Hybridation de techniques d'apprentissage de clauses en programmation par contraintes

Gael Glorian

Hybridation de techniques d'apprentissage de clauses en programmation par contraintes

Thèse de Doctorat

présentée et soutenue publiquement le 11 décembre 2019

pour obtenir le titre de

Docteur de l'Université d'Artois

(Informatique)

par

Gaël Glorian

Composition du jury

Président :	Christine Solnon	INSA Lyon	Professeure des universités
Rapporteurs :	Laurent Simon	Université de Bordeaux	Professeur des universités
	Cyril Terrioux	Aix-Marseille Université	Maître de conférences HDR
Examinateur :	Marie-José Huguet	INSA Toulouse	Professeure des universités
Invité :	Gilles Audemard	Université d'Artois	Professeur des universités
Encadrants :	Frédéric Boussemart	Université d'Artois	Maître de conférences
	Jean-Marie Lagniez	Huawei Technologies Ltd	Chercheur principal
Directeurs :	Bertrand Mazure	Université d'Artois	Professeur des universités
	Christophe Lecoutre	Université d'Artois	Professeur des universités

Centre de Recherche en Informatique de Lens – CNRS UMR 8188
Université d'Artois, rue Jean Souvraz, S.P. 18 F-62307, Lens Cedex France
Secretariat : Tél.: +33 (0)3 21 79 17 23
http://www.cril.univ-artois.fr

Mise en page par thesul v0.14 (D. Roegel, LORIA) et memcril v0.20 (B. Mazure, CRIL)
La conception des boîtes est faite par xeboiboites (A. Flesch)

Sommaire

II Contributions

Liste des algorithmes

Introduction générale

Le contexte général des travaux de thèse de ce manuscrit s'inscrit dans celui de l'intelligence artificielle et plus précisément la programmation par contraintes (CP). Ce paradigme de « programmation » a émergé au cours des années 70 (Mackworth (1977)), et s'est montré particulièrement efficace pour résoudre de nombreuses familles de problèmes combinatoires difficiles. Un problème combinatoire est un problème dont les instances, pour être résolues, nécessitent d'évaluer un nombre de combinaisons croissant de manière exponentielle avec leurs tailles. On trouve des problèmes combinatoires dans de nombreuses situations de la vie courante (calcul d'emploi du temps, calcul de trajets, résolution de jeux, etc.) et dans de nombreuses applications industrielles (vérification de circuits, placement d'antennes relais, etc.).

Le programmation par contraintes s'inscrit pleinement dans le contexte de recherche contemporain. En effet, CP a de nombreuses applications avec d'autres champs de recherche et/ou l'industrie. Un des exemples notables de ces dernières années est l'atterrisseur *Philae*, de la sonde spatiale *Rosetta*, qui s'est posé sur la comète *67P/Churyumov-Gerasimenko* en 2014. Lors de son réveil en août 2015, un outil piloté par un système de programmation par contraintes a permis à ce laboratoire robot de gérer ses ressources limitées (Simonin *et al.* (2015)) ; les deux grandes limites étant l'énergie disponible ainsi que la mémoire embarquée. Les expériences effectuées par *Philae* ont été planifiées à l'aide de la programmation par contraintes, avec comme contrainte d'obtenir un temps de calcul d'un nouveau plan pour *Philae* relativement rapide. L'approche de planification par programmation par contraintes, grâce à de nouveaux propagateurs et contraintes globales, a permis de réduire le temps de calcul d'un plan de plusieurs heures à quelques secondes. Calculer et vérifier les plans dans un délai raisonnable était extrêmement important dans ce contexte mais cela reste vrai dans le cas général où un utilisateur préfère obtenir le plus rapidement possible une réponse à un problème donné.

Les problèmes combinatoires ne comportent pas toujours de solutions. Cela signifie qu'il faut parcourir l'ensemble des combinaisons avant de pouvoir conclure qu'aucune solution n'existe. En pratique, toutes les combinaisons possibles ne peuvent pas être explorées (pour des problèmes de taille relativement importante). De nombreuses techniques ont été proposées dans l'optique de réduire le nombre de combinaisons testées avant de trouver une solution voire de prouver qu'aucune n'existe. Ici, nous nous intéressons aux approches dites complètes qui ont pour vocation d'effectuer une recherche exhaustive et donc théoriquement, d'explorer tout l'arbre de recherche dans le pire cas. Il existe une autre approche afin de trouver une solution : l'exploration incomplète. Elle permet de parcourir des parties de l'espace de recherche jugées comme intéressantes. Ce type d'approche doit être guidée par des heuristiques afin de sélectionner les parties prometteuses qui seront explorées. De plus, comme celle-ci ne parcourt pas la totalité de l'espace, prouver qu'il n'existe pas de solution est en général impossible.

Dans un environnement de recherche complète nous voulons trouver une solution ou prouver qu'il n'existe pas de solution de manière efficace. Cela signifie qu'il nous faut explorer l'arbre de recherche de manière parcimonieuse. Pour cela, des techniques ont été proposées afin de guider la recherche vers des endroits prometteurs de manière heuristique ; ou au contraire, vers des parties de l'arbre de recherche qui ont une grande chance de ne contenir aucune solution afin d'élaguer l'arbre plus efficacement. Cela est généralement fait dans un contexte où les redémarrages (relancer la recherche régulièrement à partir du début) sont utilisés, conjointement aux *nogoods*, des instanciations partielles qui ne peuvent pas être étendues vers des solutions.

Le cadre CSP se situe au cœur de la programmation par contraintes, et le problème de satisfaction booléenne (SAT) est un problème particulier du problème de satisfaction de contraintes qui trouve de nombreuses applications, par exemple, en planification et vérification de circuits. Le problème SAT se limite aux variables binaires contrairement à CSP où les variables ont des domaines finis de taille quelconque (variables discrètes). Concernant le type de contraintes (booléennes) pouvant être gérées, le problème SAT implique uniquement des clauses, c'est-à-dire des disjonctions de littéraux (variables booléennes et leurs négations), qui peuvent être également représentées sous forme de sommes positives ($sum(x_i) > 0$). Malgré ces limites, la force de ce paradigme réside dans ses méthodes de résolution et ses solveurs extrêmement performants. Dans le cas général, il nous faut choisir entre l'expressivité de CSP et la puissance de SAT.

Le but global des travaux présentés dans ce manuscrit est de cerner et formaliser des informations qui permettent une recherche plus efficace, notamment dans un environnement hybride SAT/CSP. Ces travaux s'inscrivent dans une politique de recherche contemporaine car applicable au contexte de l'optimisation et de la résolution parallèle ou distribuée. En effet, à l'heure de la multiplicité des machines, résoudre de manière parallèle ou distribuée un problème devient monnaie courante. Trouver des informations et une façon de les représenter permet de les partager. De plus, une meilleure utilisation de ces informations permet, par conséquent, d'améliorer l'efficacité de la recherche.

Au cours de la dernière décennie, la communauté de l'intelligence artificielle a consacré de nombreux efforts à la conception d'algorithmes généraux de résolution de problèmes de satisfaction de contraintes discrets (Rossi *et al.* (2006)). Les approches classiques en matière de résolution générale CSP reposent sur des méthodes complètes intégrant des procédures de filtrage, des méthodes heuristiques et des mécanismes d'apprentissage. Même pour les approches de l'état de l'art, plusieurs problèmes sont hors de portée. En effet, décider si un CSP est satisfiable est un problème NP-complet (Mackworth (1977)). C'est pour cette raison que de nouvelles approches doivent être proposées et évaluées. Cependant, la mise en œuvre d'une nouvelle technique dans un solveur CSP nécessite généralement une connaissance approfondie du solveur. De plus, pour être acceptée par la communauté, une nouvelle technique doit être comparée à des méthodes de l'état de l'art, ce qui implique qu'elles doivent être soigneusement mises en œuvre.

Les progrès impressionnants réalisés en SAT au cours des deux dernières décennies l'ont été grâce à l'enregistrement de *nogoods* (apprentissage de clauses) dans le cadre d'une politique de redémarrage renforcée par une structure de données paresseuse très efficace (Moskewicz *et al.* (2001)). L'intérêt pratique de l'apprentissage de clauses a crû avec la mise à disposition d'instances réelles volumineuses (applications pratiques) comportant une structure et présentant des phénomènes dits *heavy-tailed*. L'apprentissage de clauses en SAT est un exemple de technique fructueuse dérivée de l'intersection de la recherche à la fois en programmation par contraintes et en SAT. En effet, l'enregistrement de nogood (Dechter (1990)) et le retour arrière non chronologique dirigé par les conflits (Prosser (1993)) ont été introduits à l'origine pour CSP et ont ensuite été importés dans les solveurs SAT (Marques-Silva et Sakallah (1996), Jr. et Schrag (1997), Marques-Silva et Sakallah (1999)). Les progrès réalisés dans le cadre SAT ont suscité un regain d'intérêt de la communauté CSP pour l'enregistrement de nogood dans les années 2000 (Katsirelos et Bacchus (2003), Lecoutre *et al.* (2009)). Cependant, une partie des approches modernes proposées nécessitent l'utilisation d'un solveur SAT externe. Dans ce cas, le problème est entièrement encodé en CNF (Soh *et al.* (2017)) où il est traduit de manière incrémentale, voire paresseuse (Ohrimenko *et al.* (2007; 2009)). Mais, dans les deux cas, le solveur SAT a la priorité sur le solveur CP (exception faite pour la nouvelle architecture de la génération paresseuse de clauses, Feydy et Stuckey (2009)).

Le manuscrit est découpé en deux parties. La première partie contient les notions et définitions de l'état de l'art ainsi que les notations utilisées. Elle traite à la fois de la programmation par contraintes et de l'apprentissage d'information dans ce contexte, puis du problème de satisfaction booléenne et enfin de l'hybridation de ces deux paradigmes.

Au sein du premier chapitre de cette partie, nous introduisons formellement le problème de satisfaction de contraintes (et plus généralement la programmation par contraintes). Nous présentons le concept de filtrage et de cohérence ainsi que différentes méthodes de résolution complètes ; puis, nous expliquons le principe du guidage heuristique dans le contexte du problème de satisfaction de contraintes en présentant une sélection des heuristiques de choix de variables et de valeurs de l'état de l'art.

Le deuxième chapitre traite de l'apprentissage d'informations en programmation par contraintes. Nous y présentons les *nogoods*, introduits dans les années 90 (Dechter (1990)). Dans la suite deux formes modernes de *nogoods* sont présentés, à savoir les *nld-nogoods* (Lecoutre *et al.* (2007b)) et leur forme compressée, qui permet aussi de gagner en efficacité, les *increasing-nogoods* (Lee *et al.* (2016)).

Après cette introduction à la résolution de problèmes de satisfaction de contraintes et à l'apprentissage de *nogoods*, nous nous intéressons, dans le chapitre 3, a un problème central en informatique et en intelligence artificielle : le problème de satisfaction booléenne (SAT). Dans ce chapitre, après une introduction formelle à la logique propositionnelle et la présentation de l'algorithme DPLL (Davis *et al.* (1962)), nous nous concentrons majoritairement sur les briques composant les solveurs SAT modernes, à savoir les solveurs CDCL (Marques-Silva et Sakallah (1996)). Notamment, nous présentons de manière détaillée l'analyse de conflits qui permet l'apprentissage d'information (de clauses) en SAT.

Le dernier chapitre de cette partie concerne l'hybridation entre les problèmes de satisfaction de contraintes et de satisfaction booléenne. Ces deux problèmes, étant très proches, ont permis l'émergence de techniques hybrides en programmation par contraintes exploitant les forces de chaque problème afin de corriger les faiblesses de l'autre. Toujours dans le contexte de résolution complète de problèmes de satisfaction de contraintes, nous présentons les techniques, inspirée par les progrès en SAT, qui permettent, la générations de clauses et le raisonnement par résolution grâce aux explications en programmation par contraintes. Nous introduisons trois méthodes dans ce chapitre, à savoir les *nogoods* généralisés (Katsirelos et Bacchus (2003)), les explications paresseuses (Gent *et al.* (2010)) ainsi que la génération paresseuse de clauses (Ohrimenko *et al.* (2007)).

Dans la seconde partie, nous présentons les résultats obtenus, qu'ils soient théoriques ou pratiques. Cette partie est subdivisée en trois chapitres.

Dans le premier, nous nous intéressons à la combinaison d'*increasing-nogoods* afin de permettre d'anticiper les conflits qui pourraient survenir sur l'espace de recherche considéré. Ceci est réalisé dans un contexte d'algorithme complet avec retours arrière et redémarrages. Des similitudes identifiées entre les *increasing-nogoods* générés lors de différents redémarrages sont exploitées, dans le but d'augmenter le pouvoir de filtrage des *increasing-nogoods* et donc d'élaguer l'arbre de recherche de manière plus importante.

Le deuxième chapitre de la seconde partie présente l'outil NACRE et plus particulièrement NACRE 1.0.4. C'est un moteur de raisonnement générique, construit pour recevoir des techniques utilisant à la fois des *nogoods* et des clauses générées de manière hybride. La nécessité d'un tel outil s'est faite ressentir lors de l'implémentation des travaux présentés dans le premier chapitre des contributions. En effet, une majorité des solveurs de programmation par contraintes standards

ne sont pas optimaux pour l'apprentissage de *nogoods* ou de clauses. Dans ce chapitre, nous proposons donc un outil qui répond à cette problématique ; et qui permet de gérer une base de *nogoods* facilement ainsi qu'un moteur de raisonnement sur les clauses inspiré des explications paresseuses et dotés de plusieurs améliorations. Nous proposons aussi une nouvelle politique de réduction d'une base de clauses (apprises) dans un contexte hybride qui considère les variables CSP dont les littéraux composant les clauses sont issues.

Le chapitre final de ce manuscrit est composé des deux sections principales. En effet, le point de départ de ces deux sections est le même, améliorer l'utilité et l'efficacité des clauses générées dans un contexte hybride. Pour cela, deux pistes s'offrent à nous : proposer des algorithmes de raisonnement *ad-hoc* afin d'obtenir des raisons plus fines ; ou améliorer la qualité des clauses, notamment en les minimisant. Nous avons exploré ces deux pistes. Dans un premier temps, nous proposons des algorithmes de raisonnement pour la contrainte globale *element* sous plusieurs formes (en particulier, les formes *constante* et *variable*). La seconde section du chapitre se focalise sur la qualité des clauses en se posant la question suivante :

« Pouvons-nous faire confiance aux clauses ? »

Avec cette interrogation en tête, nous commençons cette section en proposant une heuristique de recherche plus adaptée à un contexte hybride qui se base sur l'heuristique SAT VSIDS (Moskewicz *et al.* (2001)). Par la suite, afin de corriger les clauses produites par un algorithme de raisonnement générique, nous proposons une méthode hybride de minimisation faisant appel à des formes de minimisation prenant en compte les domaines et des formes SAT basées sur la trace (*trail*).

Première partie
État de l'art

Chapitre 1

Programmation par contraintes

Sommaire

> « Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it. »
>
> *Eugene C. Freuder 1997*

Le problème de satisfaction de contraintes (CSP – *Constraint Satisfaction Problem*) est généralement présenté sous la forme d'un ensemble de variables (auxquelles sont associés des domaines de valeurs) et d'un ensemble de contraintes, le tout formant un réseau de contraintes. L'objectif du problème de satisfaction de contraintes est alors de déterminer s'il existe une affectation des variables telle que toutes les contraintes soient satisfaites. C'est un problème de décision `NP-complet`. Un grand nombre de problèmes en intelligence artificielle peuvent être considérés comme des cas particuliers de problèmes CSP.

Dans ce chapitre, nous allons introduire formellement ce qu'est le problème de satisfaction de contraintes ainsi que l'algorithmique autour de ce problème. Nous décrivons les algorithmes de résolution basés sur le concept de cohérence ainsi que le filtrage *via* les contraintes par les propagateurs. Par la suite, nous introduisons les méthodes de résolution les plus connus, à savoir le *Backtracking*, le *Forward Checking* et enfin *Maintaining Arc Consistency*. Avant de conclure, nous présentons les heuristiques de choix de valeurs et de choix de variables essentielles en programmation par contraintes.

1.1 Réseaux de contraintes

La programmation par contraintes a été développée dès les années 60 afin de formaliser et résoudre informatiquement de nombreux problèmes. Cette formalisation s'appuie sur la construction d'un réseau de contraintes permettant de modéliser le problème à résoudre.

Définition 1 (Réseau de contraintes)

Un **réseau de contraintes** peut être représenté sous différentes formes, ici nous choisissons une formalisation par deux ensembles, $P = (\mathcal{X}, \mathcal{C})$ où :

— $\mathcal{X}$ est un ensemble fini de variables composant le réseau ;

— $\mathcal{C}$ est un ensemble fini de contraintes.

Chaque variable $x \in \mathcal{X}$ possède un domaine (courant) noté $\texttt{dom}(\mathbf{x})$ qui est l'ensemble fini de valeurs a qui peuvent être affectées à la variable x. Le domaine initial d'une variable x est noté $\texttt{dom}^{\mathbf{init}}(\mathbf{x})$. $\mathbf{vars}(P)$ décrit l'ensemble des variables de P, par conséquent $\mathcal{X} = vars(P)$. Les variables ainsi définies sont groupées sous forme de contraintes afin de modéliser les problèmes.

Définition 2 (Portée d'une contrainte)

Une contrainte $c \in \mathcal{C}$ est définie sur un sous-ensemble de variables de $\mathcal{X}$. Ce sous-ensemble est noté $scp(c)$ et appelé **portée** (ou *scope*) de la contrainte.

Définition 3 (Arité d'une contrainte)

Le nombre d'éléments de $scp(c)$ est appelé l'**arité** de c.

Remarque 1

L'arité d'un réseau de contraintes est la plus grande arité de ses contraintes.

Définition 4 (Contrainte)

Une **contrainte** $c \in \mathcal{C}$ est une relation $rel(c) \subseteq \prod_{x \in scp(c)} \texttt{dom}^{init}(x)$ qui représente l'ensemble des tuples de valeurs qui satisfont c.

Un tuple τ peut être perçu comme l'instanciation d'un sous-ensemble de variables de $\mathcal{X}$. $\tau[x]$ représente la valeur de la variable x dans τ.

Définition 5 (Instanciation)

Une **instanciation** t est un ensemble $\{(x, v)|(x \in \mathcal{X}) \wedge (v \in \texttt{dom}(x))\}$, c'est-à-dire qu'à chaque variable est associée une valeur de son domaine.

Le problème CSP est un problème de décision qui cherche à déterminer si un réseau de contraintes admet une solution. Pour cela nous avons besoin de définir la notion de contrainte satisfaite.

Définition 6 (Contrainte satisfaite)

Une contrainte $c \in \mathcal{C}$ est **satisfaite** par une instanciation t où $vars(t) \subseteq scp(c)$ si et seulement si $\{(x, v)|(x \in scp(c)) \wedge (t[x] = v)\} \in rel(c)$.

Comme un réseau de contraintes interprète conjonctivement les contraintes, la notion de solution du réseau de contraintes s'étend directement de la notion de satisfaction d'une contrainte.

Définition 7 (Solution)

Une **solution** d'un réseau de contraintes P est une instanciation de $\mathcal{X}$ telle que toutes ses contraintes soient satisfaites.

Une contrainte peut être représentée de différentes manières, en intention décrite par un prédicat ($X < Y$, $Z \neq X$, ...), ou en extension par tableaux. La représentation en extension peut décrire la contrainte de deux manières, en supports, qui listent les tuples qui sont solutions de la contrainte, ou en conflits, qui indiquent les tuples qui sont en contradiction (incohérents) avec la contrainte.

x	y
1	2
1	3
2	3

TABLE 1.1 – Contrainte décrite en extension support.

La table 1.1 représente la contrainte $x < y$ avec $\texttt{dom}(x) = \{1, 2\}$ et $\texttt{dom}(y) = \{1, 2, 3\}$. Sa portée est $\{x, y\}$ et son arité est de 2. Cette contrainte est définie en extension support, par conséquent, les tuples $(1, 2)$, $(1, 3)$ et $(2, 3)$ apparaissant dans la table satisfont la contrainte tandis que tous les autres tuples la violent (par exemple, le tuple $(2, 1)$ la viole).

Définition 8 (Contrainte globale)

Une **contrainte globale** est un modèle de contrainte qui capture une relation sémantique précise et qui peut être appliquée sur un nombre arbitraire de variables.

Les contraintes globales ont été introduite dans l'optique de faciliter la modélisation et d'améliorer la résolution car elles ont souvent des algorithmes de propagation dédiés. Il existe aujourd'hui plus de 400 contraintes globales différentes, pour plus de détails, voir Beldiceanu *et al.* (2007).

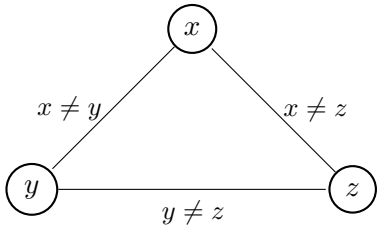


FIGURE 1.1 – Exemple : Clique de contraintes primitives en intention $\neq$.

Exemple 1

Soient les variables x, y et z avec $\texttt{dom}(x) = \texttt{dom}(y) = \texttt{dom}(z) = \{0, 1\}$. Si nous souhaitons que les trois variables aient une valeur différente, nous pouvons utiliser une clique de contraintes en intention $\neq$ (Figure 1.1). Nous pouvons choisir de représenter ces contraintes grâce à une unique contrainte globale `AllDifferent` (Laurière (1978)) qui force chaque variable de sa portée à prendre une valeur distincte (Figure 1.2).

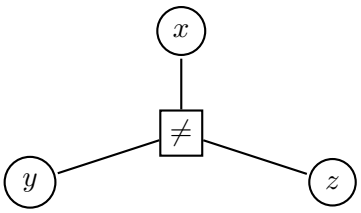


FIGURE 1.2 – Exemple : Représentation avec la contrainte globale `AllDifferent`.

Nous avons défini formellement dans cette section le problème de satisfaction de contraintes ainsi que les différents types de contraintes. La section suivante introduit les concepts de cohérence du réseau et de filtrage de contraintes.

1.2 Cohérence et résolution

Un concept très important dans la résolution de problèmes de satisfaction en programmation par contraintes est la cohérence. Un réseau doit être cohérent selon un opérateur Φ de cohérence.

De nombreux opérateurs ont été présentés dans l'état de l'art et permettent d'obtenir un réseau cohérent.

Dans cette section, nous présentons dans un premier temps différents types de cohérence, à savoir la cohérence d'arc et la cohérence de chemin. La cohérence d'arc (généralisée) est la forme de cohérence la plus utilisée en pratique. Ensuite, nous allons présenter deux propagateurs de contraintes, à savoir celui pour la contrainte d'inégalité ainsi que celui pour la contrainte de somme inférieure ou égale. Ceux-ci permettent, dans le cas général, de garder la cohérence au sein des différentes contraintes. Nous clôturons cette section par différentes méthodes de résolution afin de voir comment appliquer ses opérateurs dans le but de chercher une solution au réseau de contraintes.

1.2.1 Filtrage et cohérence

Le but d'un filtrage est d'éliminer des valeurs dont nous sommes assurés qu'elles ne peuvent figurer dans aucune solution (du réseau de contraintes ou d'un sous-ensemble de contraintes). De plus, les algorithmes de filtrage peuvent, dans certains cas, détecter l'incohérence d'un réseau de contraintes. De nombreuses formes de cohérences plus ou moins fortes existent, notamment la cohérence de nœud (Mackworth (1977)), d'arc (AC) (Mackworth (1977)), de chemin (PC) (Mackworth (1977)), *singleton arc consistency* (SAC) (Debruyne et Bessière (1997b)), ainsi que des formes dirigées de cohérences (Debruyne et Bessière (1997a)).

1.2.1.1 Cohérence d'arc

L'algorithme de filtrage le plus utilisé, dans le cas général, est la cohérence d'arc (AC) (Mackworth (1977)). La cohérence d'arc est le plus ancien moyen de propagation de contraintes binaires (d'arité 2). La forme généralisée (GAC) permet de prendre en compte les contraintes n-aires (avec $n > 2$). C'est un concept très simple qui garantit la cohérence de chaque valeur d'un domaine d'une variable avec les contraintes associées à celle-ci. Une contrainte est (généralement) arc-cohérente si pour toute variable impliquée dans la contrainte, et pour toute valeur du domaine de celle-ci, il existe au moins une valeur dans le domaine de chaque autre variable impliquée telle que la contrainte soit satisfaite.

Définition 9 (Cohérence d'arc (généralisée))

Soient un réseau de contraintes $P = (\mathcal{X}, \mathcal{C})$, une contrainte $c \in \mathcal{C}$ et une variable $x \in scp(c)$. Une valeur $a \in \texttt{dom}(x)$ est cohérente avec c si et seulement si $\exists \tau \in rel(c)$ tel que $\tau[x] = a$. Une variable x est **(généralement) arc cohérente** sur c si et seulement si toutes les valeurs $a \in \texttt{dom}(x)$ sont cohérentes avec c. Une variable x est (généralement) arc cohérente si elle est (généralement) arc cohérente vis-à-vis de chaque contrainte c tel que $x \in scp(c)$.

Remarque 2

Par extension, un CSP est (généralement) arc cohérent si toutes ses variables le sont.

Exemple 2

Soit la table 1.1 représentant la contrainte $x < y$ avec $\texttt{dom}(x) = \{1, 2\}$ et $\texttt{dom}(y) = \{1, 2, 3\}$, nous pouvons voir que la valeur 1 du domaine de la variable y n'apparaît pas au sein de celle-ci car elle n'a pas de support dans la relation de la contrainte. Cette valeur n'est pas arc-cohérente car $\nexists \tau \in rel(c)$ tel que $\tau[y] = 1$.

Étant donné que cette forme de cohérence permet un filtrage de qualité tout en restant simple, la cohérence d'arc a donné naissance à plusieurs algorithmes de filtrage : AC2001 (Bessière *et al.* (2005)), AC3^{rm} (Lecoutre et Hemery (2007)), AC3^{bit+rm} (Lecoutre et Vion (2008)), etc. Le plus connu étant AC3 (Mackworth (1977)) qui consiste à utiliser une file pour créer un ensemble d'arcs à revisiter, ceux qui ont été touchés lors de la réduction d'un domaine. AC3 a une complexité en $O(e \times d^3)$ (où e est le nombre de contraintes du réseau et d la taille du plus grand domaine des variables du réseau). L'algorithme 1.2 décrit la méthode de filtrage AC3, elle commence par remplir l'ensemble Q de chaque arc (il faut deux arcs pour décrire une contrainte) composant le réseau, puis tant que Q n'est pas vide un arc est choisi (et retiré) puis l'algorithme 1.1 est appliqué sur celui-ci. L'algorithme 1.1 vérifie que chaque valeur de la variable i a un support dans j. Cela est réalisé en ligne 3 par la fonction $P_{ij}(a, b)$ qui retourne vrai si le couple (a, b) est accepté par l'arc (i, j). Si ce n'est pas le cas, la valeur est supprimée du domaine. Si une valeur est supprimée, toutes les contraintes (les paires d'arcs) contenant cette valeur sont ajoutées dans Q pour une nouvelle révision si elles n'y sont pas déjà.

Algorithme 1.1 : Revise$((i, j))$

```
   Data : (i, j) an arc
   Result : true if dom(i) is reduced ; false otherwise
1  DELETE ← false;
2  foreach a ∈ dom(i) do
3      if ∄b ∈ dom(j) such that P_ij(a, b) then
4          dom(i) ← dom(i)\{a};
5          DELETE ← true;
6  return DELETE;
```

Algorithme 1.2 : AC3

```
1  Q ← {(i, j) | (i, j) ∈ arcs(G), i ≠ j};
2  while Q ≠ ∅ do
3      select and delete any arc (k, m) from Q;
4      if Revise((k, m)) then
5          Q ← Q ∪ {(i, k) | (i, k) ∈ arcs(G), i ≠ k, i ≠ m};
```

Nous allons présenter l'algorithme GAC (*Generalized Arc Consistency*) qui généralise l'algo-

rithme 1.2 pour l'appliquer sur les réseaux contenant des contraintes n-aires.

Algorithme 1.3 : GAC($\mathcal{P}$))

Data : $\mathcal{P} = (\mathcal{X}, \mathcal{C})$ a network
Result : *true* if $\mathcal{P}$ is GAC ; *false* otherwise
1 $Q \leftarrow \{\langle x, c\rangle \mid c \in \mathcal{C}$ and $x \in scp(c)\}$;
2 **while** $Q \neq \emptyset$ **do**
3 select and delete any $\langle x, c\rangle$ from Q;
4 $ND_x \leftarrow \{a \mid a \in \texttt{dom}(x)$ and $\exists\{y_1, ..., y_m\}$ s.t. $y_k \in scp(c), 1 \leq k \leq m$ and $\{x = a, y_1 = b_1, ..., y_m = b_m\} \in rel(c)$ where $b_k \in \texttt{dom}(y_k)\}$;
5 **if** $ND_x = \emptyset$ **then**
6 **return** *false*;
7 **else**
8 $\texttt{dom}(x) \leftarrow ND_x$;
9 $Q \leftarrow Q \cup \{\langle y, c'\rangle \mid x \in scp(c'), c' \in C, c \neq c', y \in scp(c'), y \neq x\}$;
10 **return** *true*;

La ligne 4 correspond à `Revise` (algorithme 1.1) appliqué aux contraintes n-aires, l'idée est de lister les valeurs de x dans les supports de la contrainte, et ensuite de comparer l'ensemble obtenu avec le domaine initial de la variable actuellement traitée.

1.2.1.2 Cohérence de chemin

La cohérence de chemin est similaire à la cohérence d'arc. Au lieu de vérifier la cohérence d'une variable unique avec une autre, elle considère des couples de variables avec une autre. C'est une forme de cohérence plus forte lorsque combiné à la cohérence d'arc plutôt que la cohérence d'arc utilisée seule. Par contre, celle-ci vient avec des désavantages, notamment une complexité plus élevée.

Définition 10 (Cohérence de chemin)

Soient trois variables x, y et z et trois contraintes c_1, c_2 et c_3 avec $scp(c_1) = \{x, y\}$, $scp(c_2) = \{x, z\}$ et $scp(c_3) = \{y, z\}$. Les variables x et y sont **chemin cohérentes** avec z si, pour chaque paire $(a, b) \in rel(c_1)$ de valeurs, il existe une valeur $c \in \texttt{dom}(z)$ telle que (a, c) et (b, c) satisfont respectivement les contraintes c_2 et c_3.

Exemple 3

Soit la figure 1.1 représentant trois variables x, y et z différentes deux à deux. Si nous considérons que le domaine de ces variables est équivalent et égal à $\{0, 1\}$. Avec la cohérence d'arc, toutes les valeurs du réseau sont cohérentes. En effet, en considérant les variables par paire, les tuples $(0, 1)$ et $(1, 0)$ sont supports de la contrainte $\neq$. Avec la cohérence de chemin, le réseau est directement trouvé inconsistant. Par exemple, en considérant le couple de variables x et y et les tuples supports de la contrainte qui les lie

$(0, 1)$ et $(1, 0)$, il nous est impossible de trouver une valeur cohérente dans le domaine de la variable z qui pourrait satisfaire à la fois la contrainte de différence entre x et z et celle entre y et z.

1.2.1.3 Propagateurs

Dans cette section nous allons présenter deux propagateurs afin d'illustrer le filtrage de contraintes. Nous avons choisi une contrainte simple (dite primitive) entre deux variables : l'*inégalité*. Nous décrivons aussi l'algorithme de propagation d'une contrainte un peu plus complexe : la *somme inférieure ou égale*. Commençons avec cette dernière.

Algorithme 1.4 : SumLessEqualPropagate())

Data : $c \equiv sum(X_i) \leq limit$
Result : return $true$ if a domain is emptied

```
1  min ← 0;
2  max ← 0;
3  for X_i ∈ scp(c) do
4      min ← min + lb(X_i);
5      max ← max + ub(X_i);
6  if max <= limit then
7      return false;
8  if min > limit then
9      return true;
10 for X_i ∈ scp(c) do
11     max ← max − ub(X_i);
12     curLim ← limit − (min − lb(X_i));
13     for a ∈ dom(X_i) do
14         if a > curLim then
15             dom(X_i) ← dom(X_i)\{a};
16     if dom(X_i) = ∅ then
17         return true;
18     max ← max + ub(X_i);
19     if max <= limit then
20         return false;
21 return false
```

L'algorithme 1.4 est le filtrage d'une contrainte c qui est $sum(X_i) \leq limit$, les lignes 1 à 5 servent à calculer les bornes de la somme, c'est-à-dire le minimum et le maximum obtenables avec les domaines courants des variables dans la portée de c. Pour cela les fonctions `lb` et `ub` sont utilisées. Elles retournent respectivement la valeur minimale et celle maximale d'une variable passée en paramètre. Les lignes 7-8 permettent de ne pas effectuer le filtrage si la somme des valeurs maximales est inférieure (ou égale) à la limite de la contrainte. Dans ce cas, rien ne

pourra être filtré. Ce test est effectué aussi à chaque fin de boucle (lignes 19-20). Les lignes 8-9 effectuent une vérification similaire mais dans l'autre sens, c'est-à-dire que si la somme des valeurs minimales est supérieure à la limite de la contrainte, la contrainte est clairement incohérente. La suite de l'algorithme est une itération sur toutes les variables de la portée de la contrainte et permet de filtrer les valeurs incohérentes. Les lignes 10 et 18 maintiennent à jour la borne maximale de la somme tout au long du filtrage afin de passer outre les suppressions potentielles. La ligne 12 calcule une limite locale à la variable présentement considérée afin de pouvoir regarder quelle valeur de son domaine est cohérente avec les bornes de la somme globale. Pour cela, nous retranchons à la limite de la contrainte le minimum calculé précédemment puis nous ajoutons le minimum de la variable considérée. Cela permet de vérifier les valeurs une à une en simulant le cas où toutes les autres variables de la portée sont affectées à leur minimum. Les lignes 13 à 15 suppriment concrètement toutes les valeurs du domaine de la variable actuellement considérée qui sont supérieures à la limite locale calculée ligne 12.

Algorithme 1.5 : NotEqualPropagate())

Data : A constraint $c \equiv x \neq y$
Result : return *true* if a domain is emptied

```
1  if |dom(x)| = 1 then
2      Let a be the only value in dom(x);
3      remove a in dom(y);
4      if dom(y) = ∅ then
5          return true

6  if |dom(y)| = 1 then
7      Let a be the only value in dom(y);
8      remove a in dom(x);
9      if dom(x) = ∅ then
10         return true

11 return false
```

L'algorithme 1.5 est le filtrage d'une contrainte dite primitive car c'est une contrainte en intension sur deux variables (x et y) avec un seul opérateur ($\neq$). Il filtre lorsqu'une des variables est affectée afin d'éviter que la seconde le soit à la même valeur. Une incohérence est détectée si les deux variables ont la même valeur.

1.2.2 Méthodes de résolution

Il existe de nombreuses méthodes de résolution, la plus simple étant le *generate and test* qui consiste à générer un ensemble de valeurs et à tester si celui-ci est cohérent. Afin d'obtenir l'ensemble des solutions d'un problème simple tel que les 8-reines [1], cet algorithme génère et vérifie la validité de $8^8 = 16\ 777\ 216$ instanciations différentes. Clairement cette approche est irréalisable en pratique. Nous allons voir dans la suite de cette section des méthodes qui ont été utilisées ou le sont à l'heure actuelle.

1. Ce problème consiste à placer n reines sur un échiquier $n \times n$ afin qu'aucune ne soit en prise avec une autre en suivant les règles des échecs, c'est-à-dire que 2 reines ne peuvent pas être sur la même ligne, la même colonne ou la même diagonale.

1.2.2.1 *Backtrack*

Le *backtrack* (ou retour arrière), consiste à associer à chaque variable une valeur de son domaine. S'il n'y a pas de valeurs supportées, il suffit de faire un retour arrière pour changer la variable précédente, etc. Sa complexité est en $O(d^n)$ (où d est la taille maximum des domaines et n le nombre de variables).

Algorithme 1.6 : BacktrackSearch($\mathcal{P}$)

Data : $\mathcal{P} = (\mathcal{X}, \mathcal{C})$ a network
Result : true if SAT
if $\exists x \in \mathcal{X}, \texttt{dom}(x) = \emptyset$ **then**
  return $false$;
if $\forall x \in \mathcal{X}, |\texttt{dom}(x)| = 1$ **then**
  return $true$;
select a value (x, a) of $\mathcal{P}$ such that $|\texttt{dom}(x)| > 1$;
return BacktrackSearch($\mathcal{P}|_{x=a}$) $\vee$ BacktrackSearch($\mathcal{P}|_{x\neq a}$);

Contrairement au *generate and test*, il est possible de réduire le nombre d'instanciations considérées. Cependant, le *backtrack* devient efficace dès lors qu'un processus de filtrage lui est associé. Cette combinaison a conduit à l'élaboration des approches de type MAC (*Maintaining Arc consistency*) (Sabin et Freuder (1994a;b)), qui sont les plus utilisées à l'heure actuelle.

1.2.2.2 *Forward Checking* (FC) et *Maintaining Arc Consistency* (MAC)

Le *forward checking* est une technique qui consiste à maintenir une cohérence sur les nœuds adjacents (qui ont une contrainte qui les lie) après l'instanciation d'une variable. Le *forward checking* est une forme affaiblie de MAC, qui maintient la cohérence d'arc sur le réseau complet. MAC consiste à intégrer un filtrage ϕ au backtrack, en l'occurrence (G)AC. Avec une heuristique appropriée, l'algorithme MAC est, en général, plus efficace.

Algorithme 1.7 : MAC($\mathcal{P}$)

Data : $\mathcal{P} = (\mathcal{X}, \mathcal{C})$ a network
Result : true if SAT
$\mathcal{P} \leftarrow (G)AC(\mathcal{P})$;
if $\exists x \in \mathcal{X}, \texttt{dom}(x) = \emptyset$ **then**
  return $false$;
if $\forall x \in \mathcal{X}, |\texttt{dom}(x)| = 1$ **then**
  return $true$;
select a value (x, a) of $\mathcal{P}$ such that $|\texttt{dom}(x)| > 1$;
return MAC($\mathcal{P}|_{x=a}$) $\vee$ MAC($\mathcal{P}|_{x\neq a}$);

L'algorithme précédent représente une version pseudo code de l'algorithme MAC. Il prend en entrée un réseau de contraintes et retourne *vrai* si une affectation complète des variables a été trouvée sans violer de contraintes. Il consiste à choisir une variable et à réaliser un processus de filtrage tant qu'une solution n'a pas été trouvée ou tant que l'instanciation courante n'est pas contradictoire avec une contrainte. Dans ce dernier cas un retour arrière est réalisé. Dans la situation où le retour arrière est réalisé sur l'instanciation vide alors le problème est globalement montré insatisfiable.

Malgré l'utilisation d'heuristiques dynamiques et d'algorithmes avancés comme MAC, il existe certains cas où le solveur peut perdre une quantité importante de temps de recherche dans des impasses (phénomène *heavy-tail*, Gomes *et al.* (2000)). Pour contrer ce problème, les redémarrages ont été introduits (par exemple, la suite de Luby *et al.* (1993)). En effet, si un algorithme n'obtient pas de solutions après un certain nombre de retours arrière, un redémarrage est lancé. Cette politique bien qu'améliorant les résultats ne suffit pas du fait qu'à chaque redémarrage l'arbre de recherche précédemment exploré est totalement oublié. Afin de pallier cela, nous allons voir dans la suite comment rendre les redémarrages plus efficaces grâce aux heuristiques et à l'apprentissage de *nogoods*.

1.3 Heuristiques

> « –Lookahead and anticipate the future in order to succeed in the present.
> –To succeed, try first where you are most likely to fail.
> –Remember what you have done to avoid repeating the same mistake.
> –Lookahead to the future in order not to worry about the past »
>
> *Robert M. Haralick and Gordon L. Elliott 1980*

Les heuristiques permettent de guider la recherche et, grâce à des choix judicieux, d'améliorer l'efficacité de l'exploration de l'espace de recherche au sein des solveurs. Elles interviennent dans plusieurs étapes de la recherche. Elles sont importantes car elles permettent de guider la recherche et ainsi d'éviter une partie du *thrashing*, le fait d'explorer de manière répétée des sous-arbres incohérents semblables. De très nombreuses heuristiques ont été proposées dans la littérature et le sont encore aujourd'hui (Habet et Terrioux (2019)). Dans cette section, nous allons présenter majoritairement des heuristiques de sélection de variables ainsi que quelques heuristiques de choix de valeur.

1.3.1 Choix de valeur

Nous présentons brièvement dans cette section une sélection d'heuristiques usuelles de choix de valeur. En général ces heuristiques choisissent les valeurs les plus prometteuses pour la variable associée dans le but de trouver une solution plus rapidement, et d'éviter d'aller dans des sous arbres incohérents.

min-value L'heuristique `min-value` sélectionne la valeur minimale du domaine de la variable considérée. Il est aussi possible de sélectionner le maximum (`max-value`). Cette heuristique peut être adaptée en fonction du type de problème afin d'être plus performante.

min-conflicts L'heuristique `min-conflicts` (Frost et Dechter (1995)) sélectionne la valeur du domaine de la variable qui est le moins impliquée dans des conflit lors de la recherche. Elle permet de se rapprocher du principe de valeur prometteuse souvent admis en programmation par contraintes. À l'instar de `min-value`, l'heuristique `min-conflicts` admet aussi une opposée, `max-conflicts`. Cette dernière permet d'orienter aussi la sélection de valeur vers un paradigme *fail-first*.

first-value L'heuristique `first-value`, aussi appelée `lexico` dans certain cas, est une autre façon de choisir la valeur d'une variable. Elle se base sur l'implémentation des domaines au sein d'un solveur. Par exemple, ceux utilisant un ensemble éparse (*Sparse Set*) (Aho *et al.* (1974)) afin de représenter les domaines des variables peuvent bénéficier de cette heuristique. Ce type d'implémentation représente les domaines grâce à deux ensembles, *sparse* et *dense* (une forme est présentée dans le chapitre 6 de la partie 2). Elle sélectionne le premier indice (voire le dernier dans de rare cas) dans l'ensemble *sparse* afin de faire son choix car l'ordre des éléments de celui-ci est modifié au cours de la recherche (lors de la suppression de valeur notamment).

Ces heuristiques de choix de valeur doivent être combinées aux heuristiques de choix de variable. Une sélection de ces dernières est présentée dans la section suivante.

1.3.2 Choix de variable

Les heuristiques de choix de variables se découpent en trois catégories principales : statiques, c'est-à-dire que l'ordonnancement peut être défini dès le début de la recherche ; dynamiques, l'ordonnancement se concentre uniquement sur le nœud courant (là où nous sommes arrivés dans l'arbre de recherche) et donc peut varier au cours de la recherche, ces heuristiques utilisent des paramètres du problème, la taille du domaine courant par exemple ; et enfin adaptatives, la sélection se fait par une forme d'apprentissage grâce à des mesures supplémentaires réalisés à chaque nœud exploré. Le but des heuristiques adaptatives est, en général, lorsque associées aux redémarrages de corriger une partie du *thrashing* en identifiant les parties difficiles du problème (les variables qui rentrent le plus souvent en conflit). Dans cette section, les heuristiques les plus utilisées sont présentées, elle n'a pas vocation à être exhaustive. Notamment, le raisonnement sur les derniers conflits (Lecoutre *et al.* (2009)) n'est pas présenté ici. Il arrive lors de la sélection de variable que plusieurs variables aient le même score. Dans ce cas il faut gérer les égalités (*tie*). Des heuristiques combinant le choix de variable ainsi que la gestion des égalités ont été proposé dans l'état de l'art, du type `dom+deg` ou encore `bz` (`dom+ddeg`, Brélaz (1979)). L'heuristique décrite avant le `+` sert pour la sélection et celle après est utilisée pour casser les égalités (*tie break*).

1.3.2.1 Heuristiques statiques

Ces heuristiques utilisent uniquement les informations disponibles à l'état initial du problème.

lexico L'heuristique `lexico` est une heuristique statique qui, comme son nom l'indique, ordonne les variables par ordre lexicographique.

deg L'heuristique `deg` (Ullmann (1976)) (aussi appelée `max-deg`) est une heuristique statique qui utilise le degré dans l'ordre décroissant afin de choisir les variables. Pour rappel, le degré d'une variable est le nombre de contraintes dans lesquelles elle apparaît.

1.3.2.2 Heuristiques dynamiques

Les heuristiques dynamiques prennent en compte l'état courant du problème. L'ordre de sélection des variables varie durant la recherche.

ddeg L'heuristique `ddeg` est très similaire à `deg` mais utilise le degré dynamique à la place du degré simple. Le degré dynamique d'une variable x est le nombre de contraintes où elle apparaît et dans lesquelles au moins une des variables (différente de x) de la portée de la contrainte n'est pas affectée.

dom L'heuristique `dom` (Haralick et Elliott (1980)), est l'une des premières heuristiques dynamiques. Elle se base sur la taille du domaine courant des variables et choisit celle dont le domaine est le plus petit comme prochaine variable à affecter.

dom/deg – dom/ddeg Il est possible de combiner les heuristiques pour obtenir de meilleurs résultats. Par exemple, l'heuristique `dom/deg` (Bessière et Régin (1996)) est en général plus efficace que `dom` utilisée seule. Le principe est de choisir la variable ayant le plus petit ratio de taille du domaine courant sur le degré (dynamique) de la variable (le nombre de contraintes dans lesquelles elle est impliquée).

1.3.2.3 Heuristiques adaptatives

Les heuristiques adaptatives peuvent être vue comme un système d'apprentissage car elles permettent de faire des choix basés à la fois sur l'état courant du problème et les états précédents.

Weighted degree (wdeg) L'heuristique `wdeg` (Boussemart *et al.* (2004)) consiste à choisir les variables apparaissant dans les contraintes les plus souvent falsifiées. Pour cela un compteur, initialisé à 1, est associé à chaque contrainte. Il est incrémenté lorsque la contrainte qui lui est associée est violée au cours de la recherche. Il est ensuite possible de savoir où se situent les parties les plus difficiles du problème pour les traiter en priorité et ainsi essayer de résoudre des conflits avant d'affecter d'autres variables. Le degré d'une variable est calculé de la manière suivante, nous effectuons la somme des compteurs de toutes les contraintes impliquant celle-ci et au moins une autre variable non affectée.

dom/wdeg L'heuristique `dom/wdeg` (Boussemart *et al.* (2004)) consiste à sélectionner en priorité la variable avec le plus petit ratio taille du domaine courant sur degré pondéré courant. L'intérêt de cette heuristique vient du fait qu'elle soit adaptative et qu'elle permette de se focaliser sur les parties difficiles de l'instance.

Impact L'heuristique `Impact` (Refalo (2004)) est une heuristique dynamique, elle mesure l'importance des variables dans la réduction de l'espace de recherche. Ces mesures sont faites grâce à l'observation des réductions de domaines lors de la recherche. Les variables ayant le plus d'impact sur l'espace de recherche sont sélectionnés en priorité.

Activity L'heuristique `Activity` (Michel et Hentenryck (2012)) se base sur l'activité des variables lors de la propagation afin de guider la recherche. Elle compte le nombre de fois où le domaine d'une variable est réduit afin de choisir la variable ayant le plus d'activité de réduction. S'inspirant de l'heuristique SAT `VSIDS` (Moskewicz *et al.* (2001)), elle applique l'oubli afin de pouvoir travailler avec des informations récentes.

1.4 Conclusion

Au cours de ce chapitre, nous avons présenté la programmation par contraintes et plus particulièrement le problème de satisfaction de contraintes de manière formelle. Nous avons vu à la fois quelques niveaux de cohérence (cohérence d'arc et cohérence de chemin), des algorithmes de filtrage de contraintes (propagateurs de somme inférieure ou égale ainsi que de la contrainte $x \neq y$), des méthodes de résolution complète (*backtrack*, *forward checking* et *maintaining arc consistency*) et des heuristiques de choix de variables (statiques, dynamiques et adaptatives) et de valeurs (*min-value*, *min-conflicts* et *first-value*). Les heuristiques que nous avons vues permettent de corriger une partie du *thrashing*, c'est-à-dire l'exploration par l'algorithme de recherche des mêmes sous-arbres incohérents de manière répétée. Afin d'éviter ce phénomène, il est intéressant de combiner celles-ci à l'apprentissage de *nogoods*, qui ont pour but d'empêcher de répéter les états incohérents détectés lors de la recherche. Nous présentons cela dans le chapitre 2.

Chapitre 2

Apprentissage en programmation par contraintes

Sommaire

Un *nogood* est une instanciation partielle qui ne peut être prolongée vers une solution. L'intérêt des *nogoods* est d'éviter le phénomène de *thrashing*, c'est-à-dire d'explorer de manière répétée les mêmes sous-arbres incohérents. Deux approches classiques existent pour les identifier et les enregistrer que ce soit au cours de la recherche ou au redémarrage.

L'apprentissage de *nogoods* est un thème qui a été introduit dans les années 90 (Dechter (1990), Schiex et Verfaillie (1994), Frost et Dechter (1994)) pour la résolution de problèmes de satisfaction de contraintes. Les *nogoods* classiques, dits standards, sont des instanciations partielles ne pouvant mener à aucune solution. Ils ont été assez rapidement utilisés pour gérer l'explication (Ginsberg (1993), Jussien *et al.* (2000)) de valeurs supprimées au cours de la recherche et de la propagation de contraintes. Ils ont ensuite été généralisés (Katsirelos et Bacchus (2003), abordés en section 4.1) en permettant la combinaison d'affectations de variables (décisions positives) et de réfutations de valeurs (décisions négatives). L'intérêt pratique des *nogoods* (généralisés) a été revisité par les travaux portant sur la génération paresseuse de clauses (Feydy et Stuckey (2009), détaillée en section 4.3).

Les *nogoods* peuvent également être utiles dans un contexte de redémarrage régulier d'un algorithme de recherche arborescente. Il est en effet possible d'identifier (Lecoutre *et al.* (2007a;b)) sur la dernière branche de l'arbre de recherche (traditionnellement représentée par celle qui est la plus à droite) un ensemble de *nogoods* représentant la partie de l'espace de recherche qui vient d'être explorée. Par le fait de simplement enregistrer ces *nogoods*, appelés *nld-nogoods* (réduits), nous avons la garantie de ne jamais explorer de nouveau les mêmes sous-arbres. Certaines extensions de ces travaux ont porté sur l'élimination de symétries (Lecoutre et Tabary (2011), Lee et Zhu (2014)) et l'exploitation du caractère croissant des *nld-nogoods* (Lee *et al.* (2016)), appelés de ce fait *increasing-nogoods*.

Dans ce chapitre, nous présentons l'apprentissage en programmation par contraintes, basé sur le concept de *nogoods*. Dans un premier temps nous nous intéressons aux *nld-nogoods* puis nous montrons les dernières avancées dans le domaine de l'apprentissage en programmation par contraintes à savoir les *increasing-nogoods*. Dans les deux cas, nous décrivons les algorithmes de filtrages et les manières de les minimiser.

2.1 *Nogoods* standards

Un *nogood* est un ensemble de décisions qui n'apparaît dans aucune solution d'un problème donné, il ne pourra donc jamais être étendu à une solution dudit problème. L'idée est de garder une trace des calculs effectués pour ne pas avoir à les refaire et de les stocker dans une base de connaissances. Celle-ci pour être efficace doit persister lors des redémarrages car pour un algorithme de résolution avec retours arrière, les *nogoods* ne servent qu'après un redémarrage pour élaguer les branches déjà parcourues et connues comme insatisfiables.

Définition 11 (Décision positive/négative)

Soit $\mathcal{P} = (\mathcal{X}, \mathcal{C})$ un réseau de contraintes et (x, v) un couple tel que $x \in \mathcal{X}$ et $a \in \texttt{dom}(x)$. L'affectation $x = a$ est appelée **décision positive** tandis que $x \neq a$ est appelée **décision négative**.

Remarque 3

$\neg(x = a)$ (resp. $\neg(x \neq a)$) signifie $x \neq a$ (resp. $x = a$).

Définition 12 (Nogood standard)

Un ***nogood* standard** est un ensemble de décisions positives, c'est-à-dire d'instanciations de la forme $X = a$, parfois noté $X \leftarrow a$, qui ne sont cohérentes avec aucune solution.

2.2 Enregistrement et exploitation des *nogoods*

Les *nogoods* ont initialement été développés pour la programmation par contraintes mais l'absence de résultats convaincants pendant de longues années n'a pas permis de retenir cette technique. Celle-ci a par contre obtenu des résultats en satisfaction booléenne (SAT) notamment grâce aux *watched literals* (Zhang *et al.* (2001), présentés section 3.4.2) combinés avec l'heuristique *VSIDS* (présentée section 3.4.3) et les retours arrière non chronologiques (*backjump*). Ces techniques sont présentées dans le chapitre 3 de l'état de l'art.

Pour rendre cette technique efficace en CP, des optimisations sont nécessaires, notamment, limiter la taille des *nogoods*. Traiter les *nogoods* revient ainsi à ajouter des contraintes à satisfaire.

Cela alourdit le problème et augmente le temps d'exécution des algorithmes de cohérence, donc limiter leur taille est capital. Afin de pallier ce problème, en SAT, les *nogoods* enregistrés ne sont pas systématiquement minimaux et l'enregistrement de *nogoods* est restreint aux *nogoods* inférieurs à une certaine taille définie auparavant et à un seul par échec, selon le principe du *First Unique Implication Point* (1-UIP, Zhang *et al.* (2001)). Le but n'est pas de tous les générer mais de garder ceux qui permettent d'élaguer le plus de branches en ayant le moins d'impact sur le temps du processus de recherche.

Définition 13 (*nogood* minimal)

Un **nogood minimal** est un *nogood* dans lequel toutes les décisions participent à l'explication de l'échec. Autrement dit, si nous supprimons une décision de ce *nogood*, celui-ci ne garantit plus de mener à une impasse et par conséquent ce n'est plus un *nogood*.

Définition 14 (*nogood* minimum)

Un **nogood minimum** est le plus petit *nogood* (en terme de nombre de décisions) qu'il est possible d'extraire et expliquant l'échec d'une instanciation.

Remarque 4

Un *nogood* minimum est minimal mais la réciproque n'est pas forcément vraie.

Remarque 5

Pour un *nogood* non minimisé donné, il peut exister plusieurs *nogood* minimum différents.

Utiliser des *nogoods* minimums permet de simplifier encore plus l'arbre de recherche moyennant un coût de calcul (au moment de leur création) plus important. En contrepartie, leur coût de stockage est moindre (moins de variables dans les *nogoods* minimaux), tout comme leur coût d'exploitation.

2.3 *Negative last decision nogoods*

Une première technique d'identification des *nogoods* est l'énumération, c'est-à-dire à partir d'un *nogood* identifié lors d'un conflit, nous énumérons tous les *nogoods* de taille 1, puis 2, etc. Cette approche n'est clairement pas envisageable. Une autre approche, les *nld-nogoods* (Lecoutre *et al.* (2007b)), semble bien plus efficace. Ces *nogoods* minimalisés sont enregistrés au redémarrage et l'utilisation des *watched literals* (voir chapitre 3) avec propagation permet de les utiliser efficacement.

2.3.1 Découverte et enregistrement

Afin de découvrir ces *nld-nogoods* lors des redémarrages, nous devons les définir ainsi que les nld-sous-séquences qui permettent de les identifier.

Définition 15 (nld-sous-séquence)

Soit $\Sigma = \langle \delta_1, ..., \delta_i, ..., \delta_m \rangle$ une séquence de décisions où δ_i est une décision négative. La séquence $\langle \delta_1, ..., \delta_i \rangle$ est appelée **nld-sous-séquence** (*negative last decision subsequence*) de Σ.

L'ensemble de décisions positives (resp. négatives) de Σ est noté $pos(\Sigma)$ (resp. $neg(\Sigma)$).

Définition 16 (*nld-nogood*)

Soit $\mathcal{P}$ un réseau de contraintes et Σ une séquence de décisions prises sur une branche de l'arbre de recherche menant à un sous-arbre incohérent. Pour toute nld-sous-séquence $\langle \delta_1, ..., \delta_i \rangle$ de Σ, l'ensemble $\Delta = \{\delta_1, ..., \delta_{i-1}, \neg\delta_i\}$ est un *nogood* de $\mathcal{P}$ appelé ***nld-nogood*** (pour chaque branche de l'arbre de recherche un *nogood* peut être extrait de chaque décision négative).

Ce type de *nogood* contient à la fois des décisions positives et négatives, ce qui correspond à la définition des *nogoods* généralisés. Nous allons par la suite montrer que les *nld-nogoods* peuvent être réduits en taille en ne gardant que les décisions positives.

Définition 17 (*nld-nogood* réduit)

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

La complexité spatiale pour stocker tous les *nld-nogoods* de Σ est $O(n^2d^2)$ tandis que celle pour stocker les *nld-nogoods* réduits est de $O(n^2d)$.

Pour profiter de ces *nld-nogoods*, Lecoutre *et al.* (2007b) ont proposé de les utiliser de la manière suivante : les *nld-nogoods* sont découverts sur la dernière branche parcourue de l'arbre de recherche avant chaque redémarrage ; le but étant de bénéficier à la fois des redémarrages et des capacités d'apprentissage sans sacrifier les performances aussi bien en temps qu'en espace. Pour cela, ils se basent sur la proposition suivante :

Proposition 1

Tous les *nld-nogoods* réduits extraits de la dernière branche de l'arbre de recherche subsument tous les *nld-nogoods* réduits qui pourraient être extraits des branches précédemment explorées.

Théoriquement afin d'obtenir ces *nld-nogoods* réduits et retirer les décisions négatives, il nous faut utiliser la résolution appliquée aux contraintes.

Définition 18 (Résolution appliquée aux contraintes)

Soit $\mathcal{P}$ un réseau de contraintes, et soient $\Delta_1 = \Gamma \cup \{x_i = a_i\}$ et $\Delta_2 = \Lambda \cup \{x_i \neq a_i\}$ deux *nogoods* de $\mathcal{P}$. La **résolution appliquée aux contraintes** se définit comme suit : $\text{C-Res}(\Delta_1, \Delta_2) = \Gamma \cup \Lambda$.

Remarque 6

$\text{C-Res}(\Delta_1, \Delta_2)$ est un *nogood* de $\mathcal{P}$.

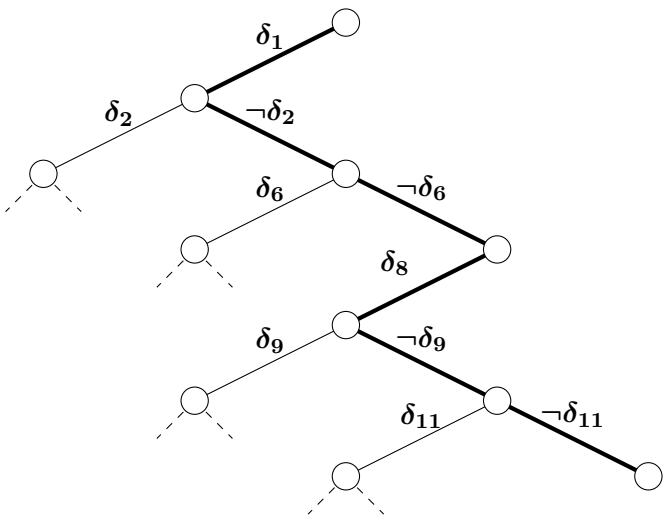


FIGURE 2.1 – Arbre de recherche avant redémarrage

Exemple 4

Supposons que l'algorithme soit sur le point d'effectuer un redémarrage, et que nous ayons l'arbre de recherche illustré par la figure 2.1. La dernière branche de l'arbre

de recherche est $\Sigma = \{\delta_1, \neg\delta_2, \neg\delta_6, \delta_8, \neg\delta_9, \neg\delta_{11}\}$. Nous pouvons en extraire les 4 *nld-nogoods* suivants :

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

En utilisant maintenant la résolution appliquée aux contraintes (définition 18), nous pouvons obtenir les *nld-nogoods* réduits suivant :

$$\begin{aligned}\Delta'_1 &= \text{C-Res}(\text{C-Res}(\text{C-Res}(\Delta_1, \Delta_2), \Delta_3), \Delta_4)\\ &= \text{C-Res}(\text{C-Res}(\{\delta_1, \neg\delta_2, \neg\delta_6, \delta_8, \delta_{11}\}, \Delta_3), \Delta_4)\\ &= \text{C-Res}(\{\delta_1, \neg\delta_2, \delta_8, \delta_{11}\}, \Delta_4)\\ &= \{\delta_1, \delta_8, \delta_{11}\}\end{aligned}$$
$$\Delta'_2 = \text{C-Res}(\text{C-Res}(\Delta_2, \Delta_3), \Delta_4) = \{\delta_1, \delta_8, \delta_9\}$$
$$\Delta'_3 = \text{C-Res}(\Delta_3, \Delta_4) = \{\delta_1, \delta_6\}$$
$$\Delta'_4 = \Delta_4 = \{\delta_1, \delta_2\}$$

Algorithme 2.1 : StoreNogoods($\Sigma = \langle\delta_1, ..., \delta_m\rangle$)

```
   Data : Σ is the sequence of literals labelling the current branch
1  Δ ← ∅;
2  for i ∈ [1, m] do
3      if δ_i ∈ pos(Σ) then
4          Δ ← Δ ∪ {δ_i};
5      else
6          if Δ = ∅ then
7              with δ_i = (x, v), remove v from dom(x);
8          else
9              addNogood(Δ ∪ {¬δ_i});
```

L'algorithme 2.1 montre comment stocker les *nld-nogoods* réduits lors d'un redémarrage, dans le pire cas sa complexité en temps est en $O(pn)$ avec p le nombre de décisions positives et n le nombre de décisions négatives de la branche courante. L'idée générale est de parcourir la branche depuis la racine, lorsqu'une décision positive est rencontrée, elle est enregistrée dans Δ. Quand une décision négative est rencontrée, un *nld-nogood* est enregistré à partir de sa négation et de Δ (tous les positifs précédemment rencontrés). Si le *nld-nogood* est de taille 1 (c'est-à-dire $\Delta = \emptyset$), la valeur du domaine de la variable impliquée peut directement être supprimée. Il suffit de stocker

les *nld-nogoods* ainsi calculés dans une structure gérant les *watched literals* (voir chapitre 3).

Algorithme 2.2 : GAC+nogoods($\mathcal{P}$)

Data : $\mathcal{P} = (\mathcal{X}, \mathcal{C})$ a network
1 $Q \leftarrow \{\langle x, c\rangle \mid c \in \mathcal{C} \text{ and } x \in scp(c)\}$;
2 **while** $Q \neq \emptyset$ **do**
3 select and delete any $\langle X, c\rangle$ from Q;
4 **if** $|\texttt{dom}(x)| = 1$ **then**
5 Let a be the only value left in $\texttt{dom}(x)$;
6 **if** *checkWatches*$(x = a) = false$ **then**
7 **return** $false$;
8 $ND_x \leftarrow \{a \mid a \in \texttt{dom}(x)$ and $\exists\{y_1, ..., y_m\}$ s.t. $y_k \in scp(c), 1 \leq k \leq m$ and $\{x = a, y_1 = b_1, ..., y_m = b_m\} \in c$ where $b_k \in \texttt{dom}(y_k)\}$;
9 **if** $ND_x = \emptyset$ **then**
10 **return** $false$;
11 **else**
12 $\texttt{dom}(x) \leftarrow ND_x$;
13 $Q \leftarrow Q \cup \{\langle y, c'\rangle \mid x \in scp(c'), c' \in C, c \neq c', y \in scp(c'), y \neq x\}$;

Afin d'exploiter les *nld-nogoods* réduits, il faut utiliser des algorithmes dérivés de ceux présentés dans la section 1.2.1. L'algorithme 2.2 (GAC+nogoods) est une version modifiée de l'algorithme 1.3 (GAC), il permet d'exploiter les *nogoods* lors de l'exécution de MAC. Les lignes 4 à 6 permettent l'exploitation des *nogoods*. La fonction *checkWatches* permet de vérifier la cohérence de la base de *nogoods* et de passer les *nogoods* qui contiennent la valeur v comme *watched*. La fonction *checkWatches* a une complexité en $O(n\gamma)$ (où n est le nombre de variables et γ le nombre de *nogoods* réduits dans la base de connaissances) dans le pire cas.

Nous pouvons remarquer qu'il existe des similitudes entre les différents *nld-nogoods* extraits lors d'un même redémarrage, ils sont dits croissants. Les *increasing-nogoods* (Lee *et al.* (2016), Lee et Zhu (2014)) présentés dans la section 2.4 exploitent ce caractère croissant et permettent de représenter les *nld-nogoods* réduits extraits au redémarrage sous la forme d'une (seule) contrainte globale.

2.3.2 Minimisation des *nld-nogoods*

Dans l'extension de leur travaux, Lecoutre *et al.* (2007b) proposent des méthodes de minimisation des *nld-nogoods* réduits.

Définition 19 (Φ-*nogood* (minimal))

Soit Φ un opérateur d'inférence et Δ un ensemble de décisions. 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 $\Phi(P|_{\Delta'}) = \bot$.

Pour construire les Φ-*nogoods* minimaux depuis les *nld-nogoods* réduits, il est nécessaire d'identifier itérativement les décisions qui participent au conflit.

Définition 20 (Décision de transition)

Soit Φ un opérateur d'inférence et un *nld-nogood* $\Delta = \langle \delta_1, \delta_2, \ldots, \delta_m \rangle$ d'un réseau de contraintes P, il existe une décision δ_i tel que $\Phi(P|_{\{\delta_1,\ldots,\delta_{i-1}\}}) \neq \bot$ et $\Phi(P|_{\{\delta_1,\ldots,\delta_i\}}) = \bot$. δ_i est appelé **décision de transition** de Δ.

Remarque 7

Toutes décisions δ_j où $j > i$ peuvent être supprimées sans problèmes du *nld-nogood* réduit.

Il est possible d'utiliser trois types d'approches afin d'identifier les décisions de transition : constructive, destructive et dichotomique. Dans la première, il suffit d'ajouter (en conservant l'ordre initial du *nld-nogood* réduit) les décisions une par une jusqu'à obtenir un conflit ($\Phi(P|_\Delta) = \bot$). La méthode destructive consiste à ajouter toutes les décisions de Δ puis de les retirer une par une jusqu'à ce que $\Phi(P|_\Delta) \neq \bot$. Après avoir trouvé la première décision de transition δ_i de Δ, il est possible de continuer à en chercher, après avoir supprimé les décisions δ_j avec $j > i$. En appliquant ce processus incrémentalement, nous obtenons un *nogood* constitué uniquement de décisions de transition.

2.4 *Increasing nogoods*

Les *increasing-nogoods* (*IncNG*) (Lee et Zhu (2014), Lee *et al.* (2016)) sont une extension des *nld-nogoods* réduits. À chaque redémarrage une seule contrainte globale *IncNG* est ajoutée au réseau. Celle-ci représente tous les *nld-nogoods* réduits qui auraient pu être extraits lors de la recherche. Le but est de mettre à disposition une structure compacte ainsi qu'un filtrage supérieur aux *nld-nogoods* réduits traités indépendamment. Dans cette section nous allons présenter les *increasing-nogoods* (Lee *et al.* (2016)), leur provenance (Lee et Zhu (2014)), la contrainte globale *IncNG*, ainsi que les algorithmes associés.

2.4.1 La contrainte globale *IncNG*

La contrainte globale *IncNG* (Lee et Zhu (2014)) permet de mettre à disposition une structure compacte d'une collection de *nogoods* dit *increasing* ainsi qu'un algorithme de filtrage dédié dans le but d'avoir un pouvoir de filtrage supérieur à chaque *nogood* traité indépendamment. Nous allons dans un premier temps définir un *IncNG* et ensuite montrer comment en obtenir. Les travaux de Lee et Zhu (2014) enregistrent les *nogoods* à chaque conflit (dans l'optique de casser des symétries durant la recherche) et trois ensembles sont utilisés pour encoder un *IncNG*, ils représentent respectivement les variables concernées (*index*), les valeurs affectées (*equal*) et les valeurs interdites (*not equal*). Afin d'illustrer plus simplement les définitions, nous allons introduire les *nld-nogoods* réduits dirigés qui apparaissent dans des travaux plus récents des mêmes auteurs (Lee *et al.* (2016), présentés Section 2.4.2).

Définition 21 (*nld-nogood* réduit dirigé)

Soit un *nld-nogood* réduit $\Delta = \{\delta_0, \ldots, \delta_m\}$, le ***nld-nogood* réduit dirigé** correspondant s'écrit $\delta_0 \wedge \cdots \wedge \delta_{m-1} \Rightarrow \neg\delta_m$.

Exemple 5

Reprenons l'exemple donné figure 2.1, transformons les *nld-nogoods* réduits extraits précédemment en *nld-nogoods* réduits dirigés :

$$\Delta'_4 = \{\delta_1, \delta_2\} \qquad \delta_1 \Rightarrow \neg\delta_2$$
$$\Delta'_3 = \{\delta_1, \delta_6\} \qquad \delta_1 \Rightarrow \neg\delta_6$$
$$\Delta'_2 = \{\delta_1, \delta_8, \delta_9\} \qquad \delta_1 \wedge \delta_8 \Rightarrow \neg\delta_9$$
$$\Delta'_1 = \{\delta_1, \delta_8, \delta_{11}\} \qquad \delta_1 \wedge \delta_8 \Rightarrow \neg\delta_{11}$$

Lemme 1

L'ensemble des *nld-nogoods* réduits dirigés extraits d'une branche sont croissants (*Increasing*).

Notation 1

$lhs(ng_i)$ décrit la partie gauche (*left hand side*) d'un *nld-nogood* réduit dirigé ng_i.

Définition 22 (Séquence croissante)

Une **séquence** de *nogoods* est dite **croissante** si elle est de la forme :

$$ng_0 \equiv \delta_{s_{00}} \wedge \ \ldots \wedge \delta_{s_{0r_0}} \Rightarrow \neg\delta_{k_0}$$
$$ng_1 \equiv lhs(ng_0) \wedge \delta_{s_{10}} \wedge \ \ldots \wedge \ \delta_{s_{1r_1}} \Rightarrow \neg\delta_{k_1}$$
$$\vdots$$
$$ng_t \equiv lhs(ng_{t-1}) \wedge \delta_{s_{t0}} \wedge \ \ldots \wedge \delta_{s_{tr_t}} \Rightarrow \neg\delta_{k_t}$$

Exemple 6

Nous pouvons voir sur l'exemple 5 que les *nld-nogoods* réduits dirigés obtenus précédemment forment une séquence croissante :

$$\begin{aligned} &\delta_1 \Rightarrow \neg\delta_2 && ng_0 \equiv \delta_1 \Rightarrow \neg\delta_2 \\ &\delta_1 \Rightarrow \neg\delta_6 && ng_1 \equiv lhs(ng_0) \Rightarrow \neg\delta_6 \\ &\delta_1 \wedge \delta_8 \Rightarrow \neg\delta_9 && ng_2 \equiv lhs(ng_1) \wedge \delta_8 \Rightarrow \neg\delta_9 \\ &\delta_1 \wedge \delta_8 \Rightarrow \neg\delta_{11} && ng_3 \equiv lhs(ng_2) \Rightarrow \neg\delta_{11} \end{aligned}$$

À partir de la séquence croissante obtenue, nous pouvons l'encoder pour en faire une contrainte globale *IncNG*. Cette contrainte permet de gérer dynamiquement un ensemble de *nogoods* croissants. Pour cela, considérons qu'une décision δ_i puisse s'expliciter comme une affectation de la forme $x_i = v_i$.

$$\begin{aligned} I = \langle\ & s_{00},\ \ldots, s_{0r_0}, k_0, \\ & s_{10},\ \ldots, s_{1r_1}, k_1, \\ & \vdots \\ & s_{t0},\ \ldots, s_{tr_t}, k_t\ \rangle \end{aligned} \qquad \begin{aligned} E = \langle\ & v_{s_{00}},\ \ldots, v_{s_{0r_0}}, \bot, \\ & v_{s_{10}},\ \ldots, v_{s_{1r_1}}, \bot, \\ & \vdots \\ & v_{s_{t0}},\ \ldots, v_{s_{tr_t}}, \bot\ \rangle \end{aligned} \qquad \begin{aligned} N = \langle\ & \bot,\ \ldots, \bot, v_{k_0}, \\ & \bot,\ \ldots, \bot, v_{k_1}, \\ & \vdots \\ & \bot,\ \ldots, \bot, v_{k_t}\ \rangle \end{aligned}$$

Cet encodage se lit de la manière suivante : prenons le $i^{\text{ème}}$ tuple (I_i, E_i, N_i) dans les trois listes, si $N_i = \bot$ alors le tuple encode $x_{I_i} = E_i$ sinon $E_i = \bot$ et le tuple encode $x_{I_i} \neq N_i$.

Exemple 7

Avant de poursuivre l'exemple récurrent il est nécessaire de l'enrichir. Notons $\delta_1 \equiv (x_1 = 1)$, $\neg\delta_2 \equiv (x_2 \neq 1)$, $\neg\delta_6 \equiv (x_6 \neq 2)$, $\delta_8 \equiv (x_8 = 5)$, $\neg\delta_9 \equiv (x_9 \neq 3)$ et enfin $\neg\delta_{11} \equiv (x_{11} \neq 2)$. Notre séquence croissante s'encode donc :

$$I = \langle 1, 2, 6, 8, 9, 11\rangle, \qquad E = \langle 1, \bot, \bot, 5, \bot, \bot\rangle, \qquad N = \langle \bot, 1, 2, \bot, 3, 2\rangle.$$

Définition 23 (*IncNG*)

Soit $P = (\mathcal{X}, \mathcal{C})$ un CSP, et les trois listes précédemment définies I, E et N de même taille m. La contrainte globale correspondante ***IncNG*** s'écrit de la manière suivante :

$$\begin{aligned} \forall i \in [0, m-1], N_i = \bot \vee (((E_0 = \bot) \vee (x_{I_0} = E_0)) \\ \wedge ((E_1 = \bot) \vee (x_{I_1} = E_1)) \\ \wedge \qquad \vdots \\ \wedge ((E_{i-1} = \bot) \vee (x_{I_{i-1}} = E_{i-1})) \Rightarrow x_{I_i} \neq N_i) \end{aligned}$$

Ranger un *nogood* à chaque conflit dans la contrainte globale *IncNG* correspondante est un traitement lourd pour un élagage faible, l'utilité réside dans l'objectif de casser des symétries (cette notion, n'étant pas utile pour lire la suite du manuscrit, n'est pas présentée ici). Utiliser une contrainte globale pour gérer les *nogoods* semble une méthode intéressante, c'est pourquoi les auteurs ont étendu leur travaux (Lee *et al.* (2016)) en enregistrant des *nogoods* aux redémarrages, une nouvelle compression plus efficace y est aussi présentée. Dans la suite nous détaillons cette méthode.

2.4.2 *IncNG* adapté aux redémarrages

L'encodage décrit dans Lee *et al.* (2016) est fait sous la forme d'une séquence de décisions. Nous allons tout d'abord expliquer comment passer d'une séquence croissante de *nld-nogoods* réduits dirigés à une séquence de décisions :

$$
\begin{aligned}
ng_0 &\equiv \delta_{s_{00}} \wedge \ldots \wedge \delta_{s_{0r_0}} \Rightarrow \neg\delta_{k_0} \\
ng_1 &\equiv \ lhs(ng_0) \ \wedge \delta_{s_{10}} \wedge \ldots \wedge \delta_{s_{1r_1}} \Rightarrow \neg\delta_{k_1} \\
&\vdots \\
ng_t &\equiv \ lhs(ng_{t-1}) \ \wedge \delta_{s_{t0}} \wedge \ldots \wedge \delta_{s_{tr_t}} \Rightarrow \neg\delta_{k_t}
\end{aligned}
\qquad
\begin{aligned}
\Sigma = \langle\ & \delta_{s_{00}},\ \ldots, \delta_{s_{0r_0}}, \neg\delta_{k_0}, \\
& \delta_{s_{10}},\ \ldots, \delta_{s_{1r_1}}, \neg\delta_{k_1}, \\
& \vdots \\
& \delta_{s_{t0}},\ \ldots, \delta_{s_{tr_t}}, \neg\delta_{k_t}\ \rangle
\end{aligned}
$$

Exemple 8

Appliquons cela sur notre exemple :

$$
\begin{aligned}
ng_0 &\equiv \delta_1 \Rightarrow \neg\delta_2 \\
ng_1 &\equiv lhs(ng_0) \Rightarrow \neg\delta_6 \\
ng_2 &\equiv lhs(ng_1) \wedge \delta_8 \Rightarrow \neg\delta_9 \\
ng_3 &\equiv lhs(ng_2) \Rightarrow \neg\delta_{11}
\end{aligned}
\qquad
\begin{aligned}
\Sigma = \langle\ & \delta_1, \neg\delta_2, \\
& \neg\delta_6, \\
& \delta_8, \neg\delta_9, \\
& \neg\delta_{11}\ \rangle
\end{aligned}
$$

Remarque 8

$\Sigma = \langle \delta_1, \neg\delta_2, \neg\delta_6, \delta_8, \neg\delta_9, \neg\delta_{11} \rangle$ est la séquence de décisions initiale (la dernière branche de l'arbre de décisions).

Nous pouvons voir que, grâce à la méthode d'encodage de Lee *et al.* (2016), nous n'avons besoin d'aucun calcul supplémentaire pour créer une contrainte *IncNG*. Il suffit, lors de la recherche, de conserver une partie de l'arbre afin obtenir la dernière branche lors du redémarrage. Dans la suite nous détaillons la méthode de gestion des *nogoods* ainsi que l'algorithme de filtrage de la contrainte globale *IncNG*. Lee *et al.* (2016) ont proposé un algorithme (voir algorithme 2.3) de filtrage léger grâce à l'utilisation des *watched litterals* (complexité dans le pire cas $O(\sum_{x \in X} |\texttt{dom}(x)|)$, c'est-à-dire la taille maximum de Σ) que nous présentons avant de détailler

les cas d'appel à cet algorithme. Tout d'abord commençons par expliquer pourquoi le filtrage peut avoir lieu sur un *IncNG*.

Propriété 1

Soit $\Lambda = \langle ng_0, \ldots, ng_t \rangle$ une séquence composée de *nld-nogoods* croissants. Si $lhs(ng_i)$ contient deux décisions positives pouvant encore être falsifiées alors les *nogoods* ng_j tel que $j \geq i$ sont nécessairement arc-cohérents (ou GAC pour *Generalized Arc Consistency*) car les parties gauches des *nogoods* plus grands subsument celles des plus petits.

Propriété 2

Soit $\Lambda = \langle ng_0, \ldots, ng_t \rangle$ une séquence composée d'*increasing-nogoods*. GAC sur IncNG(Λ) est équivalent à GAC sur chaque ng_i individuellement.

Lors de la création de la contrainte globale *IncNG*, les *nogoods* dont la partie gauche est vide sont directement filtrés. Ensuite, deux indices α et β sont utilisés. Ces indices correspondent aux deux décisions positives non affectées les plus à gauche dans la séquence (c'est-à-dire pouvant encore être falsifiées). Ces deux décisions positives ainsi que toutes les parties droites (décisions négatives) se situant entre α et β sont surveillées par un système de sentinelles (*watch*). Concernant les notations, la décision pointée par alpha (resp. bêta) sera notée δ_α (resp. δ_β), de même l'alpha (resp. bêta) d'un *increasing-nogood* Σ peut être consulté avec $\alpha(\Sigma)$ (resp. $\beta(\Sigma)$).

Exemple 9

Soit l'exemple étoffé précédemment (Exemple 7).

$$\begin{aligned} &\delta_1 \Rightarrow \neg\delta_2 && ng_0 \equiv (x_1 = 1) \Rightarrow (x_2 \neq 1) \\ &\delta_1 \Rightarrow \neg\delta_6 && ng_1 \equiv lhs(ng_0) \Rightarrow (x_6 \neq 2) \\ &\delta_1 \wedge \delta_8 \Rightarrow \neg\delta_9 && ng_2 \equiv lhs(ng_1) \wedge (x_8 = 5) \Rightarrow (x_9 \neq 3) \\ &\delta_1 \wedge \delta_8 \Rightarrow \neg\delta_{11} && ng_3 \equiv lhs(ng_2) \Rightarrow (x_{11} \neq 2) \end{aligned}$$

La séquence correspondante est :

$$\Sigma = \langle \overbrace{\underbrace{x_1 = 1}_{\alpha}, x_2 \neq 1, x_6 \neq 2, \underbrace{x_8 = 5}_{\beta}}^{\text{Watched}}, x_9 \neq 3, x_{11} \neq 2 \rangle$$

Il existe trois situations principales d'appel à l'algorithme de filtrage (algorithme 2.3). Considérons la séquence suivante afin de les illustrer :

$$\Sigma = \langle \overbrace{\underbrace{x_2 = 1}_{\alpha}, x_3 \neq 1, \underbrace{x_4 = 1}_{\beta}}^{\text{Watched}}, x_1 \neq 1, x_5 = 1, x_6 \neq 2 \rangle$$

Cas 1 Une décision négative contenue entre α et β est falsifiée, cela force δ_α à être faux, par conséquent tous les *nogoods* contenus dans la contrainte sont falsifiés.

Exemple 10

$$
\begin{aligned}
ng_0 &\equiv & x_2 = 1 &\Rightarrow x_3 \neq 1\\
ng_1 &\equiv & x_2 = 1 \wedge x_4 = 1 &\Rightarrow x_1 \neq 1\\
ng_2 &\equiv & x_2 = 1 \wedge x_4 = 1 \wedge x_5 = 1 &\Rightarrow x_6 \neq 2
\end{aligned}
$$

$$
\begin{aligned}
ng_0 &\equiv & x_2 = 1 &\Rightarrow \bot\\
ng_1 &\equiv & x_2 = 1 \wedge x_4 = 1 &\Rightarrow x_1 \neq 1\\
ng_2 &\equiv & x_2 = 1 \wedge x_4 = 1 \wedge x_5 = 1 &\Rightarrow x_6 \neq 2
\end{aligned}
$$

$$
\begin{aligned}
ng_0 &\equiv & \bot &\Rightarrow \bot\\
ng_1 &\equiv & \bot \wedge x_4 = 1 &\Rightarrow x_1 \neq 1\\
ng_2 &\equiv & \bot \wedge x_4 = 1 \wedge x_5 = 1 &\Rightarrow x_6 \neq 2
\end{aligned}
$$

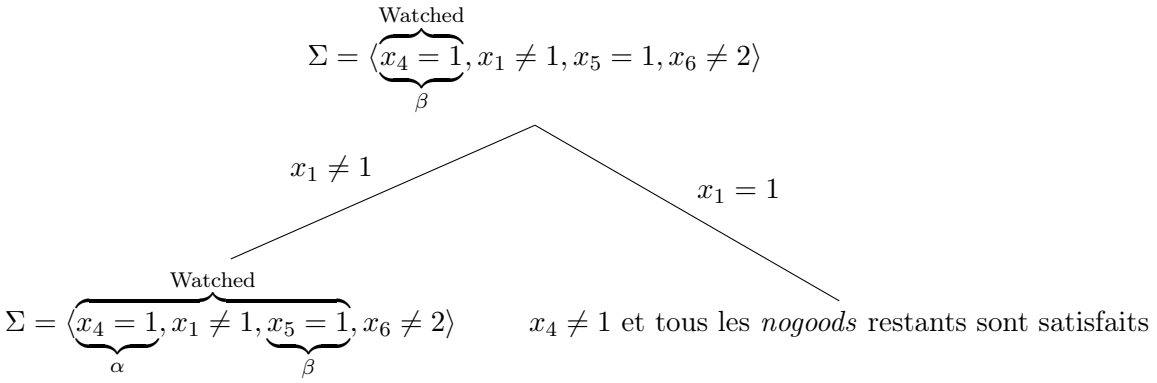
Cas 2 La décision positive désignée par α est satisfaite : nous forçons toutes les parties droites qui ne contiennent que δ_α dans leur partie gauche à être vraies, c'est-à-dire toutes les décisions négatives contenues entre α et β et nous recherchons la prochaine décision positive non affectée. Lors de ce parcours, nous vérifions si les décisions négatives sont falsifiées, si c'est le cas le β précédent doit être falsifié jusqu'à avoir traité Σ au complet ou à trouver de nouveaux indices.

Exemple 11

$$
\begin{aligned}
ng_0 &\equiv & x_2 = 1 &\Rightarrow x_3 \neq 1\\
ng_1 &\equiv & x_2 = 1 \wedge x_4 = 1 &\Rightarrow x_1 \neq 1\\
ng_2 &\equiv & x_2 = 1 \wedge x_4 = 1 \wedge x_5 = 1 &\Rightarrow x_6 \neq 2
\end{aligned}
$$

$$
\begin{aligned}
ng_0 &\equiv & \top &\Rightarrow x_3 \neq 1\\
ng_1 &\equiv & \top \wedge x_4 = 1 &\Rightarrow x_1 \neq 1\\
ng_2 &\equiv & \top \wedge x_4 = 1 \wedge x_5 = 1 &\Rightarrow x_6 \neq 2
\end{aligned}
$$

$$
\begin{aligned}
ng_0 &\equiv & \top &\Rightarrow \top\\
ng_1 &\equiv & \top \wedge x_4 = 1 &\Rightarrow x_1 \neq 1\\
ng_2 &\equiv & \top \wedge x_4 = 1 \wedge x_5 = 1 &\Rightarrow x_6 \neq 2
\end{aligned}
$$

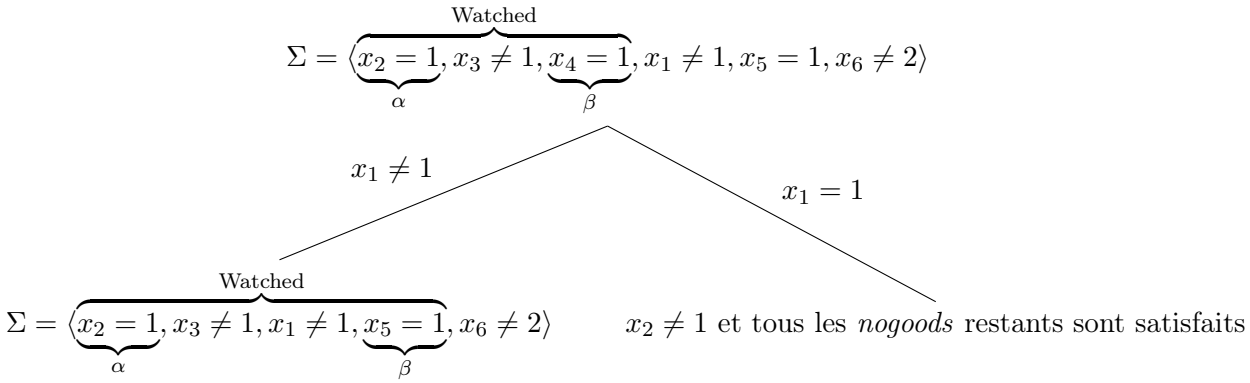


Cas 3 La décision positive désignée par β est satisfaite : ceci est semblable au cas précédent, nous recherchons la prochaine décision positive non affectée.

Exemple 12

$$
\begin{aligned}
ng_0 &\equiv & x_2 = 1 &\Rightarrow x_3 \neq 1 \\
ng_1 &\equiv & x_2 = 1 \wedge x_4 = 1 &\Rightarrow x_1 \neq 1 \\
ng_2 &\equiv & x_2 = 1 \wedge x_4 = 1 \wedge x_5 = 1 &\Rightarrow x_6 \neq 2
\end{aligned}
$$

$$
\begin{aligned}
ng_0 &\equiv & x_2 = 1 &\Rightarrow x_3 \neq 1 \\
ng_1 &\equiv & x_2 = 1 \wedge \top &\Rightarrow x_1 \neq 1 \\
ng_2 &\equiv & x_2 = 1 \wedge \top \wedge x_5 = 1 &\Rightarrow x_6 \neq 2
\end{aligned}
$$



2.4.3 Algorithme de filtrage de la contrainte globale *IncNG*

Maintenant que nous avons vu les différents cas d'appel, nous allons étudier l'algorithme de filtrage en lui-même.

Algorithme 2.3 : FilterIncNG(Σ)

Data : $m = |\Sigma|$ where Σ is the sequence of literals labelling the current branch

```
1  UpdateAlpha();
2  if m ≠ 0 ∧ β ≠ m then
3  |  UpdateBeta();
4  if m = 0 then
5  |  delete constraint;
```

L'algorithme 2.3 est appelé une première fois lors de l'ajout de l'*IncNG* à la base de connaissances puis lorsqu'un des trois cas décrits précédemment est identifié.

Algorithme 2.4 : UpdateAlpha()

```
1  if δ_α is satisfied then
2  |  unsubscribe δ_α;
3  |  for i ∈ [α + 1, β) do
4  |  |  if Neg(δ_i) then
5  |  |  |  satisfy δ_i;
6  |  |  |  unsubscribe δ_i;
7  |  if β = m then
8  |  |  m ← 0 ; return ;
9  |  α ← β;
10 |  watchFollowDec();
11 |  if m ≠ 0 then
12 |  |  UpdateAlpha();
13 else
14 |  for i ∈ [α + 1, β) do
15 |  |  if Neg(δ_i) ∧ δ_i is falsified then
16 |  |  |  falsify δ_α;
17 |  |  |  m ← 0;
18 |  |  |  return ;
```

Remarque 9

Neg(δ_i) retourne vrai si δ_i est une décision négative.

Les algorithmes 2.4 et 2.5 qui ont été proposés par Lee *et al.* (2016) ont un fonctionnement relativement similaire. Si δ_α est satisfait, un nouvel α est trouvé et l'algorithme est appelé de

Algorithme 2.5 : UpdateBeta()

```
1 if δ_β is satisfied then
2     unsubscribe δ_β;
3     watchFollowDec();
4     if m ≠ 0 then
5         UpdateBeta();
```

nouveau pour voir si δ_α est satisfaite jusqu'à arriver à un point fixe ou que la contrainte soit entièrement traitée (de la même manière pour β). La fonction `watchFollowDec` permet de trouver la prochaine décision positive à partir de β si elle existe et de surveiller toutes les décisions négatives rencontrées. Si une décision négative falsifiée est rencontrée, m, qui représente la taille de l'*IncNG* traité, est mis à zéro, ce qui correspond à la désactivation de cet *IncNG*.

2.4.4 Minimisation des *nogoods* dans la contrainte globale *IncNG*

Dans le but d'optimiser l'efficacité des *nogoods*, Lee *et al.* (2016) ont proposé une méthode de minimisation qui conserve la propriété de croissance. Cette méthode utilise les décisions de transition de Lecoutre *et al.* (2007b) (les décisions dont nous savons à coup sûr qu'elles appartiennent à l'explication du conflit). Il est important, afin de conserver la propriété de croissance, de modifier la méthode originale car après application, $lhs(ng_i) \subseteq lhs(ng_{i+1})$ n'est pas forcément vraie, pour cela la minimisation est approximée (nous obtenons des *nogoods* minimisés mais pas minimaux).

Les *nogoods* contenus dans un *IncNG* sont minimisés du plus petit au plus grand en vérifiant à chaque fois que la propriété de croissance est conservée. Pour cela les décisions de transition d'un *nogood* $k+1$ sont cherchées uniquement dans l'ensemble $(lhs(ng_{k+1}) - lhs(ng'_k) \cup \neg rhs(ng_{k+1})$ où ng'_k est le *nogood* ng_k minimisé.

Notation 2

$rhs(ng_i)$ décrit la partie droite (*right hand side*) d'un *nld-nogood* réduit dirigé ng_i.

Propriété 3

Les *nogoods* minimisés produits par l'algorithme 2.6 sont croissants.

Algorithme 2.6 : Minimization(P,Σ,ϕ)

Data : $\Delta = \emptyset$: the set of transition decisions
Data : $\alpha = -1$: the position of the current last negative decision
$P' \leftarrow P$;
for $i \in [0, |\Sigma| - 1]$ **do**
  if $Neg(\delta_i)$ **then**
    $\Delta = Constructive(P', \phi, \{\delta_{\alpha+1}, \ldots, \neg\delta_i\})$;
    reorder Σ from $\alpha + 1$ to i to :
    put $\Delta - \{\neg\delta_i\}$ into the first place,
    δ_i into the second place, and
    the remaining into the third place;
    $\alpha \leftarrow \alpha + |\Delta|$;
    $P' \leftarrow P' \cup (\Delta - \{\neg\delta_i\})$;

L'algorithme 2.6 décrit la procédure utilisée par Lee *et al.* (2016) pour obtenir des *nogoods* minimisés. Il a une complexité en $O(\sum_{x \in X} |\texttt{dom}(x)|\xi)$ où ξ est le coût d'une application unique de l'opération d'inférence ϕ sur P. La fonction `Constructive` retourne un ensemble de décisions de transition Δ qui appartiennent au *nogood* que nous voulons minimiser lors de son appel.

2.5 Conclusion

Dans ce chapitre, après l'introduction de l'apprentissage en programmation par contraintes et des *nogoods* standards, nous avons présenté dans le détail les *nld-nogoods* ainsi que les *increasing-nogoods*. Nous avons décrit comment, à partir de la dernière branche de l'arbre de recherche, nous pouvons extraire des informations permettant d'éviter le phénomène de *thrashing*. Puis nous avons étudié la représentation de ces informations sous forme de séquences et la manière de les minimiser, à la fois pour les *nld-nogoods* et les *increasing-nogoods*. Ces derniers sont la base de notre première contribution qui est détaillée au sein du chapitre 5. Nous verrons comment combiner ces *increasing-nogoods* afin d'anticiper les conflits potentiels afin de rendre l'algorithme de recherche plus robuste.

Dans le chapitre suivant, nous présentons le problème de satisfaction booléenne (SAT). Ce problème, très lié à la programmation par contrainte, est un cas particulier du problème de satisfaction de contrainte. Cela a pour effet d'avoir des algorithmes très efficaces dédiés à ce problème. Notamment, les algorithmes de type CDCL qui génèrent des clauses, c'est-à-dire une forme évoluée inspirée des *nogoods* standards.

Chapitre 3

Le problème SAT

Sommaire

Le problème de satisfaction booléenne (SAT) est le premier problème prouvé `NP-complet` (Cook (1971)) et probablement l'un des plus connus. Il est très utile en pratique car de nombreux problèmes possèdent une réduction vers celui-ci. Par exemple, le problème de satisfaction de contraintes peut être traduit vers SAT *via* de nombreux encodages ; les plus connus étant l'encodage direct (de Kleer (1989)), logarithmique (Iwama et Miyazaki (1994)) ou encore l'encodage d'ordre (Crawford et Baker (1994)). Le problème SAT peut être défini comme un cas particulier du problème de satisfaction de contraintes qui utilise des variables avec des domaines binaires et des clauses. Ces dernières peuvent être vues comme des contraintes de sommes positives ($sum(x_i) > 0$). Les solveurs SAT modernes sont capables de gérer des millions de variables et de clauses. Afin de présenter les mécanismes utilisés par ceux-ci, nous commençons par introduire formellement la logique propositionnelle ; puis la procédure de résolution DPLL (*Davis–Putnam–Logemann–Loveland*, Davis *et al.* (1962)), une extension de l'algorithme DP (Davis et Putnam (1960)) non présenté ici ; et enfin, la base des solveurs SAT modernes, la procédure CDCL (*Conflict-Driven Clause Learning*, Marques-Silva et Sakallah (1996)).

3.1 Logique propositionnelle

La logique propositionnelle classique permet de raisonner avec ce qui est vrai (*true*, $\top$, 1) et ce qui est faux ($false$, $\bot$, 0). La syntaxe de la logique propositionnelle peut-être formellement définie comme suit :

Définition 24 (Langage de la logique propositionnelle)

Soit $\mathbb{P}$ un ensemble infini dénombrable de variables propositionnelles. Le **langage de la logique propositionnelle** est l'ensemble des formules contenant $\mathbb{P}$, fermé sous l'ensemble des connecteurs logiques $\{\neg, \wedge\}$.

Sans perte de généralité, nous supposons que toutes les formules de la logique propositionnelle sont en forme normale conjonctive (CNF), c'est-à-dire une conjonction finie de clauses, car n'importe quelle formule peut être transformée en une CNF équisatisfiable (équivalente du point de vue de la satisfiabilité, voir définition 31) en utilisant l'algorithme de Tseitin (1968).

Définition 25 (Littéral)

Un **littéral** est une variable propositionnelle associée à un signe. Soit l une variable propositionnelle ; alors l est appelé littéral positif, tandis que $\neg l$ est appelé littéral négatif. Nous dirons que l (resp. $\neg l$) est le littéral complémentaire de $\neg l$ (resp. l).

Définition 26 (Clause)

Une **clause** est définie comme une disjonction finie de littéraux distincts.

Sur les aspects sémantiques de la logique propositionnelle, la notion d'interprétation est importante. Elle est définie comme suit :

Définition 27 (Interprétation)

Une **interprétation** est un ensemble de valuations des variables propositionnelles. Formellement, c'est une application $\mathbb{P} \to \{true, false\}$.

Définition 28 (Clause satisfaite/falsifiée)

Une clause σ est **satisfaite** par une interprétation $\mathcal{I}$ si au moins un littéral de σ est affecté à $vrai$. Au contraire, une clause est **falsifiée** si tous les littéraux la composant sont affectés à $faux$.

Définition 29 (Modèle)

Nous dirons qu'une interprétation est un **modèle** d'une formule Σ si Σ est vraie pour cette interprétation, c'est-à-dire que l'interprétation satisfait les clauses de Σ.

L'opérateur $\models$ exprime l'implication et $\alpha \models \beta$ exprime que β est une conséquence logique de α et par conséquent β est satisfait par tous les modèles de α. Si une formule a au moins un modèle $\mathcal{M}$, nous dirons que cette formule est *satisfiable* ; $\mathcal{M} \models \Sigma$ indiquera que $\mathcal{M}$ satisfait Σ. Formellement, la relation de satisfiabilité est définie comme suit :

Définition 30 (Relation de satisfiabilité dans la logique propositionnelle)

La relation $\models$ entre les interprétations $\mathcal{M}$ et les formules Σ dans la logique propositionnelle est définie récursivement comme suit :

$\mathcal{M} \models p$	ssi	$p \in \mathcal{M}$
$\mathcal{M} \models \neg\Sigma$	ssi	$\mathcal{M} \not\models \Sigma$
$\mathcal{M} \models \Sigma_1 \wedge \Sigma_2$	ssi	$\mathcal{M} \models \Sigma_1$ et $\mathcal{M} \models \Sigma_2$
$\mathcal{M} \models \Sigma_1 \vee \Sigma_2$	ssi	$\mathcal{M} \models \Sigma_1$ ou $\mathcal{M} \models \Sigma_2$

Si une formule est satisfiable par toutes les interprétations, nous dirons que cette formule est *valide* ; dans ce cas, la formule est une *tautologie*. Si une formule est fausse pour toutes les interprétations, nous dirons que cette formule est *insatisfiable*.

Définition 31 (Formules équisatisfiables)

Deux formules Σ_1 et Σ_2 sont **équisatisfiables** si et seulement si Σ_1 et Σ_2 sont toutes deux satisfiables ou toutes deux insatisfiables.

Définition 32 (Formules équivalentes)

Deux formules Σ_1 et Σ_2 sont **équivalentes**, noté $\Sigma_1 \equiv \Sigma_2$, si et seulement si l'ensemble des modèles de Σ_1 est égal à l'ensemble des modèles de Σ_2.

Définition 33 (Le problème de satisfaction booléenne)

Le **problème de satisfaction booléenne** (SAT) est le problème de décision qui consiste à déterminer si une formule CNF possède un modèle.

Dans les sections suivantes, nous allons présenter quelques techniques algorithmiques de résolution du problème de satisfaction booléenne, basée majoritairement sur le parcours en profondeur d'un arbre de recherche.

3.2 Principe de résolution

L'un des algorithmes les plus simples afin de décider la satisfiabilité d'une formule est le principe de résolution (Robinson (1965)).

Définition 34 (Résolution)

Soient $\sigma_1 = x \vee \alpha$ et $\sigma_2 = \neg x \vee \beta$ deux clauses ayant en commun une variable propositionnelle x. Cette variable propositionnelle est présente dans les deux clauses sous la forme de littéraux complémentaires. Appliquer la **résolution** à ces deux clauses (notée $\eta[x, \sigma_1, \sigma_2]$), en supprimant toutes les occurrences du littéral x et $\neg x$ dans respectivement α et β, permet d'obtenir la clause $\sigma = \alpha \vee \beta$. σ est appelée clause résolvante.

L'algorithme consiste à appliquer la résolution jusqu'à ce que le problème soit insatisfiable (clause vide en résolvante) ou qu'aucune résolution ne soit possible, dans ce cas le problème est considéré satisfiable. Cet algorithme est possible car la résolution est complète pour la réfutation, c'est-à-dire qu'elle est garantie de retourner la clause vide si la CNF est insatisfiable.

3.3 DPLL

DPLL (ou DLL Davis *et al.* (1962)) est un algorithme complet de recherche arborescente en profondeur d'abord à retour arrière qui est associé à une heuristique de choix de littéraux et à la propagation unitaire (ou résolution unitaire). Cette dernière permet de considérer lorsqu'une clause devient unitaire, c'est-à-dire qu'il ne reste qu'un unique littéral non affecté au sein de celle-ci, que le littéral restant doit obligatoirement être satisfait. En effet, satisfaire le littéral opposé aurait pour effet d'obtenir une clause vide et par conséquent de rendre la formule insatisfiable. De ce fait, quand un tel cas arrive, le littéral est propagé et la formule est simplifiée. Il est alors dit que la formule est *conditionnée* par le littéral. Cela revient à satisfaire toutes les clauses contenant le littéral propagé et à supprimer la négation du littéral (son complémentaire) dans toutes celles où celle-ci apparaît. La formule ainsi conditionnée contient donc moins de clauses et les clauses restantes sont de plus petite taille si elles contenaient le complémentaire. La propagation unitaire revient donc à conditionner la formule de manière répétée tant que des clauses unitaires la compose et qu'une clause vide n'est pas générée.

En pratique, une grande partie des formules sont réduites par des enchaînements de propagations unitaires. Ce processus est représenté par la fonction `UnitPropagation` dans l'algorithme `DPLL` (Algorithme 3.1). Cette fonction retourne deux éléments : I, l'ensemble des littéraux qui ont été satisfaits par des clauses unitaires ou déduits par la propagation, et Γ la nouvelle formule correspondant à l'application de l'ensemble de littéraux I sur la formule initiale Σ passée en paramètre.

Le choix de la variable à propager (variable dite de décision) se fait heuristiquement. À l'instar des algorithmes de programmation par contraintes, plusieurs heuristiques ont été proposées. Elles sont statiques ou dynamiques. Les heuristiques statiques décident de l'ordre au début de la recherche tandis que les heuristiques dynamiques font évoluer l'ordre de sélection au cours de la

recherche. Ces dernières voient leur intérêt renforcé lorsque combinées à des redémarrages. Nous nous intéressons de plus près aux heuristiques dans la section 3.4.

Après le choix de la variable, la règle de séparation est appliquée : nous allons dans un premier temps considérer la variable propositionnelle de manière positive, puis ensuite de manière négative.

L'insatisfiabilité d'une interprétation est détectée lorsqu'une clause devient vide, c'est-à-dire que tous les littéraux la composant sont faux. La satisfiabilité de la formule est détectée lorsque toutes les clauses sont satisfaites.

Algorithme 3.1 : DPLL(Σ)

```
Data : Σ a CNF
Result : Returns a set of literals ; UNSAT otherwise
(I, Γ) = UnitPropagation(Σ);
if Γ = {} then
    return I;
else if ⊥ ∈ Γ then
    return UNSAT;
else
    choose a literal l in Γ;
    if L = DPLL(Γ|_l) ≠ UNSAT then
        return L ∪ I ∪ {l};
    else if L = DPLL(Γ|_¬l) ≠ UNSAT then
        return L ∪ I ∪ {¬l};
    else
        return UNSAT;
```

Nous pouvons noter qu'à la manière de MAC (Algorithme 1.7, section 1.2.1), l'algorithme 3.1 est une recherche complète avec retour arrière chronologique. Lorsqu'un conflit est détecté, l'algorithme revient au nœud précédent, qui était encore cohérent. En effet, si les deux valeurs de vérité sont testées pour une variable propositionnelle donnée au niveau n, et que chacune a mené à une contradiction, il faut remonter au niveau précédent, $n-1$, en défaisant les affectations qui ont pu être faites entre les deux niveaux. Ensuite, il faut tester une valeur à ce niveau s'il en reste une ; dans le cas contraire, nous remontons au niveau $n-2$ et ainsi de suite. Lorsque que le niveau 0 est atteint, si un retour arrière est requis car toutes les valeurs ont conduit à une contradiction, nous pouvons déduire que la formule est incohérente.

3.4 CDCL

Cette section présente l'algorithme CDCL (*Conflict-Driven Clause Learning*). Cet algorithme est une extension de l'algorithme DPLL, introduit par Marques-Silva et Sakallah (1996; 1999) et amélioré par Moskewicz *et al.* (2001). Le nom CDCL désigne, à l'origine, la procédure qui permet d'apprendre de nouvelles clauses à chaque conflit lors de la recherche. Le nom est maintenant plus généralement utilisé pour désigner tous les solveurs SAT modernes utilisant cette procédure. Cette méthode, associée aux redémarrages et aux heuristiques adaptatives présentés au sein de cette section permet au solveur de s'échapper des parties de l'arbre de recherche qui n'ont aucune solution et qui peuvent être très difficiles à prouver insatisfiables. Les solveurs CDCL

sont aujourd'hui les solveurs SAT les plus performants. Par la suite, nous décrivons chaque brique indépendamment, à savoir, l'analyse de conflits, les structures de données paresseuses, les heuristiques, la gestion de la base de clauses ainsi que les politiques de redémarrage.

3.4.1 Analyse de conflits

Dans cette section, nous présentons la brique principale des solveurs CDCL, à savoir, l'analyse de conflits. Elle permet de générer les clauses qui vont permettent d'éviter, de façon analogue aux *nogoods* en programmation par contraintes (Chapitre 2), de reproduire les mêmes conflits. Pour présenter cette notion, nous devons introduire plusieurs notations. Le niveau de décision d'un littéral l, noté $niv(l)$ est le niveau auquel il a été décidé. Le niveau d'un littéral propagé est égal au niveau de décision du littéral décidé avant la phase de propagation. Les littéraux propagés avant la première décision ont le niveau 0.

L'algorithme CDCL, à la manière de DPLL, réalise des enchainements de décisions puis de propagations. Une décision x_i, qui a lieu au niveau l, est notée $[x_i@l]$ (voire $[x_i^l]$) au sein de la séquence de décisions-propagations. Les propagations x_j induites par cette décision sont notées, $x_j@l$ (voire x_j^l). L'enchainement de décisions-propagations est appelé *trace* (*trail*).

Exemple 13

Soit la base de clauses suivante:

$$\sigma_1 = (a \vee b \vee \neg c) \qquad \sigma_4 = (\neg e \vee \neg f)$$
$$\sigma_2 = (a \vee \neg d) \qquad \sigma_5 = (\neg e \vee \neg g \vee h)$$
$$\sigma_3 = (c \vee d \vee e) \qquad \sigma_6 = (f \vee g)$$

Appliquons la séquence de décisions $\neg h$ puis $\neg b$ et enfin $\neg a$.

Décision : $\neg h$

Trace : $[\neg h@0]$

$$\sigma_1 = (a \vee b \vee \neg c) \qquad \sigma_4 = (\neg e \vee \neg f)$$
$$\sigma_2 = (a \vee \neg d) \qquad \sigma_5 = (\neg e \vee \neg g \vee h)$$
$$\sigma_3 = (c \vee d \vee e) \qquad \sigma_6 = (f \vee g)$$

Décision : $\neg b$

Trace : $[\neg h@0], [\neg b@1]$

$$\sigma_1 = (a \vee b \vee \neg c) \qquad \sigma_4 = (\neg e \vee \neg f)$$
$$\sigma_2 = (a \vee \neg d) \qquad \sigma_5 = (\neg e \vee \neg g)$$
$$\sigma_3 = (c \vee d \vee e) \qquad \sigma_6 = (f \vee g)$$

Décision : $\neg a$

Trace : $[\neg h@0], [\neg b@1], [\neg a@2]$

$$\sigma_1 = (a \vee \neg c) \qquad \sigma_4 = (\neg e \vee \neg f)$$
$$\sigma_2 = (a \vee \neg d) \qquad \sigma_5 = (\neg e \vee \neg g)$$
$$\sigma_3 = (c \vee d \vee e) \qquad \sigma_6 = (f \vee g)$$

Propagation : $\neg c, \neg d$

Trace : $[\neg h@0], [\neg b@1], [\neg a@2], \neg c, \neg d$

$$\sigma_1 = (\neg c) \qquad \sigma_4 = (\neg e \vee \neg f)$$
$$\sigma_2 = (\neg d) \qquad \sigma_5 = (\neg e \vee \neg g)$$
$$\sigma_3 = (c \vee d \vee e) \qquad \sigma_6 = (f \vee g)$$

Propagation : e

Trace : $[\neg h@0], [\neg b@1], [\neg a@2], \neg c, \neg d, e$

$$\sigma_1 = (\neg c) \qquad \sigma_4 = (\neg e \vee \neg f)$$
$$\sigma_2 = (\neg d) \qquad \sigma_5 = (\neg e \vee \neg g)$$
$$\sigma_3 = (e) \qquad \sigma_6 = (f \vee g)$$

Propagation : $\neg f, \neg g$

Trace : $[\neg h@0], [\neg b@1], [\neg a@2], \neg c, \neg d, e, \neg f, \neg g$

$$\sigma_1 = (\neg c) \qquad \sigma_4 = (\neg f)$$
$$\sigma_2 = (\neg d) \qquad \sigma_5 = (\neg g)$$
$$\sigma_3 = (e) \qquad \sigma_6 = (f \vee g) = () = \bot$$

CONFLIT !

Si l'exemple 13 est appliqué à l'algorithme DPLL, alors un retour arrière (au niveau 1) aurait lieu et $[a@2]$ serait décidé et la recherche pourrait continuer. Dans le cas de l'algorithme CDCL, une analyse de conflits va avoir lieu afin de générer une clause qui va empêcher le conflit de se reproduire par la suite. Naïvement, il est tout à fait possible d'ajouter la négation de la séquence de décision $\neg(\neg h \wedge \neg b \wedge \neg a)$ qui correspond à la clause $(h \vee b \vee a)$. Cela permet d'éviter de reproduire la même séquence. Une technique plus évoluée, l'analyse de conflits, permet d'obtenir des clauses de meilleure qualité : les clauses obtenues sont plus fines et représentent mieux le conflit. En effet, nous pouvons nous baser sur le graphe, appelé graphe d'implication, représentant d'état du solveur lors de la séquence de décisions-propagations ainsi que la clause responsable de chaque propagation.

Définition 35 (Graphe d'implication)

Soit une séquence de décisions-propagations T. Le **graphe d'implication** de T est un graphe orienté acyclique (DAG) où les nœuds sont les littéraux de T et chaque arc représente une propagation unitaire. Un arc (a, b) est étiqueté par σ_i, la clause qui a déclenché la propagation. Cela signifie que l'affectation (représentée par un carré) ou la propagation (représentée par un cercle) d'un littéral a a participé à la propagation de b dans la clause σ_i. Si un conflit survient, deux littéraux contradictoires propagés ou une clause vide, nous ajoutons le nœud spécial $\bot$.

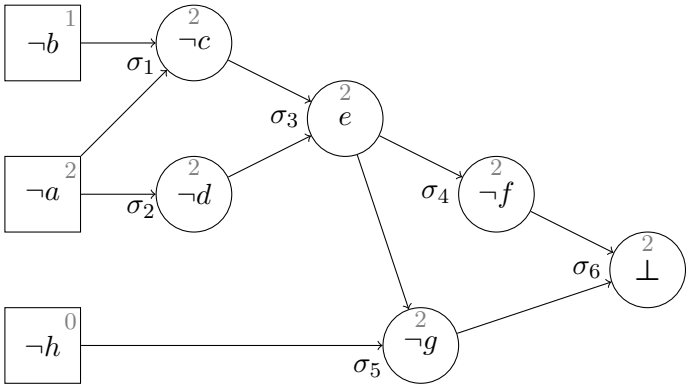


FIGURE 3.1 – Graphe d'implication associé à l'exemple 13

Définition 36 (Nœud dominant)

Soit un graphe d'implication G, un nœud $a \in noeuds(G)$ **domine** un nœud $b \in noeuds(G)$ si et seulement si $niv(a) = niv(b)$ et $\forall c \in noeuds(G)$, avec $niv(a) = niv(c)$, tous les chemins de b vers c passent par a.

Définition 37 (Point d'implication unique (UIP))

Soit un graphe d'implication G, un nœud $a \in noeuds(G)$ est un **point d'implication unique** si et seulement si a domine le nœud conflit (nœud étiqueté $\bot$).

Effectuer l'analyse de conflits revient à chercher un nœud dominant du dernier niveau de décision (ici 2). Un tel nœud dominant le conflit est appelée point d'implication unique, UIP (*Unique Implication Point*).

Les littéraux UIP sont multiples au sein d'un graphe d'implication. Ils sont ordonnés en fonction de leur distance avec le conflit et numérotés de 1 à p, où le premier UIP (1-UIP/F-IUP) est celui le plus proche du conflit, et le dernier (p-UIP/L-UIP) est la dernière décision prise avant le conflit. En pratique, nous ne listons pas tous les UIP, seul le premier est calculé (Zhang *et al.* (2001)). Afin de calculer ces UIP, nous appliquons la règle de résolution entre les clauses qui sont responsables du conflit. Il suffit d'appliquer cette règle en remontant vers la variable de décision en suivant la trace, jusqu'à obtenir une clause résolvante qui contient un seul littéral du dernier niveau de décision : le premier UIP. Une telle clause est appelée clause assertive.

Exemple 14

Soit la base de clauses de l'exemple 13 :

$$\sigma_1 = (a \vee b \vee \neg c) \qquad \sigma_4 = (\neg e \vee \neg f)$$
$$\sigma_2 = (a \vee \neg d) \qquad \sigma_5 = (\neg e \vee \neg g \vee h)$$
$$\sigma_3 = (c \vee d \vee e) \qquad \sigma_6 = (f \vee g)$$

Trace : $[\neg h@0], [\neg b@1], [\neg a@2], \neg c, \neg d, e, \neg f, \boldsymbol{\neg g}$

$$\begin{aligned}\alpha_1 = \eta[g, \sigma_5, \sigma_6] &= \eta[g, \neg e@2 \vee \neg g@2 \vee h@0, f@2 \vee g@2] \\ &= \neg e@2 \vee \cancel{\neg g@2} \vee h@0 \vee f@2 \vee \cancel{g@2} \\ &= \neg e@2 \vee h@0 \vee f@2\end{aligned}$$

Trace : $[\neg h@0], [\neg b@1], [\neg a@2], \neg c, \neg d, e, \boldsymbol{\neg f}$

$$\begin{aligned}\alpha_2 = \eta[f, \sigma_4, \alpha_1] &= \eta[f, \neg e@2 \vee \neg f@2, \neg e@2 \vee h@0 \vee f@2] \\ &= \neg e@2 \vee \cancel{\neg f@2} \vee \cancel{\neg e@2} \vee h@0 \vee \cancel{f@2} \\ &= \neg e@2 \vee h@0\end{aligned}$$

La clause α_2 ne contient plus qu'un littéral de niveau 2 : le premier UIP (1-UIP) !

Continuons les résolutions :

Trace : $[\neg h@0], [\neg b@1], [\neg a@2], \neg c, \neg d, \boldsymbol{e}$

$$\begin{aligned}\alpha_3 = \eta[e, \sigma_3, \alpha_2] &= \eta[e, c@2 \vee d@2 \vee e@2, \neg e@2 \vee h@0] \\ &= c@2 \vee d@2 \vee \cancel{e@2} \vee \cancel{\neg e@2} \vee h@0 \\ &= c@2 \vee d@2 \vee h@0\end{aligned}$$

Trace : $[\neg h@0], [\neg b@1], [\neg a@2], \neg c, \boldsymbol{\neg d}$

$$\begin{aligned}\alpha_4 = \eta[d, \sigma_2, \alpha_3] &= \eta[d, a@2 \vee \neg d@2, c@2 \vee d@2 \vee h@0] \\ &= a@2 \vee \cancel{\neg d@2} \vee c@2 \vee \cancel{d@2} \vee h@0 \\ &= a@2 \vee c@2 \vee h@0\end{aligned}$$

Trace : $[\neg h@0], [\neg b@1], [\neg a@2], \boldsymbol{\neg c}$

$$\begin{aligned}\alpha_5 = \eta[c, \sigma_1, \alpha_4] &= \eta[c, a@2 \vee b@1 \vee \neg c@2, a@2 \vee c@2 \vee h@0] \\ &= a@2 \vee b@1 \vee \cancel{\neg c@2} \vee \cancel{a@2} \vee \cancel{c@2} \vee h@0 \\ &= a@2 \vee b@1 \vee h@0\end{aligned}$$

La clause α_5 ne contient plus qu'un littéral de niveau 2, c'est la décision, le dernier UIP (L-UIP) !

La clause α_2 contient le 1-UIP, c'est-à-dire que ce littéral aurait dû être propagé avant que la décision ait lieu afin d'éviter le conflit. Ajouter cette résolvante à la base de clauses apprises ne change rien au problème. En effet, la formule est équivalente car la clause nouvellement ajoutée est une conséquence logique. Il nous faut juste remonter dans l'arbre là où le littéral assertif (l'IUP) aurait dû être propagé. Dans le cas de la clause α_2 : au niveau 0. Nous allons maintenant voir comment ajouter l'analyse de conflits à l'algorithme DPLL de façon à obtenir l'algorithme

CDCL.

Algorithme 3.2 : CDCL(Σ)

```
   Data : Σ a CNF
   Result : Returns a set of literals ; UNSAT otherwise
1  L ← ∅;
2  dl ← 0;
3  while true do
4      if UnitPropagation(Σ, L) = CONFLICT then
5          if dl = 0 then
6              return UNSAT;
7          BTlvl ← ConflictAnalysis(Σ, L);
8          Backtrack(Σ, L, BTlvl);
9          dl ← BTlvl;
10     else
11         if |L| = |vars(Σ)| then
12             return L;
13         l ← PickBranchingVariable(Σ, L);
14         L ← L ∪ {l};
15         dl ← dl + 1;
```

Afin de simplifier l'algorithme 3.2, nous considérons que la fonction UnitPropagation (ligne 4) retourne maintenant CONFLICT si un conflit a lieu ; sinon elle modifie en conséquence la formule Σ et met dans L les littéraux propagés. Tant que chaque variable propositionnelle n'a pas été décidée (ligne 11), nous choisissons un littéral et sa polarité (ligne 13), les heuristiques de sélection sont détaillées dans la section 3.4.3. Lorsqu'un conflit est décelé dans la propagation unitaire, si le niveau courant est supérieur à zéro, nous appliquons l'analyse de conflits ; sinon le problème est prouvé insatisfiable. Lorsque la clause assertive a été calculée (ligne 7), il suffit d'effectuer le saut arrière (ligne 8 - fonction Backtrack).

Dans la suite de la section, nous présentons les améliorations apportées à l'algorithme CDCL pour arriver aux solveurs SAT dit modernes.

3.4.2 Structure de données

Le nombre de clauses apprises pouvant être très important et dans le but d'exploiter les clauses générées de manière optimale, une structure de données paresseuse nommée *two-watched literals* (2WL) a été proposée (Zhang et Malik (2002)). Elle permet d'identifier les clauses conflictuelles, celle satisfaites, ainsi que les clauses qui deviendraient unitaires lors de la propagation.

Cette structure paresseuse utilise, comme son nom l'indique, deux sentinelles afin de détecter ces cas intéressants. Concernant les clauses conflictuelles, un seul littéral pourrait suffire. En effet, surveiller un seul littéral ne permet que de détecter les clauses qui entrent en conflit. Lorsque le littéral surveillé est falsifié, il suffit de parcourir la clause afin de trouver un littéral non décidé à surveiller. Dans le cas où aucun littéral ne peut être choisi, la clause est donc falsifiée et un conflit est détecté. Dans le cas où un littéral est décidé positivement, alors la clause courante est satisfaite et ne sera plus utilisée. Mais dans le cas des clauses unitaires, un seul littéral surveillé

obligerait à parcourir la clause intégralement à chaque fois. C'est pour cela que Zhang et Malik (2002) ont proposé d'en utiliser deux (Figure 3.2).

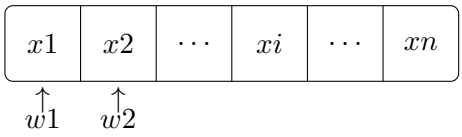


FIGURE 3.2 – Cas de départ des 2WL.

Plusieurs cas sont possibles en utilisant deux littéraux : les deux littéraux ne sont pas décidés ; au moins un des deux littéraux est vrai ou au moins un des deux littéraux est faux. Le premier cas (Figure 3.3) signifie que la clause n'est ni falsifiée ni unitaire. Aucune action supplémentaire n'est donc requise.

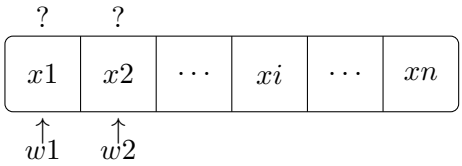


FIGURE 3.3 – Cas 1 des 2WL : les deux littéraux ne sont pas décidés.

Le second cas (Figure 3.4), lorsqu'au moins un des deux littéraux surveillés est vrai, implique que la clause est satisfaite, la surveiller n'est plus nécessaire et par conséquent aucune action supplémentaire est requise. En effet, dans le but de conserver la structure non coûteuse dans le cas des retours arrière (*backtrack-free*), les sentinelles peuvent rester en place afin d'être utilisées à nouveau si l'algorithme de recherche devait revenir à un nœud précédent.

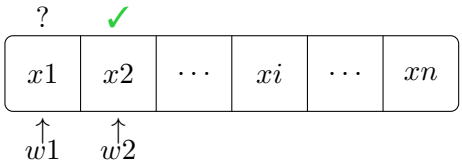


FIGURE 3.4 – Cas 2 des 2WL : au moins un des deux littéraux surveillés est vrai.

Le dernier cas à décrire est lorsque au moins un des littéraux surveillés est falsifié, ce cas dépend de l'état du second littéral surveillé.

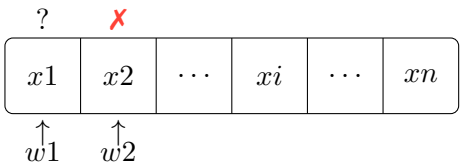


FIGURE 3.5 – Cas 3 des 2WL : au moins un des littéraux surveillés est falsifié.

Deux nouveaux cas se présentent : soit la seconde sentinelle est indéterminée, soit elle est falsifiée. Si la seconde sentinelle est indéterminée, il faut parcourir la clause pour trouver un nouveau littéral afin de remplacer celui falsifié (Figure 3.6, à gauche), si aucun n'est disponible

alors le second littéral doit être propagé car la clause est devenue unitaire (Figure 3.6, à droite). En effet, ne pas trouver de littéral sentinelle signifie que tous les littéraux potentiels sont falsifiés.

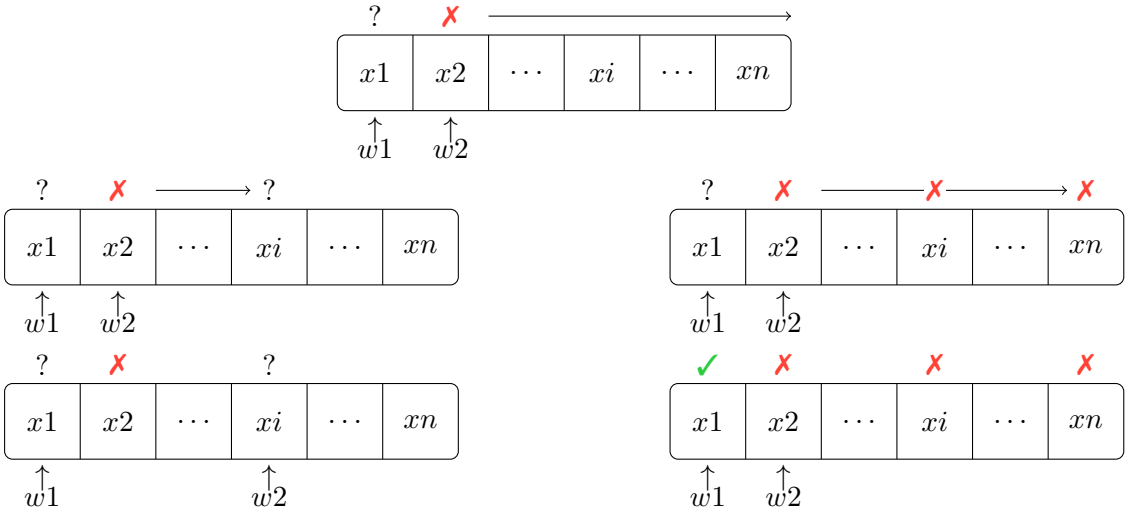


FIGURE 3.6 – Cas 3.a des 2WL : un des littéraux surveillés est falsifié et la seconde sentinelle est indéterminée.

Si la seconde sentinelle est falsifiée (Figure 3.7), de la même manière il faut parcourir la clause afin de trouver un nouveau littéral, si aucun n'est disponible, la clause est falsifiée et un conflit est détecté.

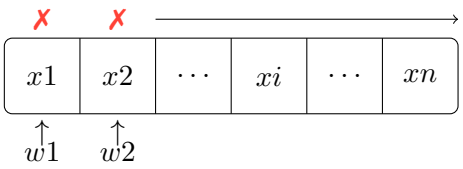


FIGURE 3.7 – Cas 3.b des 2WL : les deux sentinelles sont falsifiées.

En pratique, toujours dans le but d'être le plus efficace possible, les deux premières places de la clause seront réservées pour les sentinelles. En effet, comme l'ordre des littéraux au sein d'une clause n'est pas déterminant, nous pouvons choisir de modifier la place des littéraux plutôt que de bouger les pointeurs effectifs. Les deux premiers littéraux choisis pour être surveillées seront le littéral assertif lors de la création de la clause, ainsi que celui qui permet de déterminer le niveau de saut arrière, le littéral au niveau le plus élevé hors littéral assertif.

Afin d'éviter de parcourir intégralement l'ensemble des clauses apprises, à chacun des littéraux de la formule est associé un tableau. Dans ce tableau, sont placés les indices des clauses dont le littéral associé est sentinelle. Cela permet, lorsqu'un littéral x est décidé (ou propagé), de ne devoir regarder que le tableau associé à son complémentaire ($\neg x$) afin de vérifier qu'aucun conflit ne survient lors de cette décision. Ces tableaux doivent, bien entendu, être maintenus lorsque les sentinelles changent. Mais l'efficacité de leur utilisation compense largement ce coût de mise à jour lors des changements de sentinelles. En effet, ce parcours, bien que partiel, de la base de clauses est suffisant afin de déclencher les propagations unitaires durant le parcours de l'espace de recherche.

Exemple 15

Soit la base de clauses apprises suivante : $\sigma_1 = a \vee b \vee c$, $\sigma_2 = \neg b \vee c$ et $\sigma_3 = \neg a \vee \neg b$. La figure 3.8 représente le tableau d'indices des sentinelles.

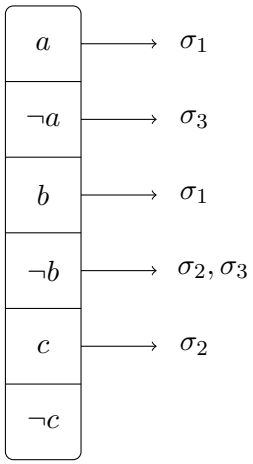


FIGURE 3.8 – Tableau d'indices des littéraux sentinelles de l'exemple 15

L'utilisation de cette structure paresseuse explique, en partie, l'efficacité des solveurs SAT modernes, car une très grand partie (plus de 80% sur certaines instances) des affectations sont faites par propagation unitaire.

3.4.3 Heuristiques

Au sein des algorithmes 3.1 (ligne 7) et 3.2 (ligne 13) des fonctions de choix de variables sont appelées. Le choix de la variable, voire de sa polarité, est déterminant quand à l'efficacité globale de l'algorithme. En effet, à la manière de la programmation par contrainte, utiliser la structure avec des heuristiques adaptatives permet, en général, d'obtenir de meilleurs résultats en pratique. Des choix de variable plus judicieux peuvent réduire l'arbre de recherche de manière exponentielle (Li et Anbulagan (1997)). Choisir la variable optimale (afin d'obtenir l'arbre de recherche le plus petit) est un problème `NP-difficile` (Liberatore (2000)). En pratique choisir la variable optimale n'est clairement pas viable, ce choix se fait donc de manière heuristique.

De nombreuses heuristiques ont été proposées afin d'améliorer l'efficacité des solveurs SAT. Les premières proposées, les heuristiques dites statiques (Buro et Büning (1992), Jeroslow et Wang (1990), Hooker et Vinay (1995), Barth (1995)) se basent que sur l'étude de la formule initiale afin de déduite un ordre de propagation des littéraux. En général le but recherché est de satisfaire le maximum de clauses possible à chaque choix effectué.

Ensuite, des heuristiques basées sur les informations des structures de données, à savoir, le nombre de littéraux satisfaits, falsifiés et non définis ont été proposées par Marques-Silva (1999). Ce type d'heuristique est mis-à-jour durant la recherche et prend aussi en compte les clauses apprises.

D'autres types d'heuristiques ont été proposés, les heuristiques dites *look-ahead* (Li et Anbulagan (1997), Freeman (1995), Pretolani (1993)). Le but des heuristiques de type *look-ahead* est

d'anticiper le comportement du solveur. Elles peuvent, par exemple, utiliser la propagation unitaire afin d'estimer le meilleur choix de variable en fonction du nombre de propagations induites.

Enfin, les heuristiques de type *look-back* (Zhang *et al.* (2001), Liang *et al.* (2016)) ont été introduites. La première étant l'heuristique VSIDS (*Variable State Independent Decaying Sum*). Cette heuristique a été introduite car les structures de données paresseuses ne permettent pas de connaitre l'état complet du problème (taille des clauses, nombre de littéraux falsifiés, etc.). De ce fait, les heuristiques dynamiques ne sont pas compatibles avec cette amélioration des structures de données.

L'heuristique VSIDS fonctionne selon le principe suivant : elle sélectionne les littéraux qui apparaissent le plus fréquemment dans les clauses apprises ; autrement dit, les causes des conflits récents. De ce fait, à chaque littéral est associé un compteur initialisé à zéro appelé *activité*. Quand une clause est ajoutée à la base, le compteur de chaque littéral apparaissant au sein de la clause est incrémenté. L'heuristique, qui a un système de vieillissement (*Decaying*) va, de temps en temps, réduire tous les compteurs en les divisant par une constante. De ce fait, les littéraux des clauses les plus récentes auront un poids plus important. Cela permet d'intensifier la recherche dans la partie du problème qui est estimée la plus difficile.

Ce type d'heuristique choisit le littéral le plus prometteur mais ne considère pas la polarité. En effet, contrairement à l'algorithme DPLL qui utilise la règle de séparation, CDCL est équipé de retours arrière non chronologiques. C'est-à-dire que lorsque un littéral est décidé, son complémentaire ne sera pas forcément décidé si un conflit est décelé. Pour cela, il faut faire appel à des heuristiques dites de polarité. Au sein du solveur `Minisat`, Eén et Sörensson (2003) affectent les variables propositionnelles toujours de manière négative. Une autre façon de choisir la polarité est le *progress saving* (Pipatsrisawat et Darwiche (2007)). Lorsque une variable propositionnelle reçoit une polarité, elle va être sauvegardée afin d'être utilisée à nouveau après un retour arrière. Cela permet d'éviter d'utiliser à nouveau du temps de calcul pour retrouver une polarité intéressante car, lorsqu'un conflit est détecté, il est probable que plusieurs sous problèmes ont été résolus avant d'y arriver. Le fait de se souvenir de la manière dont les variables ont été affectées permet de les résoudre à nouveau. Le seul point négatif de cette approche est la diversification faible de la recherche.

3.4.4 Réduction de la base de clauses apprises

Garder l'intégralité des clauses apprises n'est, en général, pas une idée intéressante. En effet, la base de clause apprises grandissante va consommer de plus en plus de mémoire, ce qui va rendre le parcours plus lourd voire arriver à consommer l'intégralité de la mémoire disponible. De plus, toutes les clauses apprises n'ont pas la même valeur au sein d'un solveur. En effet, les clauses très grandes ont très peu de chances d'être utiles dans la recherche et risque de ralentir le processus de résolution (Marques-Silva et Sakallah (1999), Eén et Sörensson (2003)).

Différentes politiques de réduction de la base de clauses apprises ont été proposées (Audemard *et al.* (2011), Eén et Sörensson (2003), Audemard et Simon (2009b)) afin de pallier ce problème. À la manière des heuristiques, elles peuvent se baser sur différentes informations. Notamment, la taille des clauses apprises peut être limitée afin d'avoir uniquement des clauses de tailles raisonnables, qui ont plus de chances d'être utiles à la recherche (Dechter (1990)). La plupart des solveurs conservent les clauses de taille 2, dites binaires, car celles-ci sont souvent utilisées et améliorent la propagation unitaire. Bien-sûr, les clauses raisons, c'est-à-dire les clauses qui ont propagé un littéral, sont aussi conservées afin de pouvoir compléter le processus d'analyse

de conflits. Les solveurs SAT modernes filtrent rarement les clauses en entrée mais appliquent la réduction de la base de clauses de manière périodique. Cela signifie que toutes les clauses sont ajoutées dans la base, lorsque que la base est pleine (selon une valeur définie initialement), une partie de la base est oubliée selon une heuristique et la taille de la base est agrandie selon une politique.

La première des heuristiques utilisée afin de simplifier la base s'appuie sur l'heuristique de choix de variables VSIDS (Eén et Sörensson (2003), voir section 3.4.3). De manière similaire, un compteur d'activité est associé à chaque clause apprise. Il est incrémenté lorsque la clause associée est utile à l'analyse de conflits. Comme pour VSIDS, un processus de vieillissement est associé à cette heuristique afin de conserver les parties conflictuelles récentes les plus importantes. La moitié des clauses (les moins actives) est donc supprimée lorsque le processus de réduction est invoqué.

Une seconde heuristique est le LBD (*Literal Block Distance*, Audemard et Simon (2009b)). Quand une clause est apprise, son LBD est calculé. Cette mesure correspond au nombre de niveaux de décision différents dans la clause générée.

Exemple 16

Soit une clause $x_2^5 \vee x_8^3 \vee x_6^4 \vee x_1^3$, son LBD est de 3.

Elle peut être mise à jour lorsque cette clause est utilisée ultérieurement dans l'analyse de conflits ; si le nouveau LBD est inférieur, il remplace le précédent. Audemard et Simon (2009b) ont montré que les clauses ayant un LBD faible sont plus importantes pour la recherche. On supprime donc, lors de la réduction de la base, la moitié de la base qui contient les clauses avec le LBD le plus important.

La réduction de la base de clauses est appliquée de manière périodique, car, des appels trop fréquents peuvent ralentir le solveur et faire perdre des clauses apprises potentiellement intéressantes. Au contraire, faire trop peu d'appels à cette procédure peut réduire l'efficacité de la base de clauses apprises. À chaque application de la procédure, la taille de la base est augmentée dans le but de conserver la complétude. En effet, pratiquer l'oubli de clauses, en plus des redémarrages (Section 3.4.5), sur une base de clause limitée pourrait conduire à une boucle infinie car le même espace de recherche peut être exploré de manière répétée.

Deux approches largement utilisées pour réduire la base de clauses apprises sont celles de Eén et Sörensson (2003) et Audemard et Simon (2009b). La première approche appelle la procédure de réduction de la base de clauses apprises lorsque le nombre de clauses ajoutée est égal à un tiers de la taille de la base du problème initial. Ensuite le seuil est augmenté de 10% à chaque appel ultérieur. La seconde consiste à donner une première taille fixe à la base de clauses, 20000 dans le cas de Audemard et Simon (2009b), puis de l'augmenter de 500 à chaque nouvel appel.

3.4.5 Redémarrages

Les heuristiques de choix de variables n'étant pas, par définition, optimales, il arrive que les premiers choix ne soient pas judicieux. En effet, les heuristiques basées sur, par exemple, l'activité, doivent dans un premier temps se régler grâce à un certain nombre de conflits. Au début de la résolution du problème, les compteurs étant tous initialisés à la même valeur, le choix des premières variables de décision se fait soit aléatoirement, soit lexicographiquement.

Afin de corriger ces premiers choix qui peuvent, potentiellement, compliquer la résolution du problème, recommencer la recherche sur un arbre vierge est une idée judicieuse. Le parcours d'arbre précédent n'est pas pour autant mis à l'écart, les clauses apprises, ainsi que les scores des heuristiques sont conservés. Cela permet de s'échapper d'impasses très difficiles à prouver insatisfiables ou d'attaquer le même sous problème difficile d'un angle différent. Les redémarrages sont régis par des politiques, à la manière de la réduction de la base de clauses apprises. En effet, deux valeurs sont importantes, la valeur dites de coupure (*cutoff*), qui est le nombre de conflits à atteindre avant de déclencher un redémarrage ; et la formule qui permet d'augmenter cette valeur de coupure afin de conserver la complétude. À l'instar des heuristiques, les stratégies de redémarrage se divise en deux catégories : les politiques statiques et dynamiques (ou adaptatives).

Les stratégies statiques utilisent en général des suites mathématiques, soit géométriques soit plus complexes, par exemple, la suite de Luby *et al.* (1993). Les suites géométriques ont été utilisées pour la première fois efficacement par Minisat 1.13 (Sorensson et Een (2005)). En effet, la valeur de coupure est initialisée à 100 conflits, cette valeur est augmentée de 50% à chaque redémarrage (100, 150, 225, ...). La suite de Luby est une séquence de la forme (1,1,2,1,1,2,4,...) qui en pratique est multipliée par une constante. Elle est définie comme suit :

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

avec $i, k = 1, 2, \ldots$ Cette suite est notamment utilisée avec une constance de 100 dans Minisat 2.1 (Sörensson et Eén (2009)) avec succès.

Les stratégies dynamiques exploitent les informations du problème, l'évolution de la hauteur moyenne des sauts (Hamadi *et al.* (2009)), les changements de polarité des variables lors de l'affectation (Biere (2008)), la moyenne glissante des niveaux de décision sur les 100 derniers conflits (Audemard et Simon (2009a)) (si elle est supérieure à 0.7 fois la moyenne de tous les niveaux de décision depuis le début de la recherche alors un redémarrage est effectué) ou la qualité des clauses apprises grâce au LBD (Audemard et Simon (2012)). Dans ce dernier cas, le solveur redémarre si les clauses produites commence à avoir un LBD plus important que la moyenne glissante des clauses précédemment générées.

3.5 Conclusion

Le problème SAT est l'un des plus connus de la programmation par contrainte, par conséquent de nombreuses méthodes ont été proposées afin d'en améliorer la résolution. Le problème SAT est une pierre angulaire de la programmation par contrainte car il utilise un langage simple et l'efficacité de ses méthodes de résolution n'est plus à prouver. De ce fait, il est souvent utilisé en boite noire pour des problèmes d'une complexité plus importante.

Précédemment, la recherche SAT était focalisée sur la découverte de nouvelles méthodes et heuristiques, l'amélioration de la qualité des clauses et des sauts arrière, etc. En effet, depuis l'avènement de l'apprentissage de clauses avec l'algorithme CDCL, la majorité des solveurs SAT utilisent le niveau de saut arrière (ou retour arrière non chronologique) obtenu avec la clause assertive générée lors d'un conflit. Cependant, Nadel et Ryvchin (2018) proposent et démontrent expérimentalement que cela n'est pas forcément plus efficace. Ils reviennent donc à une forme de retour arrière chronologique, à la manière de l'algorithme DPLL.

À la vue des résultats convainquants sur le problème SAT grâce à l'algorithme CDCL et notamment l'apprentissage de clauses qui s'inspire des *nogoods*, une partie de la recherche s'est concentrée sur l'adaptation des algorithmes SAT vers CSP. Dans le chapitre suivant, nous présentons trois méthodes les plus connus dans ce que nous appelons l'hybridation SAT/CP ; à savoir les *nogoods* généralisés, les explications paresseuses ainsi que la génération paresseuse de clauses.

Chapitre 4

Hybridation SAT/CSP

Sommaire

Dans les chapitres précédents, nous avons décrit le problème de satisfaction de contraintes ainsi que le problème de satisfaction booléenne. Mais ces deux piliers possèdent leurs forces et faiblesses. Dans le cas du problème de satisfaction de contraintes, l'utilisation de variables ainsi que de contraintes, parfois globales, permet une conception des problèmes concise et précise. En contrepartie, un haut niveau de représentation nécessite souvent des algorithmes dédiés coûteux. Cela est bien moins marqué en SAT, où les seules contraintes disponibles sont les clauses. La propagation unitaire, très puissante sur les clauses est l'un des points forts du cadre SAT. Pour toutes ces raisons, l'hybridation des deux domaines est très intéressante et a déjà donné lieu à des méthodes performantes.

Dans ce chapitre, nous décrivons trois approches qui ont pour but de rapprocher et d'hybrider le problème CSP avec le problème SAT. La première concerne les *nogoods* généralisés (Katsirelos et Bacchus (2003)), une extension des *nogoods* standards en s'inspirant de l'analyse de conflits SAT. La seconde méthode porte sur les explications paresseuses (Gent *et al.* (2010)), qui permettent de générer des clauses au sein d'un solveur CP grâce à des fonctions de raisonnement *ad-hoc* tout en limitant le coût de calcul des explications. La dernière approche présentée au sein de ce chapitre est la génération paresseuse de clauses (Ohrimenko *et al.* (2007; 2009)). Cette méthode intègre à la fois un moteur CP et un moteur SAT, le premier pilotant le second.

4.1 *Nogoods* généralisés

En programmation par contraintes, contrairement aux clauses SAT, seules des formes dites restreintes de *nogoods* sont généralement utilisées. En nous plaçant dans le contexte d'enregistrement de *nogoods* à chaque conflit, gérer une base conséquente de nogoods peut être très coûteux en espace et en temps. De plus, ces *nogoods* ne supportent pas la propagation unitaire. Sur ce constat, Katsirelos et Bacchus (2003) ont proposé une forme généralisée des *nogoods* dans le but d'adapter les techniques SAT en satisfaction de contraintes.

Afin de comprendre le parallèle entre les clauses et les nogoods, il suffit de considérer l'encodage direct (de Kleer (1989), Walsh (2000)). Chaque affectation possible $x = a$ ($x \leftarrow a$) devient une variable propositionnelle, qui est vraie si la variable x est affectée à la valeur a. La négation est notée $x \neq a$ ($x \nleftarrow a$) et indique, si elle est vraie, que la valeur a est supprimée du domaine

de x. En satisfaction de contraintes, le fait qu'une variable doive être assignée à une et une seule valeur est implicitement codé par la simple définition des domaines. Pour traduire ce concept en SAT, il faut ajouter des contraintes dites *atMostOne* (au plus une) et *atLeastOne* (au moins une) afin qu'une variable ait au moins une valeur propositionnelle qui représente une affectation. Les supports des contraintes peuvent aussi être encodés de cette manière afin de compléter la traduction.

Exemple 17

Soit la variable x avec $\texttt{dom}(x) = \{0, 1, 2\}$.

Son encodage direct est réalisé par l'introduction de trois variables propositionnelles $x \leftarrow 0$, $x \leftarrow 1$ et $x \leftarrow 2$.

La contrainte *atLeastOne* sur x correspond à la clause suivante :

$$x \leftarrow 0 \vee x \leftarrow 1 \vee x \leftarrow 2$$

En effet, pour que la clause soit satisfaite, il suffit d'avoir au moins un des littéraux du domaine à vrai.

La contrainte *atMostOne* sur x correspond aux implications suivantes :

$$x \leftarrow 0 \Rightarrow x \nleftarrow 1$$
$$x \leftarrow 0 \Rightarrow x \nleftarrow 2$$
$$x \leftarrow 1 \Rightarrow x \nleftarrow 2$$

Ces implications peuvent être mises sous forme clausale de la manière suivante :

$$x \nleftarrow 0 \vee x \nleftarrow 1$$
$$x \nleftarrow 0 \vee x \nleftarrow 2$$
$$x \nleftarrow 1 \vee x \nleftarrow 2$$

Lorsqu'un des littéraux encodant une affectation devient vrai, cela va impliquer la négation des autres littéraux du domaine afin de satisfaire cet ensemble de clauses.

Nous voyons clairement que sous cet encodage, un *nogood* standard $x_1 \leftarrow a_1, \ldots, x_i \leftarrow a_i$ qui peut s'écrire $\neg(x_1 \leftarrow a_1 \wedge \ldots \wedge x_i \leftarrow a_i)$ est équivalent à la clause $x_1 \nleftarrow a_1 \vee \ldots \vee x_i \nleftarrow a_i$. En effet, afin d'éviter de reproduire le conflit, au moins une affectation du *nogood* doit être falsifiée. Cela est équivalent, sous forme clausale, à considérer qu'au moins une réfutation doit être vérifiée.

Les *nogoods* généralisés sont une forme plus puissante des *nogoods* standards. De manière analogue aux clauses en SAT, ils ne contiennent pas uniquement des décisions positives mais également des décisions négatives. Une solution d'un problème de satisfaction de contraintes affecte à chaque variable du problème une unique valeur. Implicitement, cela induit que toutes les autres valeurs sont supprimées des domaines des variables. Nous pouvons définir le concept de solution étendue d'un problème de satisfaction de contraintes, en capturant aussi toutes les suppressions, de la manière suivante.

Définition 38 (Solution étendue)

Une **solution étendue** à un problème de satisfaction de contraintes est une solution standard à laquelle toutes les valeurs supprimées ont été ajoutées. Formellement, c'est un ensemble $\{(x\ op\ v) | x \in \mathcal{X} \wedge v \in \texttt{dom}(x)\}$ avec $op \in \{=, \neq\}$.

Remarque 10

Une solution étendue doit contenir, pour chaque valeur a de chaque variable x, soit $x \leftarrow a$ ou $x \nleftarrow a$. Si la solution étendue ne contient pas $x \leftarrow a$ alors elle doit contenir $x \nleftarrow a$ et la variable x doit être assignée à une autre valeur.

La définition de solution étendue nous permet de définir formellement un *nogood* généralisé.

Définition 39 (*Nogood* généralisé)

Un ***nogood* généralisé** est un ensemble de décisions positives et/ou négatives qui n'est contenu dans aucune solution étendue.

Remarque 11

Les *nogoods* standards sont un sous-ensemble des *nogoods* généralisés.

Exemple 18

Soit la contrainte $x + y < z$ sur les variable x, y et z.

On considère $\texttt{dom}(x) = \texttt{dom}(y) = \{0, 1, 2\}$ et $\texttt{dom}(z) = \{1, 2, 3\}$.

La contrainte est violée par l'affectation $\Sigma = \{x \leftarrow 0, y \leftarrow 1, z \leftarrow 1\}$. Par conséquent, Σ est un *nogood*.

Si plus tard lors de la recherche, les choix $y \leftarrow 1$ et $z \leftarrow 1$ sont repris, la valeur 0 devrait être supprimée du domaine de x. Mais il est clairement impossible de satisfaire la contrainte en affectant la valeur 1 à la variable z et en supprimant la valeur 0 du domaine de x. En effet, la somme des variables x et y serait alors au minimum de 1 si la variable y prenait la valeur 0. Un nouveau conflit serait alors soulevé.

Cela ne peut pas être exprimé avec les *nogoods* standards, car ils ne gèrent que des affectations. Avec les *nogoods* généralisés, il est tout-à-fait possible d'encoder cela avec le *nogood* suivant :

$$\{x \nleftarrow 0, z \leftarrow 1\}$$

Remarque 12

Les *nogoods* généralisés peuvent encoder plusieurs *nogoods* standards.

Exemple 19

Soient les variables x et y avec $\texttt{dom}(x) = \texttt{dom}(y) = \{0, 1, 2\}$.

Soient $\{x \leftarrow 0, y \leftarrow 0\}$ et $\{x \leftarrow 0, y \leftarrow 1\}$ des *nogoods* standards. Il peuvent être réécrit en un seul *nogood* généralisé qui capture les mêmes informations.

$$\{x \leftarrow 0, y \nleftarrow 2\}$$

Afin de découvrir des *nogoods* généralisés, Katsirelos et Bacchus (2003) proposent le schéma de première décision (*first-decision scheme*). À la manière de l'analyse de conflits SAT, cette méthode itérative remplace les décisions les plus profondes dans le *nogood* jusqu'à obtenir un *nogood* généralisé dont la décision la plus profonde est une décision du solveur. Ce point d'arrêt est semblable au dernier UIP de l'analyse de conflits SAT. La profondeur d'une décision (positive ou négative) est définie le principe suivant : le niveau de décision puis l'ordre des décisions si le niveau de décision est le même.

Inclure des décisions négatives au sein des *nogoods* permet d'induire des suites de propagations, à la manière de la propagation unitaire du problème de satisfaction booléenne. De plus, les *nogoods* généralisés permettent d'adapter les techniques performantes d'enregistrement de clauses de SAT en satisfaction de contraintes.

Dans la section suivante, les explications paresseuses sont introduites. Elles s'inspirent des *nogoods* généralisés afin de générer des explications à la volée des contraintes impliquées dans la raison d'un conflit grâce à des fonctions de raisonnement dédiées.

4.2 Explications paresseuses

Les explications forment une pierre angulaire du raisonnement et de l'apprentissage en programmation par contraintes. Cependant, une grande partie des techniques qui utilisent les explications, le font de manière assez coûteuse. Gent *et al.* (2010) ont donc proposé une forme paresseuse de génération d'explication. En effet, l'apprentissage dans le domaine de la programmation par contraintes est principalement employé afin de découvrir de nouvelles contraintes non décrites dans le problème original, mais qui se devinent au fur et à mesure de la recherche.

Prenant comme base les *nogoods* généralisés, qui utilisent aussi à leur manière les explications, nous présentons ici les travaux de Gent *et al.* (2010).

Définition 40 (Explication)

L'**explication** d'une suppression $x \nleftarrow a$ est un ensemble de décisions positives et négatives qui suffisent à un propagateur pour inférer $x \nleftarrow a$. De manière équivalente, l'**explication** d'une affectation $x \leftarrow a$ est un ensemble de décisions positives et négatives qui suffisent à un propagateur pour inférer $x \leftarrow a$.

Une explication peut donc être extraite pour chaque suppression (ou affectation) de valeur effectuée dans le cadre du processus de filtrage. Mais les décisions du solveur, faites lors des branchements, obtiendront une explication vide (car faites heuristiquement). De plus, nous devons considérer deux cas d'explications supplémentaires : lorsqu'une seule valeur reste dans le domaine d'une variable, le solveur agit comme si la variable avait été affectée ; et lorsqu'à l'inverse, une assignation est effectuée, le solveur va supprimer toutes les autres valeurs du domaine.

À la manière des *nogoods* généralisés, un ordre doit être conservé sur les affectations et les suppressions. L'ordonnancement est fait selon le niveau de décision puis l'ordre des décisions si le niveau de décision demeure le même.

En général, manipuler des explications implique de stocker une explication (un ensemble de décisions positives et négatives) quand la suppression d'une valeur se produit. Gent *et al.* (2010) proposent de sauvegarder suffisamment d'informations afin de reconstruire l'explication plus tard, si elle doit servir dans la même branche de l'arbre de recherche. C'est ici que se situe l'idée d'explications paresseuses. En effet, conserver, au moment de la suppression, une trace de celle-ci ainsi qu'une fonction qui permet de calculer son explication offre un gain d'espace et de temps non négligeable. Ce gain reste substantiel au sein d'un solveur de programmation par contraintes. Lors de la phase de propagation, prendre du temps afin de stocker des ensembles (parfois de grande taille) peut devenir très dommageable. Si une explication participe à un conflit, nous pouvons passer la suppression induite en paramètre de la fonction de raisonnement associée dans l'intention d'en avoir la raison.

Afin de réaliser ce calcul paresseux d'explications, les auteurs de Gent *et al.* (2010) ont proposé des fonctions de raisonnement pour différentes contraintes. Ces fonctions sont présentées par la suite. Dans le cas où aucune fonction n'est disponible, un stockage lourd des explications reste, bien sûr, toujours possible.

Explications paresseuses pour les clauses Les clauses peuvent déclencher des décisions grâce à la propagation unitaire. À la manière des techniques SAT, les clauses ajoutées dans un solveur de programmation par contraintes vont pouvoir fournir une explication des propagations unitaires réalisées. En effet, par définition, une propagation unitaire a lieu uniquement lorsque tous les littéraux d'une clause sont faux, et qu'un seul n'est pas affecté. Ce dernier est par conséquent propagé afin de satisfaire la clause considérée. L'explication de cette propagation induite est, dans ce cas, tous les autres littéraux de la clause, c'est-à-dire ceux qui sont faux.

Exemple 20

Soit la clause $\Sigma = a \vee \neg b \vee c \vee d$. Si dans l'état courant a, $\neg b$ et c sont faux, alors la propagation unitaire va affecter d à vrai et nous sauvegardons le fait que cette

propagation est réalisée par Σ. Si l'explication de d est requise pour expliquer un conflit qui a eu lieu plus bas dans la branche, nous savons que sa propagation provient de la clause Σ. L'explication de d est donc $a \vee \neg b \vee c$.

Explications paresseuses pour les contraintes en extension Au sein d'une contrainte en extension (positive), une valeur peut être éliminée du domaine d'une variable dans le cas où elle n'apparaîtrait plus dans aucun support valide. La façon d'expliquer ce genre de suppression est dépendante de la manière dont les tuples sont représentés. Une méthode simple afin de construire cette explication est de considérer la cause de l'invalidité de chaque tuple où la valeur à expliquer apparaît. Un tuple devient caduc lorsqu'une valeur qui appartient à celui-ci est invalidée. L'explication est donc un ensemble de valeur (appelé *couverture*) qui invalide tous les tuples où la valeur à expliquer apparaît.

Explications paresseuses pour les inégalités Si une contrainte du type $x < y$ réfute la valeur a du domaine de x, cela signifie que toutes les valeurs de y supérieures ou égales à a ont été supprimée du domaine de y. L'explication de $x \nleftarrow a$ peut donc être calculée sur demande et contient toutes les valeurs supérieures ou égales à a dans le domaine de y. Formellement, l'explication de $x \nleftarrow a$ est l'ensemble $\{y \nleftarrow a+1, \ldots, y \nleftarrow max(\texttt{dom}(y))\}$.

Exemple 21

Soit la contrainte $x < y$ avec $\texttt{dom}(x) = \{0, 1, 2\}$ et $\texttt{dom}(y) = \{2, 3, 4\}$. Supposons que la valeur 3 du domaine de la variable y ait précédemment été supprimée (le propagateur ne peut rien inférer) et considérons que dans l'état courant, la valeur 4 vient d'être supprimée de $\texttt{dom}(y)$. Dans cet état, la variable y est alors assignée à 2 et la valeur 2 du domaine de la variable x va être supprimée par le propagateur. L'explication de $x \nleftarrow 2$ est $\{y \nleftarrow 3, y \nleftarrow 4\}$.

Explications paresseuses pour la contrainte globale *allDifferent* Dans Gent *et al.* (2010), un calcul d'explication pour la contrainte globale *allDifferent* est proposé dans le cas où celle-ci est propagée par l'algorithme GAC de Régin (1994), utilisant les couplages sur un graphe biparti. Ce graphe représente les variables de la portée de la contrainte *allDifferent* ainsi que les valeurs des domaines de ces variables. Sachant que cet algorithme produit des couplages complets sur ces deux parties afin d'effectuer la propagation, nous pouvons considérer que le dernier couplage généré par l'algorithme avant une suppression est consistant. En considérant l'ordre des suppressions, nous pouvons donc retrouver le couplage parfait qui a permis la suppression d'une valeur. L'explication d'une suppression est l'ensemble de toutes les suppressions dans les domaines des variables impliquées dans ce couplage parfait.

Explications paresseuses génériques Enfin, expliquer de manière générique une suppression de n'importe quel propagateur est possible. Pour cela, nous devons considérer toutes les suppressions des autres variables de la portée de la contrainte examinée qui ont eu lieu avant la suppression que nous voulons expliquer. Bien sûr, une explication dédiée à l'implémentation

d'une contrainte produira des explications plus petites, par conséquent plus efficaces et pourra réduire leur temps de calcul.

Dans la section suivante, la génération paresseuse de clauses est présentée. Cette approche, qui s'inspire du problème de satisfiabilité modulo des théories (Barrett *et al.* (2009)), utilise les propagateurs CP afin de générer des clauses qui décrivent les réductions de domaines et les conflits qui peuvent survenir lors de la propagation.

4.3 Génération paresseuse de clauses

Dans cette section, nous présentons la génération paresseuse de clauses (*lazy clause generation*, Ohrimenko *et al.* (2007; 2009)) ainsi que sa forme révisée (Feydy et Stuckey (2009)). La génération de clauses paresseuses est une approche hybride performante. Dans sa forme originale, le point novateur de cette stratégie est qu'elle considère les propagateurs CP comme des générateurs de clauses. Les propagateurs décrivent leur comportement peu à peu au solveur SAT afin que celui-ci puisse raisonner sur ces clauses transmises à la manière du problème de satisfiabilité modulo des théories (Barrett *et al.* (2009)). L'implémentation originale (Ohrimenko *et al.* (2007; 2009)) exploite un solveur de programmation par contraintes à l'intérieur d'un solveur SAT. La méthode révisée (Feydy et Stuckey (2009)) utilise le solveur SAT comme un propagateur au sein d'un solveur de programmation par contraintes.

Le but de la génération paresseuse de clauses est de tirer parti de la représentation concise de CSP et de profiter de la puissance de propagation de SAT. En effet, de par son expressivité, grâce aux contraintes globales notamment, représenter clairement des problèmes en programmation par contrainte est très facile contrairement à SAT où le seul outil disponible est la clause. Le point fort de SAT est la propagation unitaire puissante et l'analyse de conflits couplées au retour arrière non chronologique ainsi qu'à l'activité des littéraux grâce à l'heuristique VSIDS. Le principe de l'hybridation est donc pertinent. Nous avons vu précédemment que traduire un domaine en encodage direct requiert une clause *atLeastOne* et $\binom{n}{2}$ clauses où n est la taille du domaine pour encoder le *atMostOne*. Clairement, pour représenter un domaine CP en SAT en utilisant l'encodage direct, un nombre important de clauses est requis. Cela illustre que la représentation est plus concise en CP.

Avec la génération paresseuse de clauses, les propagateurs CP sont utilisés afin de générer des clauses de la manière suivante : quand un propagateur f est appliqué sur un domaine D afin d'obtenir $f(D)$, si $f(D) \neq D$ alors une ou plusieurs clauses sont générées. Ces clauses encodent les changements qui ont eu lieu sur le domaine D. Dans le but de réaliser une traduction itérative plus concise, Ohrimenko *et al.* (2007) utilisent l'encodage d'ordre (Crawford et Baker (1994)).

Afin d'encoder une variable CP en SAT avec l'encodage d'ordre, nous avons besoin de définir les variables propositionnelles qui sont utilisées. Pour chaque valeur a du domaine d'une variable x, une variable propositionnelle $(x \leq a)$ est requise. Celle-ci indique que la variable X est inférieure ou égale à la valeur considérée lorsqu'elle est affectée positivement. Sa négation, $\neg(x \leq a)$, indique que la variable x est strictement supérieure à la valeur considérée. Grâce à ces variables propositionnelles, nous pouvons encoder plusieurs choses : l'affectation d'une variable à une valeur de son domaine, la suppression d'une valeur dans le domaine d'une variable ainsi que les bornes minimum et maximum d'une variable. Celles-ci s'écrivent de la manière suivante :

— si la borne maximum de la variable x est a alors $(x \leq a)$

— si la borne minimum de la variable x est a alors $\neg(x \leq a - 1)$

— si la variable x est affectée à a alors $\neg(x \leq a-1) \wedge (x \leq a)$

— si la valeur a est supprimée du domaine de x alors $(x \leq a-1) \vee \neg(x \leq a)$

Afin que l'encodage reste correct, certaines clauses doivent être ajoutées aux variables. En effet, décider $(x \leq a)$ devrait propager $(x \leq a+1)$. Il faut donc ajouter des clauses de domaine de la forme $(x \leq a) \Rightarrow (x \leq a+1) = \neg(x \leq a) \vee (x \leq a+1)$ au sein des variables.

Exemple 22

Soit la variable CP x avec $\texttt{dom}(x) = \{0, 1, 2, 3\}$.

Afin d'encoder en SAT cette variable avec l'encodage d'ordre, nous avons besoin de 4 variables propositionnelles :

— $(x \leq 0)$

— $(x \leq 1)$

— $(x \leq 2)$

— $(x \leq 3)$

ainsi que 3 clauses :

— $\neg(x \leq 0) \vee (x \leq 1)$

— $\neg(x \leq 1) \vee (x \leq 2)$

— $\neg(x \leq 2) \vee (x \leq 3)$

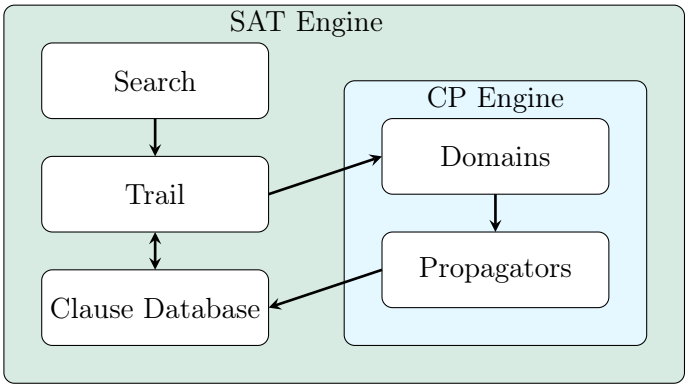


FIGURE 4.1 – Structure du système original de génération paresseuse de clauses

Forme originale La figure 4.1 montre le système original de la génération paresseuse de clauses. La recherche est contrôlée par le moteur SAT qui va d'abord appliquer la propagation unitaire lorsqu'une décision est faite. Au moment où le point fixe de la propagation unitaire est atteint, les modifications sont appliquées sur les domaines des variables CP. Ces changements vont déclencher les propagateurs associés dans le moteur de programmation par contraintes. Si l'utilisation d'un propagateur doit modifier le domaine d'une variable, cette modification n'est pas faite directement. À la place, une ou plusieurs clauses qui décrivent la modification vont être générées et envoyées au moteur SAT afin que celui-ci prenne en compte ces informations en réalisant un appel à la propagation unitaire. Cela va continuer jusqu'à un point fixe et une nouvelle décision sera choisie du côté SAT si aucun conflit n'est identifié.

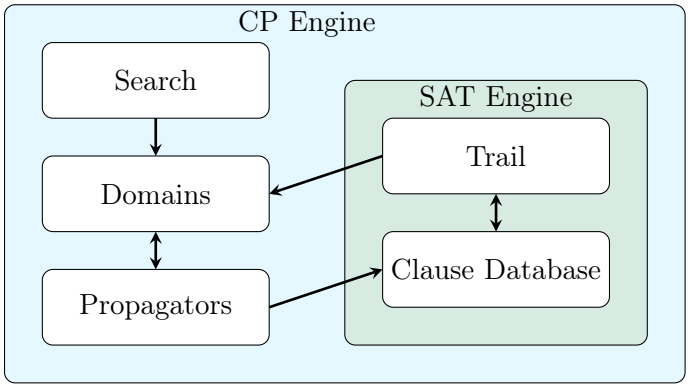


FIGURE 4.2 – Structure du système révisé de génération paresseuse de clauses

Forme révisée La forme révisée utilise un moteur SAT comme propagateur au sein d'un solveur CP. Ce choix permet plus de flexibilité. En effet, la recherche est contrôlée par le moteur CP qui est maintenant en position de maître (Figure 4.2). Cela implique que la recherche peut soit être réalisée de manière CP standard ou en tenant compte des informations du propagateur spécifique qui englobe le moteur SAT. De plus, une file de propagations est à nouveau utilisée et le moteur SAT devient un propagateur au sein du solveur CP. Ce propagateur spécial est ajouté en tête de la file de propagations lorsqu'un littéral est décidé ou que des clauses sont générées. De la même manière que la version originale, les propagateurs du moteur de programmation par contraintes continuent à produire des clauses quand un conflit ou une réduction de domaine a lieu.

Clairement, dans les deux configurations et particulièrement la forme révisée, nous pouvons voir que la génération paresseuse de clauses exploite les forces de la programmation par contraintes et du problème de satisfaction booléenne. De plus, la forme révisée apporte une flexibilité supplémentaire et supporte l'utilisation d'heuristiques des deux paradigmes. Cela permet d'obtenir en pratique des résultats convaincants.

4.4 Conclusion

Nous avons vu, au cours de ce chapitre, trois méthodes de l'état de l'art (les *nogoods* généralisés, les explications paresseuses et la génération paresseuse de clauses) qui permettent d'hybrider la programmation par contraintes avec le problème de satisfaction booléenne. Qu'elles soient basées sur une génération de *nogoods* qui s'apparentent à des clauses, sur une forme d'explication plus efficace ou sur un moteur SAT sous-jacent, les approches hybrides sont originales et performantes dans le cas général. En effet, elles tirent parti des forces de chaque paradigme afin de corriger les faiblesses de l'autre.

Les travaux présentés dans ce chapitre, en particulier les *nogoods* généralisés et les explications paresseuses, sont le point de départ utilisé pour nos contributions présentés dans les chapitres 6 et 7. En effet, dans le chapitre 6 nous proposons un moteur générique de raisonnement inspiré des *nogoods* généralisés et des explications paresseuses ainsi qu'une heuristique d'ordonnancement pour la réduction de la base de clauses qui prend en compte la provenance des littéraux (la variable CP à laquelle ils appartiennent). Dans le chapitre 7, nous étendons les fonctions de raisonnement proposées par Gent *et al.* (2010) à la contrainte globale *element*. Nous proposons

aussi une heuristique hybride de sélection de couples variable-valeur qui évalue la qualité des clauses produites pour le contexte de programmation par contraintes ainsi qu'une méthode de minimisation de clauses.

Deuxième partie
Contributions

Chapitre 5

Combinaison de *nogoods* extraits au redémarrage

Sommaire

Dans ce chapitre, nous nous intéressons à l'enregistrement de *nogoods*, instanciations partielles globalement incohérentes, pouvant être extraits systématiquement lors du redémarrage d'un algorithme complet (avec retour arrière) de résolution CSP. Nous proposons plusieurs mécanismes pour raisonner sur les *increasing-nogoods* qui correspondent à l'état exact d'un solveur avant un redémarrage. Cela nous permet d'accroître leur capacité de filtrage. La base de cette approche est une généralisation des *nld-nogoods* (présentés section 2.3) correspondant au concept introduit récemment d'*increasing-nogoods* (présentés section 2.4.2). Fait intéressant, certaines similitudes qui peuvent être observées entre les *increasing-nogoods* nous permettent de proposer de nouvelles manières originales de les combiner de façon dynamique afin d'améliorer la capacité de filtrage globale du système d'apprentissage. Nous proposons [2] plusieurs algorithmes portant sur des combinaisons de sous-ensembles de *nogoods* identifiés de manière dynamique. Les similarités entre les différents *increasing-nogoods* permettent un meilleur élagage de l'arbre de recherche, notamment grâce à l'exploitation d'équivalences entre décisions. Nous identifions également quelques pistes prometteuses permettant de renforcer l'efficacité du processus de détection.

5.1 Combinaison d'*increasing-nogoods*

Considérés comme des contraintes, il est tout à fait naturel que *nld-nogoods* et *increasing-nogoods* soient sollicités indépendamment pour des tâches de filtrage. Cependant, nous montrons qu'il est possible d'exploiter les similitudes qui existent (assez fréquemment) entre de tels *nogoods*. Les techniques existantes de traitement de *nogoods*, que ce soient les *nld-nogoods* ou les *increasing-nogoods*, ne tirent pas partie des autres informations disponibles, à savoir l'état des variables (c'est-à-dire, leur domaine) ainsi que les autres *nogoods* qui partagent de nombreux points communs. Plus précisément, en utilisant à bon escient ces informations, nous introduisons

2. Ces travaux ont été publié aux *Treizièmes journées Francophones de Programmation par Contraintes – JFPC'17* (Glorian *et al.* (2017a)) et y ont remporté le prix du meilleur article étudiant. Ils ont été, par la suite, publié à *the 23rd International Conference on Principles and Practice of Constraint Programming – CP'17* (Glorian *et al.* (2017b)).

dans cette section trois règles pour raisonner plus efficacement avec les *increasing-nogoods*. La première règle permet d'améliorer le pouvoir de filtrage des *increasing-nogoods* en continuant de les traiter indépendamment mais en les faisant toutefois interagir sur la base des domaines des variables qui les composent. Les deux autres règles regroupent les *increasing-nogoods* par sous-ensembles en fonction des décisions positives présentes au sein des *increasing-nogoods*, ou en fonction des décisions négatives présentes, c'est-à-dire celles qui sont comprises entre les deux bornes alpha et bêta.

5.1.1 Combinaison de décisions négatives

Nous détaillons, dans la suite, la première règle de combinaison qui continue à traiter les *increasing-nogoods* indépendamment. Le but de la combinaison de décisions négatives au sein d'un *IncNG* est de sécuriser le fait qu'une affectation ou une suppression ne puisse pas causer de conflit direct avec le reste de la base de *nogoods*, c'est-à-dire avant que l'*increasing-nogood* relatif à la combinaison n'entre en conflit.

Utiliser cette règle permet de déceler les conflits en amont et par conséquent d'augmenter le pouvoir de filtrage des *increasing-nogoods*. En effet, en vérifiant pour chaque variable x et chaque *increasing-nogood* Σ qu'il existe une valeur dans $\texttt{dom}(x)$ qui ne fait pas l'objet d'une décision négative pour x entre deux seuils de la séquence notés $\alpha(\Sigma)$ et $\beta(\Sigma)$, nous avons la garantie de ne rater aucune déduction grâce un simple raisonnement sur des décisions négatives surveillées.

Exemple 23

Soit l'*increasing-nogood* suivant :

$$\Sigma = \langle x_2 = 1, x_3 \neq 2, x_3 \neq 4, x_5 = 3 \rangle$$

Supposons qu'il soit dans son état initial, c'est-à-dire que $\alpha(\Sigma)$ et $\beta(\Sigma)$ soient des indices pour $x_2 = 1$ et $x_5 = 3$. Supposons aussi que les variables aient toutes le même domaine $\texttt{dom}(x_i) = \{1, 2, 3, 4\}$.

Si x_2 est affectée à la valeur 1, alors il est possible de supprimer les valeurs 2 et 4 du domaine de x_3 (voir la situation 2 de l'algorithme 2.3). Si le domaine de x_3 ne contenait plus que ces deux valeurs alors il y aurait un conflit. Ce conflit aurait pu être évité (plus précisément anticipé) si la valeur 1 avait été supprimée du domaine de x_2 lorsque le domaine de x_3 a été réduit à $\{2, 4\}$.

Afin de traduire cette règle vers un algorithme utilisable, nous devons définir certaines fonctions que nous utiliserons dans la conception de celui-ci.

Premièrement, nous introduisons une fonction $\texttt{diffValues}(\Sigma, x_i)$ qui renvoie pour un *increasing-nogood* Σ passé en paramètre, l'ensemble des valeurs présentes dans les décisions négatives de Σ impliquant x_i et situées entre $\alpha(\Sigma)$ et $\beta(\Sigma)$.

Exemple 24

Soit l'*increasing-nogood* suivant :

$$\Sigma = \langle \underbrace{x_2 = 1}_{\alpha}, x_3 \neq 2, x_1 \neq 1, x_3 \neq 4, \underbrace{x_5 = 3}_{\beta}, x_1 \neq 2 \rangle$$

avec $\texttt{dom}(x_i) = \{1, 2, 3, 4\}$.

Nous avons alors, pour les variables x_1, x_2 et x_3 :

$$\texttt{diffValues}(\Sigma, x_1) = \{1\}$$
$$\texttt{diffValues}(\Sigma, x_2) = \emptyset$$
$$\texttt{diffValues}(\Sigma, x_3) = \{2, 4\}$$

De la même façon, nous introduisons également une fonction $\texttt{diffVars}(\Sigma)$ qui renvoie l'ensemble des variables impliquées dans une décision négative de Σ située entre $\alpha(\Sigma)$ et $\beta(\Sigma)$.

Exemple 25

Soit l'*increasing-nogood* suivant :

$$\Sigma = \langle \underbrace{x_2 = 1}_{\alpha}, x_3 \neq 2, x_1 \neq 1, x_3 \neq 4, \underbrace{x_5 = 3}_{\beta}, x_1 \neq 2 \rangle$$

avec $\texttt{dom}(x_i) = \{1, 2, 3, 4\}$.

Nous avons alors :

$$\texttt{diffVars}(\Sigma) = \{x_1, x_3\}$$

Algorithme 5.1 : CheckNegativeDecisions(Σ)

```
Data : Let Σ be an increasing-nogood
1 foreach x ∈ diffVars(Σ) do
2     if dom(x) ⊆ diffValues(Σ, x) then
3         falsify α(Σ);
4         break;
```

L'algorithme 5.1 implémente cette méthode de raisonnement, c'est-à-dire effectue une inférence en réfutant la valeur impliquée dans $\alpha(\Sigma)$, chaque fois qu'un conflit peut être anticipé, comme indiqué ci-dessus. Même si les *increasing-nogoods* sont toujours examinés indépendamment (chacun leur tour), la capacité de filtrage de l'algorithme proposé dans Lee *et al.* (2016) (Algorithme 2.3) est clairement améliorée si cette procédure simple est systématiquement appelée.

Théorème 1

La complexité temporelle dans le pire cas de l'algorithme 5.1 est $\mathcal{O}(nd)$ où n est le nombre de variables et d est la taille du plus grand domaine. En effet, nous pouvons précalculer les ensembles $\texttt{diffVars}(\Sigma)$ et $\texttt{diffValues}(\Sigma, x)$ en balayant les décisions dans Σ dont la plus grande taille possible est $O(nd)$. Avec ces ensembles précalculés, l'exécution des lignes 1 à 2 est également en $O(nd)$.

Dans la section suivante, nous proposons de généraliser cette règle. Pour cela, nous présentons une nouvelle variation de ce principe par analyse de la base complète d'*increasing-nogoods*.

5.1.2 Combinaison par équivalence d'alpha

Nous étendons maintenant le principe présenté précédemment à des ensembles composés de plusieurs *increasing-nogoods*. En effet, en analysant différentes bases d'*increasing-nogoods*, nous nous sommes aperçus qu'un certain nombre d'*increasing-nogoods* étaient très similaires et formaient des groupes. Cela peut s'expliquer par le fait que, lors des redémarrages, selon l'heuristique choisie, le solveur se concentre sur un même sous-problème difficile lorsque celui est identifié. Cela peut se rencontrer lors de plusieurs descentes de l'arbre de recherche de suite et, par conséquent, faire les mêmes choix de variables de décisions à chacune de ces descentes. Dans le cadre de ces travaux nous avons utilisé une heuristique adaptative, *dom/wdeg*. Ce comportement est donc très commun car le but de cette heuristique est, lors des redémarrages, de choisir en priorité les variables appartenant à des sous-problèmes difficiles (voir section 1.3).

Afin d'étendre le principe présenté dans la section 5.1.1, il nous faut partitionner l'ensemble des *increasing-nogoods* selon les décisions indexées par α : deux *increasing-nogoods* Σ_i et Σ_j sont dans le même groupe si et seulement si $\alpha(\Sigma_i)$ correspond à la même décision que $\alpha(\Sigma_j)$. Bien entendu, il est donc nécessaire de mettre à jour la partition à chaque fois qu'un α est modifié (c'est-à-dire lors du filtrage et du retour arrière). Malgré cela, le raisonnement sur les groupes d'*increasing-nogoods* nous permet d'améliorer la capacité de filtrage des *increasing-nogoods*, et s'avère englober le cas précédent.

Exemple 26

Considérons les trois *increasing-nogoods* suivants :

$$\Sigma_0 \equiv \quad \ldots, x_6 \neq 2, \underbrace{\underline{x_2 = 1}}_{\alpha}, x_1 \neq 3, \underline{x_3 \neq 1}, \ldots$$

$$\Sigma_1 \equiv \quad \ldots, x_2 \neq 0, x_1 \neq 2, \underbrace{\underline{x_2 = 1}}_{\alpha}, \underline{x_3 \neq 0}, \ldots$$

$$\Sigma_2 \equiv \quad \ldots, \underbrace{\underline{x_2 = 1}}_{\alpha}, \underline{x_3 \neq 2}, x_6 \neq 1, x_8 \neq 3, \ldots$$

Et supposons que toutes les variables aient le même domaine $\{0, 1, 2, 3\}$.

Sur cet exemple, nous pouvons observer que $x_2 = 1$ est l'*alpha* commun à ce groupe de trois *increasing-nogoods*. En regardant les décisions négatives qui suivent ces trois occurrences d'*alpha* (la valeur de bêta n'est pas importante pour notre illustration), nous remarquons que nous pouvons collecter $\{0, 1, 2\}$ comme valeurs impliquées dans les décisions négatives surveillées pour x_3 (elles sont nécessairement placées avant les β de chaque *increasing-nogood* qui ne sont pas représentés ici). Cela signifie que si x_2 venait à être affecté à la valeur 1, alors la seule valeur restante dans le domaine de x_3 serait 3. Par contre, si à un moment donné, le domaine de x_3 ne contient plus la valeur 3, il faut absolument éviter que x_2 ne se voit attribuer la valeur 1. Si ce cas survient, et afin d'anticiper le conflit, tous les *increasing-nogoods* du groupe vont être désactivés (car forcés à être toujours vérifiés) après avoir supprimé la valeur 1 du domaine de x_2. L'algorithme 5.2 réalise ce filtrage.

L'algorithme 5.2 est une généralisation de l'algorithme 5.1. Le différence réside dans le fait que nous considérons des groupes d'*increasing-nogoods* au lieu de les considérer individuellement. Techniquement, nous considérons leur union (car les variables n'ont pas besoin d'apparaitre dans tous les *increasing-nogoods* du groupe) afin d'identifier l'ensemble de variables des décisions négatives dans les groupes à l'alpha commun (fonction `diffVars` adaptée aux groupes). Puis, comme précédemment, nous regardons les domaines de chacune des variables identifiées et les comparons avec l'ensemble des valeurs présentes au sein du groupe considéré. Si au moins une inclusion est trouvée, nous devons falsifier l'alpha commun afin d'anticiper le conflit.

Théorème 2

L'algorithme 5.2 a une complexité temporelle dans le pire cas en $\mathcal{O}(nd + g)$ où n est le nombre de variables, d est la taille du plus grand domaine et g est la somme de la taille des *increasing-nogoods* dans l'ensemble Σs. Nous avons :

$$g = \sum_{\Sigma \in \Sigma s} |\Sigma|$$

En effet, précalculer les ensembles

$$\bigcup_{\Sigma \in \Sigma s} \texttt{diffVars}(\Sigma)$$

et

$$\bigcup_{\Sigma \in \Sigma s} \texttt{diffValues}(\Sigma, x)$$

peut être effectué en $O(g)$ en analysant chaque décision dans les *increasing-nogoods* de Σs. Avec ces ensembles précalculés, l'exécution des lignes 1 et 2 se fait en $O(nd)$.

Nous allons voir, dans la section suivante, que nous pouvons traiter les *increasing-nogoods* dont la variable alpha est similaire, mais où les valeurs pointées sont différentes. Ainsi, nous pouvons envisager d'extraire des sous-groupes ayant, dans ce cas, des décisions négatives communes de ces ensembles.

Algorithme 5.2 : CheckNegativeDecisions(Σs)

```
Data : Let Σs be a set of increasing-nogoods with a common α
foreach x ∈ ⋃_{Σ∈Σs} diffVars(Σ) do
    if dom(x) ⊆ ⋃_{Σ∈Σs} diffValues(Σ, x) then
        falsify α(Σ) ;                    // Σ can be any increasing-nogood from Σs
        break;
```

5.1.3 Combinaison par équivalence de décisions négatives

Nous allons maintenant traiter, de la même manière que précédemment, les *increasing-nogoods* par ensembles. La différence demeure dans le fait que nous regroupons les *increasing-nogoods* qui possèdent un alpha de variable commune (la valeur importe peu pour créer le groupe) et qui surveillent une décision négative identique (même variable, même valeur) située entre les alphas et bêtas courants. L'idée est la suivante : si toutes les valeurs restantes dans le domaine courant de la variable commune des alphas figurent dans le groupe, alors la décision négative commune doit être vérifiée. Commençons par définir le concept de variable pivot.

Définition 41 (Variable pivot)

Nous appelons **pivot** une variable x telle que pour toute valeur $a \in \texttt{dom}(x)$, il existe un *increasing-nogood* Σ tel que $\alpha(\Sigma)$ est la décision positive $x = a$; dans ce cas, nous disons que Σ est un support de la variable pivot x pour a.

Fait intéressant, une fois une variable pivot x identifiée, il est possible d'inférer des décisions négatives partagées par tous les supports de x. C'est le principe de l'algorithme que nous présentons après une illustration.

Exemple 27

Considérons les trois *increasing-nogoods* suivants :

$$\Sigma_0 \equiv \quad \dots\ ,\ x_6 \neq 2,\ \underbrace{x_2 = 1}_{\alpha},\ x_1 \neq 0,\ \underline{x_3 \neq 1},\ \dots$$

$$\Sigma_1 \equiv \quad \dots\ , x_7 \neq 0,\ x_1 \neq 2,\ \underbrace{x_2 = 0}_{\alpha},\ \underline{x_3 \neq 1},\ \dots$$

$$\Sigma_2 \equiv \quad \dots\ , \underbrace{x_2 = 2}_{\alpha},\ \underline{x_3 \neq 1},\ x_6 \neq 1,\ x_8 \neq 2,\ \dots$$

Et supposons que toutes les variables aient le même domaine $\{0, 1, 2\}$.

Sur cet exemple, nous pouvons voir que la variable x_2 est un pivot de cet ensemble d'*increasing-nogoods* car toutes ses valeurs possibles (0, 1 et 2) de son domaine courant (ici, initial en l'occurrence) sont impliquées dans les α de différents *increasing-nogoods*.

Comme $x_3 \neq 1$ est une décision négative surveillée dans les trois *increasing-nogoods*, nous pouvons en déduire que x_3 doit toujours être différent de 1. Autrement dit, peu importe la valeur que prend la variable x_2 : x_3 sera toujours différent de 1 à ce stade de la recherche.

En considérant cela, nous pouvons minimiser indirectement les *increasing-nogoods* selon les scénarios de recherche (en fonction de l'évolution du domaine des variables pivots), voire directement si toutes les valeurs du domaine initial de l'alpha d'un groupe apparaissent.

Notation 3

Soit $\delta \equiv (x = v)$, la $i^{\text{ème}}$ décision au sein d'un *increasing-nogood* Σ :

- $var(\delta)$ désigne la variable x ;
- $val(\delta)$ désigne la valeur v ;
- $ind(\delta)$ désigne l'indice i (c'est-à-dire la position de δ dans la séquence).

Algorithme 5.3 : CheckPivots(Σs)

```
Data : Let Σs be the full set of increasing-nogoods
1 foreach x ∈ {var(α(Σ)) | Σ ∈ Σs} do
2     if dom(x) ⊆ {val(α(Σ)) | Σ ∈ Σs ∧ var(α(Σ)) = x} then
3         foreach δ ∈ ⋂_{Σ∈Σs var(α(Σ))=x} {δ_i ∈ Σ | ind(α(Σ)) < i < ind(β(Σ))} do
4             satisfy δ;
```

L'algorithme 5.3 implémente l'utilisation de variables pivots pour faire des inférences supplémentaires. La ligne 1 itère uniquement sur les variables impliquées comme alpha dans la base d'*increasing-nogoods*. La ligne 2 teste si la variable x considérée est effectivement une variable pivot. Si une variable pivot est identifiée alors la ligne 3 itère sur les décisions négatives partagées par tous les supports de x dans la base d'*increasing-nogoods*. Chacune de ces décisions doit être forcée à être satisfaite.

Il est à noter qu'une optimisation possible consiste uniquement à vérifier qu'une décision est partagée par certains sous-ensembles de supports de x, les sous-ensembles avec exactement un support de x pour chaque valeur dans le scénario courant, au lieu de prendre l'ensemble complet.

Théorème 3

L'algorithme 5.3 a une complexité temporelle dans le pire cas en $\mathcal{O}(n^2dp)$ où n est le nombre de variables, d est la taille du plus grand domaine et p est le nombre *increasing-nogoods* dans la base.

5.2 Expérimentations

Nous avons effectué une expérimentation sur un ordinateur *Intel Xeon X5550* cadencé à 2,67 GHz et équipé de 8 Go de RAM, avec un *timeout* réglé à 15 minutes (900 secondes). Notre base de tests initiale était composée de toutes les instances utilisées lors des compétitions de solveur *XCSP 2.1* des années 2006 et 2008. Nous avons écarté de la série d'instances celles trop faciles à résoudre (moins d'une seconde) ou trop difficiles à résoudre (plus de 900 secondes) lorsque l'algorithme MAC sans enregistrement de *nogoods* est utilisé. Cela nous a donné un ensemble composé exactement de 3744 instances.

Pour nos expériences, nous avons utilisé le solveur *rclCSP* introduit dans Grégoire *et al.* (2011). L'heuristique d'ordonnancement des variables est *dom/wdeg* et la politique de redémarrage correspond à une série géométrique de raison 1,1 et de premier terme 10. Étant donnée la complexité de l'algorithme 5.3 et parce que l'algorithme 5.1 est généralisé par l'algorithme 5.2, nous avons choisi de mener nos expériences uniquement avec l'algorithme 5.2 (la combinaison d'*increasing-nogoods* par équivalence d'alpha). Pour notre comparaison, nous avons testé les trois méthodes suivantes :

— NRR (enregistrement de *nld-nogoods* à partir de redémarrages comme proposé dans Lecoutre *et al.* (2007b), présentés section 2.3) ;

— IncNG (gestion des *increasing-nogoods* comme proposé dans Lee *et al.* (2016), présentés

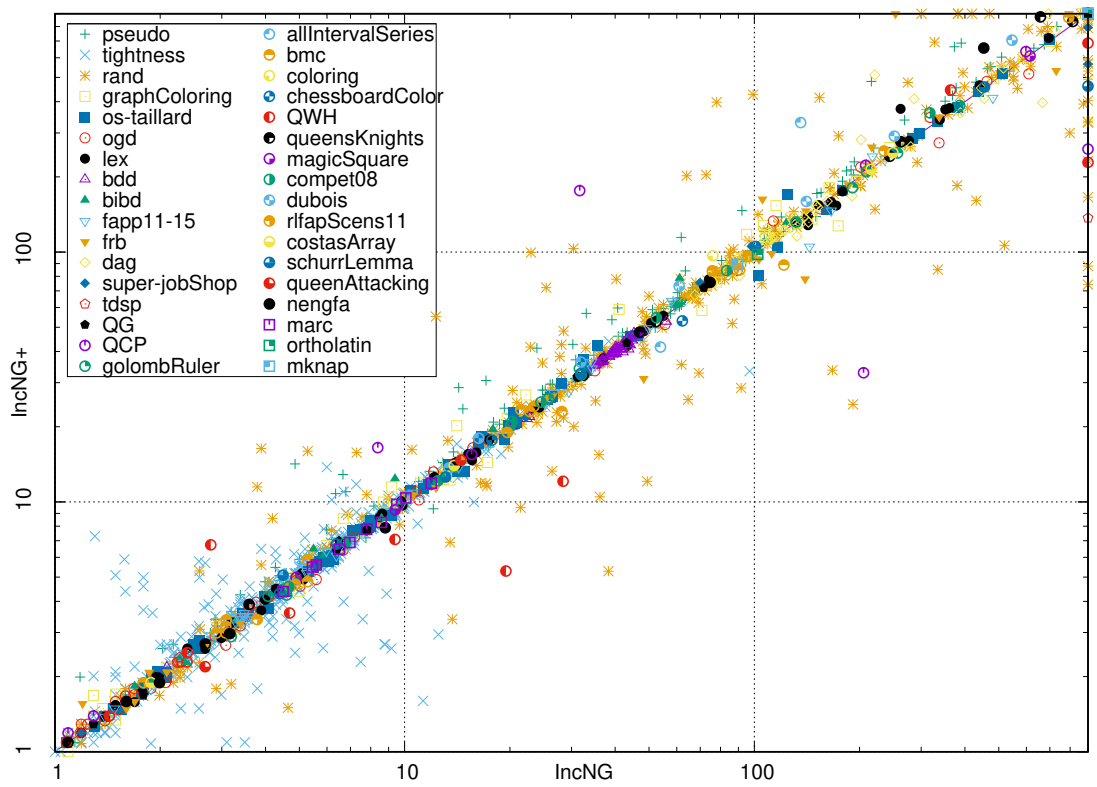


FIGURE 5.1 – Comparaison par paires (temps de calcul exprimé en secondes) des méthodes IncNG et IncNG+. Les résultats obtenus sur les 3,744 instances utilisées. Le *timeout* pour résoudre une instance est défini à 900 secondes.

Series	#inst	NRR		IncNG		IncNG+	
		#sols	PAR10	#sols	PAR10	#sols	PAR10
costasArray	11	9	1745	9	1686	9	1686
fapp11-15	55	42	2162	42	2157	42	2155
frb45-21	10	9	1138	**10**	**232**	**10**	282
nengfa	10	9	970	9	954	9	974
ogdVg	65	35	4197	39	3637	**40**	**3511**
ortholatin	9	**4**	**5045**	3	6011	3	6010
os-taillard-7	30	**19**	**3353**	18	3679	18	3680
QCP-20	15	6	5437	7	4870	**8**	**4289**
QWH-20	10	10	79	10	47	10	48
QWH-25	10	1	8187	0	9000	**2**	**7291**
rand-2-50-23-fcd	50	5	8146	12	6953	**15**	**6436**
rand-2-50-23	50	4	8335	**9**	7476	**9**	**7462**
rand-3-24-24	50	14	6596	16	6248	**18**	**5900**
rlfapScens11	12	11	852	11	844	11	852
super-jobShop	46	34	2358	33	2547	**35**	**2186**

Table 5.1 – Résultats sur 15 séries d'instances (*timeout* défini à 900 secondes).

section 2.4.2) ;

— IncNG+ (combinant les *increasing-nogoods* simples de la méthode IncNG et notre approche présentée dans la section 5.1.2).

La figure 5.1 montre l'ensemble des résultats obtenus. Le nuage de points montre que notre approche (IncNG+) a généralement un léger surcoût (quand elle s'avère peu efficace en terme d'inférences), et rend la recherche un peu plus robuste. En effet, en regardant les points sur la verticale à droite de la figure qui correspondent aux instances non résolues par la méthode IncNG seule, nous pouvons nous apercevoir qu'il y en a plus que sur la droite horizontale du haut de la figure. De plus, les instances résolues par notre méthode IncNG+, contrairement à la méthode IncNG, ne sont pas uniquement des instances aléatoires.

Le tableau 5.1 présente une comparaison détaillée entre les trois méthodes testées (NRR, IncNG et IncNG+) sur 15 séries intéressantes d'instances. Le tableau contient les informations suivantes :

— *Series*, le nom de la série ;

— *#inst*, le nombre d'instances au sein de la série ;

et pour les trois approches testées :

— *#sols*, le nombre d'instances résolues ;

— *PAR10*, le score qui correspond à la moyenne des durées d'exécution tout en prenant en compte 10 fois le *timeout* (900) pour les instances non résolues.

En général, notre approche (IncNG+) résout au moins autant d'instances que IncNG et parfois plus (voir, par exemple, *super-jobShop*). Lorsque IncNG et IncNG+ résolvent le même nombre d'instances, nous remarquons généralement une légère perte pour notre approche en

Instance	IncNG				IncNG+			
	cpu	#nld	#del	#fail	cpu	#nld	#del (dont $\alpha_{\Leftrightarrow}$)	#fail
qcp-20-187-0	**208,05**	671	38	1	222,61	671	38	1
qcp-20-187-1	**31,75**	569	2287	107	176,41	806	11033(708)	289
qcp-20-187-2	timeout	1007	54728	6980	timeout	1019	57232(2088)	6665
qcp-20-187-3	timeout	844	53057	5043	timeout	770	33255(2027)	1881
qcp-20-187-4	timeout	862	47013	3692	timeout	818	39867(2548)	1905
qcp-20-187-5	**0,7**	121	15	0	**0,7**	121	15	0
qcp-20-187-6	**597,85**	835	5177	573	634,87	835	5177(1)	573
qcp-20-187-7	timeout	847	1582	90	**258,52**	679	1267(17)	40
qcp-20-187-8	205,33	667	5286	350	**32,94**	462	807(116)	6
qcp-20-187-9	15,62	396	72	2	**15,54**	396	72	2
qcp-20-187-10	timeout	892	43964	3780	timeout	921	21082(667)	1362
qcp-20-187-11	0,3	62	38	8	**0,27**	63	44(22)	3
qcp-20-187-12	timeout	926	34894	6362	timeout	880	21178(540)	1524
qcp-20-187-13	timeout	946	191009	33687	timeout	863	102101(2906)	8127
qcp-20-187-14	timeout	930	30693	1819	timeout	983	137251(13389)	6261

TABLE 5.2 – Résultats expérimentaux détaillés sur la famille *qcp-20* composée de 15 instances.

raison de la gestion des partitions des *increasing-nogoods* (par exemple, voir *frb45-21*). Mais, il est intéressant de noter qu'il existe des séries dans lesquelles ce temps de traitement est compensé par un meilleur élagage de l'arbre de recherche, avec pour résultat un gain de temps substantiel (*rand-2-50-23*). De plus, nous observons que, d'une manière générale, plus l'instance est difficile, plus notre approche est compétitive, comme l'illustrent les séries *QWH-20* et *QWH-25* qui admettent des tailles et des complexités croissantes.

Le tableau 5.2 reporte une comparaison entre la méthode IncNG et IncNG+ sur la famille de problème *QCP-20*. Dans ce tableau, nous reportons le nombre total de *nld-nogoods* apparaissant dans les *increasing-nogoods* (*#nld*), le nombre total de suppressions induites par les *increasing-nogoods* (*#del*), et entre parenthèses apparaît le nombre de suppressions dues à notre méthode uniquement (ce nombre est inclus dans le total). Ce tableau reporte aussi le nombre de conflits imputés par la révision des *increasing-nogoods* (*#fail*), ainsi que le temps de résolution, exprimé en secondes (*cpu*). Sur ce tableau, nous pouvons remarquer que sur cette famille d'instances l'ajout de notre algorithme permet de résoudre une instance de plus. Nous pouvons aussi remarquer que de nombreuses suppressions de valeurs sont induites par l'ajout de notre méthode. Cette série est intéressante car tous les comportements y sont représentés. Notamment, lorsque notre méthode n'infère pas de suppressions (instances *qcp-20-187-0*, *qcp-20-187-5*, *qcp-20-187-6* et *qcp-20-187-9*), nous pouvons distinguer un léger surcoût sur les instances *qcp-20-187-0* et *qcp-20-187-6*. Nous pouvons aussi distinguer des instances où la méthode est très efficace (instances *qcp-20-187-7* et *qcp-20-187-8*). L'instance *qcp-20-187-7* qui n'était pas résolue avec la méthode IncNG seule, l'est maintenant et ce, assez rapidement (258, 52 secondes). Finalement, nous observons que pour la majorité des problèmes de cette famille, le nombre de conflits obtenus dans les *increasing-nogoods* est plus faible. Cela peut être expliqué par le fait que les conflits sont anticipés et donc que l'arbre de recherche est élagué et donc plus petit.

Dans un second temps, afin de montrer la corrélation entre le nombre et la taille des *nogoods*, nous avons effectué une expérimentation supplémentaire avec différentes stratégies de redémar-

rage. Celles-ci sont, soit basées sur la suite de Luby *et al.* (1993) (1,1,2,1,1,2,4,...), soit sur une suite géométrique. Pour ces expérimentations, nous avons utilisé les politiques suivantes :

— P10 - suite géométrique de premier terme 10 et de raison 1,1 ;

— P50 - suite géométrique de premier terme 50 et de raison 1,5 ;

— L100 - suite de Luby dont les termes sont multipliés par 100.

Les tableaux 5.3, 5.4 et 5.5 permettent d'évaluer l'impact de la stratégie de redémarrage sur les performances des méthodes considérées. Pour cela nous avons comparé les trois méthodes, NRR, IncNG et IncNG+. Cette table reporte, pour l'ensemble des problèmes considérés dans cette expérience, le nom de la famille considérée (*Family name*) ainsi que le nombre d'instances de celle-ci (*#inst*). Pour chacune des méthodes, nous reportons la stratégie de redémarrage considérée ainsi que le nombre d'instances résolues (*#sols*) et le temps moyen de résolution (*cpu*). Comme nous pouvons voir sur ce tableau, une stratégie de redémarrage moins agressive est souhaitable lorsque nous considérons NRR ou IncNG. Dans le cas de notre méthode IncNG+, une stratégie plus agressive permet d'utiliser pleinement la puissance des *increasing-nogoods*. Cela peut en partie s'expliquer par le fait d'une base plus importante de *nogoods* permet d'induire un nombre plus important de suppressions.

5.3 Discussion

Lors de nos expérimentations nous avons remarqué que plus nous utilisons une politique de redémarrage agressive plus les combinaisons sont efficaces d'un point de vue suppressions mais la complexité, dépendante de la taille de la base, augmente. Nous avons donc lancé quelques simulations afin de mesurer l'impact en temps de calcul sur une politique relativement agressive, à savoir *luby* $\times$ *10*. Nous avons empêché les algorithmes relatifs aux *increasing-nogoods* ainsi qu'à IncNG+ de faire le moindre changement sur le réseau mais ils continuent à être appelés normalement. Le résultat est le suivant sur l'instance *scen11-f5* le temps de résolution est de 120 secondes dont 85 de calcul d'équivalence d'alpha et 2,5 secondes de gestion des *increasing-nogoods*. Cette même instance est résolue en 17,45 secondes (dont 3,9 secondes de combinaisons) si nous activons les suppressions.

À la vue de ces résultats encourageants, nous avons décidé de proposer des améliorations aux algorithmes présentés précédemment grâce à diverses méthodes. Dans cette section nous allons présenter une première méthode, à savoir un système de sentinelles.

Les sentinelles peuvent s'apparenter à un système de *1-watch* appliqué à des domaines relatifs à des *increasing-nogoods*. Nous allons pouvoir les utiliser dans deux des trois méthodes de combinaison présentées : la combinaison de décisions négatives (présentée section 5.1.1) ainsi que la combinaison par équivalence de décisions négatives (présentée section 5.1.3).

Dans le cas de la combinaison de décisions négatives, le principe est le suivant : dans les décisions négatives contenues entre alpha et bêta dans un *IncNG*, c'est-à-dire là où nous espérons avoir des valeurs supprimées, nous pouvons ajouter un certain nombre de sentinelles. Une sentinelle marque une valeur que nous ne regardons pas normalement dans le domaine de la variable associée. Tant que celle-ci existe, nos suppressions ne créeront pas de conflits. De plus, grâce à ce système de sûreté de domaine nous n'avons plus à nous soucier des valeurs négatives lors du filtrage d'un *IncNG* dans le cas où la décision pointée par alpha n'est pas affectée. Nous pouvons, par conséquent, supprimer les lignes 13 à 21 de l'algorithme 2.4.

Nous proposons de mettre en place un système de sentinelles qui, associé à chaque *IncNG*, observe si une valeur différente de celle apparaissant dans le *IncNG* existe. Tant qu'il reste un élément autre que ceux apparaissant dans le *IncNG*, les suppressions pourront se faire sans conflit si la décision pointée par alpha venait à être satisfaite. Si la sentinelle venait à être supprimée sans possibilité d'en trouver une nouvelle, il faudrait alors supprimer la valeur pointée par α. Cela peut se faire en utilisant soit des *1-watch*, soit de manière paresseuse en vérifiant la taille des domaines concernés par rapport aux nombres de valeurs qui apparaissent négativement dans le *IncNG*, cette dernière étant beaucoup moins précise.

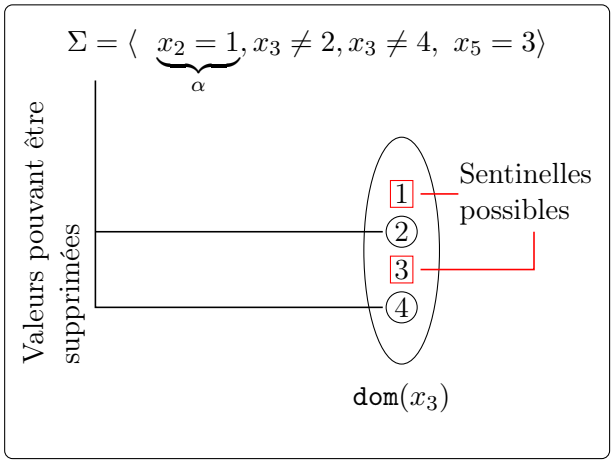


FIGURE 5.2 – Exemple de fonctionnement des sentinelles

Nous pouvons voir, dans la figure 5.2, qu'il est possible de choisir deux sentinelles, les valeurs 1 et 3 du domaine de x_3. Tant qu'au moins une de ces valeurs est dans le domaine, nous n'avons pas besoin de falsifier la décision pointée par alpha.

Dans le cas de la combinaison par équivalence de décisions négatives, nous pouvons utiliser une sentinelle afin de détecter lorsque la taille du domaine de l'alpha potentiellement pivot (même variable, valeur différente) devient équivalent au nombre d'*increasing-nogoods* dans le sous-groupe considéré.

Toujours dans le but d'étendre les interactions entre *nogoods* nous proposons une méthode alternative, voire complémentaire, aux combinaisons proposées au cours de ce chapitre. Nous considérons toujours des groupes avec décisions négatives équivalentes, mais avec des alphas quelconques (non-pivots). Grâce à ces groupes, nous pouvons générer des *nld-nogoods* réduits composés des prémices, c'est-à-dire les décisions positives qui précèdent l'alpha courant, ainsi que des alphas eux-mêmes. Cette méthode peut se généraliser aux combinaisons par équivalence de décisions négatives ainsi qu'aux combinaisons par équivalence d'alpha. Cela permettrait d'avoir des *nld-nogoods* qui encodent les combinaisons possibles. Il suffirait alors de générer les groupes et les *nld-nogoods* encodant les anticipations de conflits à chaque redémarrage afin d'éviter de calculer les groupes à la volée lors de la recherche avec nos algorithmes. Cela permettrait de conserver le pouvoir d'élagage des méthodes tout en réduisant le surcoût induit.

5.4 Conclusion

Afin d'augmenter significativement le pouvoir de filtrage des *increasing-nogoods*, nous avons mis en évidence qu'il était possible de combiner les informations entre les *increasing-nogoods* et

la structure du problème (notamment les domaines des variables). Plus précisément, nous avons d'abord proposé une approche qui, étant donné un *IncNG*, prend en considération les informations relatives aux domaines des variables afin d'identifier de nouvelles valeurs à supprimer. Ensuite, nous avons montré qu'il est possible d'étendre ce processus à la base entière d'*increasing-nogoods* de deux manières, soit en fonction de leurs décisions positives, soit en fonction de leurs décisions négatives.

Nous avons montré expérimentalement l'intérêt pratique de la deuxième règle (qui généralise la première): lorsqu'elle est efficace, c'est-à-dire que des inférences peuvent être effectuées en raisonnant sur des groupes d'*increasing-nogoods*, elle permet de gagner du temps de calcul, et lorsqu'elle n'est pas efficace, le surcoût de l'utilisation de la méthode est plutôt limité.

Cependant, l'utilisation de ces algorithmes a deux inconvénients. Le premier est que le fait de détecter les conflits en amont a un impact sur l'heuristique de choix de variables. En effet, puisque l'heuristique utilisée est ajustée en fonction des conflits, les éviter ne permet pas d'augmenter la pondération des contraintes, et donc de choisir la meilleure variable. L'autre inconvénient est la complexité des algorithmes présentés, qui dépend de la taille de la base d'*increasing-nogoods*. Comme énoncé dans la section 5.3, cette situation peut être améliorée en considérant des mécanismes de mise à jour basés sur l'utilisation d'un système de sentinelles ou d'ensembles persistants *backtrack-ready*. Nous pensons que certaines améliorations sont encore possibles, notamment en ce qui concerne l'exploitation des variables pivot.

Enfin, travailler dans ce contexte, et plus particulièrement avec les *increasing-nogoods* nous, a permis de réaliser que, dans le cas général, les solveurs de programmation par contraintes ne sont pas optimaux afin de gérer des *nogoods*. En effet, identifier uniquement chaque valeur de chaque variable et leur associer un signe ressemble d'avantage au problème de satisfaction booléenne. De plus, les clauses SAT étant l'équivalent des *nogoods* CP, nous nous sommes intéressés à l'intersection des deux domaines. C'est pour ces raisons que nous avons choisi de créer un outil de raisonnement, NACRE, afin de pallier ce problème et permettre d'expérimenter autour des clauses et des *nogoods* dans un contexte hybride. Cet outil, présenté dans le chapitre suivant, est inspiré des explications paresseuses et a pour vocation de les améliorer.

		NRR						IncNG						IncNG+					
		L100		P50		P10		L100		P50		P10		L100		P50		P10	
Family name	#inst	#sols	cpu	#sols	cpu	#sols	cpu	#sols	cpu	#sol	cpu	#sols	cpu	#sols	cpu	#sols	cpu	#sols	cpu
allIntervalSeries	25	10	15,77	**12**	**45,12**	11	15,48	10	13,45	11	7,31	11	19,39	10	13,93	11	7,34	11	35,81
bddLarge	35	35	60,3	35	52,26	35	128,88	**35**	**30,49**	35	30,54	35	33,15	35	30,56	35	30,74	35	33,16
bddSmall	35	35	109,52	35	43,35	35	113,69	35	25,4	35	25,2	35	25,65	35	25,3	**35**	**24,99**	35	25,48
bibd10-11	6	3	3,88	3	232,16	1	0,21	2	0,85	**4**	**175,44**	2	2,43	2	1,27	4	178,82	2	2,31
bibd12-13	7	**5**	10,09	3	1,78	4	1,97	4	4,87	4	10,63	3	5,29	4	4,39	4	11,36	3	5,12
bibd8	7	3	166,18	3	95,17	3	58,05	3	88,95	3	61,82	**3**	41,97	3	180,02	3	63,5	3	44,52
bibd9	10	10	1,46	10	2,47	10	4,53	10	1,41	10	2,41	8	4,3	**10**	**1,37**	10	2,2	8	4,31
bibdVariousK	29	23	7,22	23	60,36	22	28,68	**25**	**31,27**	21	4,44	23	6,47	24	12,78	22	42,04	23	7,53
bmc	24	24	8,19	24	14,32	24	30,47	24	7,99	24	15,11	24	22,57	**24**	**7,94**	24	16,06	24	20,82
chessboardColoration	20	15	6,76	15	7,82	15	21,81	15	3,66	15	5,67	15	6,4	**15**	**3,65**	15	5,67	15	5,76
coloring	22	21	0,58	22	4,37	22	8,51	22	9,44	**22**	**3,48**	22	3,89	22	16,64	22	4,28	22	4,85
compet08	16	9	85,22	9	107,12	9	132,69	8	70,17	**10**	**151,98**	9	81,72	9	157,09	10	157,34	9	87,2
costasArray	11	7	52,19	9	78,21	9	133,82	8	104,29	8	71,13	**9**	**62,42**	8	96,51	8	68,77	9	64,41
dag-half	25	1	898,31	6	631,48	0	NaN	17	412,64	16	370,99	17	385,25	**18**	**408,44**	16	406,4	17	373,83
dag-rand	25	24	448,24	25	163,81	25	282,27	25	136,01	25	132,93	25	131,57	25	134,59	25	132,27	**25**	**129,97**
dubois	13	0	NaN	6	275,72	4	261,5	4	246,52	**6**	**155,64**	6	174,93	3	319,69	6	166,99	6	214,52
fapp11-15	55	41	80,75	41	41,49	42	46,44	43	62,34	42	52,14	42	128,43	**43**	**61,48**	42	53,05	42	37,93
frb30-15	10	10	0,91	**10**	**0,68**	10	1,02	10	0,87	10	0,78	10	0,84	10	0,84	10	0,83	10	0,87
frb35-17	10	10	5,22	10	2,75	10	3,13	10	1,95	**10**	**1,81**	10	1,97	10	2,24	10	1,91	10	2,05
frb40-19	10	10	210,05	10	31,11	10	39,64	10	33,4	10	29,61	10	29,57	10	33,87	**10**	**27,25**	10	29,02
frb45-21	10	1	398,13	10	248,76	9	265,95	9	222,56	9	229,33	**10**	**233,17**	9	243,82	9	251,02	10	283,14
frb50-23	10	0	NaN	**2**	**457,53**	1	456,36	0	NaN	1	844,67	1	794,87	1	454,4	1	850,63	1	858,43
golombRulerArity3	14	9	57,76	11	131,92	10	61,76	10	50,99	10	51,37	10	51,36	10	56,17	**11**	**123,79**	10	49,83
golombRulerArity4	14	8	78,86	**10**	**55,28**	10	79,38	10	62,02	10	58,74	10	59,17	10	62,57	10	58,96	10	59,15
graphColoring	458	185	19,15	196	25,86	196	30,84	196	27,81	**198**	**32,11**	197	27,81	195	30,54	197	28,66	197	28,68
lexHerald	47	39	33,22	41	39,32	41	41,51	**44**	**55,01**	41	66,24	43	59,36	44	69,71	40	33,47	43	63,28
lexVg	63	62	50,3	63	32,6	63	44,45	63	30,74	**63**	**27,55**	63	30,74	63	29,04	63	28,38	63	30,58

TABLE 5.3 – Résultats expérimentaux par familles des trois méthodes : NRR, IncNG et IncNG+ sur différentes politiques de redémarrage - Partie 1.

		NRR						IncNG						IncNG+					
		L100		P50		P10		L100		P50		P10		L100		P50		P10	
Family name	#inst	#sols	cpu	#sols	cpu	#sols	cpu	#sols	cpu	#sol	cpu	#sols	cpu	#sols	cpu	#sols	cpu	#sols	cpu
magicSquare	18	5	3,01	7	114,26	6	6,51	6	103,33	6	59,44	7	89,88	6	105,37	6	59,93	**7**	**89,02**
marc	10	**10**	**8,1**	10	8,11	10	8,15	10	8,5	10	8,23	10	8,35	10	8,41	10	8,32	10	8,22
mknap	6	4	78,03	4	39,8	4	91,26	4	23,06	**4**	**22,67**	4	23,2	4	22,72	4	22,74	4	22,91
nengfa	10	9	123,78	8	59,26	9	79,37	9	85,85	9	89,78	**9**	**61,02**	9	89,71	9	79,23	9	83,71
ogdHerald	50	**50**	**2,4**	50	2,85	50	4,61	50	2,56	50	2,58	50	3,07	50	2,48	50	2,6	50	3,08
ogdPuzzle	22	22	0,76	22	0,76	22	0,78	22	0,84	22	0,83	22	0,71	22	0,79	**22**	**0,7**	22	0,75
ogdVg	65	34	73,9	39	93,04	35	81,07	**40**	**74,33**	39	55,27	39	63,24	40	77,1	40	81,11	40	81,09
ortholatin	9	3	31,98	3	15,46	**4**	**102,44**	3	54,38	3	13,77	3	34,57	3	54,27	3	13,7	3	33,37
os-taillard-10	30	20	33,42	17	42,09	20	31,87	20	32,34	18	65,09	**20**	**30,4**	20	32,69	18	64,08	20	30,68
os-taillard-15	30	21	7,49	20	7,22	21	9,39	21	6,94	20	6,68	21	8,84	21	7,01	**20**	**6,86**	21	8,86
os-taillard-20	30	22	24,23	21	26,03	22	29,06	22	22,18	21	24,66	22	28,69	**22**	**20,47**	21	24	22	28,65
os-taillard-4	30	30	0,78	30	0,67	30	0,79	30	0,59	30	0,69	30	4,07	30	0,69	**30**	**0,58**	30	0,7
os-taillard-5	30	28	49,5	29	39,24	29	47,16	29	37,04	28	28,33	**30**	**27,97**	30	57,02	29	57,61	30	28,44
os-taillard-7	30	16	72,55	17	72,95	**19**	**85,25**	16	65,69	17	54,9	18	133,96	17	73,57	17	55,04	18	134,68
pseudoGLB	384	132	310,84	187	135,54	152	296,18	**198**	**73,22**	190	66,81	186	67,01	189	72,04	189	64,38	186	77,17
pseudo	486	257	36,88	268	29,1	258	35,12	**281**	**36,13**	272	30,34	274	26,8	278	32,09	272	31,46	274	30,61
QCP-15	15	15	9,2	15	2,85	15	4,79	15	2,04	15	2,57	**15**	**1,77**	15	2,35	15	2,61	15	2,31
QCP-20	15	5	137,33	7	49,31	6	93,76	8	208,02	7	133,62	7	151,93	6	55,57	7	164,33	**8**	**168,18**
QG3	7	5	2,12	5	1,71	5	2,45	5	1,46	**5**	**1,15**	5	1,7	5	1,6	5	1,33	5	1,67
QG4	7	5	2,78	5	2	5	2,88	5	1,95	**5**	**1,53**	5	2,36	5	1,58	5	1,54	5	2,31
QG5	7	6	75,46	6	15,97	6	22,21	6	13,24	6	12,6	**6**	**12,52**	6	13,96	6	12,69	6	12,62
QG6	7	6	6,97	6	3,28	6	6,04	6	2,06	6	2,12	6	1,12	6	2,09	6	1,97	**6**	**1,1**
QG7	7	4	15,62	5	7,37	5	11,91	**5**	**4,7**	5	53,16	5	11,92	5	4,76	5	54,03	5	12,2
queenAttacking	10	5	25,4	5	4,11	5	2,42	5	2,63	5	50,82	5	3,99	**5**	**2,37**	5	50,75	5	3,74
queensKnights	18	16	70,54	17	99,89	17	91,62	16	57,53	**18**	**105,63**	18	116,45	16	58,46	18	105,68	18	130,59
QWH-20	10	9	50,02	10	101,23	10	80,04	10	88,01	**10**	**40,99**	10	48,26	9	22,38	10	68,25	10	49,01
QWH-25	10	0	NaN	1	590,09	1	875,89	1	794,82	**3**	**310,66**	0	NaN	1	838,62	2	265,2	2	458,58

Table 5.4 – Résultats expérimentaux par familles des trois méthodes : NRR, IncNG et IncNG+ sur différentes politiques de redémarrage - Partie 2.

		NRR						IncNG						IncNG+					
		L100		P50		P10		L100		P50		P10		L100		P50		P10	
Family name	#inst	#sols	cpu	#sols	cpu	#sols	cpu	#sols	cpu	#sol	cpu	#sols	cpu	#sols	cpu	#sols	cpu	#sols	cpu
rand-10-20-10	20	20	2,78	20	2,81	20	2,97	**20**	**2,64**	20	2,84	20	2,94	20	2,66	20	2,82	20	2,89
rand-2-23	10	10	103,09	10	33,17	10	38,01	10	29,65	**10**	**26,72**	10	32,53	10	36,07	10	29,35	10	28,17
rand-2-24	10	10	405,85	10	82,45	10	79,37	10	81,45	10	69,53	**10**	**67,9**	10	98,83	10	71,09	10	71,03
rand-2-25	10	3	515,25	10	155,91	10	173,81	10	180,25	10	141,47	10	149,84	10	280,6	**10**	**125,12**	10	138,37
rand-2-26	10	1	55,54	10	372,33	10	397,49	10	401,18	10	339,67	10	346,05	5	161,03	**10**	**320,91**	10	392,58
rand-2-27	10	0	NaN	3	357,2	4	275,15	4	435,21	4	295,76	**4**	**236,98**	3	295,63	4	459,59	4	341,94
rand-2-30-15-fcd	50	50	0,83	50	0,72	50	0,88	50	0,74	**50**	**0,7**	50	0,77	50	0,75	50	0,79	50	0,8
rand-2-30-15	50	50	1,13	50	1	50	1,06	**50**	**0,94**	50	0,98	50	1	50	1,04	50	0,99	50	1
rand-2-40-19-fcd	50	48	194,81	50	26,88	50	41,12	50	29,1	**50**	**25,52**	50	29,75	50	32,4	50	31,95	50	29,92
rand-2-40-19	50	40	296,87	50	66,82	50	81,37	50	63,86	50	61,55	**50**	**61,36**	50	71,01	50	64,06	50	62,17
rand-2-50-23-fcd	50	0	NaN	14	505,07	5	462,74	12	524,45	15	457,11	12	470,7	9	381,09	15	522,33	**15**	**456,83**
rand-2-50-23	50	0	NaN	12	514,2	4	685,69	12	473,67	12	597,81	9	539,14	8	519,13	**13**	**568,28**	9	460,74
rand-3-20-20-fcd	50	50	56,15	50	28,07	50	34,29	50	28,55	**50**	**22,67**	50	26,77	50	30,42	50	23,41	50	24,69
rand-3-20-20	50	50	104,04	50	51,46	50	56,07	50	47,14	50	45,94	50	47,82	50	48,9	**50**	**44,71**	50	47,33
rand-3-24-24-fcd	50	10	413,14	26	272,73	23	326,17	28	314,43	26	297,64	27	314,23	**29**	**272,14**	26	271,52	28	301,76
rand-3-24-24	50	6	536,52	16	467,63	14	415,31	15	425,36	14	367,36	16	401,12	16	405,91	15	478,29	**18**	**390,44**
rlfapScens11	12	10	84,9	11	103,52	11	112,19	11	98,85	11	101,48	11	103,31	10	39,88	**11**	**98,48**	11	112,5
schurrLemma	10	8	65,17	8	76,64	8	92,92	9	71,41	9	158,85	8	77,14	**9**	**71,1**	8	78,43	9	120,94
super-jobShop	46	33	12,35	35	31,83	34	15,24	36	36,12	35	24,63	33	6,68	**36**	**26,86**	34	20,62	35	45,39
tdsp	42	5	30,47	5	38,76	5	171,76	5	2,12	5	73,06	5	5,43	5	1,71	6	103,54	**6**	**27,42**
tightness0,1	100	100	11,33	100	4,6	100	6,13	100	4,42	**100**	**4,07**	100	4,28	100	4,54	100	4,17	100	4,62
tightness0,2	100	100	17,92	100	6,49	100	8,67	100	6,07	**100**	**5,7**	100	6,17	100	6,57	100	6,01	100	6,18
tightness0,35	100	100	13,27	100	6,18	100	7,18	100	5,53	**100**	**5,33**	100	5,59	100	5,88	100	5,44	100	5,68
tightness0,5	100	100	16,68	100	7,29	100	8,21	100	6,87	100	7,19	100	7,1	100	7,24	**100**	**6,86**	100	7
tightness0,65	100	100	7,23	100	5,05	100	5,41	**100**	**4,55**	100	4,74	100	4,63	100	4,86	100	4,82	100	4,66
tightness0,8	100	100	6,27	100	4,67	100	5,35	100	5,01	**100**	**4,64**	100	4,75	100	4,68	100	4,69	100	4,78
tightness0,9	100	100	7,83	100	6,86	100	7,08	100	7,12	100	7,4	100	7,35	100	7,11	100	7,47	**100**	**6,66**

Table 5.5 – Résultats expérimentaux par familles des trois méthodes : NRR, IncNG et IncNG+ sur différentes politiques de redémarrage - Partie 3.

Chapitre 6

NACRE : un moteur de raisonnement générique

Sommaire

En nous intéressant de près aux *increasing-nogoods* (présentés dans le chapitre 5), nous avons constaté que les solveurs de programmation par contraintes standards ne sont pas optimaux pour l'apprentissage de *nogoods* ou de clauses. En effet, utiliser les valeurs des variables sous forme de littéraux n'est pas un comportement usuel dans un solveur de programmation par contraintes. Afin de pallier ce problème, nous proposons le solveur NACRE (*Nogood And Clause Reasoning Engine*). C'est un solveur de programmation par contraintes écrit en C++. Il est basé sur une architecture modulaire conçue pour fonctionner avec des contraintes génériques et implémente plusieurs méthodes de résolution et heuristiques de l'état de l'art. De plus, les structures de données utilisées au sein de NACRE ont été soigneusement conçues afin de pouvoir gérer efficacement les *nogoods* et les clauses, ce qui en fait un outil idéal pour mettre en œuvre une stratégie d'apprentissage.

Nous présentons dans la suite NACRE 1.0.4 (Glorian (2019)), qui a été soumis à la *MiniTrack - CSP* de la compétition XCSP3 (Boussemart *et al.* (2016)) 2018 où il a terminé à la première place. Ce chapitre donne une description détaillée de NACRE 1.0.4 en tant que *framework*. Nous montrons, en particulier, comment le noyau a été conçu, quelles sont les méthodes de recherche et les heuristiques disponibles, et quels paramètres de configuration ont été utilisés lors de la compétition 2018 afin de rendre NACRE efficace en pratique à la fois en tant que solveur boîte noire et en tant que solveur modulaire.

6.1 Au cœur de NACRE

NACRE a été conçu pour être un solveur hybride, résolvant les problèmes de satisfaction de contraintes avec des méthodes dédiées ou inspirées de SAT. Grâce à des structures de données

dédiées, il est capable de travailler efficacement avec les *nogoods* et les clauses. Comme présenté dans la section 6.1.1, le noyau de NACRE a été conçu avec cet objectif particulier. De plus, NACRE a vocation à être utilisé comme *framework*. Il est donc conçu de manière modulaire, ce qui permet à l'utilisateur d'implémenter de nouvelles idées, par exemple sur les propagateurs de contraintes ou les méthodes de résolution, de manière simple. De plus, il est possible de mettre en œuvre facilement de nouvelles méthodes de raisonnement de type *nogoods* ou clauses, et même de spécifier des techniques de minimisation originales afin de réduire la taille de ces derniers. Même si NACRE 1.0.4 a été développé en tant que solveur extensible modulaire, ses performances n'ont pas été laissées de côté. Afin de le prouver, la version 1.0.4 de NACRE a été soumise à la *MiniTrack - CSP* de la compétition XCSP3 2018.

Lors de cette compétition, 11 solveurs concouraient et NACRE a terminé à la première place. Cette catégorie (*MiniTrack*) est destinée aux mini-solveurs open-source et vise à découvrir de nouvelles idées et de nouveaux solveurs indépendants. Les sources NACRE 1.0.4 sont disponibles sur `https://github.com/crillab/nacre_mini`. Cette catégorie a aussi pour vocation de trouver un solveur de programmation par contraintes pouvant facilement être mis à jour à la manière de *Minisat* (Sörensson et Eén (2009)). Notez que les solveurs ont été évalués, dans cette catégorie, sur un ensemble restreint de contraintes les plus répandues (plus de détails dans la section 6.2).

6.1.1 Structures de données

Dans NACRE, nous avons choisi de représenter les variables et les valeurs conformément aux techniques SAT au moyen de variables propositionnelles. Formellement, il y a une application $A : Variables \times Values \rightarrow \mathbb{N}$ telle que chaque couple (x, a), avec $x \in \mathcal{X}$ et $a \in \texttt{dom}(x)$, est associé à un entier x_a. Par conséquent, les variables sont stockées en tant qu'objets virtuels, ce qui signifie que chaque variable ne possède pas directement son propre domaine. La variable connaît uniquement l'index du début du domaine au sein d'un grand tableau de valeurs contiguës ainsi que la taille initiale de son domaine (parmi d'autres données, telles que les bornes supérieure et inférieure du domaine, etc.).

Il existe deux tableaux de ce type (trois dans le cas de l'apprentissage de clauses, voir la section 6.1.4). Le premier, appelé VARPROPS, sauvegarde ce que nous appelons des variables propositionnelles et peut être interprété comme la partie *dense* d'un ensemble éparse plein (*sparse set*, Aho *et al.* (1974)). Les variables propositionnelles (parfois référencées comme *VP*) sont des objets encapsulant des informations utiles telles que la valeur de la VP au sein du domaine de la variable CP d'origine. Elles contiennent aussi l'état actuel de la variable propositionnelle encapsulée : 0 si la valeur est affectée, 1 si la valeur est supprimée et 2 si elle n'est pas décidée. La position *locale* de la VP au sein du deuxième tableau DOMVALUES décrit ci-dessous. Nous appelons *locale* une position normalisée en fonction de l'index du début du domaine de la variable associée dans le tableau.

Le second tableau, appelée DOMVALUES, est similaire à la partie *sparse* d'un ensemble éparse réversible plein (*reversible sparse set*), c'est-à-dire qu'il sauvegarde pour chaque variable un tableau en deux parties séparées par une limite. Cette limite correspond à la taille actuelle du domaine de la variable. Les valeurs situées à gauche de cette limite sont toujours dans le domaine de la variable alors que celles à droite ont été supprimées. Cela nous permet de supprimer et de restaurer des valeurs facilement et efficacement dans le domaine d'une variable car nous avons uniquement besoin d'un échange d'entier et d'une instruction de décrémentation afin de supprimer une valeur ou d'affecter une variable. Cette structure légère est pratiquement exempte de coût de calcul pour le retour arrière (*backtrack-free*), car nous avons simplement besoin de

sauvegarder la position locale de la limite à chaque niveau de décision. Notez que nous aurions pu mettre la valeur originale de la VP (c'est-à-dire non normalisée) dans ce tableau également, mais nous avons décidé de faire une distinction par souci de clarté.

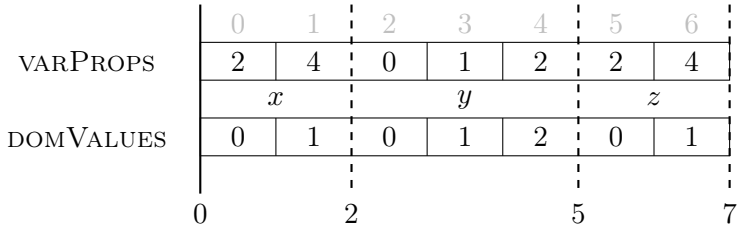


Figure 6.1 – Structures de données utilisées par NACRE afin de représenter les variables et leurs valeurs.

La figure 6.1 montre un exemple sur trois variables x, y et z avec $\texttt{dom}(y) = \{0, 1, 2\}$ et $\texttt{dom}(x) = \texttt{dom}(z) = \{2, 4\}$ telles qu'elles sont définies initialement (état initial). Le tableau supérieur est varProps dans lequel nous pouvons clairement voir les valeurs du domaine (nous ne montrons pas les autres informations des VP dans la figure, par souci de clarté). Le tableau inférieur est domValues contenant les positions locales. Nous pouvons observer qu'ils sont initialement triés de manière croissante. Au-dessus de ces deux tableaux, en gris clair, nous pouvons voir les entiers associés à ces variables propositionnelles (qui sont essentiellement les indices de la table varProps).

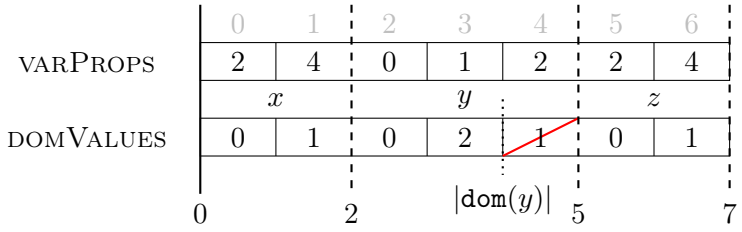


Figure 6.2 – État des structures de données utilisées par NACRE après la suppression de la valeur 1 au sein de la variable y.

La figure 6.2 conserve le même exemple que la figure 6.1 mais une fois que la valeur 1 ait été supprimée du domaine de la variable y. Techniquement, les domValues 1 et 2 ont été permutés et la limite dans domValues associée à la variable y a été décrémentée (voir le segment en pointillé).

6.1.2 Heuristiques

Dans NACRE 1.0.4, l'utilisateur peut sélectionner un certain nombre d'heuristiques afin d'ajuster la recherche. Deux types d'heuristiques d'ordonnancement des variables sont disponibles dans cette version : les heuristiques dynamiques (*dom* et *dom/deg*) et les heuristiques adaptatives (*dom/wdeg*). Les descriptions détaillées de ces heuristiques sont disponibles dans la section 1.3 du chapitre 1. Un résumé est mis à disposition dans la tableau 6.1 ainsi que les différentes options.

Une fois qu'une variable est choisie pour affectation, une valeur doit être sélectionnée dans son domaine à l'aide d'une heuristique de choix de valeurs. Différentes heuristiques de classement

Nom	Référence	Option
dom	Haralick et Elliott (1980), Bitner et Reingold (1975)	`-dom`
dom/deg	Bessière et Régin (1996)	`-domdeg`
dom/wdeg	Boussemart *et al.* (2004)	`-domwdeg` (default)

TABLE 6.1 – Tableau récapitulatif des heuristiques de choix de variables et des options associées dans NACRE 1.0.4.

des valeurs sont disponibles dans NACRE 1.0.4. Outre les heuristiques classiques de valeurs (valeur minimale, valeur maximale, etc.), une version simplifiée de l'heuristique SAT *polarity* a été implémentée (Pipatsrisawat et Darwiche (2007)). Cette heuristique tente d'affecter la variable sélectionnée au dernier choix de valeur qui a été effectué. Si le choix n'est plus valide, cela signifie que la valeur considérée a déjà été supprimée, l'heuristique de valeur est alors sollicitée. Deux options principales sont disponibles, chacune ayant une version minimale et maximale. D'une part, avec *min-value*, nous choisissons la valeur minimale dans le domaine de la variable considérée ; il est également possible avec *max-value* de sélectionner la valeur maximale dans le domaine de la variable. Le choix de la valeur minimale ou maximale, en fonction du problème, peut améliorer considérablement les performances du solveur. D'autre part, l'heuristique *first-value* est disponible. C'est une méthode lexicographique de sélection de valeurs basée sur l'implémentation des domaines. Comme nous utilisons des structures similaires à des ensembles éparses, nous pouvons sélectionner la première VP au sein de l'ensemble *domValues*. Nous pouvons également sélectionner la dernière avant la limite en utilisant d'heuristique *last-value*. Il est également possible de rendre la sélection de valeurs complètement aléatoire.

Nom	Option
Polarity-based	`-saving` (à combiner)
min-value	`-valMin` (default)
max-value	`-valMax`
first-value	`-valFirst`
last-value	`-valLast`
Aléatoire	`-valRand`

TABLE 6.2 – Tableau récapitulatif des heuristiques de choix de valeurs et des options associées dans NACRE 1.0.4.

6.1.3 Stratégies de redémarrage

NACRE dispose de stratégies de redémarrage intégrées qui permettent de tester l'apprentissage de *nogoods* ou de clauses sous différents paramètres. En effet, la stratégie de redémarrage peut avoir un impact important sur l'apprentissage et la résolution. C'est la raison pour laquelle nous avons décidé de mettre en œuvre différentes stratégies, allant de la suite géométrique à la suite de Luby *et al.* (1993). La première stratégie disponible permet de ne pas faire de redémarrages, parfois utile si nous souhaitons l'utiliser comme référence, voire dans un contexte sans apprentissage ni heuristiques adaptatives. Viennent ensuite les séquences basées sur la suite de Luby: les premiers termes sont de la forme $(1, 1, 2, 1, 1, 2, 4, 1, \ldots)$.

Trois stratégies basées sur Luby sont disponibles, `Luby(#rst)` $\times N$ où N peut prendre les valeurs 10, 50 et 100 et où $\#rst$ est le nombre de redémarrages déjà effectués (afin d'obtenir le terme correct de la séquence originale). Ceci est utile pour essayer des stratégies de redémarrage plus ou moins agressives. Enfin, des suites géométriques sont disponibles pour une progression plus constante de la valeur de coupure (*cutoff*). Quatre d'entre elles sont implémentées dans NACRE 1.0.4. Par défaut, la progression est de 3%. Les autres suites disponibles ont des progressions de 10%, 50% et 100%. Cela permet à l'utilisateur de choisir parmi une grande variété de stratégies, voire de les modifier aisément au cours de la recherche.

Nom	Option
Pas de redémarrage	`-noRst`
`Luby(#rst)` $\times$ 10	`-luby10`
`Luby(#rst)` $\times$ 50	`-luby50`
`Luby(#rst)` $\times$ 100	`-luby100`
Suite géométrique de raison $1,03$	(default)
Suite géométrique de raison $1,10$	`-10perc`
Suite géométrique de raison $1,50$	`-50perc`
Suite géométrique de raison 2	`-double`

TABLE 6.3 – Tableau récapitulatif des stratégies de redémarrage et des options associées dans NACRE 1.0.4.

6.1.4 Méthodes de résolution

Les méthodes de résolution disponibles au sein de NACRE sont de trois types. Le premier type concerne une méthode standard de programmation par contraintes : *Maintaining Arc Consistency* (MAC, présentée dans le chapitre 1, section 1.2.1.1). Le second type englobe les méthodes standards de programmation par contraintes complétées par un système d'apprentissage de *nogoods*. Deux méthodes, utilisant les *nld-nogoods* et les *increasing-nogoods*, sont disponibles. Le dernier type concerne le moteur dit hybride qui permet la génération de clauses conformément aux techniques SAT. Ce moteur sur-mesure est inspiré des *nogoods* généralisés et des explications paresseuses. Le tableau 6.4 résume les méthodes disponibles ainsi que les options associées à chacune.

Nom	Référence	Option
Maintaining Arc Consistency	Sabin et Freuder (1994a;b)	`-complete`
Negative last decision nogoods	Lecoutre *et al.* (2007a;b; 2009)	`-nld`
Increasing nogoods	Lee et Zhu (2014), Lee *et al.* (2016)	`-incng`
Analyse de conflits	Katsirelos et Bacchus (2003), Gent *et al.* (2010)	`-ca`

TABLE 6.4 – Tableau récapitulatif des méthodes de résolution et des options associées dans NACRE 1.0.4.

La figure 6.3 représente le diagramme de classes des solveurs au sein de NACRE. Nous pouvons apercevoir les quatre méthodes : à savoir la classe `CompleteSolver` qui implémente la méthode `MAC`, la classe `LecoutreCompleteSolver` qui implémente la méthode `Negative last`

`decision nogoods` ; la classe `IncNGCompleteSolver` qui implémente la méthode `Increasing nogoods` et enfin la classe `ConflictAnalysisCompleteSolver` qui implémente une méthode d'`Analyse de conflits` hybride qui prend comme base les explications paresseuses.

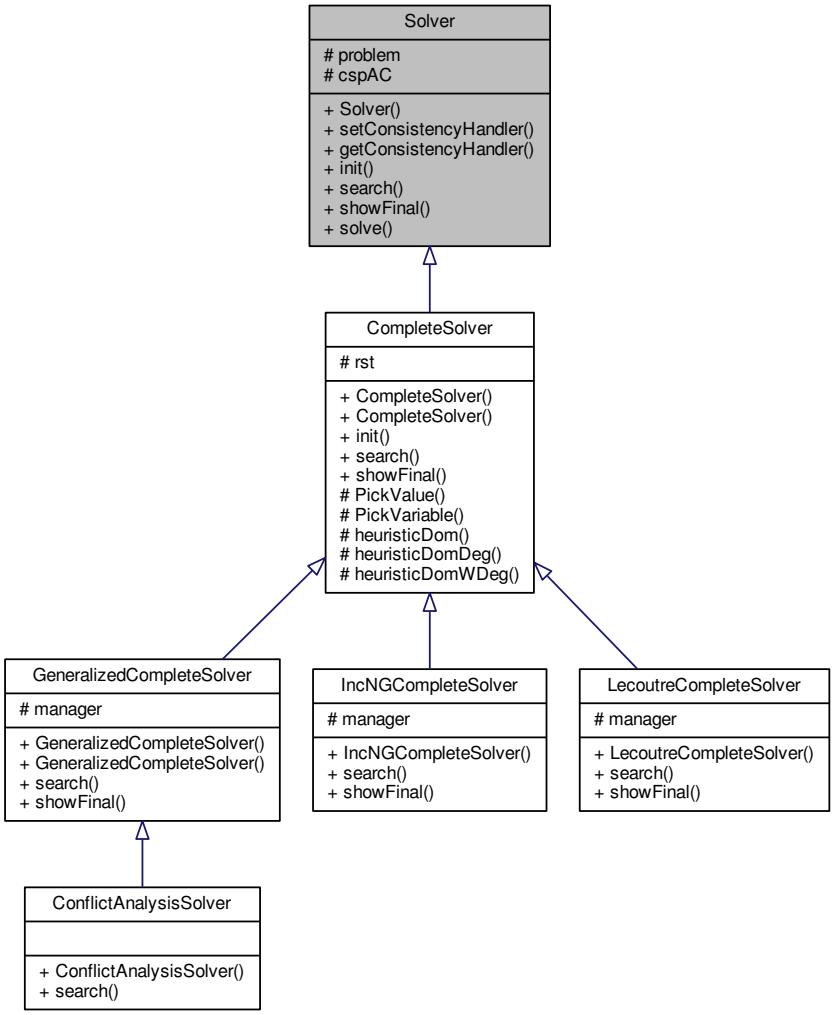


FIGURE 6.3 – Graphe de classes des méthodes de résolutions

Il est très facile d'étendre NACRE 1.0.4 avec de nouvelles méthodes de résolution personnalisées. Pour cela, il faut, au minimum, créer une classe héritant de la classe abstraite `Solver` (ou un de ses enfants, `CompleteSolver` notamment où des heuristiques sont déjà implémentées) et implémentant la méthode `search`. Les autres méthodes de la classe abstraite `Solver` peuvent aussi être réécrites, en particulier la méthode `showFinal` qui permet de personnaliser l'affichage lorsque qu'un signal d'arrêt est reçu (par exemple, lorsque l'instance est satisfiable ou que le *timeout* est atteint). La méthode `init` peut aussi être intéressante. En effet, elle est utilisée pour le traitement initial sur le réseau. Il est possible d'y intégrer un prétraitement.

6.1.4.1 Méthodes simples

NACRE fournit diverses méthodes pour résoudre les CSP, présentées au format XCSP3 [3]. NACRE se voulant un *framework* pour gérer les clauses et les nogoods, plusieurs méthodes de résolution de l'état de l'art sont mises en œuvre. La première est une approche standard de maintien de cohérence d'arcs, MAC (Sabin et Freuder (1994a;b), présentée dans le chapitre 1, section 1.2.2.2). Cette méthode maintient la cohérence d'arcs (présentée dans le chapitre 1, section 1.2.1.1) au sein d'une recherche en profondeur d'abord par arbre binaire avec retour arrière. Un branchement binaire signifie que, à chaque nœud de l'arbre de recherche, une branche gauche étiquetée avec une décision positive $x = a$ est développée en premier lieu. Ensuite, si la première branche mène à un état conflictuel, une branche droite étiquetée avec une décision négative $x \neq a$ est développée. Afin de le faire efficacement, NACRE utilise des estampilles (*stamps*) pour les contraintes, les variables et également pour gérer la file de propagations. Techniquement, avec les estampilles, un ensemble n'est pas requis pour représenter la file de propagations, ce qui rend la gestion moins coûteuse.

Algorithme 6.1 : $Propagate(x)$

```
   Data : P = (V, C) a CN and x ∈ V a variable
   Result : true if there is a conflict, false otherwise
1  propagateQ ← {x};
2  while propagateQ ≠ ∅ do
3      y ← pickVariable(propagateQ);
4      forall c ∈ C s.t. y ∈ scp(c) do
5          if stamp(c) < stamp(y) then
6              touched ← ∅;
7              if revise(c, touched) then
8                  return true;
9              propagateQ ← propagateQ ∪ touched;
10             updateStamp(c);
11 return false;
```

L'algorithme 6.1 montre comment la propagation est effectuée après que le solveur ait pris une décision sur une variable x. Jusqu'à ce qu'un point fixe soit atteint, il sélectionne (ligne 3) une variable de `propagateQ` à l'aide de la fonction `pickVariable()`. Cette fonction renvoie la variable ayant le plus petit domaine courant de la file de propagations et la supprime de *propagateQ*. Ensuite, nous parcourons les contraintes impliquant cette variable (ligne 4). La ligne 5 montre que l'estampille de la variable actuelle est comparée à celle de la contrainte. L'estampille d'une contrainte est mise à jour avec l'estampille maximale des variables de sa portée à la ligne 10. L'estampille des variables est mise à jour lorsqu'elles sont ajoutées à la file de propagations (lignes 1 et 9), cela signifie que leur domaine a été modifié (*touched*).

Ce système d'estampillage évite les appels inutiles de propagateurs de contraintes qui pourraient être très coûteux. En effet, quand une contrainte c est utilisée pour être filtrée, alors toutes les variables de $scp(c)$ sont GAC selon la contrainte c (cela est assuré par la fonction `revise` de la ligne 7) et $stamp(c)$ est mis à jour de telle sorte que $stamp(c) \geq stamp(y)$ pour tout $y \in scp(c)$.

3. www.xcsp.org

Il est clair qu'il n'est pas nécessaire de reconsidérer c tant qu'aucune variable de sa portée n'a été modifiée. Lorsqu'une variable y est modifiée, $stamp(y)$ est mis à jour et y est ajoutée à la file de propagations. Par conséquent, si une variable y de $scp(c)$ est touchée après le filtrage de c, alors $stamp(y) > stamp(c)$ et donc la condition *if* (ligne 5) est valide lorsque y est choisie dans la file de propagations. Cela implique que la contrainte est considérée à nouveau pour un appel à son propagateur. En effet, si la variable considérée avait une estampille plus petite (ou égale) à la contrainte actuelle c, cela signifierait que cette même variable avait été modifiée avant le déclenchement du propagateur de contrainte de c. Ainsi, le domaine de cette variable doit déjà être cohérent par rapport à cette contrainte.

Les lignes 6 à 10 ne sont exécutées que si l'estampille de la contrainte considérée n'a pas été mise à jour depuis qu'une variable de sa portée a été modifiée. La fonction `revise` (ligne 7) appelle le propagateur dédié pour la contrainte c donnée en paramètre. Le dernier paramètre de cette fonction, `touched`, est un ensemble qui regroupe toutes les variables modifiées lors du filtrage de la contrainte afin de les ajouter à la file de propagations (ligne 9).

6.1.4.2 Méthodes avec apprentissage

L'enregistrement de *nogoods* présente l'avantage d'éviter certaines formes de *thrashing* (Gomes *et al.* (2000), voir le chapitre 2), c'est-à-dire d'explorer plusieurs fois les mêmes sous-arbres insatisfiables. Il existe deux méthodes classiques pour identifier et stocker les *nogoods* : pendant la recherche (étudiée dans la section suivante) ou au redémarrage. La découverte de *nogoods* au redémarrage (présenté dans le chapitre 2) extrait les informations en analysant l'arbre de recherche actuel lorsqu'un redémarrage est déclenché. Deux méthodes bien connues et efficaces ont été mises en œuvre dans NACRE, les *nld-nogoods* (Lecoutre *et al.* (2007b), présentés dans le chapitre 2, section 2.3) et les *increasing-nogoods* (Lee *et al.* (2016), présentés dans le chapitre 2, section 2.4).

Les *nld-nogoods* sont extraits à chaque redémarrage à partir de la dernière branche de l'arbre de recherche. Supposons que la séquence de décisions de la branche la plus à droite de l'arbre de recherche actuel soit $\Sigma = \langle \delta_1, \ldots, \delta_m \rangle$, où chaque décision de Σ est une décision positive ou négative. On sait que pour tout i tel que $1 \leq i \leq m$ et $neg(\delta_i)$, l'ensemble $\Delta = \{\delta_j : 1 \leq j < i \wedge pos(\delta_j)\} \cup \{\neg\delta_i\}$ est un *nld-nogood* réduit qui peut être écrit $\bigwedge_{\delta_j \in \Delta} \delta_j \Rightarrow \neg\delta_i$ sous forme dirigée.

Un *increasing nogood* est une forme compressée de tous les *nld-nogoods* réduits pouvant être extraits à chaque redémarrage, sans perte d'efficacité. En effet, il est assez facile de déduire qu'il existe certaines similitudes entre les *nld-nogoods* extrait d'un même redémarrage : la partie gauche de l'implication est partagée entre plusieurs nogoods. Ensuite, il est possible de réécrire l'ensemble de *nld-nogoods* en tant que contrainte et d'utiliser un propagateur dédié assurant GAC.

L'algorithme 6.2 montre comment intégrer l'appel à la propagation des *nogoods* (*nld-nogoods* ou *increasing-nogoods*). Les lignes 4 à 8 sont ajoutées à l'algorithme 6.1. Pour les deux types de *nogoods*, le filtrage a lieu uniquement si la variable est assignée. En effet, pour les *nld-nogoods*, qui sont composés uniquement de décisions positives, le filtrage ne peut s'effectuer que lorsque toutes les décisions ont été faites à nouveau sauf une. Pour les *increasing-nogoods*, trois cas de filtrage sont possibles (voir chapitre 2, section 2.4.3 pour plus de détails) : la décision positive pointée par alpha est satisfaite, la décision positive pointée par bêta est satisfaite et une décision négative située entre alpha et bêta est falsifiée. Par définition, satisfaire une décision positive

Algorithme 6.2 : $Propagate + NG(x)$

```
Data : P = (V, C) a CN and x ∈ V a variable
Result : true if there is a conflict, false otherwise
1  propagateQ ← {x};
2  while propagateQ ≠ ∅ do
3      y ← pickVariable(propagateQ);
4      if |dom(y)| = 1 then
5          touched ← ∅;
6          if reviseNGbase(y, touched) then
7              return true;
8          propagateQ ← propagateQ ∪ touched;
9      forall c ∈ C s.t. y ∈ scp(c) do
10         if stamp(c) < stamp(y) then
11             touched ← ∅;
12             if revise(c, touched) then
13                 return true;
14             propagateQ ← propagateQ ∪ touched;
15             updateStamp(c);
16 return false ;
```

ou falsifier une décision négative revient à effectuer une affectation. Clairement, dans les deux approches de *nogoods*, appeler le filtrage uniquement lorsque les variables sont assignées suffit afin d'éviter des appels inutiles. De la même manière que précédemment, les variables touchées sont ajoutées dans la file de propagations afin d'être filtrées par la suite si nécessaire.

6.1.4.3 Méthodes hybrides

Il est aussi possible d'identifier des *nogoods* lors de la recherche lorsqu'un conflit est détecté. NACRE implémente un moteur de raisonnement de clauses générique utilisant des explications paresseuses. Nous considérons la procédure d'apprentissage de *nogoods* généralisés (Katsirelos et Bacchus (2003), présentée dans le chapitre 4, section 4.1), et les explications paresseuses proposées dans (Gent *et al.* (2010), présentées dans le chapitre 4, section 4.2) comme point de départ.

Une manière de générer un *nogood* non trivial à chaque conflit consiste à analyser en profondeur la séquence des étapes de propagation en construisant un graphe d'implication à la manière de SAT et en identifiant la raison de l'échec (Zhang *et al.* (2001)). Un graphe d'implication est un graphe acyclique dirigé (DAG) qui enregistre les relations existantes entre les littéraux, comme nous pouvons l'observer au cours du processus de résolution. Chaque sommet représente un littéral λ (avec son niveau de propagation associé) et ses arcs entrants dans le DAG représentent les raisons qui le propagent. Lorsqu'un conflit survient, chaque coupure dans le graphe d'implication qui mène au conflit peut l'expliquer. Cet ensemble de littéraux, utilisé comme explication de conflit, peut être bloqué à l'avenir en dérivant une clause. En règle générale, nous nous intéressons uniquement aux points d'implication uniques (UIP, Marques-Silva et Sakallah (1999)) qui sont des sommets, au niveau de décision actuel, qui dominent le conflit. Il est important de

noter qu'ils peuvent être utilisés pour effectuer en toute sécurité un retour arrière non chronologique jusqu'au niveau de décision qui correspond à la valeur maximale parmi tous les niveaux de décision associés aux littéraux de l'explication de conflit.

En pratique, une construction explicite du graphe d'implication peut être évitée. En effet, les clauses conflictuelles peuvent être dérivées de la formule courante si nous enregistrons indépendamment l'explication $\texttt{expl}(\lambda)$ de chaque déduction λ. Plus précisément, en supposant que toutes ces explications soient stockées et en partant d'un ensemble conflictuel Λ de littéraux, la procédure d'analyse de conflits sélectionne de manière itérative un littéral $\lambda \in \Lambda$ impliqué au niveau de décision le plus récent (c'est-à-dire le plus grand niveau de décision actuel) et remplace λ par $\texttt{expl}(\lambda)$ dans Λ. Ce processus s'arrête lorsqu'un seul littéral du niveau actuel reste dans Λ. Par conséquent, les littéraux restant dans Λ, une fois la procédure terminée, forment un UIP. Comme Λ représente une explication du conflit, la clause $\bigvee_{\lambda \in \Lambda} \neg\lambda$ est une conséquence logique du problème (au moins un littéral de Λ doit être faux). Nous pouvons observer que lors du retour arrière, cette clause reste unitaire (avec le littéral λ_{unit}) jusqu'à ce que le niveau de décision le plus élevé des autres littéraux dans $\Lambda \setminus \{\lambda_{unit}\}$ soit atteint.

D'une part, le formalisme SAT est très simple (à chaque fois qu'un littéral est déduit, son explication peut être directement calculée en considérant la clause qui a déclenchée la déduction) et très bien adapté à une extraction de *nogoods* par la simple analyse d'un graphe d'implication. D'autre part, le formalisme CSP est plus sophistiqué (par exemple, en tenant compte des contraintes globales) et rend plus difficile l'identification précise de la raison d'une simple déduction : la construction du graphe d'implication n'est plus aussi directe. De plus, même si nous pouvons envisager de construire de tels graphes, le calcul de l'explication de chaque déduction peut considérablement ralentir l'algorithme de recherche si cela n'est pas fait de manière optimale.

Dans NACRE, une méthode générique non intrusive nous permettant de calculer des explications a été implémentée. Les explications sont générées (à la manière des explications paresseuses) à la demande dans trois situations. Premièrement, si un littéral λ est propagé à partir d'une clause Λ, l'explication est donnée par la négation des littéraux de $\Lambda \setminus \{\lambda\}$. Deuxièmement, un littéral peut être propagé en raison des contraintes de domaine qui sont de deux types : `atMostOne` et `atLeastOne`. La contrainte `atLeastOne` est utilisée comme explication lorsqu'il ne reste qu'une seule valeur a dans le domaine de x. Dans ce cas, nous avons $\texttt{expl}(x = a) = \{x \neq b : b \in dom^{init}(x) \setminus \{a\}\}$. La contrainte `atMostOne` est utilisée comme explication lorsqu'une valeur b est supprimée de x quand x est affectée à a. Dans ce cas, nous avons $\texttt{expl}(x \neq b) = \{x = a\}$.

Enfin, un littéral λ d'une variable x peut être propagé lors de l'exécution de l'algorithme de filtrage ϕ_c associé à une contrainte c qui n'est pas une clause. Dans ce cas, il est possible d'expliquer pourquoi λ a été propagé en considérant les valeurs supprimées dans les domaines des variables de $scp(c) \setminus \{x\}$. Pour éviter le calcul systématique des explications à chaque propagation, NACRE utilise à nouveau un système d'estampilles qui associe un entier à chaque valeur (littéral). Quand une valeur a d'une variable x est supprimée, son estampille est mise à jour de manière à ce que $stamp(x, a)$ soit supérieur à tout $stamp(x', a')$ tel que a' est supprimée de x' (avant la suppression de a). Pour éviter que les retours arrière ne soient trop coûteux, nous utilisons un entier qui est incrémenté à chaque suppression. Ainsi, lorsqu'une explication est requise pour une valeur a dans une variable x, il suffit de garder en mémoire quelle contrainte a été utilisée pour effectuer la propagation. En effet, il suffit de considérer la valeur a' de $x' \in scp(c) \setminus \{x\}$ tel que $stamp(x, a) > stamp(x', a')$.

Une fois le processus d'analyse de conflits terminé, une nouvelle clause est ajoutée à la base

de clauses et un retour arrière non chronologique est effectué. Afin de gérer la base de clauses, nous adoptons la stratégie des *two-watched literals* (Moskewicz *et al.* (2001), présentée dans le chapitre 3, section 3.4.2).

Algorithme 6.3 : PropagateClause(x)

```
   Data : P = (V, C) a CN and x a variable of V
   Result : a constraint if a conflict occurs, null otherwise
 1 continue ← true;
 2 clauseQ ← ∅;
 3 propagateQ ← {x};
 4 while continue do
 5     while propagateQ ≠ ∅ do
 6         y ← pickVariable(propagateQ);
 7         clauseQ ← clauseQ ∪ {y};
 8         forall c ∈ C s.t. y ∈ scp(c) do
 9             if stamp(c) < stamp(y) then
10                 touched ← ∅;
11                 if revise(c, touched) then
12                     return c;
13                 propagateQ ← propagateQ ∪ touched;
14                 updateStamp(c);
15     continue ← false;
16     while clauseQ ≠ ∅ do
17         y ← pickVariable(clauseQ);
18         touched ← ∅;
19         c ← bcp(y, touched);
20         if c is not null then
21             return c;
22         if touched ≠ ∅ then
23             continue ← true;
24             propagateQ ← propagateQ ∪ touched;
25 return null ;
```

L'algorithme 6.3 adapte l'algorithme 6.1 afin de gérer la propagation des clauses. Nous choisissons de propager entièrement les contraintes (lignes 5 à 14) avant de réaliser la propagation des clauses (lignes 16 à 24). En effet, même si cette méthode de résolution se veut hybride, nous considérons que la partie CP, c'est-à-dire la propagation de contraintes, doit être réalisée avant la propagation de clauses pour ne pas trop perturber les heuristiques CP dédiées qui peuvent être utilisées. Les deux boucles de propagations sont effectuées jusqu'à ce qu'un point fixe soit atteint. La boucle de propagation de contrainte reste similaire à l'algorithme 6.1 avec une légère variation, à la ligne 7, une fois la variable sélectionnée, elle est ajoutée à une nouvelle file, `clauseQ`. Cette file permet à la boucle de la ligne 16 d'enregistrer toutes les différentes variables altérées au cours du processus de cohérence d'arc généralisée (lignes 5 à 14). Le point fixe global de l'algorithme est atteint lorsque la propagation de clauses (`bcp`) ne modifie aucune variable

(lignes 22 à 24).

Si un conflit est découvert par l'algorithme 6.3, une contrainte est retournée. Cette contrainte sera le point de départ de l'analyse de conflits. Si cette contrainte est une clause alors le début du processus est semblable à l'analyse de conflits SAT. Dans le cas où le conflit provient d'une contrainte CP (ligne 12), alors il faut faire appel à une fonction de raisonnement afin de récupérer les valeurs qui ont conduit à ce conflit. Pour cela, nous proposons nos deux algorithmes pour contraintes génériques 6.4 et 6.5. Ils permettent respectivement de décrire les littéraux à l'origine du conflit lorsqu'une contrainte quelconque entre en conflit et de donner la raison de la propagation d'un littéral au sein d'une contrainte.

Algorithme 6.4 : ConstraintConflict(c)

Data : A conflictual constraint c
Result : A set of literals that explain the conflict

```
1 cl ← ∅
2 foreach x ∈ scp(c) do
3     if CAstamp(x) < GlobalCAStamp then
4         CAstamp(x) ← GlobalCAStamp;
5         cl ← cl ∪ {lit(a) | a ∈ dom(x)^init \ dom(x) ∧ lvl(a) > 0};
6 return cl;
```

L'algorithme 6.4 présente la procédure générique qui permet de récupérer l'explication d'un conflit provenant d'une contrainte quelconque. En effet, en s'inspirant des explications paresseuses et de la génération de clauses en SAT nous obtenons cet algorithme. Une explication paresseuse générique est produite en considérant toutes les suppressions de valeurs de la portée de la contrainte considérée (ligne 2). À cela, nous ajoutons la vérification du niveau de propagation de la suppression de valeur (ligne 5) grâce à la fonction `lvl`(a). Elle retourne le niveau de propagation d'un littéral (ou d'une valeur plus simplement) passé en paramètre. Nous n'ajoutons pas les littéraux propagés au niveau 0 car ceux-ci ne participent pas à l'explication du conflit. De plus, nous utilisons à nouveau un système d'estampilles. Une estampille globale `GlobalCAStamp`, qui est initialisée à 1 et incrémentée à chaque appel à l'analyse de conflits, est comparée à une estampille, initialisée à 0, locale aux variables CP. Cette dernière est accessible grâce à la fonction `CAstamp`(x). Si l'estampille locale est inférieure alors la variable n'a jamais été rencontrée par une fonction de raisonnement lors de cette analyse de conflits. Dans le cas contraire, la variable a déjà été rencontrée, par conséquent ses valeurs supprimées ont déjà été considérées. Les valeurs pertinentes de la variable seront alors ajoutée à l'ensemble retourné.

L'algorithme 6.5 détaille la méthode d'extraction d'explications paresseuses génériques implémentée dans NACRE. De la même manière que l'algorithme précédent, celui utilise les estampilles. En plus de l'estampille, d'autres méthodes de contournement sont utilisées. Premièrement, après la vérification de l'estampille, nous regardons si la variable traitée est différente de celle du littéral dont la raison est demandée (ligne 3). Le seul cas où nous pouvons continuer à traiter cette variable se produit lorsque le littéral dont la raison est demandée est négatif. En effet, comme nous construisons une clause (disjonction) de manière itérative et que les explications sont des conjonctions cela signifie que nous voulons savoir pourquoi l a été affecté. Par conséquent, demander la raison d'une affectation par une contrainte peut être aussi lié aux valeurs supprimées de la même variable. Dans un second temps, si les tests de la ligne 3 sont vérifiés, nous ajoutons à l'ensemble retourné les valeurs négatives du domaine de la variable considé-

Algorithme 6.5 : ConstraintReason(c, l)

Data : A literal l propagated by a constraint c
Result : The reason of l as a set of literals

```
1 cl ← ∅
2 foreach x ∈ scp(c) do
3     if CAstamp(x) ≠ GlobalCAStamp and (var(l) ≠ x or neg(l)) then
4         CAstamp(x) ← GlobalCAStamp;
5         cl ← cl ∪ {lit(a) | a ∈ dom(x)^init \ dom(x) ∧ lvl(a) > 0 ∧ order(a) < order(l)};
6 return cl;
```

rée (toujours avec un niveau supérieur à 0). Un filtre supplémentaire permet de ne prendre en compte que les valeurs pertinentes. Pour cela nous utilisons l'ordre de propagation des littéraux grâce à la fonction $\texttt{order}(a)$ qui retourne un entier qui correspond à l'ordre chronologique de sa propagation. En effet, la raison de la propagation d'un littéral (positivement ou négativement) ne peut se situer que dans les propagation faites avant celui-ci.

Il est tout-à-fait possible de redéfinir ces fonctions de raisonnement (et analyseurs de conflits) génériques de manière *ad-hoc* afin d'obtenir des explications plus précises et, en général, plus petites. Nous en donnons un exemple dans le chapitre 7, où nous proposons des fonctions de raisonnement pour la contrainte *Element*.

6.1.5 Réduction de la base de clauses

La procédure qui garantit la propagation unitaire sur les clauses dépend fortement de la taille de la base. Pour que la base des clauses reste gérable et que la propagation unitaire soit efficace, il est courant de réduire la base en supprimant les clauses considérées comme non pertinentes pour les prochaines étapes de recherche. Plusieurs mesures ont été proposées à cet effet (Eén et Sörensson (2003), Audemard et Simon (2009b), Jabbour *et al.* (2018), Audemard *et al.* (2011)). Généralement, les solveurs SAT utilisent l'une des stratégies suivantes: *activity* qui considère une clause apprise comme non pertinente si son activité ou son implication dans l'analyse de conflits récente est marginale (Eén et Sörensson (2003)) ; ou *literal block distance (LBD)* qui utilise le nombre de niveaux différents impliqués dans une clause apprise donnée pour quantifier sa qualité (Audemard et Simon (2009b)), les clauses avec un LBD plus petit sont considérées plus pertinentes.

Même si ces mesures sont bien adaptées au comportement SAT, elles ne le sont pas vraiment dans notre cas. En effet, les clauses incluent souvent un grand nombre de littéraux, car nous utilisons une procédure générique. Mais contrairement à SAT, nous pouvons identifier des littéraux liés par une variable commune du point de vue de la programmation par contraintes (variable CP). Clairement, dans ce cas, les clauses contenant peu de variables CP sont plus puissantes en ce qui concerne la propagation unitaire. Pour prendre en compte ces informations, nous proposons de déployer une nouvelle mesure qui prend en compte le nombre de variables CP différentes au sein d'une clause. Lorsque deux clauses possèdent le même nombre de variables CP, leur activité est utilisée pour les départager.

Un point important concernant la réduction de la base concerne sa fréquence. Dans NACRE 1.0.4, nous avons choisi de définir la limite initiale de taille de la base de clauses à 4000 clauses. Lorsque la limite est atteinte, la moitié des clauses apprises est supprimée à l'aide de notre mesure

Rank	Solver (Version)	#solved(%)	#SAT/#UNS	% VBS	SumCPU	MedCPU	AvgCPU
	Virtual Best Solver (VBS)	113 (64)	53/60	100	11899.26	0.62	105.30
1	NACRE (1.0.4)	86 (49)	43/43	76	9948.56	0.40	115.68
2	miniBTD_12 (180727_12)	79 (45)	36/43	70	9534.55	0.88	120.69
3	miniBTD (180727_3)	75 (43)	32/43	66	13679.82	1.17	182.40
4	cosoco (1.12)	72 (41)	42/30	64	14074.67	2.12	195.48
5	minimacht (180727)	69 (39)	37/32	61	13299.29	4.85	192.74
6	GG's minicp (180429)	56 (32)	37/19	50	5861.32	3.16	104.67
7	Solver of X. Schul & Y. Smal (180428)	54 (31)	23/31	48	2575.11	1.63	47.69
8	MiniCPFever (180429)	54 (31)	34/20	48	7928.71	3.01	146.83
9	slowpoke (180429)	38 (22)	38/0	34	1238.28	1.93	32.59
10	SuperSolver M. Stevenart (180427)	31 (18)	31/0	27	1726.51	1.64	55.69
11	The dodo solver (180429)	25 (14)	0/25	22	5214.63	12.89	208.59

TABLE 6.5 – MiniTrack – CSP: SAT+UNSAT answers

personnalisée. Après chaque réduction, la limite de taille est augmentée de 500 (une politique SAT classique).

6.2 Compétition XCSP3 2018

Nous avons soumis NACRE 1.0.4 (Glorian (2019)) à la compétition 2018 XCSP3 (Lecoutre et Roussel (2019)) dans la catégorie CSP – MiniTrack. Cette catégorie est conçue pour les logiciels *open-source* séquentiels et permet aux développeurs de solveurs de programmation par contraintes d'entrer dans la compétition sans avoir à implémenter toutes les contraintes du *XCSP3-core* (une collection de 21 contraintes globales). Les solveurs sont évalués dans cette catégorie sur un ensemble restreint de contraintes : *intension*, *extension*, *allDifferent*, *sum* et *element*. Les solveurs ont été exécutés sur un cluster de Xeon@2.67GHz avec 32 Go de mémoire. 15500 Mo de mémoire

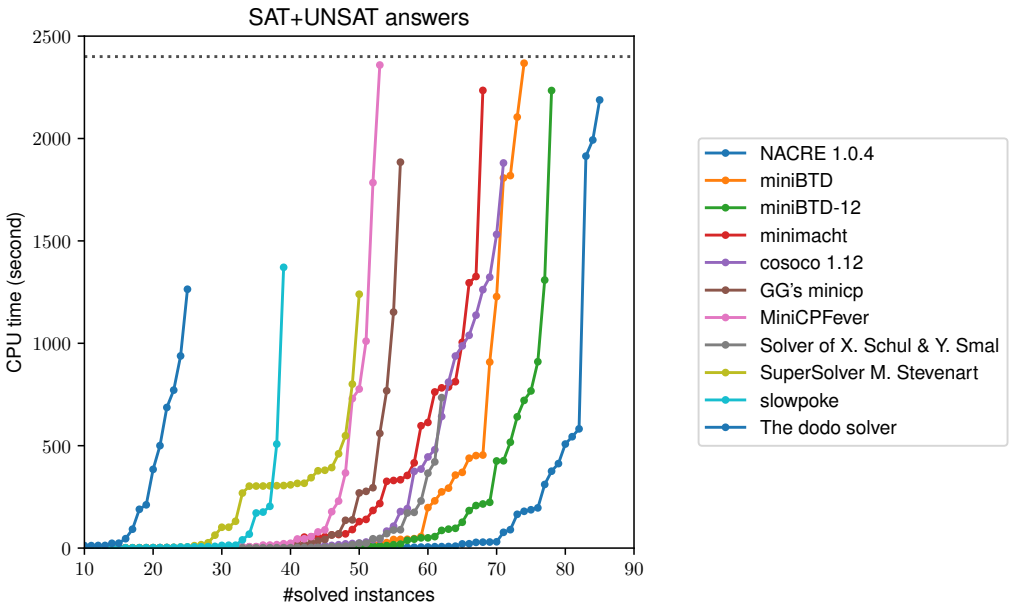


FIGURE 6.4 – MiniTrack – CSP: SAT+UNSAT answers cactus plot

Rank	Solver (Version)	#solved(%)	% VBS	SumCPU	MedCPU	AvgCPU
	Virtual Best Solver (VBS)	53 (30)	100	7533.70	0.70	142.15
1	NACRE (1.0.4)	43 (24)	81	3468.77	0.55	80.67
2	cosoco (1.12)	42 (24)	79	8946.84	0.80	213.02
3	slowpoke (180429)	38 (22)	72	1238.28	1.93	32.59
4	GG's minicp (180429)	37 (21)	70	2024.12	1.90	54.71
5	minimacht (180727)	37 (21)	70	5083.68	1.92	137.40
6	miniBTD_12 (180727_12)	36 (20)	68	3984.00	1.18	110.67
7	MiniCPFever (180429)	34 (19)	64	5634.13	1.99	165.71
8	miniBTD (180727_3)	32 (18)	60	5715.80	1.83	178.62
9	SuperSolver M. Stevenart (180427)	31 (18)	58	1726.51	1.64	55.69
10	Solver of X. Schul & Y. Smal (180428)	23 (13)	43	1006.31	3.05	43.75

TABLE 6.6 – MiniTrack – CSP: SAT answers

ont été attribués aux solveurs séquentiels et le *timeout* de la catégorie était de 2400 secondes (40 minutes). La CSP – MiniTrack était composée d'une sélection de 176 instances.

NACRE 1.0.4 a été soumis avec les options suivantes : `-ca -luby100` et `-cm`. Pour rappel, cela active notre méthode d'analyse de conflits et utilise la séquence de Luby (avec chaque terme multiplié par 100) comme stratégie de redémarrage. La dernière option correspond au mode d'affichage compétition, ce qui signifie simplement que seuls le résultat et la solution (le cas échéant) sont affichés.

Les résultats de la compétition 2018 XCSP3 sont présentés ci-après. Tous les tableaux et graphiques de cactus sont générés à partir des données brutes [4] du site web officiel de la compétition [5].

4. www.cril.univ-artois.fr/XCSP18/results/export.php?idev=95
5. www.cril.univ-artois.fr/XCSP18/

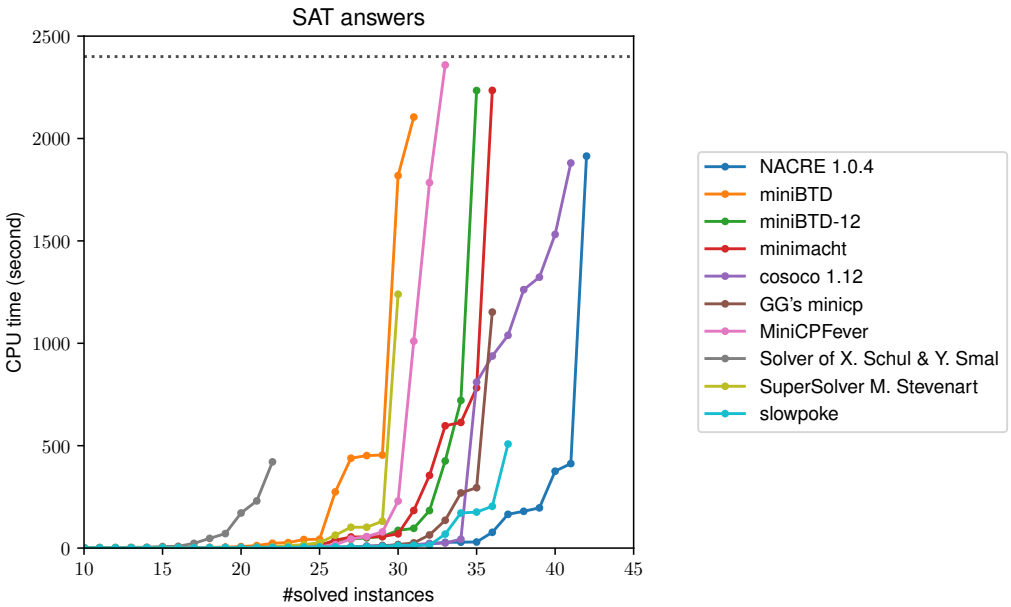


FIGURE 6.5 – MiniTrack – CSP: SAT answers cactus plot

Rank	Solver (Version)	#solved(%)	% VBS	SumCPU	MedCPU	AvgCPU
	Virtual Best Solver (VBS)	60 (34)	100	4365.56	0.62	72.76
1	miniBTD_12 (180727_12)	43 (24)	72	5550.56	0.76	129.08
2	NACRE (1.0.4)	43 (24)	72	6479.80	0.30	150.69
3	miniBTD (180727_3)	43 (24)	72	7964.02	0.76	185.21
4	minimacht (180727)	32 (18)	53	8215.62	58.96	256.74
5	Solver of X. Schul & Y. Smal (180428)	31 (18)	52	1568.80	1.26	50.61
6	cosoco (1.12)	30 (17)	50	5127.83	5.17	170.93
7	The dodo solver (180429)	25 (14)	42	5214.63	12.89	208.59
8	MiniCPFever (180429)	20 (11)	33	2294.57	10.70	114.73
9	GG's minicp (180429)	19 (11)	32	3837.20	12.74	201.96

TABLE 6.7 – MiniTrack – CSP: UNSAT answers

Les résultats sont divisés en trois catégories : les résultats *SAT* qui affichent des informations sur les instances satisfiables, les résultats *UNSAT* qui affichent des informations sur les instances insatisfiables et les résultats *SAT + UNSAT* qui regroupent les résultats précédents. Pour chaque catégorie, un tableau détaillé et un graphique sont affichés.

Dans les résultats globaux (figure 6.4 et tableau 6.5, `SAT`), NACRE a résolu 49% des 176 instances, tout en représentant 76% du *Virtual Best Solver* (`VBS`). Cela signifie que 76% des instances pouvant être résolues par au moins un solveur de la compétition sont résolues par NACRE 1.0.4. Nous pouvons également constater que NACRE 1.0.4 est performant à la fois sur les instances `SAT` et les instances `UNSAT`. Un indicateur intéressant à relever est le temps CPU médian : NACRE 1.0.4 obtient $0,40$ secondes alors que le deuxième meilleur solveur, *miniBTD_12*, obtient $0,88$ secondes. Nous pouvons aussi noter que le `VBS` obtient $0,62$ secondes.

Concernant uniquement les instances satisfiables (figure 6.5 et tableau 6.6, `SAT + UNSAT`),

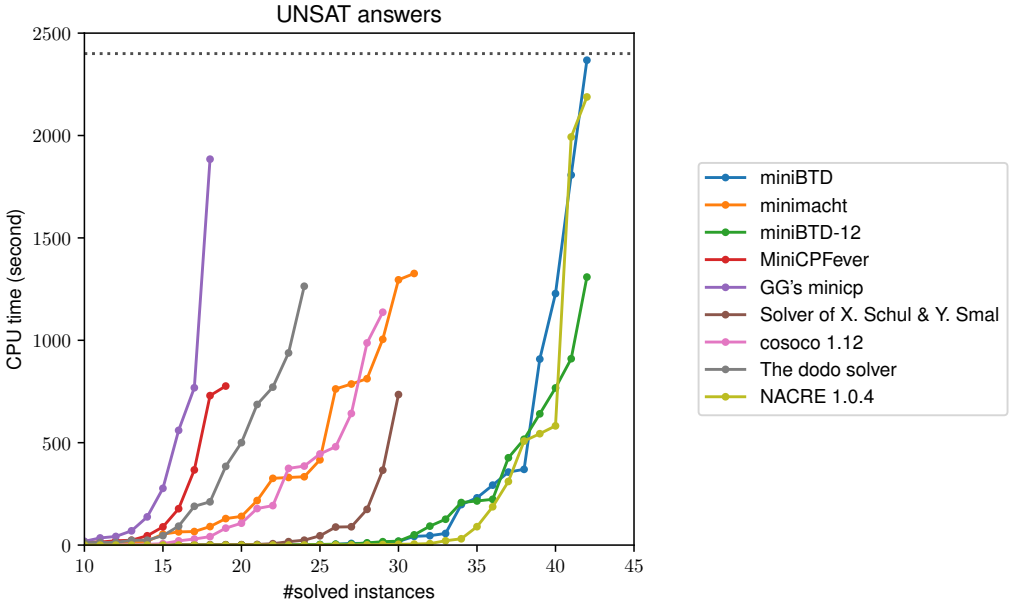


FIGURE 6.6 – MiniTrack – CSP: UNSAT answers cactus plot

nous pouvons voir que NACRE 1.0.4 a résolu 43 instances. La différence du nombre d'instances satisfiables résolues avec le deuxième solveur *cosoco* n'est que d'une instance mais le temps *CPU* cumulé place NACRE (3468, 77 secondes) largement devant *cosoco* (8946, 84 secondes).

Sur les instances insatisfiables (figure 6.6 et tableau 6.7, `UNSAT`), le même nombre d'instances sont résolues par NACRE 1.0.4 et *miniBTD_12*. Par contre, le temps de traitement cumulé place *miniBTD_12* devant. Néanmoins, en regardant le temps CPU médian (0, 3), nous pouvons voir que NACRE conserve un temps de résolution généralement rapide. Cela signifie que NACRE 1.0.4 a résolu beaucoup d'instances insatisfiables très rapidement.

Pour conclure sur l'ensemble des résultats de la compétition, il est clair que NACRE affiche de bonnes performances, à la fois sur les instances satisfiables et insatisfiables. Les instances sont généralement soit résolues très rapidement, soit pas du tout par NACRE, comme le montre le graphique `SAT + UNSAT` (figure 6.4) où la plupart des instances (sauf 3) sont résolues en moins de 600 secondes.

6.3 Conclusion

Dans ce chapitre, nous avons présenté NACRE 1.0.4, un moteur de raisonnement centré sur les *nogoods* et les clauses. Notre objectif principal était de fournir à la communauté un outil extensible, utile notamment pour expérimenter facilement sur les *nogoods* et les clauses. Nous avons aussi porté notre attention sur les performances du *framework* et nous l'avons confirmé en prenant la première place de la catégorie *CSP – Minitrack* à la compétition 2018 XCSP3 [6].

Pour la prochaine version majeure, nous prévoyons de prendre en compte toutes les contraintes du *XCSP3-core* afin de concourir dans la catégorie principale. Il pourrait également être intéressant d'implémenter un module d'optimisation afin de traiter ce type de problème avec les différentes approches. Dans le chapitre suivant, nous nous sommes appliqués à améliorer le moteur de NACRE.

6. NACRE a été soumis, dans la même catégorie, à la compétition de 2019 et a de nouveau remporté la première place

Chapitre 7

Hybridation

Sommaire

Afin d'améliorer le moteur de raisonnement, deux pistes s'offrent à nous. Proposer de nouveaux algorithmes de raisonnement pour différentes contraintes globales ; ou améliorer la qualité des clauses générées depuis les explications génériques. Nous avons choisi d'explorer ces deux pistes. Ce chapitre se découpe donc en deux sections. Celles-ci sont à la fois proches et opposées. En effet, elles prennent toutes deux comme base les explications paresseuses et l'hybridation. La différence réside dans le fait que dans la première section nous proposons des algorithmes *ad-hoc* et que dans la seconde nous améliorons l'approche générique en augmentant la qualité des clauses générées.

7.1 Raisonnement *ad-hoc* pour la contrainte *element*

Nous partons des travaux de Gent *et al.* (2010) sur les fonctions de raisonnement *ad-hoc* en vue de les étendre. En effet, nous proposons des fonctions qui permettent de décrire un conflit qui survient dans une contrainte globale *element*. Nous proposons aussi une mécanique qui permet d'obtenir la raison de la propagation d'un littéral au sein d'une de ces contraintes. Ces fonctions sont décrites à la fois pour la contrainte *element* avec un paramètre constant ou variable. La section est organisée comme suit : dans un premier temps nous décrivons ci-dessous la contrainte globale *element* ainsi que le théorème (Gent *et al.* (2006)) qui permet d'assurer sa

cohérence généralisée. Puis, nous précisons ce théorème dans le cas où le résultat est constant. Nous présentons ensuite le propagateur que nous utilisons pour la version constante et décrivons nos algorithmes de raisonnement. Le même processus est ensuite appliqué à la version « variable ».

La contrainte globale *element* (Hentenryck et Carillon (1988)) permet d'utiliser une variable comme indice d'un tableau afin de vérifier une égalité. En effet, elle est composée de trois parties, la variable dite *Index*, un tableau de variables (ou de vecteurs de valeurs) que nous appelons A et une variable (voire une constante) *Result*. Le but de cette contrainte globale est de vérifier $A[Index] = Result$.

Théorème 4 (Gent *et al.* (2006))

Une contrainte $element(Index, A, Result)$ est GAC si et seulement si :

- $\texttt{dom}(Index) = \{i\} \Rightarrow \texttt{dom}(A[i]) \subseteq \texttt{dom}(Result)$
- $i \in \texttt{dom}(Index) \Rightarrow \texttt{dom}(A[i]) \cap \texttt{dom}(Result) \neq \emptyset$
- $\texttt{dom}(Result) \subseteq \bigcup_{i \in \texttt{dom}(Index)} \texttt{dom}(A[i])$

Plusieurs cas particuliers de la contrainte *element* existent. Notamment lorsque la variable *Index* et/ou la variable *Result* appartiennent à A. Les algorithmes présentés dans cette section peuvent traiter certains de ces cas particuliers. Toutefois, par soucis de simplicité, nous considérerons que les variables *Index* et *Result* sont différentes et n'appartiennent pas au tableau A.

Remarque 13

Nous supposons que pour les algorithmes proposés dans cette section, nous n'ajoutons pas à l'ensemble retourné (cl) les littéraux dont le niveau de propagation est 0 ainsi que ceux dont l'ordre de propagation est supérieur au littéral dont la raison est demandée (l). Cela permet de réduire la taille des définitions des ensembles au sein des algorithmes 7.2, 7.3, 7.5 et 7.6.

7.1.1 *Element* version « constant »

Nous allons, dans un premier temps, nous intéresser au cas particulier où *Result* est une constante. Afin d'effectuer le filtrage nous simplifions le théorème 4 pour ce cas et utilisons le propagateur décrit dans l'algorithme 7.1.

Corollaire 1

Les variables composants une contrainte *element* où *Result* est une constante sont GAC si et seulement si :

— $\texttt{dom}(Index) = \{i\} \Rightarrow A[i] = Result$

— $i \in \texttt{dom}(Index) \Rightarrow Result \in \texttt{dom}(A[i])$

Preuve. Considérons les règles du théorème 4 :

— $\texttt{dom}(Index) = \{i\} \Rightarrow \texttt{dom}(A[i]) \subseteq \texttt{dom}(Result)$

— $i \in \texttt{dom}(Index) \Rightarrow \texttt{dom}(A[i]) \cap \texttt{dom}(Result) \neq \emptyset$

— $\texttt{dom}(Result) \subseteq \bigcup_{i \in \texttt{dom}(Index)} \texttt{dom}(A[i])$

Si *Result* est une constante, nous pouvons considérer que c'est un ensemble singleton $\{r\}$. Nous obtenons alors les règles suivantes :

— $\texttt{dom}(Index) = \{i\} \Rightarrow \texttt{dom}(A[i]) \subseteq \{r\}$

— $i \in \texttt{dom}(Index) \Rightarrow \texttt{dom}(A[i]) \cap \{r\} \neq \emptyset$

— $\{r\} \subseteq \bigcup_{i \in \texttt{dom}(Index)} \texttt{dom}(A[i])$

Nous pouvons voir que $\texttt{dom}(A[i]) \subseteq \{r\}$ peut se réécrire $A[i] = Result$ de manière triviale. De même pour la deuxième règle où $\texttt{dom}(A[i]) \cap \{r\} \neq \emptyset$ peut se réécrire $Result \in \texttt{dom}(A[i])$ dans ce cas. La troisième règle devient clairement redondante avec la deuxième pour ce cas particulier, nous pouvons alors la supprimer.

q.e.d □

L'algorithme 7.1 applique les règles décrites par le corollaire 1. Les lignes 1 à 7 traite la seconde règle. En effet, cette règle assure que pour chaque valeur i du domaine de la variable *Index* (ligne 2), la constante *Result* doit figurer dans le domaine de la variable associée du tableau A : $A[i]$ (ligne 3). Si ce n'est pas le cas, la valeur est supprimée dans le domaine de la variable *Index* (ligne 7). Évidemment, nous vérifions si la suppression n'implique pas de conflits (lignes 4-5). Les lignes 8 à 13 appliquent la première règle. Si la variable *Index* est assignée, il faut s'assurer que la variable pointée par l'unique valeur du domaine de l'*Index* soit assignée à la constante *Result* (ligne 11). Si l'assignation est impossible, un conflit est découvert.

7.1.1.1 Explication d'un conflit

Afin de décrire un conflit arrivant sur une contrainte, la méthode générique présentée au chapitre 6 prend en compte une grande partie des valeurs supprimées dans le domaine des variables de la portée de la contrainte considérée. Nous avons vu au chapitre 4 qu'il était possible de fournir des explications (raisons) de propagation de littéraux de manière *ad-hoc*. En s'inspirant de cela, nous proposons ici un algorithme permettant de décrire de manière concise un conflit causé par une contrainte globale *element* constante.

Algorithme 7.1 : *ElementConstantPropagator*()

```
   Result : true if conflict found ; false otherwise
 1 if |dom(Index)| > 1 then
 2     foreach i ∈ dom(Index) do
 3         if Result ∉ A[i] then
 4             if |dom(Index)| = 1 then
 5                 return true;
 6             else
 7                 Remove i from Index;
 8 if |dom(Index)| = 1 then
 9     Let i be the only value left in dom(Index);
10     if Result ∈ A[i] then
11         Assign A[i] to Result;
12     else
13         return true;
14 return false;
```

La première question à se poser, lorsque nous souhaitons construire un tel algorithme, est la suivante : *Comment l'algorithme de filtrage de la contrainte considérée peut entrer en conflit ?* Nous pouvons répondre à cette question en analysant le propagateur associé à la contrainte. Dans notre cas, celui-ci est décrit dans l'algorithme 7.1. Les lignes retournant *true* nous indiquent les situation de conflits. Nous pouvons en identifier deux, les lignes 5 et 13.

Le premier cas (ligne 5) survient lorsque le domaine de la variable *Index* va être vidé intégralement. Les valeurs sont toutes supprimées pour *Index* car dans tous les domaines des variables pointées dans le tableau A, la valeur *Result* a été supprimée. Ces valeurs sont donc importantes. Le second cas (ligne 13) est lié à l'affectation de la variable *Index*. En effet, par définition, lorsque la variable *Index* est affectée, la variable pointée dans le tableau A doit contenir la valeur *Result*. Si ce n'est pas le cas, un conflit survient ; cela signifie que les valeurs supprimées pour la variable *Index* sont pertinentes pour décrire le conflit.

Pour résumer, l'explication d'un conflit sur une contrainte *element* constante doit contenir :

— (i) toutes les valeurs supprimées dans la variable *Index* ;

— (ii) les valeurs *Result* des variables $A[i]$ dont l'indice i apparaît dans les valeurs restantes du domaine de la variable *Index*.

Algorithme 7.2 : *ElementConstantConflict*()

```
   Result : A set of literals that explain the conflict
 1 cl ← {lit(i) | i ∈ dom^init(Index) \ dom(Index)};
 2 foreach i ∈ dom(Index) do
 3     cl ← cl ∪ {lit(a) | a ∈ dom(A[i])^init \ dom(A[i]) ∧ a = Result};
 4 return cl;
```

L'algorithme 7.2 implémente les cas décrits ci-dessus. La ligne 1 correspond à la partie (i) et

les lignes 2–3 au cas (*ii*).

Exemple 28

Soit la contrainte *element* constante avec $Result = 2$ composée d'un tableau A de 3 variables et soient les domaines suivants :

- $dom(Index) = \{0, 1, 2\}$
- $dom(A[0]) = \{0, 1, 2, 3\}$
- $dom(A[1]) = \{0, 1, 2, 3\}$
- $dom(A[2]) = \{0, 1, 3, 4\}$

Supposons la suite de décisions et les propagations suivantes (provenant à la fois de la contrainte *element* et d'autres contraintes non décrites par soucis de simplicité) :

Niveau 1 : $[A[2] = 0]$, $A[1] \neq 1$, $A[1] \neq 3$

Etat :

- $dom(Index) = \{0, 1, 2\}$
- $dom(A[0]) = \{0, 1, 2, 3\}$
- $dom(A[1]) = \{0, \not{1}, 2, \not{3}\}$
- $dom(A[2]) = \{0, \not{1}, \not{3}, \not{4}\}$

Niveau 2 : $[A[0] = 1]$, $A[1] \neq 2$, $Index = 1$

Etat :

- $dom(Index) = \{\not{0}, 1, \not{2}\}$
- $dom(A[0]) = \{\not{0}, 1, \not{2}, \not{3}\}$
- $dom(A[1]) = \{0, \not{1}, \not{2}, \not{3}\}$
- $dom(A[2]) = \{0, \not{1}, \not{3}, \not{4}\}$

Lorsque le propagateur de la contrainte *element* est appelé, la variable *Index* étant assignée à 1, et le domaine de la variable $A[1]$ ne contenant plus la valeur 2, un conflit va être découvert et une analyse de conflits lancée.

Dans le cas de la fonction d'explication de conflits générique (algorithme 6.4 du chapitre 6), nous avons : $Index \not\leftarrow 0, Index \not\leftarrow 2, A[0] \not\leftarrow 0, A[0] \not\leftarrow 2, A[0] \not\leftarrow 3, A[1] \not\leftarrow 1, A[1] \not\leftarrow 2, A[1] \not\leftarrow 3, A[2] \not\leftarrow 1, A[2] \not\leftarrow 3, A[2] \not\leftarrow 4$.

Considérons maintenant l'explication retournée par l'algorithme 7.2 : $Index \not\leftarrow 0, Index \not\leftarrow 2, A[1] \not\leftarrow 2$. Cette explication est clairement plus précise (et beaucoup plus concise). Elle peut, de plus, se lire aisément de la manière suivante : *nous obtenons un conflit sur la contrainte car la variable Index est assignée à 1 et la variable A[1] ne peut pas être égale à 2 (la valeur de Result).*

7.1.1.2 Raison d'un littéral propagé

Afin de créer un raisonneur sur un propagateur quelconque, il faut identifier les cas de modifications des domaines des variables de la portée de la contrainte considérée (cela peut s'apparenter à un *reverse-engineering* de l'algorithme de filtrage de la contrainte). Dans le cas de la contrainte globale *element*, et plus particulièrement dans le cas particulier où *Result* est une constante, nous pouvons identifier deux cas dans l'algorithme 7.1. Le premier cas est la suppression de valeurs pour la variable *Index*. Ces suppressions surviennent lorsque le domaine d'une variable pointée par le domaine de la variable *Index* dans le tableau A ne contient plus la valeur *Result*. Le second cas est l'assignation d'une variable du tableau A à la valeur *Result* lorsque la variable *Index* est assignée.

Algorithme 7.3 : $ElementConstantReason(l)$

Data : l a literal
Result : The reason of l as a set of literals
1 **if** $var(l) = Index$ **then**
2 $\quad cl \leftarrow \{lit(a) \mid a \in \texttt{dom}(A[val(l)])^{init} \setminus \texttt{dom}(A[val(l)]) \wedge a = Result\}$;
3 **else**
4 $\quad$ Let i be the only value left in $\texttt{dom}(Index)$;
5 $\quad cl \leftarrow \{\neg lit(i)\}$;
6 **return** cl;

L'algorithme 7.3 est composé des deux cas identifiés précédemment. Si la variable associée au littéral (récupérée par $var(l)$) dont la raison est demandée est la variable *Index* (ligne 1) alors la raison sera le littéral de valeur *Result* du domaine de la variable du tableau A pointée par la valeur du littéral (récupérée par $val(l)$) dans *Index*. Au contraire, si la variable du littéral dont la raison est requise est différente de *Index*, c'est-à-dire appartient au tableau A, alors il suffit de considérer la négation du littéral construit à partir de la valeur affectée à la variable *Index*. En effet, la variable *Index* dans ce cas est forcément affectée (ligne 8 à 13 de l'algorithme 7.1).

Exemple 29

Soit la contrainte *element* constante avec $Result = 1$ composée d'un tableau A de 2 variables et soient les domaines suivants :

- $dom(Index) = \{0, 1\}$
- $dom(A[0]) = \{0, 1, 2\}$
- $dom(A[1]) = \{0, 1, 2\}$

Et soit l'état courant :

- $dom(Index) = \{0, 1\}$
- $dom(A[0]) = \{\not{0}, \not{1}, 2\}$
- $dom(A[1]) = \{0, 1, \not{2}\}$

L'appel au propagateur (algorithme 7.1) va dans un premier temps parcourir les valeurs de la variable *Index* et par conséquent réduire son domaine au singleton $\{1\}$ car la valeur

de *Result* (1) est supprimée pour la variable $A[0]$. Puis, comme la variable *Index* est maintenant assignée, la seconde partie de l'algorithme de filtrage va assigner la variable $A[1]$ à la valeur de *Result* (1) afin de satisfaire la contrainte.

L'état des domaines des variables qui composent la contrainte est, après filtrage, le suivant :

- $dom(Index) = \{\not{0}, 1\}$
- $dom(A[0]) = \{\not{0}, \not{1}, 2\}$
- $dom(A[1]) = \{\not{0}, 1, \not{2}\}$

Supposons maintenant que nous souhaitons obtenir les explications des modifications des domaines des variables de la contrainte depuis l'état précédent grâce à l'algorithme 7.3 (et aux explications paresseuses pour les explications de domaines). Nous obtenons :

- $\texttt{expl}(A[1] \nleftarrow 0) = A[1] \leftarrow 1$ (domaine – *atMostOne*)
- $\texttt{expl}(A[1] \leftarrow 1) = Index \leftarrow 1$ (*element* constante)
- $\texttt{expl}(Index \leftarrow 1) = Index \nleftarrow 0$ (domaine – *atLeastOne*)
- $\texttt{expl}(Index \nleftarrow 0) = A[0] \nleftarrow 1$ (*element* constante)

7.1.2 *Element* Variable

Pour le cas général de la contrainte *element*, l'algorithme de filtrage et les fonctions de raisonnement doivent être modifiés. Cette section présente une première version afin de raisonner avec la contrainte globale *element* variable. Présentons dans un premier temps le propagateur que nous utilisons. Cet algorithme de propagation utilise deux listes de sentinelles afin de réaliser le filtrage. La première liste, dénommée `resultSentinels`, est définie comme ceci : pour chaque valeur v dans le domaine de la variable *Result*, nous sauvegardons l'indice i d'une variable du tableau A tel que v est dans le domaine de $A[i]$. Cette liste nous permet de vérifier rapidement que chaque valeur du domaine de la variable *Result* a un support dans les variables du tableau A. La seconde liste de sentinelles, dénommée `vectorSentinels`, concerne les variables du tableau A. Pour chacune des variables dans ce tableau, nous sauvegardons dans la liste une valeur qui est à la fois dans le domaine de la variable et dans le domaine de *Result*. Cela permet de s'assurer efficacement que la variable du tableau A est toujours cohérente.

Comme précédemment pour la version constante, l'algorithme se découpe en deux parties. La première partie (ligne 1 à 15) est exécutée seulement si la variable *Index* n'est pas assignée. Lorsque plusieurs valeurs sont encore disponibles dans le domaine de la variable *Index*, nous allons essayer de supprimer des valeurs à la fois dans le domaine de la variable *Index* elle-même et dans le domaine de la variable *Result*. Concernant le domaine de la variable *Result*, nous supprimons les valeurs qui n'apparaissent dans aucun domaine des variables du tableau A. Cela est réalisé (lignes 2 et 11) grâce à la fonction `reduceResultDomain` et au tableau de sentinelles `resultSentinels` (qui est mis à jour en conséquence). Ensuite nous essayons de supprimer des valeurs dans le domaine de la variable *Index*. En effet, nous supprimons au sein de la variable *Index* les valeurs v pour lesquelles il n'y a pas de valeurs j tel que $j \in A[v] \cap Result$. De la même manière que pour la variable *Result*, cela est réalisé (ligne 6) grâce à la fonction `reduceIndexDomain` et au

Algorithme 7.4 : *ElementVariablePropagator*()

```
   Result : true if conflict found ; false otherwise
 1 if |dom(Index)| > 1 then
 2     if reduceResultDomain() then
 3         if dom(Result) = ∅ then
 4             return true;
 5     while true do
 6         if reduceIndexDomain() then
 7             if dom(Index) = ∅ then
 8                 return true;
 9             else
10                 break;
11         if reduceResultDomain() then
12             if dom(Result) = ∅ then
13                 return true;
14             else
15                 break;
16 if |dom(Index)| = 1 then
17     Let i be the only value left in dom(Index);
18     if dom(A[i]) ∩ dom(Result) ≠ ∅ then
19         interDom ← dom(A[i]) ∩ dom(Result);
20         dom(A[i]) ← interDom;
21         dom(Result) ← interDom;
22     else
23         return true;
24 return false;
```

tableau de sentinelles `vectorSentinels` (qui est mis à jour en conséquence). Ces deux fonctions sont appelées jusqu'à ce qu'un conflit soit découvert par domaine vide ou jusqu'à ce qu'une des deux variables (*Index* ou *Result*) ne subisse pas de réduction de domaine. Nous considérons alors qu'elles sont filtrées pour cette partie. La seconde partie (lignes 16 à 23) est effectuée si la variable *Index* est assignée. Il suffit de modifier le domaine de la variable *Result* et de la variable dans le tableau A pointée par la valeur affectée à *Index*. Les deux domaines vont être forcés à devenir égaux à leur intersection afin d'être cohérents. Si l'intersection est vide, un conflit est découvert.

7.1.2.1 Explication d'un conflit

L'explication d'un conflit, même dans le cas général, reste assez simple à déduire en ayant les règles conflictuelles du propagateur. En effet, dans l'algorithme de filtrage 7.4 nous pouvons identifier quatre lignes où un conflit est détecté. Lorsque les variables *Index* (ligne 8) et *Result* (lignes 4 et 13) sont réduites à un domaine vide et lorsque le domaine de la variable *Result* et de la variable dans le tableau A pointée par la valeur affectée à *Index* deviennent égaux à leur

intersection (ligne 23) et que celle-ci est vide. Selon un raisonnement semblable à celui de la contrainte *element* « constante », l'explication de la version « variable » doit contenir :

— (i) toutes les valeurs supprimées dans la variable *Index* ;

— (ii) toutes les valeurs supprimées dans la variable *Result* ;

— (iii) les valeurs des variables $A[i]$, qui apparaissent dans le domaine de la variable *Result*, dont l'indice i apparaît dans les valeurs restantes du domaine de la variable *Index*.

L'algorithme 7.5 implémente ces cas. Les lignes 1 et 2 traitent respectivement les cas (i) et (ii) en considérant les valeurs supprimées des domaines des variables *Index* et *Result*. La boucle (lignes 3 à 4) traite le dernier cas (iii) ; pour chaque valeur encore dans le domaine de la variable *Index* les valeurs supprimées de la variable associée du tableau A appartenant aux valeurs encore valides de la variable *Result* sont ajoutées à l'ensemble de littéraux décrivant le conflit.

Algorithme 7.5 : $ElementVariableConflict()$

Result : A set of literals that explain the conflict

1 $cl \leftarrow \{lit(i) \mid i \in \texttt{dom}^{init}(Index) \setminus \texttt{dom}(Index)\}$;

2 $cl \leftarrow cl \cup \{lit(i) \mid i \in \texttt{dom}^{init}(Result) \setminus \texttt{dom}(Result)\}$;

3 **foreach** $i \in \texttt{dom}(Index)$ **do**

4 $\quad cl \leftarrow cl \cup \{lit(a) \mid a \in \texttt{dom}(A[i])^{init} \setminus \texttt{dom}(A[i]) \wedge a \in dom(Result)\}$;

5 **return** cl;

7.1.2.2 Raison d'un littéral propagé

Donner la raison d'un littéral de manière concise est un peu plus délicat dans le cas de la contrainte *element* variable. En effet, le propagateur de la contrainte *element* variable est plus complexe que celui de la version constante. Effectuer le « *reverse-engineering* » et identifier les cas est bien sûr faisable, la difficulté réside dans l'identification précise des valeurs utiles. L'explication d'un littéral peut être demandée pour toutes les variables de la portée de la contrainte car l'algorithme de filtrage 7.4 peut modifier les variables *Index* et *Result* dans sa première partie et à nouveau la variable *Result* ainsi qu'une des variables du tableau A si la variable *Index* venait à être assignée. Cela nous apprend que trois cas sont possibles. Nous les détaillons dans la suite en pointant les lignes associées dans l'implémentation au sein de l'algorithme 7.6.

Index La variable du littéral l dont on veut la raison est la variable *Index*. Cela induit que nous voulons la raison de la suppression d'une valeur du domaine de la variable *Index*. Le domaine est modifié uniquement au sein de la fonction `reduceIndexDomain` (ligne 6 de l'algorithme 7.4) en suivant la règle suivante : nous supprimons au sein de la variable *Index* les valeurs v pour lesquelles il n'y a pas de valeurs j tel que $j \in A[v] \cap Result$. Nous devons donc considérer toutes les valeurs supprimées dans le domaine de la variable $A[i]$ avec $i = val(l)$ qui apparaissent dans le domaine initial de la variable *Result* (qui ont été propagées avant l). De plus, cela doit être fait dans l'autre sens, c'est-à-dire que nous devons prendre en compte les valeurs supprimées du domaine de la variable *Result* qui apparaissent dans le domaine initial de la variable $A[i]$.

Result La variable du littéral l dont on veut la raison est la variable *Result*. Une valeur est supprimée du domaine de la variable *Result* lorsqu'elle n'apparaît dans aucun domaine des variables du tableau A. Cela implique deux parties. La première concerne les valeurs supprimées

Algorithme 7.6 : $ElementVariableReason(l)$

Data : l a literal
Result : The reason of l as a set of literals

1 **if** $var(l) = Index$ **then**
2 $\quad cl \leftarrow \{lit(a) \mid a \in \texttt{dom}(A[val(l)])^{init} \setminus \texttt{dom}(A[val(l)]) \wedge a \in \texttt{dom}^{init}(Result)\}$;
3 $\quad cl \leftarrow cl \cup \{lit(r) \mid r \in \texttt{dom}(Result)^{init} \setminus \texttt{dom}(Result) \wedge r \in \texttt{dom}^{init}(A[val(l)])\}$;
4 **else if** $var(l) = Result$ **then**
5 $\quad cl \leftarrow \{lit(i) \mid i \in \texttt{dom}^{init}(Index) \setminus \texttt{dom}(Index)\}$;
6 $\quad cl \leftarrow cl \cup \{lit(a) \mid a \in \texttt{dom}(A[i])^{init} \setminus \texttt{dom}(A[i]), \forall i \in \texttt{dom}^{init}(Index) \wedge a = val(l)\}$;
7 **else**
8 $\quad$ Let i be the only value left in $\texttt{dom}(Index)$;
9 $\quad cl \leftarrow \{\neg lit(i)\}$;
10 $\quad$ **if** l *is negative* **then**
11 $\quad\quad cl \leftarrow cl \cup \{lit(r) \mid r \in \texttt{dom}(Result)^{init} \setminus \texttt{dom}(Result) \wedge r = val(l)\}$;
12 $\quad$ **else**
13 $\quad\quad cl \leftarrow cl \cup \{\neg lit(r) \mid r \in \texttt{dom}(Result) \wedge r = val(l)\}$;
14 **return** cl;

dans le domaine de la variable *Index* que nous devons considérer (ligne 5). En effet, si ces valeurs n'avaient pas été supprimées, $val(l)$ aurait possiblement été valide. La seconde partie permet de prendre en compte les valeurs dans les domaines des variables du tableau A qui avaient permis à l de rester valide (ligne 6).

Tableau A La variable du littéral l dont on veut la raison appartient au tableau A. Ce cas est plus facile que les précédents. En effet, le domaine d'une variable du tableau A ne peut être modifié que si la variable *Index* est affectée à une valeur qui pointe vers celle-ci. Nous devons donc considérer la valeur assignée à la variable *Index* (ligne 9). Ensuite un second littéral est requis dans le domaine de la variable *Result*. Deux cas sont possibles selon le signe de l. Lorsque la variable *Index* est réduite à un singleton $\{i\}$ alors les domaines de *Result* et de $A[i]$ sont réduits à leur intersection. Cela signifie que les valeurs supprimées de chacune des variables qui étaient communes dans leur domaine initial seront retirées dans le domaine de l'autre. Si l a un signe négatif (c'est-à-dire que nous voulons la raison de $\neg l$, autrement dit, de la suppression d'une valeur) alors nous allons chercher dans les valeurs supprimées de la variable *Result* celle qui est égale à la valeur du littéral dont on veut la raison (ligne 11). Au contraire, si l a un signe positif, nous regardons dans les valeurs encore valides de la variable *Result* (ligne 13).

7.1.3 Validation expérimentale

Afin de valider expérimentalement les algorithmes 7.2 et 7.3, nous les avons implémentés au sein de NACRE 1.0.4. Nous avons effectué cette expérimentation sur un ordinateur équipé d'un processeur *Xeon* de 4 cœurs cadencé à $3,3$ GHz et équipé de 64 Go de RAM. Nous nous sommes inspirés de la configuration compétition et nous avons limité la mémoire disponible à $15,5$ Go et le temps imparti (*timeout*) est réglé à 40 minutes (2400 secondes). Les instances utilisées proviennent des familles *Langford* et *Quasigroup*. Ce sont toutes les instances CSP disponibles,

qui contiennent des contraintes *element*, dans la base XCSP3. Cela forme un ensemble de 181 instances de problèmes décomposé de la manière suivante : 33 instances de la famille *Langford* et 148 de la famille *Quasigroup*. Cette dernière se découpe en 4 sous-familles qui contiennent chacune 37 instances. NACRE est configuré pour utiliser l'analyse de conflits et l'heuristique de choix de variable *dom/wdeg* ainsi que la politique de redémarrage *Luby* × 100.

	Générique		Ad-hoc	
	SAT	UNS	SAT	UNS
Langford	12	3	12	**5**
Quasigroup-all	10	15	**11**	15
Quasigroup-3	2	3	**3**	3
Quasigroup-4	2	4	2	4
Quasigroup-5	4	4	4	4
Quasigroup-7	2	4	2	4
Total	22	18	**23**	**20**
Total par méthode	40		**43**	

TABLE 7.1 – Validation expérimentale – *element* constante – Nombre d'instances résolues en 2400 secondes

Il est clair, sur le tableau 7.1 que la méthode dédiée est plus efficace que l'algorithme générique (8% sur les instances utilisées). Concernant les algorithmes préliminaires 7.5 et 7.6, la validation expérimentale est plus délicate. En effet, au sein des instances disponibles, seules les sous-familles *Quasigroup-5* et *Quasigroup-7* contiennent la version variable de la contrainte globale *element*. Le passage à l'échelle pour ces instances ne nous permet pas de résoudre plus d'instances (en utilisant les algorithmes 7.5 et 7.6 couplés aux algorithmes 7.2 et 7.3) que la méthode *ad-hoc* pour la version constante de la contrainte globale seule.

7.1.4 Discussion

Dans cette section, nous avons proposé des algorithmes de raisonnement pour la version constante de la contrainte globale *element* ; l'un afin de décrire un conflit, l'autre permettant de donner la raison d'un littéral. Ces algorithmes sont prouvés efficaces en pratique. Ils permettent une amélioration de la robustesse d'un système hybride basé sur les explications paresseuses génériques, à savoir NACRE. Concernant la version variable de la contrainte globale *element*, nous avons proposé une version préliminaire des mêmes types d'algorithmes. Il n'a malheureusement pas encore été possible de les valider expérimentalement.

Afin d'améliorer les algorithmes de cette dernière version, nous pensons qu'il est encore possible d'augmenter la précision de l'explication d'un conflit. Une piste pour cela est de considérer les cas (i) et (ii) raffinés suivants au lieu des cas détaillés section 7.1.2.1 (le cas (iii) reste le même) :

— (i) toutes les valeurs i supprimées dans la variable *Index* dont les $A[i]$ associées ont des valeurs a de leur domaine apparaissant dans celui de la variable *Result* ;

— (ii) toutes les valeurs supprimées dans la variable *Result* qui apparaissent uniquement dans les domaines des $A[i]$;

— (*iii*) les valeurs des variables $A[i]$, qui apparaissent dans le domaine de la variable *Result*, dont l'indice i apparaît dans les valeurs restantes du domaine de la variable *Index*.

Cette piste mérite d'être explorée afin d'obtenir une raison plus fine (et donc potentiellement plus efficace) pour le cas général de la contrainte *element*.

Une autre perspective intéressante est de considérer une nouvelle forme de la contrainte *element* où A est un tableau d'entier. Ce type de contrainte globale est, généralement, traduit en contraintes binaires en pratique. Il pourrait être intéressant d'avoir un propagateur ainsi que des fonctions de raisonnement dédiés à ce cas.

7.2 Pouvons-nous faire confiance aux clauses ?

Au sein d'un système hybride générique comme NACRE, la qualité des clauses peut, selon les problèmes, se dégrader. En effet, une génération paresseuse de clauses générique ne permet pas d'avoir les meilleures explications pour un conflit donné. Le principal point négatif étant la taille des clauses. Il est bien connu que des clauses trop grandes sont moins efficaces. De plus, si certaines clauses sont approximatives, elle peuvent dérègler les heuristiques de sélection de variables (voire de littéraux) qui les exploitent (VSIDS, par exemple).

Nous avons vu que l'un des points forts du problème de satisfaction de contraintes est sa structure. La majorité des méthodes hybrides de l'état de l'art en font abstraction. Nous estimons qu'il est primordial, au sein d'un système hybride SAT/CP, d'exploiter la structure CP du problème afin d'améliorer la résolution.

Dans cette section, nous présentons une heuristique, basée sur VSIDS, qui jauge de l'utilité des clauses dynamiquement afin de savoir si nous devons exploiter plus ou moins les clauses apprises et plus généralement la partie SAT du système hybride. Nous présentons ensuite des méthodes de minimisation afin d'augmenter la pertinence des clauses et de corriger le principal problème d'une méthode générique, à savoir la taille des clauses obtenues.

7.2.1 Une heuristique de recherche pertinente

Comme nous avons pu le voir lors du chapitre 5, les solveurs CSP ne sont pas initialement conçus pour traiter les clauses (ou anticiper des conflits). Ainsi, une recherche heuristique classique telle que dom/wdeg (Boussemart *et al.* (2004)) peut être impactée par le fait que certains conflits sont déclenchés par des clauses. Dans un tel cas, le solveur peut se tromper car les structures associées à l'heuristique ne sont pas bien mises à jour. C'est la raison pour laquelle nous proposons une version modifiée de l'heuristique VSIDS (Moskewicz *et al.* (2001)) (présentée dans le chapitre 3) pour sélectionner directement le couple suivant (x, a) (un littéral) composé d'une variable x et d'une valeur $a \in dom(x)$.

L'heuristique VSIDS peut être vue comme une fonction d'ordonnancement (*ranking*) qui maintient un score pour chaque littéral associé à un couple variable-valeur (x, a). Le score d'une paire (x, a) est augmenté chaque fois qu'il apparaît dans une clause nouvellement apprise (lorsqu'un conflit a lieu) ou lorsqu'il est remplacé par son explication lors de l'analyse du conflit. La paire (x, a) qui maximise ce score est donc sélectionnée pour la décision suivante. Il est usuel, dans les solveurs SAT, d'avoir une heuristique de polarité auxiliaire séparée, appelée *progress saving* (Pipatsrisawat et Darwiche (2007)), qui décide si la variable x doit être affectée à a ou différente de a.

Dans notre cas, en raison du grain grossier de la procédure générique utilisée pour générer des explications de contraintes, lorsque l'arité est élevée et/ou les domaines sont étendus, il est probable que le score de nombreuses paires qui ne sont pas vraiment responsables des conflits augmente considérablement. De plus, l'heuristique *progress saving* peut également être erronée car, par construction, seuls des littéraux négatifs peuvent apparaître dans des explications de contrainte (et donc, il existe un biais en faveur de la polarité négative).

Pour résoudre ce problème, nous proposons une mesure afin de décider si nous pouvons faire confiance aux clauses générées afin de guider la recherche. Lorsqu'une majorité d'explications de contrainte (et donc d'une minorité d'explications de clause) sont impliquées dans les analyses de conflits, nous proposons de considérer uniquement la variable x de la paire (x, a) sélectionnée par l'heuristique VSIDS. Plus précisément, tous les t conflits nous calculons le ratio r entre le nombre de clauses et le nombre de contraintes utilisées comme explications lors des dernières t analyses de conflits. Si le ratio r est supérieur à un seuil prédéfini $\mathcal{T}^{max}$, la paire sélectionnée est considérée comme un tout (un littéral) à affecter en suivant l'heuristique de polarité (*progress saving*). Sinon, seule la variable x est prise en compte. Ensuite, une valeur dans $\texttt{dom}(x)$ est sélectionnée par une heuristique CP indépendante de choix de valeurs (la valeur minimale pour nos expériences, voir section 7.2.3).

Nous utilisons également le ratio r pour ajuster la politique d'affectation lorsque l'apprentissage par clause ne semble pas très efficace. Plus précisément, le comportement attendu optimal est le suivant : plus la recherche progresse, plus le ratio r doit être grand (c'est-à-dire exploiter de plus en plus de clauses apprises utiles et précises). Ainsi, si le ratio r passe au dessous d'un seuil, défini dynamiquement par $\frac{\#conf}{t} \times \mathcal{T}^{min}$, avec $\#conf$ le nombre de conflits qui ont eu lieu depuis le début de la recherche, alors nous passons définitivement à l'heuristique VSIDS couplée à l'heuristique de polarité *progress saving*. Cela dans le but d'obtenir moins de conflits sur les contraintes et une meilleure utilisation des clauses apprises.

Cette mesure nous permet de jouer efficacement sur l'hybridation du solveur en fonction des problèmes. Nous pouvons régler au cours de la recherche le curseur qui permet d'aller vers une exploitation accrue du côté SAT, de l'heuristique VSDIS (et du *progress saving*) et des clauses apprises ou, au contraire, si les clauses nous éloignent d'une résolution efficace, se concentrer sur le cœur CP et ses choix de valeurs.

7.2.2 Minimisation de clauses

Afin d'améliorer l'efficacité de l'apprentissage des clauses, nous proposons de les minimiser au moment de leur construction de différentes manières : statiquement en utilisant les domaines (c'est-à-dire la structure des réseaux de contraintes), et dynamiquement en utilisant des explications (et des clauses nouvellement apprises).

L'idée générale est d'exploiter des formes de minimisation de clauses à la fois provenant de SAT et de CP (en particulier les *nogoods* généralisés). En effet, nous souhaitons profiter des avantages des deux mondes. La minimisation de clauses est très efficace du côté SAT mais ne considère pas la structure CP. Combiner les deux s'inscrit dans une politique hybride, et permet d'obtenir des clauses plus courtes donc généralement plus efficaces.

7.2.2.1 Minimisation basée sur les domaines

Il est possible de diminuer la taille des clauses en considérant les contraintes de domaine implicites. Comme nous savons qu'une seule valeur peut être affectée dans le domaine de chaque

variable, quelques simplifications sont possibles en considérant le lemme suivant (qui peut être assimilé à de l'*hyper-bin resolution* Bacchus (2002)) :

Lemme 2

Soient P un réseau de contraintes et $\Lambda = \Lambda^x \vee \Lambda'$ une clause où $\Lambda^x = (x = v_1 \vee \ldots \vee x = v_k)$ est composée de littéraux positifs provenant d'une variable $x \in vars(P)$, et Λ' contient des littéraux quelconques (sans aucun littéral positif de x). Si Λ est une conséquence logique de P alors pour toutes les valeurs $v \in \texttt{dom}(x) \setminus \{v_1, \ldots, v_k\}$, la clause $x \neq v \vee \Lambda'$ est une conséquence logique de P.

Preuve. Par la définition des domaines des variables en programmation par contraintes, nous avons $(x \neq v) \equiv (\bigvee_{v' \in \texttt{dom}(x) \setminus \{v\}} x = v')$. L'ajout de littéraux supplémentaires à Λ produit toujours une clause qui est une conséquence logique de P. Arbitrairement, pour toute valeur $v \in \texttt{dom}(x) \setminus \{v_1, \ldots, v_k\}$, nous pouvons décider d'ajouter à Λ tous les littéraux positifs manquants de la variable x excepté $x = v$. Ainsi, en remplaçant $\bigvee_{v' \in \texttt{dom}(x) \setminus \{v\}} x = v'$ par le littéral négatif équivalent $x \neq v$ donne une clause qui est une conséquence logique de P. q.e.d □

À partir du lemme 2, il est possible d'appliquer deux règles produisant des clauses plus courtes équivalentes :

1. Si une clause Λ contient tous les littéraux positifs d'une variable x, à l'exception d'une seule valeur v de $\texttt{dom}(x)$, nous pouvons remplacer tous ces littéraux positifs par $x \neq v$ dans Λ. Cette proposition est équivalente aux lignes 2-3 de l'algorithme 4.2 de Katsirelos (2008).
2. Si une clause Λ contient un littéral négatif $x \neq v$, tous les littéraux positifs de x dans Λ peuvent être supprimés de Λ en toute sécurité.

La première règle est un cas restreint du lemme 2 où Λ contient tous les littéraux positifs d'une variable x sauf un. La deuxième règle ne peut être appliquée que lorsque les explications proviennent de clauses.

Exemple 30

Soit la clause :

$$x_4 = 2 \vee x_2 = 0 \vee x_2 = 1 \vee x_5 \neq 1 \vee x_3 = 0 \vee x_1 = 2 \vee x_3 = 2 \vee x_5 = 0$$

Supposons que $\texttt{dom}(x_i) = \{0, 1, 2\}$. En appliquant la première règle nous obtenons la clause :

$$x_4 = 2 \vee x_2 \neq 2 \vee x_3 \neq 1 \vee x_1 = 2 \vee x_5 \neq 1 \vee x_5 = 0$$

En appliquant la seconde règle nous obtenons la clause :

$$x_4 = 2 \vee x_2 \neq 2 \vee x_3 \neq 1 \vee x_1 = 2 \vee x_5 \neq 1$$

7.2.2.2 Minimisation basée sur les explications

La minimisation de clauses est une technique efficace utilisée par les solveurs SAT pour réduire à la fois l'utilisation de la mémoire et le temps de résolution (Beame *et al.* (2004), Sörensson et Biere (2009), Knuth (2015)).

Nous utilisons ici une forme faible de minimisation de clauses, appelée *minimisation locale* (voir l'implémentation de l'algorithme 7.2.2.2C de Knuth (2015) [7]), qui tente de minimiser une clause Λ en supprimant tout littéral λ telle que l'explication de sa négation, $expl(\neg\lambda)$, est présente dans la clause Λ que nous voulons minimiser. C'est une forme de résolution dite auto-subsumante (voire auto-sous-sommante, *self-subsuming*). Cette procédure est appliquée de manière incrémentale en considérant les littéraux dans l'ordre chronologique inverse d'affectation pour s'assurer qu'un nombre maximal de littéraux sera supprimé.

Exemple 31

Soit la clause :

$$x_2 = 0 \vee x_2 = 1 \vee x_3 = 0 \vee x_7 = 0 \vee x_3 = 2$$

Supposons que $\texttt{expl}(x_7 \neq 0) = \{x_3 \neq 0\}$. Le littéral $x_7 = 0$ peut alors être retiré en toute sécurité en utilisant la minimisation locale avec $x_3 = 0 \vee x_7 \neq 0$. Nous obtenons la clause

$$x_2 = 0 \vee x_2 = 1 \vee x_3 = 0 \vee x_3 = 2$$

et la minimisation peut ainsi continuer de la même manière.

En raison de la manière dont les explications de contrainte sont calculées, appliquer d'abord la procédure de minimisation basée sur les domaines peut réduire le nombre de littéraux pouvant être supprimés par la minimisation locale. Par exemple, le littéral $x_7 = 0$ de l'exemple 31 ne peut plus être supprimé de la clause obtenue après la minimisation basée sur les domaines. En effet, si elle est appliquée avant sur la clause initiale, les littéraux $x_3 = 0$ et $x_3 = 2$ seront remplacés par le littéral $x_3 \neq 1$ et l'explication de $x_7 = 0$ ne serait plus contenue dans la clause.

Pour éviter cette situation, nous émettons une hypothèse dite *supplémentaire* (la contraposée de la règle 2 dérivée du lemme 2) : nous supposons sans risque que, pour chaque littéral négatif $x \neq a$ (n'étant pas le littéral que nous essayons de supprimer), l'ensemble complémentaire des littéraux positifs sur x est aussi implicitement présents dans la clause que nous essayons de minimiser.

Ainsi, le problème identifié ci-dessus est résolu ; dans notre illustration, le littéral $x_7 = 0$ peut maintenant être réellement supprimé. Il est important de noter que l'hypothèse supplémentaire ne peut pas être utilisée pour le littéral courant, c'est-à-dire le littéral (négatif) $x \neq a$ que nous essayons actuellement de simplifier. En effet, supposons que nous essayons de supprimer $x \neq 1$ de la clause $x \neq 1 \vee \Lambda'$ avec Λ' ne contenant aucun littéral sur x. Supposons également que $\texttt{expl}(x = 1) = \{x \neq 2\}$, ce qui équivaut à la clause $x = 1 \vee x = 2$. Si nous utilisons l'hypothèse supplémentaire sur $x \neq 1$, nous pouvons déduire que le littéral $x \neq 1$ peut être supprimé. Ce serait une déduction erronée car Λ' n'est pas une conséquence logique de $(x \neq 1 \vee \Lambda') \wedge (x = 1 \vee x = 2)$.

7. https://www-cs-faculty.stanford.edu/~knuth/programs/sat13.w

Il est possible d'aller encore plus loin en termes de minimisation. Pour cela, il faut prendre en compte les informations contenues dans la trace (*trail*). Plus précisément, pour toute clause Λ, nous notons $\texttt{gsd}(\Lambda)$ la valeur maximale ℓ telle que la négation $\neg\delta_1, \neg\delta_2, \ldots, \neg\delta_\ell$ des ℓ premières décisions $\delta_1, \delta_2, \ldots, \delta_\ell$ prises par l'algorithme de recherche sont présentes dans Λ.

La propriété suivante garantit que si un littéral λ a été déduit (et non décidé par le solveur) à un niveau inférieur ou égal à ℓ, il peut être supprimé de la clause en toute sécurité.

Propriété 4

Soit P un réseau de contraintes, et soit Λ une clause $\lambda \vee \Lambda'$ qui est une conséquence logique de P. Si $\texttt{niv}(\lambda) \leq \texttt{gsd}(\Lambda)$ alors Λ' est une conséquence logique de P.

Preuve. La preuve vient du fait que $\neg\delta_1 \vee \neg\delta_2 \vee \ldots \vee \neg\delta_{\texttt{lev}(\lambda)} \vee \neg\lambda$ est une conséquence logique de P. Ensuite, en appliquant une résolution auto-subsumante entre cette clause et Λ, nous obtenons que Λ' est une conséquence logique de P. q.e.d □

Il est possible d'envisager la contraposée de cette propriété pour l'extension de clauses. Ainsi, la négation de tous les littéraux déduits à un niveau inférieur ou égal à $\texttt{gsd}(\Lambda)$ peut être ajoutée à la clause que nous essayons de minimiser. L'ajout de ces littéraux peut être utile pour renforcer la minimisation locale des littéraux d'un niveau supérieur à $\texttt{gsd}(\Lambda)$.

Remarque 14

Notez qu'il est inutile de considérer les littéraux avec un niveau inférieur ou égal à $\texttt{gsd}(\Lambda)$ lors de la minimisation locale, car les déductions seront nécessairement supprimées par la propriété 4, et les décisions du solveur ne peuvent pas être supprimées.

En pratique, nous ne prolongerons pas les clauses. Il suffira de faire référence à la valeur de $\texttt{gsd}(\Lambda)$ pour identifier les littéraux pertinents.

7.2.3 Expérimentations

Les expériences ont été conduites sur les instances utilisées dans la catégorie *Minitrack - CSP* de la compétition XCSP3 de 2018 [8]. La catégorie est composée de 176 instances réparties sur 14 problèmes.

Nous avons comparé quatre méthodes (exactement deux méthodes principales, avec trois variantes pour la seconde) implémentées dans notre système NACRE. Pour cette validation expérimentale, nous avons utilisé une version de développement de NACRE. Cette version n'est pas encore une version compétition et modifie le système d'explications paresseuses afin de le simplifier et de l'améliorer. Les méthodes sont les suivantes :

8. http://xcsp.org/competition

— MAC, qui est un algorithme de recherche CSP classique conservant la cohérence d'arc généralisée pendant la recherche, couplé à dom/wdeg (Boussemart *et al.* (2004)) comme heuristique de choix de variable ;

— $\text{CA}_{\texttt{VSIDS}}$, la méthode exploitant la procédure d'analyse de conflits générique, avec VSIDS comme heuristique (combiné avec la sauvegarde de phase) ;

— $\text{CA}^{\texttt{minim.}}_{\texttt{VSIDS}}$, qui correspond à la méthode $\text{CA}_{\texttt{VSIDS}}$ combinée aux deux formes de minimisation de clauses introduites dans ce chapitre ;

— $\text{CA}^{\texttt{minim.}}_{\texttt{VSIDS}}+$, qui est $\text{CA}^{\texttt{minim.}}_{\texttt{VSIDS}}$ équipée de l'heuristique de recherche dynamique introduite dans la section précédente (avec $t = 100000$, $\mathcal{T}^{max} = 1$ et $\mathcal{T}^{min} = 0,1$, pour nos expériences).

Pour toutes les méthodes, la stratégie de redémarrage utilisée est `Luby` (Luby *et al.* (1993)). Les expériences ont été réalisées avec les mêmes paramètres que la compétition XCSP3, avec un *timeout* de 2400 secondes et une limite de mémoire de 15500 Mo. Ils ont été exécutés sur un processeur de 3,5 GHz.

Problèmes (# instances)	MAC	$\text{CA}_{\texttt{VSIDS}}$	$\text{CA}^{\texttt{minim.}}_{\texttt{VSIDS}}$	$\text{CA}^{\texttt{minim.}}_{\texttt{VSIDS}}+$
ColouredQueens (12)	3	3	3	3
Crossword (13)	3	3	3	3
Dubois (12)	5	**12**	**12**	**12**
Eternity (15)	6	6	**7**	6
Frb (16)	3	3	3	3
GracefulGraph (11)	4	**5**	**5**	**5**
Haystacks (10)	2	7	8	**10**
Langford (11)	4	**6**	**6**	**6**
MagicHexagon (11)	4	2	2	**5**
MagicSquare (13)	5	**13**	**13**	**13**
PseudoBoolean-dec (13)	3	**5**	**5**	**5**
Quasigroups_mini (16)	**7**	5	6	6
Rlfap-dec-scens11_mini (12)	11	9	10	**12**
Subisomorphism (11)	**4**	3	3	3
SAT	36	48	48	**49**
UNSAT	28	34	38	**43**
Total (176)	64	82	86	**92**

TABLE 7.2 – Nombre d'instances résolues – Comparaison des 4 méthodes sur les instances de la catégorie *Minitrack - CSP* de la compétition XCSP3 de 2018.

Comme nous pouvons le voir dans le tableau 7.2, la meilleure méthode, $\text{CA}^{\texttt{minim.}}_{\texttt{VSIDS}}+$, résout 12% d'instances de plus que $\text{CA}_{\texttt{VSIDS}}$, notre moteur de raisonnement de clause générique avec une heuristique pilotée par les clauses et aucune minimisation ou indications du ratio. Notez que notre approche est compétitive dans la résolution d'instances à la fois satisfaisables et insatisfiables. De plus, ajouter la minimisation puis le guidage par ratio nous permet d'observer une augmentation significative de la résolution d'instances insatisfiables. L'augmentation est de 21% de MAC à $\text{CA}_{\texttt{VSIDS}}$, puis de 12% de $\text{CA}_{\texttt{VSIDS}}$ à $\text{CA}^{\texttt{minim.}}_{\texttt{VSIDS}}$ et enfin de 13% de $\text{CA}^{\texttt{minim.}}_{\texttt{VSIDS}}$ à $\text{CA}^{\texttt{minim.}}_{\texttt{VSIDS}}+$. Cela correspond à une augmentation de 43% de MAC à $\text{CA}^{\texttt{minim.}}_{\texttt{VSIDS}}+$, 26% de $\text{CA}_{\texttt{VSIDS}}$ à $\text{CA}^{\texttt{minim.}}_{\texttt{VSIDS}}+$ et enfin 13% de $\text{CA}^{\texttt{minim.}}_{\texttt{VSIDS}}$ à $\text{CA}^{\texttt{minim.}}_{\texttt{VSIDS}}+$.

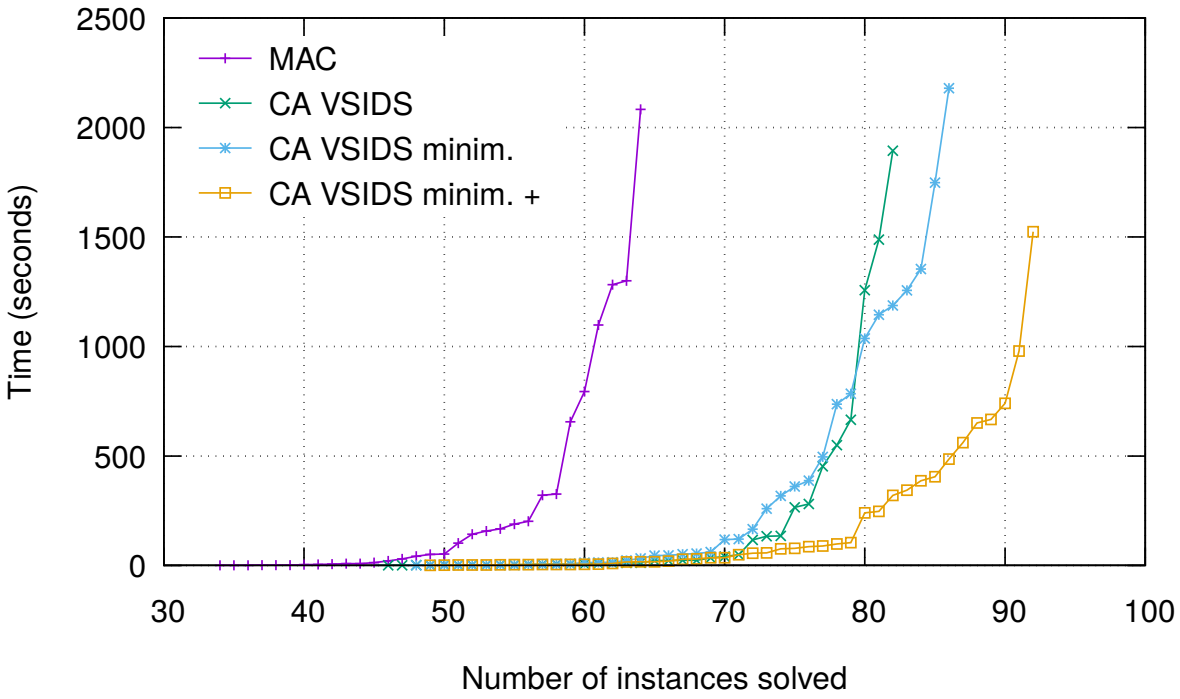


FIGURE 7.1 – Comparaison des 4 méthodes sur les instances de la catégorie *Minitrack - CSP* de la compétition XCSP3 de 2018.

Sur le graphique 7.1, il est clairement visible que $\text{CA}^{\texttt{minim.}}_{\texttt{VSIDS}}+$, en terme de performances, surpasse toutes les autres méthodes, en particulier la méthode classique MAC. Il est intéressant de noter que, sur notre ordinateur d'expérimentations, $\text{CA}^{\texttt{minim.}}_{\texttt{VSIDS}}+$ a résolu 7 instances ouvertes de la catégorie *Minitrack - CSP* (c'est-à-dire 7 instances laissées non résolues par tous les concurrents de la catégorie), augmentant potentiellement ainsi les performances du *Virtual Best Solver* de la catégorie. Une grande partie (toutes sauf 2) des instances sont résolues sous les 750 secondes par la meilleure méthode $\text{CA}^{\texttt{minim.}}_{\texttt{VSIDS}}+$. Cela confirme les observations faites dans le chapitre 6. De la même manière, au sein de cette version de développement, les instances sont soit résolues très vite, soit pas du tout.

	Maximum	Moyenne	Médiane
Taille des clauses avant minimisation	13433	625	274
Taille des clauses après minimisation	3622	310	138
Réduction de la minimisation dynamiques	87.8 %	33.3 %	30.0 %

TABLE 7.3 – Statistiques globales de minimisation sur la méthode $\text{CA}^{\texttt{minim.}}_{\texttt{VSIDS}}+$.

Comme nous pouvons le voir dans la Table 7.3, appliquer la minimisation est généralement assez efficace. La minimisation, qui est exécutée à la création de la clause, est importante car elle permet de diviser la taille des clauses par 2 (voir les différences entre les deux premières lignes du tableau).

Si nous considérons uniquement les minimisations dynamiques, environ 30% de littéraux des clauses (pourcentage de réduction médian) sont supprimés et jusqu'à 87% pour certains pro-

blèmes. En particulier, cela fonctionne très bien dans les cas des problèmes *Crossword* (réduction de 57% à 87%) et *LangfordBin* (réduction de 44% à 70%). Cela peut s'expliquer par le fait que de nombreux conflits liés à ces problèmes sont déclenchés par des contraintes et que le processus de minimisation permet d'obtenir de meilleures clauses pour ces problèmes.

7.2.4 Discussion

Dans cette section, nous avons proposé une heuristique de recherche basée sur l'heuristique de recherche SAT VSIDS, afin de résoudre le problème de la taille des clauses. En effet, celles-ci de par leur taille peuvent induire en erreur la recherche lorsque l'on utilise une heuristique classique. La seconde contribution de cette section est la combinaison de techniques de minimisation de clauses SAT et CP. Nous avons dans un premier temps détaillé la minimisation basée sur les domaines. Celle-ci introduit l'exploitation de la structure CP dans la méthode de minimisation. Nous avons pu extraire deux règles, cela a permis notamment de formaliser une partie de l'algorithme de 4.2 Katsirelos (2008). Dans un second temps, nous avons présenté des méthodes de minimisation SAT et résolu les problèmes qui auraient pu être introduits de par leur combinaison avec les 2 règles précédentes. Nous avons montré expérimentalement que la combinaison de ces techniques permet un gain d'efficacité significatif.

Comme il apparaît que la taille des clauses apprises, même minimisées, est encore assez élevée, une perspective intéressante consiste à essayer de les minimiser d'avantage en introduisant des lemmes (Huang (2010)). De plus, afin de pallier ce problème de taille de clauses, nous avons essayé de minimiser à nouveau les clauses lors de l'analyse de conflits. En effet, en considérant la minimisation par explications, nous pouvons nous apercevoir que des raisons différentes voire plus petites peuvent être identifiées plus tard dans la recherche. Cela peut être exploité lors de l'analyse de conflits afin de réduire à nouveau la taille des clauses utiles. Nous avons essayé cela en pratique mais pour l'instant les résultats ne sont pas concluants. La minimisation de chaque clause intervenant lors de l'analyse de conflits est bien plus couteuse que le gain obtenu à les réduire. Cette piste reste intéressante car il est possible d'essayer de minimiser les clauses les plus utiles d'abord.

Enfin, nous nous sommes principalement concentrés, afin de guider la recherche, sur l'heuristique VSIDS. Cependant, nous pensons qu'il est possible de concevoir des heuristiques de recherche hybrides basées sur des métriques basculant dynamiquement entre des heuristiques VSIDS et des heuristiques CSP classiques dirigées par l'impact (Refalo (2004)), l'activité (Michel et Hentenryck (2012)) et des degrés pondérés (Boussemart *et al.* (2004)).

7.3 Conclusion

Dans la première section, nous avons proposé des fonctions de raisonnement *ad-hoc* pour la contrainte *element* dans différentes formes. Nous avons notamment traité la forme dite *element constante* en détail, et avons proposé une adaptation de ces algorithmes de manière préliminaire à la version générale dite *element variable*. Au sein de la deuxième section, nous avons présenté différents mécanismes originaux afin de réaliser la combinaison de minimisation de clauses inspirées de SAT ou de systèmes orientés CP, à savoir les *nogoods* généralisés. De plus, nous avons proposé une heuristique de recherche basée sur VSIDS qui prend en compte la qualité des clauses générées afin de savoir si celles-ci ne sont pas en train de pénaliser la recherche. Nous avons vu

au cours de ce chapitre, que considérer la structure CP au sein des technologies SAT est une piste intéressante, qui a été validée expérimentalement.

Conclusion générale

La seconde partie du manuscrit regroupe l'ensemble des contributions originales présentées dans cette thèse. La première traite du sujet des *nogoods* et plus particulièrement des *increasing-nogoods*. En étudiant ces derniers, nous avons identifié des similitudes pouvant être exploitées afin d'améliorer leur pouvoir de filtrage. Nous avons introduit trois règles : la combinaison de décisions négatives ; généralisée ensuite avec la combinaison par équivalence d'alpha ; ainsi que la combinaison par équivalence de décisions négatives basée sur la nouvelle propriété de variables pivots. Ces trois règles permettent la combinaison d'*increasing-nogoods* dans un contexte d'algorithme complet avec retours arrière et redémarrages. Notamment, elles anticipent les conflits qui pourraient avoir lieu dans l'espace de recherche considéré.

Nous avons déjà introduit quelques perspectives lors de ce chapitre (dans la section 5.3), à savoir une perspective technique concernant un système de sentinelles permettant l'anticipation de conflits de manière plus légère ; ainsi qu'une perspective plus théorique qui concerne la génération, à chaque redémarrage, de *nld-nogoods* réduits. De plus, nous aimerions implémenter ces règles de combinaison au sein de notre outil de raisonnement NACRE. En effet, étant donnée sa conception, nous pourrions probablement obtenir des résultats plus marqués. En outre, cela nous permettrait d'identifier différents comportement, déduire de nouvelles règles, voire de corriger le problème de pondération des heuristiques adaptatives (en testant, par exemple, de nombreuses heuristiques différentes).

La deuxième contribution porte sur le moteur de raisonnement générique NACRE (plus particulièrement NACRE 1.0.4). Cet outil répond au besoin, qui s'est très vite ressenti, d'avoir un solveur modulaire et évolutif qui permet de gérer des *nogoods* et des clauses tout en restant efficace en pratique. Rappelons que NACRE 1.0.4 a pris la première place de la catégorie *Minitrack - CSP* de la compétition XCSP 2018. Ce chapitre ne se limite pas à la description technique de l'outil mais apporte aussi un certain nombre d'améliorations aux explications paresseuses génériques. De plus, une nouvelle politique de réduction de la base de clauses apprises est décrite. Elle prend en compte une mesure originale, le nombre de variables CP différentes des littéraux qui apparaissent dans les clauses. Dans la littérature, et particulièrement dans un contexte hybride SAT/CP, les heuristiques d'un domaine ou de l'autre sont utilisées. A notre connaissance, ce type d'heuristiques n'a jamais été proposé, ce qui ouvre de nouvelles perspectives de recherche.

Afin d'apporter de la robustesse à notre outil, nous souhaiterions par la suite prendre en compte toutes les contraintes du *XCSP3-core*. Cela permettrait également de soumettre l'outil dans la catégorie principale de la compétition. De plus, nous voulons valider les améliorations ainsi que les performances de l'outil dans un contexte parallèle (grâce à Audemard *et al.* (2019) notamment), voire distribué afin de confirmer celles-ci et d'expérimenter sur l'échange de clauses et/ou *nogoods* dans ce contexte. Par ailleurs, nous espérons pouvoir mettre en place un module d'optimisation qui prenne en compte un raisonnement clausal. Cela permettrait de tendre plus rapidement vers l'optimalité. Le principal problème actuellement réside dans la difficulté de combiner l'apprentissage de clauses avec une recherche dichotomique. En effet, un retour insatisfiable lors du test d'une borne nous oblige à écarter les clauses générées lors de cette recherche. Grâce à une conception orientée vers le raisonnement générique, NACRE a permis plusieurs contributions. Notamment celles du chapitre 7 rappelées ci-après.

Dans le dernier chapitre de ce manuscrit, nous nous sommes intéressés à améliorer sensiblement le pouvoir de propagation et par conséquent la qualité des clauses générées. Pour cela, nous

avons exploré trois pistes : la génération de clauses plus fines grâce à de nouvelles fonctions de raisonnement *ad-hoc* ; l'amélioration de la qualité des clauses par la minimisation ; et pour finir la confiance portée aux clauses, en particulier d'un point de vue heuristique.

La première piste nous a permis d'introduire de nouvelles fonctions de raisonnement pour la contrainte globale *element* sous différentes formes. Cela a aussi mis en lumière la complexité d'effectuer une telle tâche. En effet, effectuer le *reverse engineering* d'un propagateur particulier peut s'avérer fastidieux. Cela ouvre une perspective très intéressante : proposer un générateur automatique de fonction de raisonnement *ad-hoc* à partir du propagateur. Nous pourrions identifier les cas possibles, les variables touchées en fonction des différents cas possibles afin de produire la fonction permettant d'expliquer un conflit voire la propagation d'un littéral particulier. En effet, il existe plus de 400 contraintes globales répertoriées, proposer ces deux fonctions de raisonnement pour toutes les fonctions de filtrage de l'ensemble de ces contraintes serait un travail hors de portée : l'automatisation peut en être la réponse.

Combiner des techniques provenant de plusieurs domaines est le principe de l'hybridation. C'est ce que nous avons choisi de faire pour la seconde piste. En effet, appliquer des règles de minimisation de *nogoods* et de clauses sur les clauses générées depuis une fonction de raisonnement générique et combiner cela à des règles qui considèrent la provenance des littéraux (en l'occurrence, les variables CP) nous a permis de réduire énormément les clauses ; et par conséquent, améliorer la qualité des clauses.

La dernière piste que nous avons explorée répond à l'interrogation que nous nous sommes posée lors de l'introduction de ce manuscrit : « Pouvons-nous faire confiance aux clauses ? ». La réponse a cette question est, comme souvent malheureusement, « ça dépend ». En effet, sur certains problèmes, les clauses générées vont permettre une efficacité redoutable du moteur de raisonnement. Mais cela n'est pas vrai dans le cas général. Afin de donner une réponse à la nouvelle question posée : « Comment pouvons-nous faire confiance aux clauses ? », nous avons proposé une heuristique, basée sur VSIDS, qui grâce à un calcul de ratio permet de basculer entre une heuristique SAT ou CP. Cette heuristique, combinée à la minimisation rappelée précédemment, nous a permis d'obtenir des résultats très intéressants en pratique.

Travailler dans un contexte hybride nous a permis de mettre en avant les compétences acquises dans le domaine SAT au sein d'autres travaux. En particulier, la résolution et la réalisation de réseaux de contraintes qualitatives ainsi que le problème de coloriage de graphes. Cela a été possible grâce à une traduction incrémentale de ces problèmes vers SAT et au *framework CEGAR* (Clarke *et al.* (2003)). CEGAR (*Counter-Example-Guided Abstraction Refiniment*) est une manière incrémentale de décider la satisfiabilité de formules en logique propositionnelle. Le but de résoudre une abstraction (relaxation) de la formule originale, c'est à dire une formule plus simple à résoudre en pratique (car moins contrainte). Si la formule moins contrainte est fausse alors l'insatisfiabilité est prouvée ; sinon, nous testons si la solution s'étend au problème original. Si ce n'est pas le cas, nous pouvons ajouter une (ou plusieurs) contrainte et opérer de cette manière itérativement jusqu'à, dans le pire cas, devoir résoudre la formule originale.

La première contribution, publiée à *the 24rd International Conference on Principles and Practice of Constraint Programming – CP'18* (Glorian *et al.* (2018)) et aux *Quinzièmes journées Francophones de Programmation par Contraintes – JFPC'19* (Glorian *et al.* (2019a)), concerne les réseaux de contraintes qualitatives, notamment le langage *RCC8*. RCC8 est un langage utilisé afin de décrire des arrangements topologiques de régions dans l'espace. Dans ces travaux, nous permettons à la fois la résolution (retourner la satisfiabilité) et la réalisation (produire un modèle spatial) de problèmes RCC8 qui se réalisent (lorsque satisfiables) grâce à des rectangles. En

traitant ces deux problèmes à la fois, nous obtenons des résultats qui rivalisent avec les meilleurs méthodes de résolution de l'état de l'art du domaine.

La seconde contribution non présentée dans ce manuscrit s'attaque au problème de coloriage de graphes et a fait l'objet d'une publication à *the 25rd International Conference on Principles and Practice of Constraint Programming – CP'19* (Glorian *et al.* (2019b)). Ce problème consiste à utiliser le moins de couleurs afin de colorier tous les nœuds d'un graphe tel que deux nœuds liés par une arête ont une couleur différente. Le coloriage de graphes demeure encore sujet à de nombreuses publications (Hebrard et Katsirelos (2018; 2019b;a)) [9]. Nous avons, de la même manière que précédemment, proposé une traduction vers le problème de satisfaction booléenne. Nous effectuons cette traduction de manière incrémentale, grâce au *framework CEGAR* et à la récurrence de suppression-contraction de Zykov (1949). Cela nous a permis d'obtenir des résultats significatifs, dépassant souvent les approches de l'état de l'art.

9. Hebrard et Katsirelos (2018) a, par ailleurs, reçu le prix du meilleur papier CP'18

Bibliographie

Alfred V. Aho, John E. Hopcroft et Jeffrey D. Ullman : *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974. ISBN 0-201-00029-6.

Gilles Audemard, Gael Glorian, Jean-Marie Lagniez, Valentin Montmirail et Nicolas Szczepanski : pfactory: A generic library for designing parallel solvers. *In the 16th International Conference on Applied Computing, AC'19*, 2019. to appear.

Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure et Lakhdar Sais : On freezing and reactivating learnt clauses. *In* Karem A. Sakallah et Laurent Simon, éditeurs : *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*, volume 6695 de *Lecture Notes in Computer Science*, pages 188–200. Springer, 2011. URL https://doi.org/10.1007/978-3-642-21581-0_16.

Gilles Audemard et Laurent Simon : Glucose: a solver that predicts learnt clauses quality. *SAT Competition*, pages 7–8, 2009a.

Gilles Audemard et Laurent Simon : Predicting learnt clauses quality in modern SAT solvers. *In* Craig Boutilier, éditeur : *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 399–404, 2009b. URL http://ijcai.org/Proceedings/09/Papers/074.pdf.

Gilles Audemard et Laurent Simon : Refining restarts strategies for SAT and UNSAT. *In* Michela Milano, éditeur : *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, volume 7514 de *Lecture Notes in Computer Science*, pages 118–126. Springer, 2012. URL https://doi.org/10.1007/978-3-642-33558-7_11.

Fahiem Bacchus : Enhancing davis putnam with extended binary clause reasoning. *In* Rina Dechter, Michael J. Kearns et Richard S. Sutton, éditeurs : *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada.*, pages 613–619. AAAI Press / The MIT Press, 2002. URL http://www.aaai.org/Library/AAAI/2002/aaai02-092.php.

Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia et Cesare Tinelli : Satisfiability modulo theories. *In* Armin Biere, Marijn Heule, Hans van Maaren et Toby Walsh, éditeurs : *Handbook of Satisfiability*, volume 185 de *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009. URL https://doi.org/10.3233/978-1-58603-929-5-825.

Peter Barth : A davis-putnam based enumeration algorithm for linear pseudo-boolean optimization. 1995.

Paul Beame, Henry A. Kautz et Ashish Sabharwal : Towards understanding and harnessing the potential of clause learning. *J. Artif. Intell. Res.*, 22:319–351, 2004. URL https://doi.org/10.1613/jair.1410.

Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey et Thierry Petit : Global constraint catalogue: Past, present and future. *Constraints*, 12(1):21–62, 2007. URL https://doi.org/10.1007/s10601-006-9010-8.

Christian Bessière et Jean-Charles Régin : MAC and combined heuristics: Two reasons to forsake FC (and cbj ?) on hard problems. *In* Eugene C. Freuder, éditeur : *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming, Cambridge, Massachusetts, USA, August 19-22, 1996*, volume 1118 de *Lecture Notes in Computer Science*, pages 61–75. Springer, 1996. URL https://doi.org/10.1007/3-540-61551-2_66.

Christian Bessière, Jean-Charles Régin, Roland H. C. Yap et Yuanlin Zhang : An optimal coarse-grained arc consistency algorithm. *Artif. Intell.*, 165(2):165–185, 2005. URL https://doi.org/10.1016/j.artint.2005.02.004.

Armin Biere : Adaptive restart strategies for conflict driven SAT solvers. *In* Hans Kleine Büning et Xishun Zhao, éditeurs : *Theory and Applications of Satisfiability Testing - SAT 2008, 11th International Conference, SAT 2008, Guangzhou, China, May 12-15, 2008. Proceedings*, volume 4996 de *Lecture Notes in Computer Science*, pages 28–33. Springer, 2008. URL https://doi.org/10.1007/978-3-540-79719-7_4.

James R. Bitner et Edward M. Reingold : Backtrack programming techniques. *Commun. ACM*, 18(11):651–656, 1975. URL https://doi.org/10.1145/361219.361224.

Frédéric Boussemart, Fred Hemery, Christophe Lecoutre et Lakhdar Sais : Boosting systematic search by weighting constraints. *In* Ramón López de Mántaras et Lorenza Saitta, éditeurs : *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, pages 146–150. IOS Press, 2004.

Frédéric Boussemart, Christophe Lecoutre et Cédric Piette : XCSP3: an integrated format for benchmarking combinatorial constrained problems. *CoRR*, abs/1611.03398, 2016. URL http://arxiv.org/abs/1611.03398.

Daniel Brélaz : New methods to color vertices of a graph. *Commun. ACM*, 22(4):251–256, 1979. URL https://doi.org/10.1145/359094.359101.

Michael Buro et H Kleine Büning : *Report on a SAT competition.* Fachbereich Math.-Informatik, Univ. Gesamthochschule, 1992.

Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu et Helmut Veith : Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50 (5):752–794, 2003. URL https://doi.org/10.1145/876638.876643.

Stephen A. Cook : The complexity of theorem-proving procedures. *In* Michael A. Harrison, Ranan B. Banerji et Jeffrey D. Ullman, éditeurs : *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM, 1971. URL https://doi.org/10.1145/800157.805047.

James M. Crawford et Andrew B. Baker : Experimental results on the application of satisfiability algorithms to scheduling problems. *In* Barbara Hayes-Roth et Richard E. Korf, éditeurs : *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 2.*, pages 1092–1097. AAAI Press / The MIT Press, 1994. URL http://www.aaai.org/Library/AAAI/1994/aaai94-168.php.

Martin Davis, George Logemann et Donald W. Loveland : A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962. URL https://doi.org/10.1145/368273.368557.

Martin Davis et Hilary Putnam : A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960. URL http://doi.acm.org/10.1145/321033.321034.

Johan de Kleer : A comparison of ATMS and CSP techniques. *In* N. S. Sridharan, éditeur : *Proceedings of the 11th International Joint Conference on Artificial Intelligence. Detroit, MI, USA, August 1989*, pages 290–296. Morgan Kaufmann, 1989. URL http://ijcai.org/Proceedings/89-1/Papers/046.pdf.

Romuald Debruyne et Christian Bessière : From restricted path consistency to max-restricted path consistency. *In* Gert Smolka, éditeur : *Principles and Practice of Constraint Programming - CP97, Third International Conference, Linz, Austria, October 29 - November 1, 1997, Proceedings*, volume 1330 de *Lecture Notes in Computer Science*, pages 312–326. Springer, 1997a. URL https://doi.org/10.1007/BFb0017448.

Romuald Debruyne et Christian Bessière : Some practicable filtering techniques for the constraint satisfaction problem. *In Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes*, pages 412–417. Morgan Kaufmann, 1997b.

Rina Dechter : Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artif. Intell.*, 41(3):273–312, 1990. URL https://doi.org/10.1016/0004-3702(90)90046-3.

Niklas Eén et Niklas Sörensson : An extensible sat-solver. *In* Enrico Giunchiglia et Armando Tacchella, éditeurs : *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 de *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003. URL https://doi.org/10.1007/978-3-540-24605-3_37.

Thibaut Feydy et Peter J. Stuckey : Lazy clause generation reengineered. *In* Ian P. Gent, éditeur : *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings*, volume 5732 de *Lecture Notes in Computer Science*, pages 352–366. Springer, 2009. URL https://doi.org/10.1007/978-3-642-04244-7_29.

Jon William Freeman : *Improvements to propositional satisfiability search algorithms*. Thèse de doctorat, University of Pennsylvania Philadelphia, PA, 1995.

Eugene C. Freuder : In pursuit of the holy grail. *Constraints*, 2(1):57–61, 1997. URL https://doi.org/10.1023/A:1009749006768.

Daniel Frost et Rina Dechter : Dead-end driven learning. *In* Barbara Hayes-Roth et Richard E. Korf, éditeurs : *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 1.*, pages 294–300. AAAI Press / The MIT Press, 1994. URL http://www.aaai.org/Library/AAAI/1994/aaai94-045.php.

Daniel Frost et Rina Dechter : Look-ahead value ordering for constraint satisfaction problems. *In Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montréal Québec, Canada, August 20-25 1995, 2 Volumes*, pages 572–578. Morgan Kaufmann, 1995. URL http://ijcai.org/Proceedings/95-1/Papers/075.pdf.

Ian P. Gent, Christopher Jefferson et Ian Miguel : Watched literals for constraint propagation in minion. *In* Frédéric Benhamou, éditeur : *Principles and Practice of Constraint Programming - CP 2006, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings*, volume 4204 de *Lecture Notes in Computer Science*, pages 182–197. Springer, 2006. URL https://doi.org/10.1007/11889205_15.

Ian P. Gent, Ian Miguel et Neil C. A. Moore : Lazy explanations for constraint propagators. *In* Manuel Carro et Ricardo Peña, éditeurs : *Practical Aspects of Declarative Languages, 12th International Symposium, PADL 2010, Madrid, Spain, January 18-19, 2010. Proceedings*, volume 5937 de *Lecture Notes in Computer Science*, pages 217–233. Springer, 2010. URL https://doi.org/10.1007/978-3-642-11503-5_19.

Matthew L. Ginsberg : Dynamic backtracking. *J. Artif. Intell. Res.*, 1:25–46, 1993. URL https://doi.org/10.1613/jair.1.

Gael Glorian : Nacre. *In* Christophe Lecoutre et Olivier Roussel, éditeurs : *Proceedings of the 2018 XCSP3 Competition*, pages 85–85, 2019. URL http://arxiv.org/abs/1901.01830.

Gael Glorian, Frédéric Boussemart, Jean-Marie Lagniez, Christophe Lecoutre et Bertrand Mazure : Combinaison de nogoods extraits au redémarrage. *In Treizièmes journées Francophones de Programmation par Contraintes, JFPC 2017, Montreuil-sur-mer, France, 13 au 15 juin, 2017, Actes*, volume 13, pages 55–64. AFPC, 2017a.

Gael Glorian, Frédéric Boussemart, Jean-Marie Lagniez, Christophe Lecoutre et Bertrand Mazure : Combining nogoods in restart-based search. *In* J. Christopher Beck, éditeur : *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10416 de *Lecture Notes in Computer Science*, pages 129–138. Springer, 2017b. URL https://doi.org/10.1007/978-3-319-66158-2_9.

Gael Glorian, Jean-Marie Lagniez, Valentin Montmirail et Michael Sioutis : An incremental sat-based approach to reason efficiently on qualitative constraint networks. *In* John N. Hooker, éditeur : *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, volume 11008 de *Lecture Notes in Computer Science*, pages 160–178. Springer, 2018. URL https://doi.org/10.1007/978-3-319-98334-9_11.

Gael Glorian, Jean-Marie Lagniez, Valentin Montmirail et Michael Sioutis : Une approche sat incrémentale pour raisonner efficacement sur les réseaux de contraintes qualitatives. *In Quizièmes journées Francophones de Programmation par Contraintes, JFPC 2019, Albi, France, 10 au 13 juin, 2019, Actes*, volume 15, pages 67–76. AFPC, 2019a.

Gael Glorian, Jean-Marie Lagniez, Valentin Montmirail et Nicolas Szczepanski : An incremental sat-based approach to the graph colouring problem. *In* Thomas Schiex et Simon de Givry, éditeurs : *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings*,

volume 11802 de *Lecture Notes in Computer Science*, pages 213–231. Springer, 2019b. URL https://doi.org/10.1007/978-3-030-30048-7_13.

Carla P. Gomes, Bart Selman, Nuno Crato et Henry A. Kautz : Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *J. Autom. Reasoning*, 24(1/2):67–100, 2000. URL https://doi.org/10.1023/A:1006314320276.

Éric Grégoire, Jean-Marie Lagniez et Bertrand Mazure : A CSP solver focusing on fac variables. *In* Jimmy Ho-Man Lee, éditeur : *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings*, volume 6876 de *Lecture Notes in Computer Science*, pages 493–507. Springer, 2011. URL https://doi.org/10.1007/978-3-642-23786-7_38.

Djamal Habet et Cyril Terrioux : Conflict history based search for constraint satisfaction problem. *In* Chih-Cheng Hung et George A. Papadopoulos, éditeurs : *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, SAC 2019, Limassol, Cyprus, April 8-12, 2019*, pages 1117–1122. ACM, 2019. URL https://doi.org/10.1145/3297280.3297389.

Youssef Hamadi, Said Jabbour et Lakhdar Sais : Lysat: solver description. *SAT Competition*, pages 23–24, 2009.

Robert M. Haralick et Gordon L. Elliott : Increasing tree search efficiency for constraint satisfaction problems. *Artif. Intell.*, 14(3):263–313, 1980. URL https://doi.org/10.1016/0004-3702(80)90051-X.

Emmanuel Hebrard et George Katsirelos : Clause learning and new bounds for graph coloring. *In* John N. Hooker, éditeur : *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, volume 11008 de *Lecture Notes in Computer Science*, pages 179–194. Springer, 2018. URL https://doi.org/10.1007/978-3-319-98334-9_12.

Emmanuel Hebrard et George Katsirelos : Clause learning and new bounds for graph coloring. *In* Sarit Kraus, éditeur : *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 6166–6170. ijcai.org, 2019a. URL https://doi.org/10.24963/ijcai.2019/856.

Emmanuel Hebrard et George Katsirelos : A hybrid approach for exact coloring of massive graphs. *In* Louis-Martin Rousseau et Kostas Stergiou, éditeurs : *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 16th International Conference, CPAIOR 2019, Thessaloniki, Greece, June 4-7, 2019, Proceedings*, volume 11494 de *Lecture Notes in Computer Science*, pages 374–390. Springer, 2019b. URL https://doi.org/10.1007/978-3-030-19212-9_25.

Pascal Van Hentenryck et Jean-Philippe Carillon : Generality versus specificity: An experience with AI and OR techniques. *In* Howard E. Shrobe, Tom M. Mitchell et Reid G. Smith, éditeurs : *Proceedings of the 7th National Conference on Artificial Intelligence, St. Paul, MN, USA, August 21-26, 1988.*, pages 660–664. AAAI Press / The MIT Press, 1988. URL http://www.aaai.org/Library/AAAI/1988/aaai88-117.php.

John N. Hooker et V. Vinay : Branching rules for satisfiability. *J. Autom. Reasoning*, 15 (3):359–383, 1995. URL https://doi.org/10.1007/BF00881805.

Jinbo Huang : Extended clause learning. *Artif. Intell.*, 174(15):1277–1284, 2010. URL https://doi.org/10.1016/j.artint.2010.07.008.

Kazuo Iwama et Shuichi Miyazaki : Sat-varible complexity of hard combinatorial problems. *In* Björn Pehrson et Imre Simon, éditeurs : *Technology and Foundations - Information Processing '94, Volume 1, Proceedings of the IFIP 13th World Computer Congress, Hamburg, Germany, 28 August - 2 September, 1994*, volume A-51 de *IFIP Transactions*, pages 253–258. North-Holland, 1994.

Saïd Jabbour, Jerry Lonlac, Lakhdar Saïs et Yakoub Salhi : Revisiting the learned clauses database reduction strategies. *International Journal on Artificial Intelligence Tools*, 27(8): 1850033, 2018. URL https://doi.org/10.1142/S0218213018500331.

Robert G. Jeroslow et Jinchang Wang : Solving propositional satisfiability problems. *Ann. Math. Artif. Intell.*, 1:167–187, 1990. URL https://doi.org/10.1007/BF01531077.

Roberto J. Bayardo Jr. et Robert Schrag : Using CSP look-back techniques to solve real-world SAT instances. *In* Benjamin Kuipers et Bonnie L. Webber, éditeurs : *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island, USA.*, pages 203–208. AAAI Press / The MIT Press, 1997. URL http://www.aaai.org/Library/AAAI/1997/aaai97-032.php.

Narendra Jussien, Romuald Debruyne et Patrice Boizumault : Maintaining arc-consistency within dynamic backtracking. *In* Rina Dechter, éditeur : *Principles and Practice of Constraint Programming - CP 2000, 6th International Conference, Singapore, September 18-21, 2000, Proceedings*, volume 1894 de *Lecture Notes in Computer Science*, pages 249–261. Springer, 2000. URL https://doi.org/10.1007/3-540-45349-0_19.

George Katsirelos : *Nogood processing in CSPs*. Thèse de doctorat, 2008.

George Katsirelos et Fahiem Bacchus : Unrestricted nogood recording in CSP search. *In* Francesca Rossi, éditeur : *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*, volume 2833 de *Lecture Notes in Computer Science*, pages 873–877. Springer, 2003. URL https://doi.org/10.1007/978-3-540-45193-8_70.

Donald E Knuth : *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. Addison-Wesley Professional, 2015.

Jean-Louis Laurière : A language and a program for stating and solving combinatorial problems. *Artif. Intell.*, 10(1):29–127, 1978. URL https://doi.org/10.1016/0004-3702(78)90029-2.

Christophe Lecoutre et Fred Hemery : A study of residual supports in arc consistency. *In* Manuela M. Veloso, éditeur : *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 125–130, 2007. URL http://ijcai.org/Proceedings/07/Papers/018.pdf.

Christophe Lecoutre et Olivier Roussel : Proceedings of the 2018 XCSP3 competition. *CoRR*, abs/1901.01830, 2019. URL http://arxiv.org/abs/1901.01830.

Christophe Lecoutre, Lakhdar Sais, Sébastien Tabary et Vincent Vidal : Nogood recording from restarts. *In* Manuela M. Veloso, éditeur : *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 131–136, 2007a. URL http://ijcai.org/Proceedings/07/Papers/019.pdf.

Christophe Lecoutre, Lakhdar Sais, Sébastien Tabary et Vincent Vidal : Recording and minimizing nogoods from restarts. *JSAT*, 1(3-4):147–167, 2007b. URL https://satassociation.org/jsat/index.php/jsat/article/view/13.

Christophe Lecoutre, Lakhdar Sais, Sébastien Tabary et Vincent Vidal : Reasoning from last conflict(s) in constraint programming. *Artif. Intell.*, 173(18):1592–1614, 2009. URL https://doi.org/10.1016/j.artint.2009.09.002.

Christophe Lecoutre et Sébastien Tabary : Symmetry-reinforced nogood recording from restarts. *In 11th International Workshop on Symmetry in Constraint Satisfaction Problems (SymCon'11)*, pages 13–27, 2011.

Christophe Lecoutre et Julien Vion : Enforcing arc consistency using bitwise operations. 2008.

Jimmy H. M. Lee, Christian Schulte et Zichen Zhu : Increasing nogoods in restart-based search. *In* Dale Schuurmans et Michael P. Wellman, éditeurs : *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, pages 3426–3433. AAAI Press, 2016. URL http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12357.

Jimmy H. M. Lee et Zichen Zhu : An increasing-nogoods global constraint for symmetry breaking during search. *In* Barry O'Sullivan, éditeur : *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, volume 8656 de *Lecture Notes in Computer Science*, pages 465–480. Springer, 2014. URL https://doi.org/10.1007/978-3-319-10428-7_35.

Chu Min Li et Anbulagan : Heuristics based on unit propagation for satisfiability problems. *In Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes*, pages 366–371. Morgan Kaufmann, 1997. URL http://ijcai.org/Proceedings/97-1/Papers/057.pdf.

Jia Hui Liang, Vijay Ganesh, Pascal Poupart et Krzysztof Czarnecki : Learning rate based branching heuristic for SAT solvers. *In* Nadia Creignou et Daniel Le Berre, éditeurs : *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 de *Lecture Notes in Computer Science*, pages 123–140. Springer, 2016. URL https://doi.org/10.1007/978-3-319-40970-2_9.

Paolo Liberatore : On the complexity of choosing the branching literal in DPLL. *Artif. Intell.*, 116(1-2):315–326, 2000. URL https://doi.org/10.1016/S0004-3702(99)00097-1.

Michael Luby, Alistair Sinclair et David Zuckerman : Optimal speedup of las vegas algorithms. *Inf. Process. Lett.*, 47(4):173–180, 1993. URL https://doi.org/10.1016/0020-0190(93)90029-9.

Alan K. Mackworth : Consistency in networks of relations. *Artif. Intell.*, 8(1):99–118, 1977. URL https://doi.org/10.1016/0004-3702(77)90007-8.

João P. Marques-Silva : The impact of branching heuristics in propositional satisfiability algorithms. *In* Pedro Barahona et José Júlio Alferes, éditeurs : *Progress in Artificial Intelligence, 9th Portuguese Conference on Artificial Intelligence, EPIA '99, Évora, Portugal, September 21-24, 1999, Proceedings*, volume 1695 de *Lecture Notes in Computer Science*, pages 62–74. Springer, 1999. URL https://doi.org/10.1007/3-540-48159-1_5.

João P. Marques-Silva et Karem A. Sakallah : GRASP - a new search algorithm for satisfiability. *In ICCAD*, pages 220–227, 1996. URL https://doi.org/10.1109/ICCAD.1996.569607.

João P. Marques-Silva et Karem A. Sakallah : GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999. URL https://doi.org/10.1109/12.769433.

Laurent Michel et Pascal Van Hentenryck : Activity-based search for black-box constraint programming solvers. *In* Nicolas Beldiceanu, Narendra Jussien et Eric Pinson, éditeurs : *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems - 9th International Conference, CPAIOR 2012, Nantes, France, May 28 - June1, 2012. Proceedings*, volume 7298 de *Lecture Notes in Computer Science*, pages 228–243. Springer, 2012. URL https://doi.org/10.1007/978-3-642-29828-8_15.

Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang et Sharad Malik : Chaff: Engineering an efficient SAT solver. *In Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001. URL https://doi.org/10.1145/378239.379017.

Alexander Nadel et Vadim Ryvchin : Chronological backtracking. *In* Olaf Beyersdorff et Christoph M. Wintersteiger, éditeurs : *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10929 de *Lecture Notes in Computer Science*, pages 111–121. Springer, 2018. URL https://doi.org/10.1007/978-3-319-94144-8_7.

Olga Ohrimenko, Peter J. Stuckey et Michael Codish : Propagation = lazy clause generation. *In* Christian Bessiere, éditeur : *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 de *Lecture Notes in Computer Science*, pages 544–558. Springer, 2007. URL https://doi.org/10.1007/978-3-540-74970-7_39.

Olga Ohrimenko, Peter J. Stuckey et Michael Codish : Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009. URL https://doi.org/10.1007/s10601-008-9064-x.

Knot Pipatsrisawat et Adnan Darwiche : A lightweight component caching scheme for satisfiability solvers. *In* João Marques-Silva et Karem A. Sakallah, éditeurs : *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings*, volume 4501 de *Lecture Notes in Computer Science*, pages 294–299. Springer, 2007. URL https://doi.org/10.1007/978-3-540-72788-0_28.

Daniele Pretolani : Satisfiability and hypergraphs. *Bulletin-european association for theoretical computer science*, 50:542–542, 1993.

Patrick Prosser : Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993. URL https://doi.org/10.1111/j.1467-8640.1993.tb00310.x.

Philippe Refalo : Impact-based search strategies for constraint programming. *In* Mark Wallace, éditeur : *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, volume 3258 de *Lecture Notes in Computer Science*, pages 557–571. Springer, 2004. URL https://doi.org/10.1007/978-3-540-30201-8_41.

Jean-Charles Régin : A filtering algorithm for constraints of difference in csps. *In* Barbara Hayes-Roth et Richard E. Korf, éditeurs : *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 1.*, pages 362–367. AAAI Press / The MIT Press, 1994. URL http://www.aaai.org/Library/AAAI/1994/aaai94-055.php.

John Alan Robinson : A machine-oriented logic based on the resolution principle. *J. ACM*, 12 (1):23–41, 1965. URL http://doi.acm.org/10.1145/321250.321253.

Francesca Rossi, Peter van Beek et Toby Walsh, éditeurs. *Handbook of Constraint Programming*, volume 2 de *Foundations of Artificial Intelligence*. Elsevier, 2006. ISBN 978-0-444-52726-4. URL http://www.sciencedirect.com/science/bookseries/15746526/2.

Daniel Sabin et Eugene C. Freuder : Contradicting conventional wisdom in constraint satisfaction. *In* Alan Borning, éditeur : *Principles and Practice of Constraint Programming, Second International Workshop, PPCP'94, Rosario, Orcas Island, Washington, USA, May 2-4, 1994, Proceedings*, volume 874 de *Lecture Notes in Computer Science*, pages 10–20. Springer, 1994a. URL https://doi.org/10.1007/3-540-58601-6_86.

Daniel Sabin et Eugene C. Freuder : Contradicting conventional wisdom in constraint satisfaction. *In ECAI*, pages 125–129, 1994b.

Thomas Schiex et Gérard Verfaillie : Nogood recording for static and dynamic constraint satisfaction problems. *International Journal on Artificial Intelligence Tools*, 3(2):187–208, 1994. URL https://doi.org/10.1142/S0218213094000108.

Gilles Simonin, Christian Artigues, Emmanuel Hebrard et Pierre Lopez : Scheduling scientific experiments for comet exploration. *Constraints*, 20(1):77–99, 2015. URL https://doi.org/10.1007/s10601-014-9169-3.

Takehide Soh, Mutsunori Banbara et Naoyuki Tamura : Proposal and evaluation of hybrid encoding of CSP to SAT integrating order and log encodings. *International Journal on Artificial Intelligence Tools*, 26(1):1–29, 2017. URL https://doi.org/10.1142/S0218213017600053.

Niklas Sörensson et Armin Biere : Minimizing learned clauses. *In* Oliver Kullmann, éditeur : *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 de *Lecture Notes in Computer Science*, pages 237–243. Springer, 2009. URL https://doi.org/10.1007/978-3-642-02777-2_23.

Niklas Sorensson et Niklas Een : Minisat v1. 13-a sat solver with conflict-clause minimization. *SAT*, 2005(53):1–2, 2005.

Niklas Sörensson et Niklas Eén : Minisat 2.1 and minisat++ 1.0-sat race 2008 editions. *SAT*, page 31, 2009.

G. S. Tseitin : On the complexity of derivation in propositional calculus. 1968.

Julian R. Ullmann : An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976. URL http://doi.acm.org/10.1145/321921.321925.

Toby Walsh : SAT v CSP. *In* Rina Dechter, éditeur : *Principles and Practice of Constraint Programming - CP 2000, 6th International Conference, Singapore, September 18-21, 2000, Proceedings*, volume 1894 de *Lecture Notes in Computer Science*, pages 441–456. Springer, 2000. URL https://doi.org/10.1007/3-540-45349-0_32.

Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz et Sharad Malik : Efficient conflict driven learning in boolean satisfiability solver. *In* Rolf Ernst, éditeur : *Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2001, San Jose, CA, USA, November 4-8, 2001*, pages 279–285. IEEE Computer Society, 2001. URL https://doi.org/10.1109/ICCAD.2001.968634.

Lintao Zhang et Sharad Malik : The quest for efficient boolean satisfiability solvers. *In* Andrei Voronkov, éditeur : *Automated Deduction - CADE-18, 18th International Conference on Automated Deduction, Copenhagen, Denmark, July 27-30, 2002, Proceedings*, volume 2392 de *Lecture Notes in Computer Science*, pages 295–313. Springer, 2002. URL https://doi.org/10.1007/3-540-45620-1_26.

Alexander Aleksandrovich Zykov : On some properties of linear complexes. *Matematicheskii sbornik*, 66(2):163–188, 1949.

Résumé

Cette thèse s'inscrit dans le domaine de la programmation par contraintes (CP), un des paradigmes les plus efficaces pour résoudre de nombreux problèmes (de nature combinatoire) en IA. Nous nous sommes intéressés à l'amélioration des techniques de résolution CSP (problème de satisfaction de contraintes), notamment dans un contexte hybride en utilisant la puissance des moteurs d'inférence SAT (problème de satisfaction booléenne). Nous avons ainsi développé plusieurs techniques concernant l'analyse de conflits afin d'extraire des informations utiles pour la résolution, grâce à une analyse fine suivie d'une forme d'apprentissage. Pour atteindre cet objectif, nous avons dans un premier temps renforcé la portée (puissance) des instanciations partielles incohérentes (nogoods) enregistrées par le système de résolution en proposant plusieurs règles permettant de les combiner. De plus, nous avons intégré un moteur de résolution SAT au sein d'un système CP reposant sur la notion d'explications paresseuses. L'utilisation d'un moteur SAT est motivée par leur grande efficacité à produire et à manipuler des formes simples de nogoods sous forme clausale. Le logiciel NACRE, moteur générique de raisonnement, est le résultat de ce travail ; il a notamment été conçu pour être un solveur hybride, résolvant les problèmes de satisfaction de contraintes à l'aide de méthodes dédiées ou inspirées de SAT. Notre approche générique (c'est-à-dire, valide quelque soit le problème traité) s'est avérée très efficace en pratique (NACRE a gagné 2 médailles d'or aux compétitions XCSP 2018 et 2019). Dans l'objectif d'enrichir l'hybridation SAT/CP, nous avons conduit une étude sur la qualité des clauses produites par le moteur SAT. Celle-ci nous permis de proposer de nouvelles méthodes de réduction de base de clauses, de minimisation ainsi que de nouvelles heuristiques de recherche.

Mots-clés: programmation par contraintes, SAT, clauses, nogoods, hybridation

Abstract

This thesis belongs to the field of constraint programming (CP), one of the most efficient paradigms for solving many problems (of a combinatorial nature) in AI. We have been interested in improving the CSP (constraint satisfaction problem) solving techniques, especially in a hybrid context using the power of SAT (Boolean satisfaction problem) inference engines. We have developed several conflict analysis techniques to extract useful information for resolution, through fine analysis followed by a form of learning. To achieve this goal, we first strengthened the power of nogoods recorded by the resolution system by proposing several rules to combine them. In addition, we integrated a SAT resolution engine into a CP system based on the notion of lazy explanations. The use of a SAT engine is motivated by their high efficiency in producing and manipulating simple forms of nogoods in clausal form. The NACRE software, a generic reasoning engine, is the result of this work ; In particular, it was designed to be a hybrid solver, solving constraint satisfaction problems using dedicated or SAT-inspired methods. Our generic (i.e., valid for any problem) approach has proved very effective in practice (NACRE has won 2 gold medals at the XCSP competitions 2018 and 2019). In order to enrich the SAT / CP hybridization, we conducted a study on the quality of clauses produced by the SAT engine. This allowed us to propose new methods for clauses database reduction, minimization and new search heuristics.

Keywords: constraint programming, SAT, clauses, nogoods, hybridization