nuds	6 535*K*	1 345*K*	1 191*K*	1 101*K*	1 127*K*	1 162*K*	3 162*K*	2 274*K*	2 270*K*
Instances Ehi (100 instances par séries)										
ehi-85-297	*cpu*	475 (13)	0,91	0,62	0,87	0,43	0,43	301 (8)	0,69	0,53
	nuds	529*K*	557	281	1 292	146	146	362*K*	311	172
ehi-90-315	*cpu*	601 (23)	1,17	0,65	0,85	0,44	0,48	402 (14)	0,70	0,53
	nuds	616*K*	674	264	1 210	140	140	431*K*	282	155
Instances open-shop Taillard (30 instances par séries)										
os-5	*cpu*	646 (16)	230 (5)	170 (2)	119 (2)	77,0	92,2 (1)	727 (18)	200 (2)	185 (3)
	nuds	3409*K*	959*K*	697*K*	541*K*	324*K*	396*K*	4189*K*	832*K*	763*K*
os-7	*cpu*	1103 (27)	820 (19)	772 (17)	918 (22)	759 (18)	820 (18)	1 143 (28)	862 (20)	809 (19)
	nuds	804*K*	315*K*	291*K*	504*K*	276*K*	350*K*	897*K*	378*K*	311*K*
os-10	*cpu*	1 095 (27)	862 (21)	819 (19)	781 (19)	806 (19)	802 (19)	1 081 (27)	914 (22)	841 (20)
	nuds	435*K*	170*K*	155*K*	176*K*	132*K*	155*K*	377*K*	160*K*	155*K*
QCP (15 instances par séries)										
qcp-10-67	*cpu*	98,2	0,56	0,50	0,47	0,52	0,47	80,3	0,54	0,49
	nuds	1 038*K*	885	366	171	169	168	897*K*	854	369
qcp-15-120	*cpu*	736 (7)	704 (7)	637 (6)	34,4	36,5	35,2	727 (7)	729 (7)	628 (6)
	nuds	3 377*K*	3 594*K*	3 334*K*	232*K*	254*K*	241*K*	3 907*K*	3 845*K*	3 491*K*
qcp-20-187	*cpu*	1 200 (15)	1 200 (15)	1 200 (15)	959 (11)	971 (11)	861 (9)	1 200 (15)	1 200 (15)	1 200 (15)
	nuds	1 651*K*	1 780*K*	1 824*K*	4 425*K*	4 298*K*	3 720*K*	1 769*K*	2 107*K*	2 202*K*

TAB. 1.5 – Résultats obtenus avec MAC, MAC-LC_1 et MAC-LC_2 sur des instances structurées (temps limite fixé à 1 200 secondes)

		dom/ddeg			*dom/wdeg*			*brelaz*		
		$\neg LC$	LC_1	LC_2	$\neg LC$	LC_1	LC_2	$\neg LC$	LC_1	LC_2
Instances Driver (7 instances par séries)										
driverlogw	*cpu*	79,5	10,5	9,98	3,09	2,58	2,31	79,4	14,0	13,9
	nuds	25 153	3 042	2 901	3 429	2 362	2 018	30 651	4 651	4 742
Instances FAPP (11 instances par séries)										
fapp02	*cpu*	564 (5)	8,03	7,71	9,14	7,51	7,42	318 (2)	7,20	7,04
	nuds	688*K*	582	415	966	369	337	244*K*	291	270
fapp03	*cpu*	115 (1)	7,83	7,74	7,48	7,79	7,74	115 (1)	8,41	8,09
	nuds	9 694	153	208	168	181	147	11 023	237	211
fapp04	*cpu*	225 (2)	9,60	9,79	12,0	9,15	20,2	658 (6)	96,8	42,5
	nuds	123*K*	297	364	738	319	1 693	397*K*	17 771	3 836
RLFAP Graphs (12 et 14 instances par séries)										
graphMods	*cpu*	800 (8)	315 (3)	51,5	2,28	3,80	1,50	1 000 (10)	303 (3)	2,53
	nuds	1 585*K*	858*K*	140*K*	5 509	14 601	2 208	2 350*K*	1 642*K*	3 185
graphs	*cpu*	1,35	1,37	1,37	1,19	1,28	1,26	86,8 (1)	1,59	1,46
	nuds	313	313	313	313	313	313	521*K*	497	378
Radar surveillance (50 instances par séries)										
radar-8-24-3-2	*cpu*	408 (17)	1,70	0,48	0,18	0,17	0,16	210 (8)	0,84	0,22
	nuds	4 651*K*	14 699	3 085	122	106	107	2 214*K*	5 804	657
radar-8-30-3-0	*cpu*	423 (17)	24,8 (1)	8,03	0,21	0,19	0,21	101 (4)	0,94	1,92
	nuds	4 727*K*	141*K*	43 635	219	209	213	1 001*K*	6 067	10 217

TAB. 1.6 – Résultats obtenus avec MAC, MAC-LC_1 et MAC-LC_2 sur des instances réelles (temps limite fixé à 1 200 secondes par instance)

		brelaz					*dom/wdeg*				
		$\neg LC$	LC_1	LC_2	LC_3	LC_4	$\neg LC$	LC_1	LC_2	LC_3	LC_4
BlackHole-4	*cpu*	*time-out*	12,4	5,85	4,98	4,36	2,9	3,11	2,84	2,95	2,80
	nuds		106*K*	35 021	23 687	19 964	6 293	9 166	7 662	8 437	8 416
cc-7-7-3	*cpu*	154	36,1	46,3	42,0	41,7	23,6	30,3	26,6	27,1	30,9
	nuds	732*K*	171*K*	217*K*	198*K*	194*K*	131*K*	174*K*	146*K*	144*K*	165*K*
cc-9-9-2	*cpu*	89,9	4,94	4,68	5,15	5,13	1,01	1,00	0,96	0,98	0,99
	nuds	216*K*	10 823	10 823	10 823	10 823	3 387	3 457	3 457	3 457	3 457
driverlogw-02c	*cpu*	7,64	3,58	3,72	3,19	3,26	3,19	2,88	2,78	2,92	2,92
	nuds	8 130	1 389	1 522	1 055	907	1 588	911	869	869	869
driverlogw-09	*cpu*	534	87,2	87,0	80,5	79,8	10,2	8,36	7,07	7,80	7,57
	nuds	200*K*	28 959	29 678	28 456	28 456	12 862	9 496	7 464	8 063	8 063
e0ddr2-10-by-5-1	*cpu*	*time-out*	*time-out*	463	374	379	110	281	191	272	724
	nuds			3 052*K*	2 471*K*	2 757*K*	812*K*	2 024*K*	1 329*K*	2 085*K*	5 319*K*
enddr1-10-by-5-10	*cpu*	*time-out*	186	30,0	34,9	23,9	20,4	33,6	31,9	28,1	24,5
	nuds		1 472*K*	254*K*	282*K*	210*K*	141*K*	268*K*	261*K*	233*K*	184*K*
fapp02-0250-5	*cpu*	*time-out*	7,82	7,00	7,23	7,26	7,99	8,80	8,37	8,48	8,57
	nuds		323	323	323	323	851	685	632	638	638
fapp04-0300-5	*cpu*	*time-out*	753	344	264	315	18,8	9,74	9,67	10,5	14,2
	nuds		97 127	32 394	23 368	28 227	1 734	353	332	318	1 078
langford-3-12	*cpu*	25,7	207	383	370	344	23,2	120	129	108	115
	nuds	157*K*	1 186*K*	2 086*K*	1 837*K*	1 548*K*	90 122	441*K*	433*K*	416*K*	384*K*
langford-4-12	*cpu*	6,34	29,2	54,2	61,4	49,4	5,22	17,6	18,6	17,2	16,2
	nuds	18 608	94 941	162*K*	172*K*	124*K*	12 437	36 742	39 112	35 844	33 769
os-taillard-10-100-5	*cpu*	*time-out*	*time-out*	493	30,2	*time-out*	*time-out*	*time-out*	*time-out*	*time-out*	*time-out*
	nuds			118*K*	5 265						
os-taillard-5-100-5	*cpu*	*time-out*	25,9	15,6	12,3	42,3	14,9	5,30	5,52	3,43	3,55
	nuds		91 561	47 521	36 897	144*K*	55 637	12 054	15 651	7 075	7 903
qcp-15-120-12	*cpu*	*time-out*	*time-out*	611	14,3	94,9	0,55	0,52	0,51	0,52	0,45
	nuds			3 206*K*	84 304	477*K*	782	660	505	531	531
qcp-20-187-11	*cpu*	*time-out*	*time-out*	*time-out*	*time-out*	*time-out*	3,91	1,46	1,40	1,25	1,13
	nuds						15 992	4 992	5 083	3 860	3 753
queenAttacking-5	*cpu*	9,94	3,62	3,13	1,96	3,17	1,42	2,79	2,19	3,1	2,13
	nuds	93 677	31 272	24 995	16 107	22 937	10 533	19 482	14 189	14 990	14 295
queenAttacking-6	*cpu*	*time-out*	290	401	195	420	130	120	143	263	81,4
	nuds		1 980*K*	2 407*K*	1217*K*	2 689*K*	769*K*	676*K*	760*K*	1 450*K*	431*K*
rlfap-graph9-f10	*cpu*	*time-out*	*time-out*	7,49	5,54	4,22	1,53	1,52	1,49	1,43	1,63
	nuds			14 661	15 520	14 950	2 041	2 693	2 754	2 477	3 716
rlfap-scen11	*cpu*	558	7,80	2,32	1,87	2,61	1,87	7,47	5,90	4,86	2,60
	nuds	2 456*K*	31 793	5 134	4 103	5 854	4 540	35 028	29 465	21 120	4 948
ruler-34-9-a3	*cpu*	24,5	14,2	13,8	15,2	16,8	13,2	9,30	9,57	10,7	12,5
	nuds	18 230	8 011	8 296	9 254	11 066	9 144	7 740	8 647	10 671	12 767
ruler-34-9-a4	*cpu*	83,8	20,5	35,5	34,2	30,1	16,7	17,5	26,3	24,1	28,9
	nuds	55 129	11 159	22 480	22 908	20 840	8 723	9 163	15 645	15 714	20 536
tsp-20-366	*cpu*	12,6	11,3	6,09	6,06	4,61	1,67	1,77	2,32	1,35	1,15
	nuds	26 777	24 013	12 286	10 444	7 564	2 029	2 261	3 063	1 457	1 175
tsp-25-190	*cpu*	78,0	29,3	213	88,7	272	66,6	118	175	205	64,9
	nuds	147*K*	56 091	336*K*	133*K*	404*K*	83 894	133*K*	232*K*	246*K*	83 311

TAB. 1.7 – Résultats obtenus par MAC-LC$_k$ avec $k \in 0..4$, en utilisant les heuristiques *brelaz* et *dom/wdeg*, sur des instances académiques et réelles)

LC améliore l'efficacité de l'algorithme MAC, que ce soit en temps CPU ou en nombre d'instances résolues. LC_2 est même meilleur que LC_1, spécialement sur les séries *taillard* et *graphMods*. En fait, ces instances ont une structure et une recherche "aveugle" (i.e. sans analyser de manière approfondie les raisons des conflits) est donc sujette au thrashing. Le raisonnement à partir du dernier conflit permet d'éviter ce phénomène sans influencer le comportement général des heuristiques. D'un autre coté, lorsque l'heuristique dirigée par les conflits *dom/wdeg* est utilisée, les améliorations apportées par LC sont moins importantes puisque les effets du thrashing sont déjà limités par cette heuristique. Néanmoins, le raisonnement à partir du dernier conflit peut, sur certaines séries, être un mécanisme complémentaire intéressant lorsqu'on utilise l'heuristique *dom/wdeg*. Ceci est notamment vrai sur la série d'instances *radar surveillance*.

Finalement, nous présentons dans le tableau 1.7 les résultats représentatifs, obtenus sur des instances sélectionnées de la seconde compétition internationale de CSP. Ici, les résultats sont fournis pour LC_k avec k variant de 1 à 4, et le temps limite a été fixé à 1 heure. Une fois encore, il apparaît clairement que l'utilisation de LC avec les heuristiques standard améliore de façon significative l'efficacité de l'algorithme MAC. Ce n'est pas toujours le cas lorsqu'on utilise l'heuristique *dom/wdeg* (greffée à l'algorithme MAC) pour les raisons indiquées précédemment. Notons que la plupart de ces instances ne peuvent pas être résolues de façon efficace avec une technique de retour-arrière intelligent comme CBJ ou DBT combinée à une heuristique standard. Cet aspect a été montré dans [Lecoutre *et al.*, 2004]. Globalement, LC_2 et LC_3 semblent offrir le meilleur compromis.

Finalement, Les figures 1.3 et 1.4 représentent graphiquement le temps cpu nécessaire à la résolution des instances *scen*11-fx (x variant de 1 à 8 –*scen*11-$f1$ étant l'instance la plus difficile– et tracées de haut en bas sur les figures) en fonction du LC_k utilisé (k variant de 1 à 9). Le temps limite a été fixé à 48 heures. Sur les instances difficiles et structurées de la série *scen*11, il est intéressant d'observer que les meilleurs résultats sont obtenus lorsqu'on utilise LC_k avec un k élevé. Ceci est notamment vrai lorsqu'on utilise l'heuristique *dom/ddeg* (c.f. figure 1.3), et un peu moins évident avec l'heuristique *dom/wdeg* (c.f. figure 1.4). Néanmoins notons que nous utilisons une échelle logarithmique sur l'axe des ordonnées. Ces instances contiennent un petit noyau insatisfaisable qu'il est possible de détecter avec LC_k (avec un k suffisamment grand pour capturer l'ensemble des variables du noyau).

1.3.2 Adaptation au domaine de la planification

Le raisonnement à partir du dernier conflit peut être adapté à d'autres domaines que celui des CSP. Ici, LC_1 a été adapté au domaine de la planification de type STRIPS [Fikes et Nilsson, 1971]. Un problème de planification consiste à trouver une séquence d'actions (c'est-à-dire un plan) qui permet de passer d'un état du monde initial à un état final (i.e. ensemble de buts). Un état est représenté par un ensemble de prédicats, appelé *fluents*. Une action permet de modifier l'état courant du monde, c'est-à-dire qu'elle permet de passer d'un état du monde à un autre en modifiant un certain nombre de fluents. Pour qu'une action soit applicable, il faut que l'état courant du monde vérifie les préconditions de l'action. On utilise le langage de représentation : "PDDL" pour définir les problèmes de planification [Fox et Long, 2003]. La planification temporelle est une extension de la planification classique, où on associe à chaque action une durée d'exécution et où il est permis d'exécuter plusieurs actions en parallèle (si bien sûr elles n'entrent pas en conflit l'une avec l'autre). Ces problèmes sont fortement combinatoires et de nombreux algorithmes de recherche ont été développés pour les résoudre. Classiquement pour résoudre un problème de planification, on peut effectuer une recherche dans les espaces d'états, dans les espaces de plans partiels, en utilisant un graphe de planification.

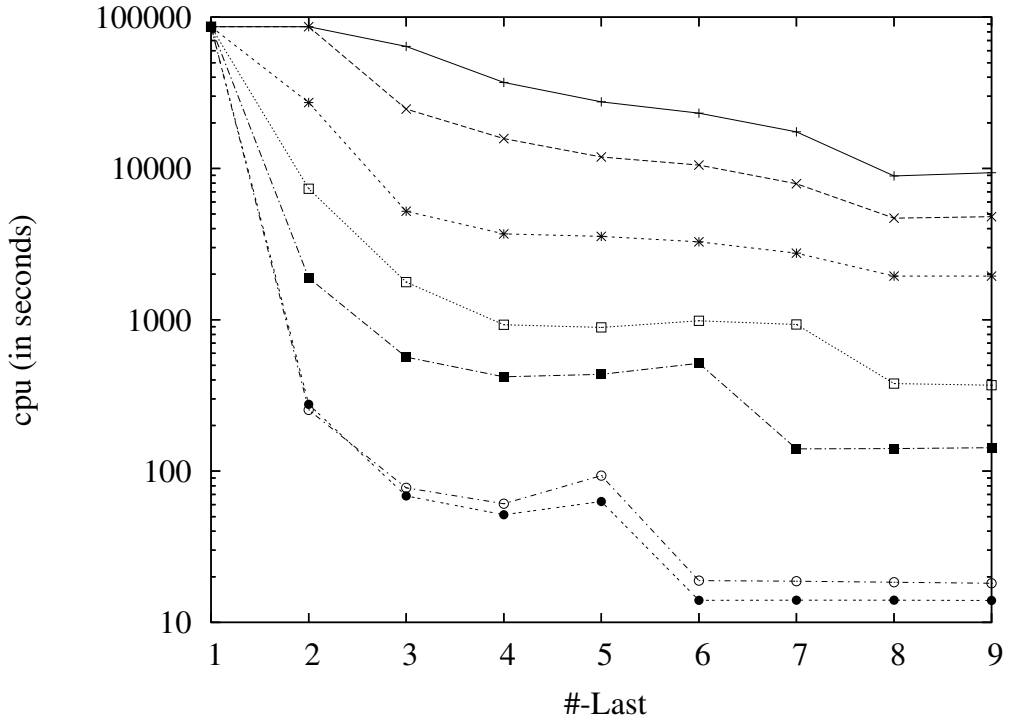


FIG. 1.3 – ddeg/dom

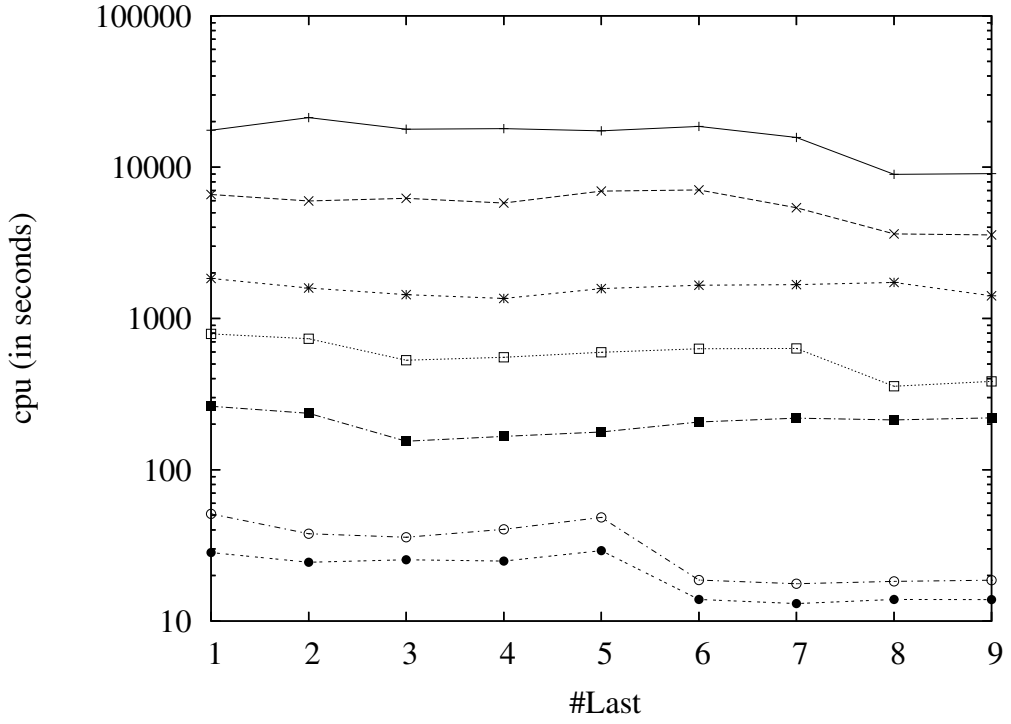


FIG. 1.4 – wdeg/dom

	#-instances			*temps total*	
	CPT	*CPT-LC*$_1$	*les deux*	*CPT*	*CPT-LC*$_1$
blocks	383	417	383	78 333, 08	42 504, 61
depots	338	401	338	40 606, 81	14 978, 33
driver	384	439	384	64 704, 73	14 613, 09
logistics	399	462	399	107 552, 01	45 387, 30
rovers	347	396	347	53 245, 29	26 371, 81
satellite	442	464	442	63 406, 42	41 149, 28

TAB. 1.8 – Nombre d'instances résolues pour chaque domaine (500 instances par domaine, temps limite fixé à 1 800 secondes.) et temps total cumulé pour les instances résolues par les deux méthodes

Le planificateur *CPT* [Vidal et Geffner, 2006], est un planificateur temporel optimal qui combine les possibilités d'exploration de l'espace de recherche de la planification dans les espaces de plans partiels avec des règles d'élagage puissantes. La principale innovation est la capacité de raisonner autour des supports, des relations de précédence et des liens causaux, en faisant intervenir les actions qui n'appartiennent pas encore à un plan partiel. L'adaptation du raisonnement à partir des derniers conflits (LC_1) à ce type de planificateur est assez immédiat. Lorsqu'on cherche un support (c'est-à-dire un couple (action,précondition)) pour une action du plan, ce couple est sélectionné en priorité tant qu'il mène à un échec.

Le tableau 1.8 présente les résultats obtenus sur des séries de problèmes des 2^{eme} et 3^{eme} compétitions internationales de planification par le planificateur optimal *CPT* (noté *CPT* dans le tableau) équipé du raisonnement à partir du dernier conflit (noté *CPT-LC*$_1$ dans le tableau). Le temps limite a été fixé à 1 800 secondes par instance et les résultats ont été comparés en terme de nombre d'instances résolues (*#-instances*) et de temps CPU cumulé (*temps total*) –pour les instances résolues par les deux méthodes–. Tout d'abord on peut noter que *CPT* équipé du raisonnement à partir du dernier conflit permet de résoudre beaucoup plus d'instances, quelque soit la série de problème. En effet au total, *CPT-LC*$_1$ résout 286 instances de plus que *CPT*. De plus, le temps total nécessaire pour résoudre les instances d'une série donnée a été nettement amélioré. Encore une fois, ceci est vrai quelque soit la série de problème.

Les figures 1.5, 1.6, 1.7, 1.8, 1.9, 1.10 représentent par domaine, sous la forme d'un nuage de points, les résultats obtenus par *CPT-LC*$_1$ comparés à ceux obtenus par *CPT*. Chaque point des graphiques représente une instance. Les coordonnées de ce point sont définies par : en abscisse, le temps nécessaire pour résoudre cette instance par *CPT* et en ordonnée, le temps nécessaire pour résoudre cette instance par *CPT-LC*$_1$. Le temps limite a été fixé à 1 800 secondes. Globalement, on remarque que *CPT* équipé du raisonnement à partir du dernier conflit est nettement plus efficace que la version standard de *CPT*. En effet la plupart des points se situent sous la diagonale ce qui indique que *CPT-LC*$_1$ est plus rapide que *CPT* pour résoudre les instances. De plus on peut noter un nombre important de points à l'extrême droite des figures. Ces points représentent des instances résolues par *CPT-LC*$_1$ mais pas par *CPT*.

1.4 Conclusion

Dans ce chapitre, nous avons introduit le concept de "raisonnement à partir des derniers conflits". Celui-ci peut être intégré à n'importe quel algorithme de recherche basé sur une exploration en profondeur d'abord. Le principe est de sélectionner en priorité la variable impliquée

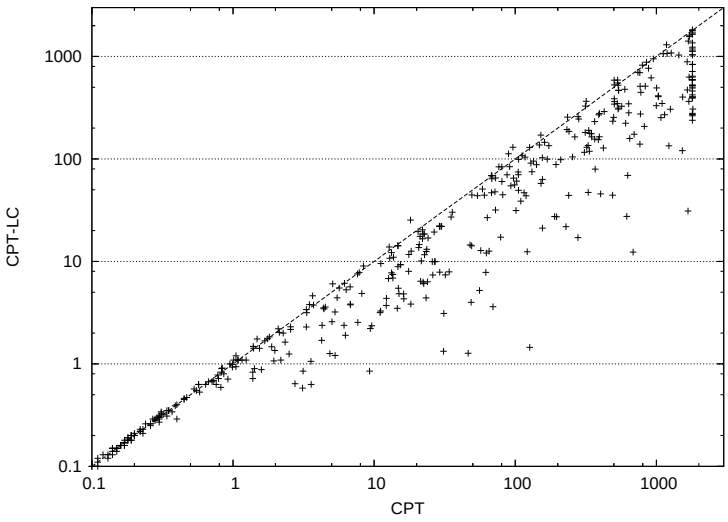


FIG. 1.5 – domaine BlocksWorld

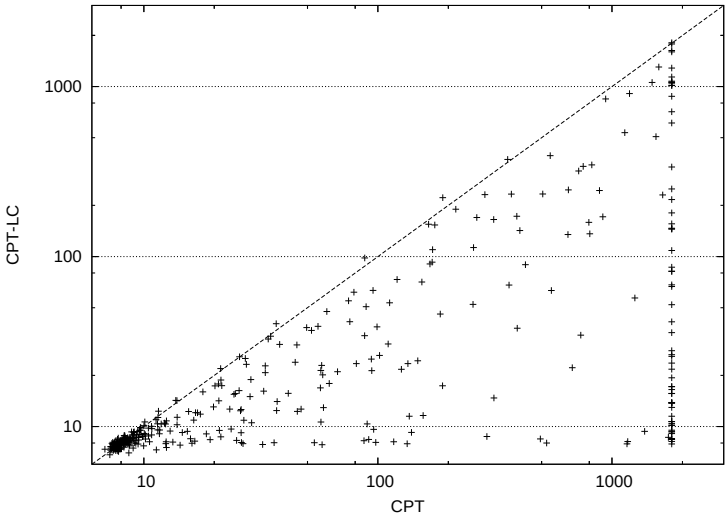


FIG. 1.6 – domaine Depots

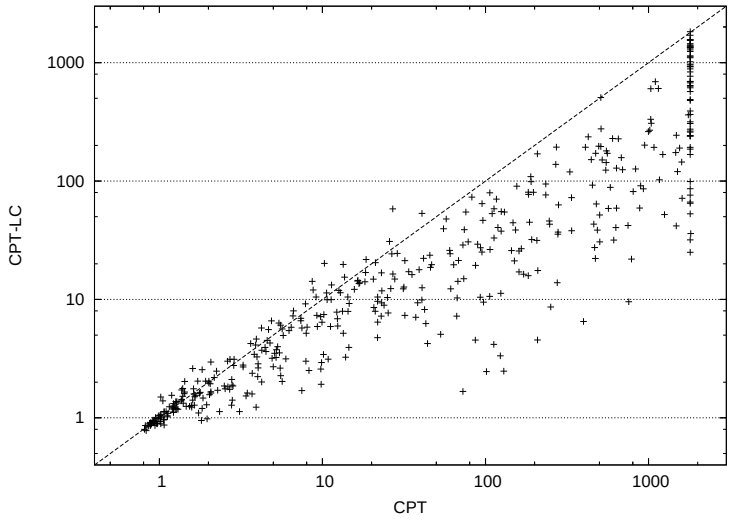


FIG. 1.7 – domaine DriverLog

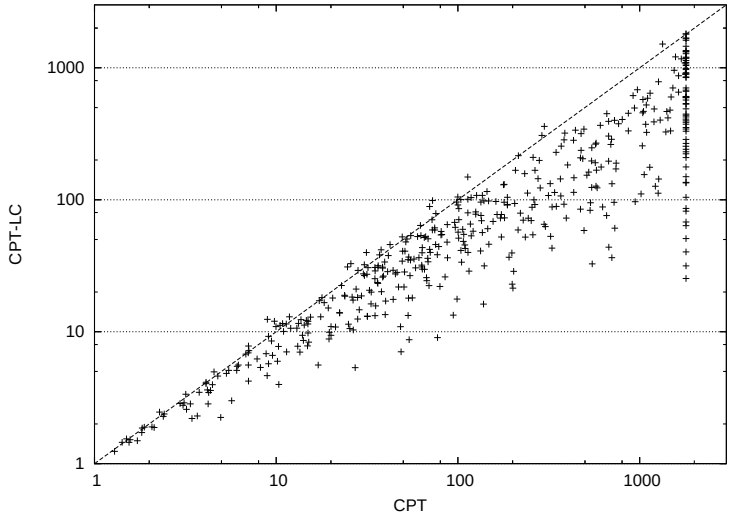


FIG. 1.8 – domaine Logistics

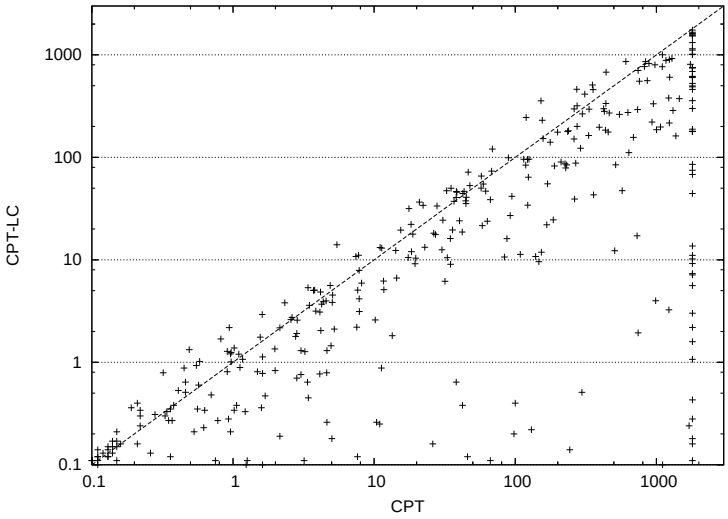


FIG. 1.9 – domaine Rovers

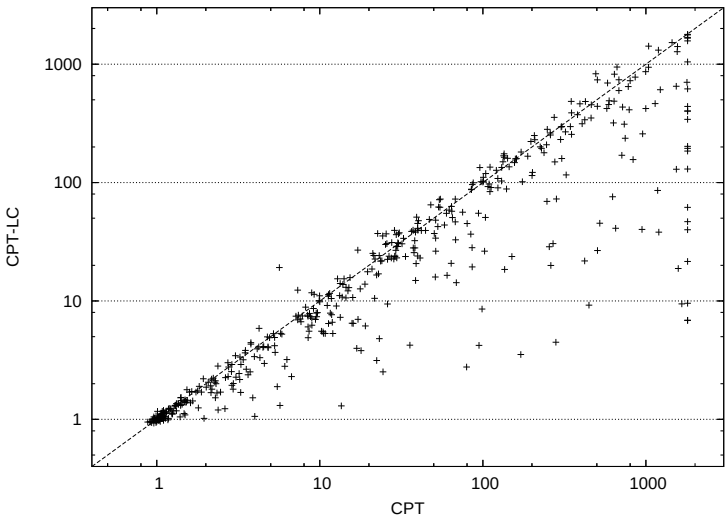


FIG. 1.10 – domaine Satellite

dans le dernier conflit (i.e. la dernière assignation qui a provoqué un échec) tant que le réseau de contraintes ne peut être rendu consistant. Ce schéma de raisonnement permet notamment de prévenir en partie l'apparition du phénomène de thrashing en effectuant une succession de retours-arrière sur la plus récente décision coupable du dernier conflit. Cette technique simule alors en quelque sorte les sauts effectués dans l'arbre de recherche par une technique de retours-arrière intelligents. Notons que ceci peut être fait sans coût additionnel en espace et avec une complexité temporelle dans le pire des cas en $O(end^3)$. De nombreuses expérimentations ont démontré l'intérêt du raisonnement à partir des derniers conflits. La généralisation du raisonnement à partir des derniers conflits permet l'identification de nogoods sous la forme d'un ensemble de variables ne pouvant être successivement assignées sans provoquer d'échec (i.e. un ensemble de variables ne pouvant être traversées). Comme cela a été mis en évidence sur le problème des "reines et pions", la généralisation de ce raisonnement permet éventuellement d'identifier des sous-ensembles de variables difficiles et d'accélérer ainsi la résolution de ces problèmes.

Dans cette approche, l'heuristique de choix de variable est violée jusqu'à ce qu'un retour-arrière soit effectué sur la décision coupable et qu'une valeur puisse être assignée à la précédente variable en échec. Cependant l'une des alternatives envisageables consiste à ne plus assigner la variable impliquée dans le dernier conflit mais juste de vérifier si celle-ci possède dans son domaine une valeur ne provoquant pas directement un échec. Dans ce cas, cette approche devient une technique d'inférence pure qui correspond (partiellement) à maintenir une singleton consistance (SAC, par exemple) sur la variable impliquée dans le dernier conflit. Ceci est alors à mettre en relation avec la technique de "quick shaving" proposée dans [Lhomme, 2005]. D'un autre coté, comme ce type de raisonnement permet d'obtenir un "effet backjumping" (c'est-à-dire simuler les effets d'une technique de retours-arrière intelligents) il serait intéressant de pouvoir comparer plus précisément le raisonnement à partir des derniers conflits aussi bien théoriquement que pratiquement avec des approches comme la technique de retours-arrière à la Gaschnig ou CBJ.

2

Enregistrement de nogoods à partir des redémarrages

Le phénomène de thrashing consiste à explorer de façon répétitive les mêmes sous-arbres, et donc à redécouvrir les mêmes impasses plusieurs fois au cours de la recherche. Une technique (décrite en section 1.3) permettant d'éviter le phénomène de thrashing consiste à effectuer des redémarrages pour éviter les exécutions défavorables du type longues traînées. On introduit généralement un caractère aléatoire au sein de l'heuristique employée afin de diriger la recherche vers une autre partie de l'espace de recherche dans laquelle il est éventuellement plus facile de trouver une solution. Cet aléa permet notamment de départager des variables (ou des valeurs) jugées équivalentes par l'heuristique. Cependant il est tout à fait envisageable d'explorer plusieurs fois une même portion de l'espace de recherche puisque traditionnellement, aucune information relative à l'état d'avancement de la recherche n'est sauvegardée d'une exécution à l'autre. Explorer à nouveau le même sous-espace de recherche constitue alors un effort inutile (qui peut même se répéter plusieurs fois) puisqu'il a été prouvé dans une exécution précédente que cette portion de l'espace de recherche ne contenait aucune solution.

Les nogoods (c.f. section 2.2) peuvent être utilisés pour élaguer des branches de l'arbre de recherche, déjà explorées, et dans lesquelles aucune solution ne peut être rencontrée. Nous proposons ici d'éviter que les mêmes situations ne puissent apparaître d'une exécution à une autre en enregistrant un ensemble de nogoods éliminant les parties de l'arbre de recherche déjà explorées pour les exécutions suivantes [Lecoutre *et al.*, 2007c, Lecoutre *et al.*, 2007d]. Autrement dit les portions de l'arbre de recherche déjà explorées sont délimitées par un ensemble de nogoods, ce qui garantit qu'un même espace de recherche ne sera pas exploré deux fois de suite. Plus précisément, les nogoods sont enregistrés lorsque la valeur de coupure est atteinte, i.e. avant de redémarrer l'algorithme de recherche. Cet ensemble de nogoods est extrait à partir de la dernière branche courante de l'arbre de recherche et exploité en utilisant la structure des *watched literals* initialement proposé pour le problème SAT. De façon générale, les nogoods sont exploités en introduisant en quelque sorte une contrainte globale avec un algorithme de filtrage dédié qui exploite les watched literals.

2.1 Définitions

On considère un arbre de recherche construit par un algorithme de recherche avec retours-arrière (e.g. MAC), basé sur un schéma de branchement binaire. Les décisions positives sont prises en premier et l'algorithme maintient une consistance ϕ (e.g. la consistance d'arc généralisée) à

chaque nœud.

Chaque branche de l'arbre de recherche peut être vue comme une séquence de décisions positives et négatives. Autrement dit, à chaque nœud de l'arbre de recherche, on peut associer une séquence de décisions menant jusqu'à lui. Etant donné une séquence de décisions Σ, l'ensemble des décisions négatives de Σ est noté $neg(\Sigma)$ et l'ensemble des décisions positives de Σ est noté $pos(\Sigma)$.

Une séquence de décisions terminée par une décision négative est appelée *nld-sous-séquence* (pour Negative Last-Decision sous-séquence).

Définition 28 (nld-sous-séquence)**.** *Soit $\Sigma = \langle \delta_1, \ldots, \delta_i, \ldots, \delta_m \rangle$ une séquence de décisions. La séquence $\langle \delta_1, \ldots, \delta_i \rangle$, où δ_i est une décision négative, est appelée une nld-sous-séquence de Σ. Les ensembles de décisions positives et négatives de Σ sont notés respectivement $pos(\Sigma)$ et $neg(\Sigma)$.*

Nous rappelons qu'un ensemble de décisions Δ est un nogood d'un réseau P, si le réseau P restreint aux décisions de Δ (noté $P|_\Delta$) est insatisfaisable. A partir de n'importe quelle branche de l'arbre de recherche, de la racine à une feuille, un nogood peut être extrait de chaque décision négative rencontrée sur la branche (également mentionné dans [Puget, 2005]). Ceci est établi par la proposition suivante :

Proposition 8. *Soient P un réseau de contraintes et Σ une séquence de décisions prises le long d'une branche de l'arbre de recherche. Pour toute nld-sous-séquence $\langle \delta_1, \ldots, \delta_i \rangle$ de Σ, l'ensemble $\Delta = \{\delta_1, \ldots, \neg\delta_i\}$ est un nogood de P (appelé nld-nogood)*[3]*.*

Démonstration. Comme les décisions positives sont prises en premier, lorsque la décision négative δ_i est rencontrée, le sous-arbre correspondant à la décision opposée $\neg\delta_i$ a déjà été réfuté et le sous-arbre démontré inconsistant. □

Ces nogoods contiennent à la fois des décisions positives et négatives. Ils correspondent alors à la définition des nogoods généralisés définis en section 2.2. Nous montrons que les nld-nogoods peuvent être réduits en taille en ne considérant que les décisions positives. Ils correspondent alors à la définition des nogoods standard. Pour cela la résolution propositionnelle peut être étendue afin d'être appliquée directement sur des nogoods, e.g. appelée "Constraint Resolution" (C-Res en raccourci) dans [Mitchell, 2003]. Il est important de noter que la C-Res peut être appliquée de manière sûre puisque dans l'arbre de recherche, deux décisions opposées, e.g. $X = a$ et $X \neq a$ ne peuvent pas apparaître sur la même branche et donc par conséquent dans un même nogood.

Définition 29 (C-Res (Constraint Resolution))**.** *Soient P un réseau de contraintes, et $\Delta_1 = \Gamma \cup \{X = a\}$ et $\Delta_2 = \Lambda \cup \{X \neq a\}$ deux nogoods de P. Nous définissons la Constraint Resolution comme C-Res$(\Delta_1, \Delta_2) = \Gamma \cup \Lambda$.*

Il est immédiat que C-Res(Δ_1, Δ_2) est un nogood de P.

Proposition 9. *Soient P un réseau de contraintes et Σ une séquence de décisions prises le long d'une branche de l'arbre de recherche. Pour toute nld-sous-séquence $\Sigma' = \langle \delta_1, \ldots, \delta_i \rangle$ de Σ, l'ensemble $\Delta = pos(\Sigma') \cup \{\neg\delta_i\}$ est un nogood de P (appelé nld-nogood réduit).*

Démonstration. $\Sigma' = \langle \delta_1, \ldots, \delta_i \rangle$ est une nld-sous-séquence, donc elle contient $k \geq 1$ décisions négatives, notées $\delta_{g_1}, \ldots, \delta_{g_k}$, dans l'ordre dans lequel elles apparaissent dans Σ. On peut remarquer que nous avons $\delta_{g_k} = \delta_i$.
Montrons par récurrence que $\forall j \in [1, k]$, l'hypothèse suivante $H(j)$ est vraie :

[3]La notation $\{\delta_1, \ldots, \neg\delta_i\}$ correspond à $\{\delta_j \in \Sigma \mid j < i\} \cup \{\neg\delta_i\}$ réduit à $\{\neg\delta_1\}$ quand $i = 1$.

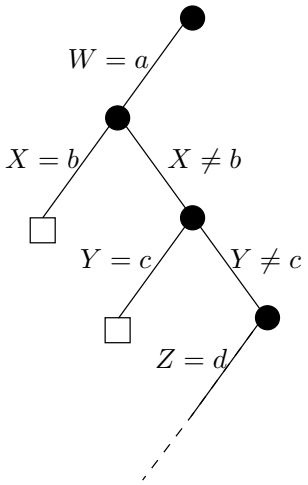
$$\Delta'_j = pos(\Sigma') \cup \{\delta_{g_1}, \ldots, \delta_{g_{j-1}}\} \cup \{\neg\delta_i\} \text{ est un nogood de } P.$$

Tout d'abord, montrons que $H(k)$ est vraie. A partir de la proposition 8, nous savons que $\Delta'_k = pos(\Sigma') \cup \{\delta_{g_1}, \ldots, \delta_{g_{k-1}}\} \cup \{\neg\delta_i\}$ est un nogood puisque Δ'_k est le nogood correspondant à la nld-sous-séquence Σ'.
Ensuite, nous savons que si $H(j+1)$ est vraie, alors $H(j)$ est vraie également. Par hypothèse, Δ'_{j+1} est un nogood de P. Soit Σ_j la nld-sous-séquence correspondant au préfixe de Σ' tel que δ_{g_j} soit sa dernière décision (négative)(nous avons $pos(\Sigma_j) \subseteq pos(\Sigma')$ et $neg(\Sigma_j) = \{\delta_{g_1}, \ldots, \delta_{g_j}\}$), et Δ_j est son nogood correspondant d'après la proposition 8. En appliquant la Constraint Resolution, nous obtenons :

$$\begin{array}{rcl}
\Delta'_{j+1} & = & pos(\Sigma') \cup \{\delta_{g_1}, \ldots, \delta_{g_{j-1}}, \delta_{g_j}\} \cup \{\neg\delta_i\} \\
\Delta_j & = & pos(\Sigma_j) \cup \{\delta_{g_1}, \ldots, \delta_{g_{j-1}}\} \cup \{\neg\delta_{g_j}\} \\
\text{C-Res}(\Delta'_{j+1}, \Delta_j) & = & pos(\Sigma') \cup pos(\Sigma_j) \cup \{\delta_{g_1}, \ldots, \delta_{g_{j-1}}\} \cup \{\neg\delta_i\} \\
& = & pos(\Sigma') \cup \{\delta_{g_1}, \ldots, \delta_{g_{j-1}}\} \cup \{\neg\delta_i\}
\end{array}$$

Comme $pos(\Sigma_j) \subseteq pos(\Sigma')$, $\neg\delta_{g_j} \in \Delta_j$ et $\delta_{g_j} \in \Delta'_{j+1}$. $\Delta'_j = \text{C-Res}(\Delta'_{j+1}, \Delta_j)$ est alors un nogood de P. Par conséquent, nous avons prouvé que $H(j)$ est vraie.
Pour $j = 1$, nous obtenons que $\Delta = pos(\Sigma') \cup \{\neg\delta_i\}$ est un nogood de P. □

A titre d'illustration, soit $\Sigma = \langle W = a, X \neq b, Y \neq c, Z = d\rangle$ une séquence de décisions prises le long d'une branche d'un arbre de recherche. D'après les définitions et propositions précédentes nous avons alors :



- $\Sigma' = \langle W = a, X \neq b, Y \neq c\rangle$ est une nld-sous-séquence ;
- $\Delta_1 = \langle W = a, X \neq b, Y = c\rangle$ est un nld-nogood ;
- $\Delta_2 = \langle W = a, Y = c\rangle$ est un nld-nogood réduit.

Remarquons que l'espace requis pour enregistrer tous les nogoods correspondant à n'importe quelle branche de l'arbre de recherche est polynomial par rapport au nombre de variables et à la taille du plus grand domaine.

Proposition 10. *Soient P un réseau de contraintes et Σ une séquence de décisions prises le long d'une branche de l'arbre de recherche. La complexité spatiale dans le pire des cas pour enregistrer tous les nld-nogoods de Σ est $O(n^2d^2)$ tandis que la complexité spatiale dans le pire des cas pour enregistrer tous les nld-nogoods réduits de Σ est $O(n^2d)$.*

Démonstration. Tout d'abord, le nombre de décisions négatives pouvant apparaître dans n'importe quelle branche est $O(nd)$. Pour chaque décision négative, on peut extraire un nld-nogood (réduit). Comme la taille de n'importe quel nld-nogood (respectivement réduit) est $O(nd)$ (respectivement $O(n)$ puisque celui-ci ne contient que des décisions positives), nous obtenons une complexité spatiale globale de $O(n^2d^2)$ (respectivement $O(n^2d)$). □

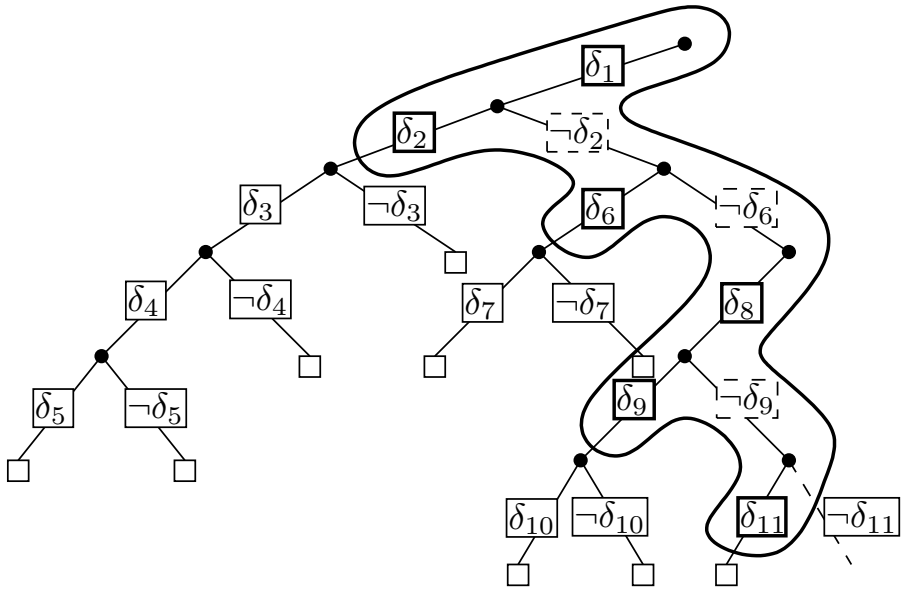


FIG. 2.1 – Arbre partiel de recherche et nld-nogoods

Les nld-nogoods réduits extraits d'une branche ont une meilleure capacité d'élagage que les nld-nogoods extraits de la même branche puisque pour chaque nld-sous-séquence, le nld-nogood correspondant est subsumé par le nld-nogood réduit.

2.2 Enregistrer des nogoods à partir des redémarrages

Il a été montré [Gomes *et al.*, 2000] que le temps de résolution nécessaire à un algorithme de recherche randomisé peut quelque fois être caractérisé par une longue traînée avec même des moments infinis. Pour certaines instances, ce phénomène heavy-tailed peut être évité en utilisant les redémarrages et en introduisant un aléa, i.e. en redémarrant plusieurs fois l'algorithme de recherche tandis que l'introduction d'un aléa est utilisée par l'heuristique de recherche (c'est le phénomène de diversification de la recherche). Dans le domaine de la satisfaction de contraintes, les redémarrages ont été montrés assez efficaces pour résoudre certaines séries de problèmes. Cependant les performances moyennes des solveurs peuvent être dégradées lorsqu'aucune technique d'apprentissage n'est utilisée (c.f. les résultats expérimentaux en section 2.4).

L'enregistrement de nogoods n'est pas une technique encore réellement adoptée par la communauté CSP. De plus c'est une technique qui peut mener à une complexité spatiale exponentielle si le nombre de nogoods n'est pas contrôlé. L'approche présentée ici propose une réponse à cette problématique en combinant l'enregistrement de nogoods et les redémarrages : des nld-nogoods réduits sont extraits (et enregistrés dans une base de nogoods) à partir de la dernière branche de l'arbre de recherche, à la fin de chaque exécution [Lecoutre *et al.*, 2007c]. Cela revient à délimiter à l'aide d'un ensemble de nogoods l'espace de recherche déjà parcouru. Notre objectif est de bénéficier à la fois de la puissance des redémarrages et des possibilités d'apprentissage sans pour autant sacrifier les performances globales du solveur et la complexité spatiale.

La figure 2.1 décrit un arbre partiel de recherche exploré quand le solveur est sur le point de redémarrer. Les décisions positives étant prises en premier, δ_i (respectivement $\neg\delta_i$) correspond à une décision positive (respectivement négative). Sur cet exemple, la recherche est arrêtée après

avoir réfuté la décision δ_{11} et pris la décision $\neg\delta_{11}$. Les nld-nogoods de P correspondant à la branche courante sont alors les suivants :

- $\Delta_1 = \{\delta_1, \neg\delta_2, \neg\delta_6, \delta_8, \neg\delta_9, \delta_{11}\}$;
- $\Delta_2 = \{\delta_1, \neg\delta_2, \neg\delta_6, \delta_8, \delta_9\}$;
- $\Delta_3 = \{\delta_1, \neg\delta_2, \delta_6\}$;
- $\Delta_4 = \{\delta_1, \delta_2\}$.

Le premier nld-nogood réduit est obtenu en appliquant la Constraint Resolution comme suit :

$$\begin{aligned}\Delta'_1 &= \text{C-Res}(\text{C-Res}(\text{C-Res}(\Delta_1, \Delta_2), \Delta_3), \Delta_4)\\ &= \text{C-Res}(\text{C-Res}(\{\delta_1, \neg\delta_2, \neg\delta_6, \delta_8, \delta_{11}\}, \Delta_3), \Delta_4)\\ &= \text{C-Res}(\{\delta_1, \neg\delta_2, \delta_8, \delta_{11}\}, \Delta_4)\\ &= \{\delta_1, \delta_8, \delta_{11}\}\end{aligned}$$

Un processus similaire peut être appliqué aux autres nld-nogoods et nous obtenons alors l'ensemble des nld-nogoods réduits suivant :

$$\begin{aligned}\Delta'_2 &= \text{C-Res}(\text{C-Res}(\Delta_2, \Delta_3), \Delta_4) = \{\delta_1, \delta_8, \delta_9\}\\ \Delta'_3 &= \text{C-Res}(\Delta_3, \Delta_4) = \{\delta_1, \delta_6\}\\ \Delta'_4 &= \Delta_4 = \{\delta_1, \delta_2\}\end{aligned}$$

Pour éviter l'exploration des mêmes parties de l'espace de recherche au cours des exécutions suivantes, on peut exploiter les nogoods enregistrés. En effet il suffit de contrôler que l'ensemble des décisions de la branche courante ne contient pas toutes les décisions d'un nogood enregistré. De plus, la négation de la dernière décision indéterminée de n'importe quel nogood peut être déduite. Ce principe est décrit plus précisément en section suivante. Par exemple, à partir du moment ou la décision δ_1 devient vrai, on peut inférer $\neg\delta_2$ à partir du nogood Δ'_4 et $\neg\delta_6$ à partir du nogood Δ'_3. En effet après avoir pris la décision δ_1, il ne reste plus qu'une seule décision indéterminée dans les nogoods Δ'_4 et Δ'_3, qui peut alors être inférée.

Finalement il est important de souligner que les *nld-nogoods réduits extraits de la dernière branche subsument tous les nld-nogoods réduits qui auraient pu être extraits de n'importe quelle branche précédemment explorée.* Ceci vient du fait que chaque sous-arbre complètement exploré (et ainsi, tous les nld-nogoods qui auraient pu être construits à partir de toutes les branches de ce sous-arbre) est préfixé par au moins un nld-nogood de la dernière branche. Remarquons que cette approche est à mettre en relation avec des travaux effectués dans le domaine de la satisfaisabilité booléenne [Baptista *et al.*, 2001, Fukunaga, 2003], dans lesquels l'objectif est d'obtenir une stratégie de redémarrage complète tout en réduisant le nombre de nogoods enregistrés.

2.3 Exploiter les nogoods

Les nld-nogoods réduits peuvent être efficacement exploités en utilisant la technique des watched literals introduite pour SAT [Moskewicz *et al.*, 2001, Zhang et Malik, 2002, Eén et Sorensson, 2003]. Un nld-nogood réduit correspond à la négation d'ensembles (conjonctions) de décisions positives qui seront enregistrés sous la forme de disjonctions de décisions négatives. Ceux-ci peuvent être considérés comme de nouvelles contraintes du problème à satisfaire. Dans cette section, nous présentons un algorithme de propagation efficace maintenant GAC sur l'ensemble des nld-nogoods réduits appris au cours des exécutions précédentes. Ces nld-nogoods réduits peuvent être considérés collectivement comme une contrainte globale.

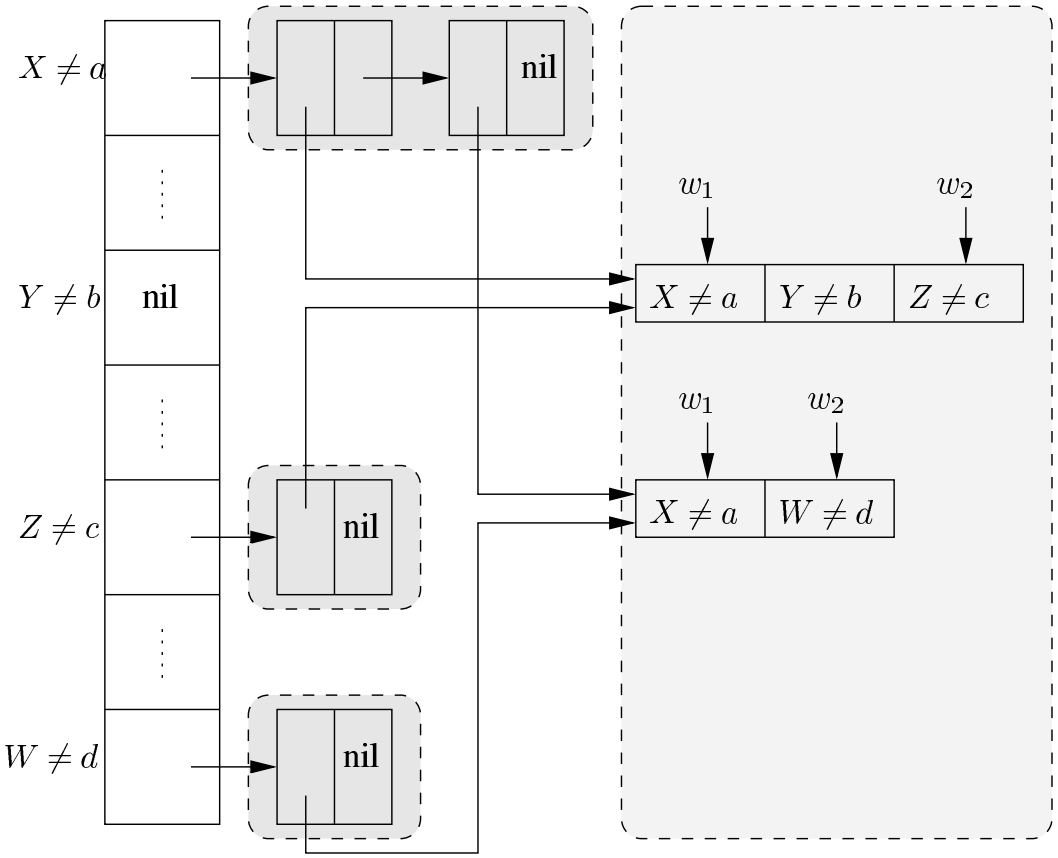


FIG. 2.2 – Vue partielle d'une base de nogood $\mathscr{B}$

2.3.1 La base de nogoods et les watched literals

Les nld-nogoods réduits qui sont extraits de la dernière branche ne contiennent que des décisions positives et sont enregistrés dans une base de nogoods $\mathscr{B}$. Pour les exploiter, il suffit, à chaque fois qu'une décision est prise pendant la recherche, de contrôler si l'ensemble des décisions courantes (i.e. l'ensemble des décisions correspondant à la séquence de décisions de la branche courante) est compatible avec tous les nogoods de $\mathscr{B}$.

Parcourir tous les nogoods de $\mathscr{B}$ à chaque étape de la recherche peut être relativement coûteux (du fait du nombre de nogoods) et n'est pas envisageable en pratique. Pour fournir un accès efficace à ces nogoods, nous utilisons la structure de données paresseuse des watched literals qui est illustrée par la figure 2.2. Le principe est de sélectionner deux décisions par nogood qui nous garantissent que le nogood n'est pas violé et qu'aucune inférence ne peut être effectuée. On peut remarquer qu'une seule décision marquée (ou watchée) est suffisante pour garantir qu'un nogood donné n'est pas violé. La seconde décision marquée est utilisée pour effectuer de l'inférence. Ces deux décisions particulières sont appelées littéraux marqués et sont référencées par w_1 et w_2 sur la figure 2.2. Tant que les deux littéraux marqués ne sont pas falsifiés, aucune inférence n'est possible. On peut noter que dans notre cas, les littéraux marqués correspondent à des décisions négatives puisque les nld-nogoods réduits ne contiennent que des décisions positives et que chaque nogood est représenté par une disjonction de décisions négatives. Lorsqu'une décision $X = a$ est effectuée, chaque nogood, où $X \neq a$ est un littéral marqué, peut éventuellement être violé. Il

faut alors déterminer si une autre décision valide peut être trouvée pour garantir que le nogood ne sera pas violé. Si tel est le cas, cette décision devient le nouveau littéral marqué (remplaçant $X \neq a$), et dans le cas contraire, la deuxième décision marquée doit être inférée pour éviter que le nogood ne soit violé.

En pratique, quand une décision $X = a$ est prise, seuls les nogoods où $X \neq a$ apparaît comme décision marquée sont vérifiés. Pour chaque décision négative, nous maintenons la liste des nogoods qui contiennent cette décision comme littéral marqué. Une structure de données supplémentaire est donc nécessaire. Il faut un tableau de nd entrées. Chaque entrée correspond à une décision négative δ et représente la tête d'une liste chaînée permettant l'accès aux nogoods de la base $\mathscr{B}$ contenant δ comme littéral marqué. L'accès à une telle liste, notée $\mathscr{B}_\delta$, s'effectue en temps constant. Chaque nogood est représenté par un tableau d'au plus n décisions négatives auquel on associe deux littéraux marqués. Sur la figure 2.2, la base de nogoods contient deux nogoods enregistrés. Le premier est marqué par les décisions $X \neq a$ et $Z \neq c$ tandis que le second est marqué par les décisions $X \neq a$ et $W \neq d$.

2.3.2 L'extraction et l'enregistrement des nld-nogoods réduits

Les nld-nogoods réduits sont extraits à partir de la branche courante de l'arbre de recherche lorsque l'exécution courante est sur le point de s'arrêter. Pour cela la fonction *storeNogoods* (voir l'algorithme 17) extrait et enregistre les nld-nogoods réduits. Les paramètres de cette fonction sont la séquence de décisions correspondant à la branche courante de la racine à la feuille de l'arbre de recherche et la base de nogoods. Comme décrit en section 2.1 on peut extraire un nld-nogood réduit à partir de chaque décision négative apparaissant dans la séquence de décisions de la branche courante. L'extraction des nld-nogoods réduits s'effectue itérativement en parcourant toutes les décisions (à partir de la racine jusqu'à la feuille) de la séquence correspondant à la branche courante. De la racine à la feuille de la branche courante, lorsqu'une décision positive est rencontrée, sa négation est enregistrée, tandis que lorsqu'une décision négative est rencontrée, on construit un nogood Δ' à partir de cette décision et de toutes celles précédemment enregistrées dans l'ensemble Δ (ligne 9). Comme chaque nogood est enregistré sous la forme d'une disjonction de décisions négatives, on enregistre la négation de la décision (positive) à la ligne 4. Si le nogood

Algorithme 17 : *storeNogoods*

Entrées : Σ : Séquence de décisions, $\mathscr{B}$: Base de nogoods

```
1  Δ ← ∅ ;
2  pour chaque décision δ ∈ Σ ordonnée de la première décision de Σ à la dernière faire
3      si δ est une décision positive alors
4          Δ ← Δ ∪ {¬δ} ;
5      sinon
6          si Δ = ∅ alors
7              supprimer a de dom(X) où δ = (X ≠ a), pour toutes les exécutions suivantes;
8          sinon
9              Δ' ← Δ ∪ {δ} ;
10             B ← B ∪ {Δ'} ;
11         fin
12     fin
13 fin
```

est de taille 1, il peut directement être exploité en réduisant le domaine de la variable impliquée (ligne 7). Dans le cas contraire, le nogood est enregistré dans la base $\mathscr{B}$ (ligne 10).

A chaque fois qu'un nogood est enregistré, deux décisions sont marquées. Il est important de remarquer que n'importe quelle décision de Δ peut être marquée puisque l'algorithme de recherche est sur le point de redémarrer. Ainsi les deux décisions sélectionnées seront valides au début de l'exécution suivante. Pour chaque décision sélectionnée, une nouvelle entrée (pour le nouveau nogood) est insérée dans la liste des nogoods marqués par cette décision.

2.3.3 L'exploitation des nogoods

De façon générale, les nogoods sont exploités en introduisant en quelque sorte une contrainte globale avec un algorithme de filtrage dédié qui exploite la structure des watched literals. De manière intéressante, cet algorithme qui nous permet d'établir la consistance d'arc généralisée (GAC) sur la base des nogoods peut facilement être intégré à n'importe quelle technique de propagation de contraintes (e.g. [Schulte et Carlsson, 2006]) mais également à n'importe quel algorithme générique établissant la consistance d'arc généralisée.

Les inférences sont réalisées à l'aide des nld-nogoods réduits lorsqu'on établit (maintient) la consistance d'arc généralisée. Ce mécanisme est intégré ici au sein d'un algorithme à gros grain établissant GAC, i.e un algorithme qui fonctionne sur la base de couples de la forme (C, X) avec X représentant une variable impliquée dans la contrainte C. L'algorithme 18 peut être appliqué à tout réseau de contraintes, impliquant des contraintes d'arité quelconque, afin d'établir GAC. Au cours de la phase de prétraitement, *propagate* doit être appelé avec l'ensemble S des variables du réseau tandis que durant la recherche, S ne contient que la variable impliquée dans la dernière décision positive ou négative. A tout moment, le principe est de regrouper dans Q toutes les variables dont les domaines ont été réduits par propagation.

Initialement, Q contient toutes les variables de l'ensemble S passé en paramètre (ligne 1). Puis, de façon itérative, chaque variable X de Q est sélectionnée (ligne 3). Si le domaine de X est un singleton $\{a\}$ (ligne 4 à 11), les nogoods précédemment enregistrés peuvent être exploités en contrôlant la consistance de la base de nogoods. Cette opération est effectuée par la fonction $inferences$ (décrite ci-dessous) qui itère tous les nogoods impliquant $X \neq a$ comme littéral marqué et retourne un ensemble d'inférences déduit des nogoods. Cet ensemble est alors pris en compte : pour chaque inférence $Y \neq b$ de l'ensemble retourné par la fonction $inferences(X \neq a)$, si b fait partie du domaine de la variable Y, alors b est simplement supprimée du domaine courant de cette variable ce qui peut mener à une inconsistance ou à la mise à jour de l'ensemble Q.

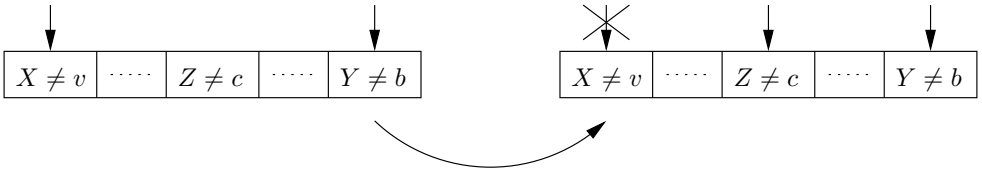
Le reste de l'algorithme (ligne 12 à 14) correspond à la partie centrale (c.f. section 1.2.1) d'un algorithme générique à gros grain maintenant GAC. Pour chaque contrainte C impliquant X, on effectue la révision de tous les arcs (C, Y) où $Y \neq X$. Une révision est effectuée par l'appel à la fonction *revise*. Cette fonction est spécifique à l'algorithme à gros grain de consistance d'arc choisi, et provoque la suppression de valeurs qui deviennent inconsistantes par rapport à C. Lorsque la révision d'un arc (C, Y) provoque la suppression de certaines valeurs du domaine de Y, la fonction *revise* renvoie *vrai* et la variable Y est insérée dans Q. L'algorithme boucle jusqu'à atteindre un point fixe.

Le principe de l'algorithme 19 est d'itérer la liste des nogoods impliquant la décision passée en paramètre comme littéral marqué. Pour chaque nogood Δ, à chaque tour de la boucle principale, comme la décision $X \neq a$ n'est plus vraie (puisque la variable X est réduite à la valeur singleton a), celle-ci ne peut plus être marquée et on recherche alors un nouveau littéral à marquer. Cette recherche est effectuée par la fonction $canFindAnotherWatch$. Si une nouvelle décision marquée ne peut être trouvée (ligne 5), la deuxième décision marquée ($Y \neq b$) est alors inférée. La figure

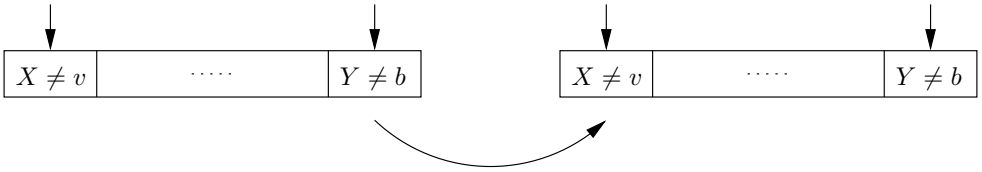
FIG. 2.3 – inferences($X \neq a$: décision) : Ensemble de décisions

$I \leftarrow \emptyset$
pour chaque nogood $\Delta \mid (X \neq v)$ est marqué dans Δ

si une autre décision peut être marquée **alors**



sinon



Inference : $I \leftarrow I \cup \{Y \neq b\}$;
fin
fin
Retourner(I) ;

2.3 illustre le fonctionnement de l'algorithme 19. Pour chaque nogood marqué par la décision $(X \neq a)$, soit une nouvelle décision peut être sélectionnée et dans ce cas on effectue la mise à jour des littéraux marqués, soit aucune autre décision ne peut être sélectionnée et on infère le second littéral marqué. Lorsque tous les nogoods marqués par la décision $(X \neq a)$ ont été contrôlés, l'algorithme retourne l'ensemble des inférences collectées dans la base de nogoods.

Finalement la fonction *canFindAnotherWatch* (algorithme 20) examine successivement toutes les décisions (qui ne sont pas marquées) d'un nogood donné dans le but de trouver une nouvelle décision à sélectionner (ligne 3). Une telle décision est soit déjà satisfaite, soit non-assignée (ligne 2).

Même si ce n'est pas décrit ici, il est important de noter que lorsqu'un nouveau littéral marqué a été trouvé, l'entrée (correspondante au nogood Δ) doit être supprimée de la liste des nogoods impliquant $X \neq a$ comme littéral marqué. De plus, on doit mettre à jour (i.e. ajouter une nouvelle entrée) la liste des nogoods impliquant $Y \neq b$ comme nouveau littéral marqué.

Limitation de l'algorithme de filtrage dédié aux nogoods L'algorithme de filtrage dédié à l'exploitation des nogoods présenté précédemment permet d'établir la consistance d'arc généralisée sur chaque nogood de la base. Autrement dit, l'algorithme garantit que chaque nogood de la base, pris individuellement, est bien GAC. Même si on peut assimiler les nogoods à de nouvelles contraintes d'un problème donné, deux nogoods impliquant le même ensemble de variables sont considérés comme deux contraintes différentes et non pas comme une seule contrainte avec deux

Algorithme 18 : *propagate*

Entrées : S : Ensemble de variables
Sorties : Booléen

```
1  Q ← S ;
2  tant que Q ≠ ∅ faire
3  |  sélectionner et supprimer X de Q ;
4  |  si | dom(X) | = 1 alors
5  |  |  Soit dom(X) = {a} ;
6  |  |  pour chaque (Y ≠ b) ∈ inferences(X ≠ a) faire
7  |  |  |  dom(Y) ← dom(Y)\{b} ;
8  |  |  |  si dom(Y) = ∅ alors
9  |  |  |  |  retourner faux ;
10 |  |  |  sinon
11 |  |  |  |  Q ← Q ∪ {Y} ;
12 |  |  |  fin
13 |  |  fin
14 |  fin
15 |  pour chaque C ∈ 𝒞 | X ∈ scp(C) faire
16 |  |  pour chaque Y ∈ scp(C) | X ≠ Y faire
17 |  |  |  si revise(C,Y) alors
18 |  |  |  |  si dom(Y) = ∅ alors
19 |  |  |  |  |  retourner faux ;
20 |  |  |  |  sinon
21 |  |  |  |  |  Q ← Q ∪ {Y} ;
22 |  |  |  |  fin
23 |  |  |  fin
24 |  |  fin
25 |  fin
26 fin
27 retourner vrai ;
```

Algorithme 19 : *inferences*

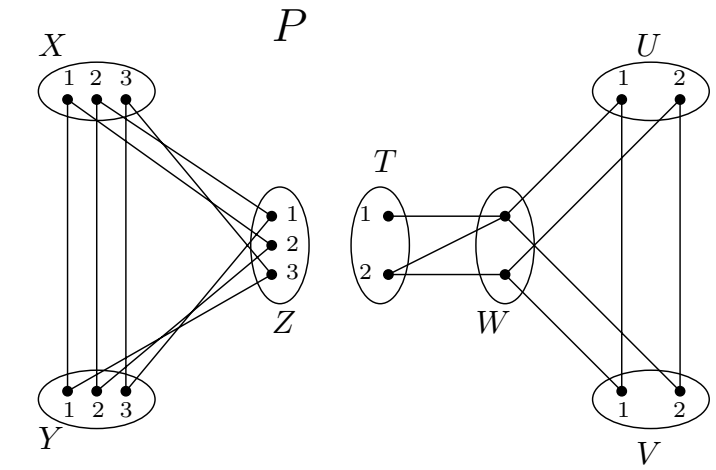
Entrées : $X \neq a$: Decision
Sorties : Ensemble de décisions

```
1  Γ ← ∅ ;
2  pour chaque nogood Δ ∈ ℬ_{X≠a} faire
3  |  Soit (Y ≠ b) la seconde décision watchée de Δ ;
4  |  si b ∈ dom(Y) alors
5  |  |  si ¬ canFindAnotherWatch(Δ, X ≠ a) alors
6  |  |  |  Γ ← Γ ∪ {Y ≠ b} ;
7  |  |  fin
8  |  fin
9  fin
10 retourner (Γ) ;
```

Algorithme 20 : *canFindAnotherWatch*

Entrées : Δ : Nogood, $X \neq a$: Décision
Sorties : Booléen

```
1 pour chaque décision (Y ≠ b) ∈ Δ | Y ≠ b n'est pas marquée dans Δ faire
2 |  si b ∉ dom(Y) ou | dom(Y) |> 1 alors
3 |  |  marquer Y ≠ b au lieu de X ≠ a dans Δ ;
4 |  |  retourner vrai ;
5 |  fin
6 fin
7 retourner faux ;
```



Exécution 1 Exécution 2

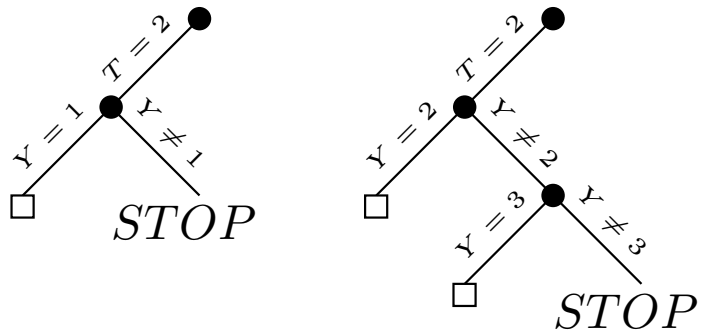


Fig. 2.4 – Nogoods et interconsistance

n-uplets interdits. Cette nuance restreint (un peu) la capacité de filtrage de l'algorithme dédié aux nogoods. Pour remédier à ce problème il faudrait appliquer une forme d'interconsistance (pairwise consistency) [Janssen *et al.*, 1989] restreinte aux nogoods de même portée, i.e. rendre les nogoods compatibles deux à deux.

La figure 2.4 met en évidence ce problème d'interconsistance entre les nogoods sur un exemple. Le réseau P dont la microstructure est représentée sur la figure est initialement arc-consistant. Celui-ci est composé de deux clusters (i.e. deux réseaux de contraintes non connectés mais regroupés dans un même problème), tous les deux insatisfaisables. L'algorithme de recherche construit un arbre de recherche binaire (les décisions positives étant prises en premier) et effectue des redémarrages en incrémentant de un le nombre de retours-arrière autorisés à chaque nouvelle exécution. Cette valeur de coupure est initialement fixée à un. Après les deux premières exécutions on obtient la base de nogood $\mathscr{B}$ suivante :

- $\delta_1 = \{T = 2 \wedge Y = 1\}$;
- $\delta_2 = \{T = 2 \wedge Y = 2\}$;
- $\delta_3 = \{T = 2 \wedge Y = 3\}$.

Evalués séparément, ces nogoods interdisent un couple de valeurs entre les variables T et Y. On peut les considérer comme trois contraintes différentes, sur lesquelles on établit GAC. Cependant, si l'on considère ces trois nogoods ensemble, on remarque que quelque soit la valeur affectée à la variable Y, l'assignation $T = 2$ est incompatible. On peut donc supprimer la valeur 2 du domaine de Y, ce qui rend le réseau arc-inconsistant. En effet la valeur 2 était un support universel pour la variable Y, la valeur restante dans le domaine de T force à assigner la valeur 1 à la variable W, ce qui rend le réseau de contraintes globalement inconsistant. L'inconsistance globale de ce réseau de contraintes n'aurait pas pu être détectée, si l'on considère les nogoods individuellement. On peut donc espérer obtenir une meilleure capacité de filtrage si l'on applique une forme d'interconsistance limitée. Notons cependant que la structure de données des watched literals n'est pas adaptée pour appliquer efficacement cette consistance.

2.3.4 Analyse de la complexité

Dans cette partie, nous discutons de la complexité des algorithmes décrits en section précédente pour extraire et exploiter les nogoods. Dans ce qui suit, $\mathscr{B}$ représente la base de nogood et $|\mathscr{B}|$ le nombre de nogoods de la base $\mathscr{B}$.

Proposition 11. *La complexité temporelle dans le pire des cas pour enregistrer les nld-nogoods réduits au moment d'un redémarrage (i.e. la complexité temporelle dans le pire des cas de la fonction storeNogoods) est de $O(n^2d)$.*

Démonstration. Tout d'abord, chaque nogood Δ ajouté à la base $\mathscr{B}$ (ligne 10 de l'algorithme 17) est composé d'au plus $|pos(\Sigma)|$ décisions, et au plus $|neg(\Sigma)|$ nogoods peuvent être extraits à partir de Σ. De plus, on peut observer que la complexité de la fonction *storeNogoods* dans le pire des cas est de $O(|pos(\Sigma)|.|neg(\Sigma)|)$. Comme $|pos(\Sigma)|$ est en $O(n)$ et $|neg(\Sigma)|$ est en $O(nd)$, on obtient une complexité globale en $O(n^2d)$. □

Proposition 12. *La complexité temporelle dans le pire des cas pour exploiter les nld-nogoods réduits à chaque nœud de la recherche, i.e. la complexité temporelle cumulée de la fonction inferences pour un même appel à la fonction propagate est en $O(n|\mathscr{B}|)$.*

Schéma de preuve. Tout d'abord, il est important de noter que lorsqu'une décision $X \neq a$ (éventuellement watchée) n'est plus valide (i.e. a est la seule valeur restante dans le domaine de X), alors celle-ci ne peut plus être watchée à nouveau (durant un même appel à la fonction *propagate*).

De plus, lorsqu'on recherche une nouvelle décision à sélectionner dans un nld-nogood réduit (i.e. un ensemble de décisions négatives), on peut itérer les décisions de ce nogood dans n'importe quel ordre. Pour obtenir la complexité indiquée ci-dessus, il faut raffiner légèrement la fonction *canFindAnotherWatch* décrite précédemment. Plus précisément, il faut considérer que l'ensemble des décisions de chaque nogood est représenté en utilisant un tableau, et on suppose ici qu'avant d'appeler la fonction *propagate*, les deux premières décisions de chaque tableau sont échangées avec celles couramment watchées. Cette opération peut être réalisée en $O(|\mathscr{B}|)$. A ce moment, à chaque fois que l'on doit trouver une nouvelle décision à sélectionner en utilisant la fonction *canFindAnotherWatch*, il suffit juste d'itérer les décisions du nogood Δ (passé en paramètre) en commençant par l'index suivant la plus grande position des deux décisions couramment watchées jusqu'à la dernière décision du tableau. Ainsi, pour un appel à l'algorithme *propagate*, quelque soit le nombre d'appels à la fonction *canFindAnotherWatch* (et donc à la fonction *inferences*), il suffira de contrôler au maximum $|\Delta|$ décisions par nogood Δ. Par conséquent, la complexité temporelle cumulée, dans le pire des cas, pour exploiter n'importe quel nogood Δ est de $O(|\Delta|)$, ce qui revient à $O(n)$. De manière globale, la complexité temporelle cumulée, dans le pire des cas de l'algorithme *inferences* dans *propagate* est alors de $O(n|\mathscr{B}|)$. □

On peut noter que la complexité temporelle, dans le pire des cas, pour exploiter les nld-nogoods réduits pour chaque branche de l'arbre de recherche, de la racine à une feuille, est en $O(n^2|\mathscr{B}|)$ puisque l'événement "la variable dont le domaine devient singleton " ne peut survenir qu'une seule fois par variable et par branche.

Corollaire 3. *La complexité temporelle, dans le pire des cas, de la fonction propagate est $O(er^2d^r + n|\mathscr{B}|)$ où r est l'arité maximale des contraintes.*

Démonstration. Le coût nécessaire pour établir GAC est de $O(er^2d^r)$ lorsqu'on utilise un algorithme générique tel que GAC2001 [Bessiere *et al.*, 2005] et l'on vient de démontrer que le coût d'exploitation des nogoods est en $O(n|\mathscr{B}|)$. □

Proposition 13. *La complexité spatiale, dans le pire des cas, pour stocker les nld-nogoods réduits est $O(n(d + |\mathscr{B}|))$.*

Démonstration. Nous savons que $|\mathscr{B}|$ nogoods de taille au plus n peuvent être enregistrés. De plus, le nombre de cellules introduites pour accéder à ces nogoods est $O(|\mathscr{B}|)$ et la taille du tableau associé à chaque décision négative est $O(nd)$. On obtient alors $O(n(d + |\mathscr{B}|))$. □

2.3.5 Des nogoods minimisés

Les nld-nogoods peuvent être réduits en taille [Lecoutre *et al.*, 2007d] en ne considérant que les décisions positives (c.f. section 2.1). Le concept de nld-nogoods réduits minimisés (par rapport à un opérateur d'inférence établissant une consistance ϕ) poursuit le même objectif. Les nogoods obtenus sont alors encore plus puissants.

Des ϕ-nogoods minimaux

Comme indiqué en section 1.2.1, étant donné un réseau P, $\phi(P)$ est le réseau de contraintes obtenu après application d'un opérateur d'inférence établissant une consistance ϕ sur le réseau P et si $\phi(P) = \bot$ alors P est clairement insatisfaisable. Nous rappelons également qu'étant

donné un réseau de contraintes P et un ensemble de décisions Δ, Δ est un ϕ-nogood de P si et seulement si $\phi(P|_\Delta) = \bot$ et Δ est un ϕ-nogood minimal de P si et seulement si $\nexists \Delta' \subset \Delta$ tel que $\phi(P|_{\Delta'}) = \bot$.

Bien sur, les ϕ-nogoods sont également des nogoods. Cependant la réciproque n'est pas nécessairement vraie. Un ϕ-nogood peut être minimisé en utilisant un algorithme polynomial tel que QuickXplain ou l'une de ses variantes (c.f. section 2.2.3). Dans notre contexte, nous savons qu'à la fin de chaque exécution, on peut extraire des nld-nogoods à partir de la branche courante et les réduire. Il est intéressant de noter que les nld-nogoods qui ne sont pas des ϕ-nogoods peuvent directement être identifiés et évités. Ce cas se produit lorsque la dernière décision δ_m de la nld-sous-séquence à partir de laquelle un nld-nogood Δ vient d'être extrait, ne mène pas directement à un échec après application de l'opérateur établissant ϕ. En effet cela veut dire que δ_m est la racine d'un sous-arbre non-trivial qu'il a fallu explorer. D'un autre coté, si δ_m mène directement à une impasse, cette décision fait nécessairement partie de n'importe quel ϕ-nogood inclus dans Δ. Par conséquent δ_m peut directement être sélectionnée comme étant la première décision (dite de transition) de l'algorithme de minimisation défini en section suivante.

La figure 2.5 représente un arbre de recherche construit lors d'une exécution d'un algorithme de recherche et interrompue après avoir réfuté la décision δ_{10}. Un opérateur d'inférence établissant une consistance ϕ est maintenu à chaque nœud de cet arbre de recherche. Parmi les quatre nld-nogoods qui peuvent être extraits, seulement deux mènent directement à une impasse :

- $\Delta_1 = \{\delta_1, \neg\delta_2, \delta_6\}$;
- $\Delta_2 = \{\delta_1, \neg\delta_2, \neg\delta_6, \delta_7, \neg\delta_8, \delta_{10}\}$.

Δ_1 et Δ_2 sont clairement des ϕ-nogoods puisque l'application de l'opérateur établissant ϕ après chacune des décisions δ_6 et δ_{10} mène directement à une inconsistance. Les nld-nogoods réduits

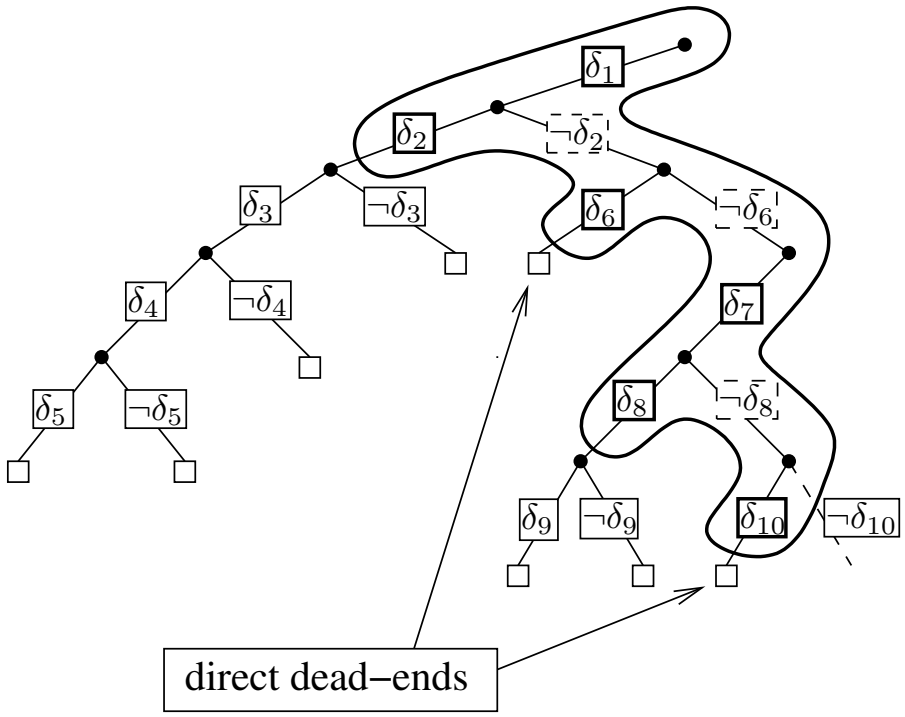


FIG. 2.5 – Identification de nld-nogoods (réduits) susceptibles d'être minimisés

qui peuvent être utilisés lors du processus de minimisation sont donc $\Delta'_1 = \{\delta_1, \delta_6\}$ et $\Delta'_2 = \{\delta_1, \delta_7, \delta_{10}\}$. Il est important de souligner que même si un nld-nogood est un ϕ-nogood, le nld-nogood réduit obtenu à partir de celui-ci n'est pas nécessairement lui même un ϕ-nogood. En effet, certaines décisions négatives, éliminées durant le processus de réduction, peuvent être impliquées dans le conflit.

Dans certains cas, les nld-nogoods réduits peuvent être grandement minimisés et l'impact sur les exécutions suivantes peut être vraiment significatif. Dans le meilleur des cas, on peut espérer isoler des ϕ-nogoods minimaux de taille 1. Ils correspondent en fait à des valeurs singleton ϕ-inconsistantes. Par exemple, considérons le problème académique des reines et cavaliers (queens-knights problem) défini en section 1.3 et l'algorithme MAC pour le résoudre. Comme les variables cavalier sont singleton arc-inconsistantes, si une telle variable est impliquée dans la dernière décision d'un nld-nogood réduit, alors un algorithme de minimisation prouvera que celle-ci est singleton arc-inconsistante. Les résultats prouvant un tel comportement sur ce type de problème sont indiqués dans la table 2.3 de la section 2.4.

Différentes techniques de minimisation

De nombreux travaux concernant l'identification de ϕ-nogoods minimaux ont déjà été proposés [Junker, 2001, Petit *et al.*, 2003]. Puisque extraire un ϕ-nogood minimal est une activité limitée à une branche de l'arbre de recherche, les algorithmes proposés utilisent (au moins partiellement) une approche constructive afin de bénéficier de l'incrémentalité du processus de propagation. D'un autre coté la dernière version de QuickXplain [Junker, 2004] exploite une approche de type "divide and conquer" (comme dans [Mauss et Tatar, 2002]) mais définie dans un contexte plus général. Cette technique peut par exemple être employée pour extraire des noyaux minimaux insatisfaisables (également appelés MUCs pour Minimal Unsatisfiable Cores) à partir de réseaux de contraintes. Etant donné un problème $P = (\mathscr{X}, \mathscr{C})$, un MUC est un réseau insatisfaisable $P' = (\mathscr{X}', \mathscr{C}')$ tel que $\mathscr{X}' \subseteq \mathscr{X}$ et $\mathscr{C}' \subseteq \mathscr{C}$. Evidemment le réseau P est lui même insatisfaisable. Cette approche a récemment été étudiée aussi bien théoriquement que pratiquement dans [Hemery *et al.*, 2006].

En résumé, pour extraire un ϕ-nogood minimal (à partir d'un ϕ-nogood), il faut itérativement identifier les décisions impliquées dans ce ϕ-nogood minimal. Plus précisément, étant donné un ϕ-nogood $\Delta = \{\delta_1, \delta_2, \ldots, \delta_m\}$ d'un réseau de contraintes P et un ordre total sur les décisions (pour simplifier, on peut considérer l'ordre naturel δ_1, δ_2, $\ldots$, δ_m des décisions), on cherche à identifier une décision δ_i telle que $\phi(P|_{\{\delta_1,\ldots,\delta_{i-1}\}}) \neq \bot$ et $\phi(P|_{\{\delta_1,\ldots,\delta_i\}}) = \bot$. Cette décision, qui fait clairement partie d'un ϕ-nogood minimal, est appelée décision de transition de Δ (d'après l'ordre précédemment fixé). On peut noter que n'importe quelle décision δ_j telle que $j > i$ peut être supprimée de manière sûre. Cette notion de décision de transition est analogue à celle de contrainte de transition définie dans [Hemery *et al.*, 2006].

Plusieurs approches peuvent être envisagées pour identifier les décisions de transition. On peut considérer par exemple une approche constructive, destructive ou dichotomique. L'approche constructive consiste à ajouter successivement (d'après l'ordre précédemment établi) les décisions de Δ au réseau de contraintes jusqu'à ce qu'une inconsistance soit détectée suite à l'application de l'opérateur d'inférence établissant ϕ. Dans l'approche destructive, le processus est inversé. On commence par ajouter initialement toutes les décisions de Δ au réseau de contraintes (ce qui rend le réseau de contraintes inconsistant), puis on retire successivement les décisions de Δ une à une, jusqu'à ce que l'inconsistance ne soit plus détectée après application de l'opérateur établissant ϕ. Dans la troisième alternative, la décision de transition est identifiée en utilisant une recherche dichotomique.

On peut adopter l'une des trois approches décrites ci-dessus pour identifier un ϕ-nogood minimal. Dans l'approche constructive, après avoir identifié la première[4] décision de transition δ_i de Δ, on cherche la seconde en ayant supprimé toutes les décisions δ_j avec $j > i$ de Δ (puisque l'insatisfaisabilité du réseau est préservée) et en considérant un nouvel ordre sur les décisions tel que toutes les décisions de transition identifiées soient les plus petites. Ce procédé peut être répété jusqu'à ce que toutes les décisions du nogood courant correspondent à des décisions de transition successivement identifiées. Le principe de ce procédé itératif a été décrit plus précisément dans [de Siqueira et Puget, 1988, Junker, 2001, Petit *et al.*, 2003, Hemery *et al.*, 2006].

Lorsqu'on s'intéresse à identifier des nogoods minimaux, les approches constructives, destructives et dichotomiques présentées succinctement ci-dessus sont à mettre en relation avec les algorithmes RobustXplain, ReplayXplain et QuickXplain [Junker, 2001]. Cependant ici, on se base sur l'hypothèse de l'incrémentalité de l'opérateur d'inférence. Ceci signifie que la complexité temporelle, dans le pire des cas, d'appliquer une consistance ϕ sur un réseau de contraintes à partir d'un ensemble de décision Δ est égale à la complexité temporelle, dans le pire des cas, d'appliquer une consistance ϕ sur un réseau de contraintes à partir d'un ensemble de décision Δ' si $\Delta \subset \Delta'$. Par exemple, tous les algorithmes (connus) établissant la consistance d'arc généralisée ($\phi = (G)AC$) sont incrémentaux. Par conséquent, utiliser une approche constructive pour identifier les décisions de transition est mieux adaptée à notre contexte. C'est l'algorithme utilisé pour les expérimentations et sa complexité est discutée en section suivante.

Analyse de la complexité

Proposition 14. *La complexité temporelle, dans le pire des cas, pour extraire un GAC-nogood minimal à partir d'un nld-nogood réduit est $O(enr^2d^r)$.*

Démonstration. Un algorithme générique tel que GAC2001 établissant *GAC* sur un réseau de contraintes est incrémental. Par conséquent, en utilisant une approche constructive, l'identification d'une décision de transition peut s'effectuer en $O(er^2d^r)$, c'est-à-dire la complexité temporelle, dans le pire des cas, établissant GAC. Si le nogood extrait est composé de k décisions, alors on obtient une complexité globale en $O(ker^2d^r)$. Comme k est en $O(n)$, on obtient $O(enr^2d^r)$. □

Corollaire 4. *Dans le cas binaire (i.e. pour $r = 2$), la complexité temporelle, dans le pire des cas, pour extraire un AC-nogood minimal à partir d'un nld-nogood réduit est $O(end^2)$.*

Proposition 15. *La complexité temporelle, dans le pire de cas, pour extraire les AC-nld-nogoods réduits minimaux (à la fin de chaque exécution) est $O(en^2r^2d^{r+1})$.*

Démonstration. Extraire un GAC-nogood à partir d'un nld-nogood réduit s'effectue en $O(enr^2d^r)$ et il y a au plus $O(nd)$ nld-nogoods réduits à minimiser. □

Corollaire 5. *Dans le cas binaire (i.e. pour $r = 2$), la complexité temporelle, dans le pire des cas, pour enregistrer les GAC-nld-nogoods réduits minimaux (à la fin de chaque exécution) est $O(en^2d^3)$.*

[4] Si $\Delta=\{\delta_1, \delta_2, \ldots, \delta_m\}$ est un ϕ-nogood (et $\delta_1, \delta_2, \ldots, \delta_m$ un ordre naturel des décisions), alors on peut directement considérer δ_m comme la première décision de transition.

2.4 Résultats expérimentaux

Pour montrer l'intérêt pratique de l'approche décrite précédemment, de nombreuses expérimentations ont été conduites sur un processeur Xeon, cadencé à 3 GHz et 1 GiB de mémoire. Le solveur CSP utilisé est Abscon. Le coeur du solveur est un algorithme M(G)AC (embarquant $GAC3^{rm}$). Au vu des résultats obtenus par Abscon aux compétitions de solveurs CSP de 2005[5] et 2006[6], on peut le considérer comme un solveur CSP générique état de l'art. Nous avons étudié l'impact des redémarrages (noté $MGAC{+}RST$), de l'enregistrement de nogoods à partir des redémarrages (noté $MGAC{+}RST{+}NG$), ainsi que cette même technique mais exploitant des nogoods minimisés (noté $MGAC{+}RST{+}NGm$) sur l'efficacité de la recherche. En ce qui concerne la politique de redémarrage utilisée, le nombre initial de retours-arrière autorisés pour la première exécution a été fixé à 10 et le facteur d'augmentation à $1,5$ (i.e. à chaque nouvelle exécution, le nombre de retours-arrière autorisés augmente d'un facteur $1,5$). Cette politique de redémarrage, basée sur une suite géométrique, a été par exemple mentionnée dans [Walsh, 1999]. D'autres politiques de redémarrages ont été testées et il est intéressant de signaler que le comportement général des algorithmes décrits précédemment reste relativement similaire.

Au cours de la recherche, trois heuristiques de choix de variable différentes ont été essayées : les heuristiques classiques *brelaz* et *dom/ddeg* ainsi que l'heuristique adaptative *dom/wdeg*. Bien sûr lorsque les redémarrages sont exploités, un phénomène aléatoire est introduit pour les heuristiques *brelaz* et *dom/ddeg* afin de départager les éventuels ex-aequo. Lorsqu'on utilise *dom/wdeg*, le poids des contraintes est conservé d'une exécution à l'autre, ce qui rend l'introduction d'un caractère aléatoire (les poids associés aux contraintes étant suffisamment discriminants).

Nous avons tout d'abord testé les quatre algorithmes sur l'ensemble des 3621 instances utilisées pour le premier tour de la compétition de solveurs CSP 2006. La durée limite pour résoudre une instance a été fixée à 20 minutes. Le tableau 2.1 fournit une vue d'ensemble des résultats obtenus en terme de nombre d'instances non résolues – étant donné la durée limite fixée – (*#timeouts*) et le temps cpu moyen en secondes (*avg time*) calculé pour les instances résolues par les quatre méthodes.

Tout d'abord, quelque soit l'heuristique utilisée, sur les instances aléatoires, redémarrer la recherche semble pénalisant. Ceci n'est pas vraiment surprenant puisqu'il n'y a pas de structures à exploiter d'une exécution à l'autre. Ce résultat est également confirmé par celui observé sur les instances aléatoires SAT. Sur un tel type d'instance, aucun phénomène de type "longues traînées" ne peut être mis en évidence. On peut cependant remarquer que les résultats obtenus par la méthode $MGAC{+}RST{+}NG$ sont assez proches de ceux obtenus par l'algorithme $MGAC$ sans redémarrage. De plus, sur les instances structurées, comme on pouvait s'y attendre, l'enregistrement de nogoods à partir des redémarrages permet d'améliorer l'efficacité de la recherche, aussi bien en nombre d'instances résolues qu'en temps cpu moyen. L'exploitation des nogoods minimisés a également un impact significatif, en particulier lorsque les heuristiques classiques sont utilisées. En analysant de manière globale les résultats obtenus, tandis que les redémarrages, sans apprentissage, mènent à des résultats plutôt mitigés, l'enregistrement de nogoods à partir des redémarrages améliore de façon significative la robustesse du solveur. En effet à la fois le nombre d'instances non-résolues et le temps cpu moyen sont réduits. Ceci est dû au fait que le solveur n'explore jamais plusieurs fois les mêmes portions de l'espace de recherche tout en bénéficiant de l'aspect opportuniste (diversification de la recherche) des redémarrages. Les résultats présentés dans le tableau 2.1 sur les instances du premier tour de la compétition de

[5] `http://cpai.ucc.ie/05/CPAI.html`
[6] `http://www.cril.univ-artois.fr/CPAI06`

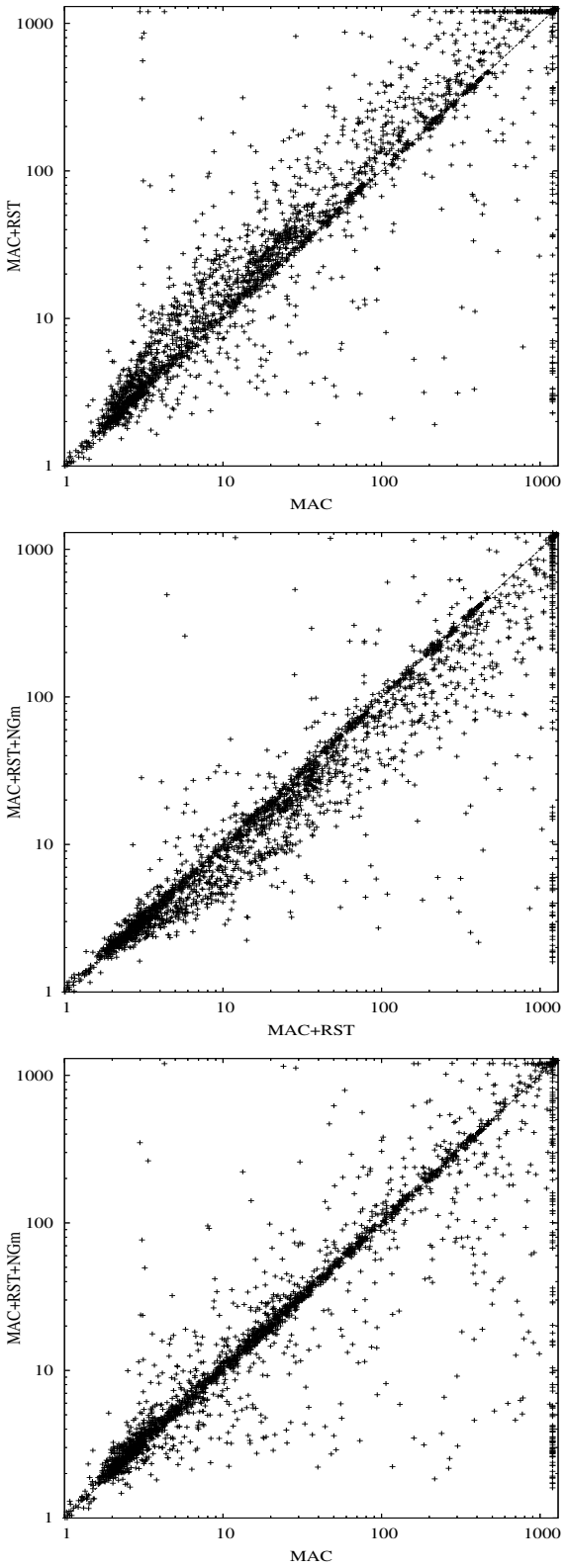


FIG. 2.6 – Comparaison deux à deux (en temps cpu) sur les 3 621 instances de la compétition 2006 de solveurs CSP (premier tour). L'heuristique de choix de variable est dom/ddeg et le temps limite pour résoudre une instance est fixé à 20 minutes.

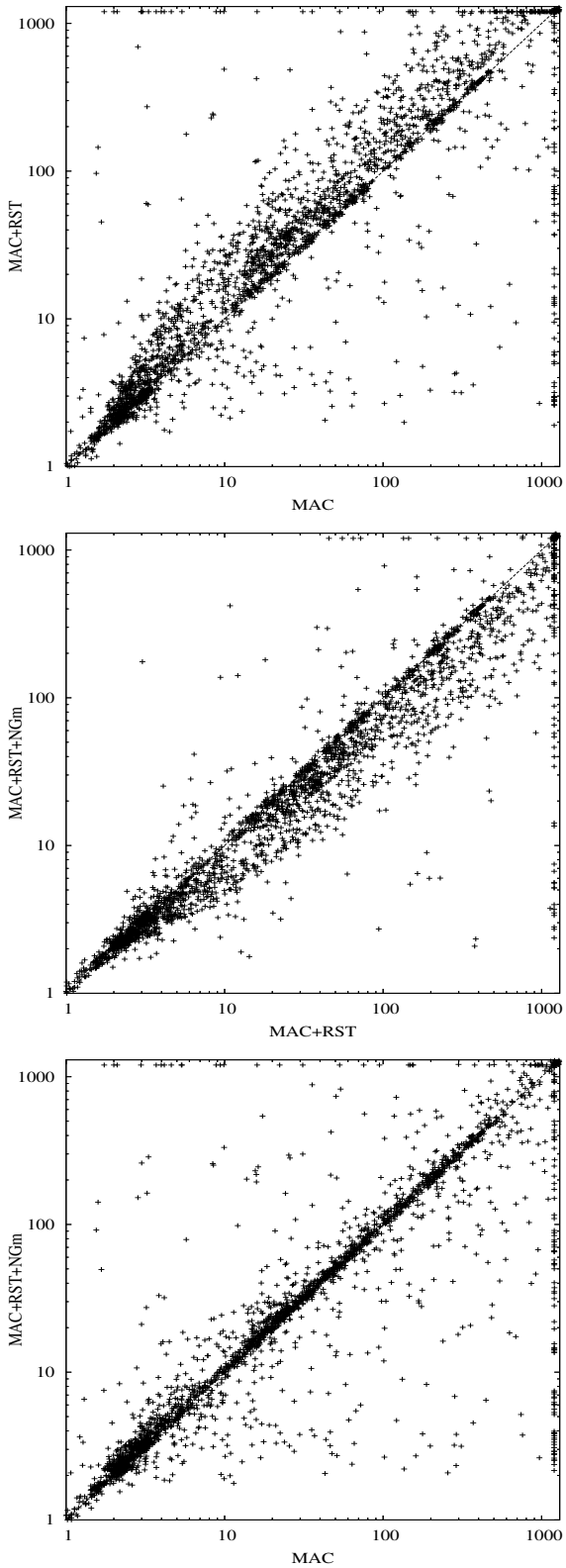


FIG. 2.7 – Comparaison deux à deux (en temps cpu) sur les 3 621 instances de la compétition 2006 de solveurs CSP (premier tour). L'heuristique de choix de variable est brelaz et le temps limite pour résoudre une instance est fixé à 20 minutes.

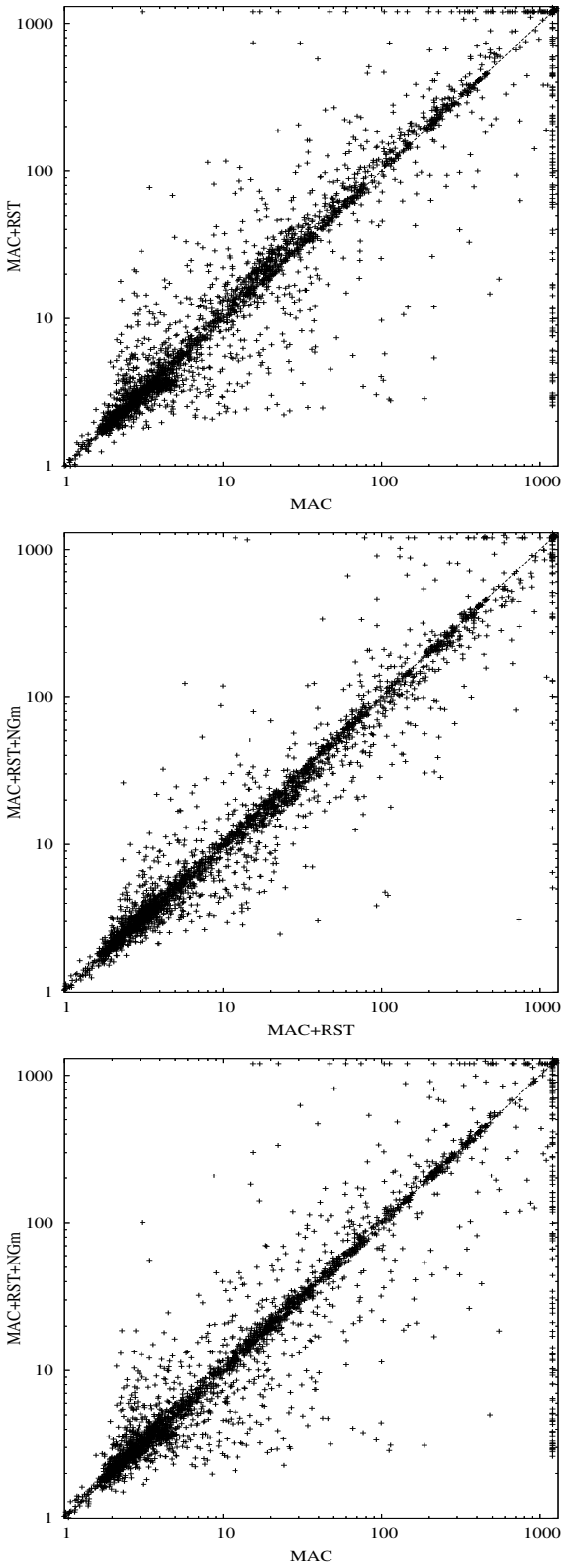


FIG. 2.8 – Comparaison deux à deux (en temps cpu) sur les 3 621 instances de la compétition 2006 de solveurs CSP (premier tour). L'heuristique de choix de variable est dom/wdeg et le temps limite pour résoudre une instance est fixé à 20 minutes.

		$M(G)AC$			
			$+RST$	$+RST+NG$	$+RST+NGm$
Instances aléatoires (1 390 instances)					
$dom/ddeg$	$\#timeouts$	270	301	276	273
	$avg\ time$	40,4	57,6	41,9	42,0
$brelaz$	$\#timeouts$	305	330	311	311
	$avg\ time$	73,2	103,2	70,5	71,1
$dom/wdeg$	$\#timeouts$	266	278	274	268
	$avg\ time$	36,4	45,8	38,1	41,2
Instances structurées (2 231 instances)					
$dom/ddeg$	$\#timeouts$	873	863	825	772
	$avg\ time$	87,5	97,8	79,7	72,4
$brelaz$	$\#timeouts$	789	788	757	738
	$avg\ time$	79,0	92,4	74,8	71,5
$dom/wdeg$	$\#timeouts$	623	554	551	551
	$avg\ time$	50,5	51,3	51,4	50,8
Toutes (3 621 instances)					
$dom/ddeg$	$\#timeouts$	1 143	1 164	1 101	1 045
	$avg\ time$	66,1	79,6	62,6	58,6
$brelaz$	$\#timeouts$	1 094	1 118	1 068	1 049
	$avg\ time$	76,4	97,3	72,8	71,3
$dom/wdeg$	$\#timeouts$	889	832	825	819
	$avg\ time$	44,1	48,8	45,4	46,5

TAB. 2.1 – Nombre d'instances non résolues et temps moyen sur les problèmes de la compétition de solveurs CSP de 2006 (premier tour), étant donné 20 minutes.

solveurs sont représentés graphiquement sous une autre forme par les figures 2.6, 2.7 et 2.8. Elles représentent sous la forme d'un nuage de points, les comparaisons deux à deux des différentes méthodes proposées pour les heuristiques *dom/ddeg*, *brelaz* et *dom/wdeg*. Remarquons que les nombreux points sur le coté droit de ces figures représentent les instances non résolues par la méthode dont le nom étiquette l'axe des abscisses (et résolues par l'autre méthode avec laquelle on effectue la comparaison).

Lorsqu'on s'intéresse plus précisément aux instances les plus difficiles (qui impliquent 680 variables et 50 valeurs dans le plus grand domaine) construites à partir du problème réel d'assignation de fréquences radios (RLFAP pour Radio Link Frequency Assignment Problem), l'utilisation d'une politique de redémarrage permet d'être plus efficace par presque un ordre de magnitude. Ces résultats ont été reportés dans le tableau 2.2. Les performances sont mesurées en terme de temps cpu (en secondes), de quantité de mémoire consommée (en octets) et en nombre de nœuds visités. Quand on exploite l'enregistrement de nogoods, le gain est d'approximativement 10%. Un point qu'il est utile de mentionner ici est que l'enregistrement de nogoods à partir des redémarrages ne requiert pas beaucoup de mémoire (en fonction de la politique de redémarrage utilisée). Ainsi on peut noter que le nombre et la taille des nld-nogoods réduits enregistrés pendant la

		MAC			
			+RST	*+RST+NG*	*+RST+NGm*
scen11-f10	*cpu*	$5,8$	$4,9$	$5,0$	$5,0$
	mem	$29M$	$31M$	$32M$	$32M$
	nuds	891	403	405	556
scen11-f8	*cpu*	$10,2$	$5,5$	$5,8$	$6,1$
	mem	$29M$	$31M$	$32M$	$32M$
	nuds	15 045	1 149	1 098	1 287
scen11-f6	*cpu*	$59,8$	$14,8$	$13,5$	$10,9$
	mem	$29M$	$31M$	$32M$	$32M$
	nuds	$217K$	35 030	25 851	19 798
scen11-f4	*cpu*	$924,1$	$141,1$	$116,6$	$125,9$
	mem	$30M$	$31M$	$32M$	$32M$
	nuds	$3\,458K$	$494K$	$450K$	$476K$
scen11-f3	*cpu*	*time-out*	$370,6$	$361,5$	$342,5$
	mem		$31M$	$32M$	$32M$
	nuds		$1\,506K$	$1\,331K$	$1\,314K$
scen11-f2	*cpu*	*time-out*	*time-out*	$1\,135,3$	$1\,137,7$
	mem			$32M$	$32M$
	nuds			$4\,406K$	$4\,370K$

TAB. 2.2 – Performances obtenues sur les instances difficiles RLFAP avec l'heuristique *dom/wdeg* (temps limite fixé à 20 minutes)

recherche sont toujours très limités. Les résultats obtenus sur l'instance *scen*11-$f1$ illustre tout à fait ce propos. MAC+RST+NG résout cette instance en 36 exécutions (et 3 750 secondes) alors que seulement 712 nogoods (dont la taille moyenne est de $8,5$ variables et la taille du nogood le plus grand est de 33 variables) ont été enregistrés.

Finalement, le tableau 2.3 présente les résultats obtenus sur les instances du problème des reines et cavaliers (queens-knights problem) en utilisant l'heuristique *dom/wdeg*. Comme cela est indiqué en section 2.3.5, la minimisation de nogoods est assez efficace sur ce type d'instances puisque les valeurs singleton arc-inconsistantes peuvent être détectées. Les résultats sont moins impressionnants lorsqu'on utilise l'heuristique *dom/wdeg*. En effet, ceci peut s'expliquer par le fait que cette heuristique est moins sensible au thrashing que les autres heuristiques. Les contraintes les plus violées correspondent à celles du sous-problème des cavaliers (qui est un noyau insatisfaisable). Les variables impliquées dans ces contraintes sont donc sélectionnées en priorité par l'heuristique de choix de variable, ce qui limite l'apparition du phénomène de thrashing.

2.5 Conclusion

Dans ce chapitre, nous avons montré l'intérêt de combiner l'enregistrement de nogoods avec une politique de redémarrage. Redémarrer la recherche permet d'éviter l'apparition du phénomène heavy-tailed observé sur certaines instances structurées. Cependant l'inconvénient de cette technique est qu'il est possible d'explorer plusieurs fois les mêmes portions de l'espace de recherche. Il est possible d'éliminer cet inconvénient en enregistrant un ensemble de nogoods à la fin de chaque exécution (ceci peut être mis en relation avec la technique de *recherche de signa-*

		MAC			
			$+RST$	$+RST+NG$	$+RST+NGm$
qk-12-5-mul	$dom/ddeg$	$265,1$	$408,9$	$256,2$	$2,1$
	$brelaz$	$255,7$	$377,9$	$250,8$	$2,1$
	$dom/wdeg$	$3,1$	$1,8$	$2,6$	$1,6$
qk-25-5-mul	$dom/ddeg$	*time-out*	*time-out*	*time-out*	$4,9$
	$brelaz$	*time-out*	*time-out*	*time-out*	$5,1$
	$dom/wdeg$	*time-out*	$4,2$	$4,8$	$4,3$
qk-50-5-mul	$dom/ddeg$	*time-out*	*time-out*	*time-out*	$67,3$
	$brelaz$	*time-out*	*time-out*	*time-out*	$65,3$
	$dom/wdeg$	*time-out*	$59,5$	$44,6$	$43,9$

TAB. 2.3 – Performances obtenues sur les instances du problème des "reines et cavaliers" (queens-knights problem) – temps limite fixé à 20 minutes –

ture proposée dans le contexte de la satisfaisabilité booléenne [Baptista *et al.*, 2001]). Pour des raisons d'efficacité, ces nogoods sont enregistrés dans une base (et ne correspondent donc pas à de nouvelles contraintes) et la propagation est effecutée en utilisant la technique des watched literals introduite pour SAT. On peut considérer la base de nogoods comme une contrainte globale unique associée à un algorithme de propagation efficace.

De façon intéressante, cet algorithme de filtrage peut être intégré à n'importe quel algorithme de propagation de contraintes mais peut également être intégré à n'importe quel algorithme générique établissant GAC. Dans cette approche, les nld-nogoods réduits correspondent à des décisions positives (donc à des nogoods standard). Ceci nous permet, pour l'exploitation de la base de nogoods, d'obtenir une complexité assez faible puique le seul événement à intercepter est celui où une variable devient singleton. L'une des perspectives que l'on peut envisager est de calculer non plus des nogoods standard mais des nogoods généralisés. Pour cela, il peut être intéressant d'extraire de la dernière branche des nld-nogoods minimaux (mais non réduits). Les aspects théoriques et pratiques de cette alternative doivent encore être étudiés.

3

Recherche basée sur les états

Pour les algorithmes classiques de recherche heuristique (A*, IDA*...) ou les algorithmes de recherche dédiés aux jeux (α-β, SSS*...), les nœuds de l'arbre de recherche représentent des états du monde et les transitions représentent des mouvements. Aussi, de nombreux états identiques peuvent être rencontrés plusieurs fois à des profondeurs différentes ; ceci étant dû au fait qu'à partir d'un même état initial, différentes séquences de mouvement peuvent mener à des situations du monde identiques. De plus, un état S d'un nœud d'un arbre de recherche rencontré à la profondeur i ne peut mener à une solution meilleure qu'un nœud contenant ce même état S mais rencontré précédemment à une profondeur $j < i$. Certaines parties de l'espace de recherche peuvent donc être explorées plusieurs fois (et inutilement), parfois de manière coûteuse. Cette exploration répétitive d'un même état est similaire à la notion de "thrashing" présentée en section 1.3 dans le contexte de la résolution du problème de satisfaction de contraintes.

Revisiter des états identiques, atteints à partir de différentes séquences de transitions, mieux connus sous le nom de *transpositions*, a été identifié très tôt notamment dans le domaine des jeux d'échec [Greenblatt *et al.*, 1967, Slate et Atkin, 1977, Marsland, 1992]. Une solution à ce problème est d'enregistrer chaque nœud rencontré ainsi que certaines informations pertinentes (e.g. la profondeur, l'évaluation de l'heuristique), dans une *table de transposition*. La structure de données utilisée pour implémenter cette table de transposition est généralement une table de hachage dont la clef est calculée à partir de la description de l'état, par exemple la clef basée sur l'opérateur logique XOR pour les échecs [Zobrist, 1970]. La quantité de mémoire des machines étant limitée, le nombre d'entrées dans la table de transposition doit être borné. Cette technique a été adaptée aux algorithmes de recherche heuristique tel que IDA* [Reinefeld et Marsland, 1994] et a été exploitée avec succès dans des planificateurs STRIPS modernes comme par exemple FF [Hoffmann et Nebel, 2001] et YAHSP [Vidal, 2004].

L'exemple 3.1 illustre sur le problème de planification du monde des blocs, un état du monde (i.e. une situation) atteinte à partir de deux séquences différentes de mouvements. Initialement, l'état du monde peut être représenté par la position des blocs suivante : $\{sur(table, A)$, $sur(table, B)$, $sur(B, C)$, $sur(table, D)$, $libre(C)$, $libre(D)$, $libre(A)\}$. L'action (i.e. mouvement) $poserSur$ permet de déplacer les blocs et modifie l'état du monde. Plus précisément, $poserSur(X, Y, Z)$ déplace l'objet X initialement posé sur l'objet Y et le place sur l'objet Z (si celui-ci est libre). On remarque alors qu'appliquer la séquence de mouvements $\langle poserSur(C, B, D), poserSur(B, table, A)$, $poserSur(C, D, B)\rangle$ ou la séquence de mouvements $\langle poserSur(C, B, table), poserSur(B, table, A)$, $poserSur(C, table, B)\rangle$ mènent à la même situation finale : $\{sur(table, A)$, $sur(A, B)$, $sur(B, C)$, $sur(table, D)$, $libre(C)$, $libre(D)\}$. Ainsi en enregistrant dans une table de transposition l'état rencontré après la première séquence de mouvements, l'exploration effectuée à partir du même

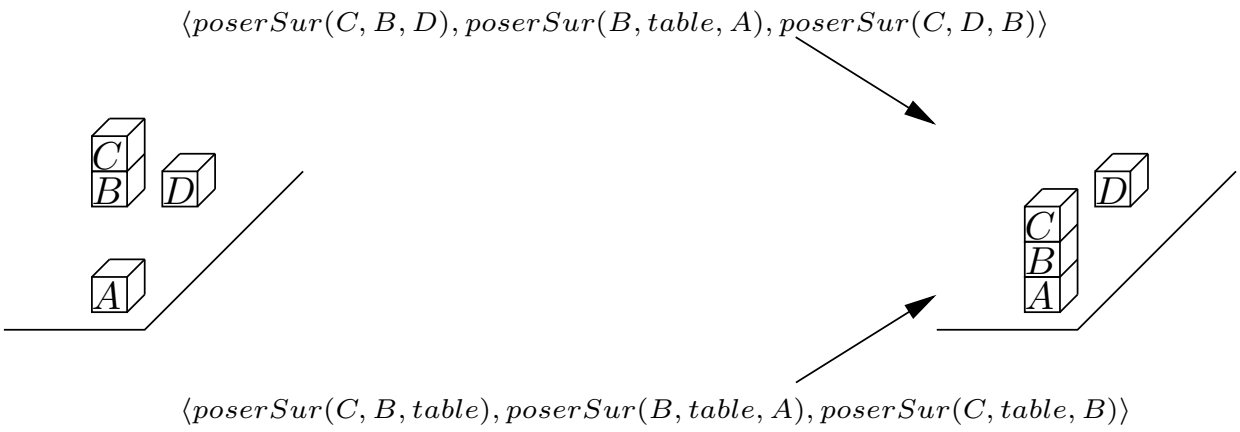
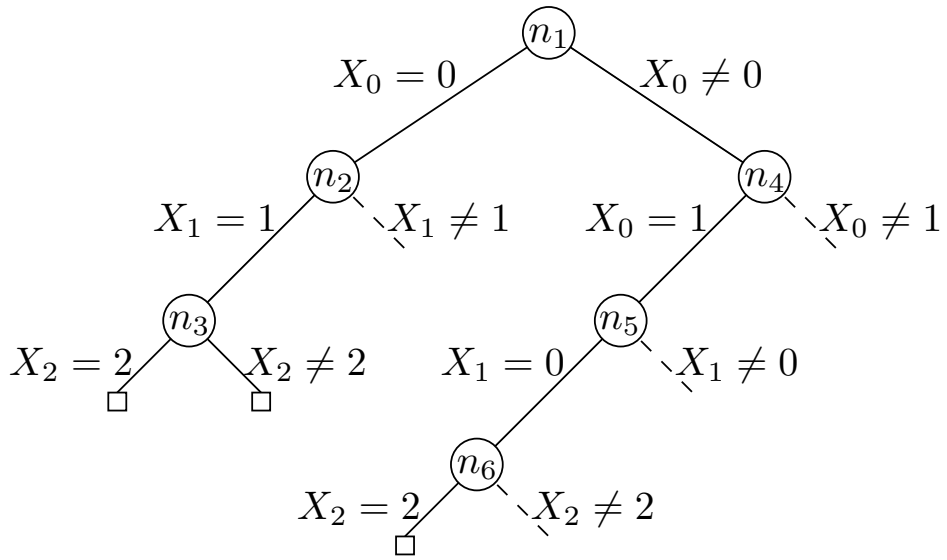


FIG. 3.1 – Redondance des situations dans le monde des blocs

état, issue de la deuxième séquence de mouvements aurait pu être évitée.

Lorsqu'on s'intéresse à la résolution complète d'un réseau de contraintes, l'utilisation directe des tables de transposition dans un algorithme de recherche avec retours-arrière n'est clairement pas intéressante. Ce type d'algorithme possède une propriété garantissant qu'un même état d'un réseau de contraintes (c'est-à-dire les variables et les domaines réduits) ne peut pas être rencontré deux fois au cours de la recherche. En effet, en utilisant par exemple un schéma de branchement binaire, une fois qu'il a été prouvé qu'une décision positive $X = v$ mène à une contradiction, la décision opposée $X \neq v$ est immédiatement prise sur l'autre branche. En d'autres termes, dans la première branche le domaine de la variable X est réduit au singleton $\{v\}$, tandis que dans la seconde branche v est supprimée du domaine de X : évidemment, aucun état où la décision $X = v$ a été prise ne peut être identique à un état où $X \neq v$ est vrai.

Cependant deux états Σ_1 et Σ_2, associés à deux nœuds de l'arbre de recherche et obtenus à partir de deux séquences de décision différentes effectuées pendant la résolution d'un réseau de contraintes P peuvent être réduits au même *état partiel*. Plusieurs *opérateurs de réduction* sont proposés pour identifier des états partiels, supprimant certaines variables sélectionnées (ainsi que leur domaine associé) [Lecoutre *et al.*, 2007e]. En pratique, à chaque nœud de la recherche on peut associer un état partiel que l'on enregistre dans une table de transposition, qu'il suffira de contrôler avant l'ouverture de chaque nouveau nœud (i.e. avant l'exploration de l'arbre de recherche dont la racine correspond au nœud obtenu après chaque décision). Plus précisément on enregistre les couples (variable,domaine) définissant cet état partiel. Les états partiels enregistrés peuvent caractériser des nœuds soit racines de sous-arbres inconsistants, soit associés aux solutions trouvées dans le sous-arbre correspondant. Une recherche dans cette table de transposition permettra alors d'éviter l'exploration d'un sous-arbre dont le nœud est associé à un état partiel déjà présent dans la table. Ceci permet notamment d'élaguer des branches de l'arbre de recherche dans lesquelles aucune solution ne peut apparaître ou dans lesquelles les solutions sont déjà connues. Une alternative à l'utilisation des tables de transposition : la structure des "watched literals" permet de contrôler la domination (inclusion) d'un état partiel dans un autre et non seulement l'équivalence [Lecoutre *et al.*, 2007b].



$$n_1: \begin{bmatrix} X_0 : \{0,1,2,3\} \\ X_1 : \{0,1,2,3\} \\ X_2 : \{0,1,2,3\} \\ X_3 : \{0,1,2,3\} \\ X_4 : \{0,1,2,3\} \end{bmatrix} \quad n_2: \begin{bmatrix} X_0 : \{0\} \\ X_1 : \{1,2,3\} \\ X_2 : \{1,2,3\} \\ X_3 : \{1,2,3\} \\ X_4 : \{1,2,3\} \end{bmatrix} \quad n_3: \begin{bmatrix} X_0 : \{0\} \\ X_1 : \{1\} \\ \mathbf{X_2} : \{\mathbf{2},\mathbf{3}\} \\ \mathbf{X_3} : \{\mathbf{2},\mathbf{3}\} \\ \mathbf{X_4} : \{\mathbf{2},\mathbf{3}\} \end{bmatrix}$$

$$n_4: \begin{bmatrix} X_0 : \{1,2,3\} \\ X_1 : \{0,1,2,3\} \\ X_2 : \{0,1,2,3\} \\ X_3 : \{0,1,2,3\} \\ X_4 : \{0,1,2,3\} \end{bmatrix} \quad n_5: \begin{bmatrix} X_0 : \{1\} \\ X_1 : \{0,2,3\} \\ X_2 : \{0,2,3\} \\ X_3 : \{0,2,3\} \\ X_4 : \{0,2,3\} \end{bmatrix} \quad n_6: \begin{bmatrix} X_0 : \{1\} \\ X_1 : \{0\} \\ \mathbf{X_2} : \{\mathbf{2},\mathbf{3}\} \\ \mathbf{X_3} : \{\mathbf{2},\mathbf{3}\} \\ \mathbf{X_4} : \{\mathbf{2},\mathbf{3}\} \end{bmatrix}$$

FIG. 3.2 – Problème des Pigeons : vue partielle de l'arbre de recherche

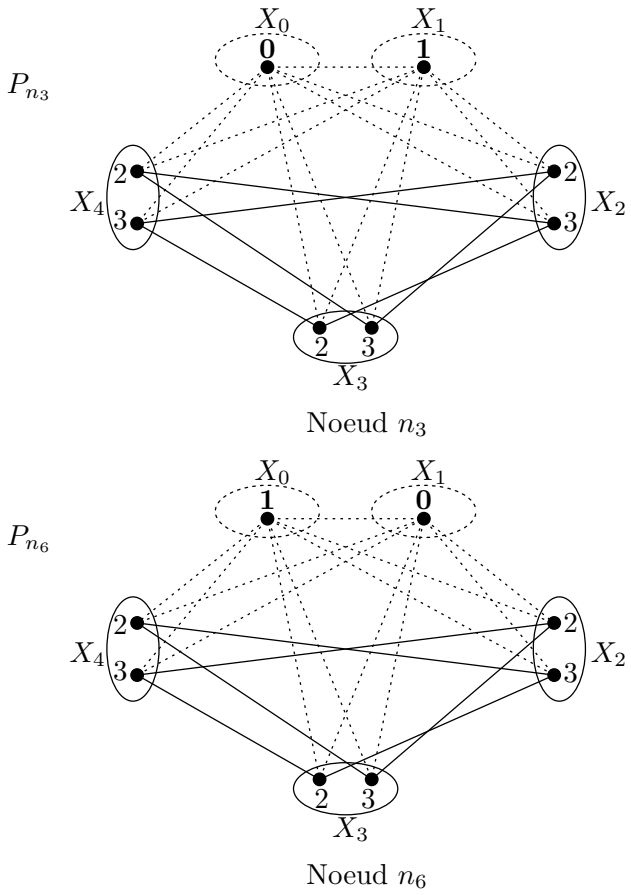


FIG. 3.3 – Problème des Pigeons : 2 sous-réseaux similaires

3.1 Analyse du problème des pigeons

Le problème académique des *pigeons* présenté ici avec cinq pigeons illustre notre propos. Ce problème est composé de cinq variables $X_0, \ldots, X_4$ représentant les pigeons, et dont le domaine initialement égal à $\{0, \ldots, 3\}$ représente les numéros des boites dans lesquelles on place les pigeons. Les contraintes de ce problème interdisent de mettre deux pigeons dans la même boîte, le rendant ainsi insatisfaisable puisqu'il y a cinq pigeons et seulement quatre boîtes. Ces contraintes peuvent être exprimées avec une clique de contraintes binaires : $X_0 \neq X_1, X_0 \neq X_2, \ldots, X_1 \neq X_2, \ldots$ La figure 3.2 décrit une vue partielle d'un arbre de recherche pour ce problème construit par MAC avec un schéma de branchement binaire ainsi que l'état des domaines à chaque nœud de cet arbre.

Remarquons tout d'abord que les six nœuds $n_1, \ldots, n_6$ représentent des réseaux qui sont tous différents, étant donné que les domaines de leurs variables diffèrent d'au moins une valeur. Cependant si on s'attarde un moment sur les nœuds n_3 et n_6, on s'aperçoit que les seules différences portent sur les variables X_0 et X_1 dont les domaines sont respectivement réduits aux singletons $\{0\}$ et $\{1\}$ dans n_3, et $\{1\}$ et $\{0\}$ dans n_6. Le domaine des autres variables X_2, X_3 et X_4 est égal à $\{2, 3\}$. Les réseaux associés aux nœuds n_3 et n_6 sont représentés sur la figure 3.3, incluant l'ensemble des instanciations autorisées pour chaque contrainte. La structure de ces deux réseaux est très ressemblante, la seule différence étant l'inversion des valeurs 0 et 1 entre

X_0 et X_1.

Les deux points cruciaux concernant les nœuds n_3 et n_6 sont les suivants : (1) X_0 et X_1 ne joueront plus aucun rôle dans la recherche associée à ces deux nœuds, et (2) vérifier la satisfaisabilité du réseau attaché au nœud n_3 est équivalent à vérifier la satisfaisabilité de n_6. Le point (1) est facile à constater : comme dans cet exemple, la consistance d'arc (AC) est maintenue, toutes les contraintes impliquant X_0 et X_1 sont universelles ; quelque soit la valeur assignée aux autres variables, ces contraintes seront satisfaites. Les variables X_0 et X_1 peuvent alors être déconnectées du réseau de contraintes associé aux nœuds n_3 et n_6. Le point (2) est alors également vrai : les deux sous-réseaux de contraintes impliquant les variables restantes X_2, X_3 et X_4 ainsi que les contraintes les impliquant sont identiques, et par conséquent n_3 est satisfaisable si et seulement si n_6 est satisfaisable. Si après avoir prouvé l'insatisfaisabilité de n_3, l'état partiel correspondant au sous-réseau extrait à partir du nœud n_3 avait été enregistré dans une table de transposition, l'exploration de n_6 aurait pu être évitée en contrôlant la table avant d'explorer le nœud.

3.2 Identifier des états partiels

Le concept d'état partiel d'un réseau de contraintes P est au centre du raisonnement. Rappelons d'abord que si P et Q sont deux réseaux de contraintes définis sur le même ensemble de variables $\mathscr{X}$ et de contraintes $\mathscr{C}$, $P \preceq Q$ si et seulement si $\forall X \in \mathscr{X}, dom^P(X) \subseteq dom^Q(X)$ et $rel^P(X) \subseteq rel^Q(X)$.

Un état partiel correspond à un ensemble de variables de P munies de leur domaine.

Définition 30 (état partiel). *Soit $P = (\mathscr{X}, \mathscr{C})$ un réseau de contraintes, un état partiel Σ de P est un ensemble de couples (X, D_X) avec $X \in \mathscr{X}$ et $D_X \subseteq dom^P(X)$ tel que chaque variable n'apparaisse au plus qu'une seule fois dans Σ.*

L'ensemble des variables apparaissant dans un état partiel Σ est noté $vars(\Sigma)$, et pour chaque couple $(X, D_X) \in \Sigma$, $dom^\Sigma(X)$ représente D_X. A chaque étape de la recherche (effectuée par un algorithme de recherche avec retours-arrière), un état partiel peut être associé avec le nœud correspondant de l'arbre de recherche. Cet état partiel est naturellement construit en prenant en compte toutes les variables et leur domaine courant, et sera appelé *état courant*.

Un réseau de contraintes peut être restreint par l'un de ses états partiels Σ en remplaçant dans P le domaine de chaque variable apparaissant dans Σ par son domaine correspondant dans Σ. Le réseau de contraintes restreint est clairement plus petit ($\preceq$) que le réseau initial.

Définition 31 (restriction d'un réseau par un état partiel). *Soit $P = (\mathscr{X}, \mathscr{C})$ un réseau de contraintes et Σ un état partiel de P. La restriction $\psi(P, \Sigma)$ de P par Σ est le réseau de contraintes $P' = (\mathscr{X}, \mathscr{C})$ tel que $\forall X \in \mathscr{X}, dom^{P'}(X) = dom^\Sigma(X)$ si $X \in vars(\Sigma)$, et $dom^{P'}(X) = dom^P(X)$ dans le cas contraire et $\forall C \in \mathscr{C}, rel^{P'}(C) = \{t \in rel^P(C) \mid \forall (X, v) \in t, v \in dom^{P'}(X)\}$.*

Un état partiel Σ d'un réseau de contraintes P est dit inconsistant par rapport à P lorsque le réseau défini comme la restriction de P par Σ est insatisfaisable.

Définition 32 (état partiel inconsistant). *Soit P un réseau de contraintes et Σ un état partiel de P. Σ est un état partiel inconsistant de P (Inconsistent Partial State, IPS_P en raccourci), si et seulement si $\psi(P, \Sigma)$ est insatisfaisable.*

Un état partiel Σ domine un réseau de contraintes P si chaque variable de Σ apparaît dans P avec un plus petit domaine.

Définition 33 (dominance d'un réseau). *Soit P un réseau de contraintes et Σ un état partiel. Σ domine P si et seulement si $\forall X \in vars(\Sigma), X \in vars(P)$ et $dom^P(X) \subseteq dom^\Sigma(X)$.*

A titre d'illustration, considérons de nouveau le problème des pigeons et le réseau de contraintes P_{n_3} de la figure 3.3. Soit P le réseau de contraintes initial associé au problème des pigeons, d'après les définitions précédentes nous avons alors :

- $\Sigma = \{(X_2, D_2), (X_3, D_3), (X_4, D_4))\}$ avec $D_2 = D_3 = D_4 = \{2, 3\}$ est un état partiel de P_{n_3} ;
- $\psi(P, \Sigma)$ est un CN obtenu à partir de P tel que $dom(X_0) = dom(X_1) = \{0, 1, 2, 3\}$ et $dom(X_2) = dom(X_3) = dom(X_4) = \{2, 3\}$;
- Σ est un état partiel inconsistant de P car $\psi(P, \Sigma)$ est un réseau de contraintes insatisfaisable ;
- Σ domine P_{n_6} puisque $vars(\Sigma) \subseteq vars(P_{n_6})$ et $\forall X \in \Sigma, dom^{P_{n_6}}(X) \subseteq dom^\Sigma(X)$.

La proposition suivante est au centre du raisonnement basé sur les états par détection de dominance.

Proposition 16. *Soient P et P' deux réseaux de contraintes tels que $P' \preceq P$, et Σ un IPS_P. Si Σ domine P', P' est insatisfaisable.*

Démonstration. La preuve est immédiate puisqu'on peut facilement déduire que $\psi(P', \Sigma) \preceq \psi(P, \Sigma)$ à partir du fait que $P' \preceq P$ et de la définition de ψ. □

Dans le contexte de la résolution d'un réseau de contraintes P par un algorithme de recherche avec retours-arrière, cette proposition peut être exploitée pour éliminer des nœuds dominés par des IPS_P précédemment identifiés. Un IPS_P peut être extrait d'un nœud prouvé comme étant la racine d'un sous-arbre insatisfaisable. Bien que l'état courant associé à un tel nœud soit lui même un IPS_P, celui-ci ne peut bien évidemment pas être rencontré plus tard au cours de la recherche : pour être utile, il doit être réduit. C'est pourquoi, dans ce qui suit, des opérateurs de réduction sont proposés. Ces opérateurs éliminent certaines variables d'un état prouvé insatisfaisable tout en préservant son status d'IPS_P.

Le concept d'état partiel (inconsistant) est à mettre en relation avec celui de "Global Cut Seed" [Focacci et Milano, 2001] et de "pattern" [Fahle *et al.*, 2001]. L'une des différences principales est qu'un état partiel peut être défini à partir d'un sous-ensemble des variables d'un réseau de contraintes tandis que "Global Cut Seed" et "pattern", introduits pour casser des symétries globales, contiennent toutes les variables du réseau de contraintes.

D'un autre coté, le concept d'état partiel inconsistant est similaire à celui de nogood généralisé introduit dans [Katsirelos et Bacchus, 2005]. Un état partiel inconsistant permet de représenter de façon compacte un ensemble de conflits entre des variables en réduisant les domaines de ces variables aux valeurs incompatibles entre elles, i.e. seules les valeurs provoquant un conflit sont représentées dans un IPS. L'utilisation de décisions négatives dans la représentation des nogoods permet également de réduire les domaines des variables (impliquées dans un nogood) aux valeurs incompatibles. Pour cela, il suffit d'interdire pour les variables du nogood les valeurs du domaine initial ne provoquant pas (éventuellement) de conflits, i.e. les valeurs compatibles sont représentées sous la forme de décisions négatives dans le nogood. Considérons par exemple deux variables X et Y dont le domaine initial comporte 4 valeurs, $dom(X) = dom(Y) = \{a, b, c, d\}$. A un nœud donné d'une recherche arborescente, prouvé comme étant la racine d'un sous-arbre inconsistant, le domaine de ces variables est réduit aux valeurs $\{a, b\}$. On peut donc extraire l'état partiel inconsistant associé à ce nœud : $\Sigma = \{(X, D_X), (Y, D_Y)\}$ avec $D_X = D_y = \{a, b\}$.

Cependant cet état partiel peut également être représenté par un nogood généralisé en interdisant les valeurs (éventuellement) compatibles du domaine initial. Ainsi Σ est équivalent au nogood généralisé $\Delta = \{X \neq c, X \neq d, Y \neq c, Y \neq d\}$.

Ces deux représentations sont tout à fait équivalentes[7]. Cependant dans notre contexte, où un état partiel correspond à "l'état des variables" (i.e. aux domaines des variables) du sous-réseau associé à un nœud donné de l'arbre de recherche, il semble plus naturel de manipuler des "états partiels" plutôt que des nogoods généralisés. De plus, notre approche est très différente de celle présentée dans [Katsirelos et Bacchus, 2005] puisque les états partiels sont extraits ici à l'aide d'opérateurs originaux.

Il est possible de construire un état partiel en supprimant des variables qu'elles soient impliquées ou non dans les décisions prises. De tels états partiels devraient être idéalement aussi généraux que possible, et ne consommer qu'une petite quantité de mémoire. Les opérateurs présentés dans les sections suivantes ont pour objectif de minimiser le nombre de variables enregistrées. Intuitivement étant donné un état partiel, plus le nombre de variables éliminées est important, meilleure est la capacité d'élagage de cet état. Cela contribue également à réduire la quantité de mémoire nécessaire pour stocker ces états partiels.

3.2.1 Opérateurs basés sur l'universalité

Préserver les solutions (opérateur ρ^{sol})

Le premier opérateur, noté ρ^{sol}, extrait des états partiels préservant l'ensemble des solutions d'un réseau de contraintes donné. Plus précisément si deux réseaux P_1 et P_2 isssus respectivement de deux nœuds μ_1 et μ_2 d'un arbre de recherche sont réduits au même état partiel par l'opérateur ρ^{sol}, alors les solutions apparaissant dans le sous-arbre dont μ_1 est la racine sont les mêmes que celles apparaissant dans le sous-arbre où μ_2 est la racine. Appliqué sur un réseau, cet opérateur permet de supprimer les variables appelées *s-éliminables* qui possèdent un domaine singleton et n'apparaissent que dans des contraintes universelles.

Définition 34 (variables s-éliminables)**.** *Soit $P = (\mathscr{X}, \mathscr{C})$ un CN. Une variable $X \in \mathscr{X}$ est s-éliminable ssi $|dom(X)| = 1$ et $\forall C \in \mathscr{C} \mid X \in scp(C)$, C est universelle. L'ensemble des variables s-éliminables de P est noté $S_{elim}(P)$.*

L'opérateur ρ^{sol} représente l'état partiel extrait en supprimant toutes les variables s-éliminables d'un réseau de contraintes.

Définition 35 (opérateur ρ^{sol})**.** *Soit P un CN. L'opérateur $\rho^{sol}(P)$ représente l'état partiel $\Sigma = \{(X, dom^P(X)) \mid X \in \mathscr{X} \setminus S_{elim}(P)\}$.*

Etant donné un état partiel Σ extrait d'un réseau $P = (\mathscr{X}, \mathscr{C})$, on peut construire le réseau de contraintes $P^\Sigma = (\mathscr{X}', \mathscr{C}')$ tel que $\mathscr{X}' = vars(\Sigma)$ et $\mathscr{C}' = \{C \in \mathscr{C} \mid scp(C) \subseteq vars(\Sigma)\}$. Autrement dit, le réseau de contraintes P^Σ est construit à partir du réseau P en supprimant les variables de P n'apparaissant pas dans $vars(\Sigma)$ et les contraintes de P dont la portée implique des variables n'apparaissant pas dans $vars(\Sigma)$.

Etant donné un état partiel $\Sigma = \rho^{sol}(P)$, les solutions de P peuvent être énumérées en étendant les solutions du réseau P^Σ avec l'interprétation construite à partir des variables s-éliminables. En effet, le domaine des variables éliminées est singleton, et les contraintes supprimées n'ont plus d'impact sur le réseau P.

[7] Le concept de nogood et d'état partiel étant de prime abord assez éloigné l'un de l'autre, l'équivalence entre ces deux notions ne nous a pas paru évidente immédiatement.

Proposition 17. *Soient P un CN, $\Sigma = \rho^{sol}(P)$ et $t = \{(X, v) \mid X \in S_{elim}(P) \wedge dom(X) = \{v\}\}$. On a alors $sol(P) = \{s \cup t \mid s \in sol(P^{\Sigma})\}$.*

Démonstration. Toute solution de P satisfait également les contraintes de P^{Σ}. La réciproque est immédiate : toute solution de P^{Σ} peut être étendue à une unique solution de P puisque les variables éliminées (de $S_{elim}(P)$) ont un domaine singleton et les contraintes éliminées sont universelles. □

Préserver la satisfaisabilité (opérateur ρ^{uni})

L'opérateur ρ^{uni} est une généralisation de l'opérateur ρ^{sol}, puisque les variables s-éliminables sont également des variables u-éliminables (définies ci-dessous). Il préserve la satisfaisabilité d'un réseau de contraintes donné mais pas nécessairement toutes les solutions. Plus précisément si un état partiel, associé à un nœud de la recherche racine d'un sous-arbre insatisfaisable et extrait avec l'opérateur ρ^{uni}, est rencontré une nouvelle fois (à partir d'un nœud différent de l'arbre de recherche) alors ce nœud est également la racine d'un sous-arbre insatisfaisable.

Définition 36 (variables u-éliminables)**.** *Soit $P = (\mathscr{X}, \mathscr{C})$ un réseau de contraintes. Une variable $X \in \mathscr{X}$ est une variable u-éliminable de P ssi $\forall C \in \mathscr{C} \mid X \in scp(C)$, C est universelle. L'ensemble des variables u-éliminables d'un réseau de contraintes P est noté $U_{elim}(P)$.*

L'opérateur ρ^{uni} représente l'état partiel extrait en supprimant toutes les variables u-éliminables d'un réseau de contraintes.

Définition 37 (opérateur ρ^{uni})**.** *Soit P un CN. L'opérateur $\rho^{uni}(P)$ représente l'état partiel $\Sigma = \{(X, dom^P(X)) \mid X \in \mathscr{X} \setminus U_{elim}(P)\}$.*

La proposition suivante établit le fait que l'opérateur ρ^{uni} peut extraire des états partiels inconsistants à chaque nœud d'un arbre de recherche, prouvé comme étant la racine d'un sous-arbre insatisfaisable.

Proposition 18. *Soient P et P' deux réseaux de contraintes tels que $P' \preceq P$. Si P' est insatisfaisable alors $\rho^{uni}(P')$ est un IPS_P.*

Démonstration. Soit $\Sigma = \rho^{uni}(P')$. Par l'absurde, si Σ n'est pas un IPS_P, alors le réseau de contraintes $\psi(P, \Sigma)$ est satisfaisable. Il existe alors au moins une solution à ce réseau et donc au moins une valeur pour chaque variable satisfaisant toutes les contraintes de P. Par conséquent, il existe au moins une valeur assignée aux variables u-éliminables compatible (ayant un support avec les autres variables –non u-éliminables–). Comme ces variables n'apparaissent que dans des contraintes universelles, il existerait alors une solution pour le sous-réseau P' ce qui contredit l'hypothèse : P' est insatisfaisable. □

Cet opérateur permet notamment de détecter une certaine forme de symétrie : l'interchangeabilité au voisinage (neighborhood interchangeability) [Freuder, 1991]. Deux valeurs du domaine d'une même variable X sont interchangeables au voisinage si elles possèdent les mêmes supports dans toutes les contraintes impliquant X.

Définition 38 (interchangeabilité au voisinage)**.** *Soient $P = (\mathscr{X}, \mathscr{C})$ un CN et $X \in \mathscr{X}$. Deux valeurs a et $b \in dom(X)$ sont interchangeables au voisinage si et seulement si $\forall C \in \mathscr{C}$ telle que $X \in scp(C)$, $\{t_a \setminus (X, a) \mid t_a \in rel(C) \wedge (X, a) \in t_a\} = \{t_b \setminus (X, b) \mid t_b \in rel(C) \wedge (X, b) \in t_b\}$.*

La figure 3.4 illustre sur un exemple l'utilisation de l'opérateur ρ^{uni}. Les valeurs 1 et 2 du domaine de la variable W sont interchangeables. En effet W n'est relié au reste du réseau de contraintes que par une seule contrainte C_0 (reliant la variable W à la variable X) et l'ensemble des supports de la valeur 1 de W sur X est égal à l'ensemble des supports de la valeur 2 de W sur X. Les sous-réseaux obtenus après avoir assigné la valeur 1 ou la valeur 2 à la variable W sont identiques. Lorsqu'on assigne la valeur 1 à la variable W, figure 3.5 (a), le sous-réseau obtenu est arc-inconsistant et l'on peut extraire alors un IPS_P en appliquant l'opérateur ρ^{uni} : $\rho^{uni}(P|_{W=1}) = \{(X, \{1,2\}), (Y, \{1,2\}), (Z, \{1,2\})\}$. Si plus tard au cours de la recherche (figure 3.5 (b)), on assigne cette fois la valeur 2 à la variable W, après application de l'opérateur ρ^{uni} on pourra détecter que $\rho^{uni}(P|_{W=1}) = \rho^{uni}(P|_{W=2})$. L'exploration du sous-arbre associé au réseau $P|_{W=2}$ pourra donc être évitée d'après la proposition 16.

3.2.2 L'opérateur ρ^{red}

Description

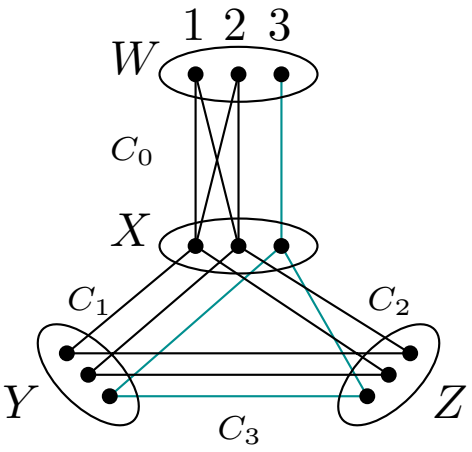
Cet opérateur extrait un IPS en éliminant à la fois les variables u-éliminables et les variables r-éliminables. Ces dernières correspondent aux variables dont le domaine est resté inchangé après avoir pris un ensemble de décisions et appliqué un opérateur d'inférence établissant une consistance ϕ.

Définition 39 (variables r-éliminables)**.** *Soient P et P' deux réseaux de contraintes tels que $P' \preceq P$. Une variable $X \in vars(P')$ est r-éliminable dans P' vis-à-vis de P ssi $dom^{P'}(X) = dom^{P}(X)$. L'ensemble des variables r-éliminables de P' est noté $R^P_{elim}(P')$.*

Définition 40 (opérateur ρ^{red})**.** *Soient P et P' deux réseaux de contraintes tels que $P' \preceq P$. L'opérateur $\rho^{red}(P')$ extrait un état partiel $\Sigma = \{(X, dom^{P'}(X)) \mid X \in vars(P') \setminus (R^P_{elim}(P') \cup U_{elim}(P'))\}$.*

La proposition suivante établit le fait que l'opérateur ρ^{red} peut extraire des états partiels inconsistants à chaque nœud d'un arbre de recherche, prouvé comme étant la racine d'un sous-arbre insatisfaisable.

Proposition 19. *Soient P et P' deux réseaux de contraintes tels que $P' \preceq P$. Si P' est insatisfaisable alors $\Sigma = \rho^{red}(P')$ est un IPS_P.*



$P = (\mathscr{X}, \mathscr{C})$:
- $\mathscr{X} = \{W, X, Y, Z\}$ avec $dom(W) = dom(X) = dom(Y) = dom(Z) = \{1,2,3\}$;
- $\mathscr{C} = \left\{ \begin{array}{ll} C_0 : & \langle W, X\rangle \in \{(1,1),(1,2),(2,1),(2,2),(3,3)\} \\ C_1 : & \langle X, Y\rangle \in \{(1,1),(2,2),(3,3)\} \\ C_2 : & \langle X, Z\rangle \in \{(1,2),(2,1),(3,3)\} \\ C_3 : & \langle Y, Z\rangle \in \{(1,1),(2,2),(3,3)\} \end{array} \right\}$.

FIG. 3.4 – Elimination de valeurs interchangeables

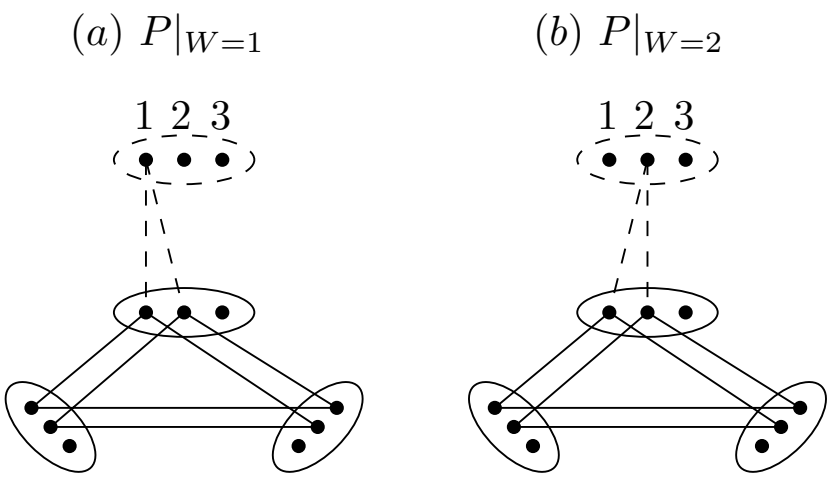


FIG. 3.5 – Sous-réseaux de contraintes obtenus après l'assignation $W = 1$ et $W = 2$

Démonstration. Soit $\Sigma' = \rho^{uni}(P')$, on a $\psi(P, \Sigma) = \psi(P, \Sigma')$ puisque :
- dans $\psi(P, \Sigma')$ le domaine de chaque variable $X \in U_{elim}(P')$ est remplacé par $dom^P(X)$;
- dans $\psi(P, \Sigma)$ le domaine de chaque variable $X \in U_{elim}(P')$ est remplacé par $dom^P(X)$ et le domaine de chaque variable $X \in R^P_{elim}(P')$ reste identique.

On obtient donc $\psi(P, \Sigma) = \psi(P, \Sigma')$, et d'après la proposition 18, Σ et Σ' sont des IPS_P. □

Dans le cas d'un réseau de contraintes binaires, l'opérateur ρ^{red} extrait un état partiel en sélectionnant les variables (et le domaine associé) dont le domaine est non singleton et ne correspond pas au domaine initial.

Exemple

Dans le contexte de la vérification de la satisfaisabilité d'une instance CSP, les capacités d'élagage des opérateurs ρ^{red} et ρ^{uni} ne sont pas équivalentes. Clairement plus la taille de l'IPS extrait par un opérateur est petite (i.e. plus le nombre de variables supprimées par l'opérateur est important), plus cet IPS pourra être utilisé pour élaguer des branches de l'arbre de recherche. Il est alors aisé de voir que $vars(\rho^{red}(P)) \subseteq vars(\rho^{uni}(P)) \subseteq vars(\rho^{sol}(P))$.

La figure 3.6 illustre les capacités d'élagage de l'opérateur ρ^{red} par rapport à l'opérateur ρ^{uni}. Le réseau de contraintes $P = (\mathscr{X}, \mathscr{C})$ est défini par :
- $\mathscr{X} = \{W, X, Y, Z\}$ et $dom(W) = dom(X) = dom(Y) = dom(Z) = \{1, 2, 3\}$;
- $\mathscr{C} = \{W + 1 \neq X, W + 1 \neq Z, Y + 1 \neq X, Y + 1 \neq Z\}$.

Sur la figure (a), l'assignation $Y = 1$ sur le réseau initial P laisse le domaine de la variable W inchangé et élimine la valeur 2 du domaine des variables X et Z, ce qui mène au réseau $P_1 = AC(P|_{Y=1})$. Sur la figure (b), l'assignation $W = 1$ sur le réseau initial P laisse le domaine de la variable Y inchangé et élimine la valeur 2 du domaine des variables X et Z, ce qui mène au réseau $P_2 = AC(P|_{W=1})$. Alors que l'opérateur ρ^{uni} produit des états partiels différents à partir de P_1 et P_2, l'opérateur ρ^{red} appliqué sur ces deux réseaux mène au même état partiel, quelque soit la raison de l'élimination d'une variable (u-éliminable ou r-éliminable). Par conséquent, d'après la proposition 19, une fois que P_1 ou P_2 a été exploré et démontré insatisfaisable, un IPS_P peut être extrait à partir de ce réseau et l'exploration du second est inutile pour déterminer la satisfaisabilité du réseau initial P.

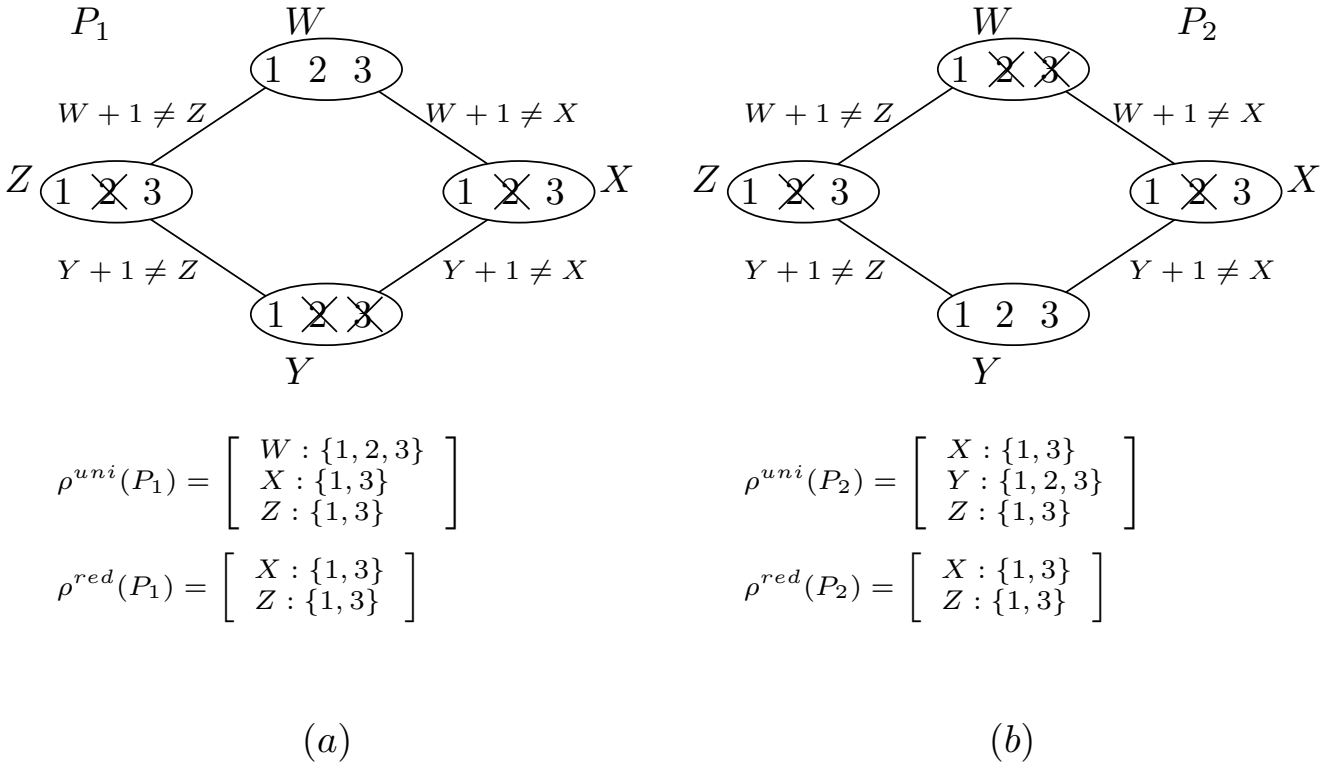


FIG. 3.6 – Capacité d'élagage des opérateurs ρ^{red} et ρ^{uni}

3.2.3 Extraction basée sur une preuve : l'opérateur ρ^{prf}

Toutes les contraintes d'un réseau de contraintes insatisfaisable ne sont pas forcément nécessaires pour prouver son insatisfaisabilité. Certaines d'entre elles constituent un noyau (minimal) insatisfaisable (Minimal Unsatisfiable Cores ou MUCs) et différentes méthodes ont été proposées pour les extraire. En effet, on peut itérativement identifier les contraintes d'un MUC en suivant une approche constructive [de Siqueira et Puget, 1988], destructive [Baker *et al.*, 1993] ou encore une approche dichotomique [Junker, 2004, Hemery *et al.*, 2006]. Plus généralement un noyau insatisfaisable peut être défini comme suit :

Définition 41 (noyau insatisfaisable d'un réseau)**.** *Soient $P = (\mathscr{X}, \mathscr{C})$ et $P' = (\mathscr{X}', \mathscr{C}')$ deux réseaux de contraintes. P' est un noyau insatisfaisable de P si P' est insatisfaisable, $\mathscr{X}' \subseteq \mathscr{X}$, $\mathscr{C}' \subseteq \mathscr{C}$ et $\forall X' \in \mathscr{X}', dom^{P'}(X') = dom^{P}(X')$.*

De façon intéressante, on peut tout à fait associer un IPS_P à tout noyau insatisfaisable extrait d'un réseau $P' \preceq P$. Ceci est introduit plus formellement par la proposition suivante.

Proposition 20. *Soient P et P' deux réseaux de contraintes tels que $P' \preceq P$. Pour tout noyau insatisfaisable P'' de P', $\Sigma = \{(X, dom^{P''}(X)) \mid X \in vars(P'')\}$ est un IPS_P.*

Démonstration. Si Σ n'est pas un IPS_P, i.e. si $\psi(P, \Sigma)$ est satisfaisable, il existe au moins une assignation d'une valeur à toutes les variables de $vars(\Sigma)$ telle que toutes les contraintes de P sont satisfaites. Comme toutes les contraintes de P'' sont incluses dans celles de P', et donc dans celles de P, cela contredit l'hypothèse initiale que P'' est un noyau insatisfaisable. □

Puisqu'un état partiel inconsistant peut directement être dérivé d'un noyau insatisfaisable, on peut s'intéresser à extraire de tels noyaux à partir de chaque nœud prouvé comme étant la racine d'un sous-arbre insatisfaisable. Calculer un MUC en utilisant l'une des approches mentionnées précédemment à chaque nœud sans profiter du sous-arbre exploré semble très coûteux car même l'approche dichotomique est en $O(log(e).K_e)$ [Hemery *et al.*, 2006], où K_e correspond au nombre de contraintes du noyau extrait. Cependant, il est possible d'identifier efficacement un noyau insatisfaisable, en gardant la trace des contraintes impliquées dans la preuve d'insatisfaisabilité [Baker *et al.*, 1993]. Pour cela, on suppose que les inférences sont effectuées localement, i.e. au niveau d'une seule contrainte C, pendant la propagation effectuée lorsqu'on établit une consistance ϕ[8]. L'opérateur d'inférence établissant ϕ peut être vu comme une collection de propagateurs locaux associés à chaque contrainte, et appelés ϕ-propagateurs. Ces propagateurs peuvent correspondre soit à la procédure générique de révision d'un algorithme GAC à gros grain appelée pour une contrainte (c.f. section 1.2.1), ou alors à une procédure de filtrage spécialisée (e.g. dans le contexte de contraintes globales). Ces contraintes, dont on a gardé la trace, sont celles utilisées pendant la recherche pour supprimer à travers leurs propagateurs, au moins une valeur dans le domaine d'une variable. Cette approche "basée sur la preuve" a été adaptée pour extraire des noyaux insatisfaisables à partir des nœuds de l'arbre de recherche en collectant incrémentalement les informations utiles.

Algorithme 21 : *solve*

Entrées : $P_{init} = (\mathscr{X}, \mathscr{C})$: CN
Sorties : Booléen, Ensemble de variables

1 $localProof \leftarrow \emptyset$;
2 $P = \phi(P_{init})$ // localProof mise a jour d'après ϕ ;
3 **si** $P = \bot$ **alors retourner** $(faux, localProof)$;
4 **si** $\forall X \in \mathscr{X}, |dom(X)| = 1$ **alors retourner** $(vrai, \emptyset)$;
5 sélectionner une paire (X, a) avec $|dom(X)| > 1 \wedge a \in dom(X)$;
6 $(sat, leftProof) \leftarrow solve(P|_{X=a})$;
7 **si** sat **alors retourner** $(vrai, \emptyset)$;
8 $(sat, rightProof) \leftarrow solve(P|_{X \neq a})$;
9 **si** sat **alors retourner** $(vrai, \emptyset)$;
10 // $leftProof \cup rightProof$ est une preuve d'inconsistance pour P ;
11 **retourner** $(faux, localProof \cup leftProof \cup rightProof)$

L'algorithme 21 décrit comment implémenter cette méthode au sein d'un algorithme de recherche avec retours-arrière embarquant un opérateur d'inférence établissant une consistance ϕ à chaque nœud de l'arbre de recherche. La fonction récursive *solve* détermine la satisfaisabilité d'un réseau de contraintes P_{init} et renvoie une paire composée d'une valeur booléenne (indiquant si P_{init} est satisfaisable ou pas), et d'un ensemble de variables. Cet ensemble est vide (lorsque P_{init} est satisfaisable) ou alors représente la preuve de l'insatisfaisabilité. La preuve ($localProof$) est composée des variables impliquées dans la portée des contraintes qui ont déclenché au moins une suppression pendant la propagation. Notons qu'en pratique, comme les IPS_P correspondent à un sous-ensemble de variables (et leur domaine), $localProof$ ne correspond pas à un ensemble de contraintes, dont on a gardé la trace (à chaque suppression d'une valeur lorsqu'on établit ϕ) mais directement à un ensemble de variables impliquées dans celles-ci.

[8] On suppose que l'opérateur établissant ϕ contrôle au moins que l'instanciation courante est consistante afin de garantir la minimalité du noyau

A chaque nœud, une preuve, initialement vide, est construite à partir de toutes les inférences produites lors de l'application de ϕ et des preuves (ligne 6 et 8) associées aux sous-arbres de gauche et de droite (une fois qu'un couple (X, a) a été sélectionné). Lorsque l'insatisfaisabilité d'un nœud est prouvée, c'est-à-dire après avoir évalué les deux branches (la première identifiée par l'assignation $X = a$ et la seconde par la réfutation $X \neq a$), on obtient une preuve d'insatisfaisabilité (ligne 10) en fusionnant les preuves associées aux branches de gauche et de droite. On peut noter que la complexité spatiale, dans le pire des cas, pour exploiter les différentes preuves locales de l'arbre de recherche est $O(n^2d)$ puisqu'enregistrer une preuve est en $O(n)$ et qu'il y a au plus $O(nd)$ nœuds par branche.

En utilisant l'algorithme 21, on peut introduire un nouvel opérateur d'extraction qui ne sélectionne que les variables impliquées dans une preuve d'insatisfaisabilité. Cet opérateur peut être incrémentalement utilisé à chaque nœud de l'arbre de recherche, démontré comme étant la racine d'un sous-arbre insatisfaisable.

Définition 42 (opérateur ρ^{prf})**.** *Soit P un réseau de contraintes tel que $solve(P) = (faux, proof)$. L'opérateur $\rho^{prf}(P)$ représente l'état partiel $\Sigma = \{(X, dom^P(X)) | X \in proof\}$. Une variable $X \in vars(P) \setminus vars(\Sigma)$ est appelée variable p-éliminable de P.*

Proposition 21. *Soient P et P' deux réseaux de contraintes tels que $P' \preceq P$. Si P' est insatisfaisable alors $\rho^{prf}(P')$ est un IPS_P.*

Démonstration. Soient $P = (\mathscr{X}, \mathscr{C})$ et $solve(P') = (faux, proof)$. Clairement, $P'' = (proof, \{C \in \mathscr{C} | scp(C) \subseteq proof\})$ est un noyau insatisfaisable de P'. $\rho^{prf}(P')$ est égal à $\{(X, dom^{P'}(X)) | X \in proof\}$, démontré comme étant un IPS_P grâce à la proposition 20. □

En pratique, on peut appeler l'opérateur ρ^{prf} afin d'extraire un IPS_P à la ligne 10 de l'algorithme 21. De façon intéressante, la proposition suivante établit le fait que l'opérateur ρ^{prf} est plus fort que l'opérateur ρ^{uni} (i.e. permet d'extraire des états partiels représentant de plus grandes portions de l'espace de recherche).

Proposition 22. *Soit P un réseau de contraintes insatisfaisable. $\rho^{prf}(P) \subseteq \rho^{uni}(P)$.*

Démonstration. Une contrainte universelle ne peut apparaître dans aucune preuve d'insatisfaisabilité puisqu'elle ne sert jamais à déclencher une suppression de valeur. Une variable u-éliminable apparaît seulement dans des contraintes universelles, donc c'est aussi une variable p-éliminable. □

Il est important de remarquer que contrairement à l'opérateur ρ^{uni}, extraire un état partiel inconsistant en utilisant l'opérateur ρ^{prf} ne peut s'effectuer que lorsque le sous-arbre a été complètement exploré. Par conséquent, il n'est pas possible d'utiliser cet opérateur pour élaguer des états équivalents en utilisant une table de transposition dont la clef correspond aux états partiels. Néanmoins, l'opérateur ρ^{prf} peut être exploité dans le contexte de la détection de dominance. La section 3.3 traite plus spécifiquement de l'exploitation des états partiels inconsistants que ce soit en terme d'équivalence ou en terme de dominance.

3.2.4 L'opérateur ρ^{exp}

Contrairement à l'opérateur d'extraction basé sur la preuve, celui-ci, basé sur des explications, peut être appliqué à chaque fois que l'on atteint un nouveau nœud de l'arbre de recherche en analysant toutes les propagations effectuées auparavant. Le principe de cet opérateur est de construire un état partiel en éliminant les variables dont le domaine peut être inféré à partir des

autres. Ceci est rendu possible en gardant la trace, pour chaque valeur supprimée du réseau initial, de la contrainte à l'origine de la suppression. Ces sortes d'explications sont à mettre en relation avec le concept de graphe d'implication utilisé au sein de la communauté SAT. Lorsqu'on tente d'établir la consistance d'arc pour les CSPs dynamiques [Bessiere, 1991], de telles explications sont également exploitées lorsqu'on revient sur des contraintes pour restaurer les valeurs des domaines.

En satisfaction de contraintes, les explications d'une suppression sont classiquement basées sur les décisions, ce qui veut dire que chaque valeur éliminée est expliquée par un ensemble de décisions positives, i.e. un ensemble de variables assignées. Celles-ci sont souvent exploitées pour effectuer une forme de retour-arrière intelligent (e.g. [Dechter et Frost, 2002, Prosser, 1993, Ginsberg, 1993]). De manière intéressante, il est possible de définir des explications dans un cadre plus général en prenant en considération non seulement les décisions prises au cours de la recherche mais aussi certaines contraintes du réseau initial.

Les explications enregistrées pour chaque suppression peuvent être représentées sous la forme d'un graphe d'implication comme c'est utilisé dans le contexte de la satisfaisabilité [Zhang *et al.*, 2001]. Etant donné un ensemble de décisions (l'instanciation partielle courante), le processus d'inférence peut être modélisé en utilisant un graphe d'implication à grain fin. Plus précisément, pour chaque valeur supprimée, on peut enregistrer les décisions positives et négatives impliquant cette suppression (au travers d'une clause). Dans notre cas, on peut simplement raisonner en utilisant un graphe d'implication à gros grain. Chaque fois qu'une valeur (X, a) est supprimée pendant la propagation associée à une contrainte C, l'explication (locale) de l'élimination de (X, a) est fournie simplement par C. En d'autres termes, comme l'objectif est de délimiter un état partiel (plus petit que l'état courant en terme du nombre de variables), il est juste suffisant de connaître pour chaque valeur (X, a) supprimée les variables responsables de cette suppression, c'est-à-dire les variables impliquées dans C. A partir de cette information, on peut construire un graphe dirigé G où les nœuds correspondent aux variables et les arcs aux dépendances entre les variables. Plus précisément, un arc existe dans G entre une variable Y et une seconde variable X s'il existe une valeur supprimée (X, a) telle que son explication locale est une contrainte impliquant Y. De plus, un nœud particulier étiqueté par *nil* est ajouté au graphe, et un arc existe entre *nil* et une variable X si cette variable est impliquée dans une décision (positive ou négative). Ce graphe d'implication peut alors être utilisé pour extraire un état partiel inconsistant à partir d'un sous-ensemble de variables S (qui représente déjà un état partiel inconsistant), en éliminant toute variable n'ayant pas d'arc entrant à partir d'une variable ne faisant pas partie de l'ensemble S. Naturellement, une telle extraction n'est pas intéressante si S est égal à $\mathscr{X}$ puisque dans ce cas l'extraction produira nécessairement un ensemble de variables correspondant à toutes les décisions.

Important : Pour toutes les définitions et propositions suivantes, on considérera donnés deux réseaux de contraintes $P = (\mathscr{X}, \mathscr{C})$ et P' tel que P' correspond à un nœud d'un arbre de recherche issu de P (une consistance ϕ étant maintenue). Bien évidemment $P' \preceq P$.

Définition 43 (explication locale)**.** *Pour tout couple (X, a) tel que $X \in \mathscr{X}$ et $a \in dom^P(X) \setminus dom^{P'}(X)$, l'explication locale de l'élimination de (X, a), notée $exp(X, a)$ est, si elle existe, la contrainte C dont le ϕ-propagateur associé a supprimé (X, a) au cours du chemin menant de P à P', et nil dans le cas contraire.*

Ces explications peuvent être utilisées pour extraire un état partiel à partir d'un réseau de contraintes vis-à-vis de P et d'un ensemble de variables S. Cet état partiel contient les variables de S qui ne peuvent pas être "expliquées" par S.

Définition 44 (variables i-éliminables). $\forall S \subseteq \mathscr{X}$, $\rho^{exp}_{P,S}(P')$ *est l'état partiel* $\Sigma = \{(X, dom^{P'}(X)) \mid X \in S$ *et* $\exists a \in dom^{P}(X) \setminus dom^{P'}(X)$ *tel que* $(exp(X,a) = nil$ *ou* $\exists Y \in scp(exp(X,a))$ *telle que* $Y \notin S)\}$. *Une variable* $X \in S \setminus vars(\Sigma)$ *est appelée une variable i-éliminable de* P' *vis-à-vis de* P *et* S.

Proposition 23. *Soit* Σ *un état partiel de* P' *et* $\Sigma' = \rho^{exp}_{P,vars(\Sigma)}(P')$. *Nous avons :* $\phi(\psi(P,\Sigma')) = \phi(\psi(P,\Sigma))$.

Démonstration. Les seules variables dont le domaine peut différer entre $\psi(P,\Sigma')$ et $\psi(P,\Sigma)$ sont les variables i-éliminables de l'ensemble $\Delta = vars(\Sigma) \setminus vars(\Sigma')$. Nous savons également que $\forall X \in vars(\Sigma')$, $dom^{\psi(P,\Sigma')}(X) = dom^{\psi(P,\Sigma)}(X) = dom^{P'}(X)$ et que $\forall X \in \Delta$, $dom^{\psi(P,\Sigma')}(X) = dom^{P}(X)$. Ceci veut dire que les domaines des variables de Σ' sont dans l'état où ils étaient après avoir effectué toutes les décisions et toutes les propagations qui ont mené à P', et que les domaines des variables de Δ sont réinitialisés à l'état dans lequel ils étaient dans P. De plus, toutes les variables $X \in vars(P) \setminus vars(\Sigma)$ sont telles que $dom^{\psi(P,\Sigma')}(X) = dom^{\psi(P,\Sigma)}(X)$.

Soit $R_\Delta = \{(X,a) | X \in \Delta \wedge a \in dom^{P}(X) \setminus dom^{P'}(X)\}$ l'ensemble des valeurs supprimées pour les variables de Δ sur la branche menant de P à P'. Pour tout $(X,a) \in R_\Delta$, nous avons un explication $C = exp(X,a)$ telle que $C \neq nil$ et $scp(C) \subseteq vars(\Sigma)$ puisque X est une variable i-éliminable. En d'autres termes, les suppressions des valeurs de R_Δ ont été déclenchées le long du chemin menant à P' par les contraintes (les explications) seulement impliquées dans les variables de Σ, c'est-à-dire les variables de Σ' et Δ lui même.

Dans $\psi(P,\Sigma')$, on peut déclencher la suppression de toutes les valeurs de R_Δ dans l'ordre dans lequel elles ont été effectuées le long du chemin menant à P'. En effet, suivant le même ordre, l'explication associée à chaque valeur de R_Δ peut déclencher une nouvelle fois sa suppression puisque (1) les domaines des variables de Σ' sont gardés après leur réduction dans la branche menant à P' ($\forall X \in \Sigma', dom^{\psi(P,\Sigma')}(X) = dom^{P'}(X)$), (2) les variables de Δ sont réinitialisées à leur état dans P ($\forall X \in \Delta, dom^{\psi(P,\Sigma')}(X) = dom^{P}(X)$) et (3) l'explication de n'importe quelle suppression implique seulement les variables de Σ. Ceci peut être montré par une récurrence sur l'ordre des valeurs qui sont supprimées dans la branche menant à P'. Finalement, comme les valeurs supprimées représentent seulement la différence entre $\psi(P,\Sigma')$ et $\psi(P,\Sigma)$, par confluence de l'opérateur établissant la consistance ϕ, on peut conclure que $\phi(\psi(P,\Sigma')) = \phi(\psi(P,\Sigma))$. □

Le corollaire suivant (dont la preuve est une conséquence directe de la proposition 23) est particulièrement intéressant puisqu'il indique que l'on peut de manière sûre, utiliser l'opérateur ρ^{exp} après n'importe quel autre opérateur produisant un IPS_P.

Corollaire 6. *Soit* Σ *un état partiel de* P' *et* $\Sigma' = \rho^{exp}_{P,vars(\Sigma)}(P')$. *Si* Σ *est un* IPS_P *alors* Σ' *est un* IPS_P.

Ceci nous assure donc que les deux opérateurs suivants produisent également des IPS_P.

Définition 45 (opérateurs ρ^{prex} et ρ^{unex}).
- $\rho^{prex}_P(P') = \rho^{exp}_{P,vars(\Sigma)}(P')$ *avec* $\Sigma = \rho^{prf}(P')$;
- $\rho^{unex}_P(P') = \rho^{exp}_{P,vars(\Sigma)}(P')$ *avec* $\Sigma = \rho^{uni}(P')$.

L'opérateur ρ^{exp} peut être implémenté à l'aide d'un tableau à deux dimensions exp tel que pour chaque paire (X,a) supprimée de P pendant la recherche, $exp[X,a]$ représente l'explication locale de son élimination. Lorsqu'une décision positive $X = a$ est prise, $exp[X,b] \leftarrow nil$ pour toute valeur restante $b \in dom(X) | b \neq a$, et lorsqu'une décision négative $X \neq a$ est prise, $exp[X,a] \leftarrow nil$. La complexité spatiale de la structure de données exp est $O(nd)$ tandis que la

complexité temporelle pour l'exploitation de cette structure est $O(1)$ à chaque fois qu'une valeur est supprimée ou restaurée pendant la recherche. La complexité temporelle dans le pire des cas de l'opérateur ρ^{exp} est $O(ndr)$ où r indique l'arité de la plus grande contrainte. En effet il y a au plus $O(nd)$ valeurs supprimées qui admettent une explication locale pour leur élimination.

La figure 3.7 illustre sur des états partiels consistants le comportement de l'opérateur ρ^{uni}, ρ^{exp} et de leur combinaison ρ^{unex}. Le problème initial implique quatre variables (X, Y, Z, W) et trois contraintes ($X \neq Y$, $Y \geq Z$, $Y \leq W$). Lorsque la décision $X = 3$ est prise, l'explication associée à la suppression des valeurs 1 et 2 du domaine de la variable X est mise à *nil*. Ces suppressions sont propagées à la variable Y au travers de la contrainte $X \neq Y$, menant à la suppression de la valeur 3 du domaine de Y. L'explication de la suppression de cette valeur est alors mise à jour et vaut $X \neq Y$. Cette suppression est également propagée aux variables Z et W : la valeur 3 est supprimée du domaine de Z à travers la propagation de la contrainte $Y \geq Z$, ce qui constitue son explication, et la valeur 0 est supprimée du domaine de W à travers la propagation de la contrainte $Y \leq W$. Plus aucune propagation n'est possible et le réseau de contraintes obtenu est noté P'. Le graphe de dépendance exploité plus tard par l'opérateur ρ^{exp} est alors construit à partir de ces explications.

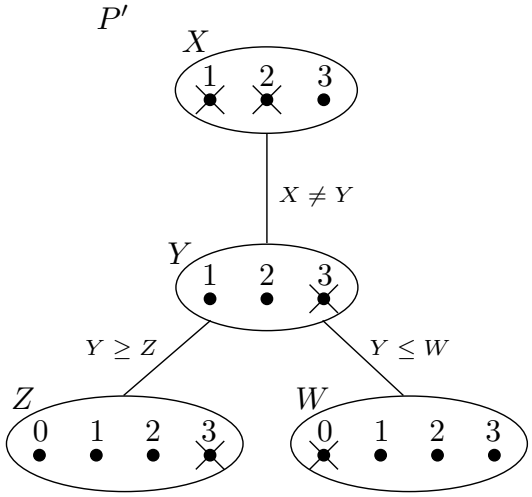
L'application de l'opérateur ρ^{uni} sur le réseau P' mène à l'élimination de la variable X et à l'état partiel Σ_1, puisque X n'apparaît maintenant que dans des contraintes universelles. En effet la valeur restante 3 du domaine de X est compatible avec les deux valeurs restantes 1 et 2 du domaine de Y suivant la contrainte $X \neq Y$. Les trois autres variables sont impliquées dans au moins une contrainte qui n'est pas universelle.

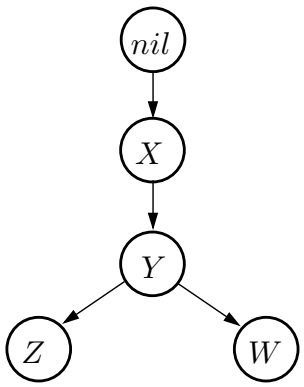
L'application de l'opérateur ρ^{exp} sur le réseau P' avec $S = vars(P)$ mène à l'élimination des variables Y, Z et W. On obtient alors l'état partiel $\Sigma_2 = \{(X, \{3\})\}$ puisque X est la seule variable à partir de laquelle une suppression est expliquée par *nil* (S étant l'ensemble des variables de P). Ceci illustre le fait qu'appliquer l'opérateur ρ^{exp} sur toutes les variables d'un réseau de contraintes n'a pas d'intérêt : on obtient dans ce cas l'ensemble des décisions de la branche courante et l'état partiel ne peut jamais être rencontré (ou dominé) à nouveau sans redémarrage. Notons que nous aurions obtenu ici le même résultat en utilisant des explications classiques basées sur les décisions.

L'application de l'opérateur ρ^{unex} est plus intéressante. Lorsque l'opérateur ρ^{uni} a été appliqué, on obtient l'état partiel Σ_1 composé des variables $\{Y, Z, W\}$. On applique alors l'opérateur ρ^{exp} afin de déterminer quelles sont les variables de Σ_1 dont le domaine peut être déduit par les autres variables de Σ_1. Dans cet exemple, la variable Y est la seule variable pour laquelle toutes les suppressions de valeurs ne peuvent pas être expliquées par des contraintes dont la portée implique uniquement des variables de Σ_1. En effet l'explication $X \neq Y$ de la suppression de la valeur $(Y, 3)$ implique une variable n'appartenant pas à Σ_1. Ceci mène donc à l'état partiel $\Sigma_3 = \{(Y, \{1, 2\})\}$ où la variable Y n'est pas une variable de décision. Cet état pourra donc être exploité ultérieurement au cours de la recherche.

3.3 Exploitation des IPS

Dans le contexte d'un algorithme de recherche embarquant un opérateur d'inférence établissant une consistance ϕ, à chaque nœud associé à un réseau insatisfaisable, on peut utiliser l'un (ou une combinaison) des opérateurs d'extraction présentés précédemment afin d'extraire un état partiel inconsistant. Celui-ci est enregistré dans une base d'IPS_P. Les IPS_P peuvent être exploités plus tard durant la recherche soit pour éviter l'exploration de certains nœuds de l'arbre de recherche, soit pour effectuer des inférences supplémentaires.





Dependency Graph

$$exp(X,1) = nil$$
$$exp(X,2) = nil$$
$$exp(Y,3) = (X \neq Y)$$
$$exp(Z,3) = (Y \geq Z)$$
$$exp(W,0) = (Y \leq W)$$

$$P \rightarrow \left\{ \begin{array}{l} X, \{1,2,3\} \\ Y, \{1,2,3\} \\ Z, \{0,1,2,3\} \\ W, \{0,1,2,3\} \end{array} \right\}$$

$$P' = \phi(P|_{X=3}) \rightarrow \left\{ \begin{array}{l} X, \{3\} \\ Y, \{1,2\} \\ Z, \{0,1,2\} \\ W, \{1,2,3\} \end{array} \right\}$$

$$\Sigma_1 = \rho^{uni}(P') \rightarrow \left\{ \begin{array}{l} Y, \{1,2\} \\ Z, \{0,1,2\} \\ W, \{1,2,3\} \end{array} \right\}$$

$$\Sigma_2 = \rho^{exp}_{P,vars(P)}(P') \rightarrow \{\ X, \{3\}\ \}$$

$$\Sigma_3 = \rho^{unex}_{P}(P') = \rho^{exp}_{P,vars(\Sigma_1)}(P') \rightarrow \{\ Y, \{1,2\}\ \}$$

FIG. 3.7 – Extraction d'états partiels avec les opérateurs ρ^{uni}, ρ^{exp} et ρ^{unex}.

3.3.1 Détection d'équivalence

L'idée principale est d'enregistrer dans une table de transposition tous les IPS_P extraits à partir de nœuds prouvés inconsistants [Lecoutre *et al.*, 2007e]. L'exploitation des nœuds dont l'état partiel inconsistant se trouve déjà dans la table de transposition peut alors être évitée. Tous les opérateurs décrits précédemment ne peuvent cependant pas être employés avec cette méthode. Ceci est notamment vrai en ce qui concerne l'opérateur ρ^{prf}. En effet lorsqu'un nœud est ouvert, l'IPS_P extrait par l'opérateur ρ^{prf} n'est pas encore connu (et par conséquent il est impossible de calculer une clef pour celui-ci pour accéder à la table de transposition). En effet, pour identifier l'IPS_P, cela nécessite l'exploration complète du sous-arbre dont le nœud courant est la racine. La détection d'équivalence à travers l'utilisation d'une table de transposition ne peut donc pas être appliquée avec cet opérateur. Néanmoins un nœud dominé (c.f. section suivante) par un IPS_P enregistré dans la base peut être évité de manière sûre.

L'algorithme présenté ici effectue une recherche en profondeur d'abord et maintient une consistance ϕ en filtrant les domaines (au moins, assurer que les contraintes impliquant des variables avec un domaine singleton sont satisfaites). Cet algorithme élague les états inconsistants grâce à l'opérateur ρ^X, ou ρ^X représente l'opérateur ρ^{uni}, ρ^{red}, ρ^{exp} ou ρ^{unex}. Seul l'opérateur ρ^{sol} nécessite une utilisation particulière (puisqu'il est utilisé pour énumérer toutes les solutions et non pas seulement pour supprimer des branches de l'arbre de recherche, i.e. il n'extrait pas seulement des IPS_P) et les opérateurs basés sur la preuve (ρ^{prf} et ρ^{prex}) qui nécessitent l'exploration totale du sous-arbre pour identifier les IPS_P (c.f. section 3.2.3).

Algorithme 22 : *ρ-solve*

Entrées : $P_{init} = (\mathscr{X}, \mathscr{C})$: CN
Sorties : Booléen

1 $P = \phi(P_{init})$;
2 **si** $P = \bot$ **alors retourner** $faux$;
3 **si** $\forall X \in \mathscr{X}, |dom(X)| = 1$ **alors retourner** $vrai$;
4 **si** $\rho^X(P) \in transposition\_table$ **alors retourner** $faux$;
5 choisir un couple (X, a) avec $|dom(X)| > 1 \wedge a \in dom(X)$;
6 **si** (ρ-$solve(P|_{X=a})$) *ou* (ρ-$solve(P|_{X\neq a})$) **alors retourner** $vrai$;
7 ajouter $\rho^X(P)$ à $transposition\_table$;
8 **retourner** $faux$;

La fonction récursive *ρ-solve* détermine la satisfaisabilité d'un réseau P (voir algorithme 22). A une étape donnée, si le réseau courant (après application de l'opérateur établissant ϕ) est inconsistant, la valeur $faux$ est retournée (ligne 2) tandis que si toutes les variables ont été assignées, la valeur $vrai$ est retournée (ligne 3). Dans le cas contraire, nous vérifions si le nœud courant ne peut pas être élagué grâce à la table de transposition (ligne 4). Si la recherche se poursuit, on sélectionne un couple (X, a) et on appelle récursivement la fonction *solve* en considérant deux branches : l'une étiquetée par $X = a$ et la seconde par $X \neq a$ (lignes 5 et 6). Si une solution est trouvée, la valeur $vrai$ est retournée, sinon le réseau courant a été prouvé inconsistant et l'état partiel inconsistant extrait qui lui correspond est ajouté dans la table de transposition (ligne 7) avant de retourner la valeur $faux$.

Cet algorithme peut être légèrement modifié pour énumérer toutes les solutions d'un réseau en utilisant l'opérateur ρ^{sol}. Pour cela, la table de transposition doit contenir tous les états partiels rencontrés (et non seulement les IPS_P) ainsi qu'une information supplémentaire : l'ensemble

solution. Quand un réseau P est tel que $\rho^{sol}(P)$ est déjà présent dans la table, les solutions de P peuvent être étendues à partir des solutions de $\rho^{sol}(P)$ enregistrées dans la table, grâce à l'interprétation construite à partir des variables s-éliminables de P (c.f. proposition 17). De manière similaire, on peut compter le nombre de solutions d'un problème en associant à chaque sous-réseau réduit enregistré dans la table, le nombre de ses solutions.

3.3.2 Détection de dominance

L'exploitation d'états partiels inconsistants peut être étendue en identifiant les valeurs dont la seule présence aurait suffit à développer des nœuds dominés par un IPS_P [Lecoutre *et al.*, 2007b]. L'approche utilisée est similaire à celle permettant une exploitation efficace des nogoods dans le chapitre précédent (c.f. section 2.3.3). La structure de données paresseuse des watched literals est utilisée pour exploiter la base des IPS_P. Un littéral watché d'un IPS_P Σ est une paire (X, a) telle que $X \in vars(\Sigma)$ et $a \in dom^{\Sigma}(X)$. Ce littéral est dit valide lorsque $a \in dom(X) \setminus dom^{\Sigma}(X)$. Deux littéraux sont associés à chaque état partiel inconsistant Σ et Σ est valide si les deux littéraux qui lui sont associés sont valides. Quand une valeur a est supprimée du domaine d'une variable X, l'ensemble des IPS_P où (X, a) est watché ne sont plus valides. Pour maintenir la validité de ces états partiels inconsistants, il faut pour chacun d'entre eux, soit trouver un autre littéral watché valide, soit supprimer les valeurs de $dom(Y) \cap dom^{\Sigma}(Y)$ où (Y, b) est le second littéral watché. En exploitant cette structure, on garantit que le nœud courant ne peut pas être dominé par un IPS_P.

Lorsqu'on utilise l'opérateur ρ^{prf}, les inférences doivent être effectuées avec précaution. En effet, l'IPS_P Σ responsable d'une inférence participe également à la preuve d'insatisfaisabilité du nœud courant. Dans ce cas, Σ peut être vu comme une contrainte additionnelle du réseau initial : chaque variable apparaissant dans $vars(\Sigma)$ doit également apparaître dans la preuve. Finalement, quelque soit les opérateurs qui sont utilisés, les variables dont le domaine courant n'a jamais été réduit sur la branche courante peuvent être éliminées de manière sûre. En effet, le test de dominance, pour de telles variables, est toujours vrai.

3.4 Résultats expérimentaux

3.4.1 Détection d'équivalence

Les expérimentations effectuées sur des instances de la seconde compétition de solveurs CSP (`http://cpai.ucc.ie/06/Competition.html`) montrent l'efficacité de l'approche proposée ci-dessus. Les tests ont été effectué sur un PC Pentium IV cadencé à $2,4GHz$ avec $1\,024Mio$ de mémoire sous Linux. La plateforme Abscon utilise l'algorithme MGAC (intégrant GAC3^{rm} [Lecoutre et Hemery, 2007]). L'impact de la recherche basée sur les états a été étudié avec différentes heuristiques de choix de variable. Les performances de l'algorithme MGAC, noté $\neg\rho$ dans les tableaux, ont été comparées avec celles obtenues avec les différents opérateurs de réduction notés ρ^X dans les tableaux. Les performances sont mesurées en termes de nombre de nœuds visités (nœuds), de temps cpu en secondes (cpu) ainsi qu'en nombre de nœuds écartés en utilisant un opérateur de réduction (elag). Il est important de remarquer que la recherche basée sur les états peut être appliquée sur des contraintes définies aussi bien en extension qu'en intension (mais, en fonction de l'opérateur de réduction utilisé, gérer des contraintes globales peut nécessiter un traitement spécifique).

L'algorithme 22 a été implémenté, mais en ne considérant qu'un sous-ensemble des variables de U_{elim}, étant donné que déterminer les variables u-éliminables impliquées dans des contraintes

non binaires croît exponentiellement avec l'arité des contraintes. Plus précisément, dans cette implémentation, l'opérateur ρ^{uni} (appelé par ρ^{red}) supprime uniquement les variables ayant un domaine singleton et impliquées dans des contraintes ayant au plus une variable dont le domaine n'est pas singleton. Un tel ensemble peut être calculé en temps linéaire. Sur des réseaux binaires, n'importe quelle variable ayant un domaine singleton est automatiquement supprimée par notre opérateur. On s'intéresse ici à la détection d'équivalence entre les IPS. La structure de données implémentant la table de transposition utilisée pour stocker les états partiels inconsistants est une table de hachage dont la clef est calculée en concaténant les couples (id, dom) où id est un entier unique associé à chaque variable et dom le domaine de la variable lui-même représenté par un vecteur de bits. Lorsqu'on ne cherche qu'une seule solution, aucune information supplémentaire n'a besoin d'être enregistrée dans la table, et la présence de la clef est suffisante pour écarter un nœud.

La table 3.1 présente les résultats obtenus sur les instances du problème des *pigeons*. On peut observer l'intérêt de l'opérateur ρ^{red} sur ce problème puisque de nombreux nœuds sont écartés. Sur ce type d'instance, on peut noter qu'il est plus intéressant d'utiliser l'heuristique *brelaz* (des résultats identiques sont obtenus avec *dom/ddeg* [Bessiere et Régin, 1996]) que *dom/wdeg* [Boussemart *et al.*, 2004a]. Ceci s'explique par le fait que *brelaz* est plus proche de l'ordre lexicographique, qui est bien adapté à ce problème.

Dans la table 3.2, un zoom est fait sur certaines instances réelles difficiles du problème d'allocation de fréquences radio (RLFAP pour Radio Link Frequency Assignment). Étant données $1,200$ secondes de temps cpu, même avec l'opérateur ρ^{red}, ces instances ne peuvent pas être résolues en utilisant *brelaz* ou *dom/ddeg*. C'est pourquoi les résultats obtenus ne sont présentés qu'avec l'heuristique *dom/wdeg*. On peut noter une amélioration par un facteur environ égal à 4 des performances lorsqu'on utilise cet opérateur. La table 3.3 montre finalement les résultats obtenus sur des instances binaires et non-binaires (les instances *dubois* et *pret* comportent des contraintes ternaires) pour lesquelles cette approche est efficace.

Les résultats peuvent être résumés comme suit. Lorsque la recherche basée sur les états (avec l'opérateur ρ^{red}) n'est pas efficace, le solveur est pénalisé d'environ 15%. Cependant, en analysant le comportement de la recherche, on peut décider à tout moment de stopper son utilisation et de libérer la mémoire requise pour stocker les différents états partiels inconsistants : le temps cpu perdu par le solveur est alors borné par le temps défini grâce à cette analyse. Quand cette approche est efficace, et c'est le cas sur certaines séries, l'amélioration peut être très significative que ce soit en temps cpu ou en nombre d'instances résolues.

La quantité de mémoire nécessaire pour stocker ces différents états n'est pas vraiment un handicap. Beaucoup d'espace mémoire peut être économisé, en particulier sur les graphes de contraintes peu denses, en supprimant les variables u-éliminables et r-éliminables. Par exemple, seulement $265Mio$ sont nécessaires pour mémoriser les $50,273$ sous-réseaux différents stockés au cours de la résolution (en 162 secondes) de l'instance $scen11\text{-}f5$. Cette instance est composée de 680 variables et comporte jusqu'à 39 valeurs par domaine.

3.4.2 Détection de dominance

Les expérimentations ont été conduites sur un ordinateur équipé d'un processeur Xeon cadencé à $3GHz$ et $1GiB$ de mémoire. Les instances utilisées sont extraites de la seconde compétition de solveurs CSP (`http://cpai.ucc.ie/06/Competition.html`) incluant des contraintes binaires et non binaires et exprimées aussi bien en intension qu'en extension. L'algorithme générique MGAC, intégré au solveur Abscon, a été utilisé comme point de référence et testé avec de nombreuses combinaisons d'opérateurs d'extraction et d'heuristiques de choix de variable. Les

		brelaz		*dom/wdeg*	
		$\neg\rho$	ρ^{red}	$\neg\rho$	ρ^{red}
Pigeons-11	*cpu*	265,48	2,33	272,73	6,35
	nuds	4 421K	5 065	4 441K	61 010
	elag	0	4 008	0	40 014
Pigeons-13	*cpu*	*timeout*	4,57	*timeout*	26,44
	nuds	–	24 498	–	327K
	elag	–	20 350	–	245K
Pigeons-15	*cpu*	*timeout*	12,66	*timeout*	81,58
	nuds	–	115K	–	900K
	elag	–	98 124	–	728K
Pigeons-18	*cpu*	*timeout*	116,19	*timeout*	*timeout*
	nuds	–	1 114K	–	–
	elag	–	983K	–	–
Pigeons-20	*cpu*	*timeout*	606,59	*timeout*	*timeout*
	nuds	–	4981K	–	–
	elag	–	4456K	–	–

TAB. 3.1 – Coût de MAC avec et sans l'opérateur ρ^{red} sur des instances du problème des Pigeons (détection d'équivalence)

		dom/wdeg	
		$\neg\rho$	ρ^{red}
scen11-f8	*cpu*	14,84	15,74
	nuds	15 045	13 858
	elag	0	370
scen11-f7	*cpu*	57,68	15,36
	nuds	113K	14 265
	elag	0	919
scen11-f6	*cpu*	110,18	18,67
	nuds	217K	18 938
	elag	0	1252
scen11-f5	*cpu*	550,55	162,32
	nuds	1 147K	257K
	elag	0	17 265
scen11-f4	*cpu*	*timeout*	1197,37
	nuds	–	2294K
	elag	–	0
scen11-f3	*cpu*	*timeout*	*timeout*
	nuds	–	–
	elag	–	–

TAB. 3.2 – Coût de MAC avec et sans l'opérateur ρ^{red} sur des instances RLFAP difficiles (détection d'équivalence)

Instances		*brelaz*		*dom/ddeg*		*dom/wdeg*	
		$\neg\rho$	ρ^{red}	$\neg\rho$	ρ^{red}	$\neg\rho$	ρ^{red}
composed-25-10-20-4-ext	*cpu*	179,84	4,27	2,82	2,6	2,7	2,57
	nuds (*elag*)	1 771*K*	9 944 (2 609)	1 644	784 (24)	262	255 (5)
composed-25-10-20-9-ext	*cpu*	857,07	3,73	12,13	10,25	2,58	2,66
	nuds (*elag*)	10*M*	7 935 (1 738)	75 589	54 245 (2 486)	323	323 (0)
dubois-21-ext	*cpu*	*timeout*	480,98	*timeout*	384,84	911,51	295,81
	nuds (*elag*)	–	6 292*K* (2 097*K*)	–	4 194*K* (2 097*K*)	16*M*	3 496*K* (1 573*K*)
dubois-22-ext	*cpu*	*timeout*	*timeout*	*timeout*	*timeout*	*timeout*	527,19
	nuds (*elag*)	–	–	–	–	–	6 641*K* (3 146*K*)
pret-60-25-ext	*cpu*	687,31	5,65	471,16	5,82	416,49	2,27
	nuds (*elag*)	12*M*	55 842 (13 188)	7 822*K*	47 890 (13 188)	7 752*K*	4 080 (1 384)
pret-150-25-ext	*cpu*	*timeout*	*timeout*	*timeout*	*timeout*	*timeout*	10,34
	nuds (*elag*)	–	–	–	–	–	97 967 (37 457)

TAB. 3.3 – Coût de MGAC avec et sans l'opérateur ρ^{red} sur des instances structurées (détection d'équivalence)

performances ont été mesurées en terme de temps cpu (*cpu*) en secondes, de nombres de nœuds visités (*nuds*), de quantité de mémoire consommée (*mem*) mesurée en mébioctet et en nombre moyen de variables éliminées lors de la construction des états partiels inconsistants (*elim*).

La même restriction qu'en section précédente a été considérée concernant l'opérateur ρ^{uni} : seules les variables ayant un domaine singleton et impliquées dans des contraintes ayant au plus une variable dont le domaine n'est pas singleton seront supprimées. Cette restriction évite de vérifier l'universalité des contraintes, ce qui peut être coûteux. Comme l'opérateur ρ^{red} est fortement associé à l'opérateur ρ^{uni} (puisqu'ils ont un comportement identique face à la détection de dominance), les résultats obtenus avec cet opérateur sont mentionnés entre parenthèse dans la colonne de ρ^{uni}.

La table 3.4 présente les résultats obtenus sur certaines séries d'instances structurées. Aucun résultat concernant les instances aléatoires n'est mentionné, puisque sans surprise cette technique d'apprentissage n'est pas adaptée à ce type d'instance. Les configurations testées sont notées $\neg\rho$ (MGAC sans raisonnement basé sur les états), ρ^{uni} et ρ^{prex} combinées avec les trois heuristiques de choix de variable *brelaz*, *dom/ddeg* et *dom/wdeg*. On peut tout d'abord observer que quelque soit l'heuristique utilisée, plus d'instances sont résolues lorsqu'on utilise l'opérateur ρ^{prex}. De plus il faut noter que les performances globales de l'approche basée sur la détection de dominance peuvent être moins bonnes avec l'opérateur ρ^{uni} : plus d'instances sont résolues avec l'heuristique *brelaz* et *dom/ddeg* en utilisant la détection d'équivalence (les résultats sont entre parenthèses). En effet pour ρ^{uni}, la taille des IPS_P peut souvent être assez élevée, ce qui affecte directement la vérification de la dominance alors que la détection d'équivalence peut être effectuée en temps (quasi) constant si l'on utilise une table de hachage.

La table 3.5 se concentre sur certaines instances avec les mêmes configurations testées (l'heuristique *brelaz* est omise, des résultats similaires sont obtenus avec l'heuristique *dom/ddeg*). On peut tout d'abord observer une réduction importante du nombre de nœuds développés en utilisant la détection de dominance, plus particulièrement lorsqu'on emploie l'opérateur ρ^{prex}. Ceci est principalement dû au pourcentage assez élevé de variables éliminées d'un IPS_P (aux alentours de 90% pour l'opérateur ρ^{prex}, beaucoup moins pour l'opérateur ρ^{uni}), ce qui compense le coût assez élevé nécessaire à l'exploitation de la base d'IPS_P. Les mauvais résultats obtenus avec l'opérateur ρ^{prex} sur les problèmes des pigeons peuvent être expliqués par le fait que beaucoup de décisions positives sont enregistrées dans les preuves d'insatisfaisabilité au moment où l'on effectue les propagations de la base d'IPS_P.

Series	*#Inst*	*brelaz*			*dom/ddeg*			*dom/wdeg*		
		$\neg\rho$	ρ^{uni}	ρ^{prex}	$\neg\rho$	ρ^{uni}	ρ^{prex}	$\neg\rho$	ρ^{uni}	ρ^{prex}
aim	48	32	25 (29)	39	32	25 (29)	38	48	43 (47)	48
dubois	13	4	0 (2)	13	4	1 (2)	13	5	13 (3)	11
ii	41	10	10 (10)	13	10	9 (10)	16	20	18 (19)	31
os-taillard-10	30	5	5 (5)	5	4	4 (4)	4	10	10 (10)	13
pigeons	25	13	17 (19)	13	13	17 (19)	13	13	16 (18)	10
pret	8	4	4 (4)	8	4	4 (4)	8	4	8 (4)	8
ramsey	16	3	3 (3)	6	5	3 (5)	5	6	5 (6)	6
scens-11	12	0	0 (0)	0	0	0 (0)	4	9	7 (8)	9
	193	71	64 (72)	92	73	63 (73)	105	115	120 (115)	136

TAB. 3.4 – Nombre d'instances par séries résolues avec différents opérateurs de réduction (1 800 secondes autorisées par instance) en détection de dominance

		dom/ddeg				*dom/wdeg*			
		$\neg\rho$	ρ^{uni}		ρ^{prex}	$\neg\rho$	ρ^{uni}		ρ^{prex}
	cpu	2,39	2,31	(1,9)	2,36	2,3	3,07	(2,25)	3,37
BlackHole-4-4-e-0	*nuds*	6 141	1 241	(1 310)	931	6 293	3 698	(4 655)	5 435
($\#V = 64$)	*elims*	0	13,61	(14,16)	58,88	0	14,35	(14,84)	58,02
	cpu	11,54	65,16	(19,2)	2,45	2,14	2,45	(2,34)	2,2
aim-100-1-6-1	*nuds*	302K	302K	(302K)	737	987	998	(909)	616
($\#V = 200$)	*elims*	0	161,80	(161,78)	190,22	0	174,94	(176,18)	189,78
	cpu	–	–	(–)	4,059	3,4	4,44	(6,56)	3,18
aim-200-1-6-1	*nuds*				6 558	9 063	7 637	(26 192)	1 814
($\#V = 400$)	*elims*				382,73	0	356,39	(348,35)	388,63
	cpu	7,39	6,02	(6,95)	2,65	2,56	2,77	(2,56)	2,47
composed-25-10-20-9	*nuds*	75 589	20 248	(54 245)	184	323	315	(323)	164
($\#V = 105$)	*elims*	0	48,90	(48,95)	87,75	0	67,02	(65,91)	89,25
	cpu	422,2	189,71	(378,87)	83,85	13,69	14,61	(13,13)	12,25
driverlogw-09-sat	*nuds*	118K	37 258	(98 320)	17 623	12 862	8 592	(11 194)	6 853
($\#V = 650$)	*elims*	0	511,69	(514,14)	541,81	0	513,21	(523,26)	544,05
	cpu	183,38	110,71	(78,94)	1,4	65,95	1,62	(47,72)	1,96
dubois-20	*nuds*	24M	787K	(2 097K)	379	8 074K	1 252	(1 660K)	2 133
($\#V = 60$)	*elims*	0	42,00	(48,50)	56,63	0	52,25	(45,93)	51,17
	cpu	–	–	(–)	1,7	–	2,41	(–)	3,39
dubois-30	*nuds*				724		4 267		12 190
($\#V = 90$)	*elims*				86		81,2		78,5
	cpu	16,87	–	(44,23)	4,09	3,08	3,99	(3,69)	2,99
ii-8a2	*nuds*	214K		(214K)	5 224	4 390	4 391	(4 390)	1 558
($\#V = 360$)	*elims*	0		(276,08)	317,04	0	291,67	(291,90)	316,98
	cpu	–	–	(–)	7,92	9,16	26,86	(22,48)	6,71
ii-8b2	*nuds*				2 336	11 148	11 309	(11 148)	3 239
($\#V = 1\,152$)	*elims*				1 090	0	979,59	(980,08)	1 050
	cpu	–	–	(–)	–	–	–	(–)	467,26
os-taillard-10-100-3	*nuds*								134K
($\#V = 100$)	*elims*								66,90
	cpu	–	53,34	(5,63)	–	–	882,41	(23,62)	–
pigeons-15	*nuds*		106K	(115K)			517K	(900K)	
($\#V = 15$)	*elims*		6,99	(7,49)			8,13	(8,63)	
	cpu	–	–	(–)	3,11	–	59,66	(6,32)	4,4
pret-150-25	*nuds*				9 003		203K	(97 967)	17 329
($\#V = 150$)	*elims*				135,72		132,62	(133,71)	137,67
	cpu	66,71	3,17	(3,38)	1,79	76,57	1,97	(1,94)	2,0
pret-60-25	*nuds*	7 822K	17 530	(47 890)	1 503	7 752K	2 631	(4 080)	2 501
($\#V = 60$)	*elims*	0	45,56	(45,71)	52,32	0	51,47	(52,46)	51,98
	cpu	72,72	–	(108,41)	18,25	–	–	(–)	–
ramsey-16-3	*nuds*	1 162K		(1 162K)	46 301				
($\#V = 120$)	*elims*	0		(84,44)	105,49				
	cpu	3,86	4,18	(4,11)	4,15	3,81	4,17	(4,14)	4,04
ramsey-25-4	*nuds*	591	591	(591)	570	590	(591)	(590)	537
($\#V = 300$)	*elims*	0	191,62	(191,40)	274,72	0	159,81	(159,73)	269,36
	cpu	–	–	(–)	371,02	42,15	13,15	(10,17)	5,56
scen11-f6	*nuds*				110K	217K	16 887	(18 938)	2 585
($\#V = 680$)	*elims*				655,80	0	22,72	(22,17)	654,81

TAB. 3.5 – Resultats obtenus sur des instances structurées avec différents opérateurs de réduction (1 800 secondes autorisées par instance) en détection de dominance

Instance		$\neg\rho$	ρ^{uni}		ρ^{unex}		ρ^{prf}	ρ^{prex}
	cpu	9,0	10,4	(8,54)	12,3	(9,6)	5,7	5,5
*scen*11-*f*8	*mem*	29	164	(65)	49	(37)	33	33
($\#V = 680$)	*nuds*	15 045	13 319	(13 858)	13 291	(13 309)	1 706	1 198
	elim		39,0	(38,1)	590,2	(590,2)	643,3	656,0
	cpu	26,0	11,1	(9,23)	10,4	(10,06)	5,5	5,6
*scen*11-*f*7	*mem*	29	168	(73)	49	(37)	33	33
($\#V = 680$)	*nuds*	113K	13 016	(14 265)	12 988	(13 220)	2 096	1 765
	elim		25,2	(25,4)	584,1	(584,7)	647,5	654,8
	cpu	41,2	15,0	(10,61)	15,5	(10,14)	6,4	6,8
*scen*11-*f*6	*mem*	29	200	(85)	53	(37)	33	33
($\#V = 680$)	*nuds*	217K	16 887	(18 938)	16 865	(17 257)	2 903	2 585
	elim		22,7	(22,1)	588,6	(589,3)	648,8	654,8
	cpu	202	–	(72,73)	195	(98,16)	31,5	12,2
*scen*11-*f*5	*mem*	29		256	342	(152)	53	41
($\#V = 680$)	*nuds*	1 147K		257K	218K	(244K)	37 309	14 686
	elim			24,1	592,6	(583,36)	651,6	655,7
	cpu	591	–	(–)	555	(261,67)	404	288
*scen*11-*f*4	*mem*	29			639	(196)	113	93
($\#V = 680$)	*nuds*	3 458K			365K	(924K)	148K	125K
	elim				586,6	(593,1)	651,7	655,0

TAB. 3.6 – Résultats obtenus sur des instances RLFAP difficiles en utilisant *dom/wdeg* (1 800 secondes autorisées) en détection de dominance

La table 3.6 met en valeur certains résultats obtenus sur les instances difficiles de type RLFAP. On considère uniquement l'heuristique *dom/wdeg* avec tous les opérateurs d'extraction présentés précédemment. Assez clairement, l'approche basée sur la détection de dominance avec l'opérateur ρ^{uni} souffre d'une consommation mémoire très importante. Ceci est notamment dû à la taille des IPS_P enregistrés et deux instances restent non résolues. La combinaison des opérateurs ρ^{uni} et ρ^{exp} (i.e. ρ^{unex}) permet d'économiser beaucoup d'espace mémoire (les résultats mentionnés entre parenthèses concernent la combinaison des opérateurs ρ^{red} et ρ^{exp}). Cependant les meilleures performances sont obtenues en combinant le raisonnement basé sur les explications avec celui basé sur la preuve d'inconsistance, i.e. avec ρ^{prex}. On peut également remarquer que la taille moyenne des états partiels inconsistants enregistrés dans la base est très petite : ces états ne portent que sur $680 - 655 = 25$ variables.

Pour résumer les résultats obtenus avec les opérateurs présentés précédemment et les différentes approches proposées, on peut noter que la détection par dominance couplée aux opérateurs d'extraction ρ^{exp} et ρ^{prf} offre globalement de meilleurs résultats que l'opérateur ρ^{red} qui est plus dédié à la détection d'équivalence. L'opérateur ρ^{red} combiné avec ρ^{prf} fourni des résultats similaires à ceux obtenus en utilisant l'opérateur ρ^{red} seul, avec cependant une réduction de la quantité de mémoire nécessaire sur certaines instances. En outre, cela permet de résoudre plus d'instances dans un temps raisonnable.

3.5 Conclusion

Dans ce chapitre, nous avons introduits différents opérateurs permettant l'extraction d'états partiels (inconsistants) à chaque nœud d'un arbre de recherche. Les premiers opérateurs présentés ici, permettent la suppression de variables n'apparaissant que dans des contraintes universelles,

ou dont le domaine est resté inchangé après avoir pris un ensemble de décisions (et appliqué un opérateur établissant une consistance ϕ). Deux autres opérateurs plus sophistiqués ont également été proposés. Tandis que le premier collecte des informations à propos du nœud courant (analyse de la propagation de la racine au nœud) pour effectuer une extraction basée sur des explications, le second collecte ces informations à partir du sous-arbre dont le nœud courant est la racine pour effectuer une extraction basée sur la preuve. Notons d'ailleurs que ces deux approches peuvent être considérées comme complémentaires. Différentes combinaisons de ces opérateurs ont également été envisagées. Nous avons montré dans un second temps qu'il était possible d'exploiter efficacement ces états partiels pour élaguer l'espace de recherche soit par des tests d'équivalence (en utilisant une table de hachage) soit par des tests de dominance (en utilisant la structure de données des watched literals). Plusieurs expérimentations sur une large variété de problèmes ont montré l'intérêt de cette approche.

L'approche présentée ici permet également de détecter et casser certaines formes de symétries locales. Une perspective directe de ces nouveaux paradigmes est de combiner cette approche avec des techniques de détection et d'élimination de symétries [Fahle *et al.*, 2001, Focacci et Milano, 2001, Puget, 2005, Sellmann et Hentenryck, 2005]. Dans le domaine de la satisfaisabilité booléenne, une approche comparable, appelée *component caching* [Kitching et Bacchus, 2007, Sang *et al.*, 2004] peut être utilisée pour mémoriser des sous-formules booléennes et éviter leur résolution si celles-ci ont déjà été rencontrées précédemment. Il serait intéressant de comparer les méthodes d'extraction et d'exploitation utilisées dans le contexte SAT avec celles définies dans ce chapitre.

Conclusion

Dans ce mémoire, nous nous sommes intéressés à l'analyse et l'exploitation des conflits pour réduire l'effort de recherche en satisfaction de contraintes. Les informations extraites à partir des échecs permettent d'identifier plus ou moins précisément la (ou les) sources des conflits et sont principalement utilisées pour réduire le thrashing. Nous avons proposé différentes techniques alternatives aux traditionnels retours-arrière intelligents pour limiter les effets du thrashing sur la recherche. Celles-ci sont dans la mesure du possible suffisament génériques pour pouvoir être greffées à n'importe quel algorithme de recherche, qui le plus souvent intègre un opérateur d'inférence établissant une consistance donnée. Nos résultats expérimentaux montrent que ces méthodes peuvent être compétitives notamment lorsqu'elles sont utilisées dans le cadre de la résolution de problèmes réels et structurés.

Le raisonnement à partir des derniers conflits tout en étant considéré comme une approche prospective (puisqu'elle ne fait que guider la recherche) permet de simuler les sauts effectués par un algorithme de retours-arrière intelligents au cours de la recherche. Le principe est de sélectionner en priorité les variables impliquées dans le dernier conflit. Le fait de "persister" sur ces variables permet d'identifier précisément la raison du dernier échec. De façon originale, l'analyse des échecs s'effectue à la volée, grâce aux retours-arrière successifs, et non pas immédiatement à chaque conflit rencontré. Outre le fait que cette technique soit simple à mettre en oeuvre (ceci est notamment vrai pour LC_1 puisqu'aucune structure de données additionnelle n'est requise), elle permet d'améliorer considérablement les performances des méthodes de résolution basées sur les heuristiques traditionnelles comme *brelaz* et *dom/ddeg*. Nous avons montré également qu'il était possible de l'adapter efficacement à la résolution de problèmes de planification.

Dans cette approche l'heuristique de choix de variable est violée jusqu'à ce qu'un retour-arrière soit effectué sur la décision coupable et qu'une valeur puisse être assignée à la précédente variable en échec. En laissant un degré de liberté plus important à l'heuristique on peut envisager une approche basée uniquement sur l'inférence. Dans ce cas, la variable impliquée dans le dernier conflit n'est alors plus assignée mais on vérifie juste si celle-ci possède dans son domaine une valeur ne provoquant pas directement un échec. Cela correspond à maintenir une singleton consitance sur la variable impliquée dans le dernier conflit.

Comme cette approche permet de simuler un effet backjumping, il serait intéressant de positionner le raisonnement à partir des derniers conflits par rapport aux techniques de retours-arrière intelligents. Plus précisément, la comparaison en terme de parcours de l'arbre de recherche élagué est une question théorique importante. Intuitivement, le raisonnement à partir du dernier conflit semble élaguer au moins le même espace de recherche que la méthode de retours-arrière intelligents à la Gashnig. La comparaison avec les autres techniques de retours-arrière intelligents est plus délicate, dans la mesure où ces méthodes fusionnent les explications provenant d'échecs différents.

L'enregistrement de nogoods à partir des redémarrages apporte une solution originale au problème de redondance qui peut survenir lorsqu'on envisage plusieurs tentatives de résolution successives d'un même problème. Plus précisément les redémarrages sont utilisés pour diversifier la recherche mais n'offrent aucune garantie quant à la portion de l'espace de recherche qui est explorée pendant les exécutions. La méthode proposée permet de délimiter l'espace de recherche parcouru au cours d'une exécution par un ensemble de nogoods enregistrés dans une base. Cette base est exploitée dans les exécutions suivantes pour éviter que le même espace de recherche ne soit exploré plusieurs fois. Les nogoods ne sont pas considérés comme de nouvelles contraintes et un algorithme de propagation spécifique exploite la base de nogoods. La propagation est effectuée en utilisant la technique des watched literals initialement introduite pour SAT. Contrairement aux techniques classiques d'identification et d'enregistrement de nogoods, le nombre de nogoods enregistrés ici reste très réduit. Les problèmes généralement associés à l'exploitation de nogoods (limitation du nombre, réduction de la taille, ...) ne sont pas une préoccupation dans ce contexte.

Les nogoods enregistrés ici correspondent à des nogoods standard (puisque nous avons montré que les décisions négatives peuvent être éliminées de manière sûre). Une perspective intéressante consiste à ne plus exploiter des nogoods standard mais généralisés. Pour cela, les nogoods extraits de la dernière branche de l'arbre de recherche pourraient être minimisés, augmentant ainsi leur capacité d'élagage. Même s'il peut être coûteux de minimiser les nogoods, ce coût ne semble pas dans notre contexte très pénalisant puisque le nombre de nogoods enregistrés est très limité (lorsqu'on utilise une progression géométrique du nombre de retours-arrière autorisés). Les aspects à la fois théoriques et pratiques de cette alternative doivent encore être étudiés.

Dans le contexte SAT, une technique assez proche de "recherche de signature" a également été proposée. Une comparaison minutieuse de ces deux techniques permettrait éventuellement d'améliorer la représentation et l'exploitation des nogoods.

La recherche basée sur les états évite l'exploration de nœuds similaires d'un arbre de recherche. Nous mettons en évidence le fait que lors de la résolution d'un réseau de contraintes, des situations (états de la recherche) jugées identiques peuvent apparaître plusieurs fois au cours de la recherche et qu'il n'est pas nécessaire d'en considérer (i.e. explorer) plusieurs occurences. Différents opérateurs d'extraction permettent l'identification d'états partiels inconsistants, i.e les sous-ensembles de variables du problème avec leur domaine, suffisants pour identifier le fait qu'ils ne mèneront à aucune solution. Les opérateurs utilisent notamment les raisons de la suppression de valeurs ou encore la preuve d'insatisfaisabilité pour éliminer des variables inutiles. Ces états partiels inconsistants (IPS_P) sont enregistrés dans une base, et il suffit de comparer un état avec ceux de la base pour vérifier si une occurence de cet état a déjà été rencontrée. Deux méthodes de détection sont proposées : la détection d'équivalence et la détection de dominance. La détection d'équivalence contrôle l'égalité d'un état avec les états partiels inconsistants de la base alors que la détection de dominance ne contrôle que l'inclusion d'un état avec les états partiels inconsistants. Cette approche permet dynamiquement de détecter et d'éliminer certaines formes de symétries comme l'interchangeabilité au voisinage.

Une perspective à ce travail consiste à exploiter plus intensément les symétries du problème. Il est en effet possible d'enregistrer non pas uniquement un état partiel inconsistant mais tous les états partiels symétriques. La base d'IPS_P doit être minutieusement contrôlée pour éviter une explosion du nombre d'IPS_P enregistrés. Certains aspects théoriques sont encore à étudier afin de déterminer s'il est possible d'identifier de manière sûre des IPS_P subsumés par d'autres précédemment enregistrés. De même de nombreux travaux restent également à effectuer pour définir de nouveaux opérateurs d'extraction plus puissants (i.e. permettant de supprimer plus de

valeurs).

Cette technique est à mettre en relation avec le concept de “component caching” introduit dans le contexte SAT pour mémoriser des sous-formules booléennes précédemment rencontrées. Il serait intéressant de pouvoir comparer les méthodes d'extraction et d'exploitation utilisées en SAT avec celles définies ici.

Bibliographie

[Baker *et al.*, 1993] Baker, R., Dikker, F., F.Tempelman et Wognum, P. (1993). Diagnosing and solving over-determined constraint satisfaction problems. *In Proceedings of IJCAI'93*, pages 276–281.

[Baptista *et al.*, 2001] Baptista, L., Lynce, I. et Marques-Silva, J. (2001). Complete search restart strategies for satisfiability. *In Proceedings of SSA'01 workshop held with IJCAI'01*.

[Bartak et Erben, 2004] Bartak, R. et Erben, R. (2004). A new algorithm for singleton arc consistency. *In Proceedings of FLAIRS'04*.

[Bayardo et Shrag, 1997] Bayardo, R. et Shrag, R. (1997). Using CSP look-back techniques to solve real-world SAT instances. *In Proceedings of AAAI'97*, pages 203–208.

[Beck *et al.*, 2004] Beck, J., Prosser, P. et Wallace, R. (2004). Variable ordering heuristics show promise. *In Proceedings of CP'04*, pages 711–715.

[Berlandier, 1995] Berlandier, P. (1995). Improving domain filtering using restricted path consistency. *In Proceedings of IEEE-CAIA'95*.

[Bessiere, 1991] Bessiere, C. (1991). Arc-consistency in dynamic constraint satisfaction problems. *In Proceedings of AAAI'91*, pages 221–226.

[Bessiere, 1994] Bessiere, C. (1994). Arc consistency and arc consistency again. *Artificial Intelligence*, 65:179–190.

[Bessiere, 2006] Bessiere, C. (2006). Constraint propagation. Rapport technique, LIRMM, Montpellier.

[Bessiere et Debruyne, 2004] Bessiere, C. et Debruyne, R. (2004). Optimal and suboptimal singleton arc consistency algorithms. *In Proceedings of CP'04 workshop on constraint propagation and implementation*, pages 17–27.

[Bessiere et Debruyne, 2005] Bessiere, C. et Debruyne, R. (2005). Optimal and suboptimal singleton arc consistency algorithms. *In Proceedings of IJCAI'05*, pages 54–59.

[Bessiere *et al.*, 1999] Bessiere, C., Freuder, E. et Régin, J. (1999). Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107:125–148.

[Bessiere *et al.*, 2002] Bessiere, C., Meseguer, P., Freuder, E. et Larrosa, J. (2002). On forward checking for non binary constraint satisfaction. *Artificial Intelligence*, 141:205–224.

[Bessiere et Régin, 1996] Bessiere, C. et Régin, J. (1996). MAC and combined heuristics : two reasons to forsake FC (and CBJ ?) on hard problems. *In Proceedings of CP'96*, pages 61–75.

[Bessiere *et al.*, 2005] Bessiere, C., Régin, J., Yap, R. et Zhang, Y. (2005). An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2):165–185.

[Boussemart *et al.*, 2004a] Boussemart, F., Hemery, F., Lecoutre, C. et Sais, L. (2004a). Boosting systematic search by weighting constraints. *In Proceedings of ECAI'04*, pages 146–150.

[Boussemart *et al.*, 2005] Boussemart, F., Hemery, F. et Lecoutre, C. (2005). Description and representation of the problems selected for the first international constraint satisfaction problem solver competition. *In Proceedings of CPAI'05 workshop held with CP'05*, pages 7–26.

[Boussemart *et al.*, 2004b] Boussemart, F., Hemery, F., Lecoutre, C. et Sais, L. (2004b). Support inference for generic filtering. *In Proceedings of CP'04*, pages 721–725.

[Brelaz, 1979] Brelaz, D. (1979). New methods to color the vertices of a graph. *Communications of the ACM*, 22:251–256.

[Cambazard et Jussien, 2005] Cambazard, H. et Jussien, N. (2005). Identifying and exploiting problem structures using explanation-based constraint programming. *In Proceedings of CP-AI-OR'05)*, volume 3524, pages 94–109.

[Cambazard et Jussien, 2006] Cambazard, H. et Jussien, N. (2006). Identifying and exploiting problem structures using explanation-based constraint programming. *Constraints*, 11(4):295–313.

[Chmeiss et Jégou, 1998] Chmeiss, A. et Jégou, P. (1998). Efficient path-consistency propagation. *International Journal on Artificial Intelligence Tools*, 7(2):121–142.

[Chmeiss *et al.*, 2003] Chmeiss, A., Jégou, P. et Keddar, L. (2003). On a generalization of triangulated graphs for domains decomposition of csps. *In International Joint Conference on Artificial Intelligence - IJCAI'03*, pages 203–208.

[de Siqueira et Puget, 1988] de Siqueira, J. et Puget, J. (1988). Explanation-based generalisation of failures. *In Proceedings of ECAI'88*, pages 339–344.

[Debruyne et Bessiere, 1997a] Debruyne, R. et Bessiere, C. (1997a). From restricted path consistency to max-restricted path consistency. *In Proceedings of CP'97*, pages 312–326.

[Debruyne et Bessiere, 1997b] Debruyne, R. et Bessiere, C. (1997b). Some practical filtering techniques for the constraint satisfaction problem. *In Proceedings of IJCAI'97*, pages 412–417.

[Dechter, 1990] Dechter, R. (1990). Enhancement schemes for constraint processing : backjumping, learning and cutset decomposition. *Artificial Intelligence*, 41:273–312.

[Dechter et Frost, 2002] Dechter, R. et Frost, D. (2002). Backjump-based backtracking for constraint satisfaction problems. *Artificial Intelligence*, 136:147–188.

[Dechter et Pearl, 1988] Dechter, R. et Pearl, J. (1988). Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34:1–38.

[Eén et Sorensson, 2003] Eén, N. et Sorensson, N. (2003). An extensible sat-solver. *In Proceedings of SAT'03.*

[Fahle *et al.*, 2001] Fahle, T., Schamberger, S. et Sellman, M. (2001). Symmetry breaking. *In Proceedings of CP'01*, pages 93–107.

[Fikes et Nilsson, 1971] Fikes, R. et Nilsson, N. (1971). Strips : A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208.

[Focacci et Milano, 2001] Focacci, F. et Milano, M. (2001). Global cut framework for removing symmetries. *In Proceedings of CP'01*, pages 77–92.

[Fox et Long, 2003] Fox, M. et Long, D. (2003). Pddl2.1 : An extension to pddl for expressing temporal planning domains. *J. Artif. Intell. Res. (JAIR)*, 20:61–124.

[Freuder, 1982] Freuder, E. (1982). A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32.

[Freuder, 1991] Freuder, E. (1991). Eliminating interchangeable values in constraint satisfaction problems. *In Proceedings of AAAI'91*, pages 227–233.

[Frost et Dechter, 1994] Frost, D. et Dechter, R. (1994). Dead-end driven learning. *In Proceedings of AAAI'94*, pages 294–300.

[Frost et Dechter, 1995] Frost, D. et Dechter, R. (1995). Look-ahead value ordering for constraint satisfaction problems. *In Proceedings of IJCAI'95*, pages 572–578.

[Fukunaga, 2003] Fukunaga, A. (2003). Complete restart strategies using a compact representation of the explored search space. *In Proceedings of SSA'03 workshop held with IJCAI'03.*

[Gaschnig, 1979] Gaschnig, J. (1979). Performance measurement and analysis of search algorithms. Rapport technique CMU-CS-79-124, Carnegie Mellon.

[Gent et Smith, 2000] Gent, I. et Smith, B. (2000). Symmetry breaking during search. *In Proceedings of ECAI'00*, pages 599–603.

[Ginsberg, 1993] Ginsberg, M. (1993). Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46.

[Gomes *et al.*, 2000] Gomes, C., Selman, B., Crato, N. et Kautz, H. (2000). Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24:67–100.

[Greenblatt *et al.*, 1967] Greenblatt, R., Eastlake, D. et Crocker, S. (1967). The Greenblatt chess program. *In Proc. Fall Joint Computer Conference*, pages 801–810.

[Grimes et Wallace, 2007] Grimes, D. et Wallace, R. (2007). Learning to identify global bottlenecks in constraint satisfaction search. *In Proceedings of FLAIRS'07.*

[Han et Lee, 1988] Han, C. et Lee, C. (1988). Comments on Mohr and Henderson's path consistency. *Artificial Intelligence*, 36:125–130.

[Haralick et Elliott, 1980] Haralick, R. et Elliott, G. (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313.

[Hemery *et al.*, 2006] Hemery, F., Lecoutre, C., Sais, L. et Boussemart, F. (2006). Extracting MUCs from constraint networks. *In Proceedings of ECAI'06*, pages 113–117.

[Hoffmann et Nebel, 2001] Hoffmann, J. et Nebel, B. (2001). The FF planning system : Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302.

[Hulubei et O'Sullivan, 2005] Hulubei, T. et O'Sullivan, B. (2005). Search heuristics and heavy-tailed behaviour. *In Proceedings of CP'05*, pages 328–342.

[Hwang et Mitchell, 2005] Hwang, J. et Mitchell, D. (2005). 2-way vs d-way branching for CSP. *In Proceedings of CP'05*, pages 343–357.

[Janssen *et al.*, 1989] Janssen, P., Jégou, P., Nouguier, B. et Vilarem, M. (1989). A filtering process for general constraint-satisfaction problems : Achieving pairewise-consistency using an associated binary representation. *In Proceedings of the IEEE workshop on tools for Artificial Intelligence*, pages 420–427.

[Junker, 2001] Junker, U. (2001). QuickXplain : conflict detection for abitrary constraint propagation algorithms. *In Proceedings of IJCAI'01 Workshop on modelling and solving problems with constraints*, pages 75–82.

[Junker, 2004] Junker, U. (2004). QuickXplain : preferred explanations and relaxations for over-constrained problems. *In Proceedings of AAAI'04*, pages 167–172.

[Jussien *et al.*, 2000] Jussien, N., Debruyne, R. et Boizumault, P. (2000). Maintaining arc-consistency within dynamic backtracking. *In Proceedings of CP'00*, pages 249–261.

[Katsirelos et Bacchus, 2005] Katsirelos, G. et Bacchus, F. (2005). Generalized nogoods in CSPs. *In Proceedings of AAAI'05*, pages 390–396.

[Kautz *et al.*, 2002] Kautz, H., Horvitz, E., Ruan, Y., Gomes, C. et Selman, B. (2002). Dynamic restart policies. *In Proceedings of AAAI'02.*

[Kitching et Bacchus, 2007] Kitching, M. et Bacchus, F. (2007). Symmetric component caching. *In Proceedings of IJCAI'07*, pages 118–124.

[Lecoutre *et al.*, 2003] Lecoutre, C., Boussemart, F. et Hemery, F. (2003). Exploiting multidirectionality in coarse-grained arc consistency algorithms. *In Proceedings of CP'03*, pages 480–494.

[Lecoutre *et al.*, 2004] Lecoutre, C., Boussemart, F. et Hemery, F. (2004). Backjump-based techniques versus conflict-directed heuristics. *In Proceedings of ICTAI'04*, pages 549–557.

[Lecoutre et Cardon, 2005] Lecoutre, C. et Cardon, S. (2005). A greedy approach to establish singleton arc consistency. *In Proceedings of IJCAI'05*, pages 199–204.

[Lecoutre *et al.*, 2007a] Lecoutre, C., Cardon, S. et Vion, J. (2007a). Path consistency by dual consistency. *In Proceedings of CP'07*, pages 438–452.

[Lecoutre et Hemery, 2007] Lecoutre, C. et Hemery, F. (2007). A study of residual supports in arc consistency. *In Proceedings of IJCAI'07*, pages 125–130.

[Lecoutre *et al.*, 2006] Lecoutre, C., Sais, L., Tabary, S. et Vidal, V. (2006). Last conflict-based reasonning. *In Proceedings of ECAI'06*, pages 133–137.

[Lecoutre *et al.*, 2007b] Lecoutre, C., Sais, L., Tabary, S. et Vidal, V. (2007b). Exploiting past and future : Pruning by inconsistent partial state dominance. *In Proceedings of CP'07*, pages 453–467.

[Lecoutre *et al.*, 2007c] Lecoutre, C., Sais, L., Tabary, S. et Vidal, V. (2007c). Nogood recording from restarts. *In Proceedings of IJCAI'07*, pages 131–136.

[Lecoutre *et al.*, 2007d] Lecoutre, C., Sais, L., Tabary, S. et Vidal, V. (2007d). Recording and minimizing nogoods from restarts. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 1:147–167.

[Lecoutre *et al.*, 2007e] Lecoutre, C., Sais, L., Tabary, S. et Vidal, V. (2007e). Transposition Tables for Constraint Satisfaction. *In Proceedings of AAAI'07*, pages 243–248.

[Lecoutre et Tabary, 2007] Lecoutre, C. et Tabary, S. (2007). Abscon 109 : a generic CSP solver. *In Proceedings of the 2006 CSP solver competition*, pages 51–59.

[Lhomme, 2005] Lhomme, O. (2005). Quick shaving. *In Proceedings of AAAI'05*, pages 411–415.

[Mackworth, 1977] Mackworth, A. (1977). Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118.

[Marques-Silva et Sakallah, 1996] Marques-Silva, J. et Sakallah, K. (1996). Conflict analysis in search algorithms for propositional satisfiability. Rapport technique RT/4/96, INESC, Lisboa, Portugal.

[Marsland, 1992] Marsland, T. (1992). Computer chess and search. *In Encyclopedia of Artificial Intelligence*, pages 224–241. J. Wiley & Sons.

[Mauss et Tatar, 2002] Mauss, J. et Tatar, M. (2002). Computing minimal conflicts for rich constraint languages. *In Proceedings of ECAI'02*, pages 151–155.

[Mitchell, 2003] Mitchell, D. (2003). Resolution and constraint satisfaction. *In Proceedings of CP'03*, pages 555–569.

[Mohr et Henderson, 1986] Mohr, R. et Henderson, T. (1986). Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233.

[Montanari, 1974] Montanari, U. (1974). Network of constraints : Fundamental properties and applications to picture processing. *Information Science*, 7:95–132.

[Moskewicz *et al.*, 2001] Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L. et Malik, S. (2001). Chaff : Engineering an Efficient SAT Solver. *In Proceedings of DAC'01*, pages 530–535.

[Petit *et al.*, 2003] Petit, T., Bessiere, C. et Régin, J. (2003). A general conflict-set based framework for partial constraint satisfaction. *In Proceedings of SOFT'03 workshop held with CP'03*.

[Prosser, 1993] Prosser, P. (1993). Hybrid algorithms for the constraint satisfaction problems. *Computational Intelligence*, 9(3):268–299.

[Prosser, 1995] Prosser, P. (1995). MAC-CBJ : maintaining arc consistency with conflict-directed backjumping. Rapport technique, Department of Computer Science, University of Strathclyde.

[Puget, 2005] Puget, J. (2005). Symmetry breaking revisited. *Constraints*, 10(1):23–46.

[Refalo, 2004] Refalo, P. (2004). Impact-based search strategies for constraint programming. *In Proceedings of CP'04*, pages 557–571.

[Reinefeld et Marsland, 1994] Reinefeld, A. et Marsland, T. A. (1994). Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(7): 701–710.

[Rossi *et al.*, 1990] Rossi, F., Petrie, C. et Dhar, V. (1990). On the equivalence of constraint satisfaction problems. *In Proceedings of ECAI'90*, pages 550–556.

[Sabin et Freuder, 1994] Sabin, D. et Freuder, E. (1994). Contradicting conventional wisdom in constraint satisfaction. *In Proceedings of CP'94*, pages 10–20.

[Sang *et al.*, 2004] Sang, T., Bacchus, F., Beame, P., Kautz, H. A. et Pitassi, T. (2004). Combining component caching and clause learning for effective model counting. *In Proceedings of SAT'04*.

[Schiex et Verfaillie, 1994a] Schiex, T. et Verfaillie, G. (1994a). Nogood recording for static and dynamic constraint satisfaction problems. *International Journal of Artificial Intelligence Tools*, 3(2):187–207.

[Schiex et Verfaillie, 1994b] Schiex, T. et Verfaillie, G. (1994b). Stubborness : A possible enhancement for backjumping and nogood recording. *In Proceedings of ECAI'94*, pages 165–172.

[Schulte et Carlsson, 2006] Schulte, C. et Carlsson, M. (2006). Finite domain constraint programming systems. *In Handbook of Constraint Programming*, chapitre 14, pages 495–526. Elsevier.

[Sellmann et Hentenryck, 2005] Sellmann, M. et Hentenryck, P. V. (2005). Structural symmetry breaking. *In Proceedings of IJCAI'05*, pages 298–303.

[Slate et Atkin, 1977] Slate, D. et Atkin, L. (1977). Chess 4.5 : The northwestern university chess program. *In Chess Skill in Man and Machine*, pages 82–118. Springer Verlag.

[Smith, 1999] Smith, B. (1999). The brelaz heuristic and optimal static orderings. *In Proceedings of CP'99*, pages 405–418, Alexandria, VA.

[Stallman et Sussman, 1977] Stallman, R. et Sussman, G. (1977). Forward reasonning and dependency directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196.

[van Dongen, 2002] van Dongen, M. (2002). $AC3_d$ an efficient arc consistency algorithm with a low space complexity. *In Proceedings of CP'02*, pages 755–760.

[van Dongen, 2005] van Dongen, M., éditeur (2005). *Proceedings of CPAI'05 workshop held with CP'05*, volume II.

[Vidal, 2004] Vidal, V. (2004). A lookahead strategy for heuristic search planning. *In Proceedings of ICAPS'04*, pages 150–159.

[Vidal et Geffner, 2006] Vidal, V. et Geffner, H. (2006). Branching and pruning : an optimal temporal pocl planner based on constraint programming. *Artif. Intell.*, 170(3):298–335.

[Wallace, 2005] Wallace, R. (2005). Heuristic policy analysis and efficiency assessment in constraint satisfaction search. *In Proceedings of CPAI'05 workshop held with CP'05*, pages 79–91.

[Walsh, 1999] Walsh, T. (1999). Search in a small world. *In Proceedings of IJCAI'99*, pages 1172–1177.

[Zhang *et al.*, 2001] Zhang, L., Madigan, C., Moskewicz, M. et Malik, S. (2001). Efficient conflict driven learning in a Boolean satisfiability solver. *In Proceedings of ICCAD'01*, pages 279–285.

[Zhang et Malik, 2002] Zhang, L. et Malik, S. (2002). The quest for efficient boolean satisfiability solvers. *In Proceedings of CADE'02*, pages 295–313.

[Zhang et Yap, 2001] Zhang, Y. et Yap, R. (2001). Making AC3 an optimal algorithm. *In Proceedings of IJCAI'01*, pages 316–321.

[Zobrist, 1970] Zobrist, A. L. (1970). A new hashing method with applications for game playing. Rapport technique 88, Computer Sciences Dept., Univ. of Wisconsin. Reprinted in *Int. Computer Chess Association Journal* 13(2) :169–173 (1990).

Résumé

Le cadre "Problème de Satisfaction de Contraintes" (ou CSP pour Constraint Satisfaction Problem) regroupe les problèmes modélisables sous la forme de contraintes. Pour résoudre ce type de problèmes, une approche classique, et complète, consiste à alterner prises de décisions (assignations de variables) et processus de propagation de contraintes (réduction de l'espace de recherche). Au lieu de remettre en cause systématiquement le dernier choix effectué lorsqu'un conflit (i.e. l'impossibilité de trouver une solution sur la base des décisions déjà prises) apparaît, il peut être intéressant d'identifier les raisons précises de cet échec. Nous proposons dans cette thèse plusieurs techniques d'analyse et d'exploitation des conflits qui permettent de réduire de façon significative l'effort de recherche nécessaire à la résolution des problèmes de satisfaction de contraintes. Nous montrons tout d'abord qu'il est possible de guider efficacement la recherche en se basant sur les derniers conflits rencontrés. L'approche que nous proposons permet de lutter contre le thrashing (le fait de reproduire plusieurs fois les mêmes erreurs) en orientant la recherche sur l'origine du dernier conflit rencontré. Une autre solution connue consiste à effectuer des redémarrages : l'algorithme de recherche est relancé plusieurs fois pour diversifier la recherche. Cependant, dans ce contexte, il n'est pas exclu d'explorer inutilement plusieurs fois les mêmes portions de l'espace de recherche. Pour éviter cela, nous proposons une technique consistant à calculer a posteriori, i.e. à chaque redémarrage, les explications des échecs rencontrés au cours de la recherche sous la forme d'un ensemble de nogoods (ensembles d'assignations de variables perçues comme de nouvelles contraintes). Ceux-ci sont enregistrés dans une base exploitée grace à une structure de données paresseuse : les watched literals. Finalement nous mettons en évidence le fait que des situations (i.e. des états de la recherche à un instant donné) similaires peuvent apparaître au cours de la recherche. Nous introduisons des opérateurs originaux, basés sur la détection d'états partiels inconsistants, permettant l'identification de ces situations jugées équivalentes. Un état partiel inconsistant représente un sous-ensemble de l'état de la recherche à un instant donné, suffisant pour démontrer l'absence de solution à partir de cet état. Nous montrons également que ces opérateurs permettent de détecter et casser dynamiquement certaines formes de symétries.

Abstract

The Constraint Satisfaction Problem framework (CSP) allows to deal with problems modeled using constraints. To solve this kind of problems, a classical complete approach consists in interleaving decision steps (variable assignments) and constraint propagation steps (reduction of the search space). When a conflict occurs (i.e. no solution can be found from the decisions already performed), instead of systematically backtracking on the last choice, it can be useful to identify the exact reasons of this failure. In the context of solving constraint satisfaction problems, we propose in this Phd Thesis several approaches to reduce the search effort by analysing and exploiting these conflicts. First we show that it is possible to efficiently guide search using the last encountered conflicts. The approach that we propose allows us to prevent thrashing (the fact of performing several times the same errors) by directing the search on the source of the last encountered conflict. Another well-known solution consists in exploiting restarts capabilities : the search algorithm is restarted several times in order to diversify the search. However, in this

context, one can unnecessarily explore several times the same parts of the search space. To avoid this phenomenon, we propose to compute a posteriori, i.e. at each restart, some explanations of the failures occurring during search using a set of nogoods (sets of variable assignments considered as new constraints). These nogoods are recorded in a base exploited thanks to a lazy data-structure : the watched literals. Finally we underline the fact that some similar situations (i.e. some states of the search at a given step) can occur during search. We introduce some original operators, based on the detection of inconsistent partial states, and use then to identify some situations considered as equivalent. An inconsistent partial state represents a subset of search state, sufficient to prove that no solution can be found from this state. We also show that these operators allow us to detect and dynamically break some kinds of symetries.